

C#

专业人士笔记

Chapter 19: DateTime Methods

Section 19.1: DateTime Formatting

Standard DateTime Formatting

`DateTimeFormatInfo` specifies a set of specifiers for simple date and time formatting. Every⁹ a particular `DateTimeFormatInfo` format pattern.

```
//Create datetime
DateTime dt = new DateTime(2016, 09, 19, 58, 23, 200);
```

```
var t = String.Format("{0:t}", dt); // "6:58 PM"
var d = String.Format("{0:d}", dt); // "9/19/2016"
var T = String.Format("{0:T}", dt); // "Monday, August 1, 2016 6:58:23 PM"
var S = String.Format("{0:S}", dt); // "Monday, August 1, 2016 6:58:23 PM"
var F = String.Format("{0:F3}", dt); // "8/1/2016 6:58:23 PM"
var g = String.Format("{0:g}", dt); // "8/1/2016 6:58:23 PM"
var G = String.Format("{0:G}", dt); // "8/1/2016 6:58:23 PM"
var Y = String.Format("{0:y}", dt); // "2016"
var Y = String.Format("{0:Y}", dt); // "2016-Aug-01 18:58:23"
var R = String.Format("{0:r}", dt); // "2016-08-01 18:58:23"
var R = String.Format("{0:R}", dt); // "2016-08-01 18:58:23"
var B = String.Format("{0:b}", dt); // "August 1"
var b = String.Format("{0:B}", dt); // "August 1"
```

`UniversalSortableDateTimeFormat`

There are following custom format specifiers:

- `{0:yyyy}`
- `{0:yy}`
- `{0:dd}`
- `{0:hour12}`
- `{0:hour24}`
- `{0:minute}`
- `{0:second}`
- `{0:fraction}`
- `{0:M or AM}`
- `{0:time zone}`

```
var year =
String.Format("{0:yyyy} {0:MM} {0:dd} {0:HH}:{0:MM}:{0:SS}", dt); // "2016 09 19 18:58:23"
var month =
String.Format("{0:MM} {0:dd} {0:HH}:{0:MM}:{0:SS}", dt); // "09 19 18:58:23"
var day =
String.Format("{0:dd} {0:HH}:{0:MM}:{0:SS}", dt); // "19 18:58:23"
var hour =
String.Format("{0:HH}:{0:MM}:{0:SS}", dt); // "18:58:23"
var minute =
String.Format("{0:MM}:{0:SS}", dt); // "58:23"
var second =
String.Format("{0:SS}", dt); // "23"
var fraction =
String.Format("{0:f3}", dt); // "23.000"
var period =
String.Format("{0:K}", dt); // "1"
var zone =
String.Format("{0:z:zzz}", dt); // "EST-04:00"
```

You can use also date separator / (slash) and time separator : (colon).

CH Notes for Professionals

Chapter 34: Anonymous types

Section 34.1: Anonymous vs dynamic

Anonymous types allow the creation of objects without having to explicitly define their types ahead of time, while maintaining static type checking.

```
var anon = new { value = 1 };
Console.WriteLine(anon.D); // compile time error
```

Conversely, `dynamic` has dynamic type checking, opting for runtime errors, instead of compile time errors.

```
dynamic val = "foo";
Console.WriteLine(val.D); // compiles, but throws runtime error
```

Section 34.2: Creating an anonymous type

Since anonymous types are not named, variables of those types must be implicitly typed (`var`).

```
var anon = new { Foo = 1, Bar = 2 };
```

If the member names are not specified, they are set to the name of the property/variable used to initialize the object.

```
int foo = 1;
int bar = 2;
var anon = new { Foo, Bar };
```

`anon.Foo` == 1

`anon.Bar` == 2

Note that names can only be omitted when the expression in the anonymous type declaration is a simple pre-cess; for method calls or more complex expressions, a property name must be specified.

```
string foo = "some string";
var anon = new { Foo.Length }; // OK
var anon = new { Foo.Length < 10 ? "short string" : "long string" };
// compiler error - Invalid anonymous type member declarator.
// anon.Foo = new { Description = Foo.Length < 10 ? "short string" : "long string" };
// OK
```

Section 34.3: Anonymous type equality

Anonymous type equality is given by the `Equals` instance method. Two objects are equal if they have the same and equal values (through a `Prop.Equals()` for every property).

```
var anon = new { Foo = 1, Bar = 2 };
var wond = new { Foo = 1, Bar = 2 };
var anon = new { Foo = 3, Bar = 3 };
var wond = new { Foo = 3, Bar = 3 };
var anon = new { Foo = 2, Bar = 3 };
// anon.Equals(wond) == true
// anon.Equals(anon) == false
```

CH Notes for Professionals

Chapter 98: Networking

Section 98.1: Basic TCP Communication Client

This code example creates a TCP client, sends "Hello World" over the socket connection, and then writes the server response to the console before closing the connector.

```
// Declare Variables
string host = "stackoverflow.com";
int port = 8000;
int timeout = 3000;

// Create TCP client and connect
using (var client = new TcpClient(host, port))
{
    netStream = client.GetStream();
    netStream.ReadTimeout = timeout;
}

var t = String.Format("{0:t}", dt); // "6:58 PM"
var d = String.Format("{0:d}", dt); // "9/19/2016"
var T = String.Format("{0:T}", dt); // "Monday, August 1, 2016 6:58:23 PM"
var S = String.Format("{0:S}", dt); // "Monday, August 1, 2016 6:58:23 PM"
var F = String.Format("{0:F3}", dt); // "8/1/2016 6:58:23 PM"
var g = String.Format("{0:g}", dt); // "8/1/2016 6:58:23 PM"
var G = String.Format("{0:G}", dt); // "8/1/2016 6:58:23 PM"
var Y = String.Format("{0:y}", dt); // "2016"
var Y = String.Format("{0:Y}", dt); // "2016-Aug-01 18:58:23"
var R = String.Format("{0:r}", dt); // "2016-08-01 18:58:23"
var R = String.Format("{0:R}", dt); // "2016-08-01 18:58:23"
var B = String.Format("{0:b}", dt); // "August 1"
var b = String.Format("{0:B}", dt); // "August 1"
```

CH Notes for Professionals

Downloading a file from the internet is a very common task required by almost every application you're likely to download a file from the internet is a very common task required by almost every application you're likely to accomplish this, you can use the `"System.Net.WebClient"` class.

The simplest use of this, using the "using" pattern, is shown below:

```
using (var webClient = new WebClient())
{
    webClient.DownloadFile("http://www.server.com/file.txt", "C:\file.txt");
}
```

What this example does is it uses "using" to make sure that your web client is cleaned up correctly when finished, and simply transfers the named resource from the URL, in the first parameter, to the named file on your local hard drive in the second parameter.

The first parameter is of type `"System.Uri"`, the second parameter is of type `"System.String"`.

You can also use this function is an async form, so that it goes off and performs the download in the background, which helps to keep your user interface responsive.

When you use the `Aync` methods, you can hook up event handlers that allow you to monitor the progress, so that

you can use also date separator / (slash) and time separator : (colon).

CH Notes for Professionals

Chapter 19: DateTime Methods

Section 19.1: DateTime Formatting

Standard DateTime Formatting

`DateTimeFormatInfo` specifies a set of specifiers for simple date and time formatting. Every⁹ a particular `DateTimeFormatInfo` format pattern.

```
//Create datetime
DateTime dt = new DateTime(2016, 09, 19, 58, 23, 200);
```

```
var t = String.Format("{0:t}", dt); // "6:58 PM"
var d = String.Format("{0:d}", dt); // "9/19/2016"
var T = String.Format("{0:T}", dt); // "Monday, August 1, 2016 6:58:23 PM"
var S = String.Format("{0:S}", dt); // "Monday, August 1, 2016 6:58:23 PM"
var F = String.Format("{0:F3}", dt); // "8/1/2016 6:58:23 PM"
var g = String.Format("{0:g}", dt); // "8/1/2016 6:58:23 PM"
var G = String.Format("{0:G}", dt); // "8/1/2016 6:58:23 PM"
var Y = String.Format("{0:y}", dt); // "2016"
var Y = String.Format("{0:Y}", dt); // "2016-Aug-01 18:58:23"
var R = String.Format("{0:r}", dt); // "2016-08-01 18:58:23"
var R = String.Format("{0:R}", dt); // "2016-08-01 18:58:23"
var B = String.Format("{0:b}", dt); // "August 1"
var b = String.Format("{0:B}", dt); // "August 1"
```

`UniversalSortableDateTimeFormat`

There are following custom format specifiers:

- `{0:yyyy}`
- `{0:yy}`
- `{0:dd}`
- `{0:hour12}`
- `{0:hour24}`
- `{0:minute}`
- `{0:second}`
- `{0:fraction}`
- `{0:M or AM}`
- `{0:time zone}`

```
var year =
String.Format("{0:yyyy} {0:MM} {0:dd} {0:HH}:{0:MM}:{0:SS}", dt); // "2016 09 19 18:58:23"
var month =
String.Format("{0:MM} {0:dd} {0:HH}:{0:MM}:{0:SS}", dt); // "09 19 18:58:23"
var day =
String.Format("{0:dd} {0:HH}:{0:MM}:{0:SS}", dt); // "19 18:58:23"
var hour =
String.Format("{0:HH}:{0:MM}:{0:SS}", dt); // "18:58:23"
var minute =
String.Format("{0:MM}:{0:SS}", dt); // "58:23"
var second =
String.Format("{0:SS}", dt); // "23"
var fraction =
String.Format("{0:f3}", dt); // "23.000"
var period =
String.Format("{0:K}", dt); // "1"
var zone =
String.Format("{0:z:zzz}", dt); // "EST-04:00"
```

CH Notes for Professionals

C#

Notes for Professionals

Chapter 34: Anonymous types

Section 34.1: Anonymous vs dynamic

Anonymous types allow the creation of objects without having to explicitly define their types ahead of time, while maintaining static type checking.

```
var anon = new { value = 1 };
Console.WriteLine(anon.D); // compile time error
```

Conversely, `dynamic` has dynamic type checking, opting for runtime errors, instead of compile time errors.

```
dynamic val = "foo";
Console.WriteLine(val.D); // compiles, but throws runtime error
```

Section 34.2: Creating an anonymous type

Since anonymous types are not named, variables of those types must be implicitly typed (`var`).

```
var anon = new { Foo = 1, Bar = 2 };
```

If the member names are not specified, they are set to the name of the property/variable used to initialize the object.

```
int foo = 1;
int bar = 2;
var anon = new { Foo, Bar };
```

`anon.Foo` == 1

`anon.Bar` == 2

Note that names can only be omitted when the expression in the anonymous type declaration is a simple pre-cess; for method calls or more complex expressions, a property name must be specified.

```
string foo = "some string";
var anon = new { Foo.Length }; // OK
var anon = new { Foo.Length < 10 ? "short string" : "long string" };
// compiler error - Invalid anonymous type member declarator.
// anon.Foo = new { Description = Foo.Length < 10 ? "short string" : "long string" };
// OK
```

Section 34.3: Anonymous type equality

Anonymous type equality is given by the `Equals` instance method. Two objects are equal if they have the same and equal values (through a `Prop.Equals()` for every property).

```
var anon = new { Foo = 1, Bar = 2 };
var wond = new { Foo = 1, Bar = 2 };
var anon = new { Foo = 3, Bar = 3 };
var wond = new { Foo = 3, Bar = 3 };
var anon = new { Foo = 2, Bar = 3 };
// anon.Equals(wond) == true
// anon.Equals(anon) == false
```

CH Notes for Professionals

Chapter 98: Networking

Section 98.1: Basic TCP Communication Client

This code example creates a TCP client, sends "Hello World" over the socket connection, and then writes the server response to the console before closing the connector.

```
// Declare Variables
string host = "stackoverflow.com";
int port = 8000;
int timeout = 3000;

// Create TCP client and connect
using (var client = new TcpClient(host, port))
{
    netStream = client.GetStream();
    netStream.ReadTimeout = timeout;
}

var t = String.Format("{0:t}", dt); // "6:58 PM"
var d = String.Format("{0:d}", dt); // "9/19/2016"
var T = String.Format("{0:T}", dt); // "Monday, August 1, 2016 6:58:23 PM"
var S = String.Format("{0:S}", dt); // "Monday, August 1, 2016 6:58:23 PM"
var F = String.Format("{0:F3}", dt); // "8/1/2016 6:58:23 PM"
var g = String.Format("{0:g}", dt); // "8/1/2016 6:58:23 PM"
var G = String.Format("{0:G}", dt); // "8/1/2016 6:58:23 PM"
var Y = String.Format("{0:y}", dt); // "2016"
var Y = String.Format("{0:Y}", dt); // "2016-Aug-01 18:58:23"
var R = String.Format("{0:r}", dt); // "2016-08-01 18:58:23"
var R = String.Format("{0:R}", dt); // "2016-08-01 18:58:23"
var B = String.Format("{0:b}", dt); // "August 1"
var b = String.Format("{0:B}", dt); // "August 1"
```

`UniversalSortableDateTimeFormat`

There are following custom format specifiers:

- `{0:yyyy}`
- `{0:yy}`
- `{0:dd}`
- `{0:hour12}`
- `{0:hour24}`
- `{0:minute}`
- `{0:second}`
- `{0:fraction}`
- `{0:M or AM}`
- `{0:time zone}`

```
var year =
String.Format("{0:yyyy} {0:MM} {0:dd} {0:HH}:{0:MM}:{0:SS}", dt); // "2016 09 19 18:58:23"
var month =
String.Format("{0:MM} {0:dd} {0:HH}:{0:MM}:{0:SS}", dt); // "09 19 18:58:23"
var day =
String.Format("{0:dd} {0:HH}:{0:MM}:{0:SS}", dt); // "19 18:58:23"
var hour =
String.Format("{0:HH}:{0:MM}:{0:SS}", dt); // "18:58:23"
var minute =
String.Format("{0:MM}:{0:SS}", dt); // "58:23"
var second =
String.Format("{0:SS}", dt); // "23"
var fraction =
String.Format("{0:f3}", dt); // "23.000"
var period =
String.Format("{0:K}", dt); // "1"
var zone =
String.Format("{0:z:zzz}", dt); // "EST-04:00"
```

CH Notes for Professionals

700多页
专业提示和技巧

700+ pages
of professional hints and tricks

目录

关于	1
第1章：C#语言入门	2
第1.1节：创建新的控制台应用程序（Visual Studio）	2
第1.2节：在Visual Studio中创建新项目（控制台应用程序）并以调试模式运行	4
第1.3节：使用.NET Core创建新程序	7
第1.4节：使用Mono创建新程序	9
第1.5节：使用LinqPad创建新查询	9
第1.6节：使用Xamarin Studio创建新项目	12
第2章：字面量	18
第2.1节：无符号整数字面量	18
第2.2节：整数字面量	18
第2.3节：有符号字节字面量	18
第2.4节：十进制字面量	18
第2.5节：双精度字面量	18
第2.6节：单精度字面量	18
第2.7节：长整数字面量	18
第2.8节：无符号长整数字面量	18
第2.9节：字符串字面量	19
第2.10节：字符字面量	19
第2.11节：字节字面量	19
第2.12节：短整数字面量	19
第2.13节：无符号短整数字面量	19
第2.14节：布尔字面量	19
第3章：运算符	20
第3.1节：可重载运算符	20
第3.2节：重载等号运算符	21
第3.3节：关系运算符	22
第3.4节：隐式转换和显式转换运算符	24
第3.5节：短路运算符	25
第3.6节：?:三元运算符	26
第3.7节：?:（空条件运算符）	27
第3.8节：“异或”运算符	27
第3.9节：默认操作符	28
第3.10节：赋值运算符“=”	28
第3.11节：sizeof	28
第3.12节：??空合并运算符	29
第3.13节：位运算符	29
第3.14节：=> Lambda运算符	29
第3.15节：类成员运算符：空条件成员访问	31
第3.16节：类成员运算符：空条件索引	31
第3.17节：后缀和前缀的递增与递减	31
第3.18节：typeof	32
第3.19节：带赋值的二元运算符	32
第3.20节：nameof运算符	32
第3.21节：类成员运算符：成员访问	33
第3.22节：类成员运算符：函数调用	33

Contents

About	1
Chapter 1: Getting started with C# Language	2
Section 1.1: Creating a new console application (Visual Studio)	2
Section 1.2: Creating a new project in Visual Studio (console application) and Running it in Debug mode	4
Section 1.3: Creating a new program using .NET Core	7
Section 1.4: Creating a new program using Mono	9
Section 1.5: Creating a new query using LinqPad	9
Section 1.6: Creating a new project using Xamarin Studio	12
Chapter 2: Literals	18
Section 2.1: uint literals	18
Section 2.2: int literals	18
Section 2.3: sbyte literals	18
Section 2.4: decimal literals	18
Section 2.5: double literals	18
Section 2.6: float literals	18
Section 2.7: long literals	18
Section 2.8: ulong literal	18
Section 2.9: string literals	19
Section 2.10: char literals	19
Section 2.11: byte literals	19
Section 2.12: short literal	19
Section 2.13: ushort literal	19
Section 2.14: bool literals	19
Chapter 3: Operators	20
Section 3.1: Overloadable Operators	20
Section 3.2: Overloading equality operators	21
Section 3.3: Relational Operators	22
Section 3.4: Implicit Cast and Explicit Cast Operators	24
Section 3.5: Short-circuiting Operators	25
Section 3.6: ?: Ternary Operator	26
Section 3.7: ?: (Null Conditional Operator)	27
Section 3.8: "Exclusive or" Operator	27
Section 3.9: default Operator	28
Section 3.10: Assignment operator '='	28
Section 3.11: sizeof	28
Section 3.12: ?? Null-Coalescing Operator	29
Section 3.13: Bit-Shifting Operators	29
Section 3.14: => Lambda operator	29
Section 3.15: Class Member Operators: Null Conditional Member Access	31
Section 3.16: Class Member Operators: Null Conditional Indexing	31
Section 3.17: Postfix and Prefix increment and decrement	31
Section 3.18: typeof	32
Section 3.19: Binary operators with assignment	32
Section 3.20: nameof Operator	32
Section 3.21: Class Member Operators: Member Access	33
Section 3.22: Class Member Operators: Function Invocation	33

第3.23节：类成员运算符：聚合对象索引	33	Section 3.23: Class Member Operators: Aggregate Object Indexing	33
第4章：条件语句	34	Chapter 4: Conditional Statements	34
第4.1节：If-Else语句	34	Section 4.1: If-Else Statement	34
第4.2节：If语句条件是标准布尔表达式和值	34	Section 4.2: If statement conditions are standard boolean expressions and values	34
第4.3节：If-Else If-Else语句	35	Section 4.3: If-Else If-Else Statement	35
第5章：等号运算符	36	Chapter 5: Equality Operator	36
第5.1节：C#中的相等类型和相等运算符	36	Section 5.1: Equality kinds in c# and equality operator	36
第6章：Equals和GetHashCode	37	Chapter 6: Equals and GetHashCode	37
第6.1节：编写良好的GetHashCode重写	37	Section 6.1: Writing a good GetHashCode override	37
第6.2节：默认的Equals行为	37	Section 6.2: Default Equals behavior	37
第6.3节：在自定义类型上重写Equals和GetHashCode	38	Section 6.3: Override Equals and GetHashCode on custom types	38
第6.4节：IEqualityComparer中的Equals和GetHashCode	39	Section 6.4: Equals and GetHashCode in IEqualityComparer	39
第7章：空合并运算符	41	Chapter 7: Null-Coalescing Operator	41
第7.1节：基本用法	41	Section 7.1: Basic usage	41
第7.2节：空穿透和链式调用	41	Section 7.2: Null fall-through and chaining	41
第7.3节：带方法调用的空合并运算符	42	Section 7.3: Null coalescing operator with method calls	42
第7.4节：使用现有的或创建新的	43	Section 7.4: Use existing or create new	43
第7.5节：使用空合并运算符的延迟属性初始化	43	Section 7.5: Lazy properties initialization with null coalescing operator	43
第8章：空条件运算符	44	Chapter 8: Null-conditional Operators	44
第8.1节：空条件运算符	44	Section 8.1: Null-Conditional Operator	44
第8.2节：空条件索引	44	Section 8.2: The Null-Conditional Index	44
第8.3节：避免NullReferenceException异常	45	Section 8.3: Avoiding NullReferenceExceptions	45
第8.4节：空条件运算符可用于扩展方法	45	Section 8.4: Null-conditional Operator can be used with Extension Method	45
第9章：nameof运算符	47	Chapter 9: nameof Operator	47
第9.1节：基本用法：打印变量名	47	Section 9.1: Basic usage: Printing a variable name	47
第9.2节：引发PropertyChanged事件	47	Section 9.2: Raising PropertyChanged event	47
第9.3节：参数检查和保护子句	48	Section 9.3: Argument Checking and Guard Clauses	48
第9.4节：强类型MVC操作链接	48	Section 9.4: Strongly typed MVC action links	48
第9.5节：处理PropertyChanged事件	49	Section 9.5: Handling PropertyChanged events	49
第9.6节：应用于泛型类型参数	49	Section 9.6: Applied to a generic type parameter	49
第9.7节：打印参数名称	50	Section 9.7: Printing a parameter name	50
第9.8节：应用于限定标识符	50	Section 9.8: Applied to qualified identifiers	50
第10章：逐字字符串	51	Chapter 10: Verbatim Strings	51
第10.1节：插值逐字字符串	51	Section 10.1: Interpolated Verbatim Strings	51
第10.2节：转义双引号	51	Section 10.2: Escaping Double Quotes	51
第10.3节：逐字字符串指示编译器不使用字符转义	51	Section 10.3: Verbatim strings instruct the compiler to not use character escapes	51
第10.4节：多行字符串	52	Section 10.4: Multiline Strings	52
第11章：常见字符串操作	53	Chapter 11: Common String Operations	53
第11.1节：字符串格式化	53	Section 11.1: Formatting a string	53
第11.2节：正确反转字符串	53	Section 11.2: Correctly reversing a string	53
第11.3节：将字符串填充到固定长度	54	Section 11.3: Padding a string to a fixed length	54
第11.4节：从字符串右侧获取x个字符	55	Section 11.4: Getting x characters from the right side of a string	55
第11.5节：使用String.IsNullOrEmpty()和String.IsNullOrWhiteSpace()检查空字符串	56	Section 11.5: Checking for empty String using String.IsNullOrEmpty() and String.IsNullOrWhiteSpace()	56
第11.6节：修剪字符串开头和/或结尾的不需要字符	57	Section 11.6: Trimming Unwanted Characters Off the Start and/or End of Strings	57
第11.7节：将十进制数转换为二进制、八进制和十六进制格式	57	Section 11.7: Convert Decimal Number to Binary,Octal and Hexadecimal Format	57
第11.8节：从数组构造字符串	57	Section 11.8: Construct a string from Array	57
第11.9节：使用ToString进行格式化	58	Section 11.9: Formatting using ToString	58
第11.10节：通过另一个字符串拆分字符串	59	Section 11.10: Splitting a String by another string	59

第11.11节：通过特定字符拆分字符串	59
第11.12节：获取给定字符串的子字符串	59
第11.13节：判断字符串是否以给定序列开头	59
第11.14节：获取特定索引处的字符并枚举字符串	59
第11.15节：将字符串数组连接成一个新的字符串	60
第11.16节：替换字符串中的子字符串	60
第11.17节：改变字符串中字符的大小写	60
第11.18节：将字符串数组合并成一个字符串	61
第11.19节：字符串连接	61
第12章：String.Format	62
第12.1节：自C# 6.0起	62
第12.2节：String.Format在框架中“嵌入”的位置	62
第12.3节：创建自定义格式提供程序	62
第12.4节：日期格式化	63
第12.5节：货币格式化	64
第12.6节：使用自定义数字格式	65
第12.7节：左对齐/右对齐，用空格填充	65
第12.8节：数字格式	66
第12.9节：ToString()	66
第12.10节：在String.Format()表达式中转义大括号	67
第12.11节：与ToString()的关系	67
第13章：字符串连接	68
第13.1节：+ 运算符	68
第13.2节：使用System.Text.StringBuilder连接字符串	68
第13.3节：使用String.Join连接字符串数组元素	68
第13.4节：使用\$连接两个字符串	69
第14章：字符串操作	70
第14.1节：在字符串中替换字符串	70
第14.2节：在字符串中查找字符串	70
第14.3节：去除（修剪）字符串中的空白字符	70
第14.4节：使用分隔符拆分字符串	71
第14.5节：将字符串数组连接成一个字符串	71
第14.6节：字符串连接	71
第14.7节：更改字符串中字符的大小写	71
第15章：字符串插值	73
第15.1节：字符串中的日期格式化	73
第15.2节：输出填充	73
第15.3节：表达式	74
第15.4节：字符串中的数字格式化	74
第15.5节：简单用法	75
第16章：字符串转义序列	76
第16.1节：字符串字面量中特殊符号的转义	76
第16.2节：Unicode字符转义序列	76
第16.3节：字符字面量中特殊符号的转义	76
第16.4节：标识符中使用转义序列	76
第16.5节：未识别的转义序列会产生编译时错误	77
第17章：StringBuilder	78
第17.1节：什么是StringBuilder以及何时使用	78
第17.2节：使用StringBuilder从大量记录创建字符串	79

Section 11.11: Splitting a String by specific character	59
Section 11.12: Getting Substrings of a given string	59
Section 11.13: Determine whether a string begins with a given sequence	59
Section 11.14: Getting a char at specific index and enumerating the string	59
Section 11.15: Joining an array of strings into a new one	60
Section 11.16: Replacing a string within a string	60
Section 11.17: Changing the case of characters within a String	60
Section 11.18: Concatenate an array of strings into a single string	61
Section 11.19: String Concatenation	61
Chapter 12: String.Format	62
Section 12.1: Since C# 6.0	62
Section 12.2: Places where String.Format is 'embedded' in the framework	62
Section 12.3: Create a custom format provider	62
Section 12.4: Date Formatting	63
Section 12.5: Currency Formatting	64
Section 12.6: Using custom number format	65
Section 12.7: Align left/ right, pad with spaces	65
Section 12.8: Numeric formats	66
Section 12.9: ToString()	66
Section 12.10: Escaping curly brackets inside a String.Format() expression	67
Section 12.11: Relationship with ToString()	67
Chapter 13: String Concatenate	68
Section 13.1: + Operator	68
Section 13.2: Concatenate strings using System.Text.StringBuilder	68
Section 13.3: Concat string array elements using String.Join	68
Section 13.4: Concatenation of two strings using \$	69
Chapter 14: String Manipulation	70
Section 14.1: Replacing a string within a string	70
Section 14.2: Finding a string within a string	70
Section 14.3: Removing (Trimming) white-space from a string	70
Section 14.4: Splitting a string using a delimiter	71
Section 14.5: Concatenate an array of strings into a single string	71
Section 14.6: String Concatenation	71
Section 14.7: Changing the case of characters within a String	71
Chapter 15: String Interpolation	73
Section 15.1: Format dates in strings	73
Section 15.2: Padding the output	73
Section 15.3: Expressions	74
Section 15.4: Formatting numbers in strings	74
Section 15.5: Simple Usage	75
Chapter 16: String Escape Sequences	76
Section 16.1: Escaping special symbols in string literals	76
Section 16.2: Unicode character escape sequences	76
Section 16.3: Escaping special symbols in character literals	76
Section 16.4: Using escape sequences in identifiers	76
Section 16.5: Unrecognized escape sequences produce compile-time errors	77
Chapter 17: StringBuilder	78
Section 17.1: What a StringBuilder is and when to use one	78
Section 17.2: Use StringBuilder to create string from a large number of records	79

第18章：正则表达式解析	80	Chapter 18: Regex Parsing	80
第18.1节：单次匹配	80	Section 18.1: Single match	80
第18.2节：多次匹配	80	Section 18.2: Multiple matches	80
第19章：日期时间方法	81	Chapter 19: DateTime Methods	81
第19.1节：日期时间格式化	81	Section 19.1: DateTime Formatting	81
第19.2节：DateTime.AddDays(Double)	82	Section 19.2: DateTime.AddDays(Double)	82
第19.3节：DateTime.AddHours(Double)	82	Section 19.3: DateTime.AddHours(Double)	82
第19.4节：DateTime.Parse(String)	82	Section 19.4: DateTime.Parse(String)	82
第19.5节：DateTime.TryParse(String, DateTime)	82	Section 19.5: DateTime.TryParse(String, DateTime)	82
第19.6节：DateTime.AddMilliseconds(Double)	83	Section 19.6: DateTime.AddMilliseconds(Double)	83
第19.7节：DateTime.Compare(DateTime t1, DateTime t2)	83	Section 19.7: DateTime.Compare(DateTime t1, DateTime t2)	83
第19.8节：DateTime.DaysInMonth(Int32, Int32)	83	Section 19.8: DateTime.DaysInMonth(Int32, Int32)	83
第19.9节：DateTime.AddYears(Int32)	84	Section 19.9: DateTime.AddYears(Int32)	84
第19.10节：处理DateTime时的纯函数警告	84	Section 19.10: Pure functions warning when dealing with DateTime	84
第19.11节：DateTime.TryParseExact(String, String, IFormatProvider, DateTimeStyles, DateTime)	84	Section 19.11: DateTime.TryParseExact(String, String, IFormatProvider, DateTimeStyles, DateTime)	84
第19.12节：DateTime.Add(TimeSpan)	86	Section 19.12: DateTime.Add(TimeSpan)	86
第19.13节：带有区域性信息的Parse和TryParse	86	Section 19.13: Parse and TryParse with culture info	86
第19.14节：for循环中的DateTime初始化器	87	Section 19.14: DateTime as initializer in for-loop	87
第19.15节：DateTime.ParseExact (字符串, 字符串, IFormatProvider)	87	Section 19.15: DateTime.ParseExact(String, String, IFormatProvider)	87
第19.16节：DateTime的ToString、ToShortDateString、ToLongDateString及格式化ToString	88	Section 19.16: DateTime.ToString, ToShortDateString, ToLongDateString and ToString formatted	88
第19.17节：当前日期	88	Section 19.17: Current Date	88
第20章：数组	89	Chapter 20: Arrays	89
第20.1节：声明数组	89	Section 20.1: Declaring an array	89
第20.2节：初始化填充重复非默认值的数组	89	Section 20.2: Initializing an array filled with a repeated non-default value	89
第20.3节：复制数组	90	Section 20.3: Copying arrays	90
第20.4节：比较数组是否相等	90	Section 20.4: Comparing arrays for equality	90
第20.5节：多维数组	91	Section 20.5: Multi-dimensional arrays	91
第20.6节：获取和设置数组值	91	Section 20.6: Getting and setting array values	91
第20.7节：遍历数组	91	Section 20.7: Iterate over an array	91
第20.8节：创建一个顺序数字数组	92	Section 20.8: Creating an array of sequential numbers	92
第20.9节：锯齿状数组	92	Section 20.9: Jagged arrays	92
第20.10节：数组协变性	94	Section 20.10: Array covariance	94
第20.11节：作为IEnumerable<>实例的数组	94	Section 20.11: Arrays as IEnumerable<> instances	94
第20.12节：检查一个数组是否包含另一个数组	94	Section 20.12: Checking if one array contains another array	94
第21章：数组循环旋转的O(n)算法	96	Chapter 21: O(n) Algorithm for circular rotation of an array	96
第21.1节：按给定偏移量旋转数组的泛型方法示例	96	Section 21.1: Example of a generic method that rotates an array by a given shift	96
第22章：枚举	98	Chapter 22: Enum	98
第22.1节：枚举基础	98	Section 22.1: Enum basics	98
第22.2节：作为标志的枚举	99	Section 22.2: Enum as flags	99
第22.3节：使用<<符号表示标志	101	Section 22.3: Using << notation for flags	101
第22.4节：使用按位逻辑测试标志风格的枚举值	101	Section 22.4: Test flags-style enum values with bitwise logic	101
第22.5节：向标志枚举添加和移除值	102	Section 22.5: Add and remove values from flagged enum	102
第22.6节：枚举与字符串的相互转换	102	Section 22.6: Enum to string and back	102
第22.7节：枚举可能具有意外的值	103	Section 22.7: Enums can have unexpected values	103
第22.8节：枚举的默认值为零	103	Section 22.8: Default value for enum == ZERO	103
第22.9节：为枚举值添加额外的描述信息	104	Section 22.9: Adding additional description information to an enum value	104
第22.10节：获取枚举的所有成员值	105	Section 22.10: Get all the members values of an enum	105
第22.11节：使用枚举进行按位操作	105	Section 22.11: Bitwise Manipulation using enums	105
第23章：元组	106	Chapter 23: Tuples	106

第23.1节：访问元组元素	106
第23.2节：创建元组	106
第23.3节：比较和排序元组	106
第23.4节：从方法返回多个值	107
第24章：GUID	108
第24.1节：获取Guid的字符串表示	108
第24.2节：创建Guid	108
第24.3节：声明可空的GUID	108
第25章：大整数（BigInteger）	110
第25.1节：计算第一个1000位的斐波那契数	110
第26章：集合初始化器	111
第26.1节：集合初始化器	111
第26.2节：C# 6 索引初始化器	111
第26.3节：自定义类中的集合初始化器	112
第26.4节：在对象初始化器中使用集合初始化器	113
第26.5节：带参数数组的集合初始化器	114
第27章：C# 集合概述	115
第27.1节：HashSet<T>	115
第27.2节：Dictionary< TKey, TValue >	115
第27.3节：SortedSet<T>	116
第27.4节：T[] (T数组)	116
第27.5节：List<T>	117
第27.6节：Stack<T>	117
第27.7节：链表<T>	117
第27.8节：队列	118
第28章：循环	119
第28.1节：for循环	119
第28.2节：do-while循环	120
第28.3节：foreach循环	120
第28.4节：循环风格	121
第28.5节：嵌套循环	122
第28.6节：continue语句	122
第28.7节：while循环	123
第28.8节：break语句	123
第29章：迭代器	125
第29.1节：使用yield创建迭代器	125
第29.2节：简单数字迭代器示例	126
第30章：IEnumerable	127
第30.1节：带自定义枚举器的IEnumerable	127
第30.2节：IEnumerable<int>	128
第31章：值类型与引用类型	129
第31.1节：使用ref关键字按引用传递	129
第31.2节：在其他地方更改值	130
第31.3节：ref 与 out 参数	131
第31.4节：赋值	132
第31.5节：方法参数 ref 和 out 的区别	132
第31.6节：按引用传递	133
第32章：内置类型	134
第32.1节：装箱值类型的转换	134

Section 23.1: Accessing tuple elements	106
Section 23.2: Creating tuples	106
Section 23.3: Comparing and sorting Tuples	106
Section 23.4: Return multiple values from a method	107
Chapter 24: Guid	108
Section 24.1: Getting the string representation of a Guid	108
Section 24.2: Creating a Guid	108
Section 24.3: Declaring a nullable GUID	108
Chapter 25: BigInteger	110
Section 25.1: Calculate the First 1,000-Digit Fibonacci Number	110
Chapter 26: Collection Initializers	111
Section 26.1: Collection initializers	111
Section 26.2: C# 6 Index Initializers	111
Section 26.3: Collection initializers in custom classes	112
Section 26.4: Using collection initializer inside object initializer	113
Section 26.5: Collection Initializers with Parameter Arrays	114
Chapter 27: An overview of C# collections	115
Section 27.1: HashSet<T>	115
Section 27.2: Dictionary< TKey, TValue >	115
Section 27.3: SortedSet<T>	116
Section 27.4: T[] (Array of T)	116
Section 27.5: List<T>	117
Section 27.6: Stack<T>	117
Section 27.7: LinkedList<T>	117
Section 27.8: Queue	118
Chapter 28: Looping	119
Section 28.1: For Loop	119
Section 28.2: Do - While Loop	120
Section 28.3: Foreach Loop	120
Section 28.4: Looping styles	121
Section 28.5: Nested loops	122
Section 28.6: continue	122
Section 28.7: While loop	123
Section 28.8: break	123
Chapter 29: Iterators	125
Section 29.1: Creating Iterators Using Yield	125
Section 29.2: Simple Numeric Iterator Example	126
Chapter 30: IEnumerable	127
Section 30.1: IEnumerable with custom Enumerator	127
Section 30.2: IEnumerable<int>	128
Chapter 31: Value type vs Reference type	129
Section 31.1: Passing by reference using ref keyword	129
Section 31.2: Changing values elsewhere	130
Section 31.3: ref vs out parameters	131
Section 31.4: Assignment	132
Section 31.5: Difference with method parameters ref and out	132
Section 31.6: Passing by reference	133
Chapter 32: Built-in Types	134
Section 32.1: Conversion of boxed value types	134

第32.2节：与装箱值类型的比较	134
第32.3节：不可变引用类型 - 字符串	134
第32.4节：值类型 - 字符	135
第32.5节：值类型 - short、int、long (有符号16位、32位、64位整数)	135
第32.6节：值类型 - ushort、uint、ulong (无符号16位、32位、64位整数)	135
第32.7节：值类型 - 布尔型	136
第33章：内置类型的别名	137
第33.1节：内置类型表	137
第34章：匿名类型	138
第34.1节：匿名类型与动态类型	138
第34.2节：创建匿名类型	138
第34.3节：匿名类型的相等性	138
第34.4节：带匿名类型的泛型方法	139
第34.5节：使用匿名类型实例化泛型类型	139
第34.6节：隐式类型数组	139
第35章：动态类型	141
第35.1节：创建带属性的动态对象	141
第35.2节：创建动态变量	141
第35.3节：返回动态类型	141
第35.4节：处理编译时未知的特定类型	141
第36章：类型转换	143
第36.1节：显式类型转换	143
第36.2节：MSDN隐式运算符示例	143
第37章：类型转换	145
第37.1节：无类型转换的兼容性检查	145
第37.2节：将对象转换为基类类型	145
第37.3节：转换运算符	145
第37.4节：LINQ类型转换操作	147
第37.5节：显式类型转换	147
第37.6节：安全的显式转换 (`as` 操作符)	147
第37.7节：隐式转换	148
第37.8节：显式数值转换	148
第38章：可空类型	149
第38.1节：初始化可空类型	149
第38.2节：检查可空类型是否有值	149
第38.3节：获取可空类型的值	149
第38.4节：从可空类型获取默认值	150
第38.5节：可空类型的默认值为null	150
第38.6节：底层Nullable<T>参数的有效使用	151
第38.7节：检查泛型类型参数是否为可空类型	152
第39章：构造函数和终结器	153
第39.1节：静态构造函数	153
第39.2节：单例构造函数模式	154
第39.3节：默认构造函数	154
第39.4节：强制调用静态构造函数	155
第39.5节：从一个构造函数调用另一个构造函数	156
第39.6节：调用基类构造函数	156
第39.7节：派生类的终结器	157
第39.8节：静态构造函数中的异常	157

Section 32.2: Comparisons with boxed value types	134
Section 32.3: Immutable reference type - string	134
Section 32.4: Value type - char	135
Section 32.5: Value type - short, int, long (signed 16 bit, 32 bit, 64 bit integers)	135
Section 32.6: Value type - ushort, uint, ulong (unsigned 16 bit, 32 bit, 64 bit integers)	135
Section 32.7: Value type - bool	136
Chapter 33: Aliases of built-in types	137
Section 33.1: Built-In Types Table	137
Chapter 34: Anonymous types	138
Section 34.1: Anonymous vs dynamic	138
Section 34.2: Creating an anonymous type	138
Section 34.3: Anonymous type equality	138
Section 34.4: Generic methods with anonymous types	139
Section 34.5: Instantiating generic types with anonymous types	139
Section 34.6: Implicitly typed arrays	139
Chapter 35: Dynamic type	141
Section 35.1: Creating a dynamic object with properties	141
Section 35.2: Creating a dynamic variable	141
Section 35.3: Returning dynamic	141
Section 35.4: Handling Specific Types Unknown at Compile Time	141
Chapter 36: Type Conversion	143
Section 36.1: Explicit Type Conversion	143
Section 36.2: MSDN implicit operator example	143
Chapter 37: Casting	145
Section 37.1: Checking compatibility without casting	145
Section 37.2: Cast an object to a base type	145
Section 37.3: Conversion Operators	145
Section 37.4: LINQ Casting operations	147
Section 37.5: Explicit Casting	147
Section 37.6: Safe Explicit Casting (`as` operator)	147
Section 37.7: Implicit Casting	148
Section 37.8: Explicit Numeric Conversions	148
Chapter 38: Nullable types	149
Section 38.1: Initialising a nullable	149
Section 38.2: Check if a Nullable has a value	149
Section 38.3: Get the value of a nullable type	149
Section 38.4: Getting a default value from a nullable	150
Section 38.5: Default value of nullable types is null	150
Section 38.6: Effective usage of underlying Nullable<T> argument	151
Section 38.7: Check if a generic type parameter is a nullable type	152
Chapter 39: Constructors and Finalizers	153
Section 39.1: Static constructor	153
Section 39.2: Singleton constructor pattern	154
Section 39.3: Default Constructor	154
Section 39.4: Forcing a static constructor to be called	155
Section 39.5: Calling a constructor from another constructor	156
Section 39.6: Calling the base class constructor	156
Section 39.7: Finalizers on derived classes	157
Section 39.8: Exceptions in static constructors	157

第39.9节：构造函数和属性初始化	158
第39.10节：通用静态构造函数	160
第39.11节：在构造函数中调用虚方法	160
第40章：访问修饰符	162
第40.1节：访问修饰符图示	162
第40.2节： <code>public</code> （公共）	163
第40.3节： <code>private</code> （私有）	163
第40.4节：受保护的内部	164
第40.5节：内部	165
第40.6节：受保护的	166
第41章：接口	167
第41.1节：实现接口	167
第41.2节：显式接口实现	167
第41.3节：接口基础	169
第41.4节：以 <code>IComparable<T></code> 为例实现接口	171
第41.5节：实现多个接口	172
第41.6节：我们为何使用接口	172
第41.7节：通过显式实现“隐藏”成员	173
第42章：静态类	175
第42.1节：静态类	175
第42.2节：静态类的生命周期	175
第42.3节： <code>static</code> 关键字	176
第43章：单例模式实现	177
第43.1节：静态初始化单例	177
第43.2节：延迟加载的线程安全单例（使用 <code>Lazy<T></code> ）	177
第43.3节：延迟加载的线程安全单例（使用双重检查锁定）	177
第43.4节：延迟加载的线程安全单例（适用于.NET 3.5或更早版本，替代实现）	178
第44章：依赖注入	180
第44.1节：使用Unity的C#和ASP.NET依赖注入	180
第44.2节：使用MEF的依赖注入	182
第45章：部分类和方法	185
第45.1节：部分类	185
第45.2节：继承自基类的部分类	185
第45.3节：部分方法	186
第46章：对象初始化器	187
第46.1节：简单用法	187
第46.2节：带非默认构造函数的用法	187
第46.3节：与匿名类型的用法	187
第47章：方法	189
第47.1节：调用方法	189
第47.2节：匿名方法	189
第47.3节：声明方法	190
第47.4节：参数和实参	190
第47.5节：返回类型	190
第47.6节：默认参数	191
第47.7节：方法重载	192
第47.8节：访问权限	193
第48章：扩展方法	194
第48.1节：扩展方法 - 概述	194

Section 39.9: Constructor and Property Initialization	158
Section 39.10: Generic Static Constructors	160
Section 39.11: Calling virtual methods in constructor	160
Chapter 40: Access Modifiers	162
Section 40.1: Access Modifiers Diagrams	162
Section 40.2: <code>public</code>	163
Section 40.3: <code>private</code>	163
Section 40.4: <code>protected internal</code>	164
Section 40.5: <code>internal</code>	165
Section 40.6: <code>protected</code>	166
Chapter 41: Interfaces	167
Section 41.1: Implementing an interface	167
Section 41.2: Explicit interface implementation	167
Section 41.3: Interface Basics	169
Section 41.4: <code>IComparable<T></code> as an Example of Implementing an Interface	171
Section 41.5: Implementing multiple interfaces	172
Section 41.6: Why we use interfaces	172
Section 41.7: "Hiding" members with Explicit Implementation	173
Chapter 42: Static Classes	175
Section 42.1: Static Classes	175
Section 42.2: Static class lifetime	175
Section 42.3: Static keyword	176
Chapter 43: Singleton Implementation	177
Section 43.1: Statically Initialized Singleton	177
Section 43.2: Lazy, thread-safe Singleton (using <code>Lazy<T></code>)	177
Section 43.3: Lazy, thread-safe Singleton (using Double Checked Locking)	177
Section 43.4: Lazy, thread safe singleton (for .NET 3.5 or older, alternate implementation)	178
Chapter 44: Dependency Injection	180
Section 44.1: Dependency Injection C# and ASP.NET with Unity	180
Section 44.2: Dependency injection using MEF	182
Chapter 45: Partial class and methods	185
Section 45.1: Partial classes	185
Section 45.2: Partial classes inheriting from a base class	185
Section 45.3: Partial methods	186
Chapter 46: Object initializers	187
Section 46.1: Simple usage	187
Section 46.2: Usage with non-default constructors	187
Section 46.3: Usage with anonymous types	187
Chapter 47: Methods	189
Section 47.1: Calling a Method	189
Section 47.2: Anonymous method	189
Section 47.3: Declaring a Method	190
Section 47.4: Parameters and Arguments	190
Section 47.5: Return Types	190
Section 47.6: Default Parameters	191
Section 47.7: Method overloading	192
Section 47.8: Access rights	193
Chapter 48: Extension Methods	194
Section 48.1: Extension methods - overview	194

第48章：扩展方法	196	Section 48.2: Null checking	196
第48.1节：空值检查	196	Section 48.3: Explicitly using an extension method	197
第48.2节：显式使用扩展方法	197	Section 48.4: Generic Extension Methods	197
第48.3节：通用扩展方法	197	Section 48.5: Extension methods can only see public (or internal) members of the extended class	199
第48.4节：扩展方法只能访问被扩展类的公共（或内部）成员	199	Section 48.6: Extension methods for chaining	199
第48.5节：用于链式调用的扩展方法	199	Section 48.7: Extension methods with Enumeration	200
第48.6节：带枚举的扩展方法	200	Section 48.8: Extension methods dispatch based on static type	201
第48.7节：基于静态类型的扩展方法分派	201	Section 48.9: Extension methods on Interfaces	202
第48.8节：接口上的扩展方法	202	Section 48.10: Extension methods in combination with interfaces	203
第48.9节：扩展方法与接口的结合	203	Section 48.11: Extension methods aren't supported by dynamic code	203
第48.10节：动态代码不支持扩展方法	203	Section 48.12: Extensions and interfaces together enable DRY code and mixin-like functionality	204
第48.11节：扩展方法与接口共同实现DRY代码和类似混入的功能	204	Section 48.13: IList<T> Extension Method Example: Comparing 2 Lists	205
第48.12节：IList<T>扩展方法示例：比较两个列表	205	Section 48.14: Extension methods as strongly typed wrappers	206
第48.13节：作为强类型包装器的扩展方法	206	Section 48.15: Using Extension methods to create beautiful mapper classes	206
第48.14节：使用扩展方法创建优美的映射器类	206	Section 48.16: Using Extension methods to build new collection types (e.g. DictList)	207
第48.15节：使用扩展方法构建新集合类型（例如 DictList）	207	Section 48.17: Extension methods for handling special cases	208
第48.16节：处理特殊情况的扩展方法	208	Section 48.18: Using Extension methods with Static methods and Callbacks	209
第48.17节：将扩展方法与静态方法和回调一起使用	209		
第49章：命名参数	211	Chapter 49: Named Arguments	211
第49.1节：参数顺序不是必须的	211	Section 49.1: Argument order is not necessary	211
第49.2节：命名参数和可选参数	211	Section 49.2: Named arguments and optional parameters	211
第49.3节：命名参数可以使你的代码更清晰	211	Section 49.3: Named Arguments can make your code more clear	211
第50章：命名参数和可选参数	213	Chapter 50: Named and Optional Arguments	213
第50.1节：可选参数	213	Section 50.1: Optional Arguments	213
第50.2节：命名参数	214	Section 50.2: Named Arguments	214
第51章：数据注解	217	Chapter 51: Data Annotation	217
第51.1节：数据注释基础	217	Section 51.1: Data Annotation Basics	217
第51.2节：创建自定义验证属性	217	Section 51.2: Creating a custom validation attribute	217
第51.3节：手动执行验证属性	218	Section 51.3: Manually Execute Validation Attributes	218
第51.4节：验证属性	218	Section 51.4: Validation Attributes	218
第51.5节：EditableAttribute (数据建模属性)	220	Section 51.5: EditableAttribute (data modeling attribute)	220
第52章：关键字	222	Chapter 52: Keywords	222
第52.1节：as	222	Section 52.1: as	222
第52.2节：goto	223	Section 52.2: goto	223
第52.3节：volatile	224	Section 52.3: volatile	224
第52.4节：checked, unchecked	225	Section 52.4: checked, unchecked	225
第52.5节：virtual, override, new	226	Section 52.5: virtual, override, new	226
第52.6节：stackalloc	229	Section 52.6: stackalloc	229
第52.7节：break	230	Section 52.7: break	230
第52.8节：const	232	Section 52.8: const	232
第52.9节：async, await	233	Section 52.9: async, await	233
第52.10节：for	234	Section 52.10: for	234
第52.11节：abstract	235	Section 52.11: abstract	235
第52.12节：fixed	236	Section 52.12: fixed	236
第52.13节：默认	237	Section 52.13: default	237
第52.14节：密封	238	Section 52.14: sealed	238
第52.15节：是	238	Section 52.15: is	238
第52.16节：这个	239	Section 52.16: this	239
第52.17节：只读	240	Section 52.17: readonly	240
第52.18节：typeof	241	Section 52.18: typeof	241
第52.19节：foreach	241	Section 52.19: foreach	241

第52.20节 : dynamic	242
第52.21节 : try, catch, finally, throw	243
第52.22节 : void	244
第52.23节 : 命名空间	244
第52.24节 : 引用, 输出	245
第52.25节 : 基类	246
第52.26节 : 浮点数, 双精度, 十进制	248
第52.27节 : 运算符	249
第52.28节 : 字符	250
第52.29节 : 参数	250
第52.30节 : while循环	251
第52.31节 : 空值	253
第52.32节 : 继续	254
第52.33节 : 字符串	254
第52.34节 : 返回	255
第52.35节 : 不安全	255
第52.36节 : switch (开关)	257
第52.37节 : var (变量)	258
第52.38节 : when (当)	259
第52.39节 : lock (锁)	260
第52.40节 : uint (无符号整数)	261
第52.41节 : if, if...else, if... else if	261
第52.42节 : static	262
第52.43节 : internal	264
第52.44节 : using	265
第52.45节 : where	265
第52.46节 : int	267
第52.47节 : 无符号长整型 (ulong)	268
第52.48节 : true, false	268
第52.49节 : 结构体	268
第52.50节 : 外部	269
第52.51节 : 布尔型	270
第52.52节 : 接口	270
第52.53节 : 委托	271
第52.54节 : 未检查	271
第52.55节 : 无符号短整型	272
第52.56节 : sizeof	272
第52.57节 : in	272
第52.58节 : 隐式	273
第52.59节 : do	273
第52.60节 : 长整型	274
第52.61节 : 枚举	274
第52.62节 : 部分	275
第52.63节 : 事件	276
第52.64节 : 有符号字节	277
第53章 : C#中的面向对象编程	278
第53.1节 : 类 :	278
第54章 : 递归	279
第54.1节 : 通俗易懂的递归	279
第54.2节 : 斐波那契数列	279

Section 52.20: dynamic	242
Section 52.21: try, catch, finally, throw	243
Section 52.22: void	244
Section 52.23: namespace	244
Section 52.24: ref, out	245
Section 52.25: base	246
Section 52.26: float, double, decimal	248
Section 52.27: operator	249
Section 52.28: char	250
Section 52.29: params	250
Section 52.30: while	251
Section 52.31: null	253
Section 52.32: continue	254
Section 52.33: string	254
Section 52.34: return	255
Section 52.35: unsafe	255
Section 52.36: switch	257
Section 52.37: var	258
Section 52.38: when	259
Section 52.39: lock	260
Section 52.40: uint	261
Section 52.41: if, if...else, if... else if	261
Section 52.42: static	262
Section 52.43: internal	264
Section 52.44: using	265
Section 52.45: where	265
Section 52.46: int	267
Section 52.47: ulong	268
Section 52.48: true, false	268
Section 52.49: struct	268
Section 52.50: extern	269
Section 52.51: bool	270
Section 52.52: interface	270
Section 52.53: delegate	271
Section 52.54: unchecked	271
Section 52.55: ushort	272
Section 52.56: sizeof	272
Section 52.57: in	272
Section 52.58: implicit	273
Section 52.59: do	273
Section 52.60: long	274
Section 52.61: enum	274
Section 52.62: partial	275
Section 52.63: event	276
Section 52.64: sbyte	277
Chapter 53: Object Oriented Programming In C#	278
Section 53.1: Classes:	278
Chapter 54: Recursion	279
Section 54.1: Recursion in plain English	279
Section 54.2: Fibonacci Sequence	279

第54章：幂运算计算	280	Section 54.3: PowerOf calculation	280
第54.4节：递归描述对象结构	280	Section 54.4: Recursively describe an object structure	280
第54.5节：使用递归获取目录树	281	Section 54.5: Using Recursion to Get Directory Tree	281
第54.6节：阶乘计算	284	Section 54.6: Factorial calculation	284
第55章：命名约定	285	Chapter 55: Naming Conventions	285
第55.1节：大小写约定	285	Section 55.1: Capitalization conventions	285
第55.2节：枚举	286	Section 55.2: Enums	286
第55.3节：接口	286	Section 55.3: Interfaces	286
第55.4节：异常	286	Section 55.4: Exceptions	286
第55.5节：私有字段	287	Section 55.5: Private fields	287
第55.6节：命名空间	287	Section 55.6: Namespaces	287
第56章：XML文档注释	288	Chapter 56: XML Documentation Comments	288
第56.1节：简单的方法注释	288	Section 56.1: Simple method annotation	288
第56.2节：从文档注释生成XML	288	Section 56.2: Generating XML from documentation comments	288
第56.3节：带有param和returns元素的方法文档注释	290	Section 56.3: Method documentation comment with param and returns elements	290
第56.4节：接口和类的文档注释	290	Section 56.4: Interface and class documentation comments	290
第56.5节：在文档中引用另一个类	291	Section 56.5: Referencing another class in documentation	291
第57章：注释和区域	292	Chapter 57: Comments and regions	292
第57.1节：注释	292	Section 57.1: Comments	292
第57.2节：区域	292	Section 57.2: Regions	292
第57.3节：文档注释	293	Section 57.3: Documentation comments	293
第58章：继承	295	Chapter 58: Inheritance	295
第58.1节：继承。构造函数调用顺序	295	Section 58.1: Inheritance. Constructors' calls sequence	295
第58.2节：从基类继承	297	Section 58.2: Inheriting from a base class	297
第58.3节：继承一个类并实现一个接口	298	Section 58.3: Inheriting from a class and implementing an interface	298
第58.4节：继承一个类并实现多个接口	298	Section 58.4: Inheriting from a class and implementing multiple interfaces	298
第58.5节：子类中的构造函数	299	Section 58.5: Constructors In A Subclass	299
第58.6节：继承反模式	299	Section 58.6: Inheritance Anti-patterns	299
第58.7节：扩展抽象基类	300	Section 58.7: Extending an abstract base class	300
第58.8节：测试和导航继承关系	301	Section 58.8: Testing and navigating inheritance	301
第58.9节：继承方法	301	Section 58.9: Inheriting methods	301
第58.10节：带递归类型指定的基类	302	Section 58.10: Base class with recursive type specification	302
第59章：泛型	305	Chapter 59: Generics	305
第59.1节：隐式类型推断（方法）	305	Section 59.1: Implicit type inference (methods)	305
第59.2节：类型推断（类）	306	Section 59.2: Type inference (classes)	306
第59.3节：使用以接口作为约束类型的泛型方法	306	Section 59.3: Using generic method with an interface as a constraint type	306
第59.4节：类型约束（new关键字）	307	Section 59.4: Type constraints (new-keyword)	307
第59.5节：类型约束（类和接口）	308	Section 59.5: Type constraints (classes and interfaces)	308
第59.6节：检查泛型值的相等性	309	Section 59.6: Checking equality of generic values	309
第59.7节：反射类型参数	310	Section 59.7: Reflecting on type parameters	310
第59.8节：协变	310	Section 59.8: Covariance	310
第59.9节：逆变	311	Section 59.9: Contravariance	311
第59.10节：不变性	312	Section 59.10: Invariance	312
第59.11节：变体接口	312	Section 59.11: Variant interfaces	312
第59.12节：变体委托	313	Section 59.12: Variant delegates	313
第59.13节：作为参数和返回值的变体类型	314	Section 59.13: Variant types as parameters and return values	314
第59.14节：类型参数（接口）	314	Section 59.14: Type Parameters (Interfaces)	314
第59.15节：类型约束（类和结构体）	315	Section 59.15: Type constraints (class and struct)	315
第59.16节：显式类型参数	315	Section 59.16: Explicit type parameters	315
第59.17节：类型参数（类）	315	Section 59.17: Type Parameters (Classes)	315

第59.18节：类型参数（方法）	316
第59.19节：泛型类型转换	316
第59.20节：带泛型类型转换的配置读取器	317
第60章：using语句	319
第60.1节：using语句基础	319
第60.2节：注意事项：返回你正在释放的资源	320
第60.3节：一个代码块中的多个using语句	321
第60.4节：陷阱：Dispose方法中的异常掩盖Using块中的其他错误	322
第60.5节：using语句是空安全的	322
第60.6节：使用Dispose语法定义自定义作用域	322
第60.7节：using语句与数据库连接	323
第60.8节：在约束上下文中执行代码	325
第61章：使用指令	326
第61.1节：关联别名以解决冲突	326
第61.2节：使用别名指令	326
第61.3节：访问类的静态成员	326
第61.4节：基本用法	327
第61.5节：引用命名空间	327
第61.6节：将别名与命名空间关联	327
第62章：IDisposable接口	329
第62.1节：仅包含托管资源的类中	329
第62.2节：包含托管和非托管资源的类中	329
第62.3节：IDisposable, Dispose	330
第62.4节：using关键字	330
第62.5节：包含托管资源的继承类中	331
第63章：反射	332
第63.1节：获取类型的成员	332
第63.2节：获取方法并调用	332
第63.3节：创建类型的实例	333
第63.4节：通过反射获取方法或属性的强类型委托	336
第63.5节：获取泛型方法并调用	337
第63.6节：获取System.Type	338
第63.7节：获取和设置属性	338
第63.8节：创建泛型类型的实例并调用其方法	338
第63.9节：自定义属性	339
第63.10节：实例化实现接口的类（例如插件激活）	340
第63.11节：通过带命名空间的名称获取类型	340
第63.12节：确定泛型类型实例的泛型参数	341
第63.13节：遍历类的所有属性	342
第64章：IQueryable接口	343
第64.1节：将LINQ查询转换为SQL查询	343
第65章：Linq to Objects	344
第65.1节：在C#中使用Linq to Objects	344
第66章：LINQ查询	348
第66.1节：链式方法	348
第66.2节：First、FirstOrDefault、Last、LastOrDefault、Single和SingleOrDefault	349
第66.3节：Except	352
第66.4节：SelectMany	354
第66.5节：Any	355

Section 59.18: Type Parameters (Methods)	316
Section 59.19: Generic type casting	316
Section 59.20: Configuration reader with generic type casting	317
Chapter 60: Using Statement	319
Section 60.1: Using Statement Basics	319
Section 60.2: Gotcha: returning the resource which you are disposing	320
Section 60.3: Multiple using statements with one block	321
Section 60.4: Gotcha: Exception in Dispose method masking other errors in Using blocks	322
Section 60.5: Using statements are null-safe	322
Section 60.6: Using Dispose Syntax to define custom scope	322
Section 60.7: Using Statements and Database Connections	323
Section 60.8: Executing code in constraint context	325
Chapter 61: Using Directive	326
Section 61.1: Associate an Alias to Resolve Conflicts	326
Section 61.2: Using alias directives	326
Section 61.3: Access Static Members of a Class	326
Section 61.4: Basic Usage	327
Section 61.5: Reference a Namespace	327
Section 61.6: Associate an Alias with a Namespace	327
Chapter 62: IDisposable interface	329
Section 62.1: In a class that contains only managed resources	329
Section 62.2: In a class with managed and unmanaged resources	329
Section 62.3: IDisposable, Dispose	330
Section 62.4: using keyword	330
Section 62.5: In an inherited class with managed resources	331
Chapter 63: Reflection	332
Section 63.1: Get the members of a type	332
Section 63.2: Get a method and invoke it	332
Section 63.3: Creating an instance of a Type	333
Section 63.4: Get a Strongly-Typed Delegate to a Method or Property via Reflection	336
Section 63.5: Get a generic method and invoke it	337
Section 63.6: Get a System.Type	338
Section 63.7: Getting and setting properties	338
Section 63.8: Create an instance of a Generic Type and invoke its method	338
Section 63.9: Custom Attributes	339
Section 63.10: Instantiating classes that implement an interface (e.g. plugin activation)	340
Section 63.11: Get a Type by name with namespace	340
Section 63.12: Determining generic arguments of instances of generic types	341
Section 63.13: Looping through all the properties of a class	342
Chapter 64: IQueryable interface	343
Section 64.1: Translating a LINQ query to a SQL query	343
Chapter 65: Linq to Objects	344
Section 65.1: Using LINQ to Objects in C#	344
Chapter 66: LINQ Queries	348
Section 66.1: Chaining methods	348
Section 66.2: First, FirstOrDefault, Last, LastOrDefault, Single, and SingleOrDefault	349
Section 66.3: Except	352
Section 66.4: SelectMany	354
Section 66.5: Any	355

第66.6节：连接 (JOINS)	355
第66.7节：跳过和获取	358
第66.8节：在Linq查询中定义变量 (let关键字)	358
第66.9节：Zip	359
第66.10节：范围和重复	359
第66.11节：基础	360
第66.12节：全部	360
第66.13节：总计	361
第66.14节：Distinct (去重)	362
第66.15节：SelectMany：扁平化序列的序列	362
第66.16节：GroupBy (分组)	364
第66.17节：按类型查询集合 / 将元素转换为指定类型	365
第66.18节：枚举Enumerable	366
第66.19节：在各种Linq方法中使用Range	367
第66.20节：Where	368
第66.21节：使用SelectMany代替嵌套循环	368
第66.22节：Contains	368
第66.23节：连接多个序列	370
第66.24节：基于多个键的连接	372
第66.25节：ToLookup	372
第66.26节：SkipWhile	372
第66.27节：查询排序 - OrderBy() ThenBy() OrderByDescending() ThenByDescending()	373
第66.28节：求和 (Sum)	374
第66.29节：按一个或多个字段分组 (GroupBy)	374
第66.30节：排序 (OrderBy)	375
第66.31节：Any 和 FirstOrDefault) - 最佳实践	376
第66.32节：GroupBy 求和与计数	376
第66.33节：SequenceEqual	377
第66.34节：ElementAt 和 ElementAtOrDefault	377
第66.35节：DefaultIfEmpty	378
第66.36节：ToDictionary	379
第66.37节：Concat	380
第66.38节：为 IEnumerable<T> 构建你自己的 Linq 操作符	380
第66.39节：选择 - 转换元素	381
第66.40节：降序排序 (OrderByDescending)	382
第66.41节：并集 (Union)	382
第66.42节：带外部范围变量的GroupJoin	383
第66.43节：Linq量词	383
第66.44节：条件取 (TakeWhile)	383
第66.45节：反转 (Reverse)	384
第66.46节：计数和长计数	385
第66.47节：增量构建查询	385
第66.48节：使用 Func<TSource, int, TResult> 选择器进行选择 - 用于获取元素排名	387
第67章：LINQ 到 XML	389
第67.1节：使用 LINQ 到 XML 读取 XML	389
第68章：并行 LINQ (PLINQ)	391
第68.1节：简单示例	391
第68.2节：使用并行度 (WithDegreeOfParallelism)	391
第68.3节：按顺序 (AsOrdered)	391
第68.4节：无序 (AsUnordered)	391

Section 66.6: JOINS	355
Section 66.7: Skip and Take	358
Section 66.8: Defining a variable inside a Linq query (let keyword)	358
Section 66.9: Zip	359
Section 66.10: Range and Repeat	359
Section 66.11: Basics	360
Section 66.12: All	360
Section 66.13: Aggregate	361
Section 66.14: Distinct	362
Section 66.15: SelectMany: Flattening a sequence of sequences	362
Section 66.16: GroupBy	364
Section 66.17: Query collection by type / cast elements to type	365
Section 66.18: Enumerating the Enumerable	366
Section 66.19: Using Range with various Linq methods	367
Section 66.20: Where	368
Section 66.21: Using SelectMany instead of nested loops	368
Section 66.22: Contains	368
Section 66.23: Joining multiple sequences	370
Section 66.24: Joining on multiple keys	372
Section 66.25: ToLookup	372
Section 66.26: SkipWhile	372
Section 66.27: Query Ordering - OrderBy() ThenBy() OrderByDescending() ThenByDescending()	373
Section 66.28: Sum	374
Section 66.29: GroupBy one or multiple fields	374
Section 66.30: OrderBy	375
Section 66.31: Any and FirstOrDefault) - best practice	376
Section 66.32: GroupBy Sum and Count	376
Section 66.33: SequenceEqual	377
Section 66.34: ElementAt and ElementAtOrDefault	377
Section 66.35: DefaultIfEmpty	378
Section 66.36: ToDictionary	379
Section 66.37: Concat	380
Section 66.38: Build your own Linq operators for IEnumerable<T>	380
Section 66.39: Select - Transforming elements	381
Section 66.40: OrderByDescending	382
Section 66.41: Union	382
Section 66.42: GroupJoin with outer range variable	383
Section 66.43: Linq Quantifiers	383
Section 66.44: TakeWhile	383
Section 66.45: Reverse	384
Section 66.46: Count and LongCount	385
Section 66.47: Incrementally building a query	385
Section 66.48: Select with Func<TSource, int, TResult> selector - Use to get ranking of elements	387
Chapter 67: LINQ to XML	389
Section 67.1: Read XML using LINQ to XML	389
Chapter 68: Parallel LINQ (PLINQ)	391
Section 68.1: Simple example	391
Section 68.2: WithDegreeOfParallelism	391
Section 68.3: AsOrdered	391
Section 68.4: AsUnordered	391

第69章： XmlDocument 与 System.Xml 命名空间	392	Chapter 69: XmlDocument and the System.Xml namespace	392
第69.1节： XmlDocument 与 XDocument (示例与比较)	392	Section 69.1: XmlDocument vs XDocument (Example and comparison)	392
第69.2节：从 XML 文档读取	394	Section 69.2: Reading from XML document	394
第69.3节：基本的 XML 文档交互	395	Section 69.3: Basic XML document interaction	395
第70章： XDocument 和 System.Xml.Linq 命名空间	396	Chapter 70: XDocument and the System.Xml.Linq namespace	396
第70.1节：生成 XML 文档	396	Section 70.1: Generate an XML document	396
第70.2节：使用流畅语法生成 XML 文档	396	Section 70.2: Generate an XML document using fluent syntax	396
第70.3节：修改 XML 文件	397	Section 70.3: Modify XML File	397
第71章： C# 7.0 特性	399	Chapter 71: C# 7.0 Features	399
第71.1节：元组的语言支持	399	Section 71.1: Language support for Tuples	399
第71.2节：局部函数	403	Section 71.2: Local functions	403
第71.3节：out 变量声明	404	Section 71.3: out var declaration	404
第71.4节：模式匹配	405	Section 71.4: Pattern Matching	405
第71.5节：数字分隔符	407	Section 71.5: Digit separators	407
第71.6节：二进制字面量	407	Section 71.6: Binary literals	407
第71.7节：throw 表达式	408	Section 71.7: throw expressions	408
第71.8节：扩展的表达式主体成员列表	409	Section 71.8: Extended expression bodied members list	409
第71.9节：ref 返回和 ref 局部变量	410	Section 71.9: ref return and ref local	410
第71.10节：ValueTask<T>	411	Section 71.10: ValueTask<T>	411
第72章： C# 6.0 特性	413	Chapter 72: C# 6.0 Features	413
第72.1节：异常过滤器	413	Section 72.1: Exception filters	413
第72.2节：字符串插值	417	Section 72.2: String interpolation	417
第72.3节：自动属性初始化器	423	Section 72.3: Auto-property initializers	423
第72.4节：空传播	426	Section 72.4: Null propagation	426
第72.5节：表达式主体函数成员	429	Section 72.5: Expression-bodied function members	429
第72.6节：nameof 运算符	431	Section 72.6: Operator nameof	431
第72.7节：使用静态类型	433	Section 72.7: Using static type	433
第72.8节：索引初始化器	433	Section 72.8: Index initializers	433
第72.9节：改进的重载解析	435	Section 72.9: Improved overload resolution	435
第72.10节：在 catch 和 finally 中使用 await	436	Section 72.10: Await in catch and finally	436
第72.11节：小幅更改和错误修复	437	Section 72.11: Minor changes and bugfixes	437
第72.12节：使用扩展方法进行集合初始化	437	Section 72.12: Using an extension method for collection initialization	437
第72.13节：禁用警告的增强功能	438	Section 72.13: Disable Warnings Enhancements	438
第73章： C# 5.0 特性	440	Chapter 73: C# 5.0 Features	440
第73.1节：异步与等待	440	Section 73.1: Async & Await	440
第73.2节：调用者信息属性	441	Section 73.2: Caller Information Attributes	441
第74章： C# 4.0 特性	442	Chapter 74: C# 4.0 Features	442
第74.1节：可选参数和命名参数	442	Section 74.1: Optional parameters and named arguments	442
第74.2节：协变与逆变	443	Section 74.2: Variance	443
第74.3节：动态成员查找	443	Section 74.3: Dynamic member lookup	443
第74.4节：使用 COM 时的可选 ref 关键字	444	Section 74.4: Optional ref keyword when using COM	444
第75章： C# 3.0 特性	445	Chapter 75: C# 3.0 Features	445
第75.1节：隐式类型变量 (var)	445	Section 75.1: Implicitly typed variables (var)	445
第75.2节：语言集成查询 (LINQ)	445	Section 75.2: Language Integrated Queries (LINQ)	445
第75.3节：Lambda 表达式	446	Section 75.3: Lambda expressions	446
第75.4节：匿名类型	446	Section 75.4: Anonymous types	446
第76章： 异常处理	448	Chapter 76: Exception Handling	448
第76.1节：创建自定义异常	448	Section 76.1: Creating Custom Exceptions	448
第76.2节：finally 块	450	Section 76.2: Finally block	450

第76章：异常处理	451
第76.3节：最佳实践	451
第76.4节：异常反模式	453
第76.5节：基本异常处理	455
第76.6节：处理特定异常类型	455
第76.7节：聚合异常 / 来自一个方法的多个异常	456
第76.8节：抛出异常	457
第76.9节：未处理异常和线程异常	457
第76.10节：为WCF服务实现IErrorHandler	458
第76.11节：使用异常对象	460
第76.12节：异常和try catch块的嵌套	462
第77章：空引用异常（NullReferenceException）	463
第77.1节：空引用异常（NullReferenceException）解释	463
第78章：处理将字符串转换为其他类型时的格式异常（FormatException）	464
第78.1节：将字符串转换为整数	464
第79章：阅读与理解堆栈跟踪	466
第79.1节：Windows窗体中简单空引用异常（NullReferenceException）的堆栈跟踪	466
第80章：诊断	468
第80.1节：使用TraceListeners重定向日志输出	468
第80.2节：Debug.WriteLine	468
第81章：溢出	469
第81.1节：整数溢出	469
第81.2节：操作过程中的溢出	469
第81.3节：顺序的重要性	469
第82章：入门：使用C#的Json	470
第82.1节：简单的Json示例	470
第82.2节：首先：用于处理Json的库	470
第82.3节：C#实现	470
第82.4节：序列化	471
第82.5节：反序列化	471
第82.6节：序列化与反序列化通用工具函数	471
第83章：使用json.net	473
第83.1节：在简单值上使用JsonConverter	473
第83.2节：收集JSON对象的所有字段	475
第84章：Lambda表达式	477
第84.1节：Lambda表达式作为委托初始化的简写	477
第84.2节：Lambda表达式作为事件处理器	477
第84.3节：具有多个参数或无参数的Lambda表达式	478
第84.4节：Lambda既可以作为`Func`也可以作为`Expression`发出	478
第84.5节：在语句Lambda中放置多个语句	478
第84.6节：`Func`和`Action`的Lambda表达式	479
第84.7节：使用Lambda语法创建闭包	479
第84.8节：将Lambda表达式作为参数传递给方法	479
第84.9节：基本的Lambda表达式	479
第84.10节：使用LINQ的基本Lambda表达式	480
第84.11节：带有语句块主体的Lambda语法	480
第84.12节：使用System.Linq.Expressions的Lambda表达式	480
第85章：通用Lambda查询构建器	481
第85.1节：QueryFilter类	481
第85.2节：GetExpression方法	481

Section 76.3: Best Practices	451
Section 76.4: Exception Anti-patterns	453
Section 76.5: Basic Exception Handling	455
Section 76.6: Handling specific exception types	455
Section 76.7: Aggregate exceptions / multiple exceptions from one method	456
Section 76.8: Throwing an exception	457
Section 76.9: Unhandled and Thread Exception	457
Section 76.10: Implementing IErrorHandler for WCF Services	458
Section 76.11: Using the exception object	460
Section 76.12: Nesting of Exceptions & try catch blocks	462
Chapter 77: NullReferenceException	463
Section 77.1: NullReferenceException explained	463
Chapter 78: Handling FormatException when converting string to other types	464
Section 78.1: Converting string to integer	464
Chapter 79: Read & Understand Stacktraces	466
Section 79.1: Stack trace for a simple NullReferenceException in Windows Forms	466
Chapter 80: Diagnostics	468
Section 80.1: Redirecting log output with TraceListeners	468
Section 80.2: Debug.WriteLine	468
Chapter 81: Overflow	469
Section 81.1: Integer overflow	469
Section 81.2: Overflow during operation	469
Section 81.3: Ordering matters	469
Chapter 82: Getting Started: Json with C#	470
Section 82.1: Simple Json Example	470
Section 82.2: First things First: Library to work with Json	470
Section 82.3: C# Implementation	470
Section 82.4: Serialization	471
Section 82.5: Deserialization	471
Section 82.6: Serialization & De-Serialization Common Utilities function	471
Chapter 83: Using json.net	473
Section 83.1: Using JsonConverter on simple values	473
Section 83.2: Collect all fields of JSON object	475
Chapter 84: Lambda expressions	477
Section 84.1: Lambda Expressions as Shorthand for Delegate Initialization	477
Section 84.2: Lambda Expression as an Event Handler	477
Section 84.3: Lambda Expressions with Multiple Parameters or No Parameters	478
Section 84.4: Lambdas can be emitted both as `Func` and `Expression`	478
Section 84.5: Put Multiple Statements in a Statement Lambda	478
Section 84.6: Lambdas for both `Func` and `Action`	479
Section 84.7: Using lambda syntax to create a closure	479
Section 84.8: Passing a Lambda Expression as a Parameter to a Method	479
Section 84.9: Basic lambda expressions	479
Section 84.10: Basic lambda expressions with LINQ	480
Section 84.11: Lambda syntax with statement block body	480
Section 84.12: Lambda expressions with System.Linq.Expressions	480
Chapter 85: Generic Lambda Query Builder	481
Section 85.1: QueryFilter class	481
Section 85.2: GetExpression Method	481

第85.3节：GetExpression私有重载	482	Section 85.3: GetExpression Private overload	482
第85.4节：ConstantExpression方法	483	Section 85.4: ConstantExpression Method	483
第85.5节：用法	484	Section 85.5: Usage	484
第86章：属性	485	Chapter 86: Properties	485
第86.1节：自动实现属性	485	Section 86.1: Auto-implemented properties	485
第86.2节：属性的默认值	485	Section 86.2: Default Values for Properties	485
第86.3节：公共获取	486	Section 86.3: Public Get	486
第86.4节：公共设置	486	Section 86.4: Public Set	486
第86.5节：访问属性	486	Section 86.5: Accessing Properties	486
第86.6节：只读属性	488	Section 86.6: Read-only properties	488
第86.7节：上下文中的各种属性	488	Section 86.7: Various Properties in Context	488
第87章：初始化属性	490	Chapter 87: Initializing Properties	490
第87.1节：C# 6.0：初始化自动实现属性	490	Section 87.1: C# 6.0: Initialize an Auto-Implemented Property	490
第87.2节：使用支持字段初始化属性	490	Section 87.2: Initializing Property with a Backing Field	490
第87.3节：对象实例化期间的属性初始化	490	Section 87.3: Property Initialization during object instantiation	490
第87.4节：在构造函数中初始化属性	490	Section 87.4: Initializing Property in Constructor	490
第88章：INotifyPropertyChanged接口	491	Chapter 88: INotifyPropertyChanged interface	491
第88.1节：在C# 6中实现INotifyPropertyChanged	491	Section 88.1: Implementing INotifyPropertyChanged in C# 6	491
第88.2节：带有泛型Set方法的INotifyPropertyChanged	492	Section 88.2: INotifyPropertyChanged With Generic Set Method	492
第89章：事件	494	Chapter 89: Events	494
第89.1节：声明和触发事件	494	Section 89.1: Declaring and Raising Events	494
第89.2节：事件属性	495	Section 89.2: Event Properties	495
第89.3节：创建可取消事件	496	Section 89.3: Creating cancelable event	496
第89.4节：标准事件声明	497	Section 89.4: Standard Event Declaration	497
第89.5节：匿名事件处理程序声明	498	Section 89.5: Anonymous Event Handler Declaration	498
第89.6节：非标准事件声明	498	Section 89.6: Non-Standard Event Declaration	498
第89.7节：创建包含附加数据的自定义EventArgs	499	Section 89.7: Creating custom EventArgs containing additional data	499
第90章：表达式树	501	Chapter 90: Expression Trees	501
第90.1节：使用lambda表达式创建表达式树	501	Section 90.1: Create Expression Trees with a lambda expression	501
第90.2节：使用API创建表达式树	501	Section 90.2: Creating Expression Trees by Using the API	501
第90.3节：编译表达式树	501	Section 90.3: Compiling Expression Trees	501
第90.4节：解析表达式树	502	Section 90.4: Parsing Expression Trees	502
第90.5节：表达式树基础	502	Section 90.5: Expression Tree Basic	502
第90.6节：使用访问者检查表达式结构	503	Section 90.6: Examining the Structure of an Expression using Visitor	503
第90.7节：理解表达式API	503	Section 90.7: Understanding the expressions API	503
第91章：重载解析	505	Chapter 91: Overload Resolution	505
第91.1节：基本重载示例	505	Section 91.1: Basic Overloading Example	505
第91.2节：“params”不会展开，除非必要	505	Section 91.2: “params” is not expanded, unless necessary	505
第91.3节：将null作为参数之一传递	506	Section 91.3: Passing null as one of the arguments	506
第92章：BindingList<T>	507	Chapter 92: BindingList<T>	507
第92.1节：向列表中添加项目	507	Section 92.1: Add item to list	507
第92.2节：避免N*2迭代	507	Section 92.2: Avoiding N*2 iteration	507
第93章：预处理指令	508	Chapter 93: Preprocessor directives	508
第93.1节：条件表达式	508	Section 93.1: Conditional Expressions	508
第93.2节：其他编译器指令	508	Section 93.2: Other Compiler Instructions	508
第93.3节：定义和取消定义符号	509	Section 93.3: Defining and Undefining Symbols	509
第93.4节：区域块	510	Section 93.4: Region Blocks	510
第93.5节：禁用和恢复编译器警告	510	Section 93.5: Disabling and Restoring Compiler Warnings	510
第93.6节：生成编译器警告和错误	510	Section 93.6: Generating Compiler Warnings and Errors	510

第93.7节：项目级自定义预处理器	511
第93.8节：使用条件属性	511
第94章：结构体	513
第94.1节：声明结构体	513
第94.2节：结构体的使用	514
第94.3节：结构体在赋值时的复制	515
第94.4节：实现接口的结构体	515
第95章：特性	516
第95.1节：创建自定义特性	516
第95.2节：读取特性	516
第95.3节：使用特性	517
第95.4节：DebuggerDisplay 特性	517
第95.5节：呼叫者信息属性	518
第95.6节：过时属性	519
第95.7节：从接口读取属性	519
第96章：委托	521
第96.1节：声明委托类型	521
第96.2节：Func<T, TResult>, Action<T> 和 Predicate<T> 委托类型	522
第96.3节：组合委托（多播委托）	523
第96.4节：安全调用多播委托	524
第96.5节：委托的相等性	525
第96.6节：命名方法委托的底层引用	525
第96.7节：将命名方法分配给委托	526
第96.8节：通过Lambda表达式分配给委托	527
第96.9节：在函数中封装变换	527
第96.10节：将委托作为参数传递	527
第96.11节：委托内部的闭包	528
第97章：文件和流输入输出	529
第97.1节：使用System.IO.File类从文件读取	529
第97.2节：通过IEnumerable懒惰地逐行读取文件	529
第97.3节：使用StreamWriter异步写入文本到文件	529
第97.4节：复制文件	529
第97.5节：使用System.IO.StreamWriter类写入文件行	530
第97.6节：使用System.IO.File类写入文件	530
第97.7节：创建文件	531
第97.8节：移动文件	531
第97.9节：删除文件	532
第97.10节：文件和目录	532
第98章：网络	533
第98.1节：基本TCP通信客户端	533
第98.2节：从网络服务器下载文件	533
第98.3节：异步TCP客户端	534
第98.4节：基础UDP客户端	535
第99章：执行HTTP请求	536
第99.1节：创建并发送HTTP POST请求	536
第99.2节：创建并发送HTTP GET请求	536
第99.3节：特定HTTP响应代码（如404未找到）的错误处理	537
第99.4节：获取网页的HTML（简单）	537
第99.5节：发送带有JSON主体的异步HTTP POST请求	537

Section 93.7: Custom Preprocessors at project level	511
Section 93.8: Using the Conditional attribute	511
Chapter 94: Structs	513
Section 94.1: Declaring a struct	513
Section 94.2: Struct usage	514
Section 94.3: Structs are copied on assignment	515
Section 94.4: Struct implementing interface	515
Chapter 95: Attributes	516
Section 95.1: Creating a custom attribute	516
Section 95.2: Reading an attribute	516
Section 95.3: Using an attribute	517
Section 95.4: DebuggerDisplay Attribute	517
Section 95.5: Caller info attributes	518
Section 95.6: Obsolete Attribute	519
Section 95.7: Reading an attribute from interface	519
Chapter 96: Delegates	521
Section 96.1: Declaring a delegate type	521
Section 96.2: The Func<T, TResult>, Action<T> and Predicate<T> delegate types	522
Section 96.3: Combine Delegates (Multicast Delegates)	523
Section 96.4: Safe invoke multicast delegate	524
Section 96.5: Delegate Equality	525
Section 96.6: Underlying references of named method delegates	525
Section 96.7: Assigning a named method to a delegate	526
Section 96.8: Assigning to a delegate by lambda	527
Section 96.9: Encapsulating transformations in funcs	527
Section 96.10: Passing delegates as parameters	527
Section 96.11: Closure inside a delegate	528
Chapter 97: File and Stream I/O	529
Section 97.1: Reading from a file using the System.IO.File class	529
Section 97.2: Lazily reading a file line-by-line via an IEnumerable	529
Section 97.3: Async write text to a file using StreamWriter	529
Section 97.4: Copy File	529
Section 97.5: Writing lines to a file using the System.IO.StreamWriter class	530
Section 97.6: Writing to a file using the System.IO.File class	530
Section 97.7: Create File	531
Section 97.8: Move File	531
Section 97.9: Delete File	532
Section 97.10: Files and Directories	532
Chapter 98: Networking	533
Section 98.1: Basic TCP Communication Client	533
Section 98.2: Download a file from a web server	533
Section 98.3: Async TCP Client	534
Section 98.4: Basic UDP Client	535
Chapter 99: Performing HTTP requests	536
Section 99.1: Creating and sending an HTTP POST request	536
Section 99.2: Creating and sending an HTTP GET request	536
Section 99.3: Error handling of specific HTTP response codes (such as 404 Not Found)	537
Section 99.4: Retrieve HTML for Web Page (Simple)	537
Section 99.5: Sending asynchronous HTTP POST request with JSON body	537

第100章：读取和写入.zip文件	539
第100.1节：写入zip文件	539
第100.2节：内存中写入Zip文件	539
第100.3节：从Zip文件获取文件	539
第100.4节：以下示例展示如何打开zip归档并将所有.txt文件提取到文件夹中	540
第101章：文件系统监视器	541
第101.1节：文件是否准备好	541
第101.2节：基础文件监视器	541
第102章：使用用户名和密码访问网络共享文件夹	543
第102.1节：访问网络共享文件的代码	543
第103章：异步套接字	545
第103.1节：异步套接字（客户端/服务器）示例	545
第104章：操作过滤器	552
第104.1节：自定义操作过滤器	552
第105章：多态性	553
第105.1节：多态性的类型	553
第105.2节：另一个多态性示例	554
第106章：不可变性	557
第106.1节：System.String类	557
第106.2节：字符串与不可变性	557
第107章：索引器	558
第107.1节：简单索引器	558
第107.2节：重载索引器以创建稀疏数组	558
第107.3节：带两个参数的索引器和接口	559
第108章：已检查和未检查	560
第108.1节：已检查和未检查	560
第108.2节：作为作用域的已检查和未检查	560
第109章：流	561
第109.1节：使用流	561
第110章：计时器	563
第110.1节：多线程计时器	563
第110.2节：创建计时器实例	564
第110.3节：为计时器分配“Tick”事件处理程序	565
第110.4节：示例：使用计时器执行简单倒计时	565
第111章：秒表	567
第111.1节：IsHighResolution	567
第111.2节：创建秒表实例	567
第112章：线程	569
第112.1节：避免同时读写数据	569
第112.2节：创建并启动第二个线程	570
第112.3节：Parallel.ForEach循环	570
第112.4节：死锁（持有资源并等待）	571
第112.5节：简单完整的线程演示	573
第112.6节：为每个处理器创建一个线程	574
第112.7节：使用任务的简单完整线程演示	574
第112.8节：死锁（两个线程相互等待）	575
第112.9节：显式任务并行	576

Chapter 100: Reading and writing .zip files	539
Section 100.1: Writing to a zip file	539
Section 100.2: Writing Zip Files in-memory	539
Section 100.3: Get files from a Zip file	539
Section 100.4: The following example shows how to open a zip archive and extract all .txt files to a folder	540
Chapter 101: FileSystemWatcher	541
Section 101.1: IsFileReady	541
Section 101.2: Basic FileWatcher	541
Chapter 102: Access network shared folder with username and password	543
Section 102.1: Code to access network shared file	543
Chapter 103: Asynchronous Socket	545
Section 103.1: Asynchronous Socket (Client / Server) example	545
Chapter 104: Action Filters	552
Section 104.1: Custom Action Filters	552
Chapter 105: Polymorphism	553
Section 105.1: Types of Polymorphism	553
Section 105.2: Another Polymorphism Example	554
Chapter 106: Immutability	557
Section 106.1: System.String class	557
Section 106.2: Strings and immutability	557
Chapter 107: Indexer	558
Section 107.1: A simple indexer	558
Section 107.2: Overloading the indexer to create a SparseArray	558
Section 107.3: Indexer with 2 arguments and interface	559
Chapter 108: Checked and Unchecked	560
Section 108.1: Checked and Unchecked	560
Section 108.2: Checked and Unchecked as a scope	560
Chapter 109: Stream	561
Section 109.1: Using Streams	561
Chapter 110: Timers	563
Section 110.1: Multithreaded Timers	563
Section 110.2: Creating an Instance of a Timer	564
Section 110.3: Assigning the "Tick" event handler to a Timer	565
Section 110.4: Example: Using a Timer to perform a simple countdown	565
Chapter 111: Stopwatches	567
Section 111.1: IsHighResolution	567
Section 111.2: Creating an Instance of a Stopwatch	567
Chapter 112: Threading	569
Section 112.1: Avoiding Reading and Writing Data Simultaneously	569
Section 112.2: Creating and Starting a Second Thread	570
Section 112.3: Parallel.ForEach Loop	570
Section 112.4: Deadlocks (hold resource and wait)	571
Section 112.5: Simple Complete Threading Demo	573
Section 112.6: Creating One Thread Per Processor	574
Section 112.7: Simple Complete Threading Demo using Tasks	574
Section 112.8: Deadlocks (two threads waiting on each other)	575
Section 112.9: Explicit Task Parallelism	576

第112.10节：隐式任务并行	576	Section 112.10: Implicit Task Parallelism	576
第112.11节：带参数启动线程	577	Section 112.11: Starting a thread with parameters	577
第113章：Async/await、Backgroundworker、Task和线程示例	578	Chapter 113: Async/await, Backgroundworker, Task and Thread Examples	578
第113.1节：ASP.NET 配置 Await	578	Section 113.1: ASP.NET Configure Await	578
第113.2节：任务“运行并忘记”扩展	580	Section 113.2: Task "run and forget" extension	580
第113.3节：异步/等待	580	Section 113.3: Async/await	580
第113.4节：后台工作者	581	Section 113.4: BackgroundWorker	581
第113.5节：任务	582	Section 113.5: Task	582
第113.6节：线程	583	Section 113.6: Thread	583
第114章：异步-等待	584	Chapter 114: Async-Await	584
第114.1节：等待操作符和async关键字	584	Section 114.1: Await operator and async keyword	584
第114.2节：并发调用	585	Section 114.2: Concurrent calls	585
第114.3节：Try/Catch/Finally	586	Section 114.3: Try/Catch/Finally	586
第114.4节：返回一个不带await的Task	587	Section 114.4: Returning a Task without await	587
第114.5节：只有当async/await允许机器执行额外工作时，才会提升性能	587	Section 114.5: Async/await will only improve performance if it allows the machine to do additional work	587
第114.6节：Web.config设置为目标4.5以实现正确的异步行为	588	Section 114.6: Web.config setup to target 4.5 for correct async behaviour	588
第114.7节：简单的连续调用	588	Section 114.7: Simple consecutive calls	588
第114.8节：异步代码阻塞可能导致死锁	589	Section 114.8: Blocking on async code can cause deadlocks	589
第115章：异步等待中的同步上下文	591	Chapter 115: Synchronization Context in Async-Await	591
第115.1节：async/await关键字的伪代码	591	Section 115.1: Pseudocode for async/await keywords	591
第115.2节：禁用同步上下文	591	Section 115.2: Disabling synchronization context	591
第115.3节：为什么同步上下文如此重要？	592	Section 115.3: Why SynchronizationContext is so important?	592
第116章：BackgroundWorker	593	Chapter 116: BackgroundWorker	593
第116.1节：使用BackgroundWorker完成任务	593	Section 116.1: Using a BackgroundWorker to complete a task	593
第116.2节：为BackgroundWorker分配事件处理程序	594	Section 116.2: Assigning Event Handlers to a BackgroundWorker	594
第116.3节：创建新的BackgroundWorker实例	595	Section 116.3: Creating a new BackgroundWorker instance	595
第116.4节：为BackgroundWorker分配属性	595	Section 116.4: Assigning Properties to a BackgroundWorker	595
第117章：任务并行库	596	Chapter 117: Task Parallel Library	596
第117.1节：Parallel.ForEach	596	Section 117.1: Parallel.ForEach	596
第117.2节：Parallel.For	596	Section 117.2: Parallel.For	596
第117.3节：Parallel.Invoke	597	Section 117.3: Parallel.Invoke	597
第118章：使变量线程安全	598	Chapter 118: Making a variable thread safe	598
第118.1节：在Parallel.For循环中控制对变量的访问	598	Section 118.1: Controlling access to a variable in a Parallel.For loop	598
第119章：Lock语句	599	Chapter 119: Lock Statement	599
第119.1节：在lock语句中抛出异常	599	Section 119.1: Throwing exception in a lock statement	599
第119.2节：简单用法	599	Section 119.2: Simple usage	599
第119.3节：lock语句中的return	600	Section 119.3: Return in a lock statement	600
第119.4节：反模式和陷阱	600	Section 119.4: Anti-Patterns and gotchas	600
第119.5节：使用Object实例作为锁	604	Section 119.5: Using instances of Object for lock	604
第120章：Yield关键字	605	Chapter 120: Yield Keyword	605
第120.1节：简单用法	605	Section 120.1: Simple Usage	605
第120.2节：正确检查参数	605	Section 120.2: Correctly checking arguments	605
第120.3节：提前终止	606	Section 120.3: Early Termination	606
第120.4节：更相关的用法	607	Section 120.4: More Pertinent Usage	607
第120.5节：惰性求值	608	Section 120.5: Lazy Evaluation	608
第120.6节：try...finally	609	Section 120.6: Try...finally	609
第120.7节：急切求值	610	Section 120.7: Eager evaluation	610
第120.8节：在实现IEnumerable<T>时使用yield创建IEnumerator<T>	610	Section 120.8: Using yield to create an IEnumerator<T> when implementing IEnumerable<T>	610

第120.9节：惰性求值示例：斐波那契数列	611
第120.10节：break与yield break的区别	612
第120.11节：在返回Enumerable的方法中返回另一个Enumerable	613
第121章：任务并行库（TPL）数据流构造	614
第121.1节：ActionBlock<T>	614
第121.2节：BroadcastBlock<T>	614
第121.3节：缓冲块（BufferBlock<T>）	615
第121.4节：连接块（JoinBlock<T1, T2,...>）	616
第121.5节：一次写入块（WriteOnceBlock<T>）	617
第121.6节：批量连接块（BatchedJoinBlock<T1, T2,...>）	618
第121.7节：转换块（TransformBlock<TInput, TOutput>）	619
第121.8节：TransformManyBlock<TInput, TOutput>	619
第121.9节：BatchBlock<T>	620
第122章：函数式编程	622
第122.1节：Func 和 Action	622
第122.2节：高阶函数	622
第122.3节：避免空引用	622
第122.4节：不可变性	624
第122.5节：不可变集合	625
第123章：Func 委托	626
第123.1节：无参数	626
第123.2节：多个变量	626
第123.3节：Lambda 表达式与匿名方法	627
第123.4节：协变与逆变类型参数	627
第124章：多返回值函数	629
第124.1节：“匿名对象” + “dynamic关键字”解决方案	629
第124.2节：元组解决方案	629
第124.3节：Ref和Out参数	629
第125章：二进制序列化	631
第125.1节：使用属性控制序列化行为	631
第125.2节：序列化绑定器	631
第125.3节：向后兼容性中的一些陷阱	633
第125.4节：使对象可序列化	635
第125.5节：序列化代理（实现ISerializationSurrogate）	636
第125.6节：通过实现ISerializable添加更多控制	638
第126章：ICloneable	640
第126.1节：在类中实现ICloneable	640
第126.2节：在结构体中实现ICloneable	640
第127章：IComparable	642
第127.1节：排序版本	642
第128章：访问数据库	644
第128.1节：连接字符串	644
第128.2节：实体框架连接	644
第128.3节：ADO.NET连接	645
第129章：在C#中使用SQLite	648
第129.1节：使用C#和SQLite创建简单的CRUD	648
第129.2节：执行查询	651
第130章：缓存	653
第130.1节：内存缓存（MemoryCache）	653

Section 120.9: Lazy Evaluation Example: Fibonacci Numbers	611
Section 120.10: The difference between break and yield break	612
Section 120.11: Return another Enumerable within a method returning Enumerable	613
Chapter 121: Task Parallel Library (TPL) Dataflow Constructs	614
Section 121.1: ActionBlock<T>	614
Section 121.2: BroadcastBlock<T>	614
Section 121.3: BufferBlock<T>	615
Section 121.4: JoinBlock<T1, T2,...>	616
Section 121.5: WriteOnceBlock<T>	617
Section 121.6: BatchedJoinBlock<T1, T2,...>	618
Section 121.7: TransformBlock<TInput, TOutput>	619
Section 121.8: TransformManyBlock<TInput, TOutput>	619
Section 121.9: BatchBlock<T>	620
Chapter 122: Functional Programming	622
Section 122.1: Func and Action	622
Section 122.2: Higher-Order Functions	622
Section 122.3: Avoid Null References	622
Section 122.4: Immutability	624
Section 122.5: Immutable collections	625
Chapter 123: Func delegates	626
Section 123.1: Without parameters	626
Section 123.2: With multiple variables	626
Section 123.3: Lambda & anonymous methods	627
Section 123.4: Covariant & Contravariant Type Parameters	627
Chapter 124: Function with multiple return values	629
Section 124.1: "anonymous object" + "dynamic keyword" solution	629
Section 124.2: Tuple solution	629
Section 124.3: Ref and Out Parameters	629
Chapter 125: Binary Serialization	631
Section 125.1: Controlling serialization behavior with attributes	631
Section 125.2: Serialization Binder	631
Section 125.3: Some gotchas in backward compatibility	633
Section 125.4: Making an object serializable	635
Section 125.5: Serialization surrogates (Implementing ISerializationSurrogate)	636
Section 125.6: Adding more control by implementing ISerializable	638
Chapter 126: ICloneable	640
Section 126.1: Implementing ICloneable in a class	640
Section 126.2: Implementing ICloneable in a struct	640
Chapter 127: IComparable	642
Section 127.1: Sort versions	642
Chapter 128: Accessing Databases	644
Section 128.1: Connection Strings	644
Section 128.2: Entity Framework Connections	644
Section 128.3: ADO.NET Connections	645
Chapter 129: Using SQLite in C#	648
Section 129.1: Creating simple CRUD using SQLite in C#	648
Section 129.2: Executing Query	651
Chapter 130: Caching	653
Section 130.1: MemoryCache	653

第131章：代码契约	654
第131.1节：后置条件	654
第131.2节：不变式	654
第131.3节：界面合同的定义	655
第131.4节：前提条件	656
第132章：代码契约与断言	658
第132.1节：用于检查逻辑应始终为真的断言	658
第133章：结构型设计模式	659
第133.1节：适配器设计模式	659
第134章：创建型设计模式	663
第134.1节：单例模式	663
第134.2节：工厂方法模式	664
第134.3节：抽象工厂模式	667
第134.4节：建造者模式	669
第134.5节：原型模式	673
第135章：实现装饰器设计模式	676
第135.1节：模拟自助餐厅	676
第136章：实现享元设计模式	678
第136.1节：在角色扮演游戏中实现地图	678
第137章：System.Management.Automation	681
第137.1节：调用简单同步管道	681
第138章：System.DirectoryServices.Protocols.LdapConnection	682
第138.1节：经过身份验证的SSL LDAP连接，SSL证书与反向DNS不匹配	682
第138.2节：超级简单的匿名LDAP	683
第139章：C#身份验证处理程序	684
第139.1节：身份验证处理程序	684
第140章：指针	686
第140.1节：用于数组访问的指针	686
第140.2节：指针运算	686
第140.3节：星号是类型的一部分	687
第140.4节：void*	687
第140.5节：使用 -> 访问成员	687
第140.6节：通用指针	688
第141章：指针与不安全代码	689
第141.1节：不安全代码简介	689
第141.2节：使用指针访问数组元素	690
第141.3节：编译不安全代码	690
第141.4节：使用指针检索数据值	691
第141.5节：将指针作为参数传递给方法	692
第142章：如何使用C#结构体创建联合类型（类似于C语言联合体）	693
第142.1节：C#中的C风格联合体	693
第142.2节：C#中的联合类型也可以包含结构体字段	694
第143章：响应式扩展（Rx）	696
第143.1节：观察TextBox的TextChanged事件	696
第143.2节：使用Observable从数据库流式传输数据	696
第144章：AssemblyInfo.cs示例	697
第144.1节：全局和局部AssemblyInfo	697
第144.2节：[AssemblyVersion]	697

Chapter 131: Code Contracts	654
Section 131.1: Postconditions	654
Section 131.2: Invariants	654
Section 131.3: Defining Contracts on Interface	655
Section 131.4: Preconditions	656
Chapter 132: Code Contracts and Assertions	658
Section 132.1: Assertions to check logic should always be true	658
Chapter 133: Structural Design Patterns	659
Section 133.1: Adapter Design Pattern	659
Chapter 134: Creational Design Patterns	663
Section 134.1: Singleton Pattern	663
Section 134.2: Factory Method pattern	664
Section 134.3: Abstract Factory Pattern	667
Section 134.4: Builder Pattern	669
Section 134.5: Prototype Pattern	673
Chapter 135: Implementing Decorator Design Pattern	676
Section 135.1: Simulating cafeteria	676
Chapter 136: Implementing Flyweight Design Pattern	678
Section 136.1: Implementing map in RPG game	678
Chapter 137: System.Management.Automation	681
Section 137.1: Invoke simple synchronous pipeline	681
Chapter 138: System.DirectoryServices.Protocols.LdapConnection	682
Section 138.1: Authenticated SSL LDAP connection, SSL cert does not match reverse DNS	682
Section 138.2: Super Simple anonymous LDAP	683
Chapter 139: C# Authentication handler	684
Section 139.1: Authentication handler	684
Chapter 140: Pointers	686
Section 140.1: Pointers for array access	686
Section 140.2: Pointer arithmetic	686
Section 140.3: The asterisk is part of the type	687
Section 140.4: void*	687
Section 140.5: Member access using ->	687
Section 140.6: Generic pointers	688
Chapter 141: Pointers & Unsafe Code	689
Section 141.1: Introduction to unsafe code	689
Section 141.2: Accessing Array Elements Using a Pointer	690
Section 141.3: Compiling Unsafe Code	690
Section 141.4: Retrieving the Data Value Using a Pointer	691
Section 141.5: Passing Pointers as Parameters to Methods	692
Chapter 142: How to use C# Structs to create a Union type (Similar to C Unions)	693
Section 142.1: C-Style Unions in C#	693
Section 142.2: Union Types in C# can also contain Struct fields	694
Chapter 143: Reactive Extensions (Rx)	696
Section 143.1: Observing TextChanged event on a TextBox	696
Section 143.2: Streaming Data from Database with Observable	696
Chapter 144: AssemblyInfo.cs Examples	697
Section 144.1: Global and local AssemblyInfo	697
Section 144.2: [AssemblyVersion]	697

第144.3节：自动版本控制	698
第144.4节：常用字段	698
第144.5节：[AssemblyTitle]	698
第144.6节：[AssemblyProduct]	698
第144.7节：[InternalsVisibleTo]	698
第144.8节：[AssemblyConfiguration]	699
第144.9节：[AssemblyKeyFile]	699
第144.10节：读取程序集属性	699
第145章：使用纯文本编辑器和C#编译器（csc.exe）创建控制台应用程序	
编译器（csc.exe）	701
第145.1节：使用纯文本编辑器和C#编译器创建控制台应用程序	701
第146章：CLSCCompliantAttribute	703
第146.1节：适用CLS规则的访问修饰符	703
第146.2节：违反CLS规则：无符号类型 / sbyte	703
第146.3节：违反CLS规则：相同命名	704
第146.4节：违反CLS规则：标识符	705
第146.5节：违反CLS规则：继承自非CLS兼容类	705
第147章：ObservableCollection<T>	706
第147.1节：初始化ObservableCollection<T>	706
第148章：哈希函数	707
第148.1节：用于密码哈希的PBKDF2	707
第148.2节：使用Pbkdf2的完整密码哈希解决方案	707
第148.3节：MD5	711
第148.4节：SHA1	711
第148.5节：SHA256	712
第148.6节：SHA384	712
第148.7节：SHA512	713
第149章：在C#中生成随机数	714
第149.1节：生成随机整数	714
第149.2节：生成指定范围内的随机整数	714
第149.3节：重复生成相同的随机数序列	714
第149.4节：同时使用不同种子创建多个随机类	715
第149.5节：生成一个随机双精度数	715
第149.6节：生成一个随机字符	715
第149.7节：生成一个最大值百分比的数字	715
第150章：密码学（System.Security.Cryptography）	717
第150.1节：字符串对称认证加密的现代示例	717
第150.2节：对称加密与非对称加密简介	728
第150.3节：简单对称文件加密	729
第150.4节：密码学安全的随机数据	730
第150.5节：密码哈希	731
第150.6节：快速非对称文件加密	731
第151章：ASP.NET身份认证	736
第151.1节：如何使用用户管理器在ASP.NET身份认证中实现密码重置令牌	736
第152章：.NET中的不安全代码	739
第152.1节：在数组中使用unsafe	739
第152.2节：在字符串中使用unsafe	739
第152.3节：不安全的数组索引	740
第153章：C#脚本	741

Section 144.3: Automated versioning	698
Section 144.4: Common fields	698
Section 144.5: [AssemblyTitle]	698
Section 144.6: [AssemblyProduct]	698
Section 144.7: [InternalsVisibleTo]	698
Section 144.8: [AssemblyConfiguration]	699
Section 144.9: [AssemblyKeyFile]	699
Section 144.10: Reading Assembly Attributes	699
Chapter 145: Creating a Console Application using a Plain-Text Editor and the C# Compiler (csc.exe)	701
Section 145.1: Creating a Console application using a Plain-Text Editor and the C# Compiler	701
Chapter 146: CLSCCompliantAttribute	703
Section 146.1: Access Modifier to which CLS rules apply	703
Section 146.2: Violation of CLS rule: Unsigned types / sbyte	703
Section 146.3: Violation of CLS rule: Same naming	704
Section 146.4: Violation of CLS rule: Identifier	705
Section 146.5: Violation of CLS rule: Inherit from non CLSComplaint class	705
Chapter 147: ObservableCollection<T>	706
Section 147.1: Initialize ObservableCollection<T>	706
Chapter 148: Hash Functions	707
Section 148.1: PBKDF2 for Password Hashing	707
Section 148.2: Complete Password Hashing Solution using Pbkdf2	707
Section 148.3: MD5	711
Section 148.4: SHA1	711
Section 148.5: SHA256	712
Section 148.6: SHA384	712
Section 148.7: SHA512	713
Chapter 149: Generating Random Numbers in C#	714
Section 149.1: Generate a random int	714
Section 149.2: Generate a random int in a given range	714
Section 149.3: Generating the same sequence of random numbers over and over again	714
Section 149.4: Create multiple random class with different seeds simultaneously	715
Section 149.5: Generate a Random double	715
Section 149.6: Generate a random character	715
Section 149.7: Generate a number that is a percentage of a max value	715
Chapter 150: Cryptography (System.Security.Cryptography)	717
Section 150.1: Modern Examples of Symmetric Authenticated Encryption of a string	717
Section 150.2: Introduction to Symmetric and Asymmetric Encryption	728
Section 150.3: Simple Symmetric File Encryption	729
Section 150.4: Cryptographically Secure Random Data	730
Section 150.5: Password Hashing	731
Section 150.6: Fast Asymmetric File Encryption	731
Chapter 151: ASP.NET Identity	736
Section 151.1: How to implement password reset token in asp.net identity using user manager	736
Chapter 152: Unsafe Code in .NET	739
Section 152.1: Using unsafe with arrays	739
Section 152.2: Using unsafe with strings	739
Section 152.3: Unsafe Array Index	740
Chapter 153: C# Script	741

第153.1节：简单代码评估	741
第154章：运行时编译	742
第154.1节：RoslynScript	742
第154.2节：CSharpCodeProvider	742
第155章：互操作性	743
第155.1节：从非托管C++ DLL导入函数	743
第155.2节：调用约定	743
第155.3节：C++名称修饰	744
第155.4节：非托管DLL的动态加载和卸载	744
第155.5节：使用Marshal读取结构体	745
第155.6节：处理Win32错误	746
第155.7节：固定对象	747
第155.8节：用于COM的简单类暴露代码	748
第156章：.NET编译器平台（Roslyn）	750
第156.1节：语义模型	750
第156.2节：语法树	750
第156.3节：从MSBuild项目创建工作区	751
第157章：IL生成器	752
第157.1节：创建包含Unix时间戳辅助方法的动态程序集	752
第157.2节：创建方法重写	753
第158章：T4代码生成	755
第158.1节：运行时代码生成	755
第159章：在Windows窗体应用程序中创建自定义消息框	756
第159.1节：如何在另一个Windows窗体应用程序中使用自定义消息框控件	756
第159.2节：创建自定义消息框控件	756
第160章：包含字体资源	759
第160.1节：从资源实例化“Fontfamily”	759
第160.2节：集成方法	759
第160.3节：与“按钮”的使用	759
第161章：导入谷歌联系人	761
第161.1节：要求	761
第161.2节：控制器中的源代码	761
第161.3节：视图中的源代码	764
第162章：.Net中的垃圾回收器	765
第162.1节：弱引用	765
第162.2节：大对象堆压缩	766
第163章：Microsoft.Exchange.WebServices	767
第163.1节：检索指定用户的外出设置	767
第163.2节：更新指定用户的外出设置	767
第164章：Windows通信基础	769
第164.1节：入门示例	769
鸣谢	771
你可能也喜欢	785

Section 153.1: Simple code evaluation	741
Chapter 154: Runtime Compile	742
Section 154.1: RoslynScript	742
Section 154.2: CSharpCodeProvider	742
Chapter 155: Interoperability	743
Section 155.1: Import function from unmanaged C++ DLL	743
Section 155.2: Calling conventions	743
Section 155.3: C++ name mangling	744
Section 155.4: Dynamic loading and unloading of unmanaged DLLs	744
Section 155.5: Reading structures with Marshal	745
Section 155.6: Dealing with Win32 Errors	746
Section 155.7: Pinned Object	747
Section 155.8: Simple code to expose class for com	748
Chapter 156: .NET Compiler Platform (Roslyn)	750
Section 156.1: Semantic model	750
Section 156.2: Syntax tree	750
Section 156.3: Create workspace from MSBuild project	751
Chapter 157: ILGenerator	752
Section 157.1: Creates a DynamicAssembly that contains a UnixTimestamp helper method	752
Section 157.2: Create method override	753
Chapter 158: T4 Code Generation	755
Section 158.1: Runtime Code Generation	755
Chapter 159: Creating Own MessageBox in Windows Form Application	756
Section 159.1: How to use own created MessageBox control in another Windows Form application	756
Section 159.2: Creating Own MessageBox Control	756
Chapter 160: Including Font Resources	759
Section 160.1: Instantiate 'Fontfamily' from Resources	759
Section 160.2: Integration method	759
Section 160.3: Usage with a 'Button'	759
Chapter 161: Import Google Contacts	761
Section 161.1: Requirements	761
Section 161.2: Source code in the controller	761
Section 161.3: Source code in the view	764
Chapter 162: Garbage Collector in .Net	765
Section 162.1: Weak References	765
Section 162.2: Large Object Heap compaction	766
Chapter 163: Microsoft.Exchange.WebServices	767
Section 163.1: Retrieve Specified User's Out of Office Settings	767
Section 163.2: Update Specific User's Out of Office Settings	767
Chapter 164: Windows Communication Foundation	769
Section 164.1: Getting started sample	769
Credits	771
You may also like	785

请随意免费分享此 PDF，
本书的最新版本可从以下网址下载：

<https://goalkicker.com/CSharpBook>

这本 C# 专业人士笔记是从 [Stack Overflow Documentation](#) 汇编而成，内容由 Stack Overflow 的优秀人士撰写。
文本内容采用知识共享署名-相同方式共享许可协议发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非另有说明。

这是一本非官方的免费书籍，旨在教育用途，与官方 C# 组织或公司及 Stack Overflow 无关。所有商标和注册商标均为其各自公司所有者所有。

本书中提供的信息不保证正确或准确，使用风险自负

请将反馈和更正发送至 web@petercv.com

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:

<https://goalkicker.com/CSharpBook>

This C# Notes for Professionals book is compiled from [Stack Overflow Documentation](#), the content is written by the beautiful people at Stack Overflow.
Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official C# group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：C#语言入门

版本 发布日期

1.0 2002-01-01
1.2 2003-04-01
2.0 2005-09-01
3.0 2007-08-01
4.0 2010-04-01
5.0 2013-06-01
6.0 2015-07-01
7.0 2017-03-07

第1.1节：创建新的控制台应用程序（Visual Studio）

1. 打开Visual Studio
2. 在工具栏中，依次点击文件→新建项目
3. 选择控制台应用程序项目类型
4. 在解决方案资源管理器中打开文件Program.cs
5. 在Main()中添加以下代码：

```
public class Program
{
    public static void Main()
    {
        // 在控制台打印一条消息。
        System.Console.WriteLine("Hello, World!");

        /* 等待用户按下一个键。这是防止控制台窗口在程序员查看窗口内容
        之前终止并消失的常用方法，当应用程序通过VS中的“开始”运行时尤其如此。*/
        System.Console.ReadKey();
    }
}
```

6. 在工具栏中，点击Debug->Start Debugging，或按F5，或ctrl + F5（无调试器运行）来运行程序。

[ideone上的实时演示](#)

说明

- class Program 是一个类声明。类Program包含程序使用的数据和方法定义。类通常包含多个方法。方法定义了类的行为。
然而，Program类只有一个方法：Main。
- static void Main() 定义了Main方法，这是所有C#程序的入口点。Main方法说明了类在执行时的行为。每个类只允许有一个Main方法。
- System.Console.WriteLine("Hello, world!"); 方法在控制台窗口输出给定的数据（本例中为Hello, world!）。

Chapter 1: Getting started with C# Language

Version Release Date

1.0 2002-01-01
[1.2](#) 2003-04-01
[2.0](#) 2005-09-01
[3.0](#) 2007-08-01
[4.0](#) 2010-04-01
[5.0](#) 2013-06-01
[6.0](#) 2015-07-01
[7.0](#) 2017-03-07

Section 1.1: Creating a new console application (Visual Studio)

1. Open Visual Studio
2. In the toolbar, go to **File → New Project**
3. Select the **Console Application** project type
4. Open the file Program.cs in the Solution Explorer
5. Add the following code to Main():

```
public class Program
{
    public static void Main()
    {
        // Prints a message to the console.
        System.Console.WriteLine("Hello, World!");

        /* Wait for the user to press a key. This is a common
        way to prevent the console window from terminating
        and disappearing before the programmer can see the contents
        of the window, when the application is run via Start from within VS. */
        System.Console.ReadKey();
    }
}
```

6. In the toolbar, click **Debug -> Start Debugging** or hit **F5** or **ctrl + F5** (running without debugger) to run the program.

[Live Demo on ideone](#)

Explanation

- `class Program` is a class declaration. The class Program contains the data and method definitions that your program uses. Classes generally contain multiple methods. Methods define the behavior of the class. However, the Program class has only one method: Main.
- `static void Main()` defines the Main method, which is the entry point for all C# programs. The Main method states what the class does when executed. Only one Main method is allowed per class.
- `System.Console.WriteLine("Hello, world!");` method prints a given data (in this example, Hello, world!) as an output in the console window.

- `System.Console.ReadKey()` 确保程序在显示消息后不会立即关闭。它通过等待用户按下键盘上的任意键来实现。用户按下任意键后程序终止。程序在执行完`main()`方法的最后一行代码后终止。

使用命令行

通过命令行编译时，可以使用MSBuild或csc.exe（C#编译器），它们都是Microsoft Build Tools包的一部分。

要编译此示例，请在包含`HelloWorld.cs`文件的同一目录下运行以下命令：

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe HelloWorld.cs
```

也有可能在一个应用程序中有两个`main`方法。在这种情况下，您必须通过在控制台中输入以下命令告诉编译器执行哪个`main`方法。（假设`ClassA`也在同一`HelloWorld`命名空间的`HelloWorld.cs`文件中有一个`main`方法）

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe HelloWorld.cs /main:HelloWorld.ClassA
```

其中`HelloWorld`是命名空间

注意：这是.NET Framework v4.0通常所在的路径。请根据您的.NET版本更改路径。
此外，如果您使用的是32位.NET Framework，目录可能是framework而不是framework64。从Windows命令提示符，您可以通过运行以下命令列出所有csc.exe Framework路径（第一个是32位Framework）：

```
dir %WINDIR%\Microsoft.NET\Framework\csc.exe /s/b
dir %WINDIR%\Microsoft.NET\Framework64\csc.exe /s/b
```

```
C:\Users\Main\Documents\helloworld>%WINDIR%\Microsoft.NET\Framework\v4.0.30319\csc.exe HelloWorld.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17929
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.

C:\Users\Main\Documents\helloworld>dir
Volume in drive C is System

 Directory of C:\Users\Main\Documents\helloworld

07/26/2016  03:43 PM    <DIR>    .
07/26/2016  03:43 PM    <DIR>    ..
07/26/2016  03:41 PM           258 HelloWorld.cs
07/26/2016  03:43 PM           3,584 HelloWorld.exe
              2 File(s)        3,842 bytes
              2 Dir(s)   99,699,073,024 bytes free

C:\Users\Main\Documents\helloworld>
```

现在同一目录下应该有一个名为`HelloWorld.exe`的可执行文件。要从命令提示符执行程序，只需输入可执行文件名并按 **Enter** 如下：

```
HelloWorld.exe
```

这将产生：

你好，世界！

- `System.Console.ReadKey()` ensures that the program won't close immediately after displaying the message. It does this by waiting for the user to press a key on the keyboard. Any key press from the user will terminate the program. The program terminates when it has finished the last line of code in the `main()` method.

Using the command line

To compile via command line use either MSBuild or csc.exe (the C# compiler), both part of the [Microsoft Build Tools](#) package.

To compile this example, run the following command in the same directory where `HelloWorld.cs` is located:

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe HelloWorld.cs
```

It can also be possible that you have two `main` methods inside one application. In this case, you have to tell the compiler which `main` method to execute by typing the following command in the **console**.(suppose Class A also has a `main` method in the same `HelloWorld.cs` file in `HelloWorld` namespace)

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe HelloWorld.cs /main:HelloWorld.ClassA
```

where `HelloWorld` is namespace

Note: This is the path where **.NET framework v4.0** is located in general. Change the path according to your .NET version. In addition, the directory might be **framework** instead of **framework64** if you're using the 32-bit .NET Framework. From the Windows Command Prompt, you can list all the csc.exe Framework paths by running the following commands (the first for 32-bit Frameworks):

```
dir %WINDIR%\Microsoft.NET\Framework\csc.exe /s/b
dir %WINDIR%\Microsoft.NET\Framework64\csc.exe /s/b
```

```
C:\Users\Main\Documents\helloworld>%WINDIR%\Microsoft.NET\Framework\v4.0.30319\csc.exe HelloWorld.cs
Microsoft (R) Visual C# Compiler version 4.0.30319.17929
for Microsoft (R) .NET Framework 4.5
Copyright (C) Microsoft Corporation. All rights reserved.

C:\Users\Main\Documents\helloworld>dir
Volume in drive C is System

 Directory of C:\Users\Main\Documents\helloworld

07/26/2016  03:43 PM    <DIR>    .
07/26/2016  03:43 PM    <DIR>    ..
07/26/2016  03:41 PM           258 HelloWorld.cs
07/26/2016  03:43 PM           3,584 HelloWorld.exe
              2 File(s)        3,842 bytes
              2 Dir(s)   99,699,073,024 bytes free

C:\Users\Main\Documents\helloworld>
```

There should now be an executable file named `HelloWorld.exe` in the same directory. To execute the program from the command prompt, simply type the executable's name and hit **Enter** as follows:

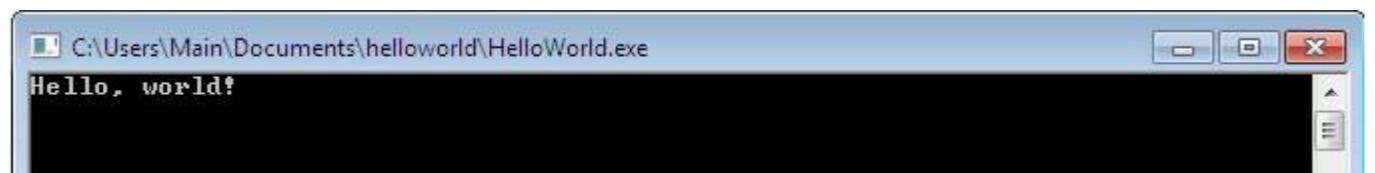
```
HelloWorld.exe
```

This will produce:

Hello, world!

C:\Users\Main\Documents\helloworld>HelloWorld
Hello, world!

你也可以双击该可执行文件，打开一个新的控制台窗口，显示消息“你好，世界！”



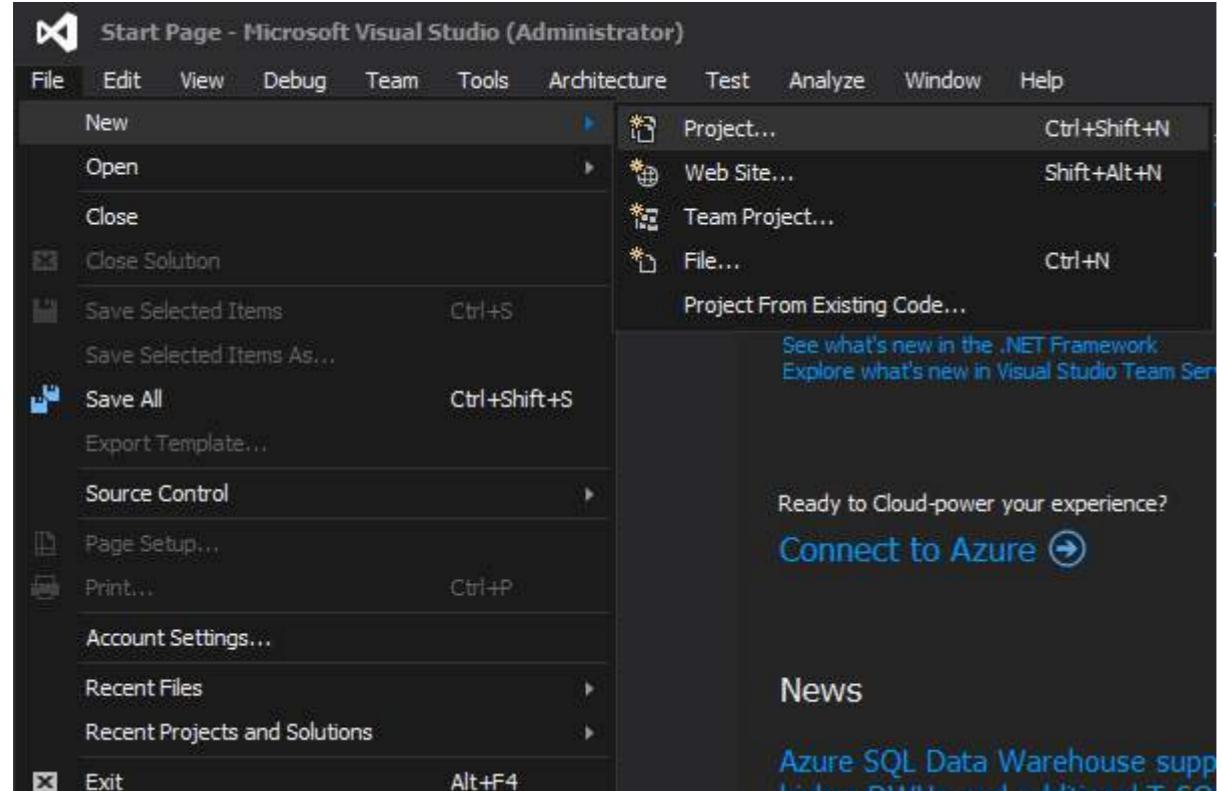
第1.2节：在Visual Studio中创建新项目（控制台应用程序）并以调试模式运行

1. 下载并安装Visual Studio。Visual Studio可以从VisualStudio.com下载。

建议使用社区版，首先因为它是免费的，其次因为它包含所有通用功能，并且可以进一步扩展。

2. 打开Visual Studio。

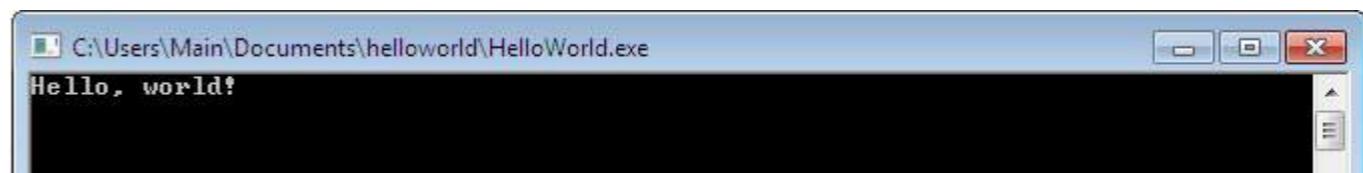
3. 欢迎。进入文件→新建→项目。



4. 点击 模板 → Visual C# → 控制台应用程序

C:\Users\Main\Documents\helloworld>HelloWorld
Hello, world!

You may also double click the executable and launch a new console window with the message "Hello, world!"

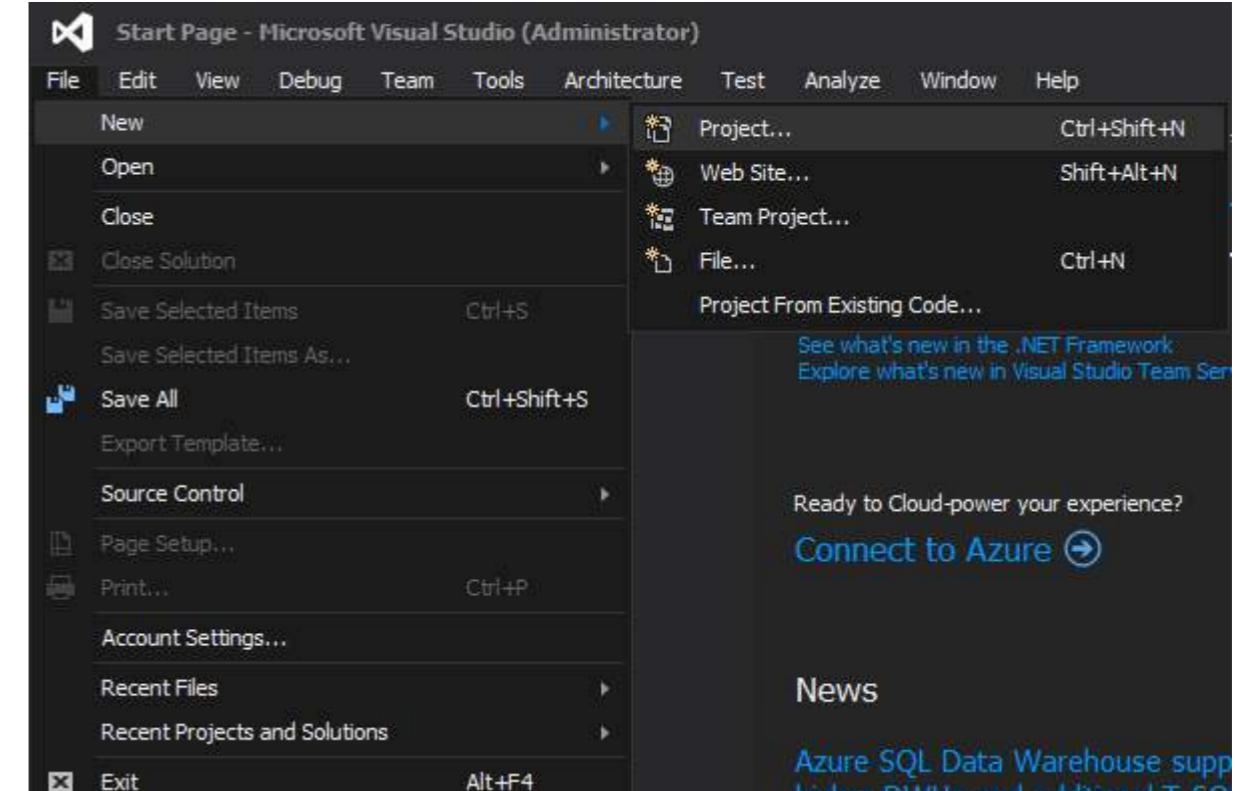


Section 1.2: Creating a new project in Visual Studio (console application) and Running it in Debug mode

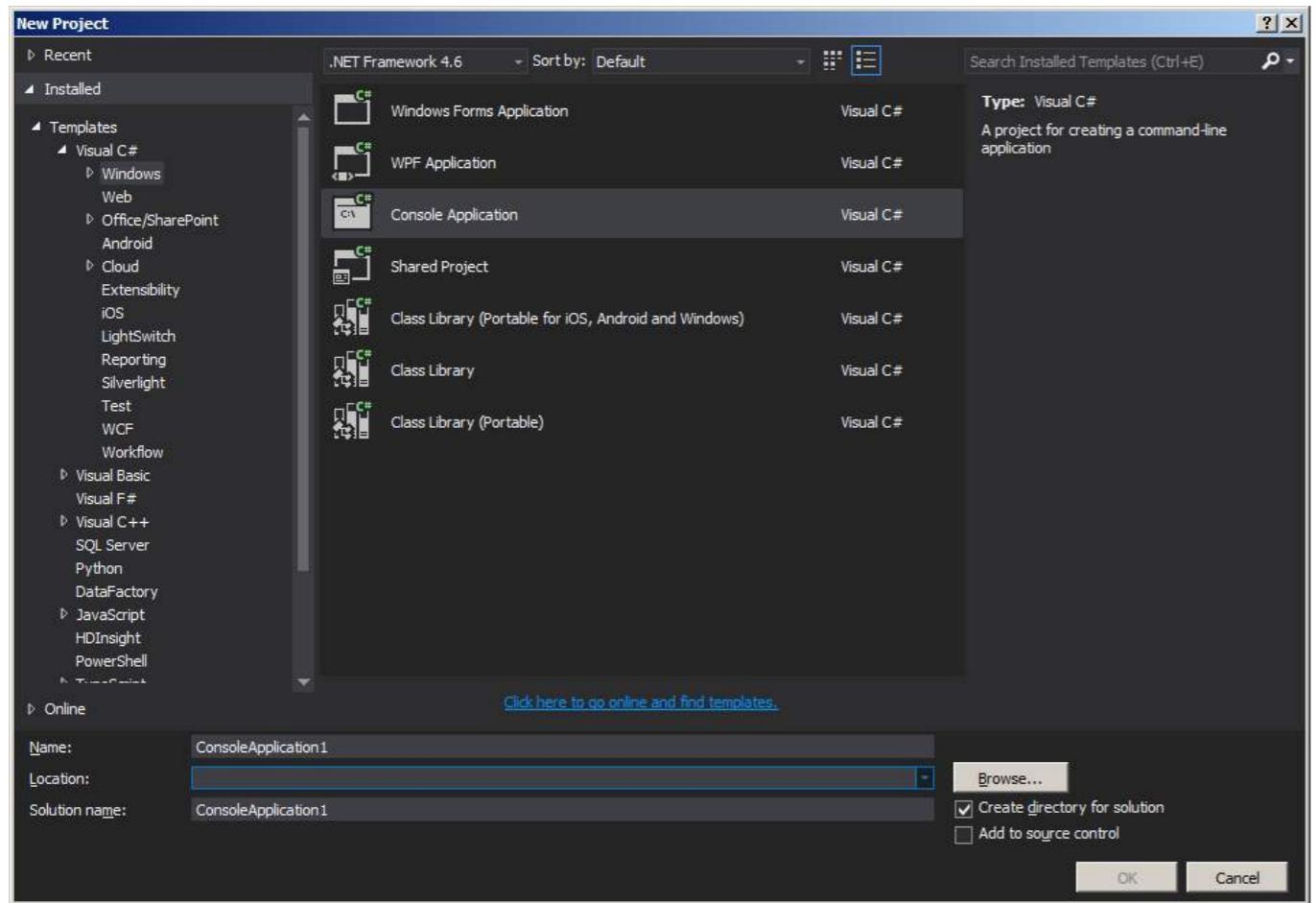
1. Download and install Visual Studio. Visual Studio can be downloaded from VisualStudio.com. The Community edition is suggested, first because it is free, and second because it involves all the general features and can be extended further.

2. Open Visual Studio.

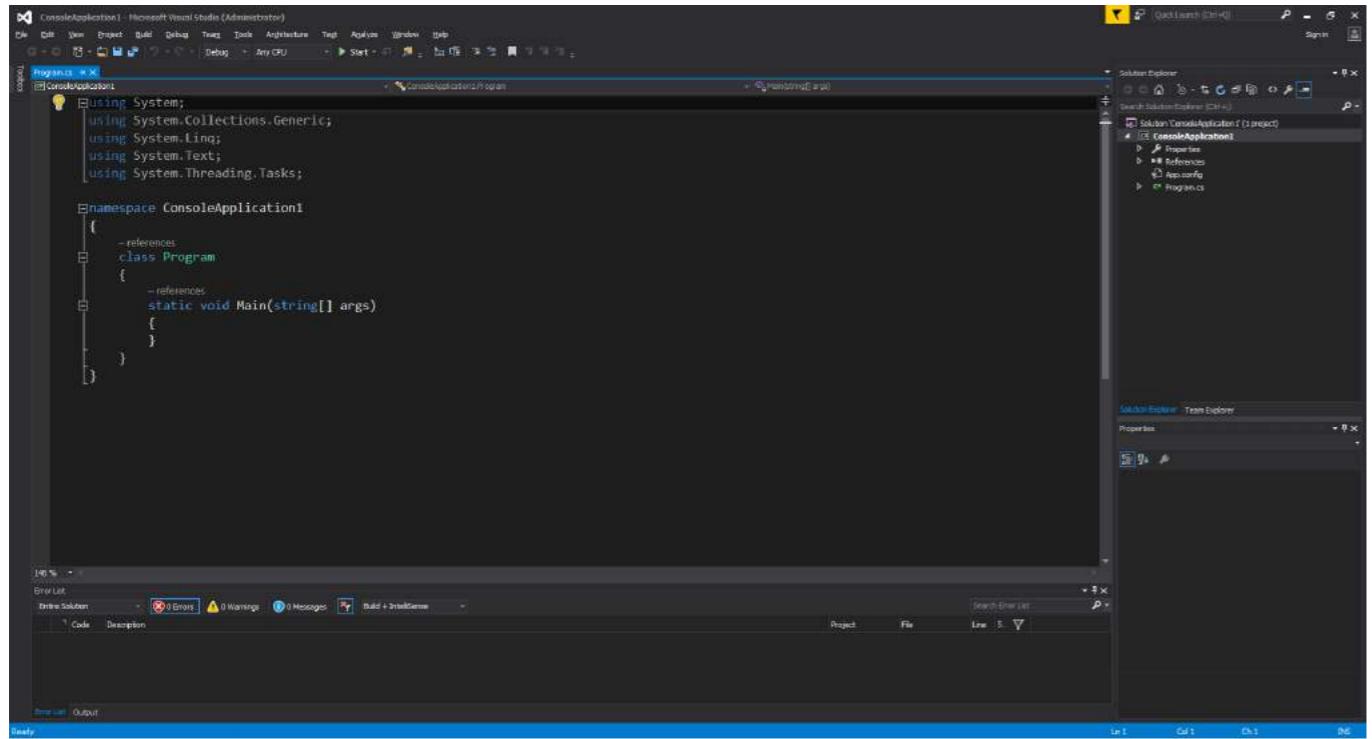
3. Welcome. Go to File → New → Project.



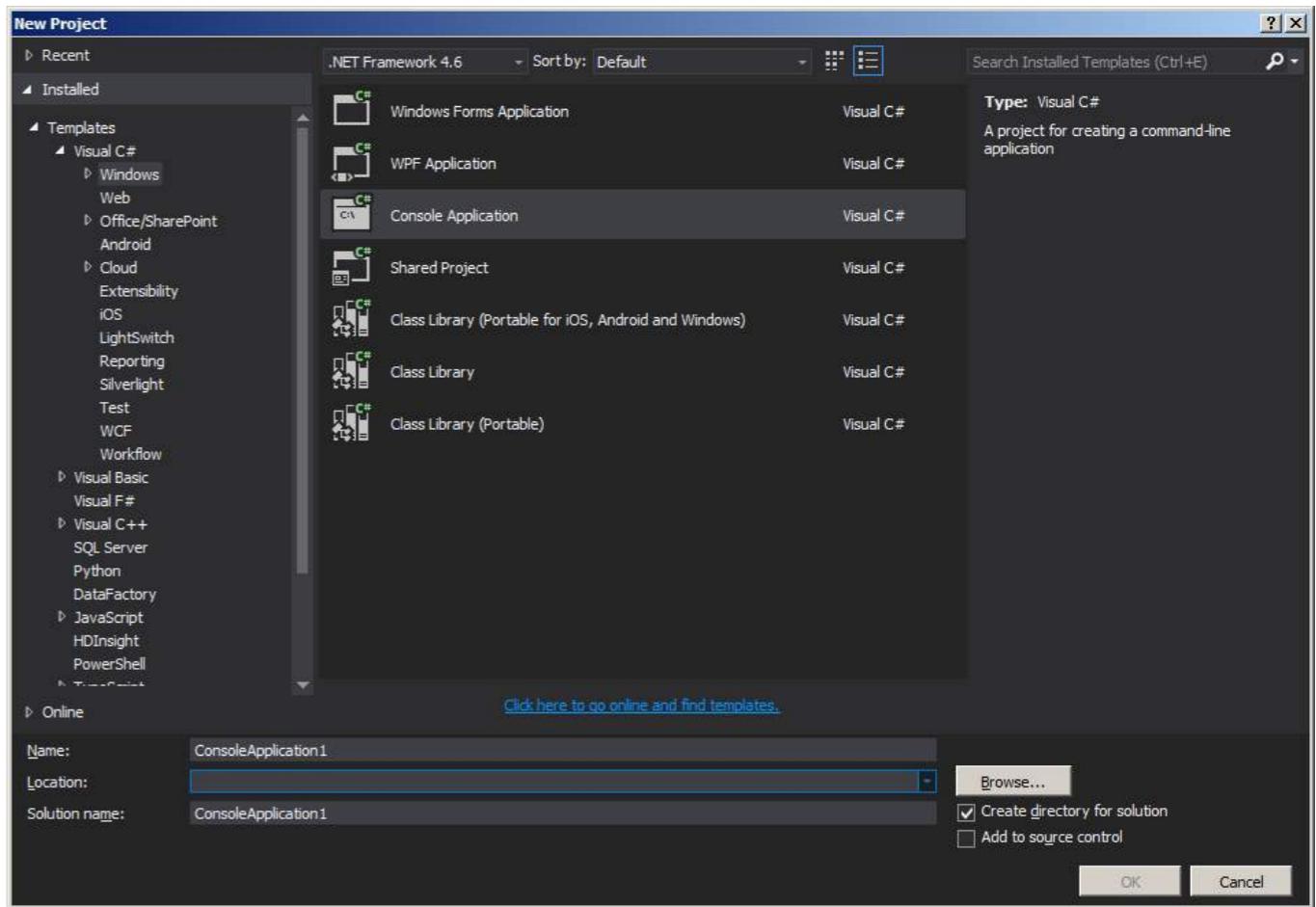
4. Click Templates → Visual C# → Console Application



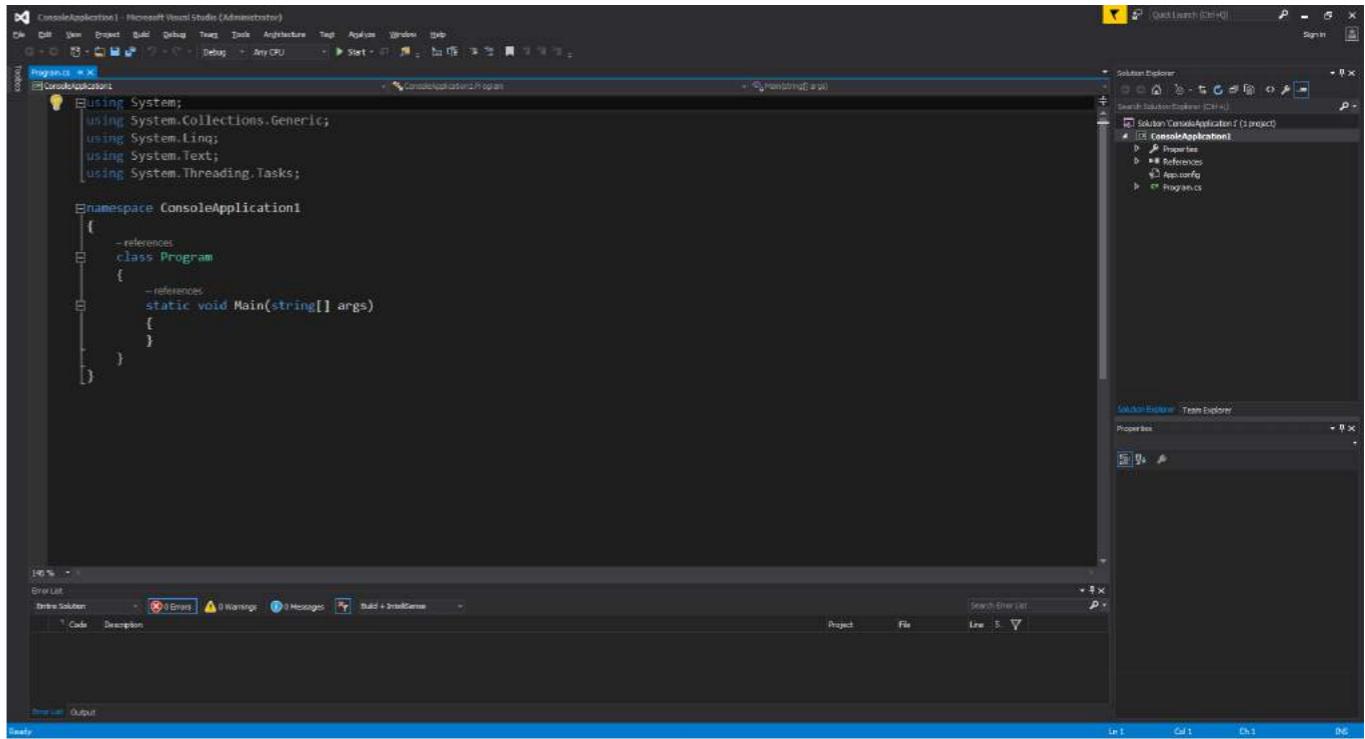
5. 选择控制台应用程序后，输入项目名称和保存位置，然后按 **确定**。
不用担心解决方案名称。
6. 项目已创建。新创建的项目将类似于：



(始终为项目使用描述性名称，以便于与其他项目区分。建议项目或类名中不要使用空格。)



5. After selecting **Console Application**, Enter a name for your project, and a location to save and press **OK**.
Don't worry about the Solution name.
6. **Project created.** The newly created project will look similar to:



(Always use descriptive names for projects so that they can easily be distinguished from other projects. It is recommended not to use spaces in project or class name.)

7. 编写代码。现在你可以更新你的 Program.cs 文件，向用户显示“Hello world!”。

```
using System;

namespace ConsoleApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
        }
    }
}
```

将以下两行添加到Program.cs中的public static void Main(string[] args)对象内：(确保在大括号内)

```
Console.WriteLine("Hello world!");
Console.Read();
```

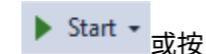
为什么使用Console.Read()？第一行将文本“Hello world!”打印到控制台，第二行等待输入一个字符；实际上，这会使程序暂停执行，以便在调试时你能够看到输出。没有Console.Read()；当你开始调试应用程序时，它只会将“Hello world!”打印到控制台，然后立即关闭。你的代码窗口现在应该如下所示：

```
using System;

namespace ConsoleApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
            Console.Read();
        }
    }
}
```

8. 调试你的程序。按窗口顶部工具栏上的开始按钮

F5 按键盘上的键运行您的应用程序。如果按钮不存在，您可以从顶部菜单运行程序：调试→开始调试。程序将编译然后打开一个控制台窗口。它应该类似于以下截图：



或按

7. Write code. You can now update your Program.cs to present "Hello world!" to the user.

```
using System;

namespace ConsoleApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
        }
    }
}
```

Add the following two lines to the `public static void Main(string[] args)` object in Program.cs: (make sure it's inside the braces)

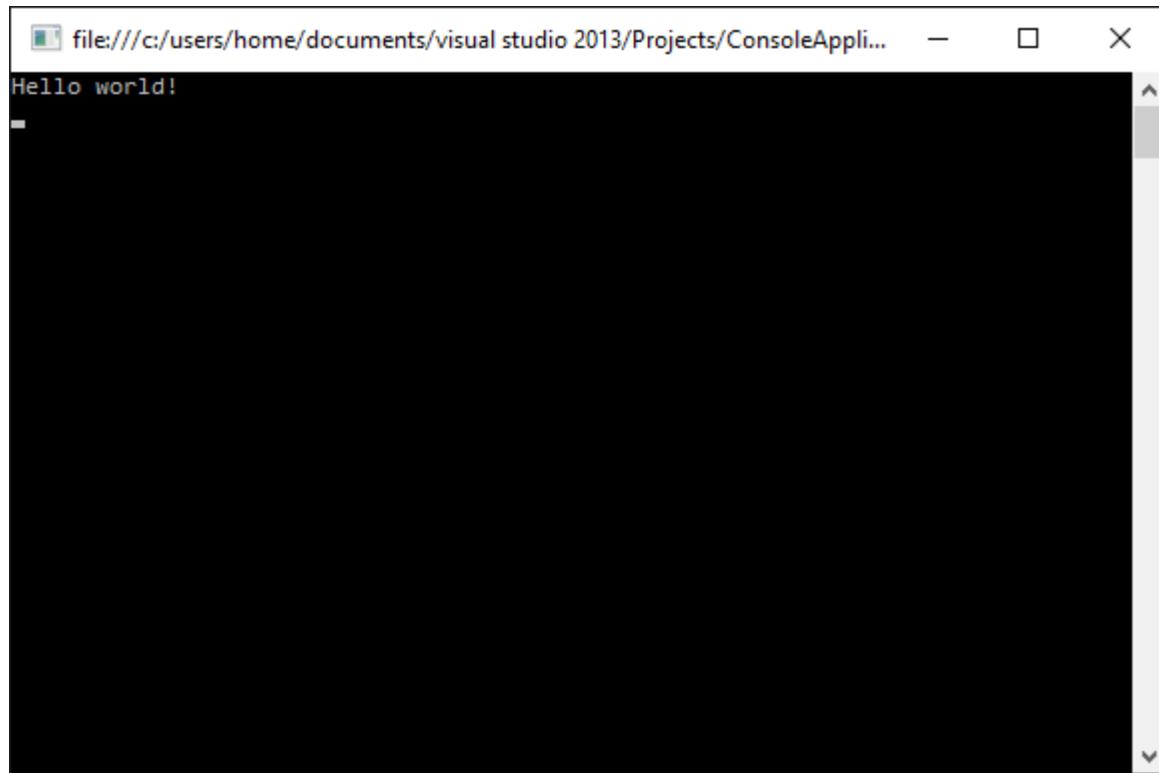
```
Console.WriteLine("Hello world!");
Console.Read();
```

Why Console.Read()? The first line prints out the text "Hello world!" to the console, and the second line waits for a single character to be entered; in effect, this causes the program to pause execution so that you're able to see the output while debugging. Without `Console.Read()`; when you start debugging the application it will just print "Hello world!" to the console and then immediately close. Your code window should now look like the following:

```
using System;

namespace ConsoleApplication1
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello world!");
            Console.Read();
        }
    }
}
```

8. Debug your program. Press the Start Button on the toolbar near the top of the window  or press F5 on your keyboard to run your application. If the button is not present, you can run the program from the top menu: **Debug → Start Debugging**. The program will compile and then open a console window. It should look similar to the following screenshot:



```
file:///c:/users/home/documents/visual studio 2013/Projects/ConsoleAppl... - □ ×  
Hello world!
```

9. 停止程序。要关闭程序，只需按键盘上的任意键。控制台。读取()我们添加的功能就是为了这个目的。关闭程序的另一种方法是进入有开始按钮的菜单，然后点击停止按钮。

第1.3节：使用.NET Core创建新程序

首先安装.NET Core SDK，按照您选择的平台的安装说明进行操作：

- [Windows](#)
- [OSX](#)
- [Linux](#)
- [Docker](#)

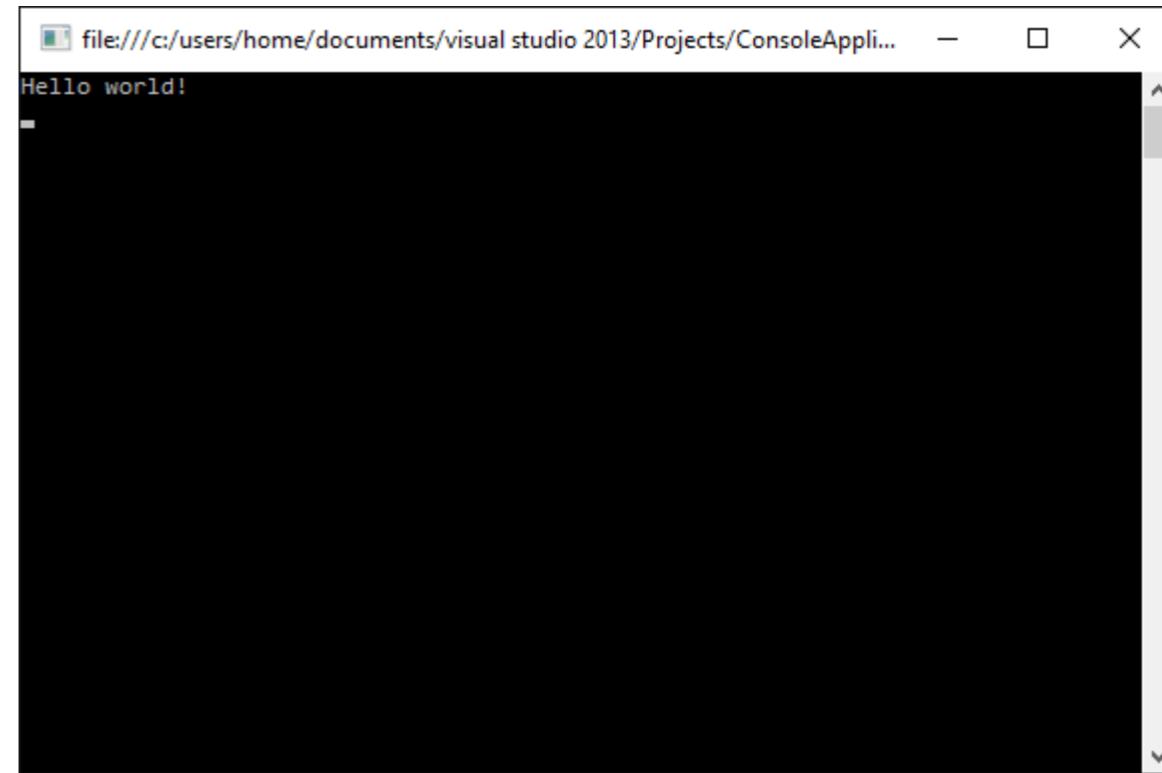
安装完成后，打开命令提示符或终端窗口。

1. 使用mkdir hello_world创建一个新目录，并使用cd切换到新创建的目录hello_world。

2. 使用dotnet new console创建一个新的控制台应用程序。
这将生成两个文件：

- **hello_world.csproj**

```
<Project Sdk="Microsoft.NET.Sdk">  
  
<PropertyGroup>  
<OutputType>Exe</OutputType>  
<TargetFramework>netcoreapp1.1</TargetFramework>  
  
</Project>  
  
◦ Program.cs  
  
using System;
```



```
file:///c:/users/home/documents/visual studio 2013/Projects/ConsoleAppl... - □ ×  
Hello world!
```

9. **Stop the program.** To close the program, just press any key on your keyboard. The Console.Read() we added was for this same purpose. Another way to close the program is by going to the menu where the Start button was, and clicking on the Stop button.

Section 1.3: Creating a new program using .NET Core

First install the [.NET Core SDK](#) by going through the installation instructions for the platform of your choice:

- [Windows](#)
- [OSX](#)
- [Linux](#)
- [Docker](#)

After the installation has completed, open a command prompt, or terminal window.

1. Create a new directory with mkdir hello_world and change into the newly created directory with cd hello_world.
2. Create a new console application with dotnet new console.
This will produce two files:

- **hello_world.csproj**

```
<Project Sdk="Microsoft.NET.Sdk">  
  
<PropertyGroup>  
<OutputType>Exe</OutputType>  
<TargetFramework>netcoreapp1.1</TargetFramework>  
</PropertyGroup>  
  
</Project>  
  
◦ Program.cs  
  
using System;
```

```

namespace hello_world
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}

```

3. 使用 dotnet restore 恢复所需的包。
4. 可选 使用 dotnet build 构建应用程序以进行调试，或使用 dotnet build -c Release 进行发布构建。dotnet run 还会运行编译器并抛出构建错误（如果有的话）。
5. 使用 dotnet run 以调试模式运行应用程序，或使用 dotnet run .\bin\Release\netcoreapp1.1\hello_world.dll 以发布模式运行。

命令提示符输出

```

Command Prompt

C:\dev>mkdir hello_world
C:\dev>cd hello_world
C:\dev\hello_world>dotnet new console
Content generation time: 75.7641 ms
The template "Console Application" created successfully.

C:\dev\hello_world>dotnet restore
Restoring packages for C:\dev\hello_world\hello_world.csproj...
Generating MSBuild file C:\dev\hello_world\obj\hello_world.csproj.nuget.g.props.
Generating MSBuild file C:\dev\hello_world\obj\hello_world.csproj.nuget.g.targets.
Writing lock file to disk. Path: C:\dev\hello_world\obj\project.assets.json
Restore completed in 3.35 sec for C:\dev\hello_world\hello_world.csproj.

NuGet Config files used:
  C:\Users\Ghost\AppData\Roaming\NuGet\NuGet.Config
  C:\Program Files (x86)\NuGet\Config\Microsoft.VisualStudio.Offline.config

Feeds used:
  https://api.nuget.org/v3/index.json
  C:\Program Files (x86)\Microsoft SDKs\NuGetPackages\

C:\dev\hello_world>dotnet build -c Release
Microsoft (R) Build Engine version 15.1.548.43366
Copyright (C) Microsoft Corporation. All rights reserved.

  hello_world -> C:\dev\hello_world\bin\Release\netcoreapp1.1\hello_world.dll

Build succeeded.
  0 Warning(s)
  0 Error(s)

Time Elapsed 00:00:03.58

C:\dev\hello_world>dotnet run .\bin\Release\netcoreapp1.1\hello_world.dll
Hello World!

C:\dev\hello_world>

```

```

namespace hello_world
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}

```

3. Restore the needed packages with dotnet restore.
4. *Optional* Build the application with dotnet build for Debug or dotnet build -c Release for Release. dotnet run will also run the compiler and throw build errors, if any are found.
5. Run the application with dotnet run for Debug or dotnet run .\bin\Release\netcoreapp1.1\hello_world.dll for Release.

Command Prompt output

```

Command Prompt

C:\dev>mkdir hello_world
C:\dev>cd hello_world
C:\dev\hello_world>dotnet new console
Content generation time: 75.7641 ms
The template "Console Application" created successfully.

C:\dev\hello_world>dotnet restore
Restoring packages for C:\dev\hello_world\hello_world.csproj...
Generating MSBuild file C:\dev\hello_world\obj\hello_world.csproj.nuget.g.props.
Generating MSBuild file C:\dev\hello_world\obj\hello_world.csproj.nuget.g.targets.
Writing lock file to disk. Path: C:\dev\hello_world\obj\project.assets.json
Restore completed in 3.35 sec for C:\dev\hello_world\hello_world.csproj.

NuGet Config files used:
  C:\Users\Ghost\AppData\Roaming\NuGet\NuGet.Config
  C:\Program Files (x86)\NuGet\Config\Microsoft.VisualStudio.Offline.config

Feeds used:
  https://api.nuget.org/v3/index.json
  C:\Program Files (x86)\Microsoft SDKs\NuGetPackages\

C:\dev\hello_world>dotnet build -c Release
Microsoft (R) Build Engine version 15.1.548.43366
Copyright (C) Microsoft Corporation. All rights reserved.

  hello_world -> C:\dev\hello_world\bin\Release\netcoreapp1.1\hello_world.dll

Build succeeded.
  0 Warning(s)
  0 Error(s)

Time Elapsed 00:00:03.58

C:\dev\hello_world>dotnet run .\bin\Release\netcoreapp1.1\hello_world.dll
Hello World!

C:\dev\hello_world>

```

第1.4节：使用Mono创建新程序

首先安装 [Mono](#), 按照其

安装部分中描述的适合您所选平台的安装说明进行操作。

Mono可用于Mac OS X、Windows和Linux。

安装完成后，创建一个文本文件，命名为 `HelloWorld.cs`，并将以下内容复制到其中：

```
public class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, world!");
        System.Console.WriteLine("Press any key to exit..");
        System.Console.Read();
    }
}
```

如果您使用的是Windows，请运行Mono安装中包含的Mono命令提示符，以确保设置了必要的环境变量。如果是在Mac或Linux上，请打开一个新的终端。

要编译新创建的文件，请在包含`HelloWorld.cs`的目录中运行以下命令：

```
mcs -out:HelloWorld.exe HelloWorld.cs
```

生成的`HelloWorld.exe`文件可以通过以下命令执行：

```
mono HelloWorld.exe
```

这将产生以下输出：

```
Hello, world!
按任意键退出..
```

第1.5节：使用LinqPad创建新查询

LinqPad是一个很棒的工具，允许你学习和测试.Net语言（C#、F#和VB.Net）的功能。

1. 安装[LinqPad](#)

2. 创建一个新的查询（**Ctrl** + **N**）

Section 1.4: Creating a new program using Mono

First install [Mono](#) by going through the install instructions for the platform of your choice as described in their [installation section](#).

Mono is available for Mac OS X, Windows and Linux.

After installation is done, create a text file, name it `HelloWorld.cs` and copy the following content into it:

```
public class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Hello, world!");
        System.Console.WriteLine("Press any key to exit..");
        System.Console.Read();
    }
}
```

If you are using Windows, run the Mono Command Prompt which is included in the Mono installation and ensures that the necessary environment variables are set. If on Mac or Linux, open a new terminal.

To compile the newly created file, run the following command in the directory containing `HelloWorld.cs`:

```
mcs -out:HelloWorld.exe HelloWorld.cs
```

The resulting `HelloWorld.exe` can then be executed with:

```
mono HelloWorld.exe
```

which will produce the output:

```
Hello, world!
Press any key to exit..
```

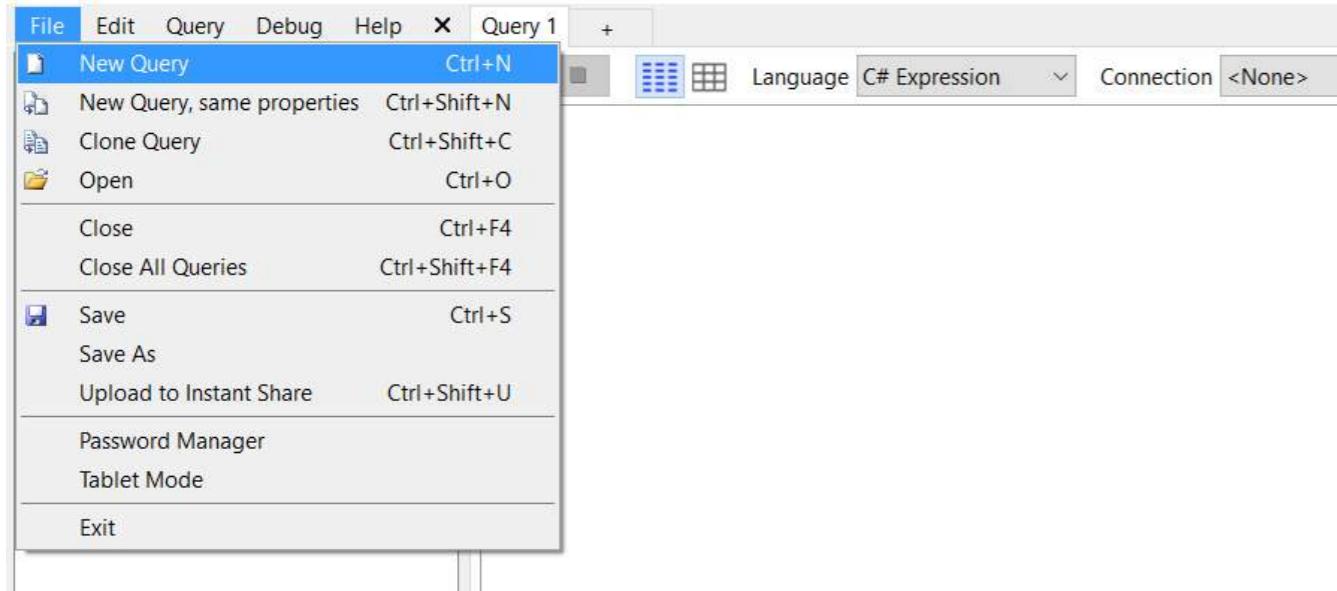
Section 1.5: Creating a new query using LinqPad

LinqPad is a great tool that allows you to learn and test features of .Net languages (C#, F# and VB.Net.)

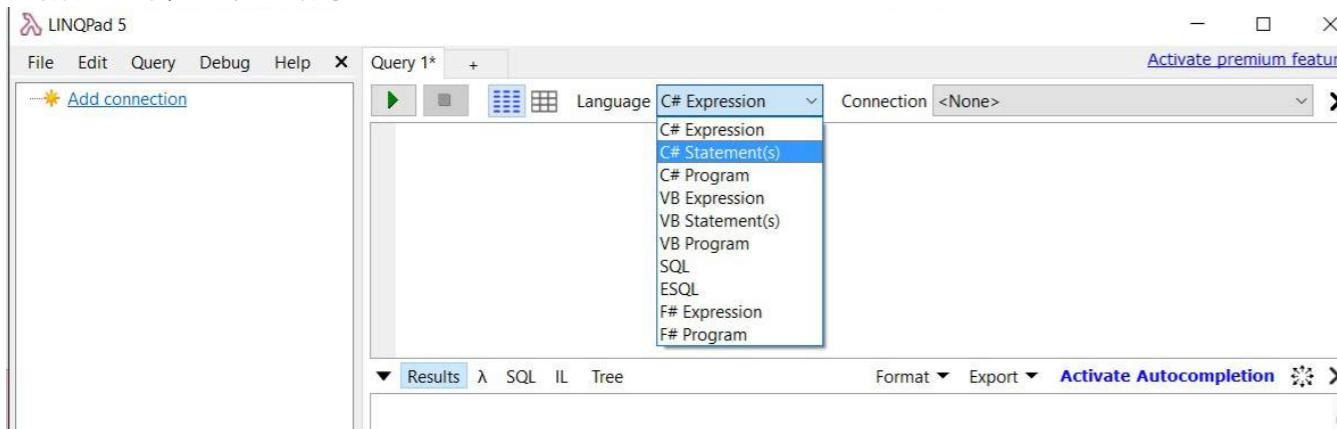
1. Install [LinqPad](#)

2. Create a new Query (**Ctrl** + **N**)

LINQPad 5



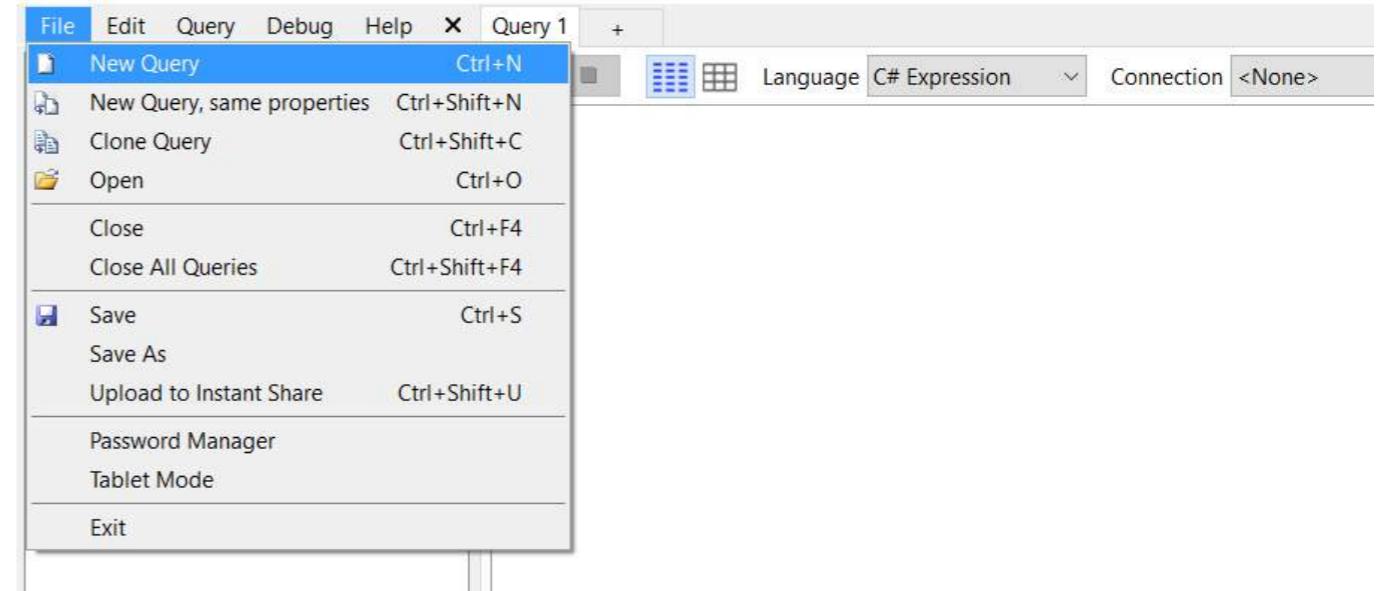
3. 在语言选项中，选择“C#语句”



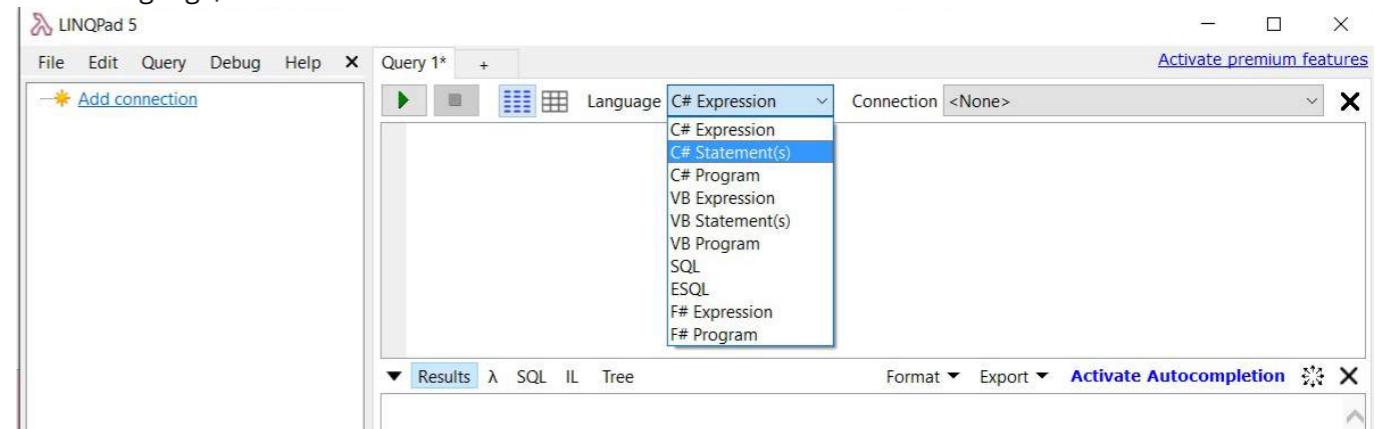
4. 输入以下代码并运行 (F5)

```
string hw = "Hello World";
hw.Dump(); //或 Console.WriteLine(hw);
```

LINQPad 5

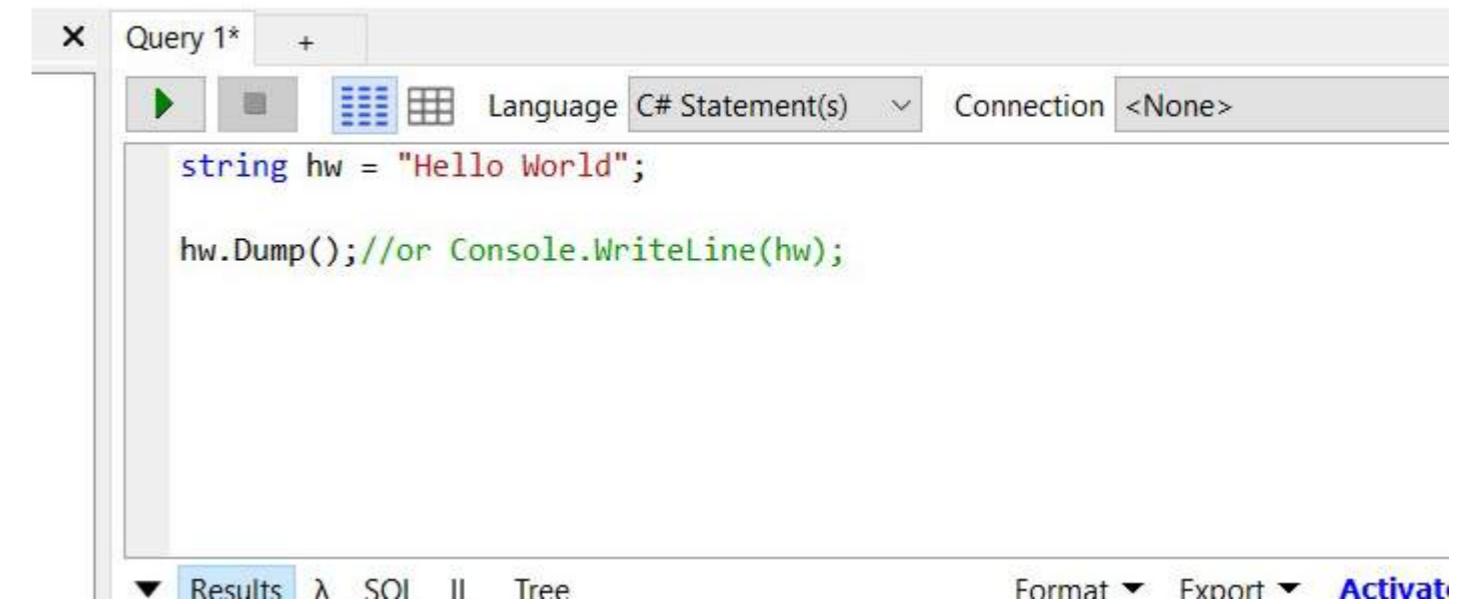
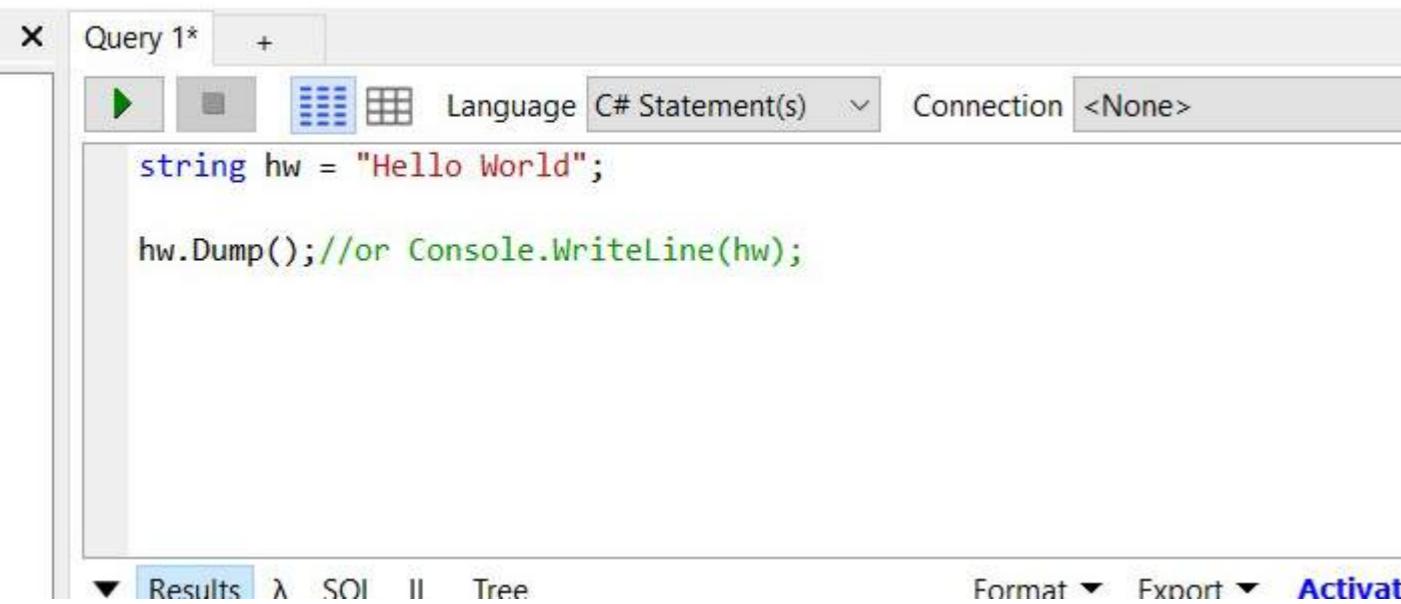


3. Under language, select "C# statements"



4. Type the following code and hit run (F5)

```
string hw = "Hello World";
hw.Dump(); //or Console.WriteLine(hw);
```



5. 你应该会在结果屏幕上看到“Hello World”的输出。

A screenshot of the LinqPad application window. The main pane shows a C# code block:

```
string hw = "Hello World";
hw.Dump(); //or Console.WriteLine(hw);
```

The results pane at the bottom displays the output: "Hello World". The "Results" tab is selected. The entire results pane is highlighted with a green box.

6. 现在你已经创建了第一个 .Net 程序，去通过“Samples”浏览器查看 LinqPad 中包含的示例吧。这里有许多很棒的例子，可以向你展示 .Net 语言的许多不同功能。

A screenshot of the LinqPad application window. On the left, there is a sidebar titled "My Queries" which contains a "Samples" section with several items listed: "LINQPad 5 minute induction", "C# 6.0 in a Nutshell", "F# Tutorial", and "Download/import more samples". The main pane shows the same C# code as the previous screenshot. The results pane at the bottom displays the output: "Hello World". The "Results" tab is selected. The entire results pane is highlighted with a green box.

注意事项：

1. 如果你点击“IL”，可以查看你的 .net 代码生成的 IL 代码。这是一个很好的学习工具。

5. You should see "Hello World" printed out in the results screen.

A screenshot of the LinqPad application window. On the left, there is a sidebar titled "My Queries" which contains a "Samples" section with several items listed: "LINQPad 5 minute induction", "C# 6.0 in a Nutshell", "F# Tutorial", and "Download/import more samples". The main pane shows the same C# code as the previous screenshots. The results pane at the bottom displays the output: "Hello World". The "Results" tab is selected. The entire results pane is highlighted with a green box.

6. Now that you have created your first .Net program, go and check out the samples included in LinqPad via the "Samples" browser. There are many great examples that will show you many different features of the .Net languages.

A screenshot of the LinqPad application window. On the left, there is a sidebar titled "My Queries" which contains a "Samples" section with several items listed: "LINQPad 5 minute induction", "C# 6.0 in a Nutshell", "F# Tutorial", and "Download/import more samples". The main pane shows the same C# code as the previous screenshots. The results pane at the bottom displays the output: "Hello World". The "Results" tab is selected. The entire results pane is highlighted with a green box.

Notes:

1. If you click on "IL", you can inspect the IL code that your .net code generates. This is a great learning tool.

LINQPad 5 interface showing the results of the following C# code in the IL tab:

```
string hw = "Hello World";
hw.Dump(); //or Console.WriteLine(hw);
```

The results pane shows the generated IL code:

```
IL_0000: nop
IL_0001: ldstr    "Hello World"
IL_0006: stloc.0  // hw
IL_0007: ldloc.0  // hw
IL_0008: call     LINQPad.Extensions.Dump
IL_000D: pop
IL_000E: ret
```

2. 使用LINQ to SQL或Linq to Entities时，你可以查看生成的SQL，这也是学习 LINQ 的另一种好方法。另一种学习 LINQ 的好方法。

第1.6节：使用 Xamarin Studio 创建新项目

1. [下载并安装Xamarin Studio Community。](#)
2. 打开 Xamarin Studio。
3. 点击文件→新建→解决方案。

LINQPad 5 interface showing the results of the following C# code in the IL tab:

```
string hw = "Hello World";
hw.Dump(); //or Console.WriteLine(hw);
```

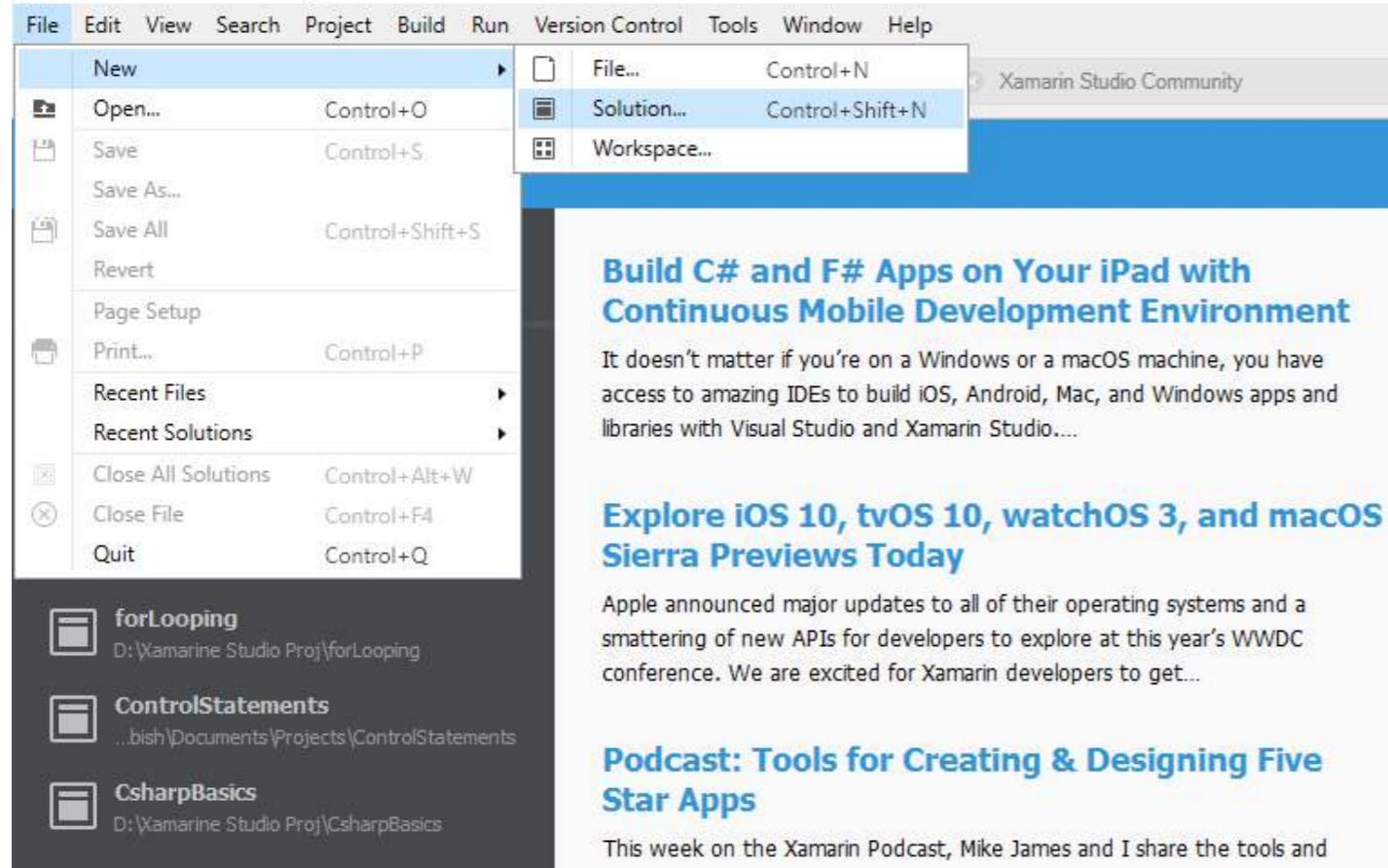
The results pane shows the generated IL code:

```
IL_0000: nop
IL_0001: ldstr    "Hello World"
IL_0006: stloc.0  // hw
IL_0007: ldloc.0  // hw
IL_0008: call     LINQPad.Extensions.Dump
IL_000D: pop
IL_000E: ret
```

2. When using LINQ to SQL or Linq to Entities you can inspect the SQL that's being generated which is another great way to learn about LINQ.

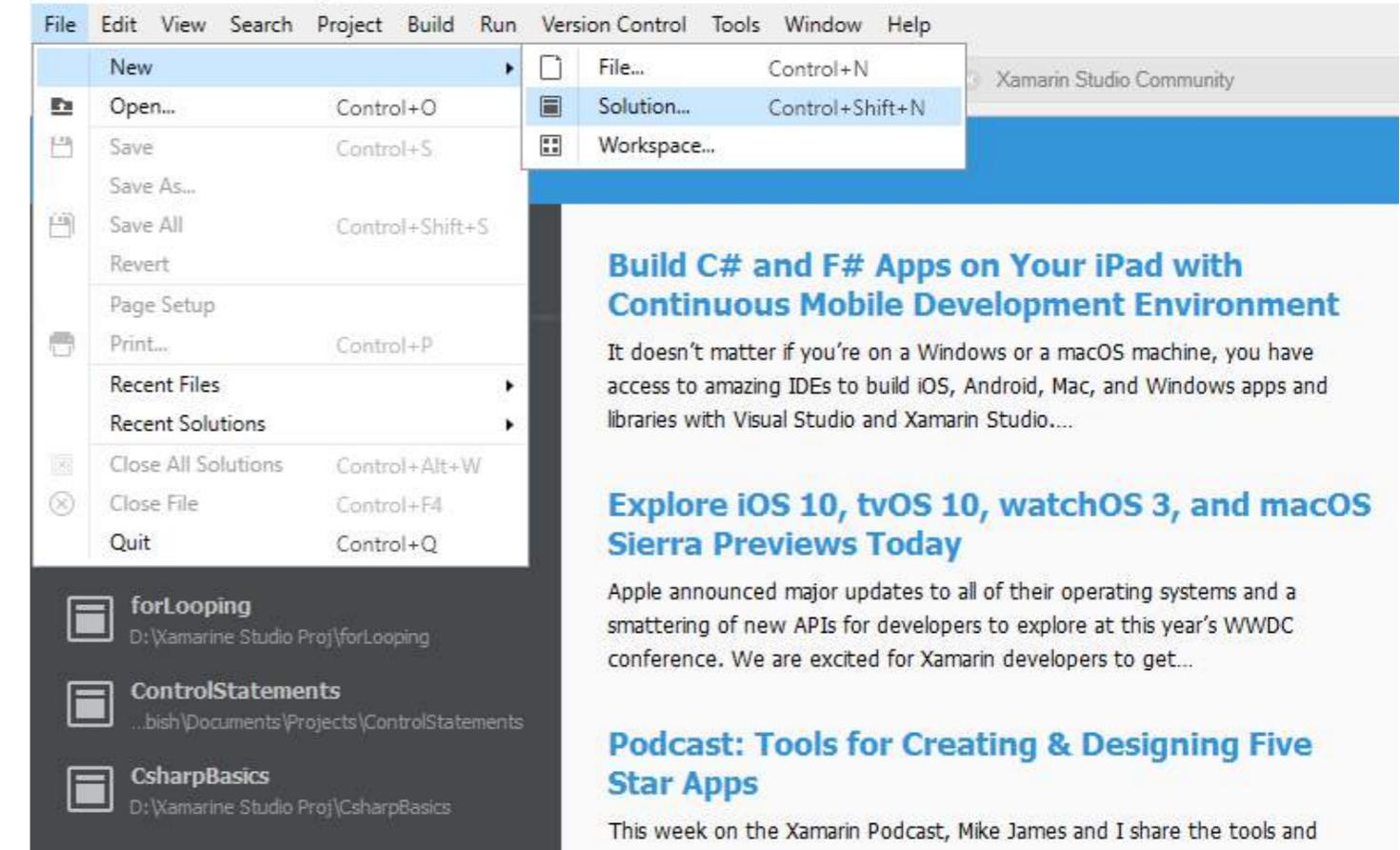
Section 1.6: Creating a new project using Xamarin Studio

1. Download and install [Xamarin Studio Community](#).
2. Open Xamarin Studio.
3. Click File → New → Solution.



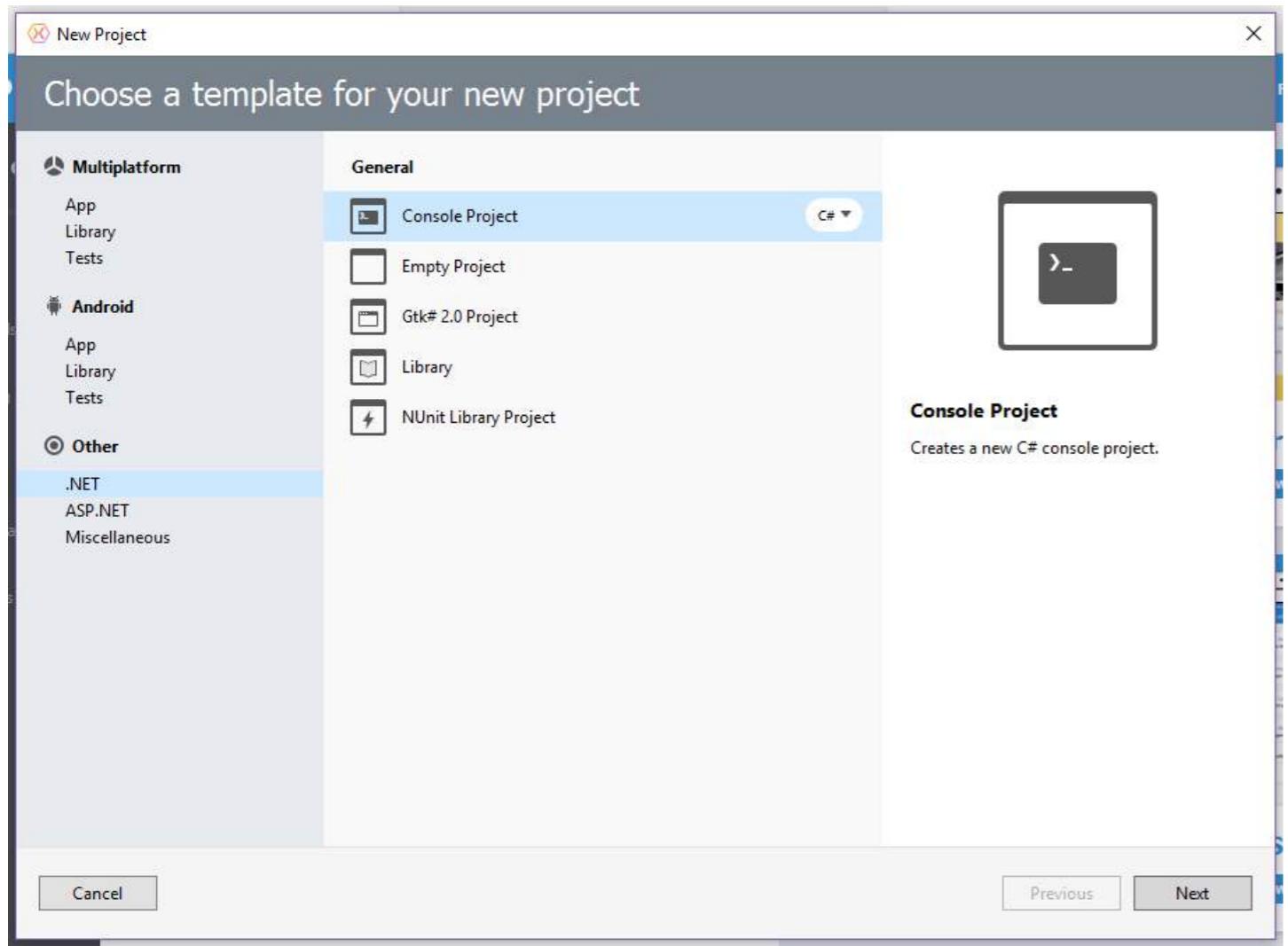
4. 点击 .NET → 控制台项目 并选择 C#。

5. 点击 **下一步** 继续。

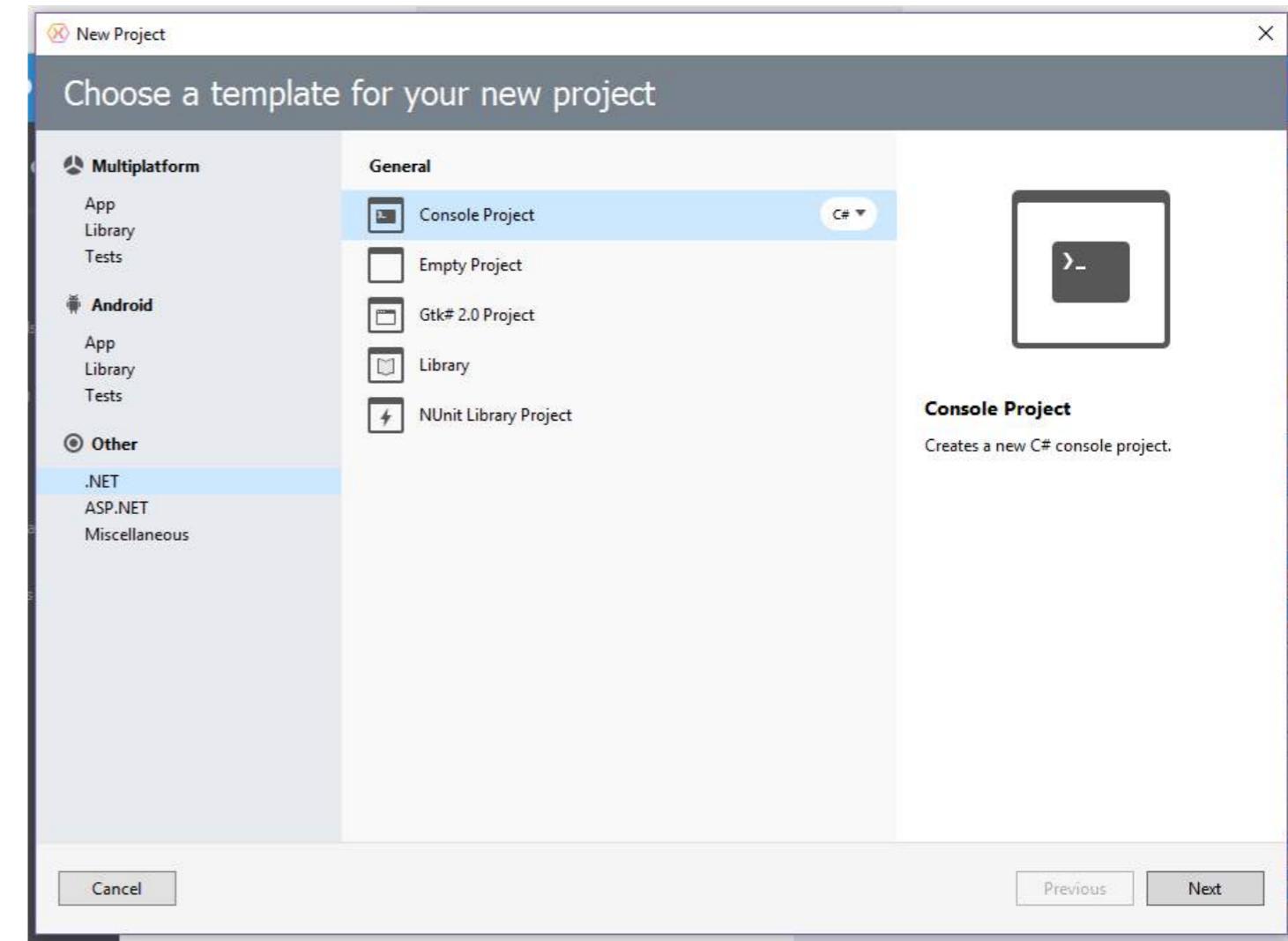


4. Click **.NET → Console Project** and choose **C#**.

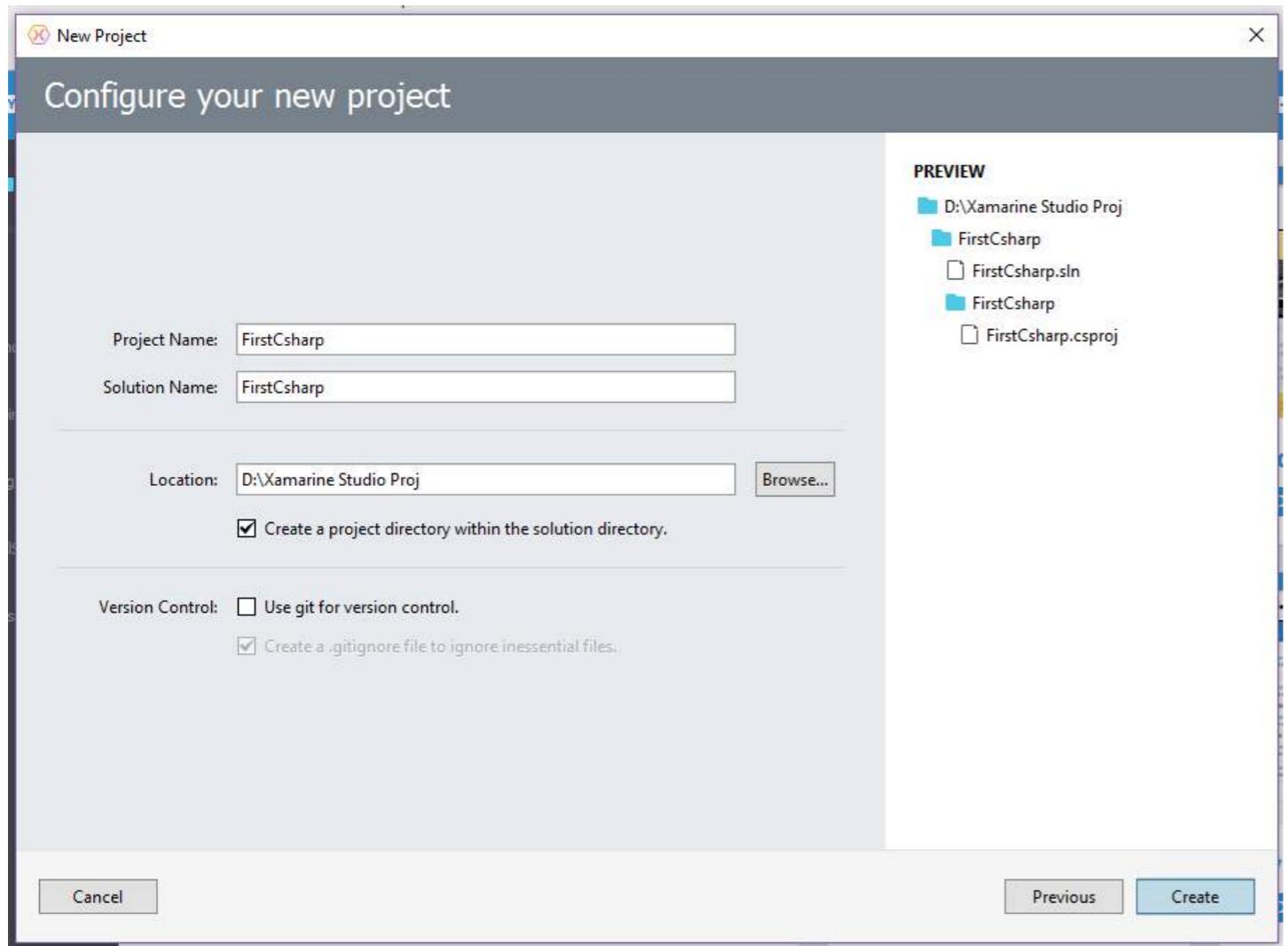
5. Click **Next** to proceed.



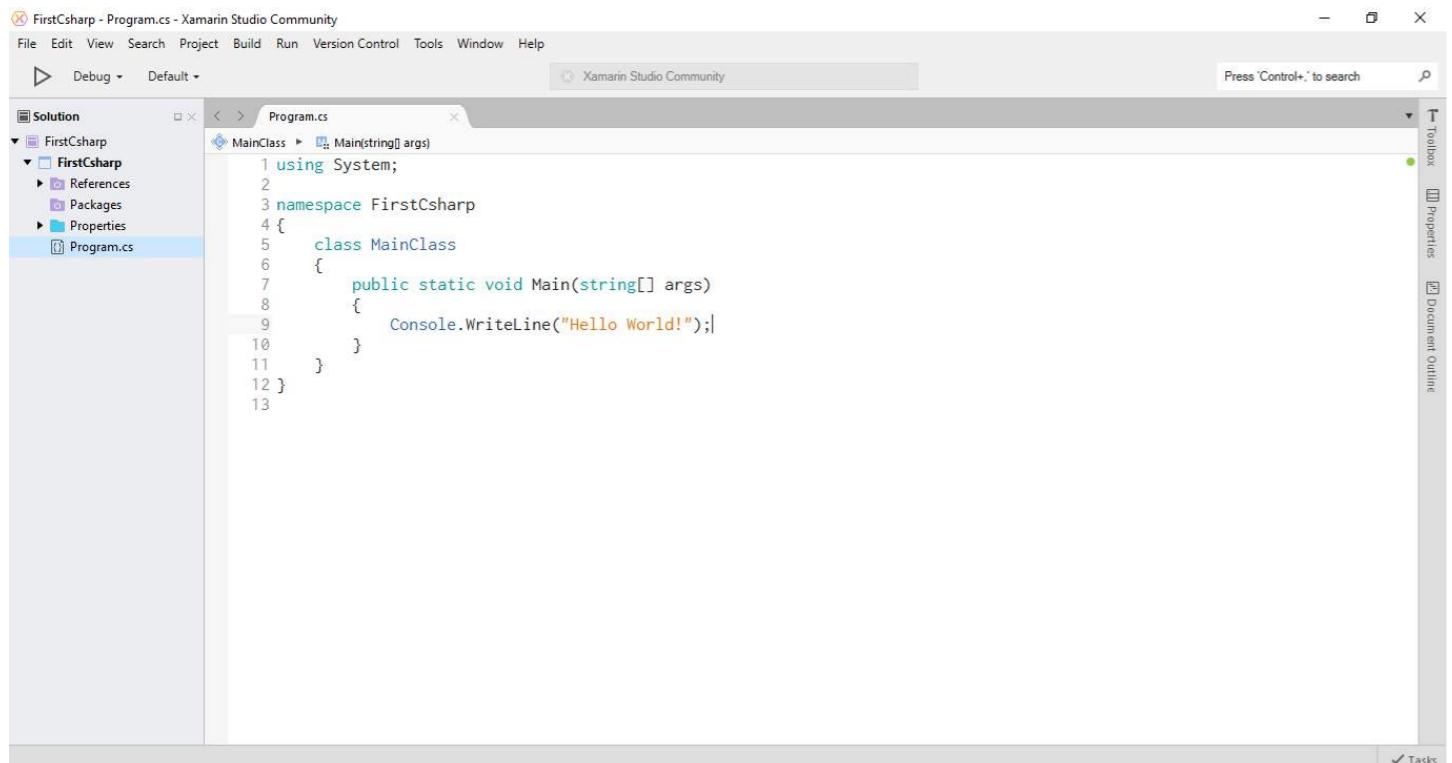
6. 输入项目名称并 选择一个保存位置, 然后点击 .



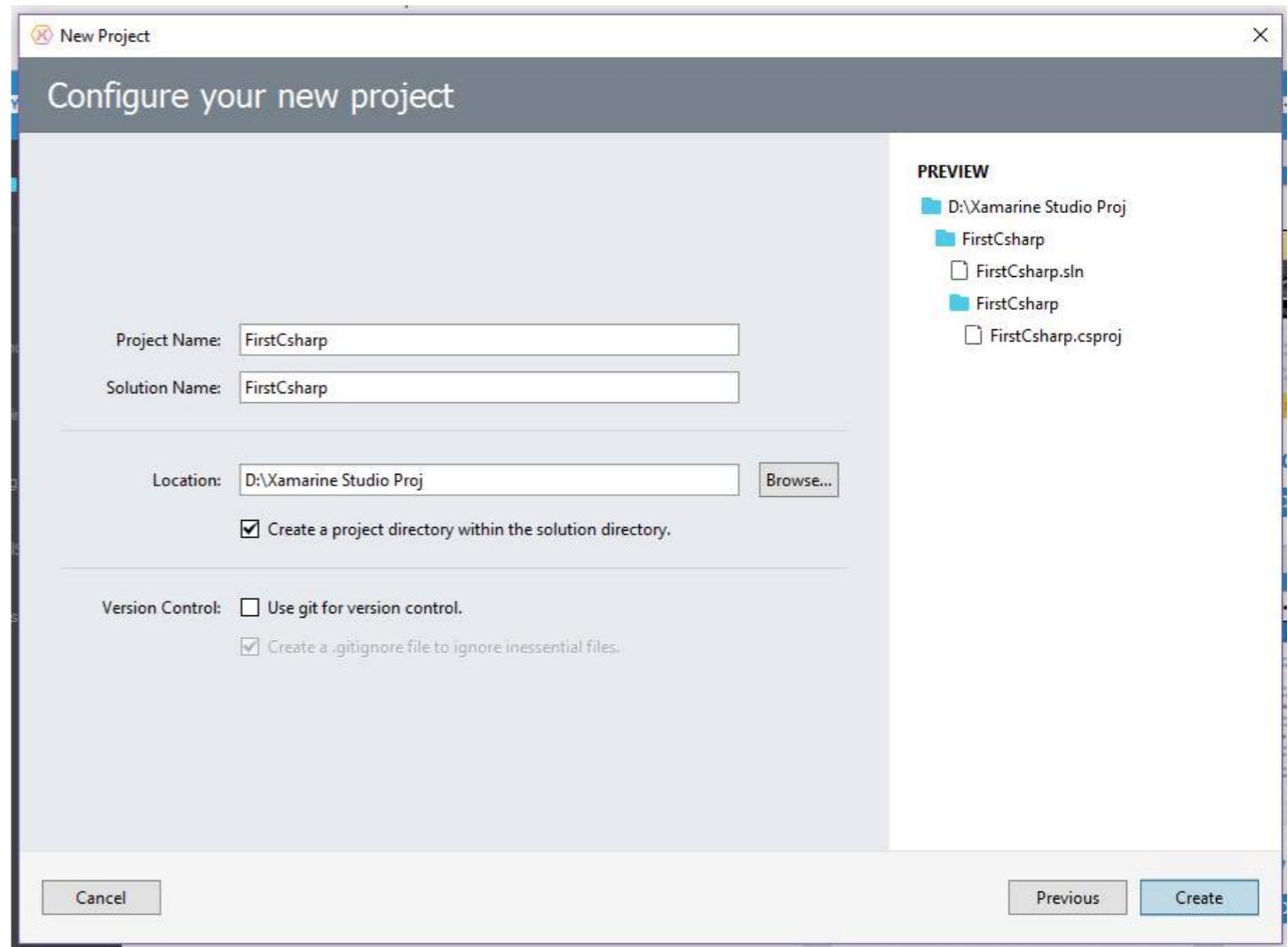
6. Enter the **Project Name** and for a **Location** to Save and then click .



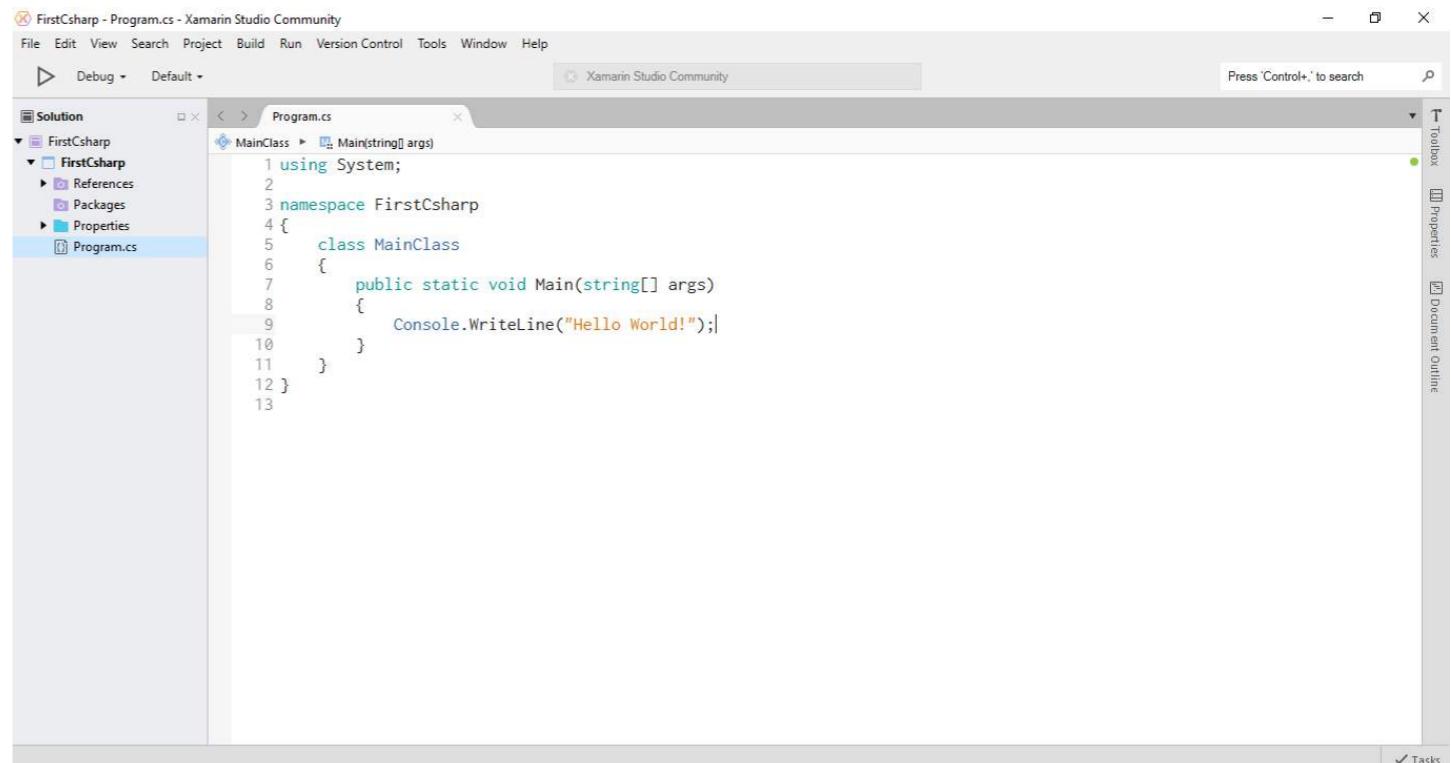
7. 新创建的项目将类似于：



8. 这是文本编辑器中的代码：



7. The newly created project will look similar to:



8. This is the code in the Text Editor:

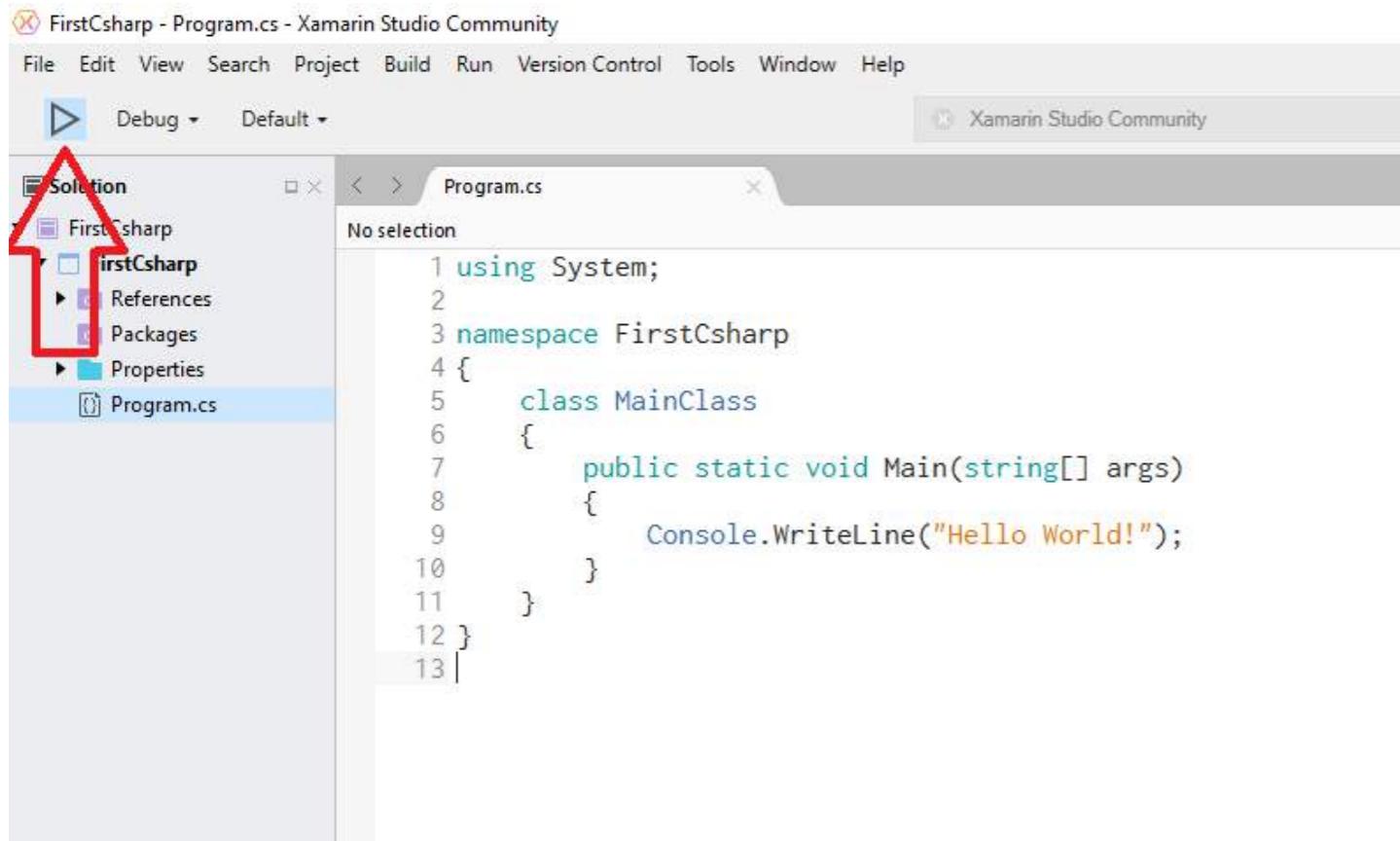
```

using System;

namespace FirstCsharp
{
    public class MainClass
    {
        public static void Main(string[] args)
        {
Console.WriteLine("Hello World!");
            Console.ReadLine();
        }
    }
}

```

9. 运行代码，按下 **F5** 或点击下方所示的播放按钮：



10. 以下是输出结果：

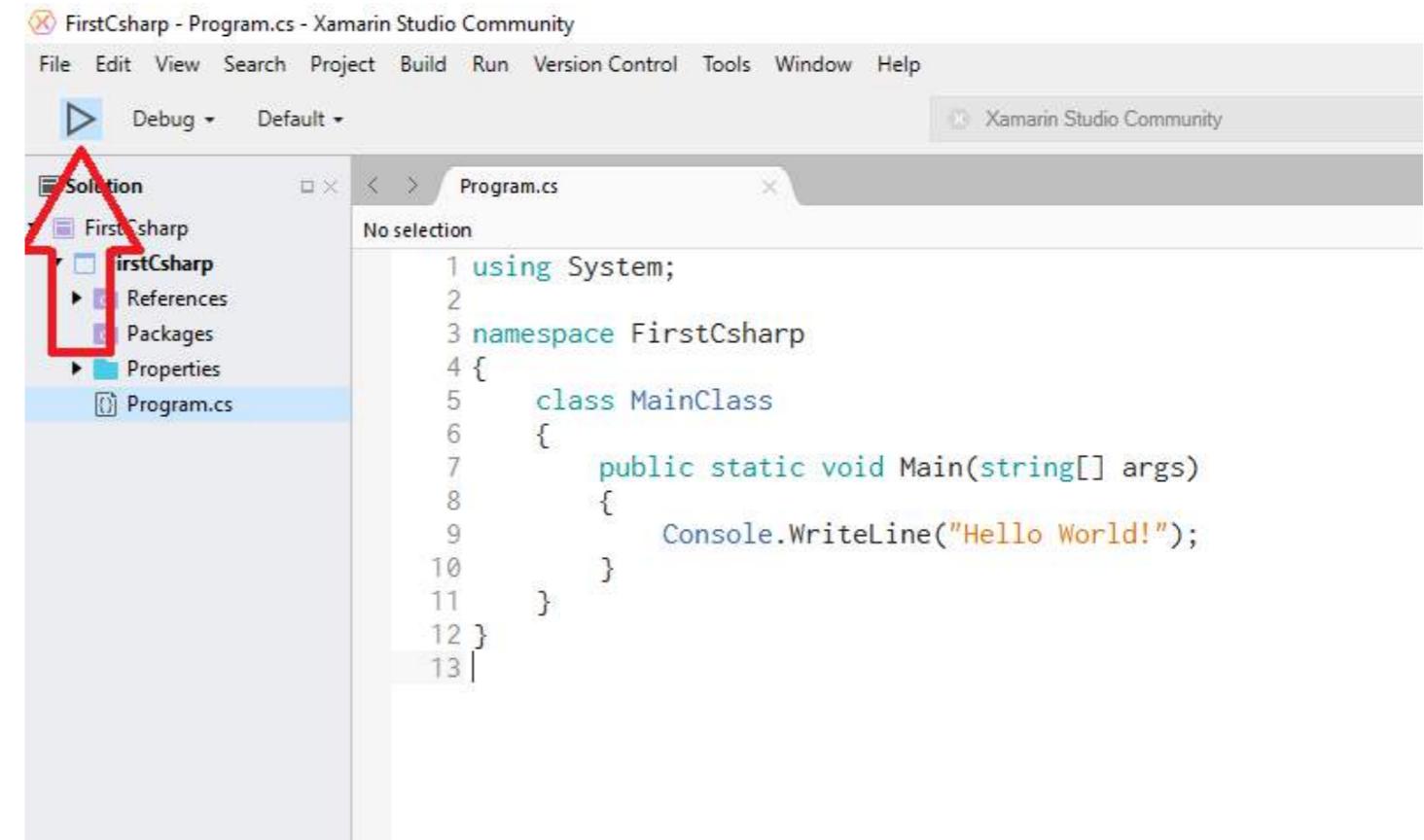
```

using System;

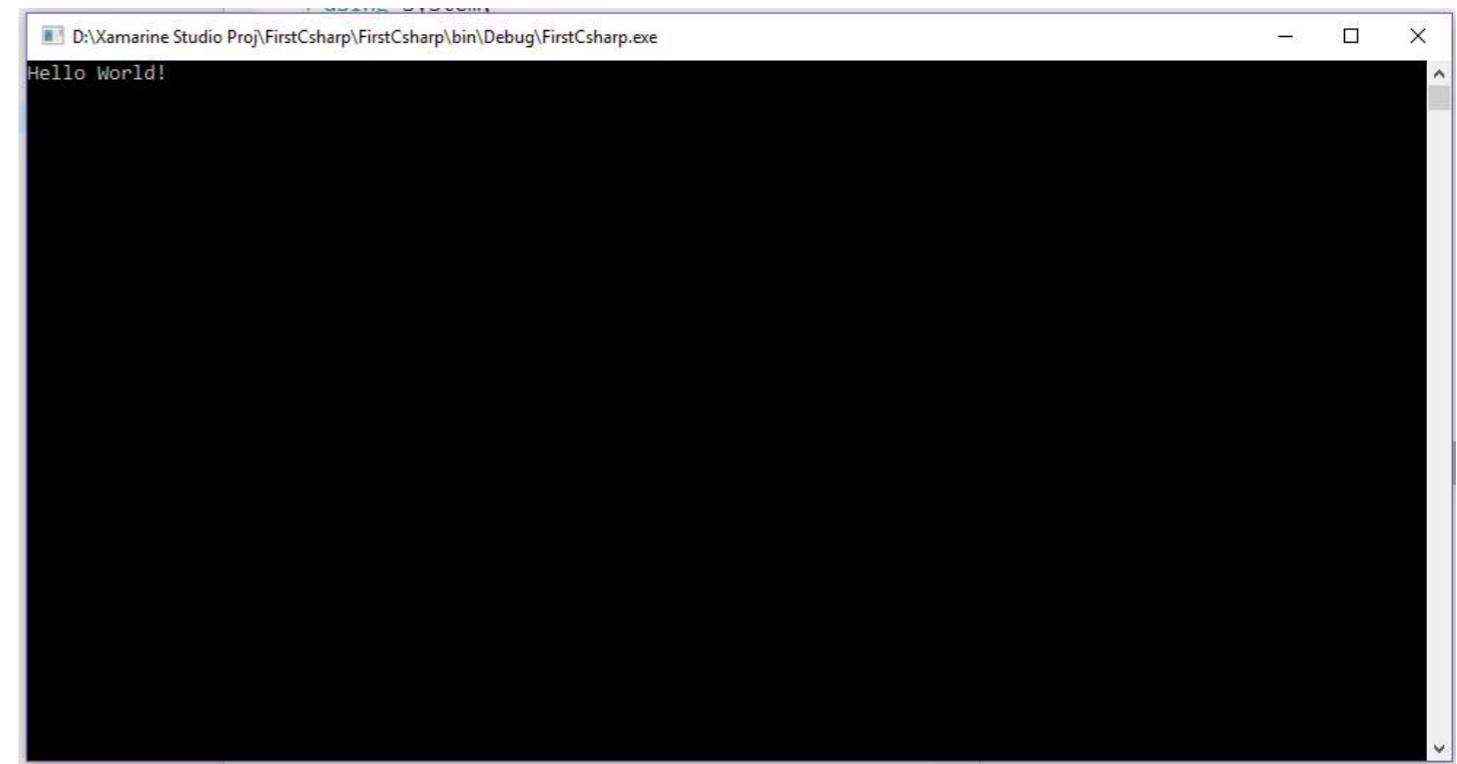
namespace FirstCsharp
{
    public class MainClass
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            Console.ReadLine();
        }
    }
}

```

9. To run the code, press **F5** or click the **Play Button** as shown below:



10. Following is the Output:



第二章：字面量

第2.1节：无符号整数字面量

uint 字面量通过使用后缀 U 或 u，或者使用位于 uint 范围内的整数值来定义：

```
uint ui = 5U;
```

第2.2节：int 字面量

int 字面量通过简单使用位于 int 范围内的整数值来定义：

```
int i = 5;
```

第2.3节：sbyte 字面量

sbyte 类型没有字面量后缀。整数字面量会隐式地从 int 转换：

```
sbyte sb = 127;
```

第2.4节：decimal 字面量

decimal 字面量通过在实数后使用后缀 M 或 m 来定义：

```
decimal m = 30.5M;
```

第2.5节：双精度字面量

双精度字面量通过使用后缀D或d，或使用实数来定义：

```
double d = 30.5D;
```

第2.6节：单精度字面量

单精度字面量通过使用后缀F或f，或使用实数来定义：

```
float f = 30.5F;
```

第2.7节：长整数字面量

长整数字面量通过使用后缀L或l，或使用long范围内的整数值来定义：

```
long l = 5L;
```

第2.8节：无符号长整数字面量

ulong 字面量通过使用后缀 UL、ul、Ul、uL、LU、lu、Lu 或 IU 定义，或者通过使用位于 ulong 范围内的整数值定义：

```
ulong ul = 5UL;
```

Chapter 2: Literals

Section 2.1: uint literals

uint literals are defined by using the suffix U or u, or by using an integral values within the range of uint:

```
uint ui = 5U;
```

Section 2.2: int literals

int literals are defined by simply using integral values within the range of int:

```
int i = 5;
```

Section 2.3: sbyte literals

sbyte type has no literal suffix. Integer literals are implicitly converted from int:

```
sbyte sb = 127;
```

Section 2.4: decimal literals

decimal literals are defined by using the suffix M or m on a real number:

```
decimal m = 30.5M;
```

Section 2.5: double literals

double literals are defined by using the suffix D or d, or by using a real number:

```
double d = 30.5D;
```

Section 2.6: float literals

float literals are defined by using the suffix F or f, or by using a real number:

```
float f = 30.5F;
```

Section 2.7: long literals

long literals are defined by using the suffix L or l, or by using an integral values within the range of long:

```
long l = 5L;
```

Section 2.8: ulong literal

ulong literals are defined by using the suffix UL, ul, Ul, uL, LU, lu, Lu, or IU, or by using an integral values within the range of ulong:

```
ulong ul = 5UL;
```

第2.9节：字符串字面量

string 字面量通过用双引号 " 包裹值来定义：

```
string s = "hello, this is a string literal";
```

字符串字面量可以包含转义序列。参见字符串转义序列此外，C# 支持逐字字

符串字面量（参见逐字字符串）。这些通过用双引号 " 包裹值，并在前面加上 @ 来定义。逐字字符串字面量中忽略转义序列，且包含所有空白字符：

```
string s = @"路径是：  
C:\Windows\System32";  
//字符串中包含反斜杠和换行符
```

第2.10节：字符字面量

字符字面量通过用单引号'包裹值来定义：

```
char c = 'h';
```

字符字面量可以包含转义序列。参见字符串转义序列

字符字面量必须恰好包含一个字符（在所有转义序列被计算后）。空的字符字面量无效。默认字符（由default(char)或new char()返回）是 '\0'，即 NULL字符（不要与null字面量和null引用混淆）。

第2.11节：字节字面量

byte类型没有字面量后缀。整数字面量会隐式转换自int:

```
byte b = 127;
```

第2.12节：short字面量

short类型没有字面量。整数字面量会隐式转换自int:

```
short s = 127;
```

第2.13节：ushort字面量

ushort类型没有字面量后缀。整数字面量会被隐式转换为int类型：

```
ushort us = 127;
```

第2.14节：布尔字面量

布尔字面量是true或false；

```
bool b = true;
```

Section 2.9: string literals

string literals are defined by wrapping the value with double-quotes ":

```
string s = "hello, this is a string literal";
```

String literals may contain escape sequences. See String Escape Sequences

Additionally, C# supports verbatim string literals (See Verbatim Strings). These are defined by wrapping the value with double-quotes "，and prepending it with @. Escape sequences are ignored in verbatim string literals, and all whitespace characters are included:

```
string s = @"The path is:  
C:\Windows\System32";  
//The backslashes and newline are included in the string
```

Section 2.10: char literals

char literals are defined by wrapping the value with single-quotes ':

```
char c = 'h';
```

Character literals may contain escape sequences. See String Escape Sequences

A character literal must be exactly one character long (after all escape sequences have been evaluated). Empty character literals are not valid. The default character (returned by default(char) or new char()) is '\0'，or the NULL character (not to be confused with the null literal and null references).

Section 2.11: byte literals

byte type has no literal suffix. Integer literals are implicitly converted from int:

```
byte b = 127;
```

Section 2.12: short literal

short type has no literal. Integer literals are implicitly converted from int:

```
short s = 127;
```

Section 2.13: ushort literal

ushort type has no literal suffix. Integer literals are implicitly converted from int:

```
ushort us = 127;
```

Section 2.14: bool literals

bool literals are either true or false;

```
bool b = true;
```

第3章：运算符

参数	详情op
operatorSymbol	被重载的运算符，例如 +、-、/、*
	重载运算符返回的类型。
operand1	执行操作时使用的第一操作数。
操作数2	在执行二元操作时使用的第二个操作数。
语句	执行操作前可选的代码，用于返回结果。

在C#中，运算符是应用于表达式或语句中一个或多个操作数的程序元素。

接受一个操作数的运算符，如递增运算符（++）或new，称为一元运算符。

接受两个操作数的运算符，如算术运算符（+、-、*、/），称为二元运算符。一个运算符，条件运算符（?：），接受三个操作数，是C#中唯一的三元运算符。

第3.1节：可重载运算符

C#允许用户自定义类型通过定义使用operator关键字的静态成员函数来重载运算符。

下面的示例演示了+运算符的实现。

如果我们有一个表示复数的Complex类：

```
public struct 复数
{
    public double 实部 { get; set; }
    public double 虚部 { get; set; }
}
```

我们想为这个类添加使用+运算符的选项。即：

```
复数 a = new 复数() { 实部 = 1, 虚部 = 2 };
复数 b = new 复数() { 实部 = 4, 虚部 = 8 };
复数 c = a + b;
```

我们需要为该类重载+运算符。这是通过静态函数和operator关键字完成的：

```
public static 复数 operator +(复数 c1, 复数 c2)
{
    return new 复数
    {
        实部 = c1.实部 + c2.实部,
        虚部 = c1.虚部 + c2.虚部
    };
}
```

运算符如+、-、*、/都可以被重载。这也包括不返回相同类型的运算符（例如，==和!=可以被重载，尽管它们返回布尔值）。下面关于成对重载的规则也适用。

比较运算符必须成对重载（例如，如果重载了<，则也需要重载>）。

可重载运算符的完整列表（以及不可重载运算符和对某些可重载运算符施加的限制）可见于MSDN - 可重载运算符（C# 编程指南）。

Chapter 3: Operators

Parameter	Details
operatorSymbol	The operator being overloaded, e.g. +, -, /, *
OperandType	The type that will be returned by the overloaded operator.
operand1	The first operand to be used in performing the operation.
operand2	The second operand to be used in performing the operation, when doing binary operations.
statements	Optional code needed to perform the operation before returning the result.

In C#, an [operator](#) is a program element that is applied to one or more operands in an expression or statement. Operators that take one operand, such as the increment operator (++) or new, are referred to as unary operators. Operators that take two operands, such as arithmetic operators (+,-,*,/), are referred to as binary operators. One operator, the conditional operator (?:), takes three operands and is the sole ternary operator in C#.

Section 3.1: Overloadable Operators

C# allows user-defined types to overload operators by defining static member functions using the [operator](#) keyword.

The following example illustrates an implementation of the + operator.

If we have a Complex class which represents a complex number:

```
public struct Complex
{
    public double Real { get; set; }
    public double Imaginary { get; set; }
}
```

And we want to add the option to use the + operator for this class. i.e.:

```
Complex a = new Complex() { Real = 1, Imaginary = 2 };
Complex b = new Complex() { Real = 4, Imaginary = 8 };
Complex c = a + b;
```

We will need to overload the + operator for the class. This is done using a static function and the [operator](#) keyword:

```
public static Complex operator +(Complex c1, Complex c2)
{
    return new Complex
    {
        Real = c1.Real + c2.Real,
        Imaginary = c1.Imaginary + c2.Imaginary
    };
}
```

Operators such as +, -, *, / can all be overloaded. This also includes Operators that don't return the same type (for example, == and != can be overloaded, despite returning booleans) The rule below relating to pairs is also enforced here.

Comparison operators have to be overloaded in pairs (e.g. if < is overloaded, > also needs to be overloaded).

A full list of overloadable operators (as well as non-overloadable operators and the restrictions placed on some overloadable operators) can be seen at [MSDN - Overloadable Operators \(C# Programming Guide\)](#).

运算符的重载是在 C# 7.0 的模式匹配机制中引入的。详情请参见模式匹配

给定如下定义的类型Cartesian

```
public class Cartesian
{
    public int X { get; }
    public int Y { get; }
}
```

例如，可以为Polar坐标定义一个可重载的operator

```
public static class Polar
{
    public static bool operator is(Cartesian c, out double R, out double Theta)
    {
        R = Math.Sqrt(c.X*c.X + c.Y*c.Y);
        Theta = Math.Atan2(c.Y, c.X);
        return c.X != 0 || c.Y != 0;
    }
}
```

可以这样使用

```
var c = 笛卡尔坐标(3, 4);
if (c 是 极坐标(var R, *))
{
    Console.WriteLine(R);
}
```

(示例取自Roslyn模式匹配文档)

第3.2节：重载相等运算符

仅重载相等运算符是不够的。在不同情况下，以下所有方法都可能被调用：

1. object.Equals 和 object.GetHashCode
2. IEquatable<T>.Equals (可选，允许避免装箱)
3. operator == 和 operator != (可选，允许使用运算符)

重写 Equals 时，必须同时重写 GetHashCode。实现 Equals 时，有许多特殊情况：与不同类型的对象比较、与自身比较等。

当未重写 Equals 方法时，Equals 和 == 运算符在类和结构体中的行为不同。对于类，仅比较引用；对于结构体，通过反射比较属性值，这可能会对性能产生负面影响。除非重写，否则不能使用 == 来比较结构体。

通常，等于操作必须遵守以下规则：

- 不得抛出异常。
- 自反性：A总是等于A（在某些系统中，NULL值可能不满足此条件）。
- 传递性：如果A等于B，且B等于C，则A等于C。
- 如果A等于B，则A和B具有相同的哈希码。

overloading of `operator is` was introduced with the pattern matching mechanism of C# 7.0. For details see Pattern Matching

Given a type `Cartesian` defined as follows

```
public class Cartesian
{
    public int X { get; }
    public int Y { get; }
}
```

An overloadable `operator is` could e.g. be defined for Polar coordinates

```
public static class Polar
{
    public static bool operator is(Cartesian c, out double R, out double Theta)
    {
        R = Math.Sqrt(c.X*c.X + c.Y*c.Y);
        Theta = Math.Atan2(c.Y, c.X);
        return c.X != 0 || c.Y != 0;
    }
}
```

which can be used like this

```
var c = Cartesian(3, 4);
if (c is Polar(var R, *))
{
    Console.WriteLine(R);
}
```

(The example is taken from the [Roslyn Pattern Matching Documentation](#))

Section 3.2: Overloading equality operators

Overloading just equality operators is not enough. Under different circumstances, all of the following can be called:

1. `object.Equals` and `object.GetHashCode`
2. `IEquatable<T>.Equals` (optional, allows avoiding boxing)
3. `operator ==` and `operator !=` (optional, allows using operators)

When overriding Equals, GetHashCode must also be overridden. When implementing Equals, there are many special cases: comparing to objects of a different type, comparing to self etc.

When NOT overridden Equals method and == operator behave differently for classes and structs. For classes just references are compared, and for structs values of properties are compared via reflection what can negatively affect performance. == can not be used for comparing structs unless it is overridden.

Generally equality operation must obey the following rules:

- Must not *throw exceptions*.
- Reflexivity: A always equals A (may not be true for `NULL` values in some systems).
- Transitivity: if A equals B, and B equals C, then A equals C.
- If A equals B, then A and B have equal hash codes.

- 继承树独立性：如果B和C是从Class1继承的Class2的实例：
Class1.Equals(A,B)必须始终返回与调用Class2.Equals(A,B)相同的值。

```
class Student : IEquatable<Student>
{
    public string Name { get; set; } = "";

    public bool Equals(Student other)
    {
        if (ReferenceEquals(other, null)) return false;
        if (ReferenceEquals(other, this)) return true;
        return string.Equals(Name, other.Name);
    }

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;

        return Equals(obj as Student);
    }

    public override int GetHashCode()
    {
        return Name?.GetHashCode() ?? 0;
    }

    public static bool operator ==(Student left, Student right)
    {
        return Equals(left, right);
    }

    public static bool operator !=(Student left, Student right)
    {
        return !Equals(left, right);
    }
}
```

第3.3节：关系运算符

Equals

检查所提供的操作数（参数）是否相等

```
"a" == "b"      // 返回 false.
"a" == "a"      // 返回 true.
1 == 0          // 返回 false.
1 == 1          // 返回 true.
false == true   // 返回 false.
false == false  // 返回 true.
```

与 Java 不同，等号比较运算符可以直接用于字符串。

如果存在从一种类型到另一种类型的隐式转换，等号比较运算符也可以用于不同类型的操作数。如果没有合适的隐式转换，可以调用显式转换或使用方法转换为兼容类型。

```
1 == 1.0          // 返回 true, 因为存在从 int 到 double 的隐式转换。
new Object() == 1.0 // 无法编译。
```

- Inheritance tree independence: if B and C are instances of Class2 inherited from Class1:
Class1.Equals(A,B) must always return the same value as the call to Class2.Equals(A,B).

```
class Student : IEquatable<Student>
{
    public string Name { get; set; } = "";

    public bool Equals(Student other)
    {
        if (ReferenceEquals(other, null)) return false;
        if (ReferenceEquals(other, this)) return true;
        return string.Equals(Name, other.Name);
    }

    public override bool Equals(object obj)
    {
        if (ReferenceEquals(null, obj)) return false;
        if (ReferenceEquals(this, obj)) return true;

        return Equals(obj as Student);
    }

    public override int GetHashCode()
    {
        return Name?.GetHashCode() ?? 0;
    }

    public static bool operator ==(Student left, Student right)
    {
        return Equals(left, right);
    }

    public static bool operator !=(Student left, Student right)
    {
        return !Equals(left, right);
    }
}
```

Section 3.3: Relational Operators

Equals

Checks whether the supplied operands (arguments) are equal

```
"a" == "b"      // Returns false.
"a" == "a"      // Returns true.
1 == 0          // Returns false.
1 == 1          // Returns true.
false == true   // Returns false.
false == false  // Returns true.
```

Unlike Java, the equality comparison operator works natively with strings.

The equality comparison operator will work with operands of differing types if an implicit cast exists from one to the other. If no suitable implicit cast exists, you may call an explicit cast or use a method to convert to a compatible type.

```
1 == 1.0          // Returns true because there is an implicit cast from int to double.
new Object() == 1.0 // Will not compile.
```

`MyStruct.AsInt() == 1` // 调用 `MyStruct` 的 `AsInt()` 方法并将结果的整数与 1 进行比较。

与 Visual Basic.NET 不同，等于比较运算符与赋值运算符不同。

```
var x = new Object();
var y = new Object();
x == y // 返回 false, 操作数 (此处为对象) 引用不同。
x == x // 返回 true, 两个操作数引用相同。
```

不要与赋值运算符 (`=`) 混淆。

对于值类型，如果两个操作数的值相等，运算符返回 `true`。

对于引用类型，如果两个操作数的引用相等（而非值相等），运算符返回 `true`。例外情况是字符串对象将进行值相等比较。

不等于

检查所提供的操作数是否不相等。

```
"a" != "b"      // 返回 true。
"a" != "a"      // 返回 false.
1 != 0          // 返回 true.
1 != 1          // 返回 false.
false != true   // 返回 true.
false != false  // 返回 false.
```

```
var x = new Object();
var y = new Object();
x != y // 返回 true, 操作数引用不同。
x != x // 返回 false, 两个操作数引用相同。
```

该运算符的结果实际上与等于 (`==`) 运算符的结果相反。

大于

检查第一个操作数是否大于第二个操作数。

```
3 > 5      // 返回 false.
1 > 0      // 返回 true.
2 > 2      // 返回 false.
```

```
var x = 10;
var y = 15;
x > y      // 返回 false.
y > x      // 返回 true.
```

小于

检查第一个操作数是否小于第二个操作数。

```
2 < 4      // 返回 true.
1 < -3     // 返回 false.
2 < 2      // 返回 false.
```

```
var x = 12;
var y = 22;
x < y      // 返回 true.
y < x      // 返回 false.
```

`MyStruct.AsInt() == 1` // Calls `AsInt()` on `MyStruct` and compares the resulting int with 1.

Unlike Visual Basic.NET, the equality comparison operator is not the same as the equality assignment operator.

```
var x = new Object();
var y = new Object();
x == y // Returns false, the operands (objects in this case) have different references.
x == x // Returns true, both operands have the same reference.
```

Not to be confused with the assignment operator (`=`).

For value types, the operator returns `true` if both operands are equal in value.

For reference types, the operator returns `true` if both operands are equal in *reference* (not value). An exception is that string objects will be compared with value equality.

Not Equals

Checks whether the supplied operands are *not* equal.

```
"a" != "b"      // Returns true.
"a" != "a"      // Returns false.
1 != 0          // Returns true.
1 != 1          // Returns false.
false != true   // Returns true.
false != false  // Returns false.
```

```
var x = new Object();
var y = new Object();
x != y // Returns true, the operands have different references.
x != x // Returns false, both operands have the same reference.
```

This operator effectively returns the opposite result to that of the equals (`==`) operator

Greater Than

Checks whether the first operand is greater than the second operand.

```
3 > 5      // Returns false.
1 > 0      // Returns true.
2 > 2      // Returns false.
```

```
var x = 10;
var y = 15;
x > y      // Returns false.
y > x      // Returns true.
```

Less Than

Checks whether the first operand is less than the second operand.

```
2 < 4      // Returns true.
1 < -3     // Returns false.
2 < 2      // Returns false.
```

```
var x = 12;
var y = 22;
x < y      // Returns true.
y < x      // Returns false.
```

大于等于

检查第一个操作数是否大于或等于第二个操作数。

```
7 >= 8    //返回 false.  
0 >= 0    //返回真.
```

小于等于

检查第一个操作数是否小于等于第二个操作数。

```
2 <= 4    //返回 true.  
1 <= -3   //返回 false.  
1 <= 1    //返回 true.
```

第3.4节：隐式转换和显式转换运算符

C# 允许用户自定义类型通过使用 `explicit` 和 `implicit` 关键字来控制赋值和类型转换。方法的签名形式为：

```
public static <implicit/explicit> operator <ResultingType>(<SourceType> myType)
```

该方法不能接受更多参数，也不能是实例方法。但它可以访问其定义类型的任何私有成员。

一个同时包含 `implicit` 和 `explicit` 转换的示例：

```
public class 二值图像  
{  
    private bool[] _像素;  
  
    public static implicit operator 彩色图像(二值图像 im)  
    {  
        return new 彩色图像(im);  
    }  
  
    public static explicit operator bool[](二值图像 im)  
    {  
        return im._像素;  
    }  
}
```

允许以下的类型转换语法：

```
var 二值图像 = new 二值图像();  
彩色图像 colorImage = 二值图像; // 隐式转换，注意缺少类型声明  
bool[] 像素 = (bool[])二值图像; // 显式转换，定义类型
```

转换操作符可以双向工作，既可以从你的类型转换，也可以转换到你的类型：

```
public class 二值图像  
{  
    public static explicit operator ColorImage(BinaryImage im)  
    {  
        return new ColorImage(im);  
    }  
}
```

Greater Than Equal To

Checks whether the first operand is greater than equal to the second operand.

```
7 >= 8    //Returns false.  
0 >= 0    //Returns true.
```

Less Than Equal To

Checks whether the first operand is less than equal to the second operand.

```
2 <= 4    //Returns true.  
1 <= -3   //Returns false.  
1 <= 1    //Returns true.
```

Section 3.4: Implicit Cast and Explicit Cast Operators

C# allows user-defined types to control assignment and casting through the use of the `explicit` and `implicit` keywords. The signature of the method takes the form:

```
public static <implicit/explicit> operator <ResultingType>(<SourceType> myType)
```

The method cannot take any more arguments, nor can it be an instance method. It can, however, access any private members of type it is defined within.

An example of both an `implicit` and `explicit` cast:

```
public class BinaryImage  
{  
    private bool[] _pixels;  
  
    public static implicit operator ColorImage(BinaryImage im)  
    {  
        return new ColorImage(im);  
    }  
  
    public static explicit operator bool[](BinaryImage im)  
    {  
        return im._pixels;  
    }  
}
```

Allowing the following cast syntax:

```
var binaryImage = new BinaryImage();  
ColorImage colorImage = binaryImage; // implicit cast, note the lack of type  
bool[] pixels = (bool[])binaryImage; // explicit cast, defining the type
```

The cast operators can work both ways, going from your type and going to your type:

```
public class BinaryImage  
{  
    public static explicit operator ColorImage(BinaryImage im)  
    {  
        return new ColorImage(im);  
    }  
}
```

```

public static explicit operator BinaryImage(ColorImage cm)
{
    return new BinaryImage(cm);
}

```

最后, `as` 关键字可以用于类型层次结构中的类型转换, 但在这种情况下不有效。即使定义了 `explicit` 或 `implicit` 转换, 也不能这样做:

```
ColorImage cm = myBinaryImage as ColorImage;
```

这将导致编译错误。

第3.5节 : 短路运算符

根据定义, 短路布尔运算符只有在第一个操作数无法确定表达式的整体结果时, 才会计算第二个操作数。

这意味着, 如果你使用 `&&` 运算符作为 `firstCondition && secondCondition`, 只有当 `firstCondition` 为真时才会计算 `secondCondition`, 当然, 只有当 `firstOperand` 和 `secondOperand` 都为真时, 整体结果才为真。这在许多场景中非常有用, 例如, 假设你想检查你的列表是否有超过三个元素, 但你还必须检查列表是否已初始化, 以避免出现

`NullReferenceException`。你可以如下实现:

```
bool hasMoreThanThreeElements = myList != null && mList.Count > 3;
```

只有在 `myList != null` 条件满足后, 才会检查 `mList.Count > 3`。

逻辑与

`&&` 是标准布尔与运算符 (`&`) 的短路版本。

```

var x = true;
var y = false;

x && x // 返回 true。
x && y // 返回 false (y 被求值)。
y && x // 返回 false (x 不被求值)。
y && y // 返回 false (右侧 y 不被求值)。

```

逻辑或

`||` 是标准布尔或运算符 (`|`) 的短路版本。

```

var x = true;
var y = false;

x || x // 返回 true (右侧的 x 不会被计算)。
x || y // 返回 true (y 不会被计算)。
y || x // 返回 true (x 和 y 都会被计算)。
y || y // 返回 false (y 和 y 都会被计算)。

```

示例用法

```

if(object != null && object.Property)
// 如果 object 为 null, 则不会访问 object.Property, 因为存在短路。

```

```

public static explicit operator BinaryImage(ColorImage cm)
{
    return new BinaryImage(cm);
}

```

Finally, the `as` keyword, which can be involved in casting within a type hierarchy, is **not** valid in this situation. Even after defining either an `explicit` or `implicit` cast, you cannot do:

```
ColorImage cm = myBinaryImage as ColorImage;
```

It will generate a compilation error.

Section 3.5: Short-circuiting Operators

By definition, the short-circuiting boolean operators will only evaluate the second operand if the first operand can not determine the overall result of the expression.

It means that, if you are using `&&` operator as `firstCondition && secondCondition` it will evaluate `secondCondition` only when `firstCondition` is true and ofcourse the overall result will be true only if both of `firstOperand` and `secondOperand` are evaluated to true. This is useful in many scenarios, for example imagine that you want to check whereas your list has more than three elements but you also have to check if list has been initialized to not run into `NullReferenceException`. You can achieve this as below:

```
bool hasMoreThanThreeElements = myList != null && mList.Count > 3;
```

`mList.Count > 3` will not be checked until `myList != null` is met.

Logical AND

`&&` is the short-circuiting counterpart of the standard boolean AND (`&`) operator.

```

var x = true;
var y = false;

x && x // Returns true.
x && y // Returns false (y is evaluated).
y && x // Returns false (x is not evaluated).
y && y // Returns false (right y is not evaluated).

```

Logical OR

`||` is the short-circuiting counterpart of the standard boolean OR (`|`) operator.

```

var x = true;
var y = false;

x || x // Returns true (right x is not evaluated).
x || y // Returns true (y is not evaluated).
y || x // Returns true (x and y are evaluated).
y || y // Returns false (y and y are evaluated).

```

Example usage

```

if(object != null && object.Property)
// object.Property is never accessed if object is null, because of the short circuit.

```

```
Action1();  
else  
Action2();
```

第3.6节 :? : 三元运算符

根据布尔表达式的值返回两个值中的一个。

语法：

```
condition ? expression_if_true : expression_if_false;
```

示例：

```
string name = "Frank";  
Console.WriteLine(name == "Frank" ? "The name is Frank" : "The name is not Frank");
```

三元运算符是右结合的，这允许使用复合三元表达式。通过在父三元表达式的真或假位置添加额外的三元表达式来实现。应注意确保可读性，但在某些情况下这可以作为有用的简写。

在此示例中，复合三元运算对clamp函数进行求值，如果当前值在范围内则返回该值，如果低于范围则返回min值，若高于范围则返回max值。

```
light.intensity = Clamp(light.intensity, minLight, maxLight);  
  
public static float Clamp(float val, float min, float max)  
{  
    return (val < min) ? min : (val > max) ? max : val;  
}
```

三元运算符也可以嵌套使用，例如：

```
a ? b ? "a 为真, b 为真" : "a 为真, b 为假" : "a 为假"  
  
// 这是从左到右进行求值的，用括号更容易看清楚：  
  
a ? (b ? x : y) : z  
  
// 结果是当 a && b 时为 x, a && !b 时为 y, !a 时为 z
```

在编写复合三元语句时，通常会使用括号或缩进来提高可读性。

expression_if_true 和 expression_if_false 的类型必须相同，或者必须存在从一种类型到另一种类型的隐式转换。

```
condition ? 3 : "Not three"; // 无法编译，因为 `int` 和 `string` 之间没有隐式转换。
```

```
condition ? 3.ToString() : "Not three"; // 可以，因为两种可能的输出都是字符串。
```

```
condition ? 3 : 3.5; // 可以，因为存在从 `int` 到 `double` 的隐式转换。三元运算符将返回 `double` 类型。
```

```
condition ? 3.5 : 3; // 可以，因为存在从 `int` 到 `double` 的隐式转换。  
三元运算符将返回一个 `double` 类型。
```

```
Action1();  
else  
Action2();
```

Section 3.6: ?: Ternary Operator

Returns one of two values depending on the value of a Boolean expression.

Syntax:

```
condition ? expression_if_true : expression_if_false;
```

Example:

```
string name = "Frank";  
Console.WriteLine(name == "Frank" ? "The name is Frank" : "The name is not Frank");
```

The ternary operator is right-associative which allows for compound ternary expressions to be used. This is done by adding additional ternary equations in either the true or false position of a parent ternary equation. Care should be taken to ensure readability, but this can be useful shorthand in some circumstances.

In this example, a compound ternary operation evaluates a clamp function and returns the current value if it's within the range, the min value if it's below the range, or the max value if it's above the range.

```
light.intensity = Clamp(light.intensity, minLight, maxLight);  
  
public static float Clamp(float val, float min, float max)  
{  
    return (val < min) ? min : (val > max) ? max : val;  
}
```

Ternary operators can also be nested, such as:

```
a ? b ? "a is true, b is true" : "a is true, b is false" : "a is false"  
  
// This is evaluated from left to right and can be more easily seen with parenthesis:  
  
a ? (b ? x : y) : z  
  
// Where the result is x if a && b, y if a && !b, and z if !a
```

When writing compound ternary statements, it's common to use parenthesis or indentation to improve readability.

The types of expression_if_true and expression_if_false must be identical or there must be an implicit conversion from one to the other.

```
condition ? 3 : "Not three"; // Doesn't compile because `int` and `string` lack an implicit conversion.
```

```
condition ? 3.ToString() : "Not three"; // OK because both possible outputs are strings.
```

```
condition ? 3 : 3.5; // OK because there is an implicit conversion from `int` to `double`. The ternary operator will return a `double`.
```

```
condition ? 3.5 : 3; // OK because there is an implicit conversion from `int` to `double`. The ternary operator will return a `double`.
```

类型和转换要求同样适用于你自己的类。

```
public class 车
{}

public class 跑车 : 车
{}

public class SUV : 车
{}

condition ? new 跑车() : new 车(); // 可以, 因为存在从 `跑车` 到 `车` 的隐式转换。
三元运算符将返回一个 `车` 类型的引用。

condition ? new 车() : new 跑车(); // 可以, 因为存在从 `跑车` 到 `车` 的隐式转换。
三元运算符将返回一个 `车` 类型的引用。

condition ? new 跑车() : new SUV(); // 无法编译, 因为不存在从 `跑车` 到 `SUV` 或从 `SUV` 到 `跑车` 的隐式转换。编译器无法智能地识别它们都隐式转换为 `车`。

condition ? new 跑车() as 车 : new SUV() as 车; // 可以, 因为两个表达式都计算为 `车` 类型的引用。三元运算符将返回一个 `车` 类型的引用。
```

第3.7节：?: (空条件运算符)

版本 ≥ 6.0

在 C# 6.0 中引入的空条件运算符?.如果其左侧表达式计算结果为null，则会立即返回null，而不是抛出NullReferenceException。如果其左侧计算结果为非null值，则行为与普通的.运算符相同。注意，由于它可能返回null，其返回类型总是可空类型。这意味着对于结构体或基本类型，它会被包装成Nullable<T>。

```
var bar = Foo.GetBar()?.Value; // 如果 GetBar() 返回 null, 则返回 null
var baz = Foo.GetBar()?.IntegerValue; // baz 的类型将是 Nullable<int>, 即 int?
```

这在触发事件时非常有用。通常你需要用 if 语句检查事件是否为null，然后再触发事件，这会引入竞态条件的可能性。使用空条件运算符可以通过以下方式解决：

```
event EventHandler<string> RaiseMe;
RaiseMe?.Invoke("Event raised");
```

第3.8节：“异或”运算符

“异或”（简称 XOR）的运算符是：^

当且仅当提供的布尔值中有且只有一个为真时，该运算符返回真。

```
true ^ false // 返回 true
false ^ true // 返回 true
false ^ false // 返回 false
true ^ true // 返回 false
```

The type and conversion requirements apply to your own classes too.

```
public class Car
{}

public class SportsCar : Car
{}

public class SUV : Car
{}

condition ? new SportsCar() : new Car(); // OK because there is an implicit conversion from `SportsCar` to `Car`. The ternary operator will return a reference of type `Car`.

condition ? new Car() : new SportsCar(); // OK because there is an implicit conversion from `SportsCar` to `Car`. The ternary operator will return a reference of type `Car`.

condition ? new SportsCar() : new SUV(); // Doesn't compile because there is no implicit conversion from `SportsCar` to `SUV` or `SUV` to `SportsCar`. The compiler is not smart enough to realize that both of them have an implicit conversion to `Car`.

condition ? new SportsCar() as Car : new SUV() as Car; // OK because both expressions evaluate to a reference of type `Car`. The ternary operator will return a reference of type `Car`.
```

Section 3.7: ?: (Null Conditional Operator)

Version ≥ 6.0

Introduced in C# 6.0, the Null Conditional Operator ?. will immediately return `null` if the expression on its left-hand side evaluates to `null`, instead of throwing a `NullReferenceException`. If its left-hand side evaluates to a non-`null` value, it is treated just like a normal . operator. Note that because it might return `null`, its return type is always a nullable type. That means that for a struct or primitive type, it is wrapped into a `Nullable<T>`.

```
var bar = Foo.GetBar()?.Value; // will return null if GetBar() returns null
var baz = Foo.GetBar()?.IntegerValue; // baz will be of type Nullable<int>, i.e. int?
```

This comes handy when firing events. Normally you would have to wrap the event call in an if statement checking for `null` and raise the event afterwards, which introduces the possibility of a race condition. Using the Null conditional operator this can be fixed in the following way:

```
event EventHandler<string> RaiseMe;
RaiseMe?.Invoke("Event raised");
```

Section 3.8: "Exclusive or" Operator

The operator for an "exclusive or" (for short XOR) is: ^

This operator returns true when one, but only one, of the supplied bools are true.

```
true ^ false // Returns true
false ^ true // Returns true
false ^ false // Returns false
true ^ true // Returns false
```

第3.9节 : default 运算符

值类型 (其中 T : struct)

内置的基本数据类型，如char、int和float，以及用struct或enum声明的用户自定义类型。它们的默认值是new T()：

```
default(int)      // 0
default(DateTime) // 0001-01-01 12:00:00 AM
default(char)      // '\0' 这是“空字符”，不是数字0或换行符。
default(Guid)      // 00000000-0000-0000-0000-000000000000
default(MyStruct)  // new MyStruct()

// 注意：枚举的默认值是0，而不是该枚举中的第一个*键*
// 因此它可能会导致Enum.IsDefined测试失败
default(MyEnum)   // (MyEnum)0
```

引用类型 (其中 T : class)

任何class、interface、数组或委托类型。它们的默认值是null：

```
default(object)    // null
default(string)    // null
default(MyClass)   // null
default(IDisposable) // null
default(dynamic)   // null
```

第3.10节 : 赋值运算符 '='

赋值运算符=将左操作数的值设置为右操作数的值，并返回该值：

```
int a = 3;      // 将值3赋给变量a
int b = a = 5; // 先将值5赋给变量a，然后同样赋给变量b
Console.WriteLine(a = 3 + 4); // 输出7
```

第3.11节 : sizeof

返回一个int类型，表示某种类型*的字节大小。

```
sizeof(bool)    // 返回1。
sizeof(byte)    // 返回1。
sizeof(sbyte)   // 返回1。
sizeof(char)    // 返回2。
sizeof(short)   // 返回2。
sizeof(ushort)  // 返回2。
sizeof(int)     // 返回4。
sizeof(uint)    // 返回4。
sizeof(float)   // 返回 4。
sizeof(long)    // 返回 8。
sizeof(ulong)   // 返回 8。
sizeof(double)  // 返回 8。
sizeof(decimal) // 返回 16。
```

*仅在安全上下文中支持某些原始类型。

Section 3.9: default Operator

Value Type (where T : struct)

The built-in primitive data types, such as `char`, `int`, and `float`, as well as user-defined types declared with `struct`, or `enum`. Their default value is `new T()` :

```
default(int)      // 0
default(DateTime) // 0001-01-01 12:00:00 AM
default(char)      // '\0' This is the "null character", not a zero or a line break.
default(Guid)      // 00000000-0000-0000-0000-000000000000
default(MyStruct)  // new MyStruct()

// Note: default of an enum is 0, and not the first *key* in that enum
// so it could potentially fail the Enum.IsDefined test
default(MyEnum)   // (MyEnum)0
```

Reference Type (where T : class)

Any `class`, `interface`, array or delegate type. Their default value is `null` :

```
default(object)    // null
default(string)    // null
default(MyClass)   // null
default(IDisposable) // null
default(dynamic)   // null
```

Section 3.10: Assignment operator '='

The assignment operator = sets the left hand operand's value to the value of right hand operand, and return that value:

```
int a = 3;      // assigns value 3 to variable a
int b = a = 5; // first assigns value 5 to variable a, then does the same for variable b
Console.WriteLine(a = 3 + 4); // prints 7
```

Section 3.11: sizeof

Returns an `int` holding the size of a type* in bytes.

```
sizeof(bool)    // Returns 1.
sizeof(byte)    // Returns 1.
sizeof(sbyte)   // Returns 1.
sizeof(char)    // Returns 2.
sizeof(short)   // Returns 2.
sizeof(ushort)  // Returns 2.
sizeof(int)     // Returns 4.
sizeof(uint)    // Returns 4.
sizeof(float)   // Returns 4.
sizeof(long)    // Returns 8.
sizeof(ulong)   // Returns 8.
sizeof(double)  // Returns 8.
sizeof(decimal) // Returns 16.
```

*Only supports certain primitive types in safe context.

在不安全上下文中，`sizeof` 可用于返回其他原始类型和结构体的大小。

```
public struct CustomType
{
    public int value;
}

static void Main()
{
    不安全
    {
        Console.WriteLine(sizeof(CustomType)); // 输出: 4
    }
}
```

第3.12节：?? 空合并运算符

空合并运算符 `??` 当左侧不为null时返回左侧。如果左侧为null，则返回右侧。

```
object foo = null;
object bar = new object();

var c = foo ?? bar;
//c 将是 bar, 因为 foo 是 null
```

`??` 运算符可以链式使用，从而省去 `if` 判断。

```
//config 将是第一个非null的返回值。
var config = RetrieveConfigOnMachine() ??
RetrieveConfigFromService() ??
new DefaultConfiguration();
```

第3.13节：位移运算符

位移运算符允许程序员通过将整数的所有位向左或向右移动来调整该整数。
下图展示了将一个值向左移动一位的效果。

左移

```
uint value = 15;           // 00001111

uint doubled = value << 1;   // 结果 = 00011110 = 30
uint shiftFour = value << 4; // 结果 = 11110000 = 240
```

右移

```
uint value = 240;          // 11110000

uint halved = value >> 1;   // 结果 = 01111000 = 120
uint shiftFour = value >> 4; // 结果 = 00001111 = 15
```

第3.14节：=> Lambda运算符

版本 ≥ 3.0

In an unsafe context, `sizeof` can be used to return the size of other primitive types and structs.

```
public struct CustomType
{
    public int value;
}

static void Main()
{
    unsafe
    {
        Console.WriteLine(sizeof(CustomType)); // outputs: 4
    }
}
```

Section 3.12: ?? Null-Coalescing Operator

The Null-Coalescing operator `??` will return the left-hand side when not null. If it is null, it will return the right-hand side.

```
object foo = null;
object bar = new object();

var c = foo ?? bar;
//c will be bar since foo was null
```

The `??` operator can be chained which allows the removal of `if` checks.

```
//config will be the first non-null returned.
var config = RetrieveConfigOnMachine() ??
RetrieveConfigFromService() ??
new DefaultConfiguration();
```

Section 3.13: Bit-Shifting Operators

The shift operators allow programmers to adjust an integer by shifting all of its bits to the left or the right. The following diagram shows the affect of shifting a value to the left by one digit.

Left-Shift

```
uint value = 15;           // 00001111

uint doubled = value << 1;   // Result = 00011110 = 30
uint shiftFour = value << 4; // Result = 11110000 = 240
```

Right-Shift

```
uint value = 240;          // 11110000

uint halved = value >> 1;   // Result = 01111000 = 120
uint shiftFour = value >> 4; // Result = 00001111 = 15
```

Section 3.14: => Lambda operator

Version ≥ 3.0

=> 运算符与赋值运算符 = 具有相同的优先级，且为右结合性。

它用于声明 lambda 表达式，并且在 LINQ 查询中被广泛使用：

```
string[] words = { "cherry", "apple", "blueberry" };

int shortestWordLength = words.Min((string w) => w.Length); //5
```

当在LINQ扩展或查询中使用时，通常可以省略对象的类型，因为编译器会推断：

```
int shortestWordLength = words.Min(w => w.Length); //也能编译，结果相同
```

lambda运算符的一般形式如下：

```
(输入参数) => 表达式
```

lambda表达式的参数在=>运算符之前指定，实际要执行的表达式/语句/代码块在运算符右侧：

```
// 表达式
(int x, string s) => s.Length > x

// 表达式
(int x, int y) => x + y

// 语句
(string x) => Console.WriteLine(x)

// 块
(string x) => {
    x += " 说你好！";
    Console.WriteLine(x);
}
```

该运算符可用于轻松定义委托，而无需编写显式方法：

```
delegate void TestDelegate(string s);

TestDelegate myDelegate = s => Console.WriteLine(s + " 世界");

myDelegate("你好");
```

代替

```
void MyMethod(string s)
{
    Console.WriteLine(s + " 世界");
}

delegate void TestDelegate(string s);

TestDelegate myDelegate = MyMethod;

myDelegate("你好");
```

The => operator has the same precedence as the assignment operator = and is right-associative.

It is used to declare lambda expressions and also it is widely used with LINQ Queries:

```
string[] words = { "cherry", "apple", "blueberry" };

int shortestWordLength = words.Min((string w) => w.Length); //5
```

When used in LINQ extensions or queries the type of the objects can usually be skipped as it is inferred by the compiler:

```
int shortestWordLength = words.Min(w => w.Length); //also compiles with the same result
```

The general form of lambda operator is the following:

```
(input parameters) => expression
```

The parameters of the lambda expression are specified before => operator, and the actual expression/statement/block to be executed is to the right of the operator:

```
// expression
(int x, string s) => s.Length > x

// expression
(int x, int y) => x + y

// statement
(string x) => Console.WriteLine(x)

// block
(string x) => {
    x += " says Hello!";
    Console.WriteLine(x);
}
```

This operator can be used to easily define delegates, without writing an explicit method:

```
delegate void TestDelegate(string s);

TestDelegate myDelegate = s => Console.WriteLine(s + " World");

myDelegate("Hello");
```

instead of

```
void MyMethod(string s)
{
    Console.WriteLine(s + " World");
}

delegate void TestDelegate(string s);

TestDelegate myDelegate = MyMethod;

myDelegate("Hello");
```

第3.15节：类成员运算符：空条件成员访问

```
var zipcode = myEmployee?.Address?.ZipCode;  
//如果左操作数为null，则返回null。  
//上述等同于：  
var zipcode = (string)null;  
if (myEmployee != null && myEmployee.Address != null)  
    zipcode = myEmployee.Address.ZipCode;
```

Section 3.15: Class Member Operators: Null Conditional Member Access

```
var zipcode = myEmployee?.Address?.ZipCode;  
//returns null if the left operand is null.  
//the above is the equivalent of:  
var zipcode = (string)null;  
if (myEmployee != null && myEmployee.Address != null)  
    zipcode = myEmployee.Address.ZipCode;
```

第3.16节：类成员运算符：空条件索引

```
var letters = null;  
char? letter = letters?[1];  
Console.WriteLine("第二个字母是 {0}",letter);  
//在上述示例中，由于letters为null，不会抛出错误  
//letter被赋值为null
```

Section 3.16: Class Member Operators: Null Conditional Indexing

```
var letters = null;  
char? letter = letters?[1];  
Console.WriteLine("Second Letter is {0}",letter);  
//in the above example rather than throwing an error because letters is null  
//letter is assigned the value null
```

第3.17节：后缀和前缀的递增与递减

后缀递增 X++ 会给 x 加 1

```
var x = 42;  
x++;  
Console.WriteLine(x); // 43
```

后缀递减

X--

将减去一

```
var x = 42  
x--;  
Console.WriteLine(x); // 41
```

++x 称为前缀递增，它先增加 x 的值然后返回 x，而 x++ 返回 x 的值然后递增

```
var x = 42;  
Console.WriteLine(++x); // 43  
System.out.println(x); // 43
```

while

```
var x = 42;  
Console.WriteLine(x++); // 42  
System.out.println(x); // 43
```

两者在for循环中都常用

```
for(int i = 0; i < 10; i++)  
{  
}
```

Section 3.17: Postfix and Prefix increment and decrement

Postfix increment X++ will add 1 to x

```
var x = 42;  
x++;  
Console.WriteLine(x); // 43
```

Postfix decrement

X--

will subtract one

```
var x = 42  
x--;  
Console.WriteLine(x); // 41
```

++x is called prefix increment it increments the value of x and then returns x while x++ returns the value of x and then increments

```
var x = 42;  
Console.WriteLine(++x); // 43  
System.out.println(x); // 43
```

while

```
var x = 42;  
Console.WriteLine(x++); // 42  
System.out.println(x); // 43
```

both are commonly used in for loop

```
for(int i = 0; i < 10; i++)  
{  
}
```

第3.18节：typeof

获取类型的System.Type对象。

```
System.Type type = typeof(Point) //System.Drawing.Point  
System.Type type = typeof(IDisposable) //System.IDisposable  
System.Type type = typeof(Colors) //System.Drawing.Color  
System.Type type = typeof(List<T>) //System.Collections.Generic.List`1[T]
```

要获取运行时类型，使用GetType方法获取当前实例的System.Type。

操作符typeof以类型名作为参数，该参数在编译时指定。

```
public class Animal {}  
public class Dog : Animal {}  
  
var animal = new Dog();  
  
Assert.IsTrue(animal.GetType() == typeof(Animal)); // 失败, animal 是 typeof(Dog)  
Assert.IsTrue(animal.GetType() == typeof(Dog)); // 通过, animal 是 typeof(Dog)  
Assert.IsTrue(animal is Animal); // 通过, animal 实现了 Animal
```

第3.19节：带赋值的二元运算符

C# 有几种运算符可以与 = 符号结合使用，先计算运算符的结果，然后将结果赋值给原变量。

示例：

```
x += y
```

等同于

```
x = x + y
```

赋值运算符：

- +=
- -=
- *=
- /=
- %=
- &=
- |=
- ^=
- <<=
- >>=

第3.20节：nameof 运算符

返回表示变量、类型或成员的未限定名称的字符串。

```
int counter = 10;  
nameof(counter); // 返回 "counter"  
Client client = new Client();
```

Section 3.18: typeof

Gets System.Type object for a type.

```
System.Type type = typeof(Point) //System.Drawing.Point  
System.Type type = typeof(IDisposable) //System.IDisposable  
System.Type type = typeof(Colors) //System.Drawing.Color  
System.Type type = typeof(List<T>) //System.Collections.Generic.List`1[T]
```

To get the run-time type, use GetType method to obtain the System.Type of the current instance.

Operator **typeof** takes a type name as parameter, which is specified at compile time.

```
public class Animal {}  
public class Dog : Animal {}  
  
var animal = new Dog();  
  
Assert.IsTrue(animal.GetType() == typeof(Animal)); // fail, animal is typeof(Dog)  
Assert.IsTrue(animal.GetType() == typeof(Dog)); // pass, animal is typeof(Dog)  
Assert.IsTrue(animal is Animal); // pass, animal implements Animal
```

Section 3.19: Binary operators with assignment

C# has several operators that can be combined with an = sign to evaluate the result of the operator and then assign the result to the original variable.

Example:

```
x += y
```

is the same as

```
x = x + y
```

Assignment operators:

- +=
- -=
- *=
- /=
- %=
- &=
- |=
- ^=
- <<=
- >>=

Section 3.20: nameof Operator

Returns a string that represents the unqualified name of a variable, type, or member.

```
int counter = 10;  
nameof(counter); // Returns "counter"  
Client client = new Client();
```

```
nameof(client.Address.PostalCode)); // 返回 "PostalCode"
```

nameof 操作符是在 C# 6.0 中引入的。它在编译时求值，返回的字符串值由编译器内联插入，因此它可以用于大多数可以使用常量字符串的场合（例如，switch 语句中的 case 标签、特性等）。它在抛出和记录异常、特性、MVC Action 链接等场景中非常有用...

第 3.21 节：类成员操作符：成员访问

```
var now = DateTime.UtcNow;  
//访问类的成员。在此例中是 UtcNow 属性。
```

第 3.22 节：类成员操作符：函数调用

```
var age = GetAge(dateOfBirth);  
//上述代码调用函数 GetAge，传入参数 dateOfBirth。
```

第3.23节：类成员运算符：聚合对象索引

```
var letters = "letters".ToCharArray();  
char letter = letters[1];  
Console.WriteLine("第二个字母是 {0}",letter);  
//在上面的例子中，我们通过调用 letters[1] 从数组中取第二个字符  
  
//注意：数组索引从0开始；即第一个字母是 letters[0]。
```

```
nameof(client.Address.PostalCode)); // Returns "PostalCode"
```

The `nameof` operator was introduced in C# 6.0. It is evaluated at compile-time and the returned string value is inserted inline by the compiler, so it can be used in most cases where the constant string can be used (e.g., the `case` labels in a `switch` statement, attributes, etc...). It can be useful in cases like raising & logging exceptions, attributes, MVC Action links, etc...

Section 3.21: Class Member Operators: Member Access

```
var now = DateTime.UtcNow;  
//accesses member of a class. In this case the UtcNow property.
```

Section 3.22: Class Member Operators: Function Invocation

```
var age = GetAge(dateOfBirth);  
//the above calls the function GetAge passing parameter dateOfBirth.
```

Section 3.23: Class Member Operators: Aggregate Object Indexing

```
var letters = "letters".ToCharArray();  
char letter = letters[1];  
Console.WriteLine("Second Letter is {0}",letter);  
//in the above example we take the second character from the array  
//by calling letters[1]  
//NB: Array Indexing starts at 0; i.e. the first letter would be given by letters[0].
```

第4章：条件语句

第4.1节：If-Else语句

编程通常需要在代码中做出决策或分支，以应对不同输入或条件下代码的运行。在C#编程语言（以及大多数编程语言）中，创建程序分支最简单且有时最有用的方法是通过If-Else语句。

假设我们有一个方法（也称函数），它接受一个int参数，表示最高为100的分数，方法将打印出是否通过的消息。

```
static void PrintPassOrFail(int score)
{
    if (score >= 50) // 如果分数大于或等于50
    {
        Console.WriteLine("通过！");
    }
    否则 // 如果分数不大于或等于50
    {
        Console.WriteLine("未通过！");
    }
}
```

查看此方法时，你可能会注意到 If 语句中的这行代码 (`score >= 50`)。这可以被视为一个 boolean 条件，如果该条件被评估为 true，则运行 if {} 之间的代码。

例如，如果这样调用此方法：`PrintPassOrFail(60);`，方法的输出将是控制台打印 通过！，因为参数值60大于或等于50。

但是，如果方法被调用为：`PrintPassOrFail(30);`，方法的输出将打印 未通过！。这是因为值30不大于或等于50，因此运行的是 else {} 之间的代码，而不是 If 语句。

在此示例中，我们假设 `score` 最大为100，但这一点尚未考虑。为了防止 `score` 超过100或可能低于0，请参见 If-Else If-Else 语句 示例。

第4.2节：If语句条件是标准的布尔表达式和数值

以下语句

```
if (conditionA && conditionB && conditionC) //...
```

完全等同于

```
bool 条件 = 条件A && 条件B && 条件C;
if (条件) // ...
```

换句话说，“if”语句中的条件只是一个普通的布尔表达式。

编写条件语句时一个常见的错误是显式地与true和false进行比较：

Chapter 4: Conditional Statements

Section 4.1: If-Else Statement

Programming in general often requires a decision or a branch within the code to account for how the code operates under different inputs or conditions. Within the C# programming language (and most programming languages for this matter), the simplest and sometimes the most useful way of creating a branch within your program is through an If-Else statement.

Lets assume we have method (a.k.a. a function) which takes an int parameter which will represent a score up to 100, and the method will print out a message saying whether we pass or fail.

```
static void PrintPassOrFail(int score)
{
    if (score >= 50) // If score is greater or equal to 50
    {
        Console.WriteLine("Pass!");
    }
    else // If score is not greater or equal to 50
    {
        Console.WriteLine("Fail!");
    }
}
```

When looking at this method, you may notice this line of code (`score >= 50`) inside the If statement. This can be seen as a boolean condition, where if the condition is evaluated to equal `true`, then the code that is in between the if {} is ran.

For example, if this method was called like this: `PrintPassOrFail(60);`, the output of the method would be a Console Print saying `Pass!` since the parameter value of 60 is greater or equal to 50.

However, if the method was called like: `PrintPassOrFail(30);`, the output of the method would print out saying `Fail!`. This is because the value 30 is not greater or equal to 50, thus the code in between the else {} is ran instead of the If statement.

In this example, we've said that `score` should go up to 100, which hasn't been accounted for at all. To account for `score` not going past 100 or possibly dropping below 0, see the **If-Else If-Else Statement** example.

Section 4.2: If statement conditions are standard boolean expressions and values

The following statement

```
if (conditionA && conditionB && conditionC) //...
```

is exactly equivalent to

```
bool conditions = conditionA && conditionB && conditionC;
if (conditions) // ...
```

in other words, the conditions inside the "if" statement just form an ordinary Boolean expression.

A common mistake when writing conditional statements is to explicitly compare to `true` and `false`:

```
if (条件A == true && 条件B == false && 条件C == true) // ...
```

这可以重写为

```
if (条件A && !条件B && 条件C)
```

第4.3节：If-Else If-Else语句

继If-Else语句示例之后，现在是介绍Else If语句的时候了。 Else If语句紧跟在If语句之后，构成If-Else If-Else结构，但本质上语法与If语句类似。它用于为代码添加比简单If-Else语句更多的分支。

在If-Else语句的示例中，示例指定分数最高为100；但从未对此进行检查。为了解决这个问题，我们修改If-Else语句中的方法，使其如下所示：

```
static void PrintPassOrFail(int score)
{
    if (score > 100) // 如果分数大于100
    {
        Console.WriteLine("错误：分数大于100！");
    }
    else if (score < 0) // 如果分数小于0
        Console.WriteLine("错误：分数小于0！");
    else if (score >= 50) // 如果分数大于或等于50
        Console.WriteLine("通过！");
    else // 如果以上条件都不满足，则分数必须在0到49之间
        Console.WriteLine("未通过！");
}
```

所有这些语句将从上到下依次执行，直到满足某个条件为止。在此方法的新更新中，我们添加了两个新的分支，以处理分数超出范围的情况。

例如，如果我们现在在代码中调用方法PrintPassOrFail(110)；，输出将是控制台打印错误：分数大于100！；如果我们调用方法PrintPassOrFail(-20)；，输出将显示错误：分数小于0！。

```
if (conditionA == true && conditionB == false && conditionC == true) // ...
```

This can be rewritten as

```
if (conditionA && !conditionB && conditionC)
```

Section 4.3: If-Else If-Else Statement

Following on from the **If-Else Statement** example, it is now time to introduce the **Else If** statement. The **Else If** statement follows directly after the **If** statement in the **If-Else If-Else** structure, but intrinsically has a similar syntax as the **If** statement. It is used to add more branches to the code than what a simple **If-Else** statement can.

In the example from **If-Else Statement**, the example specified that the score goes up to 100; however there were never any checks against this. To fix this, let's modify the method from **If-Else Statement** to look like this:

```
static void PrintPassOrFail(int score)
{
    if (score > 100) // If score is greater than 100
    {
        Console.WriteLine("Error: score is greater than 100!");
    }
    else if (score < 0) // Else If score is less than 0
    {
        Console.WriteLine("Error: score is less than 0!");
    }
    else if (score >= 50) // Else if score is greater or equal to 50
    {
        Console.WriteLine("Pass!");
    }
    else // If none above, then score must be between 0 and 49
    {
        Console.WriteLine("Fail!");
    }
}
```

All these statements will run in order from the top all the way to the bottom until a condition has been met. In this new update of the method, we've added two new branches to now accommodate for the score going *out of bounds*.

For example, if we now called the method in our code as `PrintPassOrFail(110)` ;, the output would be a Console Print saying **Error: score is greater than 100!**; and if we called the method in our code like `PrintPassOrFail(-20)` ;, the output would say **Error: score is less than 0!**.

第5章：等号运算符

第5.1节：C#中的等价类型和等号运算符

在 C# 中，有两种不同的相等性：引用相等和值相等。值相等是人们通常理解的相等含义：它表示两个对象包含相同的值。例如，两个值为 2 的整数具有值相等。引用相等意味着没有两个对象可供比较，而是有两个对象引用，它们都指向同一个对象。

```
object a = new object();
object b = a;
System.Object.ReferenceEquals(a, b); //返回 true
```

对于预定义的值类型，如果操作数的值相等，等号运算符 (==) 返回 true，否则返回 false。对于除字符串以外的引用类型，== 在其两个操作数引用同一个对象时返回 true。对于字符串类型，== 比较字符串的值。

```
// 数值相等: True
Console.WriteLine((2 + 2) == 4);

// 引用相等: 不同对象,
// 相同装箱值: False.
object s = 1;
object t = 1;
Console.WriteLine(s == t);

// 定义一些字符串:
string a = "hello";
string b = String.Copy(a);
string c = "hello";

// 比较常量和实例的字符串值: True
Console.WriteLine(a == b);

// 比较字符串引用;
// a 是常量但 b 是实例: False.
Console.WriteLine((object)a == (object)b);

// 比较字符串引用, 两个常量
// 具有相同的值, 因此字符串驻留
// 指向相同的引用: True.
Console.WriteLine((object)a == (object)c);
```

Chapter 5: Equality Operator

Section 5.1: Equality kinds in c# and equality operator

In C#, there are two different kinds of equality: reference equality and value equality. Value equality is the commonly understood meaning of equality: it means that two objects contain the same values. For example, two integers with the value of 2 have value equality. Reference equality means that there are not two objects to compare. Instead, there are two object references, both of which refer to the same object.

```
object a = new object();
object b = a;
System.Object.ReferenceEquals(a, b); //returns true
```

For predefined value types, the equality operator (==) returns true if the values of its operands are equal, false otherwise. For reference types other than string, == returns true if its two operands refer to the same object. For the string type, == compares the values of the strings.

```
// Numeric equality: True
Console.WriteLine((2 + 2) == 4);

// Reference equality: different objects,
// same boxed value: False.
object s = 1;
object t = 1;
Console.WriteLine(s == t);

// Define some strings:
string a = "hello";
string b = String.Copy(a);
string c = "hello";

// Compare string values of a constant and an instance: True
Console.WriteLine(a == b);

// Compare string references;
// a is a constant but b is an instance: False.
Console.WriteLine((object)a == (object)b);

// Compare string references, both constants
// have the same value, so string interning
// points to same reference: True.
Console.WriteLine((object)a == (object)c);
```

第6章：Equals 和 GetHashCode

第6.1节：编写良好的 GetHashCode 重写方法

GetHashCode 对 Dictionary<> 和 HashTable 有重大性能影响。

良好的 GetHashCode 方法

- 应具有均匀分布
 - 每个整数在随机实例中应该有大致相等的返回概率如果你的方法对每个实例都返回相同的
 - 整数（例如常数“999”），那么性能会很差
- 应该快速
 - 这些不是加密哈希，速度慢是加密哈希的一个特性
 - 你的哈希函数越慢，字典操作就越慢
- 必须在两个通过 Equals 判断为相等的实例上返回相同的HashCode
 - 如果不这样（例如因为 GetHashCode 返回了一个随机数），则可能无法在 List、Dictionary 或类似结构中找到对应项。

实现 GetHashCode 的一个好方法是使用一个质数作为起始值，然后将类型字段的哈希码乘以其他质数后累加到该值上：

```
public override int GetHashCode()
{
    unchecked // 溢出没关系，直接环绕

    int hash = 3049; // 起始值（质数）。

    // 当然，还要做适当的空值检查等 :)
hash = hash * 5039 + field1.GetHashCode();
hash = hash * 883 + field2.GetHashCode();
hash = hash * 9719 + field3.GetHashCode();
return hash;
}
```

只有在 Equals 方法中使用的字段才应该用于哈希函数。

如果您需要对字典/哈希表中的同一类型进行不同的处理，可以使用 IEqualityComparer。

第6.2节：默认的 Equals 行为

Equals 声明在 Object 类本身。

```
public virtual bool Equals(Object obj);
```

默认情况下，Equals 具有以下行为：

- 如果实例是引用类型，则 Equals 仅当引用相同时返回 true。
- 如果实例是值类型，则 Equals 仅当类型和值都相同时返回 true。
- string 是一个特殊情况。它的行为类似于值类型。

Chapter 6: Equals and GetHashCode

Section 6.1: Writing a good GetHashCode override

GetHashCode has major performance effects on Dictionary<> and HashTable.

Good GetHashCode Methods

- should have an even distribution
 - every integer should have a roughly equal chance of returning for a random instance
 - if your method returns the same integer (e.g. the constant '999') for each instance, you'll have bad performance
- should be quick
 - These are NOT cryptographic hashes, where slowness is a feature
 - the slower your hash function, the slower your dictionary
- must return the same HashCode on two instances that Equals evaluates to true
 - if they do not (e.g. because GetHashCode returns a random number), items may not be found in a List, Dictionary, or similar.

A good method to implement GetHashCode is to use one prime number as a starting value, and add the hashcodes of the fields of the type multiplied by other prime numbers to that:

```
public override int GetHashCode()
{
    unchecked // Overflow is fine, just wrap
    {
        int hash = 3049; // Start value (prime number).

        // Suitable nullity checks etc, of course :)
hash = hash * 5039 + field1.GetHashCode();
hash = hash * 883 + field2.GetHashCode();
hash = hash * 9719 + field3.GetHashCode();
return hash;
    }
}
```

Only the fields which are used in the Equals-method should be used for the hash function.

If you have a need to treat the same type in different ways for Dictionary/HashTables, you can use IEqualityComparer.

Section 6.2: Default Equals behavior

Equals is declared in the Object class itself.

```
public virtual bool Equals(Object obj);
```

By default, Equals has the following behavior:

- If the instance is a reference type, then Equals will return true only if the references are the same.
- If the instance is a value type, then Equals will return true only if the type and value are the same.
- string is a special case. It behaves like a value type.

```

namespace ConsoleApplication
{
    public class Program
    {
        public static void Main(string[] args)
        {
            //areFooClassEqual: False
            Foo fooClass1 = new Foo("42");
            Foo fooClass2 = new Foo("42");
            bool areFooClassEqual = fooClass1.Equals(fooClass2);
            Console.WriteLine("fooClass1 和 fooClass2 是否相等: {0}", areFooClassEqual);
            //False

            //areFooIntEqual: True
            int fooInt1 = 42;
            int fooInt2 = 42;
            bool areFooIntEqual = fooInt1.Equals(fooInt2);
            Console.WriteLine("fooInt1 和 fooInt2 是否相等: {0}", areFooIntEqual);

            //areFooStringEqual: True
            string fooString1 = "42";
            string fooString2 = "42";
            bool areFooStringEqual = fooString1.Equals(fooString2);
            Console.WriteLine("fooString1 和 fooString2 相等: {0}", areFooStringEqual);
        }

        public class Foo
        {
            public string Bar { get; }

            public Foo(string bar)
            {
                Bar = bar;
            }
        }
    }
}

```

第6.3节：在自定义类型上重写 Equals 和 GetHashCode

对于像这样的类Person：

```

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); //false 因为是引用 Equals

```

但是如下定义 Equals 和 GetHashCode：

```

public class Person
{
    public string Name { get; set; }
}

```

```

namespace ConsoleApplication
{
    public class Program
    {
        public static void Main(string[] args)
        {
            //areFooClassEqual: False
            Foo fooClass1 = new Foo("42");
            Foo fooClass2 = new Foo("42");
            bool areFooClassEqual = fooClass1.Equals(fooClass2);
            Console.WriteLine("fooClass1 和 fooClass2 are equal: {0}", areFooClassEqual);
            //False

            //areFooIntEqual: True
            int fooInt1 = 42;
            int fooInt2 = 42;
            bool areFooIntEqual = fooInt1.Equals(fooInt2);
            Console.WriteLine("fooInt1 and fooInt2 are equal: {0}", areFooIntEqual);

            //areFooStringEqual: True
            string fooString1 = "42";
            string fooString2 = "42";
            bool areFooStringEqual = fooString1.Equals(fooString2);
            Console.WriteLine("fooString1 and fooString2 are equal: {0}", areFooStringEqual);
        }

        public class Foo
        {
            public string Bar { get; }

            public Foo(string bar)
            {
                Bar = bar;
            }
        }
    }
}

```

Section 6.3: Override Equals and GetHashCode on custom types

For a class Person like:

```

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); //false because it's reference Equals

```

But defining Equals and GetHashCode as follows:

```

public class Person
{
    public string Name { get; set; }
}

```

```

public int Age { get; set; }
public string Clothes { get; set; }

public override bool Equals(object obj)
{
    var person = obj as Person;
    if(person == null) return false;
    return Name == person.Name && Age == person.Age; //比较两个人时衣服不重要
}

public override int GetHashCode()
{
    return Name.GetHashCode()*Age;
}
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); // 结果为 true

```

使用 LINQ 对 persons 进行不同查询时，也会检查 Equals 和 GetHashCode :

```

var persons = new List<Person>
{
    new Person{ Name = "Jon", Age = 20, Clothes = "some clothes" },
    new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes" },
    new Person{ Name = "Jon", Age = 20, Clothes = "" }
};

var distinctPersons = persons.Distinct().ToList(); //distinctPersons 的数量 = 2

```

第6.4节：IEqualityComparer中的Equals和GetHashCode

对于给定类型Person：

```

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

List<Person> persons = new List<Person>
{
    new Person{ Name = "Jon", Age = 20, Clothes = "some clothes" },
    new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes" },
    new Person{ Name = "Jon", Age = 20, Clothes = "" }
};

var distinctPersons = persons.Distinct().ToList(); // distinctPersons 的数量为3

```

但是将Equals和GetHashCode定义到IEqualityComparer中：

```

public class PersonComparator : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name && x.Age == y.Age; //比较时衣服不重要
    }
}

```

```

public int Age { get; set; }
public string Clothes { get; set; }

public override bool Equals(object obj)
{
    var person = obj as Person;
    if(person == null) return false;
    return Name == person.Name && Age == person.Age; //the clothes are not important when comparing two persons
}

public override int GetHashCode()
{
    return Name.GetHashCode()*Age;
}
}

var person1 = new Person { Name = "Jon", Age = 20, Clothes = "some clothes" };
var person2 = new Person { Name = "Jon", Age = 20, Clothes = "some other clothes" };

bool result = person1.Equals(person2); // result is true

```

Also using LINQ to make different queries on persons will check both Equals and GetHashCode:

```

var persons = new List<Person>
{
    new Person{ Name = "Jon", Age = 20, Clothes = "some clothes" },
    new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes" },
    new Person{ Name = "Jon", Age = 20, Clothes = "" }
};

var distinctPersons = persons.Distinct().ToList(); //distinctPersons has Count = 2

```

Section 6.4: Equals and GetHashCode in IEqualityComparer

For given type Person:

```

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Clothes { get; set; }
}

List<Person> persons = new List<Person>
{
    new Person{ Name = "Jon", Age = 20, Clothes = "some clothes" },
    new Person{ Name = "Dave", Age = 20, Clothes = "some other clothes" },
    new Person{ Name = "Jon", Age = 20, Clothes = "" }
};

var distinctPersons = persons.Distinct().ToList(); // distinctPersons has Count = 3

```

But defining Equals and GetHashCode into an IEqualityComparer:

```

public class PersonComparator : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name && x.Age == y.Age; //the clothes are not important when comparing
    }
}

```

```
两个Person ;  
}  
  
public int GetHashCode(Person obj) { return obj.Name.GetHashCode() * obj.Age; }  
}  
  
var distinctPersons = persons.Distinct(new PersonComparator()).ToList(); // distinctPersons 的  
Count = 2
```

请注意，对于此查询，如果两个对象的 Equals 返回 true 且
GetHashCode 对这两个人返回了相同的哈希码，则认为它们相等。

```
two persons;  
}  
  
public int GetHashCode(Person obj) { return obj.Name.GetHashCode() * obj.Age; }  
}  
  
var distinctPersons = persons.Distinct(new PersonComparator()).ToList(); // distinctPersons has  
Count = 2
```

Note that for this query, two objects have been considered equal if both the Equals returned true and the
GetHashCode have returned the same hash code for the two persons.

第7章：空合并运算符

参数

	详细信息
possibleNullObject	要测试是否为 null 的值。如果非 null，则返回该值。必须是可空类型。
defaultValue	如果 possibleNullObject 为 null，则返回的值。必须与 possibleNullObject 类型相同。

第7.1节：基本用法

使用空合并操作符(??)可以为可空类型指定一个默认值，如果左侧操作数为null。

```
string testString = null;
Console.WriteLine("指定的字符串是 - " + (testString ?? "未提供"));
```

[.NET Fiddle 上的实时演示](#)

这在逻辑上等同于：

```
string testString = null;
if (testString == null)
{
    Console.WriteLine("指定的字符串是 - 未提供");
}
else
{
    Console.WriteLine("指定的字符串是 - " + testString);
}
```

或者使用三元运算符(?:)：

```
string testString = null;
Console.WriteLine("指定的字符串是 - " + (testString == null ? "未提供" :
testString));
```

第7.2节：空值穿透与链式调用

左操作数必须是可空类型，而右操作数可以是可空也可以不是。结果的类型将相应确定。

非空类型

```
int? a = null;
int b = 3;
var output = a ?? b;
var type = output.GetType();

Console.WriteLine($"Output Type :{type}");
Console.WriteLine($"Output value :{output}");
```

输出：

类型 :System.Int32

Chapter 7: Null-Coalescing Operator

Parameter

possibleNullObject	The value to test for null value. If non null, this value is returned. Must be a nullable type.
defaultValue	The value returned if possibleNullObject is null. Must be the same type as possibleNullObject.

Details

Using the [null-coalescing operator](#) (??) allows you to specify a default value for a nullable type if the left-hand operand is null.

```
string testString = null;
Console.WriteLine("The specified string is - " + (testString ?? "not provided"));
```

[Live Demo on .NET Fiddle](#)

This is logically equivalent to:

```
string testString = null;
if (testString == null)
{
    Console.WriteLine("The specified string is - not provided");
}
else
{
    Console.WriteLine("The specified string is - " + testString);
}
```

or using the ternary operator (?:) operator:

```
string testString = null;
Console.WriteLine("The specified string is - " + (testString == null ? "not provided" :
testString));
```

Section 7.2: Null fall-through and chaining

The left-hand operand must be nullable, while the right-hand operand may or may not be. The result will be typed accordingly.

Non-nullable

```
int? a = null;
int b = 3;
var output = a ?? b;
var type = output.GetType();

Console.WriteLine($"Output Type :{type}");
Console.WriteLine($"Output value :{output}");
```

Output:

Type :System.Int32

值 :3

[查看演示](#)

可空类型

```
int? a = null;  
int? b = null;  
var output = a ?? b;
```

output 的类型将是 int?, 且等于 b, 或者 null。

多重合并运算符

合并运算符也可以链式使用：

```
int? a = null;  
int? b = null;  
int c = 3;  
var output = a ?? b ?? c;  
  
var type = output.GetType();  
Console.WriteLine($"Type :{type}");  
Console.WriteLine($"value :{output}");
```

输出：

类型 :System.Int32
值 :3

[查看演示](#)

空条件链

空合并运算符可以与空传播运算符配合使用，以更安全地访问对象的属性。

```
object o = null;  
var output = o?.ToString() ?? "Default Value";
```

输出：

类型 :System.String
值 :Default Value

[查看演示](#)

第7.3节：带方法调用的空合并运算符

空合并运算符使得确保可能返回null的方法回退到默认值变得简单。

value :3

[View Demo](#)

Nullable

```
int? a = null;  
int? b = null;  
var output = a ?? b;
```

output will be of type int? and equal to b, or null.

Multiple Coalescing

Coalescing can also be done in chains:

```
int? a = null;  
int? b = null;  
int c = 3;  
var output = a ?? b ?? c;
```

```
var type = output.GetType();  
Console.WriteLine($"Type :{type}");  
Console.WriteLine($"value :{output}");
```

Output:

Type :System.Int32
value :3

[View Demo](#)

Null Conditional Chaining

The null coalescing operator can be used in tandem with the null propagation operator to provide safer access to properties of objects.

```
object o = null;  
var output = o?.ToString() ?? "Default Value";
```

Output:

Type :System.String
value :Default Value

[View Demo](#)

Section 7.3: Null coalescing operator with method calls

The null coalescing operator makes it easy to ensure that a method that may return null will fall back to a default value.

没有空合并运算符时：

```
string name = GetName();  
  
if (name == null)  
    name = "Unknown!";
```

使用空合并运算符时：

```
string name = GetName() ?? "Unknown!";
```

第7.4节：使用现有的或创建新的

这个功能真正有用的一个常见使用场景是，当你在集合中查找一个对象时，如果该对象不存在，则需要创建一个新的。

```
IEnumerable<MyClass> myList = GetMyList();  
var item = myList.SingleOrDefault(x => x.Id == 2) ?? new MyClass { Id = 2 };
```

第7.5节：使用空合并运算符的延迟属性初始化

```
private List<FooBar> _fooBars;  
  
public List<FooBar> FooBars  
{  
    get { return _fooBars ?? (_fooBars = new List<FooBar>()); }  
}
```

第一次访问属性.FooBars时，变量_fooBars的值将被评估为null，因此会执行赋值语句，赋值并返回结果值。

线程安全

这不是实现延迟属性的线程安全方式。要实现线程安全的延迟加载，请使用.NET框架内置的Lazy<T>类。

C# 6 使用表达式主体的语法糖

请注意，自 C# 6 起，可以使用表达式主体简化属性的语法：

```
private List<FooBar> _fooBars;  
  
public List<FooBar> FooBars => _fooBars ?? ( _fooBars = new List<FooBar>() );
```

后续对该属性的访问将返回存储在_fooBars变量中的值。

MVVM 模式中的示例

这通常用于在 MVVM 模式中实现命令。与其在视图模型构造时立即初始化命令，不如使用此模式延迟初始化命令，具体如下：

```
private ICommand _actionCommand = null;  
public ICommand ActionCommand =>  
    _actionCommand ?? ( _actionCommand = new DelegateCommand( DoAction ) );
```

Without the null coalescing operator:

```
string name = GetName();  
  
if (name == null)  
    name = "Unknown!";
```

With the null coalescing operator:

```
string name = GetName() ?? "Unknown!";
```

Section 7.4: Use existing or create new

A common usage scenario that this feature really helps with is when you are looking for an object in a collection and need to create a new one if it does not already exist.

```
IEnumerable<MyClass> myList = GetMyList();  
var item = myList.SingleOrDefault(x => x.Id == 2) ?? new MyClass { Id = 2 };
```

Section 7.5: Lazy properties initialization with null coalescing operator

```
private List<FooBar> _fooBars;  
  
public List<FooBar> FooBars  
{  
    get { return _fooBars ?? (_fooBars = new List<FooBar>()); }  
}
```

The first time the property .FooBars is accessed the _fooBars variable will evaluate as `null`, thus falling through to the assignment statement assigns and evaluates to the resulting value.

Thread safety

This is **not thread-safe** way of implementing lazy properties. For thread-safe laziness, use the `Lazy<T>` class built into the .NET Framework.

C# 6 Syntactic Sugar using expression bodies

Note that since C# 6, this syntax can be simplified using expression body for the property:

```
private List<FooBar> _fooBars;  
  
public List<FooBar> FooBars => _fooBars ?? ( _fooBars = new List<FooBar>() );
```

Subsequent accesses to the property will yield the value stored in the _fooBars variable.

Example in the MVVM pattern

This is often used when implementing commands in the MVVM pattern. Instead of initializing the commands eagerly with the construction of a viewmodel, commands are lazily initialized using this pattern as follows:

```
private ICommand _actionCommand = null;  
public ICommand ActionCommand =>  
    _actionCommand ?? ( _actionCommand = new DelegateCommand( DoAction ) );
```

第8章：空条件运算符

第8.1节：空条件运算符

?. 操作符是语法糖，用于避免冗长的空值检查。它也被称为安全导航操作符。

以下示例中使用的类：

```
public class Person
{
    public int Age { get; set; }
    public string Name { get; set; }
    public Person Spouse { get; set; }
}
```

如果一个对象可能为 null（例如返回引用类型的函数），必须先检查该对象是否为 null，以防止可能的NullReferenceException。没有空条件运算符时，代码如下：

```
Person person = GetPerson();

int? age = null;
if (person != null)
    age = person.Age;
```

使用空条件运算符的相同示例：

```
Person person = GetPerson();

var age = person?.Age; // 即使 'person' 不为 null, 'age' 也将是 'int?' 类型
```

运算符链式调用

空条件运算符可以组合用于对象的成员及子成员。

```
// 如果 'person' 或 'person.Spouse' 为 null, 则结果为 null
int? spouseAge = person?.Spouse?.Age;
```

与空合并运算符结合使用

空条件运算符可以与空合并运算符结合使用，以提供默认值：

```
// 如果 person、Spouse 或 Name 为 null, spouseDisplayName 将为 "N/A"
var spouseDisplayName = person?.Spouse?.Name ?? "N/A";
```

第8.2节：空条件索引

与 ?. 运算符类似，空条件索引运算符在索引可能为 null 的集合时会检查 null 值。

```
string item = collection?[index];
```

是以下语法糖

```
string item = null;
```

Chapter 8: Null-conditional Operators

Section 8.1: Null-Conditional Operator

The ?. operator is syntactic sugar to avoid verbose null checks. It's also known as the [Safe navigation operator](#).

Class used in the following example:

```
public class Person
{
    public int Age { get; set; }
    public string Name { get; set; }
    public Person Spouse { get; set; }
}
```

If an object is potentially null (such as a function that returns a reference type) the object must first be checked for null to prevent a possible NullReferenceException. Without the null-conditional operator, this would look like:

```
Person person = GetPerson();

int? age = null;
if (person != null)
    age = person.Age;
```

The same example using the null-conditional operator:

```
Person person = GetPerson();

var age = person?.Age; // 'age' will be of type 'int?', even if 'person' is not null
```

Chaining the Operator

The null-conditional operator can be combined on the members and sub-members of an object.

```
// Will be null if either 'person' or 'person.Spouse' are null
int? spouseAge = person?.Spouse?.Age;
```

Combining with the Null-Coalescing Operator

The null-conditional operator can be combined with the null-coalescing operator to provide a default value:

```
// spouseDisplayName will be "N/A" if person, Spouse, or Name is null
var spouseDisplayName = person?.Spouse?.Name ?? "N/A";
```

Section 8.2: The Null-Conditional Index

Similarly to the ?. operator, the null-conditional index operator checks for null values when indexing into a collection that may be null.

```
string item = collection?[index];
```

is syntactic sugar for

```
string item = null;
```

```
if(collection != null)
{
item = collection[index];
}
```

第8.3节：避免NullReferenceException

```
var person = new Person
{
Address = null;
};

var city = person.Address.City; //抛出NullReferenceException
var nullableCity = person.Address?.City; //返回null值
```

此效果可以链式调用：

```
var person = new Person
{
Address = new Address
{
State = new State
{
Country = null
}
};
};

// 这将始终返回至少为"null"的值以存储,
// 而不是抛出 NullReferenceException
var countryName = person?.Address?.State?.Country?.Name;
```

第8.4节：空条件运算符可以与扩展方法一起使用

扩展方法可以作用于空引用，但你仍然可以使用?.进行空检查。

```
public class Person
{
    public string Name {get; set;}
}

public static class PersonExtensions
{
    public static int GetNameLength(this Person person)
    {
        return person == null ? -1 : person.Name.Length;
    }
}
```

通常，该方法会针对null引用触发，并返回-1：

```
Person person = null;
int nameLength = person.GetNameLength(); // 返回 -1
```

使用 ?. 运算符时，方法不会针对 null 引用触发，且类型为 int?：

```
if(collection != null)
{
    item = collection[index];
}
```

Section 8.3: Avoiding NullReferenceExceptions

```
var person = new Person
{
    Address = null;
};

var city = person.Address.City; //throws a NullReferenceException
var nullableCity = person.Address?.City; //returns the value of null
```

This effect can be chained together:

```
var person = new Person
{
    Address = new Address
    {
        State = new State
        {
            Country = null
        }
    };
};

// this will always return a value of at least "null" to be stored instead
// of throwing a NullReferenceException
var countryName = person?.Address?.State?.Country?.Name;
```

Section 8.4: Null-conditional Operator can be used with Extension Method

Extension Method can work on null references, but you can use ?. to null-check anyway.

```
public class Person
{
    public string Name {get; set;}
}

public static class PersonExtensions
{
    public static int GetNameLength(this Person person)
    {
        return person == null ? -1 : person.Name.Length;
    }
}
```

Normally, the method will be triggered for null references, and return -1:

```
Person person = null;
int nameLength = person.GetNameLength(); // returns -1
```

Using ?. the method will not be triggered for null references, and the type is int?:

```
Person person = null;  
int? nameLength = person?.GetNameLength(); // nameLength 为 null.
```

这种行为实际上是 `?.` 运算符工作方式所预期的：它会避免对 `null` 实例调用实例方法，以避免 `NullReferenceException`。然而，尽管扩展方法的声明方式不同，同样的逻辑也适用于扩展方法。

有关为什么在第一个示例中调用扩展方法的更多信息，请参见[扩展方法 - null 检查文档](#)。

```
Person person = null;  
int? nameLength = person?.GetNameLength(); // nameLength is null.
```

This behavior is actually expected from the way in which the `?.` operator works: it will avoid making instance method calls for null instances, in order to avoid `NullReferenceExceptions`. However, the same logic applies to the extension method, despite the difference on how the method is declared.

For more information on why the extension method is called in the first example, please see the [Extension methods - null checking documentation](#).

第9章：nameof 运算符

nameof 运算符允许你以字符串形式获取 变量、类型 或 成员 的名称，而无需将其硬编码为字面量。

该操作在编译时求值，这意味着你可以使用 IDE 的重命名功能重命名被引用的标识符，名称字符串也会随之更新。

第9.1节：基本用法：打印变量名

nameof操作符允许你以字符串形式获取变量、类型或成员的名称，而无需将其硬编码为字面量。该操作在编译时求值，这意味着你可以使用IDE的重命名功能重命名被引用的标识符，名称字符串也会随之更新。

```
var myString = "String Contents";
Console.WriteLine(nameof(myString));
```

将输出

```
myString
```

因为变量的名称是“myString”。重构变量名会改变该字符串。

如果对引用类型调用，nameof操作符返回当前引用的名称，而不是底层对象的名称或类型名称。例如：

```
string greeting = "Hello!";
Object mailMessageBody = greeting;

Console.WriteLine(nameof(greeting)); // 返回 "greeting"
Console.WriteLine(nameof(mailMessageBody)); // 返回 "mailMessageBody", 而不是 "greeting"!
```

第9.2节：触发PropertyChanged事件

代码片段

```
public class Person : INotifyPropertyChanged
{
    private string _address;

    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    public string Address
    {
        get { return _address; }
        set
        {
            if (_address == value)
            {
```

Chapter 9: nameof Operator

The `nameof` operator allows you to get the name of a **variable**, **type** or **member** in string form without hard-coding it as a literal.

The operation is evaluated at compile-time, which means that you can rename a referenced identifier, using an IDE's rename feature, and the name string will update with it.

Section 9.1: Basic usage: Printing a variable name

The `nameof` operator allows you to get the name of a variable, type or member in string form without hard-coding it as a literal. The operation is evaluated at compile-time, which means that you can rename, using an IDE's rename feature, a referenced identifier and the name string will update with it.

```
var myString = "String Contents";
Console.WriteLine(nameof(myString));
```

Would output

```
myString
```

because the name of the variable is "myString". Refactoring the variable name would change the string.

If called on a reference type, the `nameof` operator returns the name of the current reference, *not* the name or type name of the underlying object. For example:

```
string greeting = "Hello!";
Object mailMessageBody = greeting;

Console.WriteLine(nameof(greeting)); // Returns "greeting"
Console.WriteLine(nameof(mailMessageBody)); // Returns "mailMessageBody", NOT "greeting"!
```

Section 9.2: Raising PropertyChanged event

Snippet

```
public class Person : INotifyPropertyChanged
{
    private string _address;

    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    public string Address
    {
        get { return _address; }
        set
        {
            if (_address == value)
```

```

        return;
    }

    _address = value;
    OnPropertyChanged(nameof(Address));
}
}

...
var person = new Person();
person.PropertyChanged += (s,e) => Console.WriteLine(e.PropertyName);

person.Address = "123 Fake Street";

```

控制台输出

地址

第9.3节：参数检查和保护子句

推荐

```

public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException(nameof(orderLine));
        ...
    }
}

```

超过

```

public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException("orderLine");
        ...
    }
}

```

使用nameof特性使重构方法参数更加容易。

第9.4节：强类型MVC动作链接

代替通常的松散类型：

```
@Html.ActionLink("Log in", "UserController", "LogIn")
```

现在你可以创建强类型的动作链接：

```
@Html.ActionLink("Log in", @typeof(UserController), @nameof(UserController.LogIn))
```

```

        return;
    }

    _address = value;
    OnPropertyChanged(nameof(Address));
}
}

...
var person = new Person();
person.PropertyChanged += (s,e) => Console.WriteLine(e.PropertyName);

person.Address = "123 Fake Street";

```

Console Output

Address

Section 9.3: Argument Checking and Guard Clauses

Prefer

```

public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException(nameof(orderLine));
        ...
    }
}

```

Over

```

public class Order
{
    public OrderLine AddOrderLine(OrderLine orderLine)
    {
        if (orderLine == null) throw new ArgumentNullException("orderLine");
        ...
    }
}

```

Using the `nameof` feature makes it easier to refactor method parameters.

Section 9.4: Strongly typed MVC action links

Instead of the usual loosely typed:

```
@Html.ActionLink("Log in", "UserController", "LogIn")
```

You can now make action links strongly typed:

```
@Html.ActionLink("Log in", @typeof(UserController), @nameof(UserController.LogIn))
```

现在如果你想重构代码并将UserController.LogIn方法重命名为UserController.SignIn，你无需担心查找所有字符串出现的位置。编译器会完成这项工作。

第9.5节：处理PropertyChanged事件

代码片段

```
public class BugReport : INotifyPropertyChanged
{
    public string Title { ... }
    public BugStatus Status { ... }
}

...
private void BugReport_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
    var bugReport = (BugReport)sender;

    switch (e.PropertyName)
    {
        case nameof(bugReport.Title):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Title);
            break;

        case nameof(bugReport.Status):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Status);
            break;
    }
}

var report = new BugReport();
report.PropertyChanged += BugReport_PropertyChanged;

report.Title = "Everything is on fire and broken";
report.Status = BugStatus.ShowStopper;
```

控制台输出

```
标题变更为 Everything is on fire and broken
状态变更为 ShowStopper
```

第9.6节：应用于泛型类型参数

代码片段

```
public class SomeClass<TItem>
{
    public void PrintTypeName()
    {
        Console.WriteLine(nameof(TItem));
    }
}
```

Now if you want to refactor your code and rename the UserController.LogIn method to UserController.SignIn, you don't need to worry about searching for all string occurrences. The compiler will do the job.

Section 9.5: Handling PropertyChanged events

Snippet

```
public class BugReport : INotifyPropertyChanged
{
    public string Title { ... }
    public BugStatus Status { ... }
}

...
private void BugReport_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
    var bugReport = (BugReport)sender;

    switch (e.PropertyName)
    {
        case nameof(bugReport.Title):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Title);
            break;

        case nameof(bugReport.Status):
            Console.WriteLine("{0} changed to {1}", e.PropertyName, bugReport.Status);
            break;
    }
}

var report = new BugReport();
report.PropertyChanged += BugReport_PropertyChanged;

report.Title = "Everything is on fire and broken";
report.Status = BugStatus.ShowStopper;
```

Console Output

```
Title changed to Everything is on fire and broken
Status changed to ShowStopper
```

Section 9.6: Applied to a generic type parameter

Snippet

```
public class SomeClass<TItem>
{
    public void PrintTypeName()
    {
        Console.WriteLine(nameof(TItem));
    }
}
```

...

```
var myClass = new SomeClass<int>();
myClass.PrintTypeName();

Console.WriteLine(nameof(SomeClass<int>));
```

控制台输出

```
TItem
SomeClass
```

第9.7节：打印参数名称

代码片段

```
public void DoSomething(int paramValue)
{
    Console.WriteLine(nameof(paramValue));
}

int myValue = 10;
DoSomething(myValue);
```

控制台输出

```
paramValue
```

第9.8节：应用于限定标识符

代码片段

```
Console.WriteLine(nameof(CompanyNamespace.MyNamespace));
Console.WriteLine(nameof(MyClass));
Console.WriteLine(nameof(MyClass.MyNestedClass));
Console.WriteLine(nameof(MyNamespace.MyClass.MyNestedClass.MyStaticProperty));
```

控制台输出

```
MyNamespace
MyClass
MyNestedClass
MyStaticProperty
```

...

```
var myClass = new SomeClass<int>();
myClass.PrintTypeName();

Console.WriteLine(nameof(SomeClass<int>));
```

Console Output

```
TItem
SomeClass
```

Section 9.7: Printing a parameter name

Snippet

```
public void DoSomething(int paramValue)
{
    Console.WriteLine(nameof(paramValue));
}

int myValue = 10;
DoSomething(myValue);
```

Console Output

```
paramValue
```

Section 9.8: Applied to qualified identifiers

Snippet

```
Console.WriteLine(nameof(CompanyNamespace.MyNamespace));
Console.WriteLine(nameof(MyClass));
Console.WriteLine(nameof(MyClass.MyNestedClass));
Console.WriteLine(nameof(MyNamespace.MyClass.MyNestedClass.MyStaticProperty));
```

Console Output

```
MyNamespace
MyClass
MyNestedClass
MyStaticProperty
```

第10章：逐字字符串

第10.1节：插值逐字字符串

逐字字符串可以与C#6中新增的字符串插值功能结合使用。

```
Console.WriteLine($@"Testing \n 1 2 {5 - 2}New line")  
;
```

输出：

```
Testing \n 1 2 3  
New line
```

[.NET Fiddle 在线演示](#)

正如从逐字字符串中预期的那样，反斜杠不会被视为转义字符。正如从插值字符串中预期的那样，花括号内的任何表达式都会在插入字符串的该位置之前被计算。

第10.2节：转义双引号

逐字字符串中的双引号可以通过使用两个连续的双引号""来转义，以表示结果字符串中的一个双引号"。

```
var str = @"""我不这么认为， ""他说。";  
Console.WriteLine(str);
```

输出：

```
"我不这么认为， "他说。
```

[.NET Fiddle 在线演示](#)

第10.3节：逐字字符串指示编译器不使用字符转义

在普通字符串中，反斜杠字符是转义字符，它指示编译器查看下一个字符（或字符序列）以确定字符串中的实际字符。（完整的字符转义列表）

在逐字字符串中，没有字符转义（除了""会被转换成"）。要使用逐字字符串，只需在起始引号前加上@。

这个逐字字符串

```
var filename = @"c:\temp\newfile.txt"
```

输出：

Chapter 10: Verbatim Strings

Section 10.1: Interpolated Verbatim Strings

Verbatim strings can be combined with the new String interpolation features found in C#6.

```
Console.WriteLine($@"Testing \n 1 2 {5 - 2}  
New line");
```

Output:

```
Testing \n 1 2 3  
New line
```

[Live Demo on .NET Fiddle](#)

As expected from a verbatim string, the backslashes are ignored as escape characters. And as expected from an interpolated string, any expression inside curly braces is evaluated before being inserted into the string at that position.

Section 10.2: Escaping Double Quotes

Double Quotes inside verbatim strings can be escaped by using 2 sequential double quotes "" to represent one double quote " in the resulting string.

```
var str = @"""I don't think so,"" he said.";  
Console.WriteLine(str);
```

Output:

```
"I don't think so," he said.
```

[.NET Fiddle 在线演示](#)

Section 10.3: Verbatim strings instruct the compiler to not use character escapes

In a normal string, the backslash character is the escape character, which instructs the compiler to look at the next character(s) to determine the actual character in the string. (Full list of character escapes)

In verbatim strings, there are no character escapes (except for "" which is turned into a "). To use a verbatim string, just prepend a @ before the starting quotes.

This verbatim string

```
var filename = @"c:\temp\newfile.txt"
```

Output:

c:empewfile.txt

与使用普通（非逐字）字符串相反：

```
var filename = "c:empewfile.txt"
```

将输出：

```
c:    emp  
ewfile.txt
```

使用字符转义。（被替换为制表符，被替换为换行符。）[.NET Fiddle 在线演示](#)

第10.4节：多行字符串

```
var multiLine = @"This is a  
multiline paragraph";
```

输出：

```
这是一个  
多行段落
```

[.NET Fiddle 在线演示](#)

包含双引号的多行字符串也可以像单行字符串一样进行转义，因为它们是逐字字符串。

```
var multilineWithDoubleQuotes = @"我去了一个名为  
    ""圣地亚哥""的城市  
暑假期间。";
```

[.NET Fiddle 在线演示](#)

需要注意的是，这里第2行和第3行开头的空格/制表符实际上存在于变量的值中；请查看这个问题以获取可能的解决方案。

c:\temp\newfile.txt

As opposed to using an ordinary (non-verbatim) string:

```
var filename = "c:\temp\newfile.txt"
```

that will output:

```
c:    emp  
ewfile.txt
```

using character escaping. (The \t is replaced with a tab character and the \n is replace with a newline.)

[Live Demo on .NET Fiddle](#)

Section 10.4: Multiline Strings

```
var multiLine = @"This is a  
multiline paragraph";
```

Output:

```
This is a  
multiline paragraph
```

[.NET Fiddle 在线演示](#)

Multi-line strings that contain double-quotes can also be escaped just as they were on a single line, because they are verbatim strings.

```
var multilineWithDoubleQuotes = @"I went to a city named  
    ""San Diego""  
during summer vacation.;"
```

[.NET Fiddle 在线演示](#)

It should be noted that the spaces/tabulations at the start of lines 2 and 3 here are actually present in the value of the variable; check [this question](#) for possible solutions.

第11章：常见字符串操作

第11.1节：格式化字符串

使用String.Format()方法，用指定对象的字符串表示替换字符串中的一个或多个项：

```
String.Format("Hello {0} Foo {1}", "World", "Bar") //Hello World Foo Bar
```

第11.2节：正确反转字符串

大多数人在需要反转字符串时，通常会这样做：

```
char[] a = s.ToCharArray();
System.Array.Reverse(a);
string r = new string(a);
```

然而，这些人没有意识到的是，这实际上是错误的。

我并不是指缺少空指针检查（NULL check）。

实际上这是错误的，因为一个字形/字素簇（Glyph/GraphemeCluster）可以由多个代码点（即字符）组成。

要理解这是为什么，我们首先必须了解“字符”这个术语的真正含义。

参考资料：

字符是一个多义词，可以有多种含义。

代码点是信息的最小单位。文本是代码点的序列。每个代码点都是一个数字，由Unicode标准赋予其意义。

字素（grapheme）是一串一个或多个代码点，这些代码点被显示为一个单一的图形单元，读者将其识别为书写系统中的一个单独元素。例如，a 和 ä 都是字素，但它们可能由多个代码点组成（例如，ä 可能是两个代码点，一个是基础字符 a，后面跟着一个变音符号；但也有一种替代的、传统的单一代码点表示该字素）。有些代码点永远不会成为任何字素的一部分（例如零宽非连接符或方向覆盖符）。

字形（glyph）是一种图像，通常存储在字体中（字体是字形的集合），用于表示字素或其部分。字体可能将多个字形组合成一个单一的表示，例如，如果上述的 ä 是单一代码点，字体可能选择将其渲染为两个分开的、空间重叠的字形。

对于OTF字体，字体的GSUB和GPOS表包含替换和定位信息以实现此功能。字体也可能包含同一字素的多个替代字形。

所以在 C# 中，字符实际上是一个代码点。

这意味着，如果你只是简单地反转一个有效字符串，比如Les Misérables，它看起来会是这样

```
string s = "Les Mise\u0301rables";
```

作为字符序列，你会得到：

Chapter 11: Common String Operations

Section 11.1: Formatting a string

Use the `String.Format()` method to replace one or more items in the string with the string representation of a specified object:

```
String.Format("Hello {0} Foo {1}", "World", "Bar") //Hello World Foo Bar
```

Section 11.2: Correctly reversing a string

Most times when people have to reverse a string, they do it more or less like this:

```
char[] a = s.ToCharArray();
System.Array.Reverse(a);
string r = new string(a);
```

However, what these people don't realize is that this is actually wrong. And I don't mean because of the missing NULL check.

It is actually wrong because a Glyph/GraphemeCluster can consist out of several codepoints (aka. characters).

To see why this is so, we first have to be aware of the fact what the term "character" actually means.

Reference:

Character is an overloaded term than can mean many things.

A code point is the atomic unit of information. Text is a sequence of code points. Each code point is a number which is given meaning by the Unicode standard.

A grapheme is a sequence of one or more code points that are displayed as a single, graphical unit that a reader recognizes as a single element of the writing system. For example, both a and ä are graphemes, but they may consist of multiple code points (e.g. ä may be two code points, one for the base character a followed by one for the diaeresis; but there's also an alternative, legacy, single code point representing this grapheme). Some code points are never part of any grapheme (e.g. the zero-width non-joiner, or directional overrides).

A glyph is an image, usually stored in a font (which is a collection of glyphs), used to represent graphemes or parts thereof. Fonts may compose multiple glyphs into a single representation, for example, if the above ä is a single code point, a font may chose to render that as two separate, spatially overlaid glyphs. For OTF, the font's GSUB and GPOS tables contain substitution and positioning information to make this work. A font may contain multiple alternative glyphs for the same grapheme, too.

So in C#, a character is actually a CodePoint.

Which means, if you just reverse a valid string like Les Misérables, which can look like this

```
string s = "Les Mise\u0301rables";
```

as a sequence of characters, you will get:

如你所见，重音符号在R字符上，而不是在e字符上。

虽然对字符数组执行string.reverse两次反转会得到原始字符串，但这种反转绝对不是原始字符串的真正反转。

你需要只反转每个GraphemeCluster（字形簇）。

所以，如果正确操作，你会这样反转字符串：

```
private static System.Collections.Generic.List<string> GraphemeClusters(string s)
{
    System.Collections.Generic.List<string> ls = new System.Collections.Generic.List<string>();

    System.Globalization.TextElementEnumerator enumerator =
System.Globalization.StringInfo.GetTextElementEnumerator(s);
    while (enumerator.MoveNext())
    {
        ls.Add((string)enumerator.Current);
    }

    return ls;
}

// this
private static string ReverseGraphemeClusters(string s)
{
    if(string.IsNullOrEmpty(s) || s.Length == 1)
        return s;

    System.Collections.Generic.List<string> ls = GraphemeClusters(s);
    ls.Reverse();

    return string.Join("", ls.ToArray());
}

public static void TestMe()
{
    string s = "Les Mise\u0301rables";
    // s = "no\u0301l";
    string r = ReverseGraphemeClusters(s);

    // 这样写是错误的：
    // char[] a = s.ToCharArray();
    // System.Array.Reverse(a);
    // string r = new string(a);

    System.Console.WriteLine(r);
}
```

而且——哦，太好了——如果你像这样正确地做，你会发现它也适用于亚洲/南亚/东亚语言（以及法语/瑞典语/挪威语等）……

第11.3节：将字符串填充到固定长度

```
string s = "Foo";
string paddedLeft = s.PadLeft(5);      // paddedLeft = " Foo" (默认用空格填充)
string paddedRight = s.PadRight(6, '+'); // paddedRight = "Foo++"
```

As you can see, the accent is on the R character, instead of the e character.

Although string.reverse.reverse will yield the original string if you both times reverse the char array, this kind of reversal is definitely NOT the reverse of the original string.

You'll need to reverse each GraphemeCluster only.

So, if done correctly, you reverse a string like this:

```
private static System.Collections.Generic.List<string> GraphemeClusters(string s)
{
    System.Collections.Generic.List<string> ls = new System.Collections.Generic.List<string>();

    System.Globalization.TextElementEnumerator enumerator =
System.Globalization.StringInfo.GetTextElementEnumerator(s);
    while (enumerator.MoveNext())
    {
        ls.Add((string)enumerator.Current);
    }

    return ls;
}

// this
private static string ReverseGraphemeClusters(string s)
{
    if(string.IsNullOrEmpty(s) || s.Length == 1)
        return s;

    System.Collections.Generic.List<string> ls = GraphemeClusters(s);
    ls.Reverse();

    return string.Join("", ls.ToArray());
}

public static void TestMe()
{
    string s = "Les Mise\u0301rables";
    // s = "no\u0301l";
    string r = ReverseGraphemeClusters(s);

    // This would be wrong:
    // char[] a = s.ToCharArray();
    // System.Array.Reverse(a);
    // string r = new string(a);

    System.Console.WriteLine(r);
}
```

And - oh joy - you'll realize if you do it correctly like this, it will also work for Asian/South-Asian/East-Asian languages (and French/Swedish/Norwegian, etc.)...

Section 11.3: Padding a string to a fixed length

```
string s = "Foo";
string paddedLeft = s.PadLeft(5);      // paddedLeft = " Foo" (pads with spaces by default)
string paddedRight = s.PadRight(6, '+'); // paddedRight = "Foo++"
```

```
string noPadded = s.PadLeft(2); // noPadded = "Foo" (原字符串不会被缩短)
```

第11.4节：从字符串右侧获取x个字符

Visual Basic 有 Left、Right 和 Mid 函数，分别返回字符串的左侧、右侧和中间的字符。这些方法在C#中不存在，但可以通过Substring()来实现。它们可以作为扩展方法实现，如下所示：

```
public static class StringExtensions
{
    /// <summary>
    /// VB Left 函数
    /// </summary>
    /// <param name="stringparam"></param>
    /// <param name="numchars"></param>
    /// <returns>最左边的 numchars 个字符</returns>
    public static string Left( this string stringparam, int numchars )
    {
        // 处理可能传入的空值或数字字符串参数
        stringparam += string.Empty;

        // 处理可能传入的负数 numchars
        numchars = Math.Abs( numchars );

        // 验证 numchars 参数
        if (numchars > stringparam.Length)
            numchars = stringparam.Length;

        return stringparam.Substring( 0, numchars );
    }

    /// <summary>
    /// VB Right 函数
    /// </summary>
    /// <param name="stringparam"></param>
    /// <param name="numchars"></param>
    /// <returns>最右边的 numchars 个字符</returns>
    public static string Right( this string stringparam, int numchars )
    {
        // 处理可能传入的空值或数字字符串参数
        stringparam += string.Empty;

        // 处理可能传入的负数 numchars
        numchars = Math.Abs( numchars );

        // 验证 numchars 参数
        if (numchars > stringparam.Length)
            numchars = stringparam.Length;

        return stringparam.Substring( stringparam.Length - numchars );
    }

    /// <summary>
    /// VB Mid 函数 - 到字符串末尾
    /// </summary>
    /// <param name="stringparam"></param>
    /// <param name="startIndex">VB 风格的起始索引，首字符起始索引 = 1</param>
    /// <returns>从起始索引字符开始的字符串剩余部分</returns>
    public static string Mid( this string stringparam, int startIndex )
```

```
string noPadded = s.PadLeft(2); // noPadded = "Foo" (original string is never shortened)
```

Section 11.4: Getting x characters from the right side of a string

Visual Basic has Left, Right, and Mid functions that returns characters from the Left, Right, and Middle of a string. These methods does not exist in C#, but can be implemented with Substring(). They can be implemented as an extension methods like the following:

```
public static class StringExtensions
{
    /// <summary>
    /// VB Left function
    /// </summary>
    /// <param name="stringparam"></param>
    /// <param name="numchars"></param>
    /// <returns>Left-most numchars characters</returns>
    public static string Left( this string stringparam, int numchars )
    {
        // Handle possible Null or numeric stringparam being passed
        stringparam += string.Empty;

        // Handle possible negative numchars being passed
        numchars = Math.Abs( numchars );

        // Validate numchars parameter
        if (numchars > stringparam.Length)
            numchars = stringparam.Length;

        return stringparam.Substring( 0, numchars );
    }

    /// <summary>
    /// VB Right function
    /// </summary>
    /// <param name="stringparam"></param>
    /// <param name="numchars"></param>
    /// <returns>Right-most numchars characters</returns>
    public static string Right( this string stringparam, int numchars )
    {
        // Handle possible Null or numeric stringparam being passed
        stringparam += string.Empty;

        // Handle possible negative numchars being passed
        numchars = Math.Abs( numchars );

        // Validate numchars parameter
        if (numchars > stringparam.Length)
            numchars = stringparam.Length;

        return stringparam.Substring( stringparam.Length - numchars );
    }

    /// <summary>
    /// VB Mid function - to end of string
    /// </summary>
    /// <param name="stringparam"></param>
    /// <param name="startIndex">VB-Style startindex, 1st char startindex = 1</param>
    /// <returns>Balance of string beginning at startindex character</returns>
    public static string Mid( this string stringparam, int startIndex )
```

```

{
    // 处理可能传入的空值或数字字符串参数
    stringparam += string.Empty;

    // 处理可能传入的负数起始索引
    startindex = Math.Abs( startindex );

    // 验证 numchars 参数
    if (startindex > stringparam.Length)
        startindex = stringparam.Length;

    // C# 字符串基于零，转换传入的起始索引
    return stringparam.Substring( startindex - 1 );
}

/// <summary>
/// VB Mid 函数 - 指定字符数
/// </summary>
/// <param name="stringparam"></param>
/// <param name="startIndex">VB 风格起始索引，首字符起始索引 = 1</param>
/// <param name="numchars">返回的字符数</param>
/// <returns>从起始索引字符开始的字符串剩余部分</returns>
public static string Mid( this string stringparam, int startIndex, int numchars )
{
    // 处理可能传入的空值或数字字符串参数
    stringparam += string.Empty;

    // 处理可能传入的负数起始索引
    startindex = Math.Abs( startindex );

    // 处理可能传入的负数 numchars
    numchars = Math.Abs( numchars );

    // 验证 numchars 参数
    if (startIndex > stringparam.Length)
        startIndex = stringparam.Length;

    // C# 字符串基于零，转换传入的起始索引
    return stringparam.Substring( startIndex - 1, numchars );
}
}

```

此扩展方法可以按如下方式使用：

```

string myLongString = "Hello World!";
string myShortString = myLongString.Right(6); // "World!"
string myLeftString = myLongString.Left(5); // "Hello"
string myMidString1 = myLongString.Left(4); // "lo World"
string myMidString2 = myLongString.Left(2,3); // "ell"

```

第11.5节：使用 String.IsNullOrEmpty() 和 String.IsNullOrWhiteSpace() 检查空字符串

```

string nullString = null;
string emptyString = "";
string whitespaceString = "      ";
string tabString = "\t";
string newlineString = "\n";
string nonEmptyString = "abc123";

```

```

{
    // Handle possible Null or numeric stringparam being passed
    stringparam += string.Empty;

    // Handle possible negative startIndex being passed
    startindex = Math.Abs( startindex );

    // Validate numchars parameter
    if (startIndex > stringparam.Length)
        startIndex = stringparam.Length;

    // C# strings are zero-based, convert passed startIndex
    return stringparam.Substring( startIndex - 1 );
}

/// <summary>
/// VB Mid function - for number of characters
/// </summary>
/// <param name="stringparam"></param>
/// <param name="startIndex">VB-Style startIndex, 1st char startIndex = 1</param>
/// <param name="numchars">number of characters to return</param>
/// <returns>Balance of string beginning at startIndex character</returns>
public static string Mid( this string stringparam, int startIndex, int numchars )
{
    // Handle possible Null or numeric stringparam being passed
    stringparam += string.Empty;

    // Handle possible negative startIndex being passed
    startindex = Math.Abs( startIndex );

    // Handle possible negative numchars being passed
    numchars = Math.Abs( numchars );

    // Validate numchars parameter
    if (startIndex > stringparam.Length)
        startIndex = stringparam.Length;

    // C# strings are zero-based, convert passed startIndex
    return stringparam.Substring( startIndex - 1, numchars );
}
}

```

This extension method can be used as follows:

```

string myLongString = "Hello World!";
string myShortString = myLongString.Right(6); // "World!"
string myLeftString = myLongString.Left(5); // "Hello"
string myMidString1 = myLongString.Left(4); // "lo World"
string myMidString2 = myLongString.Left(2,3); // "ell"

```

Section 11.5: Checking for empty String using String.IsNullOrEmpty() and String.IsNullOrWhiteSpace()

```

string nullString = null;
string emptyString = "";
string whitespaceString = "      ";
string tabString = "\t";
string newlineString = "\n";
string nonEmptyString = "abc123";

```

```
bool result;
```

```
result = String.IsNullOrEmpty(nullString);           // true
result = String.IsNullOrEmpty(emptyString);         // true
result = String.IsNullOrEmpty(whitespaceString);    // false
result = String.IsNullOrEmpty(tabString);           // false
result = String.IsNullOrEmpty(newlineString);       // false
result = String.IsNullOrEmpty(nonEmptyString);      // false

result = String.IsNullOrWhiteSpace(nullString);      // true
result = String.IsNullOrWhiteSpace(emptyString);    // true
result = String.IsNullOrWhiteSpace(tabString);       // true
result = String.IsNullOrWhiteSpace(newlineString);   // true
result = String.IsNullOrWhiteSpace(whitespaceString); // true
result = String.IsNullOrWhiteSpace(nonEmptyString);  // false
```

第11.6节：修剪字符串开头和/或结尾的不需要字符

String.Trim()

```
string x = "Hello World! ";
string y = x.Trim(); // "Hello World!"

string q = "{(Hi*";
string r = q.Trim( '(', '*', '{' ); // "Hi!"
```

String.TrimStart() 和 String.TrimEnd()

```
string q = "{(Hi*";
string r = q.TrimStart( '{' ); // "(Hi*"
string s = q.TrimEnd( '*' ); // "{(Hi"
```

第11.7节：将十进制数转换为二进制、八进制和十六进制格式

1. 将十进制数转换为二进制格式使用基数2

```
Int32 数字 = 15;
Console.WriteLine(Convert.ToString(数字, 2)); //输出 : 1111
```

2. 将十进制数转换为八进制格式使用基数8

```
int 数字 = 15;
Console.WriteLine(Convert.ToString(数字, 8)); //输出 : 17
```

3. 将十进制数转换为十六进制格式时使用基数16

```
var Number = 15;
Console.WriteLine(Convert.ToString(Number, 16)); //输出 : f
```

第11.8节：从数组构造字符串

`String.Join` 方法将帮助我们从字符或字符串的数组/列表构造字符串。该方法接受两个参数。第一个是分隔符或分隔符，它将帮助你分隔数组中的每个元素。第二个参数是数组本身。

```
bool result;
```

```
result = String.IsNullOrEmpty(nullString);           // true
result = String.IsNullOrEmpty(emptyString);         // true
result = String.IsNullOrEmpty(whitespaceString);    // false
result = String.IsNullOrEmpty(tabString);           // false
result = String.IsNullOrEmpty(newlineString);       // false
result = String.IsNullOrEmpty(nonEmptyString);      // false

result = String.IsNullOrWhiteSpace(nullString);      // true
result = String.IsNullOrWhiteSpace(emptyString);    // true
result = String.IsNullOrWhiteSpace(tabString);       // true
result = String.IsNullOrWhiteSpace(newlineString);   // true
result = String.IsNullOrWhiteSpace(whitespaceString); // true
result = String.IsNullOrWhiteSpace(nonEmptyString);  // false
```

Section 11.6: Trimming Unwanted Characters Off the Start and/or End of Strings

String.Trim()

```
string x = "Hello World! ";
string y = x.Trim(); // "Hello World!"

string q = "{(Hi*";
string r = q.Trim( '(', '*', '{' ); // "Hi!"
```

String.TrimStart() and String.TrimEnd()

```
string q = "{(Hi*";
string r = q.TrimStart( '{' ); // "(Hi*"
string s = q.TrimEnd( '*' ); // "{(Hi"
```

Section 11.7: Convert Decimal Number to Binary,Octal and Hexadecimal Format

1. To convert decimal number to binary format use **base 2**

```
Int32 Number = 15;
Console.WriteLine(Convert.ToString(Number, 2)); //OUTPUT : 1111
```

2. To convert decimal number to octal format use **base 8**

```
int Number = 15;
Console.WriteLine(Convert.ToString(Number, 8)); //OUTPUT : 17
```

3. To convert decimal number to hexadecimal format use **base 16**

```
var Number = 15;
Console.WriteLine(Convert.ToString(Number, 16)); //OUTPUT : f
```

Section 11.8: Construct a string from Array

The `String.Join` method will help us to construct a string From array/list of characters or string. This method accepts two parameters. The first one is the delimiter or the separator which will help you to separate each element in the array. And the second parameter is the Array itself.

从char数组构造字符串：

```
string delimiter="";
char[] charArray = new[] { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charArray);
```

输出：a,b,c 如果我们将delimiter改为""，则输出将变为abc。

从List of char构造字符串：

```
string delimiter = "|";
List<char> charList = new List<char>() { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charList);
```

输出：a|b|c

来自字符串列表的字符串：

```
string delimiter = " ";
List<string> stringList = new List<string>() { "Ram", "is", "a", "boy" };
string inputString = String.Join(delimiter, stringList);
```

输出：Ram is a boy

来自字符串数组的字符串：

```
string delimiter = "_";
string[] stringArray = new [] { "Ram", "is", "a", "boy" };
string inputString = String.Join(delimiter, stringArray);
```

输出：Ram_is_a_boy

第11.9节：使用ToString进行格式化

通常我们使用String.Format方法来进行格式化，.ToString通常用于将其他类型转换为字符串。我们可以在转换时与ToString方法一起指定格式，这样就可以避免额外的格式化。让我来解释它如何与不同类型一起工作；

整数转格式化字符串：

```
int intValue = 10;
string zeroPaddedInteger = intValue.ToString("000"); // 输出将是 "010"
string customFormat = intValue.ToString("Input value is 0"); // 输出将是 "Input value is 10"
```

双精度数转格式化字符串：

```
double doubleValue = 10.456;
string roundedDouble = doubleValue.ToString("0.00"); // 输出 10.46
string integerPart = doubleValue.ToString("00"); // 输出 10
string customFormat = doubleValue.ToString("Input value is 0.0"); // Input value is 10.5
```

使用 ToString 格式化日期时间

```
DateTime currentDate = DateTime.Now; // {2016/7/21 19:23:15}
string dateTimeString = currentDate.ToString("dd-MM-yyyy HH:mm:ss"); // "21-07-2016 19:23:15"
```

String from char array:

```
string delimiter="";
char[] charArray = new[] { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charArray);
```

Output: a,b,c if we change the delimiter as "" then the output will become abc.

String from List of char:

```
string delimiter = "|";
List<char> charList = new List<char>() { 'a', 'b', 'c' };
string inputString = String.Join(delimiter, charList);
```

Output: a|b|c

String from List of Strings:

```
string delimiter = " ";
List<string> stringList = new List<string>() { "Ram", "is", "a", "boy" };
string inputString = String.Join(delimiter, stringList);
```

Output: Ram is a boy

String from array of strings:

```
string delimiter = "_";
string[] stringArray = new [] { "Ram", "is", "a", "boy" };
string inputString = String.Join(delimiter, stringArray);
```

Output: Ram_is_a_boy

Section 11.9: Formatting using ToString

Usually we are using `String.Format` method for formatting purpose, the `.ToString` is usually used for converting other types to string. We can specify the format along with the `ToString` method while conversion is taking place, So we can avoid an additional Formatting. Let Me Explain how it works with different types;

Integer to formatted string:

```
int intValue = 10;
string zeroPaddedInteger = intValue.ToString("000"); // Output will be "010"
string customFormat = intValue.ToString("Input value is 0"); // output will be "Input value is 10"
```

double to formatted string:

```
double doubleValue = 10.456;
string roundedDouble = doubleValue.ToString("0.00"); // output 10.46
string integerPart = doubleValue.ToString("00"); // output 10
string customFormat = doubleValue.ToString("Input value is 0.0"); // Input value is 10.5
```

Formatting DateTime using ToString

```
DateTime currentDate = DateTime.Now; // {7/21/2016 7:23:15 PM}
string dateTimeString = currentDate.ToString("dd-MM-yyyy HH:mm:ss"); // "21-07-2016 19:23:15"
```

```
string dateOnlyString = currentDate.ToString("dd-MM-yyyy"); // "21-07-2016"
string dateWithMonthInWords = currentDate.ToString("dd-MMMM-yyyy HH:mm:ss"); // "21-July-2016
19:23:15"
```

第11.10节：按另一个字符串拆分字符串

```
string str = "this--is--a--complete--sentence";
string[] tokens = str.Split(new[] { "--" }, StringSplitOptions.None);
```

结果：

```
[ "this", "is", "a", "complete", "sentence" ]
```

第11.11节：按特定字符拆分字符串

```
string helloWorld = "hello world, how is it going?";
string[] parts1 = helloWorld.Split(',');

//parts1: ["hello world", " how is it going?"]

string[] parts2 = helloWorld.Split(' ');

//parts2: ["hello", "world,", "how", "is", "it", "going?"]
```

第11.12节：获取给定字符串的子字符串

```
string helloWorld = "Hello World!";
string world = helloWorld.Substring(6); //world = "World!"
string hello = helloWorld.Substring(0,5); // hello = "Hello"
```

Substring 返回从给定索引开始的字符串，或两个索引之间的字符串（两端均包含）。

第11.13节：确定字符串是否以给定序列开始

```
string HelloWorld = "Hello World";
HelloWorld.StartsWith("Hello"); // true
HelloWorld.StartsWith("Foo"); // false
```

在字符串中查找字符串

使用System.String.Contains可以判断某个字符串是否存在另一个字符串中。该方法返回一个布尔值，若字符串存在则为true，否则为false。

```
string s = "Hello World";
bool stringExists = s.Contains("ello"); //stringExists =true 因为字符串包含子字符串
```

第11.14节：获取特定索引处的字符并枚举字符串

您可以使用Substring方法从字符串的任意位置获取任意数量的字符。但是，如果您只想要单个字符，可以使用字符串索引器在任意索引处获取单个字符，就像您

```
string dateOnlyString = currentDate.ToString("dd-MM-yyyy"); // "21-07-2016"
string dateWithMonthInWords = currentDate.ToString("dd-MMMM-yyyy HH:mm:ss"); // "21-July-2016
19:23:15"
```

Section 11.10: Splitting a String by another string

```
string str = "this--is--a--complete--sentence";
string[] tokens = str.Split(new[] { "--" }, StringSplitOptions.None);
```

Result:

```
[ "this", "is", "a", "complete", "sentence" ]
```

Section 11.11: Splitting a String by specific character

```
string helloWorld = "hello world, how is it going?";
string[] parts1 = helloWorld.Split(',');

//parts1: ["hello world", " how is it going?"]

string[] parts2 = helloWorld.Split(' ');

//parts2: ["hello", "world,", "how", "is", "it", "going?"]
```

Section 11.12: Getting Substrings of a given string

```
string helloWorld = "Hello World!";
string world = helloWorld.Substring(6); //world = "World!"
string hello = helloWorld.Substring(0,5); // hello = "Hello"
```

Substring returns the string up from a given index, or between two indexes (both inclusive).

Section 11.13: Determine whether a string begins with a given sequence

```
string HelloWorld = "Hello World";
HelloWorld.StartsWith("Hello"); // true
HelloWorld.StartsWith("Foo"); // false
```

Finding a string within a string

Using the System.String.Contains you can find out if a particular string exists within a string. The method returns a boolean, true if the string exists else false.

```
string s = "Hello World";
bool stringExists = s.Contains("ello"); //stringExists =true as the string contains the substring
```

Section 11.14: Getting a char at specific index and enumerating the string

You can use the Substring method to get any number of characters from a string at any given location. However, if you only want a single character, you can use the string indexer to get a single character at any given index like you

与数组相关：

```
string s = "hello";
char c = s[1]; //返回 'e'
```

注意返回类型是char，不同于返回string类型的Substring方法。

你也可以使用索引器遍历字符串中的字符：

```
string s = "hello";
foreach (char c in s)
    Console.WriteLine(c);
***** 这将把每个字符打印在新的一行:
```

do with an array:

```
string s = "hello";
char c = s[1]; //Returns 'e'
```

Notice that the return type is `char`, unlike the `Substring` method which returns a `string` type.

You can also use the indexer to iterate through the characters of the string:

```
string s = "hello";
foreach (char c in s)
    Console.WriteLine(c);
***** This will print each character on a new line:
h
e
l
l
o
*****
```

Section 11.15: Joining an array of strings into a new one

```
var parts = new[] { "Foo", "Bar", "Fizz", "Buzz" };
var joined = string.Join(", ", parts);

//joined = "Foo, Bar, Fizz, Buzz"
```

Section 11.16: Replacing a string within a string

Using the `System.String.Replace` method, you can replace part of a string with another string.

```
string s = "Hello World";
s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

All the occurrences of the search string are replaced.

This method can also be used to remove part of a string, using the `String.Empty` field:

```
string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"
```

Section 11.17: Changing the case of characters within a String

The `System.String` class supports a number of methods to convert between uppercase and lowercase characters in a string.

- `System.String.ToLowerInvariant` is used to return a String object converted to lowercase.
- `System.String.ToUpperInvariant` is used to return a String object converted to uppercase.

Note: The reason to use the *invariant* versions of these methods is to prevent producing unexpected culture-specific letters. This is explained [here in detail](#).

Example:

```
string s = "My String";
```

```
s = s.ToLowerInvariant(); // "my string"  
s = s.ToUpperInvariant(); // "MY STRING"
```

注意，你可以选择在转换大小写时指定特定的[Culture](#)，方法是分别使用[String.ToLower\(CultureInfo\)](#)和[String.ToUpper\(CultureInfo\)](#)方法。

第11.18节：将字符串数组连接成一个单一的字符串

[System.String.Join](#)方法允许使用指定的分隔符将字符串数组中的所有元素连接起来，分隔符位于每个元素之间：

```
string[] words = {"One", "Two", "Three", "Four"};  
string singleString = String.Join(", ", words); // singleString = "One, Two, Three, Four"
```

第11.19节：字符串连接

字符串连接可以使用[System.String.Concat](#)方法，或者（更简单地）使用+运算符：

```
string first = "Hello ";  
string second = "World";  
  
string concat = first + second; // concat = "Hello World"  
concat = String.Concat(first, second); // concat = "Hello World"
```

在C# 6中可以这样做：

```
string concat = $"{first},{second}";
```

```
s = s.ToLowerInvariant(); // "my string"  
s = s.ToUpperInvariant(); // "MY STRING"
```

Note that you *can* choose to specify a specific [Culture](#) when converting to lowercase and uppercase by using the [String.ToLower\(CultureInfo\)](#) and [String.ToUpper\(CultureInfo\)](#) methods accordingly.

Section 11.18: Concatenate an array of strings into a single string

The [System.String.Join](#) method allows to concatenate all elements in a string array, using a specified separator between each element:

```
string[] words = {"One", "Two", "Three", "Four"};  
string singleString = String.Join(", ", words); // singleString = "One, Two, Three, Four"
```

Section 11.19: String Concatenation

String Concatenation can be done by using the [System.String.Concat](#) method, or (much easier) using the + operator:

```
string first = "Hello ";  
string second = "World";  
  
string concat = first + second; // concat = "Hello World"  
concat = String.Concat(first, second); // concat = "Hello World"
```

In C# 6 this can be done as follows:

```
string concat = $"{first},{second}";
```

第12章：String.Format

参数	详细信息
格式化	一个复合格式字符串，定义了args应如何组合成字符串。
参数	要组合成字符串的一系列对象。由于使用了params参数，您可以使用逗号分隔的参数列表或实际的对象数组。
提供程序	将对象格式化为字符串的方法集合。典型值包括CultureInfo.InvariantCulture和CultureInfo.CurrentCulture。

Format方法是一组System.String类中的重载，用于创建将对象组合成特定字符串表示形式的字符串。此信息可应用于String.Format、各种WriteLine方法以及.NET框架中的其他方法。

第12.1节：自C# 6.0起

版本 ≥ 6.0

自C# 6.0起，可以使用字符串插值代替String.Format。

```
string name = "John";
string lastname = "Doe";
Console.WriteLine($"Hello {name} {lastname}!");
```

Hello John Doe!

关于此主题 C# 6.0 功能的更多示例：字符串插值。

第12.2节：String.Format在框架中“嵌入”的位置

有几个地方可以间接使用String.Format：关键是寻找签名名为string format, params object[] args的重载，例如：

```
Console.WriteLine(String.Format("{0} - {1}", name, value));
```

可以替换为更简短的版本：

```
Console.WriteLine("{0} - {1}", name, value);
```

还有其他方法也使用String.Format，例如：

```
Debug.WriteLine(); // 和 Print()
StringBuilder.AppendFormat();
```

第12.3节：创建自定义格式提供器

```
public class CustomFormat : IFormatProvider, ICustomFormatter
{
    public string Format(string format, object arg, IFormatProvider formatProvider)
    {
        if (!this.Equals(formatProvider))
```

Chapter 12: String.Format

Parameter	Details
format	A composite format string , which defines the way args should be combined into a string.
args	A sequence of objects to be combined into a string. Since this uses a params argument, you can either use a comma-separated list of arguments or an actual object array.
provider	A collection of ways of formatting objects to strings. Typical values include CultureInfo.InvariantCulture and CultureInfo.CurrentCulture .

The Format methods are a set of [overloads](#) in the [System.String](#) class used to create strings that combine objects into specific string representations. This information can be applied to [String.Format](#), various WriteLine methods as well as other methods in the .NET framework.

Section 12.1: Since C# 6.0

Version ≥ 6.0

Since C# 6.0 it is possible to use string interpolation in place of [String.Format](#).

```
string name = "John";
string lastname = "Doe";
Console.WriteLine($"Hello {name} {lastname}!");
```

Hello John Doe!

More examples for this under the topic C# 6.0 features: String interpolation.

Section 12.2: Places where String.Format is 'embedded' in the framework

There are several places where you can use [String.Format](#) *indirectly*: The secret is to look for the overload with the signature `string format, params object[] args`, e.g.:

```
Console.WriteLine(String.Format("{0} - {1}", name, value));
```

Can be replaced with shorter version:

```
Console.WriteLine("{0} - {1}", name, value);
```

There are other methods which also use [String.Format](#), e.g.:

```
Debug.WriteLine(); // and Print()
StringBuilder.AppendFormat();
```

Section 12.3: Create a custom format provider

```
public class CustomFormat : IFormatProvider, ICustomFormatter
{
    public string Format(string format, object arg, IFormatProvider formatProvider)
    {
        if (!this.Equals(formatProvider))
```

```

    {
        return null;
    }

    if (format == "Reverse")
    {
        return String.Join("", arg.ToString().Reverse());
    }

    return arg.ToString();
}

public object GetFormat(Type formatType)
{
    return formatType==typeof(ICustomFormatter) ? this:null;
}

```

用法：

```
String.Format(new CustomFormat(), "-> {0:Reverse} <-", "Hello World");
```

输出：

```
-> dlroW olleH <-
```

第12.4节：日期格式化

```
DateTime date = new DateTime(2016, 07, 06, 18, 30, 14);
```

// 格式：年，月，日，小时，分钟，秒

```
Console.WriteLine(String.Format("{0:dd}",date));
```

// 按文化信息格式化

```
String.Format(new System.Globalization.CultureInfo("mn-MN"), "{0:dddd}",date);
```

版本 ≥ 6.0

```
Console.WriteLine($"{date:ddd}");
```

输出：

```
0  
6  
Лхагва  
06
```

格式说明符	含义	样本	结果
d	日期	{0:d}	2016/7/6
dd	日，补零	{0:dd}	06
ddd	简短的星期名称	{0:ddd}	周三
dddd	完整的星期名称	{0:dddd}	星期三
D	长日期	{0:D}	2016年7月6日，星期三
f	完整日期和时间，简短格式	{0:f}	2016年7月6日星期三 下午6:30
ff	秒的小数部分，2位数字	{0:ff}	20
fff	秒的小数部分，3位数字	{0:fff}	201
ffff	秒的小数部分，4位数字	{0:ffff}	2016
F	完整日期和时间，长格式	{0:F}	2016年7月6日，星期三 下午6:30:14

```

    {
        return null;
    }

    if (format == "Reverse")
    {
        return String.Join("", arg.ToString().Reverse());
    }

    return arg.ToString();
}

public object GetFormat(Type formatType)
{
    return formatType==typeof(ICustomFormatter) ? this:null;
}

```

Usage:

```
String.Format(new CustomFormat(), "-> {0:Reverse} <-", "Hello World");
```

Output:

```
-> dlroW olleH <-
```

Section 12.4: Date Formatting

```
DateTime date = new DateTime(2016, 07, 06, 18, 30, 14);
// Format: year, month, day hours, minutes, seconds
```

```
Console.WriteLine(String.Format("{0:dd}",date));
```

//Format by Culture info

```
String.Format(new System.Globalization.CultureInfo("mn-MN"), "{0:dddd}",date);
```

Version ≥ 6.0

```
Console.WriteLine($"{date:ddd}");
```

output :

```
06  
Лхагва  
06
```

Specifier	Meaning	Sample	Result
d	Date	{0:d}	7/6/2016
dd	Day, zero-padded	{0:dd}	06
ddd	Short day name	{0:ddd}	Wed
dddd	Full day name	{0:dddd}	Wednesday
D	Long date	{0:D}	Wednesday, July 6, 2016
f	Full date and time, short	{0:f}	Wednesday, July 6, 2016 6:30 PM
ff	Second fractions, 2 digits	{0:ff}	20
fff	Second fractions, 3 digits	{0:fff}	201
ffff	Second fractions, 4 digits	{0:ffff}	2016
F	Full date and time, long	{0:F}	Wednesday, July 6, 2016 6:30:14 PM

g	默认日期和时间	{0:g}	2016年7月6日 下午6:30
gg	时代	{0:gg}	公元
hh	小时 (2位, 12小时制)	{0:hh}	06
HH	小时 (2位, 24小时制)	{0:HH}	18
M	月份和日期	{0:M}	7月6日
mm	分钟, 补零	{0:mm}	30
MM	月份, 补零	{0:MM}	07
MMM	三字母月份名称	{0:MMM}	七月
MMMM	完整的月份名称	{0:MMMM}	7月
秒	秒	{0:ss}	14
r	RFC1123 日期	{0:r}	周三, 2016年7月6日 18:30:14 GMT
s	可排序的日期字符串	{0:s}	2016-07-06T18:30:14
t	短时间	{0:t}	下午6:30
T	长时间	{0:T}	下午6:30:14
tt	上午/下午	{0:tt}	下午
u	通用可排序本地时间 {0:u}	{0:u}	2016-07-06 18:30:14Z
U	通用格林威治时间	{0:U}	2016年7月6日星期三 上午9:30:14
Y	月份和年份	{0:Y}	2016年7月
yy	两位数年份	{0:yy}	16
yyyy	4位数年份	{0:yyyy}	2016
zz	2位数时区偏移	{0:zz}	+09
zzz	完整时区偏移	{0:zzz}	+09:00

g	Default date and time	{0:g}	7/6/2016 6:30 PM
gg	Era	{0:gg}	A.D
hh	Hour (2 digits, 12H)	{0:hh}	06
HH	Hour (2 digits, 24H)	{0:HH}	18
M	Month and day	{0:M}	July 6
mm	Minutes, zero-padded	{0:mm}	30
MM	Month, zero-padded	{0:MM}	07
MMM	3-letter month name	{0:MMM}	Jul
MMMM	Full month name	{0:MMMM}	July
ss	Seconds	{0:ss}	14
r	RFC1123 date	{0:r}	Wed, 06 Jul 2016 18:30:14 GMT
s	Sortable date string	{0:s}	2016-07-06T18:30:14
t	Short time	{0:t}	6:30 PM
T	Long time	{0:T}	6:30:14 PM
tt	AM/PM	{0:tt}	PM
u	Universal sortable local time {0:u}	{0:u}	2016-07-06 18:30:14Z
U	Universal GMT	{0:U}	Wednesday, July 6, 2016 9:30:14 AM
Y	Month and year	{0:Y}	July 2016
yy	2 digit year	{0:yy}	16
yyyy	4 digit year	{0:yyyy}	2016
zz	2 digit timezone offset	{0:zz}	+09
zzz	full time zone offset	{0:zzz}	+09:00

第12.5节：货币格式化

“c”（或货币）格式说明符将数字转换为表示货币金额的字符串。

```
string.Format("{0:c}", 112.236677) // $112.23 - 默认为系统设置
```

精度

默认值为2。使用c1、c2、c3等来控制精度。

```
string.Format("{0:C1}", 112.236677) // $112.2
string.Format("{0:C3}", 112.236677) // $112.237
string.Format("{0:C4}", 112.236677) // $112.2367
string.Format("{0:C9}", 112.236677) // $112.236677000
```

货币符号

1. 传入CultureInfo实例以使用自定义文化符号。

```
string.Format(new CultureInfo("en-US"), "{0:c}", 112.236677); // $112.24
string.Format(new CultureInfo("de-DE"), "{0:c}", 112.236677); // 112,24 €
string.Format(new CultureInfo("hi-IN"), "{0:c}", 112.236677); // 112.24
```

2. 使用任意字符串作为货币符号。使用NumberFormatInfo来自定义货币符号。

```
NumberFormatInfo nfi = new CultureInfo("en-US", false).NumberFormat;
nfi = (NumberFormatInfo) nfi.Clone();
nfi.CurrencySymbol = "?";
```

Section 12.5: Currency Formatting

The "c" (or currency) format specifier converts a number to a string that represents a currency amount.

```
string.Format("{0:c}", 112.236677) // $112.23 - defaults to system
```

Precision

Default is 2. Use c1, c2, c3 and so on to control precision.

```
string.Format("{0:C1}", 112.236677) // $112.2
string.Format("{0:C3}", 112.236677) // $112.237
string.Format("{0:C4}", 112.236677) // $112.2367
string.Format("{0:C9}", 112.236677) // $112.236677000
```

Currency Symbol

1. Pass CultureInfo instance to use custom culture symbol.

```
string.Format(new CultureInfo("en-US"), "{0:c}", 112.236677); // $112.24
string.Format(new CultureInfo("de-DE"), "{0:c}", 112.236677); // 112,24 €
string.Format(new CultureInfo("hi-IN"), "{0:c}", 112.236677); // □ 112.24
```

2. Use any string as currency symbol. Use NumberFormatInfo as to customize currency symbol.

```
NumberFormatInfo nfi = new CultureInfo("en-US", false).NumberFormat;
nfi = (NumberFormatInfo) nfi.Clone();
nfi.CurrencySymbol = "?";
```

```
string.Format(nfi, "{0:C}", 112.236677); //?112.24  
nfi.CurrencySymbol = "?%^&";  
string.Format(nfi, "{0:C}", 112.236677); //?%^&112.24
```

货币符号的位置

正数使用CurrencyPositivePattern，负数使用CurrencyNegativePattern。

```
NumberFormatInfo nfi = new CultureInfo("en-US", false).NumberFormat;  
nfi.CurrencyPositivePattern = 0;  
string.Format(nfi, "{0:C}", 112.236677); //?112.24 - 默认  
nfi.CurrencyPositivePattern = 1;  
string.Format(nfi, "{0:C}", 112.236677); //112.24$  
nfi.CurrencyPositivePattern = 2;  
string.Format(nfi, "{0:C}", 112.236677); //$. 112.24  
nfi.CurrencyPositivePattern = 3;  
string.Format(nfi, "{0:C}", 112.236677); //112.24 $
```

负数格式的用法与正数格式相同。更多用例请参考原始链接。

自定义小数分隔符

```
NumberFormatInfo nfi = new CultureInfo("en-US", false).NumberFormat;  
nfi.CurrencyPositivePattern = 0;  
nfi.CurrencyDecimalSeparator = "..";  
string.Format(nfi, "{0:C}", 112.236677); //?112..24
```

第12.6节：使用自定义数字格式

NumberFormatInfo 可用于格式化整数和浮点数。

```
// invariantResult 是 "1,234,567.89"  
var invariantResult = string.Format(CultureInfo.InvariantCulture, "{0:#,###,##}", 1234567.89);  
  
// NumberFormatInfo 是实现 IFormatProvider 的类之一  
var customProvider = new NumberFormatInfo  
{  
    NumberDecimalSeparator = "_NS_", // 将替代 ','  
    NumberGroupSeparator = "_GS_", // 将替代 '.'  
};  
  
// customResult 是 "1_GS_234_GS_567_NS_89"  
var customResult = string.Format(customProvider, "{0:#,###,##}", 1234567.89);
```

第12.7节：左对齐/右对齐，用空格填充

大括号中的第二个值决定替换字符串的长度。通过调整第二个值为正或负，可以改变字符串的对齐方式。

```
string.Format("左对齐: 字符串: ->{0,-5}<- 整数: ->{1,-5}<-", "abc", 123);  
string.Format("右对齐: 字符串: ->{0,5}<- 整数: ->{1,5}<-", "abc", 123);
```

输出：

```
左对齐: 字符串: ->abc <- 整数: ->123 <-  
右对齐: 字符串: -> abc<- 整数: -> 123<-
```

```
string.Format(nfi, "{0:C}", 112.236677); //?112.24  
nfi.CurrencySymbol = "?%^&";  
string.Format(nfi, "{0:C}", 112.236677); //?%^&112.24
```

Position of Currency Symbol

Use [CurrencyPositivePattern](#) for positive values and [CurrencyNegativePattern](#) for negative values.

```
NumberFormatInfo nfi = new CultureInfo("en-US", false).NumberFormat;  
nfi.CurrencyPositivePattern = 0;  
string.Format(nfi, "{0:C}", 112.236677); //?112.24 - default  
nfi.CurrencyPositivePattern = 1;  
string.Format(nfi, "{0:C}", 112.236677); //112.24$  
nfi.CurrencyPositivePattern = 2;  
string.Format(nfi, "{0:C}", 112.236677); //$. 112.24  
nfi.CurrencyPositivePattern = 3;  
string.Format(nfi, "{0:C}", 112.236677); //112.24 $
```

Negative pattern usage is the same as positive pattern. A lot more use cases please refer to original link.

Custom Decimal Separator

```
NumberFormatInfo nfi = new CultureInfo("en-US", false).NumberFormat;  
nfi.CurrencyPositivePattern = 0;  
nfi.CurrencyDecimalSeparator = "..";  
string.Format(nfi, "{0:C}", 112.236677); //?112..24
```

Section 12.6: Using custom number format

NumberFormatInfo 可以用于格式化整数和浮点数。

```
// invariantResult 是 "1,234,567.89"  
var invariantResult = string.Format(CultureInfo.InvariantCulture, "{0:#,###,##}", 1234567.89);  
  
// NumberFormatInfo 是实现 IFormatProvider 的类之一  
var customProvider = new NumberFormatInfo  
{  
    NumberDecimalSeparator = "_NS_", // 将替代 ','  
    NumberGroupSeparator = "_GS_", // 将替代 '.'  
};  
  
// customResult 是 "1_GS_234_GS_567_NS_89"  
var customResult = string.Format(customProvider, "{0:#,###,##}", 1234567.89);
```

Section 12.7: Align left/ right, pad with spaces

The second value in the curly braces dictates the length of the replacement string. By adjusting the second value to be positive or negative, the alignment of the string can be changed.

```
string.Format("LEFT: string: ->{0,-5}<- int: ->{1,-5}<-", "abc", 123);  
string.Format("RIGHT: string: ->{0,5}<- int: ->{1,5}<-", "abc", 123);
```

Output:

```
LEFT: string: ->abc <- int: ->123 <-  
RIGHT: string: -> abc<- int: -> 123<-
```

第12.8节：数字格式

```
// 整数类型的十六进制表示  
string.Format("十六进制: byte2: {0:x2}; byte4: {0:X4}; 字符: {1:x2}", 123, (int)'A');  
  
// 带千位分隔符的整数  
string.Format("整数, 千位分隔符: {0:#,#}; 固定长度: >{0,10:#,#}<", 1234567);  
  
// 带前导零的整数  
string.Format("整数, 前导零: {0:00}; ", 1);  
  
// 小数  
string.Format("十进制, 固定精度:{0:0.000}; 百分比格式:{0:0.00%}", 0.12);
```

输出：

```
十六进制: byte2: 7b; byte4: 007B; 字符: 41  
整数, 千位分隔符: 1,234,567; 固定长度: > 1,234,567<  
整数, 前导零: 01;  
十进制, 固定精度: 0.120; 作为百分比: 12.00%
```

第12.9节：ToString()

ToString() 方法存在于所有引用对象类型中。这是因为所有引用类型都派生自 Object，而 Object 上有 ToString() 方法。对象基类上的 ToString() 方法返回类型名称。下面的代码片段将打印“User”到控制台。

```
public class 用户  
{  
    public string 名称 { get; set; }  
    public int 编号 { get; set; }  
}  
  
...  
  
var 用户 = new 用户 {名称 = "User1", 编号 = 5};  
Console.WriteLine(用户.ToString());
```

然而，User 类也可以重写 ToString() 方法以更改其返回的字符串。下面的代码片段将打印“Id: 5, Name: User1”到控制台。

```
public class 用户  
{  
    public string 名称 { get; set; }  
    public int 编号 { get; set; }  
    public override ToString()  
    {  
        return string.Format("Id: {0}, Name: {1}", 编号, 名称);  
    }  
}  
  
...  
  
var 用户 = new 用户 {名称 = "User1", 编号 = 5};  
Console.WriteLine(用户.ToString());
```

Section 12.8: Numeric formats

```
// Integral types as hex  
string.Format("Hexadecimal: byte2: {0:x2}; byte4: {0:X4}; char: {1:x2}", 123, (int)'A');  
  
// Integers with thousand separators  
string.Format("Integer, thousand sep.: {0:#,#}; fixed length: >{0,10:#,#}<", 1234567);  
  
// Integer with leading zeroes  
string.Format("Integer, leading zeroes: {0:00}; ", 1);  
  
// Decimals  
string.Format("Decimal, fixed precision: {0:0.000}; as percents: {0:0.00%}", 0.12);
```

Output:

```
Hexadecimal: byte2: 7b; byte4: 007B; char: 41  
Integer, thousand sep.: 1,234,567; fixed length: > 1,234,567<  
Integer, leading zeroes: 01;  
Decimal, fixed precision: 0.120; as percents: 12.00%
```

Section 12.9: ToString()

The ToString() method is present on all reference object types. This is due to all reference types being derived from Object which has the ToString() method on it. The ToString() method on the object base class returns the type name. The fragment below will print out "User" to the console.

```
public class User  
{  
    public string Name { get; set; }  
    public int Id { get; set; }  
}  
  
...  
  
var user = new User {Name = "User1", Id = 5};  
Console.WriteLine(user.ToString());
```

However, the class User can also override ToString() in order to alter the string it returns. The code fragment below prints out "Id: 5, Name: User1" to the console.

```
public class User  
{  
    public string Name { get; set; }  
    public int Id { get; set; }  
    public override ToString()  
    {  
        return string.Format("Id: {0}, Name: {1}", Id, Name);  
    }  
}  
  
...  
  
var user = new User {Name = "User1", Id = 5};  
Console.WriteLine(user.ToString());
```

第12.10节：在String.Format()表达式中转义大括号

```
string outsidetext = "我在括号外面";
string.Format("{{我在括号里！}} {0}", outsidetext);

//输出 "{我在括号里！} 我在括号外面"
```

第12.11节：与ToString()的关系

虽然String.Format()方法在将数据格式化为字符串时确实很有用，但在处理单个对象时，正如下面所示，它可能有些过于复杂：

```
String.Format("{0:C}", money); // 结果为 "$42.00"
```

更简单的方法可能是直接使用C#中所有对象都支持的ToString()方法。它支持所有相同的标准和自定义格式字符串，但不需要参数映射，因为只有一个参数：

```
money.ToString("C"); // 结果为 "$42.00"
```

注意事项与格式限制

虽然这种方法在某些情况下更简单，但ToString()方法在添加左侧或右侧填充方面有限制，而这在String.Format()方法中是可以实现的：

```
String.Format("{0,10:C}", money); // 结果为 "    $42.00"
```

为了使用ToString()方法实现相同的行为，您需要分别使用另一个方法如PadLeft()或PadRight()：

```
money.ToString("C").PadLeft(10); // 结果为 "    $42.00"
```

Section 12.10: Escaping curly brackets inside a String.Format() expression

```
string outsidetext = "I am outside of bracket";
string.Format("{{I am in brackets!}} {0}", outsidetext);

//Outputs "I am in brackets! I am outside of bracket"
```

Section 12.11: Relationship with ToString()

While the `String.Format()` method is certainly useful in formatting data as strings, it may often be a bit overkill, especially when dealing with a single object as seen below :

```
String.Format("{0:C}", money); // yields "$42.00"
```

An easier approach might be to simply use the `ToString()` method available on all objects within C#. It supports all of the same [standard and custom formatting strings](#), but doesn't require the necessary parameter mapping as there will only be a single argument :

```
money.ToString("C"); // yields "$42.00"
```

Caveats & Formatting Restrictions

While this approach may be simpler in some scenarios, the `ToString()` approach is limited with regards to adding left or right padding like you might do within the `String.Format()` method :

```
String.Format("{0,10:C}", money); // yields "    $42.00"
```

In order to accomplish this same behavior with the `ToString()` method, you would need to use another method like `PadLeft()` or `PadRight()` respectively :

```
money.ToString("C").PadLeft(10); // yields "    $42.00"
```

第13章：字符串连接

第13.1节：+ 运算符

```
string s1 = "string1";
string s2 = "string2";

string s3 = s1 + s2; // "string1string2"
```

第13.2节：使用 System.Text.StringBuilder连接字符串

使用StringBuilder连接字符串相比简单的+连接可以提供性能优势。这是由于内存分配的方式不同。字符串在每次连接时都会重新分配内存，而StringBuilder则以块为单位分配内存，仅在当前块用尽时才重新分配。这在进行大量小规模连接时可以带来显著差异。

```
StringBuilder sb = new StringBuilder();
for (int i = 1; i <= 5; i++)
{
    sb.Append(i);
    sb.Append(" ");
}
Console.WriteLine(sb.ToString()); // "1 2 3 4 5 "
```

对Append()的调用可以链式调用，因为它返回对StringBuilder的引用：

```
StringBuilder sb = new StringBuilder();
sb.Append("some string ")
    .Append("another string");
```

第13.3节：使用String.Join连接字符串数组元素

可以使用String.Join方法连接字符串数组中的多个元素。

```
string[] value = {"apple", "orange", "grape", "pear"};
string separator = ", ";

string result = String.Join(separator, value, 1, 2);
Console.WriteLine(result);
```

输出结果为："orange, grape"

此示例使用了String.Join(String, String[], Int32, Int32)重载，除了分隔符和数组外，还指定了起始索引和数量。

如果不使用startIndex和count重载，可以连接所有给定的字符串。示例如下：

```
string[] value = {"apple", "orange", "grape", "pear"};
string separator = ", ";
string result = String.Join(separator, value);
Console.WriteLine(result);
```

Chapter 13: String Concatenate

Section 13.1: + Operator

```
string s1 = "string1";
string s2 = "string2";

string s3 = s1 + s2; // "string1string2"
```

Section 13.2: Concatenate strings using System.Text.StringBuilder

Concatenating strings using a [StringBuilder](#) can offer performance advantages over simple string concatenation using +. This is due to the way memory is allocated. Strings are reallocated with each concatenation, StringBuilders allocate memory in blocks only reallocating when the current block is exhausted. This can make a huge difference when doing a lot of small concatenations.

```
StringBuilder sb = new StringBuilder();
for (int i = 1; i <= 5; i++)
{
    sb.Append(i);
    sb.Append(" ");
}
Console.WriteLine(sb.ToString()); // "1 2 3 4 5 "
```

Calls to Append() can be daisy chained, because it returns a reference to the StringBuilder:

```
StringBuilder sb = new StringBuilder();
sb.Append("some string ")
    .Append("another string");
```

Section 13.3: Concat string array elements using String.Join

The [String.Join](#) method can be used to concatenate multiple elements from a string array.

```
string[] value = {"apple", "orange", "grape", "pear"};
string separator = ", ";

string result = String.Join(separator, value, 1, 2);
Console.WriteLine(result);
```

Produces the following output: "orange, grape"

This example uses the [String.Join\(String, String\[\], Int32, Int32\)](#) overload, which specifies the start index and count on top of the separator and value.

If you do not wish to use the startIndex and count overloads, you can join all string given. Like this:

```
string[] value = {"apple", "orange", "grape", "pear"};
string separator = ", ";
string result = String.Join(separator, value);
Console.WriteLine(result);
```

这将产生：

苹果，橙子，葡萄，梨

第13.4节：使用\$连接两个字符串

\$ 提供了一种简单且简洁的方法来连接多个字符串。

```
var str1 = "text1";
var str2 = " ";
var str3 = "text3";
string result2 = $"{str1}{str2}{str3}"; //"text1 text3"
```

which will produce:

apple, orange, grape, pear

Section 13.4: Concatenation of two strings using \$

\$ provides an easy and a concise method to concatenate multiple strings.

```
var str1 = "text1";
var str2 = " ";
var str3 = "text3";
string result2 = $"{str1}{str2}{str3}"; //"text1 text3"
```

第14章：字符串操作

第14.1节：替换字符串中的字符串

使用System.String.Replace方法，可以用另一个字符串替换字符串的一部分。

```
string s = "Hello World";
s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

所有搜索字符串的出现都会被替换：

```
string s = "Hello World";
s = s.Replace("l", "L"); // s = "HeLLo WorLD"
```

String.Replace也可以用来删除字符串的一部分，通过指定空字符串作为替换值：

```
string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"
```

第14.2节：在字符串中查找字符串

使用System.String.Contains可以判断某个字符串是否存在于另一个字符串中。该方法返回一个布尔值，若字符串存在则为true，否则为false。

```
string s = "Hello World";
bool stringExists = s.Contains("ello"); //stringExists =true 因为字符串包含子字符串
```

使用System.String.IndexOf方法，可以定位子字符串在现有字符串中的起始位置。

注意返回的位置是从零开始计数的，如果未找到子字符串，则返回值为-1。

```
string s = "Hello World";
int location = s.IndexOf("ello"); // location = 1
```

要从字符串的末尾查找第一个位置，请使用 System.String.LastIndexOf 方法：

```
string s = "Hello World";
int location = s.LastIndexOf("l"); // location = 9
```

第14.3节：从字符串中移除（修剪）空白字符

可以使用System.String.Trim方法来移除字符串开头和结尾的所有空白字符：

```
string s = "    String with spaces at both ends    ";
s = s.Trim(); // s = "String with spaces at both ends"
```

此外：

- 仅从字符串的开头移除空白字符使用：System.String.TrimStart
- 仅从字符串的结尾移除空白字符使用：System.String.TrimEnd

Chapter 14: String Manipulation

Section 14.1: Replacing a string within a string

Using the [System.String.Replace](#) method, you can replace part of a string with another string.

```
string s = "Hello World";
s = s.Replace("World", "Universe"); // s = "Hello Universe"
```

All the occurrences of the search string are replaced:

```
string s = "Hello World";
s = s.Replace("l", "L"); // s = "HeLLo WorLD"
```

String.Replace can also be used to *remove* part of a string, by specifying an empty string as the replacement value:

```
string s = "Hello World";
s = s.Replace("ell", String.Empty); // s = "Ho World"
```

Section 14.2: Finding a string within a string

Using the [System.String.Contains](#) you can find out if a particular string exists within a string. The method returns a boolean, true if the string exists else false.

```
string s = "Hello World";
bool stringExists = s.Contains("ello"); //stringExists =true as the string contains the substring
```

Using the [System.String.IndexOf](#) method, you can locate the starting position of a substring within an existing string.

Note the returned position is zero-based, a value of -1 is returned if the substring is not found.

```
string s = "Hello World";
int location = s.IndexOf("ello"); // location = 1
```

To find the first location from the *end* of a string, use the [System.String.LastIndexOf](#) method:

```
string s = "Hello World";
int location = s.LastIndexOf("l"); // location = 9
```

Section 14.3: Removing (Trimming) white-space from a string

The [System.String.Trim](#) method can be used to remove all leading and trailing white-space characters from a string:

```
string s = "    String with spaces at both ends    ";
s = s.Trim(); // s = "String with spaces at both ends"
```

In addition:

- To remove white-space only from the *beginning* of a string use: [System.String.TrimStart](#)
- To remove white-space only from the *end* of a string use: [System.String.TrimEnd](#)

提取字符串的子串。

可以使用System.String.Substring方法来提取字符串的一部分。

```
string s = "保留的单词部分";
s= str.Substring(26); //s="retained"

s1 = s.Substring(0,5); //s="A por"
```

第14.4节：使用分隔符拆分字符串

使用System.String.Split方法返回一个字符串数组，该数组包含根据指定分隔符拆分的原始字符串的子字符串：

```
string sentence = "One Two Three Four";
string[] stringArray = sentence.Split(' ');

foreach (string word in stringArray)
{
    Console.WriteLine(word);
}
```

输出：

```
One
Two
Three
Four
```

第14.5节：将字符串数组连接成单个字符串

System.String.Join方法允许使用指定的分隔符将字符串数组中的所有元素连接起来，分隔符位于每个元素之间：

```
string[] words = {"One", "Two", "Three", "Four"};
string singleString = String.Join(", ", words); // singleString = "One, Two, Three, Four"
```

第14.6节：字符串连接

字符串连接可以使用System.String.Concat方法，或者（更简单地）使用+运算符：

```
string first = "Hello ";
string second = "World";

string concat = first + second; // concat = "Hello World"
concat = String.Concat(first, second); // concat = "Hello World"
```

第14.7节：更改字符串中字符的大小写

System.String类支持多种方法，用于在字符串中转换大写和小写字符。

Substring to extract part of a string.

The System.String.Substring method can be used to extract a portion of the string.

```
string s = "A portion of word that is retained";
s= str.Substring(26); //s="retained"

s1 = s.Substring(0,5); //s="A por"
```

Section 14.4: Splitting a string using a delimiter

Use the System.String.Split method to return a string array that contains substrings of the original string, split based on a specified delimiter:

```
string sentence = "One Two Three Four";
string[] stringArray = sentence.Split(' ');

foreach (string word in stringArray)
{
    Console.WriteLine(word);
}
```

Output:

```
One
Two
Three
Four
```

Section 14.5: Concatenate an array of strings into a single string

The System.String.Join method allows to concatenate all elements in a string array, using a specified separator between each element:

```
string[] words = {"One", "Two", "Three", "Four"};
string singleString = String.Join(", ", words); // singleString = "One, Two, Three, Four"
```

Section 14.6: String Concatenation

String Concatenation can be done by using the System.String.Concat method, or (much easier) using the + operator:

```
string first = "Hello ";
string second = "World";

string concat = first + second; // concat = "Hello World"
concat = String.Concat(first, second); // concat = "Hello World"
```

Section 14.7: Changing the case of characters within a String

The System.String class supports a number of methods to convert between uppercase and lowercase characters in a string.

- [System.String.ToLowerInvariant](#) 用于返回转换为小写的字符串对象。
- [System.String.ToUpperInvariant](#) 用于返回转换为大写的字符串对象。

注意： 使用这些方法的不变文化版本的原因是为了防止产生意外的特定文化字母。这里有详细说明。

示例：

```
string s = "My String";
s = s.ToLowerInvariant(); // "my string"
s = s.ToUpperInvariant(); // "MY STRING"
```

注意，你可以选择在转换大小写时指定特定的[Culture](#)，方法是分别使用[String.ToLower\(CultureInfo\)](#)和[String.ToUpper\(CultureInfo\)](#)方法。

- [System.String.ToLowerInvariant](#) is used to return a String object converted to lowercase.
- [System.String.ToUpperInvariant](#) is used to return a String object converted to uppercase.

Note: The reason to use the *invariant* versions of these methods is to prevent producing unexpected culture-specific letters. This is explained [here in detail](#).

Example:

```
string s = "My String";
s = s.ToLowerInvariant(); // "my string"
s = s.ToUpperInvariant(); // "MY STRING"
```

Note that you *can* choose to specify a specific [Culture](#) when converting to lowercase and uppercase by using the [String.ToLower\(CultureInfo\)](#) and [String.ToUpper\(CultureInfo\)](#) methods accordingly.

第15章：字符串插值

第15.1节：字符串中的日期格式化

```
var date = new DateTime(2015, 11, 11);
var str = $"It's {date:MMMM d, yyyy}, make a wish!";
System.Console.WriteLine(str);
```

你也可以使用[DateTime.ToString](#)方法来格式化DateTime对象。这将产生与上面代码相同的输出。

```
var date = new DateTime(2015, 11, 11);
var str = date.ToString("MMMM d, yyyy");
str = "It's " + str + ", make a wish!";
Console.WriteLine(str);
```

输出：

这是2015年11月11日，许个愿吧！

[.NET Fiddle上的实时演示](#)

[使用DateTime.ToString的实时演示](#)

注意：MM代表月份，mm代表分钟。使用时务必小心，因为错误可能会引入难以发现的漏洞。

第15.2节：输出填充

字符串可以格式化以接受一个填充参数，该参数指定插入字符串将占用多少字符位置：

```
 ${value, padding}
```

注意：正数填充值表示左填充，负数填充值表示右填充。

左填充

左填充5（在数字值前添加3个空格，因此结果字符串中总共占用5个字符位置。）

```
var number = 42;
var str = $"生命、宇宙以及一切的答案是 {number, 5}.";
//str is "生命、宇宙以及一切的答案是    42.";
//                                                 ^^^^^^
System.Console.WriteLine(str);
```

输出：

Chapter 15: String Interpolation

Section 15.1: Format dates in strings

```
var date = new DateTime(2015, 11, 11);
var str = $"It's {date:MMMM d, yyyy}, make a wish!";
System.Console.WriteLine(str);
```

You can also use the [DateTime.ToString](#) method to format the DateTime object. This will produce the same output as the code above.

```
var date = new DateTime(2015, 11, 11);
var str = date.ToString("MMMM d, yyyy");
str = "It's " + str + ", make a wish!";
Console.WriteLine(str);
```

Output:

It's November 11, 2015, make a wish!

[Live Demo on .NET Fiddle](#)

[Live Demo using DateTime.ToString](#)

Note: MM stands for months and mm for minutes. Be very careful when using these as mistakes can introduce bugs that may be difficult to discover.

Section 15.2: Padding the output

String can be formatted to accept a padding parameter that will specify how many character positions the inserted string will use :

```
 ${value, padding}
```

NOTE: Positive padding values indicate left padding and negative padding values indicate right padding.

Left Padding

A left padding of 5 (adds 3 spaces before the value of number, so it takes up a total of 5 character positions in the resulting string.)

```
var number = 42;
var str = $"The answer to life, the universe and everything is {number, 5}.";
//str is "The answer to life, the universe and everything is    42.";
//                                                 ^^^^^^
System.Console.WriteLine(str);
```

Output:

生命、宇宙以及一切问题的答案是42。

[.NET Fiddle 在线演示](#)

右侧填充

右侧填充，使用负填充值，会在当前值的末尾添加空格。

```
var number = 42;
var str = $"生命、宇宙以及一切问题的答案是 ${number, -5}.";
//str 是 "生命、宇宙以及一切问题的答案是 42  ";
//
System.Console.WriteLine(str);
```

^^^^^

The answer to life, the universe and everything is 42.

[Live Demo on .NET Fiddle](#)

Right Padding

Right padding, which uses a negative padding value, will add spaces to the end of the current value.

```
var number = 42;
var str = $"The answer to life, the universe and everything is ${number, -5}.";
//str is "The answer to life, the universe and everything is 42  ";
//
System.Console.WriteLine(str);
```

^^^^^

输出：

生命、宇宙以及一切问题的答案是42 。

[.NET Fiddle 在线演示](#)

带格式说明符的填充

你也可以将现有的格式说明符与填充结合使用。

```
var number = 42;
var str = $"生命、宇宙以及一切问题的答案是 ${number, 5:f1}";
//str 是 "生命、宇宙以及一切问题的答案是 42.1 ";
//
System.Console.WriteLine(str);
```

^^^^^

Output:

The answer to life, the universe and everything is 42 .

[Live Demo on .NET Fiddle](#)

Padding with Format Specifiers

You can also use existing formatting specifiers in conjunction with padding.

```
var number = 42;
var str = $"The answer to life, the universe and everything is ${number, 5:f1}";
//str is "The answer to life, the universe and everything is 42.1 ";
//
System.Console.WriteLine(str);
```

^^^^^

[.NET Fiddle 在线演示](#)

第15.3节：表达式

插值字符串中也可以使用完整的表达式。

```
var StrWithMathExpression = $"1 + 2 = {1 + 2}"; // -> "1 + 2 = 3"
string world = "world";
var StrWithFunctionCall = $"Hello, {world.ToUpper()}!"; // -> "Hello, WORLD!"
```

[.NET Fiddle 在线演示](#)

第15.4节：字符串中的数字格式化

你可以使用冒号和标准数字格式语法来控制数字的格式。

```
var decimalValue = 120.5;

var asCurrency = $"It costs {decimalValue:C}";
// 字符串值为 "It costs $120.50" (取决于你的本地货币设置)

var withThreeDecimalPlaces = $"Exactly {decimalValue:F3}";
// 字符串值为 "Exactly 120.500"

var integerValue = 57;
```

[Live Demo on .NET Fiddle](#)

Section 15.3: Expressions

Full expressions can also be used in interpolated strings.

```
var StrWithMathExpression = $"1 + 2 = {1 + 2}"; // -> "1 + 2 = 3"
string world = "world";
var StrWithFunctionCall = $"Hello, {world.ToUpper()}!"; // -> "Hello, WORLD!"
```

[Live Demo on .NET Fiddle](#)

Section 15.4: Formatting numbers in strings

You can use a colon and the [standard numeric format syntax](#) to control how numbers are formatted.

```
var decimalValue = 120.5;

var asCurrency = $"It costs {decimalValue:C}";
// String value is "It costs $120.50" (depending on your local currency settings)

var withThreeDecimalPlaces = $"Exactly {decimalValue:F3}";
// String value is "Exactly 120.500"

var integerValue = 57;
```

```
var prefixedIfNecessary = $"{integerValue:D5}";  
// 字符串值是 "00057"
```

[.NET Fiddle 在线演示](#)

第15.5节：简单用法

```
var name = "World";  
var str = $"Hello, {name}!";  
//str 现在包含: "Hello, World!"
```

幕后

内部这个

```
$"Hello, {name}!"
```

将被编译成类似这样的代码：

```
string.Format("Hello, {0}!", name);
```

```
var prefixedIfNecessary = $"{integerValue:D5}";  
// String value is "00057"
```

[Live Demo on .NET Fiddle](#)

Section 15.5: Simple Usage

```
var name = "World";  
var str = $"Hello, {name}!";  
//str now contains: "Hello, World!"
```

Behind the scenes

Internally this

```
$"Hello, {name}!"
```

Will be compiled to something like this:

```
string.Format("Hello, {0}!", name);
```

第16章：字符串转义序列

第16.1节：在字符串字面量中转义特殊符号

反斜杠

```
// 两种情况下文件名都是 c:\myfile.txt
string filename = "c:\\myfile.txt";
string filename = @"c:\\myfile.txt";
```

第二个例子使用了逐字字符串字面量，它不会将反斜杠视为转义字符。

引号

```
string text = @"\\"Hello World!\"", said the quick brown fox.";
string verbatimText = @"""Hello World!\"", said the quick brown fox.";
```

两个变量将包含相同的文本。

"Hello World!", 敏捷的棕色狐狸说。

换行符

逐字字符串字面量可以包含换行符：

```
string text = "Hello\rWorld!"; string verbatimText = @"HelloWorld!";
```

两个变量将包含相同的文本。

第16.2节：Unicode字符转义序列

```
string sqrt = @"\u221A"; // √
string emoji = @"\U0001F601"; // ?
string text = @"\u002Hello World\u002"; // "Hello World"
string variableWidth = @"\x2Hello World\x2"; // "Hello World"
```

第16.3节：在字符字面量中转义特殊符号

撇号

```
char apostrophe = '\'';
```

反斜杠

```
char oneBackslash = '\\';
```

第16.4节：在标识符中使用转义序列

转义序列不限于string和char字面量。

Chapter 16: String Escape Sequences

Section 16.1: Escaping special symbols in string literals

Backslash

```
// The filename will be c:\\myfile.txt in both cases
string filename = "c:\\myfile.txt";
string filename = @"c:\\myfile.txt";
```

The second example uses a verbatim string literal, which doesn't treat the backslash as an escape character.

Quotes

```
string text = @"\\"Hello World!\"", said the quick brown fox.";
string verbatimText = @"""Hello World!\"", said the quick brown fox.";
```

Both variables will contain the same text.

"Hello World!", said the quick brown fox.

Newlines

Verbatim string literals can contain newlines:

```
string text = "Hello\r\nWorld!";
string verbatimText = @"Hello
World!";
```

Both variables will contain the same text.

Section 16.2: Unicode character escape sequences

```
string sqrt = @"\u221A"; // √
string emoji = @"\U0001F601"; // ?
string text = @"\u002Hello World\u002"; // "Hello World"
string variableWidth = @"\x2Hello World\x2"; // "Hello World"
```

Section 16.3: Escaping special symbols in character literals

Apostrophes

```
char apostrophe = '\'';
```

Backslash

```
char oneBackslash = '\\';
```

Section 16.4: Using escape sequences in identifiers

Escape sequences are not restricted to `string` and `char` literals.

假设你需要重写一个第三方方法：

```
protected abstract IEnumerable<Texte> ObtenirŒuvres();
```

假设字符Œ在你用于C#源文件的字符编码中不可用。你很幸运，代码中允许在identifiers中使用类型为\u####或\U#####的转义序列。因此，写成以下形式是合法的：

```
protected override IEnumerable<Texte> ObtenirŒuvres()
{
    // ...
}
```

C#编译器会知道Œ和\u0152是同一个字符。

(不过，切换到UTF-8或类似能处理所有字符的编码可能是个好主意。)

第16.5节：未识别的转义序列会产生 编译时错误

以下示例将无法编译：

```
string s = "\c";
char c = '\c';
```

相反，它们将在编译时产生错误未识别的转义序列。

Suppose you need to override a third-party method:

```
protected abstract IEnumerable<Texte> ObtenirŒuvres();
```

and suppose the character Œ is not available in the character encoding you use for your C# source files. You are lucky, it is permitted to use escapes of the type \u#### or \U##### in **identifiers** in the code. So it is legal to write:

```
protected override IEnumerable<Texte> Obtenir\u0152uvres()
{
    // ...
}
```

and the C# compiler will know Œ and \u0152 are the same character.

(However, it might be a good idea to switch to UTF-8 or a similar encoding that can handle all characters.)

Section 16.5: Unrecognized escape sequences produce compile-time errors

The following examples will not compile:

```
string s = "\c";
char c = '\c';
```

Instead, they will produce the error `Unrecognized escape sequence at compile time.`

第17章：StringBuilder

第17.1节：什么是StringBuilder以及何时使用它

StringBuilder表示一系列字符，与普通字符串不同，它是可变的。通常我们需要修改已经创建的字符串，但标准字符串对象是不可变的。这意味着每次修改字符串时，都需要创建一个新的字符串对象，将内容复制过去，然后重新赋值。

```
string myString = "Apples";
myString += " are my favorite fruit";
```

在上面的例子中，myString最初只有值"Apples"。然而，当我们连接" are my favorite fruit"时，字符串类内部需要执行的操作包括：

- 创建一个新的字符数组，其长度等于myString和我们要追加的新字符串的长度之和。
- 将myString的所有字符复制到新数组的开头，并将新字符串复制到数组的末尾。
- 在内存中创建一个新的字符串对象，并重新赋值给myString。

对于单次连接操作，这相对简单。但是，如果需要执行多次追加操作，比如在循环中呢？

```
String myString = "";
for (int i = 0; i < 10000; i++)
    myString += " "; // 向字符串中添加1万个空格
```

由于反复复制和对象创建，这将显著降低我们程序的性能。我们可以通过使用StringBuilder来避免这种情况。

```
StringBuilder myStringBuilder = new StringBuilder();
for (int i = 0; i < 10000; i++)
    myStringBuilder.Append('');
```

现在，当运行相同的循环时，程序的性能和执行速度将比使用普通字符串快得多。要将StringBuilder转换回普通字符串，我们只需调用StringBuilder的ToString()方法。

然而，这并不是StringBuilder唯一的优化。为了进一步优化功能，我们可以利用其他有助于提升性能的属性。

```
StringBuilder sb = new StringBuilder(10000); // 初始化容量为10000
```

如果我们事先知道StringBuilder需要的长度，可以提前指定其大小，这将防止其内部字符数组需要重新调整大小。

```
sb.Append('k', 2000);
```

虽然使用StringBuilder进行追加比字符串快得多，但如果你只需要多次添加单个字符，速度甚至可以更快。

一旦完成构建字符串后，可以使用ToString()方法将StringBuilder转换为基本的string。这通常是必要的，因为StringBuilder类并不继承自string。

Chapter 17: StringBuilder

Section 17.1: What a StringBuilder is and when to use one

A [StringBuilder](#) represents a series of characters, which unlike a normal string, are mutable. Often times there is a need to modify strings that we've already made, but the standard string object is not mutable. This means that each time a string is modified, a new string object needs to be created, copied to, and then reassigned.

```
string myString = "Apples";
myString += " are my favorite fruit";
```

In the above example, myString initially only has the value "Apples". However, when we concatenate " are my favorite fruit", what the string class does internally needs to do involves:

- Creating a new array of characters equal to the length of myString and the new string we are appending.
- Copying all of the characters of myString into the beginning of our new array and copying the new string into the end of the array.
- Create a new string object in memory and reassign it to myString.

For a single concatenation, this is relatively trivial. However, what if needed to perform many append operations, say, in a loop?

```
String myString = "";
for (int i = 0; i < 10000; i++)
    myString += " "; // puts 10,000 spaces into our string
```

Due to the repeated copying and object creation, this will bring significantly degrade the performance of our program. We can avoid this by instead using a StringBuilder.

```
StringBuilder myStringBuilder = new StringBuilder();
for (int i = 0; i < 10000; i++)
    myStringBuilder.Append('');
```

Now when the same loop is run, the performance and speed of the execution time of the program will be significantly faster than using a normal string. To make the StringBuilder back into a normal string, we can simply call the `ToString()` method of StringBuilder.

However, this isn't the only optimization StringBuilder has. In order to further optimize functions, we can take advantage of other properties that help improve performance.

```
StringBuilder sb = new StringBuilder(10000); // initializes the capacity to 10000
```

If we know in advance how long our StringBuilder needs to be, we can specify its size ahead of time, which will prevent it from needing to resize the character array it has internally.

```
sb.Append('k', 2000);
```

Though using StringBuilder for appending is much faster than a string, it can run even faster if you only need to add a single character many times.

Once you have completed building your string, you may use the `ToString()` method on the StringBuilder to convert it to a basic `string`. This is often necessary because the StringBuilder class does not inherit from `string`.

例如，下面是如何使用StringBuilder来创建一个string：

```
string RepeatCharacterTimes(char character, int times)
{
    StringBuilder builder = new StringBuilder("");
    for (int counter = 0; counter < times; counter++)
    {
        //将一个字符实例追加到StringBuilder中。
        builder.Append(character);
    }
    //将结果转换为字符串并返回。
    return builder.ToString();
}
```

总之，当需要对字符串进行多次修改且关注性能时，应使用StringBuilder代替string。

第17.2节：使用StringBuilder从大量记录创建字符串

```
public string GetCustomerNamesCsv()
{
    List<CustomerData> customerDataRecords = GetCustomerData(); // 返回大量记录，例如，10000+
    StringBuilder customerNamesCsv = new StringBuilder();
    foreach (CustomerData record in customerDataRecords)
    {
        customerNamesCsv
            .Append(record.LastName)
            .Append(',')
            .Append(record.FirstName)
            .Append(Environment.NewLine);
    }
    return customerNamesCsv.ToString();
}
```

For example, here is how you can use a StringBuilder to create a string:

```
string RepeatCharacterTimes(char character, int times)
{
    StringBuilder builder = new StringBuilder("");
    for (int counter = 0; counter < times; counter++)
    {
        //Append one instance of the character to the StringBuilder.
        builder.Append(character);
    }
    //Convert the result to string and return it.
    return builder.ToString();
}
```

In conclusion, StringBuilder should be used in place of string when many modifications to a string need to be made with performance in mind.

Section 17.2: Use StringBuilder to create string from a large number of records

```
public string GetCustomerNamesCsv()
{
    List<CustomerData> customerDataRecords = GetCustomerData(); // Returns a large number of records, say, 10000+
    StringBuilder customerNamesCsv = new StringBuilder();
    foreach (CustomerData record in customerDataRecords)
    {
        customerNamesCsv
            .Append(record.LastName)
            .Append(',')
            .Append(record.FirstName)
            .Append(Environment.NewLine);
    }
    return customerNamesCsv.ToString();
}
```

第18章：正则表达式解析

名称	详细信息
模式	用于查找的string模式。更多信息请参见： msdn
RegexOptions [可选]	这里常见的选项有单行和多行。它们会改变模式元素的行为，比如点号(.)，在多行模式下不会匹配换行符()，但在单行模式下会匹配。默认行为： msdn
超时[可选]	当模式变得更复杂时，查找可能会消耗更多时间。这是查找操作的超时时间，类似于网络编程中的超时设置。

第18.1节：单次匹配

使用`System.Text.RegularExpressions`;

```
string pattern = ":(.*):";  
string lookup = "--:text in here:--";  
  
// 实例化你的正则表达式对象并传入一个模式  
Regex rgxLookup = new Regex(pattern, RegexOptions.Singleline, TimeSpan.FromSeconds(1));  
// 从你的正则表达式对象中获取匹配结果  
Match mLookup = rgxLookup.Match(lookup);  
  
// 组索引0总是匹配整个模式。  
// 括号内的匹配将通过索引1及以上访问。  
string found = mLookup.Groups[1].Value;
```

结果：

```
found = "这里的文本"
```

第18.2节：多重匹配

使用`System.Text.RegularExpressions`;

```
List<string> found = new List<string>();  
string pattern = ":(.*):";  
string lookup = "--:这里的文本:--:另一个:--第三个:--!123:第四个:--";  
  
// 实例化你的正则表达式对象并传入一个模式  
Regex rgxLookup = new Regex(pattern, RegexOptions.Singleline, TimeSpan.FromSeconds(1));  
MatchCollection mLookup = rgxLookup.Matches(lookup);  
  
foreach(Match match in mLookup)  
{  
    found.Add(match.Groups[1].Value);  
}
```

结果：

```
found = new List<string>() { "这里的文本", "另一个", "第三个", "第四个" }
```

Chapter 18: Regex Parsing

Name	Details
Pattern	The <code>string</code> pattern that has to be used for the lookup. For more information: msdn
RegexOptions [Optional]	The common options in here are Singleline and Multiline. They are changing the behaviour of pattern-elements like the dot(.) which won't cover a NewLine(\n) in Multiline-Mode but in SingleLine-Mode. Default behaviour: msdn
Timeout [Optional]	Where patterns are getting more complex the lookup can consume more time. This is the passed timeout for the lookup just as known from network-programming.

Section 18.1: Single match

`using System.Text.RegularExpressions;`

```
string pattern = ":(.*):";  
string lookup = "--:text in here:--";  
  
// Instanciate your regex object and pass a pattern to it  
Regex rgxLookup = new Regex(pattern, RegexOptions.Singleline, TimeSpan.FromSeconds(1));  
// Get the match from your regex-object  
Match mLookup = rgxLookup.Match(lookup);  
  
// The group-index 0 always covers the full pattern.  
// Matches inside parentheses will be accessed through the index 1 and above.  
string found = mLookup.Groups[1].Value;
```

Result:

```
found = "text in here"
```

Section 18.2: Multiple matches

`using System.Text.RegularExpressions;`

```
List<string> found = new List<string>();  
string pattern = ":(.*):";  
string lookup = "--:text in here:--:another one:-:third one:---!123:fourth:--";  
  
// Instanciate your regex object and pass a pattern to it  
Regex rgxLookup = new Regex(pattern, RegexOptions.Singleline, TimeSpan.FromSeconds(1));  
MatchCollection mLookup = rgxLookup.Matches(lookup);  
  
foreach(Match match in mLookup)  
{  
    found.Add(match.Groups[1].Value);  
}
```

Result:

```
found = new List<string>() { "text in here", "another one", "third one", "fourth" }
```

第19章：DateTime方法

第19.1节：DateTime格式化

标准DateTime格式化

DateTimeFormatInfo指定了一组用于简单日期和时间格式化的格式说明符。每个说明符对应一个特定的DateTimeFormatInfo格式模式。

```
//创建日期时间
DateTime dt = new DateTime(2016,08,01,18,50,23,230);
```

```
var t = String.Format("{0:t}", dt); // "6:50 PM"
var d = String.Format("{0:d}", dt); // "8/1/2016"
var T = String.Format("{0:T}", dt); // "6:50:23 PM"
var D = String.Format("{0:D}", dt); // "Monday, August 1, 2016"
var f = String.Format("{0:f}", dt); // "Monday, August 1, 2016 6:50 PM"
var F = String.Format("{0:F}", dt); // "Monday, August 1, 2016 6:50:23 PM"
var g = String.Format("{0:g}", dt); // "8/1/2016 6:50 PM"
var G = String.Format("{0:G}", dt); // "8/1/2016 6:50:23 PM"
var m = String.Format("{0:m}", dt); // "August 1"
var y = String.Format("{0:y}", dt); // "August 2016"
var r = String.Format("{0:r}", dt); // "Mon, 01 Aug 2016 18:50:23 GMT"
var s = String.Format("{0:s}", dt); // "2016-08-01T18:50:23"
var u = String.Format("{0:u}", dt); // "2016-08-01 18:50:23Z"
通用可排序日期时间
```

短时间
短日期
长时间
长日期
长日期+短时间
完整日期时间
短日期+短时间
短日期+长时间
月日
年月
RFC1123
可排序日期时间

自定义日期时间格式

以下是自定义格式说明符：

- y (年)
- M (月)
- d (日)
- h (12小时制)
- H (24小时制)
- m (分钟)
- s (秒)
- f (秒的小数部分)
- F (秒的小数部分, 末尾零被去除)
- t (下午或上午)
- z (时区)。

```
var year = String.Format("{0:y yy yyy yyyy}", dt); // "16 16 2016 2016" 年
var month = String.Format("{0:M MM MMM MMMM}", dt); // "8 08 Aug August" 月
var day = String.Format("{0:d dd ddd dddd}", dt); // "1 01 Mon Monday" 日
var hour = String.Format("{0:h hh H HH}", dt); // "6 06 18 18" 12/24小时制
var minute = String.Format("{0:m mm}", dt); // "50 50" 分钟
var secound = String.Format("{0:s ss}", dt); // "23 23" 秒
var fraction = String.Format("{0:f ff fff ffff}", dt); // "2 23 230 2300" 秒的小数部分
var fraction2 = String.Format("{0:F FF FFF FFFF}", dt); // "2 23 23 23" 去除末尾零
var period = String.Format("{0:t tt}", dt); // "P PM" 上午或下午
var zone = String.Format("{0:z zz zzz}", dt); // "+0 +00 +00:00" 时区
```

你也可以使用日期分隔符/（斜杠）和时间分隔符:（冒号）。

Chapter 19: DateTime Methods

Section 19.1: DateTime Formatting

Standard DateTime Formatting

DateTimeFormatInfo specifies a set of specifiers for simple date and time formatting. Every specifier correspond to a particular DateTimeFormatInfo format pattern.

```
//Create datetime
DateTime dt = new DateTime(2016,08,01,18,50,23,230);
```

```
var t = String.Format("{0:t}", dt); // "6:50 PM"
var d = String.Format("{0:d}", dt); // "8/1/2016"
var T = String.Format("{0:T}", dt); // "6:50:23 PM"
var D = String.Format("{0:D}", dt); // "Monday, August 1, 2016"
var f = String.Format("{0:f}", dt); // "Monday, August 1, 2016 6:50 PM"
var F = String.Format("{0:F}", dt); // "Monday, August 1, 2016 6:50:23 PM"
var g = String.Format("{0:g}", dt); // "8/1/2016 6:50 PM"
var G = String.Format("{0:G}", dt); // "8/1/2016 6:50:23 PM"
var m = String.Format("{0:m}", dt); // "August 1"
var y = String.Format("{0:y}", dt); // "August 2016"
var r = String.Format("{0:r}", dt); // "SMon, 01 Aug 2016 18:50:23 GMT"
var s = String.Format("{0:s}", dt); // "2016-08-01T18:50:23"
var u = String.Format("{0:u}", dt); // "2016-08-01 18:50:23Z"
UniversalSortableDateTime
```

ShortTime
ShortDate
LongTime
LongDate
LongDate+ShortTime
FullDateTime
ShortDate+ShortTime
ShortDate+LongTime
MonthDay
YearMonth
RFC1123
SortableDateTime

Custom DateTime Formatting

There are following custom format specifiers:

- y (year)
- M (month)
- d (day)
- h (hour 12)
- H (hour 24)
- m (minute)
- s (second)
- f (second fraction)
- F (second fraction, trailing zeroes are trimmed)
- t (P.M or A.M)
- z (time zone).

```
var year = String.Format("{0:y yy yyy yyyy}", dt); // "16 16 2016 2016" year
var month = String.Format("{0:M MM MMM MMMM}", dt); // "8 08 Aug August" month
var day = String.Format("{0:d dd ddd dddd}", dt); // "1 01 Mon Monday" day
var hour = String.Format("{0:h hh H HH}", dt); // "6 06 18 18" hour 12/24
var minute = String.Format("{0:m mm}", dt); // "50 50" minute
var secound = String.Format("{0:s ss}", dt); // "23 23" second
var fraction = String.Format("{0:f ff fff ffff}", dt); // "2 23 230 2300" sec.fraction
var fraction2 = String.Format("{0:F FF FFF FFFF}", dt); // "2 23 23 23" without zeroes
var period = String.Format("{0:t tt}", dt); // "P PM" A.M. or P.M.
var zone = String.Format("{0:z zz zzz}", dt); // "+0 +00 +00:00" time zone
```

You can use also date separator / (slash) and time separator : (colon).

更多信息请参见[MSDN](#)。

第19.2节：DateTime.AddDays(Double)

向DateTime对象添加天数。

```
DateTime today = DateTime.Now;
DateTime answer = today.AddDays(36);
Console.WriteLine("今天: {0:dddd}", today);
Console.WriteLine("从今天起36天后: {0:dddd}", answer);
```

你也可以通过传递负值来减去天数：

```
DateTime today = DateTime.Now;
DateTime answer = today.AddDays(-3);
Console.WriteLine("今天: {0:dddd}", today);
Console.WriteLine("从今天起-3天: {0:dddd}", answer);
```

第19.3节：DateTime.AddHours(Double)

```
double[] hours = {.08333, .16667, .25, .33333, .5, .66667, 1, 2,
                  29, 30, 31, 90, 365};
DateTime dateValue = new DateTime(2009, 3, 1, 12, 0, 0);

foreach (double hour in hours)
    Console.WriteLine("{0} + {1} 小时 = {2}", dateValue, hour,
                     dateValue.AddHours(hour));
```

第19.4节：DateTime.Parse(String)

```
// 将日期和时间的字符串表示转换为其DateTime等价物

var dateTime = DateTime.Parse("14:23 2016年7月22日");

Console.WriteLine(dateTime.ToString());
```

第19.5节：DateTime.TryParse(String, DateTime)

```
// 将指定的日期和时间字符串表示转换为其DateTime等价物，并
// 返回一个指示转换是否成功的值

string[] dateTimeStrings = new []{
    "14:23 2016年7月22日",
    "99:23 2x 2016年7月",
    "2016/7/22 14:23:00"
};

foreach(var dateTimeString in dateTimeStrings){

    DateTime dateTime;

    bool wasParsed = DateTime.TryParse(dateTimeString, out dateTime);

    string result = dateTimeString +
        (wasParsed
```

For more information [MSDN](#).

Section 19.2: DateTime.AddDays(Double)

Add days into a DateTime object.

```
DateTime today = DateTime.Now;
DateTime answer = today.AddDays(36);
Console.WriteLine("Today: {0:dddd}", today);
Console.WriteLine("36 days from today: {0:dddd}", answer);
```

You also can subtract days passing a negative value:

```
DateTime today = DateTime.Now;
DateTime answer = today.AddDays(-3);
Console.WriteLine("Today: {0:dddd}", today);
Console.WriteLine("-3 days from today: {0:dddd}", answer);
```

Section 19.3: DateTime.AddHours(Double)

```
double[] hours = {.08333, .16667, .25, .33333, .5, .66667, 1, 2,
                  29, 30, 31, 90, 365};
DateTime dateValue = new DateTime(2009, 3, 1, 12, 0, 0);

foreach (double hour in hours)
    Console.WriteLine("{0} + {1} hour(s) = {2}", dateValue, hour,
                     dateValue.AddHours(hour));
```

Section 19.4: DateTime.Parse(String)

```
// Converts the string representation of a date and time to its DateTime equivalent

var dateTime = DateTime.Parse("14:23 22 Jul 2016");

Console.WriteLine(dateTime.ToString());
```

Section 19.5: DateTime.TryParse(String, DateTime)

```
// Converts the specified string representation of a date and time to its DateTime equivalent and
// returns a value that indicates whether the conversion succeeded

string[] dateTimeStrings = new []{
    "14:23 22 Jul 2016",
    "99:23 2x Jul 2016",
    "22/7/2016 14:23:00"
};

foreach(var dateTimeString in dateTimeStrings){

    DateTime dateTime;

    bool wasParsed = DateTime.TryParse(dateTimeString, out dateTime);

    string result = dateTimeString +
        (wasParsed
```

```
? $"被解析为 {dateTime}"  
: "无法解析为 DateTime");
```

```
Console.WriteLine(result);  
}
```

第19.6节 : DateTime.AddMilliseconds(Double)

```
string dateFormat = "MM/dd/yyyy hh:mm:ss.fffffff";DateTime  
date1 = new DateTime(2010, 9, 8, 16, 0, 0);Console.WriteLine(  
"原始日期: {0} ({1:N0} 刻度)", date1.ToString(dateFo  
rmat), date1.Ticks);DateTime date2 = date1.AddMilliseconds(1);  
  
Console.WriteLine("第二个日期: {0} ({1:N0} 刻度)",  
date2.ToString(dateFormat), date2.Ticks);Console.WriteLine("日期差异:  
{0} ({1:N0} 刻度)", date2 - date1, date2.Ticks - date1.Ticks);  
  
DateTime date3 = date1.AddMilliseconds(1.5);  
  
Console.WriteLine("第三个日期: {0} ({1:N0} 刻度)",  
date3.ToString(dateFormat), date3.Ticks);  
Console.WriteLine("日期差异: {0} ({1:N0} 刻度)",  
date3 - date1, date3.Ticks - date1.Ticks);
```

第19.7节 : DateTime.Compare(DateTime t1, DateTime t2)

```
DateTime date1 = new DateTime(2009, 8, 1, 0, 0, 0);  
DateTime date2 = new DateTime(2009, 8, 1, 12, 0, 0);  
int result = DateTime.Compare(date1, date2);  
string relationship;  
  
if (result < 0)  
relationship = "早于";  
else if (result == 0)  
relationship = "与.....同时";  
else relationship = "晚于";  
  
Console.WriteLine("{0} {1} {2}", date1, relationship, date2);
```

第19.8节 : DateTime.DaysInMonth(Int32, Int32)

```
const int 七月 = 7;  
const int 二月 = 2;  
  
int 七月天数 = System.DateTime.DaysInMonth(2001, 七月);  
Console.WriteLine(七月天数);  
  
// daysInFeb 获取28, 因为1998年不是闰年。  
int 二月天数 = System.DateTime.DaysInMonth(1998, 二月);  
Console.WriteLine(二月天数);  
  
// daysInFebLeap 获取29, 因为1996年是闰年。  
int 闰年二月天数 = System.DateTime.DaysInMonth(1996, 二月);  
Console.WriteLine(闰年二月天数);
```

```
? $"was parsed to {dateTime}"  
: "can't be parsed to DateTime");
```

```
Console.WriteLine(result);  
}
```

Section 19.6: DateTime.AddMilliseconds(Double)

```
string dateFormat = "MM/dd/yyyy hh:mm:ss.fffffff";  
DateTime date1 = new DateTime(2010, 9, 8, 16, 0, 0);  
Console.WriteLine("Original date: {0} ({1:N0} ticks)\n",  
date1.ToString(dateFormat), date1.Ticks);  
  
DateTime date2 = date1.AddMilliseconds(1);  
Console.WriteLine("Second date: {0} ({1:N0} ticks)",  
date2.ToString(dateFormat), date2.Ticks);  
Console.WriteLine("Difference between dates: {0} ({1:N0} ticks)\n",  
date2 - date1, date2.Ticks - date1.Ticks);  
  
DateTime date3 = date1.AddMilliseconds(1.5);  
Console.WriteLine("Third date: {0} ({1:N0} ticks)",  
date3.ToString(dateFormat), date3.Ticks);  
Console.WriteLine("Difference between dates: {0} ({1:N0} ticks)",  
date3 - date1, date3.Ticks - date1.Ticks);
```

Section 19.7: DateTime.Compare(DateTime t1, DateTime t2)

```
DateTime date1 = new DateTime(2009, 8, 1, 0, 0, 0);  
DateTime date2 = new DateTime(2009, 8, 1, 12, 0, 0);  
int result = DateTime.Compare(date1, date2);  
string relationship;  
  
if (result < 0)  
relationship = "is earlier than";  
else if (result == 0)  
relationship = "is the same time as";  
else relationship = "is later than";  
  
Console.WriteLine("{0} {1} {2}", date1, relationship, date2);
```

Section 19.8: DateTime.DaysInMonth(Int32, Int32)

```
const int July = 7;  
const int Feb = 2;  
  
int daysInJuly = System.DateTime.DaysInMonth(2001, July);  
Console.WriteLine(daysInJuly);  
  
// daysInFeb gets 28 because the year 1998 was not a leap year.  
int daysInFeb = System.DateTime.DaysInMonth(1998, Feb);  
Console.WriteLine(daysInFeb);  
  
// daysInFebLeap gets 29 because the year 1996 was a leap year.  
int daysInFebLeap = System.DateTime.DaysInMonth(1996, Feb);  
Console.WriteLine(daysInFebLeap);
```

第19.9节 : DateTime.AddYears(Int32)

在 `dateTime` 对象上添加年份：

```
DateTime baseDate = new DateTime(2000, 2, 29);Console.WriteLine("基准日期: {0:d}", baseDate);

// 显示过去十五年的日期。
for (int ctr = -1; ctr >= -15; ctr--)
    Console.WriteLine("{0,2} 年前: {1:d}",
        Math.Abs(ctr), baseDate.AddYears(ctr));

Console.WriteLine();

// 显示未来十五年的日期。
for (int ctr = 1; ctr <= 15; ctr++)
    Console.WriteLine("{0,2} 年后: {1:d}",
        ctr, baseDate.AddYears(ctr));
```

第19.10节 : 处理

`DateTime` 时的纯函数警告

维基百科目前对纯函数的定义如下：

1. 给定相同的参数值，函数总是计算出相同的结果值。函数结果值不能依赖于程序执行过程中或程序不同执行之间可能变化的任何隐藏信息或状态，也不能依赖于任何来自输入/输出设备的外部输入。
2. 计算结果不会引起任何语义上可观察的副作用或输出，例如可变对象的变异或对输入/输出设备的输出。

作为开发者，你需要了解纯方法，并且在许多领域中你会经常遇到它们。我见过许多初级开发者被 `DateTime` 类的方法所困扰。很多方法都是纯方法，如果你不了解这一点，可能会让你感到意外。举个例子：

```
DateTime sample = new DateTime(2016, 12, 25);
sample.AddDays(1);
Console.WriteLine(sample.ToShortDateString());
```

根据上面的例子，有人可能会期望控制台输出结果是“26/12/2016”，但实际上你得到的还是同一天。这是因为 `AddDays` 是一个纯方法，不会影响原始日期。要得到预期的输出，你需要将 `AddDays` 调用修改为：

```
sample = sample.AddDays(1);
```

第19.11节 : `DateTime.TryParseExact(String, String, IFormatProvider, DateTimeStyles, DateTime)`

使用指定的格式、特定文化的格式信息和样式，将指定的日期和时间字符串表示转换为其 `DateTime` 等价物。字符串表示的格式必须与指定格式完全匹配。该方法返回一个值，指示转换是否成功。

例如

```
CultureInfo enUS = new CultureInfo("en-US");
```

Section 19.9: `DateTime.AddYears(Int32)`

Add years on the `dateTime` object:

```
DateTime baseDate = new DateTime(2000, 2, 29);
Console.WriteLine("Base Date: {0:d}\n", baseDate);

// Show dates of previous fifteen years.
for (int ctr = -1; ctr >= -15; ctr--)
    Console.WriteLine("{0,2} year(s) ago:{1:d}",
        Math.Abs(ctr), baseDate.AddYears(ctr));

Console.WriteLine();

// Show dates of next fifteen years.
for (int ctr = 1; ctr <= 15; ctr++)
    Console.WriteLine("{0,2} year(s) from now: {1:d}",
        ctr, baseDate.AddYears(ctr));
```

Section 19.10: Pure functions warning when dealing with `DateTime`

Wikipedia currently defines a pure function as follows:

1. The function always evaluates the same result value given the same argument value(s). The function result value cannot depend on any hidden information or state that may change while program execution proceeds or between different executions of the program, nor can it depend on any external input from I/O devices .
2. Evaluation of the result does not cause any semantically observable side effect or output, such as mutation of mutable objects or output to I/O devices

As a developer you need to be aware of pure methods and you will stumble upon these a lot in many areas. One I have seen that bites many junior developers is working with `DateTime` class methods. A lot of these are pure and if you are unaware of these you can be in for a surprise. An example:

```
DateTime sample = new DateTime(2016, 12, 25);
sample.AddDays(1);
Console.WriteLine(sample.ToShortDateString());
```

Given the example above one may expect the result printed to console to be '26/12/2016' but in reality you end up with the same date. This is because `AddDays` is a pure method and does not affect the original date. To get the expected output you would have to modify the `AddDays` call to the following:

```
sample = sample.AddDays(1);
```

Section 19.11: `DateTime.TryParseExact(String, String, IFormatProvider, DateTimeStyles, DateTime)`

Converts the specified string representation of a date and time to its `DateTime` equivalent using the specified format, culture-specific format information, and style. The format of the string representation must match the specified format exactly. The method returns a value that indicates whether the conversion succeeded.

For Example

```
CultureInfo enUS = new CultureInfo("en-US");
```

```
string dateString;
System.DateTime dateValue;
```

解析无样式标志的日期。

```
dateString = " 5/01/2009 8:30 AM";
if (DateTime.TryParseExact(dateString, "g", enUS, DateTimeStyles.None, out dateValue))
{
    Console.WriteLine("将 '{0}' 转换为 {1} ({2})。", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("{0} 格式不正确。", dateString);
}

// 允许日期字符串前导空格。
if(DateTime.TryParseExact(dateString, "g", enUS, DateTimeStyles.AllowLeadingWhite, out dateValue))
{
    Console.WriteLine("将 '{0}' 转换为 {1} ({2})。", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("{0} 格式不正确。", dateString);
}
```

使用带有 M 和 MM 的自定义格式。

```
dateString = "2009/5/01 09:00";
if(DateTime.TryParseExact(dateString, "M/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out dateValue))
{
    Console.WriteLine("将 '{0}' 转换为 {1} ({2})。", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("{0} 格式不正确。", dateString);
}

// 允许日期字符串前导空格。
if(DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out dateValue))
{
    Console.WriteLine("将 '{0}' 转换为 {1} ({2})。", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("{0} 格式不正确。", dateString);
}
```

解析带有时区信息的字符串。

```
dateString = "2009/05/01 01:30:42 下午 -05:00";
if (DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm:ss tt zzz", enUS, DateTimeStyles.None, out dateValue))
{
    Console.WriteLine("将 '{0}' 转换为 {1} ({2})。", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("{0} 格式不正确。", dateString);
}
```

```
string dateString;
System.DateTime dateValue;
```

Parse date with no style flags.

```
dateString = " 5/01/2009 8:30 AM";
if (DateTime.TryParseExact(dateString, "g", enUS, DateTimeStyles.None, out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2})。", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("{0} is not in an acceptable format.", dateString);

}

// Allow a leading space in the date string.
if(DateTime.TryParseExact(dateString, "g", enUS, DateTimeStyles.AllowLeadingWhite, out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2})。", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("{0} is not in an acceptable format.", dateString);
}
```

Use custom formats with M and MM.

```
dateString = "5/01/2009 09:00";
if(DateTime.TryParseExact(dateString, "M/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2})。", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("{0} is not in an acceptable format.", dateString);

}

// Allow a leading space in the date string.
if(DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm", enUS, DateTimeStyles.None, out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2})。", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("{0} is not in an acceptable format.", dateString);
}
```

Parse a string with time zone information.

```
dateString = "05/01/2009 01:30:42 PM -05:00";
if (DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm:ss tt zzz", enUS, DateTimeStyles.None, out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2})。", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("{0} is not in an acceptable format.", dateString);
}
```

```
// 允许日期字符串前导空格。
if (DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm:ss tt zzz", enUS,
    DateTimeStyles.AdjustToUniversal, out dateValue))
{
    Console.WriteLine("将 '{0}' 转换为 {1} ({2})。", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("{0}' 格式不正确。", dateString);
}
```

解析表示协调世界时 (UTC) 的字符串。

```
dateString = "2008-06-11T16:11:20.0904778Z";
if(DateTime.TryParseExact(dateString, "o", CultureInfo.InvariantCulture, DateTimeStyles.None, out
    dateValue))
{
    Console.WriteLine("将 '{0}' 转换为 {1} ({2})。", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("{0}' 格式不被接受。", dateString);
}

if (DateTime.TryParseExact(dateString, "o", CultureInfo.InvariantCulture,
    DateTimeStyles.RoundtripKind, out dateValue))
{
    Console.WriteLine("将 '{0}' 转换为 {1} ({2})。", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("{0}' 格式不正确。", dateString);
}
```

输出

```
' 5/01/2009 8:30 AM' 格式不被接受。
已将 ' 5/01/2009 8:30 AM' 转换为 5/1/2009 8:30:00 AM (未指定)。
已将 '5/01/2009 09:00' 转换为 5/1/2009 9:00:00 AM (未指定)。
'5/01/2009 09:00' 格式不被接受。
已将 '05/01/2009 01:30:42 PM -05:00' 转换为 5/1/2009 11:30:42 AM (本地时间)。
已将 '05/01/2009 01:30:42 PM -05:00' 转换为 5/1/2009 6:30:42 PM (协调世界时)。
已将 '2008-06-11T16:11:20.0904778Z' 转换为 6/11/2008 9:11:20 AM (本地时间)。
已将 '2008-06-11T16:11:20.0904778Z' 转换为 2008/6/11 下午4:11:20 (UTC)。
```

第19.12节：DateTime.Add(TimeSpan)

```
// 计算从此时刻起36天后的星期几。
System.DateTime today = System.DateTime.Now;
System.TimeSpan duration = new System.TimeSpan(36, 0, 0, 0);
System.DateTime answer = today.Add(duration);
System.Console.WriteLine("{0:dddd}", answer);
```

第19.13节：带有文化信息的 Parse 和 TryParse

当从不同文化（语言）解析 DateTime 时，您可能想使用它，以下示例解析荷兰日期。

```
// Allow a leading space in the date string.
if (DateTime.TryParseExact(dateString, "MM/dd/yyyy hh:mm:ss tt zzz", enUS,
    DateTimeStyles.AdjustToUniversal, out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2})。", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("' {0}' is not in an acceptable format.", dateString);
}
```

Parse a string representing UTC.

```
dateString = "2008-06-11T16:11:20.0904778Z";
if(DateTime.TryParseExact(dateString, "o", CultureInfo.InvariantCulture, DateTimeStyles.None, out
    dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2})。", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("' {0}' is not in an acceptable format.", dateString);
}

if (DateTime.TryParseExact(dateString, "o", CultureInfo.InvariantCulture,
    DateTimeStyles.RoundtripKind, out dateValue))
{
    Console.WriteLine("Converted '{0}' to {1} ({2})。", dateString, dateValue, dateValue.Kind);
}
else
{
    Console.WriteLine("' {0}' is not in an acceptable format.", dateString);
}
```

Outputs

```
' 5/01/2009 8:30 AM' is not in an acceptable format.
Converted ' 5/01/2009 8:30 AM' to 5/1/2009 8:30:00 AM (Unspecified).
Converted '5/01/2009 09:00' to 5/1/2009 9:00:00 AM (Unspecified).
'5/01/2009 09:00' is not in an acceptable format.
Converted '05/01/2009 01:30:42 PM -05:00' to 5/1/2009 11:30:42 AM (Local).
Converted '05/01/2009 01:30:42 PM -05:00' to 5/1/2009 6:30:42 PM (Utc).
Converted '2008-06-11T16:11:20.0904778Z' to 6/11/2008 9:11:20 AM (Local).
Converted '2008-06-11T16:11:20.0904778Z' to 6/11/2008 4:11:20 PM (Utc).
```

Section 19.12: DateTime.Add(TimeSpan)

```
// Calculate what day of the week is 36 days from this instant.
System.DateTime today = System.DateTime.Now;
System.TimeSpan duration = new System.TimeSpan(36, 0, 0, 0);
System.DateTime answer = today.Add(duration);
System.Console.WriteLine("{0:dddd}", answer);
```

Section 19.13: Parse and TryParse with culture info

You might want to use it when parsing DateTimes from [different cultures \(languages\)](#), following example parses Dutch date.

```

DateTime dateResult;
var dutchDateString = "31 oktober 1999 04:20";
var dutchCulture = CultureInfo.CreateSpecificCulture("nl-NL");
DateTime.TryParse(dutchDateString, dutchCulture, styles, out dateResult);
// 输出 {31/10/1999 04:20:00}

```

解析示例：

```

DateTime.Parse(dutchDateString, dutchCulture)
// 输出 {31/10/1999 04:20:00}

```

第19.14节：for循环中作为初始化器的DateTime

```

// 这将在两个DateTime之间的范围内迭代
// 使用给定的迭代器（任何Add方法）

DateTime start = new DateTime(2016, 01, 01);
DateTime until = new DateTime(2016, 02, 01);

// 注意：由于Add方法返回一个新的DateTime，
// 你必须像dt = dt.Add()这样覆盖迭代器中的dt
for (DateTime dt = start; dt < until; dt = dt.AddDays(1))
{
    Console.WriteLine("Added {0} days. Resulting DateTime: {1}",
                      (dt - start).Days, dt.ToString());
}

```

对TimeSpan的迭代方式相同。

第19.15节： DateTime.ParseExact(String, String, IFormatProvider)

将指定的日期和时间字符串表示形式转换为其等效的 DateTime，使用指定的格式和特定文化的格式信息。字符串表示的格式必须与指定的格式完全匹配。

将特定格式的字符串转换为等效的 DateTime

假设我们有一个特定文化的 DateTime 字符串 08-07-2016 11:30:12 PM，格式为 MM-dd-yyyy hh:mm:ss tt，我们希望将其转换为等效的 DateTime 对象

```

string str = "08-07-2016 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "MM-dd-yyyy hh:mm:ss tt", CultureInfo.CurrentCulture);

```

将日期时间字符串转换为等效的 DateTime 对象，无需任何特定文化格式

假设我们有一个日期时间字符串，格式为 dd-MM-yy hh:mm:ss tt，我们希望将其转换为等效的 DateTime 对象，无需任何特定文化信息

```

string str = "17-06-16 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "dd-MM-yy hh:mm:ss tt", CultureInfo.InvariantCulture);

```

将日期时间字符串转换为等效的 DateTime 对象，无需任何特定文化格式，且格式不同

```

DateTime dateResult;
var dutchDateString = "31 oktober 1999 04:20";
var dutchCulture = CultureInfo.CreateSpecificCulture("nl-NL");
DateTime.TryParse(dutchDateString, dutchCulture, styles, out dateResult);
// output {31/10/1999 04:20:00}

```

Example of Parse:

```

DateTime.Parse(dutchDateString, dutchCulture)
// output {31/10/1999 04:20:00}

```

Section 19.14: DateTime as initializer in for-loop

```

// This iterates through a range between two DateTimes
// with the given iterator (any of the Add methods)

DateTime start = new DateTime(2016, 01, 01);
DateTime until = new DateTime(2016, 02, 01);

// NOTICE: As the add methods return a new DateTime you have
// to overwrite dt in the iterator like dt = dt.Add()
for (DateTime dt = start; dt < until; dt = dt.AddDays(1))
{
    Console.WriteLine("Added {0} days. Resulting DateTime: {1}",
                      (dt - start).Days, dt.ToString());
}

```

Iterating on a TimeSpan works the same way.

Section 19.15: DateTime.ParseExact(String, String, IFormatProvider)

Converts the specified string representation of a date and time to its DateTime equivalent using the specified format and culture-specific format information. The format of the string representation must match the specified format exactly.

Convert a specific format string to equivalent DateTime

Let's say we have a culture-specific DateTime string 08-07-2016 11:30:12 PM as MM-dd-yyyy hh:mm:ss tt format and we want it to convert to equivalent DateTime object

```

string str = "08-07-2016 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "MM-dd-yyyy hh:mm:ss tt", CultureInfo.CurrentCulture);

```

Convert a date time string to equivalent DateTime object without any specific culture format

Let's say we have a DateTime string in dd-MM-yy hh:mm:ss tt format and we want it to convert to equivalent DateTime object, without any specific culture information

```

string str = "17-06-16 11:30:12 PM";
DateTime date = DateTime.ParseExact(str, "dd-MM-yy hh:mm:ss tt", CultureInfo.InvariantCulture);

```

Convert a date time string to equivalent DateTime object without any specific culture format with different format

假设我们有一个日期字符串，例如“23-12-2016”或“12/23/2016”，我们想将其转换为等效格式 DateTime对象，无特定的文化信息

```
string date = '23-12-2016' 或 date = 12/23/2016';
string[] formats = new string[] {"dd-MM-yyyy", "MM/dd/yyyy"}; // 甚至可以添加更多可能的
格式。
DateTime date = DateTime.ParseExact(date, formats,
CultureInfo.InvariantCulture, DateTimeStyles.None);
```

注意：System.Globalization 需要添加以使用 CultureInfo 类

第19.16节：DateTime 的 ToString、ToShortDateString、 ToLongDateString 和格式化 ToString

```
using System;

public class Program
{
    public static void Main()
    {
        var date = new DateTime(2016,12,31);

Console.WriteLine(date.ToString());          //输出：12/31/2016 12:00:00 AM
        Console.WriteLine(date.ToShortDateString()); //输出：12/31/2016
        Console.WriteLine(date.ToLongDateString()); //输出：Saturday, December 31, 2016
        Console.WriteLine(date.ToString("dd/MM/yyyy")); //输出：31/12/2016
    }
}
```

第19.17节：当前日期

要获取当前日期，可以使用DateTime.Today属性。该属性返回一个包含今天日期的DateTime对象。
当调用.ToString()方法转换时，默认会按照系统的本地设置进行转换。

例如：

```
Console.WriteLine(DateTime.Today);
```

将今天的日期以本地格式写入控制台。

Let's say we have a Date string , example like '23-12-2016' or '12/23/2016' and we want it to convert to equivalent DateTime object, without any specific culture information

```
string date = '23-12-2016' 或 date = 12/23/2016';
string[] formats = new string[] {"dd-MM-yyyy", "MM/dd/yyyy"}; // even can add more possible
formats.
DateTime date = DateTime.ParseExact(date, formats,
CultureInfo.InvariantCulture, DateTimeStyles.None);
```

NOTE : System.Globalization needs to be added for CultureInfo Class

Section 19.16: DateTime ToString, ToShortDateString, ToLongDateString and ToString formatted

```
using System;

public class Program
{
    public static void Main()
    {
        var date = new DateTime(2016,12,31);

Console.WriteLine(date.ToString());          //Outputs: 12/31/2016 12:00:00 AM
        Console.WriteLine(date.ToShortDateString()); //Outputs: 12/31/2016
        Console.WriteLine(date.ToLongDateString()); //Outputs: Saturday, December 31, 2016
        Console.WriteLine(date.ToString("dd/MM/yyyy")); //Outputs: 31/12/2016
    }
}
```

Section 19.17: Current Date

To get the current date you use the DateTime.Today property. This returns a DateTime object with today's date.
When this is then converted .ToString() it is done so in your system's locality by default.

For example:

```
Console.WriteLine(DateTime.Today);
```

Writes today's date, in your local format to the console.

第20章：数组

第20.1节：声明数组

可以使用方括号 ([]) 初始化语法声明数组并用默认值填充。例如，创建一个包含10个整数的数组：

```
int[] arr = new int[10];
```

C#中的索引是从零开始的。上述数组的索引范围是0到9。例如：

```
int[] arr = new int[3] {7,9,4};  
Console.WriteLine(arr[0]); //输出 7  
Console.WriteLine(arr[1]); //输出 9
```

这意味着系统从0开始计数元素索引。此外，对数组元素的访问是在常数时间内完成的。这意味着访问数组的第一个元素与访问第二个元素、第三个元素等的时间成本相同。

你也可以声明一个对数组的裸引用而不实例化数组。

```
int[] arr = null; // 可以，声明了一个指向数组的空引用。  
int first = arr[0]; // 抛出 System.NullReferenceException，因为没有实际的数组。
```

数组也可以使用集合初始化语法创建并初始化自定义值：

```
int[] arr = new int[] { 24, 2, 13, 47, 45 };
```

声明数组变量时，new int[] 部分可以省略。这不是一个独立的表达式，因此作为其他调用的一部分使用时不起作用（对此，请使用带有new的版本）：

```
int[] arr = { 24, 2, 13, 47, 45 }; // 可以  
int[] arr1;  
arr1 = { 24, 2, 13, 47, 45 }; // 无法编译
```

隐式类型数组

或者，结合var关键字，可以省略具体类型，从而推断数组的类型：

```
// 与 int[] 相同  
var arr = new [] { 1, 2, 3 };  
// 与 string[] 相同  
var arr = new [] { "one", "two", "three" };  
// 与 double[] 相同  
var arr = new [] { 1.0, 2.0, 3.0 };
```

第20.2节：初始化填充有重复非默认值的数组

正如我们所知，可以声明一个带有默认值的数组：

```
int[] arr = new int[10];
```

Chapter 20: Arrays

Section 20.1: Declaring an array

An array can be declared and filled with the default value using square bracket ([]) initialization syntax. For example, creating an array of 10 integers:

```
int[] arr = new int[10];
```

Indices in C# are zero-based. The indices of the array above will be 0-9. For example:

```
int[] arr = new int[3] {7,9,4};  
Console.WriteLine(arr[0]); //outputs 7  
Console.WriteLine(arr[1]); //outputs 9
```

Which means the system starts counting the element index from 0. Moreover, accesses to elements of arrays are done in **constant time**. That means accessing to the first element of the array has the same cost (in time) of accessing the second element, the third element and so on.

You may also declare a bare reference to an array without instantiating an array.

```
int[] arr = null; // OK, declares a null reference to an array.  
int first = arr[0]; // Throws System.NullReferenceException because there is no actual array.
```

An array can also be created and initialized with custom values using collection initialization syntax:

```
int[] arr = new int[] { 24, 2, 13, 47, 45 };
```

The new int[] portion can be omitted when declaring an array variable. This is not a self-contained expression, so using it as part of a different call does not work (for that, use the version with new):

```
int[] arr = { 24, 2, 13, 47, 45 }; // OK  
int[] arr1;  
arr1 = { 24, 2, 13, 47, 45 }; // Won't compile
```

Implicitly typed arrays

Alternatively, in combination with the var keyword, the specific type may be omitted so that the type of the array is inferred:

```
// same as int[]  
var arr = new [] { 1, 2, 3 };  
// same as string[]  
var arr = new [] { "one", "two", "three" };  
// same as double[]  
var arr = new [] { 1.0, 2.0, 3.0 };
```

Section 20.2: Initializing an array filled with a repeated non-default value

As we know we can declare an array with default values:

```
int[] arr = new int[10];
```

这将创建一个包含10个整数的数组，数组的每个元素值均为0（类型int的默认值）。

要创建一个用非默认值初始化的数组，我们可以使用System.Linq

命名空间中的Enumerable.Repeat：

1. 创建一个大小为10且填充为“true”的bool数组

```
bool[] booleanArray = Enumerable.Repeat(true, 10).ToArray();
```

2. 创建一个大小为5且填充为“100”的int数组

```
int[] intArray = Enumerable.Repeat(100, 5).ToArray();
```

3. 创建一个大小为5且填充为“C#”的string数组

```
string[] strArray = Enumerable.Repeat("C#", 5).ToArray();
```

第20.3节：复制数组

使用静态方法Array.Copy()，从源数组和目标数组的索引0开始复制部分数组：

```
var sourceArray = new int[] { 11, 12, 3, 5, 2, 9, 28, 17 };
var destinationArray = new int[3];
Array.Copy(sourceArray, destinationArray, 3);
```

// destinationArray 将包含 11、12 和 3

使用 CopyTo() 实例方法复制整个数组，从源数组的索引 0 开始复制到目标数组的指定索引：

```
var sourceArray = new int[] { 11, 12, 7 };
var destinationArray = new int[6];
sourceArray.CopyTo(destinationArray, 2);
```

// destinationArray 将包含 0、0、11、12、7 和 0

Clone 用于创建数组对象的副本。

```
var sourceArray = new int[] { 11, 12, 7 };
var destinationArray = (int)sourceArray.Clone();
// destinationArray 将被创建并包含 11、12、7.
```

CopyTo 和 Clone 都执行浅拷贝，这意味着内容包含对与原数组元素相同对象的引用。

第 20.4 节：数组相等性比较

LINQ 提供了一个内置函数用于检查两个IEnumerable的相等性，该函数也可以用于数组。

如果数组长度相同且对应索引的值相等，SequenceEqual 函数将返回 true，否则返回 false。

This will create an array of 10 integers with each element of the array having value 0 (the default value of type int).

To create an array initialized with a non-default value, we can use [Enumerable.Repeat](#) from the [System.Linq](#) Namespace:

1. To create a bool array of size 10 filled with "true"

```
bool[] booleanArray = Enumerable.Repeat(true, 10).ToArray();
```

2. To create an int array of size 5 filled with "100"

```
int[] intArray = Enumerable.Repeat(100, 5).ToArray();
```

3. To create a string array of size 5 filled with "C#"

```
string[] strArray = Enumerable.Repeat("C#", 5).ToArray();
```

Section 20.3: Copying arrays

Copying a partial array with the static Array.Copy() method, beginning at index 0 in both, source and destination:

```
var sourceArray = new int[] { 11, 12, 3, 5, 2, 9, 28, 17 };
var destinationArray = new int[3];
Array.Copy(sourceArray, destinationArray, 3);
```

// destinationArray will have 11, 12 and 3

Copying the whole array with the CopyTo() instance method, beginning at index 0 of the source and the specified index in the destination:

```
var sourceArray = new int[] { 11, 12, 7 };
var destinationArray = new int[6];
sourceArray.CopyTo(destinationArray, 2);
```

// destinationArray will have 0, 0, 11, 12, 7 and 0

Clone is used to create a copy of an array object.

```
var sourceArray = new int[] { 11, 12, 7 };
var destinationArray = (int)sourceArray.Clone();
//destinationArray will be created and will have 11, 12, 7.
```

Both CopyTo and Clone perform shallow copy which means the contents contains references to the same object as the elements in the original array.

Section 20.4: Comparing arrays for equality

LINQ provides a built-in function for checking the equality of two IEnumerable, and that function can be used on arrays.

The [SequenceEqual](#) function will return [true](#) if the arrays have the same length and the values in corresponding indices are equal, and [false](#) otherwise.

```
int[] arr1 = { 3, 5, 7 };
int[] arr2 = { 3, 5, 7 };
bool result = arr1.SequenceEqual(arr2);
Console.WriteLine("Arrays equal? {0}", result);
```

这将打印：

```
数组相等? True
```

第20.5节：多维数组

数组可以有多个维度。以下示例创建了一个具有十行十列的二维数组：

```
int[,] arr = new int[10, 10];
```

一个三维数组：

```
int[,] arr = new int[10, 10, 10];
```

你也可以在声明时初始化数组：

```
int[,] arr = new int[4, 2] { {1, 1}, {2, 2}, {3, 3}, {4, 4} };
```

```
// 访问多维数组的成员：
Console.Out.WriteLine(arr[3, 1]); // 4
```

第20.6节：获取和设置数组值

```
int[] arr = new int[] { 0, 10, 20, 30};

// 获取
Console.WriteLine(arr[2]); // 20

// 设置
arr[2] = 100;

// 获取更新后的值
Console.WriteLine(arr[2]); // 100
```

第20.7节：遍历数组

```
int[] arr = new int[] {1, 6, 3, 3, 9};

for (int i = 0; i < arr.Length; i++)
{
    Console.WriteLine(arr[i]);
}
```

使用 foreach：

```
foreach (int element in arr)
{
    Console.WriteLine(element);
}
```

```
int[] arr1 = { 3, 5, 7 };
int[] arr2 = { 3, 5, 7 };
bool result = arr1.SequenceEqual(arr2);
Console.WriteLine("Arrays equal? {0}", result);
```

This will print:

```
Arrays equal? True
```

Section 20.5: Multi-dimensional arrays

Arrays can have more than one dimension. The following example creates a two-dimensional array of ten rows and ten columns:

```
int[,] arr = new int[10, 10];
```

An array of three dimensions:

```
int[,] arr = new int[10, 10, 10];
```

You can also initialize the array upon declaration:

```
int[,] arr = new int[4, 2] { {1, 1}, {2, 2}, {3, 3}, {4, 4} };
```

```
// Access a member of the multi-dimensional array:
Console.Out.WriteLine(arr[3, 1]); // 4
```

Section 20.6: Getting and setting array values

```
int[] arr = new int[] { 0, 10, 20, 30};

// Get
Console.WriteLine(arr[2]); // 20

// Set
arr[2] = 100;

// Get the updated value
Console.WriteLine(arr[2]); // 100
```

Section 20.7: Iterate over an array

```
int[] arr = new int[] {1, 6, 3, 3, 9};

for (int i = 0; i < arr.Length; i++)
{
    Console.WriteLine(arr[i]);
}
```

using foreach:

```
foreach (int element in arr)
{
    Console.WriteLine(element);
}
```

不安全

```

unsafe
{
    int length = arr.Length;
    fixed (int* p = arr)
    {
        int* pInt = p;
        while (length-- > 0)
        {
Console.WriteLine(*pInt);
            pInt++; // 指针移动到下一个元素
        }
    }
}

```

输出：

unsafe

```

unsafe
{
    int length = arr.Length;
    fixed (int* p = arr)
    {
        int* pInt = p;
        while (length-- > 0)
        {
            Console.WriteLine(*pInt);
            pInt++; // move pointer to next element
        }
    }
}

```

Output:

```

1
6
3
3
9

```

Section 20.8: Creating an array of sequential numbers

LINQ provides a method that makes it easy to create a collection filled with sequential numbers. For example, you can declare an array which contains the integers between 1 and 100.

The [Enumerable.Range](#) method allows us to create sequence of integer numbers from a specified start position and a number of elements.

The method takes two arguments: the starting value and the number of elements to generate.

```
Enumerable.Range(int start, int count)
```

Note that count cannot be negative.

Usage:

```
int[] sequence = Enumerable.Range(1, 100).ToArray();
```

This will generate an array containing the numbers 1 through 100 ([1, 2, 3, ..., 98, 99, 100]).

Because the Range method returns an [IEnumerable<int>](#), we can use other LINQ methods on it:

```
int[] squares = Enumerable.Range(2, 10).Select(x => x * x).ToArray();
```

This will generate an array that contains 10 integer squares starting at 4: [4, 9, 16, ..., 100, 121].

Section 20.9: Jagged arrays

Jagged arrays are arrays that instead of primitive types, contain arrays (or other collections). It's like an array of arrays - each array element contains another array.

它们类似于多维数组，但有一点不同——多维数组的行数和列数是固定的，而锯齿数组中，每一行的列数可以不同。

They are similar to multidimensional arrays, but have a slight difference - as multidimensional arrays are limited to a fixed number of rows and columns, with jagged arrays, every row can have a different number of columns.

声明锯齿数组

例如，声明一个有8列的锯齿数组：

```
int[][] a = new int[8][];
```

第二个[]没有指定大小。要初始化子数组，需要单独进行：

```
for (int i = 0; i < a.length; i++)
{
    a[i] = new int[10];
}
```

获取/设置值

现在，获取其中一个子数组很简单。让我们打印 a 的第3列的所有数字：

```
for (int i = 0; i < a[2].length; i++)
{
    Console.WriteLine(a[2][i]);
}
```

获取特定值：

```
a[<row_number>][<column_number>]
```

设置特定值：

```
a[<row_number>][<column_number>] = <value>
```

注意：建议始终使用锯齿状数组（数组的数组）而非多维数组（矩阵）。使用锯齿状数组更快且更安全。

关于括号顺序的说明

考虑一个三维数组，元素是五维数组，元素又是一维int数组。在C#中写作：

```
int[,,][,,,][] arr = new int[8, 10, 12][,,,][];
```

在CLR类型系统中，括号的排序约定是反过来的，因此对于上述arr实例我们有：

```
arr.GetType().ToString() == "System.Int32[[,,][,,][,]]"
```

同样地：

```
typeof(int[,,][,,,][]).ToString() == "System.Int32[[,,][,,][,]]"
```

Declaring a jagged array

For example, declaring a jagged array with 8 columns:

```
int[][] a = new int[8][];
```

The second [] is initialized without a number. To initialize the sub arrays, you would need to do that separately:

```
for (int i = 0; i < a.length; i++)
{
    a[i] = new int[10];
}
```

Getting/Setting values

Now, getting one of the subarrays is easy. Let's print all the numbers of the 3rd column of a:

```
for (int i = 0; i < a[2].length; i++)
{
    Console.WriteLine(a[2][i]);
}
```

Getting a specific value:

```
a[<row_number>][<column_number>]
```

Setting a specific value:

```
a[<row_number>][<column_number>] = <value>
```

Remember: It's always recommended to use jagged arrays (arrays of arrays) rather than multidimensional arrays (matrixes). It's faster and safer to use.

Note on the order of the brackets

Consider a three-dimensional array of five-dimensional arrays of one-dimensional arrays of int. This is written in C# as:

```
int[,,][,,,][] arr = new int[8, 10, 12][,,,][];
```

In the CLR type system, the convention for the ordering of the brackets is reversed, so with the above arr instance we have:

```
arr.GetType().ToString() == "System.Int32[[,,][,,][,]]"
```

and likewise:

```
typeof(int[,,][,,,][]).ToString() == "System.Int32[[,,][,,][,]]"
```

第20.10节：数组协变

```
string[] strings = new[] {"foo", "bar"};
object[] objects = strings; // 从string[]到object[]的隐式转换
```

这种转换不是类型安全的。以下代码将在运行时引发异常：

```
string[] strings = new[] {"Foo"};
object[] objects = strings;

objects[0] = new object(); // 运行时异常, object不是string
string str = strings[0]; // 如果上面的赋值成功, 这里将会出错
```

第20.11节：数组作为IEnumerable<>实例

所有数组都实现了非泛型的 IList 接口（因此也实现了非泛型的 ICollection 和 IEnumerable 基础接口）。

更重要的是，一维数组实现了 IList<> 和 IReadOnlyList<> 泛型接口（以及它们的基接口），针对它们所包含的数据类型。这意味着它们可以被视为泛型可枚举类型，并且可以传递给各种方法，而无需先将它们转换为非数组形式。

```
int[] arr1 = { 3, 5, 7 };
IEnumerable<int> enumerableIntegers = arr1; //允许, 因为数组实现了 IEnumerable<T>
List<int> listOfIntegers = new List<int>();
listOfIntegers.AddRange(arr1); //你可以传入数组的引用来自填充 List.
```

运行这段代码后，列表 listOfIntegers 将包含一个 List<int>，里面的值是 3、5 和 7。

对 IEnumerable<> 的支持意味着数组可以使用 LINQ 查询，例如 arr1.Select(i => 10 * i)。

第20.12节：检查一个数组是否包含另一个数组

```
public static class ArrayHelpers
{
    public static bool Contains<T>(this T[] array, T[] candidate)
    {
        if (IsEmptyLocate(array, candidate))
            return false;

        if (candidate.Length > array.Length)
            return false;

        for (int a = 0; a <= array.Length - candidate.Length; a++)
        {
            if (array[a].Equals(candidate[0]))
            {
                int i = 0;
                for (; i < candidate.Length; i++)
                {
                    if (false == array[a + i].Equals(candidate[i]))
                        break;
                }
                if (i == candidate.Length)
                    return true;
            }
        }
    }
}
```

Section 20.10: Array covariance

```
string[] strings = new[] {"foo", "bar"};
object[] objects = strings; // implicit conversion from string[] to object[]
```

This conversion is not type-safe. The following code will raise a runtime exception:

```
string[] strings = new[] {"Foo"};
object[] objects = strings;

objects[0] = new object(); // runtime exception, object is not string
string str = strings[0]; // would have been bad if above assignment had succeeded
```

Section 20.11: Arrays as IEnumerable<> instances

All arrays implement the non-generic IList interface (and hence non-generic ICollection and IEnumerable base interfaces).

More importantly, one-dimensional arrays implement the IList<> and IReadOnlyList<> generic interfaces (and their base interfaces) for the type of data that they contain. This means that they can be treated as generic enumerable types and passed in to a variety of methods without needing to first convert them to a non-array form.

```
int[] arr1 = { 3, 5, 7 };
IEnumerable<int> enumerableIntegers = arr1; //Allowed because arrays implement IEnumerable<T>
List<int> listOfIntegers = new List<int>();
listOfIntegers.AddRange(arr1); //You can pass in a reference to an array to populate a List.
```

After running this code, the list listOfIntegers will contain a List<int> containing the values 3, 5, and 7.

The IEnumerable<> support means arrays can be queried with LINQ, for example arr1.Select(i => 10 * i).

Section 20.12: Checking if one array contains another array

```
public static class ArrayHelpers
{
    public static bool Contains<T>(this T[] array, T[] candidate)
    {
        if (IsEmptyLocate(array, candidate))
            return false;

        if (candidate.Length > array.Length)
            return false;

        for (int a = 0; a <= array.Length - candidate.Length; a++)
        {
            if (array[a].Equals(candidate[0]))
            {
                int i = 0;
                for (; i < candidate.Length; i++)
                {
                    if (false == array[a + i].Equals(candidate[i]))
                        break;
                }
                if (i == candidate.Length)
                    return true;
            }
        }
    }
}
```

```
        return false;
    }

    static bool IsEmptyLocate<T>(T[] array, T[] candidate)
    {
        return array == null
            || candidate == null
            || array.Length == 0
            || candidate.Length == 0
            || candidate.Length > array.Length;
    }
}
```

//示例

```
byte[] EndOfStream = Encoding.ASCII.GetBytes("---3141592---");
byte[] FakeReceivedFromStream = Encoding.ASCII.GetBytes("Hello, world!!---3141592---");
if (FakeReceivedFromStream.Contains(EndOfStream))
{
    Console.WriteLine("消息已接收");
}
```

```
        return false;
    }

    static bool IsEmptyLocate<T>(T[] array, T[] candidate)
    {
        return array == null
            || candidate == null
            || array.Length == 0
            || candidate.Length == 0
            || candidate.Length > array.Length;
    }
}
```

// Sample

```
byte[] EndOfStream = Encoding.ASCII.GetBytes("---3141592---");
byte[] FakeReceivedFromStream = Encoding.ASCII.GetBytes("Hello, world!!---3141592---");
if (FakeReceivedFromStream.Contains(EndOfStream))
{
    Console.WriteLine("Message received");
}
```

第21章：数组循环旋转的O(n)算法

在我学习编程的过程中，有一些简单但有趣的问题作为练习。其中一个问题是将数组（或其他集合）按一定的值进行旋转。在这里，我将与您分享一个简单的公式来实现它。

第21.1节：按给定偏移量旋转数组的通用方法示例

我想指出，当偏移值为负时，我们向左旋转；当偏移值为正时，我们向右旋转。

```
public static void Main()
{
    int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int shiftCount = 1;
    Rotate(ref array, shiftCount);
    Console.WriteLine(string.Join(", ", array));
    // 输出: [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]

array = new []{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
shiftCount = 15;
Rotate(ref array, shiftCount);
Console.WriteLine(string.Join(", ", array));
// 输出: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]

array = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
shiftCount = -1;
Rotate(ref array, shiftCount);
Console.WriteLine(string.Join(", ", array));
// 输出: [2, 3, 4, 5, 6, 7, 8, 9, 10, 1]

array = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
shiftCount = -35;
Rotate(ref array, shiftCount);
Console.WriteLine(string.Join(", ", array));
// 输出: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]

}

private static void Rotate<T>(ref T[] array, int shiftCount)
{
    T[] backupArray= new T[array.Length];

    for (int index = 0; index < array.Length; index++)
    {
        backupArray[(index + array.Length + shiftCount % array.Length) % array.Length] =
array[index];
    }

array = backupArray;
}
```

这段代码中重要的是我们用来在旋转后找到新索引值的公式。

$(index + array.Length + shiftCount \% array.Length) \% array.Length$

Chapter 21: O(n) Algorithm for circular rotation of an array

In my path to studying programming there have been simple, but interesting problems to solve as exercises. One of those problems was to rotate an array(or another collection) by a certain value. Here I will share with you a simple formula to do it.

Section 21.1: Example of a generic method that rotates an array by a given shift

I would like to point out that we rotate left when the shifting value is negative and we rotate right when the value is positive.

```
public static void Main()
{
    int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int shiftCount = 1;
    Rotate(ref array, shiftCount);
    Console.WriteLine(string.Join(", ", array));
    // Output: [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]

array = new []{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
shiftCount = 15;
Rotate(ref array, shiftCount);
Console.WriteLine(string.Join(", ", array));
// Output: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]

array = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
shiftCount = -1;
Rotate(ref array, shiftCount);
Console.WriteLine(string.Join(", ", array));
// Output: [2, 3, 4, 5, 6, 7, 8, 9, 10, 1]

array = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
shiftCount = -35;
Rotate(ref array, shiftCount);
Console.WriteLine(string.Join(", ", array));
// Output: [6, 7, 8, 9, 10, 1, 2, 3, 4, 5]

}

private static void Rotate<T>(ref T[] array, int shiftCount)
{
    T[] backupArray= new T[array.Length];

    for (int index = 0; index < array.Length; index++)
    {
        backupArray[(index + array.Length + shiftCount % array.Length) % array.Length] =
array[index];
    }

array = backupArray;
}
```

The thing that is important in this code is the formula with which we find the new index value after the rotation.

$(index + array.Length + shiftCount \% array.Length) \% array.Length$

这里有一些关于它的更多信息：

(shiftCount % array.Length) -> 我们将移位值规范化到数组长度范围内（因为在长度为10的数组中，移位1或11是相同的，-1和-11也是如此）。

array.Length + (shiftCount % array.Length) -> 这是针对左旋转操作，确保不会出现负索引，而是将其旋转到数组末尾。否则，对于长度为10的数组，索引0和旋转-1会得到负数 (-1)，无法得到正确的旋转索引值9。 $(10 + (-1 \% 10) = 9)$ **index + array.Length + (shiftCount % array.Length)** -> 这里没什么好说的，我们将旋转应用到索引上以获得新的索引值。 $(0 + 10 + (-1 \% 10) = 9)$

index + array.Length + (shiftCount % array.Length) % array.Length -> 第二次规范化确保新的索引值不会超出数组范围，而是循环回数组开头。这是针对右旋转的，因为在长度为10的数组中，如果没有它，索引9和旋转1会得到索引10，超出数组范围，无法得到正确的旋转索引值0。 $((9 + 10 + (1 \% 10)) \% 10 = 0)$

Here is a little more information about it:

(shiftCount % array.Length) -> we normalize the shifting value to be in the length of the array (since in an array with length 10, shifting 1 or 11 is the same thing, the same goes for -1 and -11).

array.Length + (shiftCount % array.Length) -> this is done due to left rotations to make sure we do not go into a negative index, but rotate it to the end of the array. Without it for an array with length 10 for index 0 and a rotation -1 we would go into a negative number (-1) and not get the real rotation index value, which is 9. $(10 + (-1 \% 10) = 9)$

index + array.Length + (shiftCount % array.Length) -> not much to say here as we apply the rotation to the index to get the new index. $(0 + 10 + (-1 \% 10) = 9)$

index + array.Length + (shiftCount % array.Length) % array.Length -> the second normalization is making sure that the new index value does not go outside of the array, but rotates the value in the beginning of the array. It is for right rotations, since in an array with length 10 without it for index 9 and a rotation 1 we would go into index 10, which is outside of the array, and not get the real rotation index value is 0. $((9 + 10 + (1 \% 10)) \% 10 = 0)$

第22章：枚举 (Enum)

枚举可以派生自以下任意类型：byte、sbyte、short、ushort、int、uint、long、ulong。默认类型是int，可以通过在枚举定义中指定类型来更改：

```
public enum Weekday : byte { Monday = 1, Tuesday = 2, Wednesday = 3, Thursday = 4, Friday = 5 }
```

这在调用本地代码 (P/Invoke)、映射数据源等场景中非常有用。一般情况下，应使用默认的int类型，因为大多数开发者期望枚举是int类型。

第22.1节：枚举基础

来自MSDN：

枚举类型（也称为枚举或enum）提供了一种高效的方式来定义一组命名的整型常量，这些常量可以赋值给变量。

本质上，枚举是一种只允许有限选项集合的类型，每个选项对应一个数字。默认情况下，这些数字按声明的顺序递增，从零开始。例如，可以声明一个表示一周七天的枚举：

```
public enum Day
{
    星期一,
    星期二,
    星期三,
    星期四,
    星期五,
    星期六,
    星期日
}
```

该枚举可以这样使用：

```
// 定义对应特定日期值的变量
Day myFavoriteDay = Day.星期五;
Day myLeastFavoriteDay = Day.星期一;

// 获取对应myFavoriteDay的整数值
// 星期五的数字是4
int myFavoriteDayIndex = (int)myFavoriteDay;

// 获取表示数字5的那一天
Day dayFive = (Day)5;
```

默认情况下，枚举 (enum) 中每个元素的底层类型是int，但也可以使用byte、sbyte、short、ushort、uint、long 和 ulong。如果使用除int以外的类型，必须在枚举名称后用冒号指定类型：

```
public enum Day : byte
{
    // 与之前相同
}
```

Chapter 22: Enum

An enum can derive from any of the following types: byte, sbyte, short, ushort, int, uint, long, ulong. The default is int, and can be changed by specifying the type in the enum definition:

```
public enum Weekday : byte { Monday = 1, Tuesday = 2, Wednesday = 3, Thursday = 4, Friday = 5 }
```

This is useful when P/Invoking to native code, mapping to data sources, and similar circumstances. In general, the default int should be used, because most developers expect an enum to be an int.

Section 22.1: Enum basics

From [MSDN](#):

An enumeration type (also named an enumeration or an enum) provides an efficient way to define a set of named **integral constants** that may be **assigned to a variable**.

Essentially, an enum is a type that only allows a set of finite options, and each option corresponds to a number. By default, those numbers are increasing in the order the values are declared, starting from zero. For example, one could declare an enum for the days of the week:

```
public enum Day
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

That enum could be used like this:

```
// Define variables with values corresponding to specific days
Day myFavoriteDay = Day.Friday;
Day myLeastFavoriteDay = Day.Monday;

// Get the int that corresponds to myFavoriteDay
// Friday is number 4
int myFavoriteDayIndex = (int)myFavoriteDay;

// Get the day that represents number 5
Day dayFive = (Day)5;
```

By default the underlying type of each element in the `enum` is `int`, but `byte`, `sbyte`, `short`, `ushort`, `uint`, `long` and `ulong` can be used as well. If you use a type other than `int`, you must specify the type using a colon after the enum name:

```
public enum Day : byte
{
    // same as before
}
```

名称后的数字现在是字节类型而不是整数。你可以如下获取枚举的底层类型：

```
Enum.GetUnderlyingType(typeof(Days));
```

输出：

```
System.Byte
```

示例：[.NET fiddle](#)

第22.2节：作为标志的枚举

可以将FlagsAttribute应用于枚举，改变ToString()的行为以匹配枚举的特性：

```
[Flags]
enum MyEnum
{
    //无 = 0, 可以使用但不能在按位操作中组合
    FlagA = 1,
    FlagB = 2,
    FlagC = 4,
    FlagD = 8
    //必须使用2的幂或2的幂的组合
    //才能使按位操作生效
}

var twoFlags = MyEnum.FlagA | MyEnum.FlagB;

// 这将枚举变量中的所有标志："FlagA, FlagB"。
Console.WriteLine(twoFlags);
```

因为 FlagsAttribute 依赖于枚举常量是2的幂（或它们的组合），且枚举值最终是数值类型，所以你受到底层数值类型大小的限制。你可以使用的最大数值类型是 UInt64，它允许你指定64个不同的（非组合的）标志枚举常量。 enum 关键字默认的底层类型是 int，即 Int32。编译器允许声明超过32位的值，但这些值会无警告地回绕，导致两个或多个枚举成员具有相同的值。因此，如果枚举需要容纳超过32个标志的位集合，你需要显式指定更大的类型：

```
public enum BigEnum : ulong
{
    BigValue = 1 << 63
}
```

虽然标志通常只有一位，但它们可以组合成命名的“集合”以便更容易使用。

```
[Flags]
enum FlagsEnum
{
    None = 0,
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,

    Default = Option1 | Option3,
```

The numbers after the name are now bytes instead of integers. You could get the underlying type of the enum as follows:

```
Enum.GetUnderlyingType(typeof(Days));
```

Output:

```
System.Byte
```

Demo: [.NET fiddle](#)

Section 22.2: Enum as flags

The FlagsAttribute can be applied to an enum changing the behaviour of the ToString() to match the nature of the enum:

```
[Flags]
enum MyEnum
{
    //None = 0, can be used but not combined in bitwise operations
    FlagA = 1,
    FlagB = 2,
    FlagC = 4,
    FlagD = 8
    //you must use powers of two or combinations of powers of two
    //for bitwise operations to work
}

var twoFlags = MyEnum.FlagA | MyEnum.FlagB;

// This will enumerate all the flags in the variable: "FlagA, FlagB".
Console.WriteLine(twoFlags);
```

Because FlagsAttribute relies on the enumeration constants to be powers of two (or their combinations) and enum values are ultimately numeric values, you are limited by the size of the underlying numeric type. The largest available numeric type that you can use is UInt64, which allows you to specify 64 distinct (non-combined) flag enum constants. The enum keyword defaults to the underlying type int, which is Int32. The compiler will allow the declaration of values wider than 32 bit. Those will wrap around without a warning and result in two or more enum members of the same value. Therefore, if an enum is meant to accommodate a bitset of more than 32 flags, you need to specify a bigger type explicitly:

```
public enum BigEnum : ulong
{
    BigValue = 1 << 63
}
```

Although flags are often only a single bit, they can be combined into named "sets" for easier use.

```
[Flags]
enum FlagsEnum
{
    None = 0,
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,

    Default = Option1 | Option3,
```

```
All = Option1 | Option2 | Option3,  
}
```

为了避免拼写出2的幂的十进制值，也可以使用左移运算符 (`<<`) 来声明相同的枚举

```
[Flags]  
enum FlagsEnum  
{  
    None = 0,  
    Option1 = 1 << 0,  
    Option2 = 1 << 1,  
    Option3 = 1 << 2,  
  
    Default = Option1 | Option3,  
    All = Option1 | Option2 | Option3,  
}
```

从 C# 7.0 开始，也可以使用二进制字面量。

要检查枚举变量的值是否设置了某个标志，可以使用 `HasFlag` 方法。假设我们有

```
[Flags]  
enum MyEnum  
{  
    One = 1,  
    Two = 2,  
    Three = 4  
}
```

和一个 值

```
var value = MyEnum.One | MyEnum.Two;
```

使用 `HasFlag` 我们可以检查是否设置了任意标志

```
if(value.HasFlag(MyEnum.One))  
    Console.WriteLine("枚举包含 One");  
  
if(value.HasFlag(MyEnum.Two))  
    Console.WriteLine("枚举包含 Two");  
  
if(value.HasFlag(MyEnum.Three))  
    Console.WriteLine("枚举包含 Three");
```

我们也可以遍历枚举的所有值，以获取所有已设置的标志

```
var type = typeof(MyEnum);  
var names = Enum.GetNames(type);  
  
foreach (var name in names)  
{  
    var item = (MyEnum)Enum.Parse(type, name);  
  
    if (value.HasFlag(item))  
        Console.WriteLine("枚举有 " + name);  
}
```

```
All = Option1 | Option2 | Option3,  
}
```

To avoid spelling out the decimal values of powers of two, the [left-shift operator \(`<<`\)](#) can also be used to declare the same enum

```
[Flags]  
enum FlagsEnum  
{  
    None = 0,  
    Option1 = 1 << 0,  
    Option2 = 1 << 1,  
    Option3 = 1 << 2,  
  
    Default = Option1 | Option3,  
    All = Option1 | Option2 | Option3,  
}
```

Starting with C# 7.0, binary literals can be used too.

To check if the value of enum variable has a certain flag set, the [HasFlag](#) method can be used. Let's say we have

```
[Flags]  
enum MyEnum  
{  
    One = 1,  
    Two = 2,  
    Three = 4  
}
```

And a `value`

```
var value = MyEnum.One | MyEnum.Two;
```

With `HasFlag` we can check if any of the flags is set

```
if(value.HasFlag(MyEnum.One))  
    Console.WriteLine("Enum has One");  
  
if(value.HasFlag(MyEnum.Two))  
    Console.WriteLine("Enum has Two");  
  
if(value.HasFlag(MyEnum.Three))  
    Console.WriteLine("Enum has Three");
```

Also we can iterate through all values of enum to get all flags that are set

```
var type = typeof(MyEnum);  
var names = Enum.GetNames(type);  
  
foreach (var name in names)  
{  
    var item = (MyEnum)Enum.Parse(type, name);  
  
    if (value.HasFlag(item))  
        Console.WriteLine("Enum has " + name);  
}
```

或者

```
foreach(MyEnum flagToCheck in Enum.GetValues(typeof(MyEnum)))
{
    if(value.HasFlag(flagToCheck))
    {
        Console.WriteLine("枚举包含 " + flagToCheck);
    }
}
```

所有三个示例将输出：

```
枚举 包含 One
枚举 包含 Two
```

第22.3节：使用 << 符号表示标志

左移运算符 (<<) 可以用于标志枚举声明中，以确保每个标志在二进制表示中恰好有一个1，正如标志所应有的那样。

这也有助于提高包含大量标志的大型枚举的可读性。

```
[Flags]
public enum MyEnum
{
    None = 0,
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2,
    Flag4 = 1 << 3,
    Flag5 = 1 << 4,
    ...
    Flag31 = 1 << 30
}
```

现在很明显，MyEnum 只包含合适的标志，而没有像 Flag30 = 1073741822（或二进制的 11111111111111111111111111111110）这样不合适的混乱内容。

第22.4节：使用按位逻辑测试标志风格的枚举值

标志风格的枚举值需要用按位逻辑进行测试，因为它可能不匹配任何单一值。

```
[Flags]
enum FlagsEnum
{
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,
    Option2And3 = Option2 | Option3;

    默认 = 选项1 | 选项3,
}
```

默认值实际上是另外两个值通过按位或合并的结果。因此，要测试标志的存在，我们需要使用按位与操作。

```
var value = FlagsEnum.Default;
```

Or

```
foreach(MyEnum flagToCheck in Enum.GetValues(typeof(MyEnum)))
{
    if(value.HasFlag(flagToCheck))
    {
        Console.WriteLine("Enum has " + flagToCheck);
    }
}
```

All three examples will print:

```
Enum has One
Enum has Two
```

Section 22.3: Using << notation for flags

The left-shift operator (<<) can be used in flag enum declarations to ensure that each flag has exactly one 1 in binary representation, as flags should.

This also helps to improve readability of large enums with plenty of flags in them.

```
[Flags]
public enum MyEnum
{
    None = 0,
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2,
    Flag4 = 1 << 3,
    Flag5 = 1 << 4,
    ...
    Flag31 = 1 << 30
}
```

It is obvious now that MyEnum contains proper flags only and not any messy stuff like Flag30 = 1073741822 (or 11111111111111111111111111111110 in binary) which is inappropriate.

Section 22.4: Test flags-style enum values with bitwise logic

A flags-style enum value needs to be tested with bitwise logic because it may not match any single value.

```
[Flags]
enum FlagsEnum
{
    Option1 = 1,
    Option2 = 2,
    Option3 = 4,
    Option2And3 = Option2 | Option3;

    Default = Option1 | Option3,
}
```

The Default value is actually a combination of two others merged with a bitwise OR. Therefore to test for the presence of a flag we need to use a bitwise AND.

```
var value = FlagsEnum.Default;
```

```

bool isOption2And3Set = (value & FlagsEnum.Option2And3) == FlagsEnum.Option2And3;
Assert.True(isOption2And3Set);

```

第22.5节：向带标志的枚举添加和移除值

这段代码用于向带标志的枚举实例添加和移除值：

```

[Flags]
public enum MyEnum
{
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2
}

var value = MyEnum.Flag1;

// 设置附加值
value |= MyEnum.Flag2; // 现在value是Flag1, Flag2
value |= MyEnum.Flag3; // 现在value是Flag1, Flag2, Flag3

// 移除标志
value &= ~MyEnum.Flag2; // 现在value是Flag1, Flag3

```

第22.6节：枚举与字符串相互转换

```

public enum 星期
{
    星期日,
    星期一,
    星期二,
    星期三,
    星期四,
    星期五,
    星期六
}

// 枚举转字符串
string thursday = 星期.星期四.ToString(); // "Thursday"

string seventhDay = Enum.GetName(typeof(星期), 6); // "Saturday"

string monday = Enum.GetName(typeof(星期), 星期.星期一); // "Monday"

```

```

// 字符串转枚举 (仅限.NET 4.0及以上版本 - 早期.NET版本的替代语法见下文)
星期 tuesday;
Enum.TryParse("Tuesday", out tuesday); // 星期.星期二

星期 sunday;
bool matchFound1 = Enum.TryParse("SUNDAY", out sunday); // 返回false (区分大小写匹配)

星期 wednesday;
bool matchFound2 = Enum.TryParse("WEDNESDAY", true, out wednesday); // 返回true;
星期.星期三 (不区分大小写匹配)

```

```
// 字符串转枚举 (适用于所有 .NET 版本)
```

```

bool isOption2And3Set = (value & FlagsEnum.Option2And3) == FlagsEnum.Option2And3;
Assert.True(isOption2And3Set);

```

Section 22.5: Add and remove values from flagged enum

This code is to add and remove a value from a flagged enum-instance:

```

[Flags]
public enum MyEnum
{
    Flag1 = 1 << 0,
    Flag2 = 1 << 1,
    Flag3 = 1 << 2
}

var value = MyEnum.Flag1;

// set additional value
value |= MyEnum.Flag2; //value is now Flag1, Flag2
value |= MyEnum.Flag3; //value is now Flag1, Flag2, Flag3

// remove flag
value &= ~MyEnum.Flag2; //value is now Flag1, Flag3

```

Section 22.6: Enum to string and back

```

public enum DayOfWeek
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}

// Enum to string
string thursday = DayOfWeek.Thursday.ToString(); // "Thursday"

string seventhDay = Enum.GetName(typeof(DayOfWeek), 6); // "Saturday"

string monday = Enum.GetName(typeof(DayOfWeek), DayOfWeek.Monday); // "Monday"

// String to enum (.NET 4.0+ only - see below for alternative syntax for earlier .NET versions)
DayOfWeek tuesday;
Enum.TryParse("Tuesday", out tuesday); // DayOfWeek.Tuesday

DayOfWeek sunday;
bool matchFound1 = Enum.TryParse("SUNDAY", out sunday); // Returns false (case-sensitive match)

DayOfWeek wednesday;
bool matchFound2 = Enum.TryParse("WEDNESDAY", true, out wednesday); // Returns true;
DayOfWeek.Wednesday (case-insensitive match)

// String to enum (all .NET versions)

```

```

DayOfWeek friday = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Friday"); // DayOfWeek.Friday

DayOfWeek caturday = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Caturady"); // Throws
ArgumentException

// 所有枚举类型的名称作为字符串
string[] weekdays = Enum.GetNames(typeof(DayOfWeek));

```

第22.7节：枚举可能有意外的值

由于枚举可以被转换为其底层的整型，也可以从整型转换回来，值可能会超出枚举类型定义中的取值范围。

虽然下面的枚举类型DaysOfWeek只定义了7个值，但它仍然可以保存任何int值。

```

public enum DaysOfweek
{
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
    Sunday = 7
}

```

```

DaysOfweek d = (DaysOfweek)31;
Console.WriteLine(d); // 输出31

```

```

DaysOfweek s = DaysOfweek.Sunday;
s++; // 无错误

```

目前没有办法定义一个不具有此行为的枚举。

但是，可以使用方法Enum.IsDefined来检测未定义的枚举值。例如，

```

DaysOfweek d = (DaysOfweek)31;
Console.WriteLine(Enum.IsDefined(typeof(DaysOfweek),d)); // 输出 False

```

第22.8节：枚举的默认值 == 零

枚举的默认值是零。如果枚举没有定义值为零的项，其默认值将是零。

```

public class Program
{
    enum EnumExample
    {
        one = 1,
        two = 2
    }

    public void Main()
    {
        var e = default(EnumExample);

        if (e == EnumExample.one)
            Console.WriteLine("defaults to one");
    }
}

```

```

DayOfWeek friday = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Friday"); // DayOfWeek.Friday

DayOfWeek caturday = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), "Caturady"); // Throws
ArgumentException

// All names of an enum type as strings
string[] weekdays = Enum.GetNames(typeof(DayOfWeek));

```

Section 22.7: Enums can have unexpected values

Since an enum can be cast to and from its underlying integral type, the value may fall outside the range of values given in the definition of the enum type.

Although the below enum type DaysOfWeek only has 7 defined values, it can still hold any int value.

```

public enum DaysOfWeek
{
    Monday = 1,
    Tuesday = 2,
    Wednesday = 3,
    Thursday = 4,
    Friday = 5,
    Saturday = 6,
    Sunday = 7
}

```

```

DaysOfWeek d = (DaysOfWeek)31;
Console.WriteLine(d); // prints 31

```

```

DaysOfWeek s = DaysOfWeek.Sunday;
s++; // No error

```

There is currently no way to define an enum which does not have this behavior.

However, undefined enum values can be detected by using the method `Enum.IsDefined`. For example,

```

DaysOfWeek d = (DaysOfWeek)31;
Console.WriteLine(Enum.IsDefined(typeof(DaysOfWeek),d)); // prints False

```

Section 22.8: Default value for enum == ZERO

The default value for an enum is zero. If an enum does not define an item with a value of zero, its default value will be zero.

```

public class Program
{
    enum EnumExample
    {
        one = 1,
        two = 2
    }

    public void Main()
    {
        var e = default(EnumExample);

        if (e == EnumExample.one)
            Console.WriteLine("defaults to one");
    }
}

```

```

    else
Console.WriteLine("未知");
}
}

```

示例：<https://dotnetfiddle.net/l5Rwie>

第22.9节：为枚举值添加额外的描述信息

在某些情况下，你可能想为枚举值添加额外的描述，例如当枚举值本身不如你想展示给用户的内容易读时。在这种情况下，你可以使用

[System.ComponentModel.DescriptionAttribute](#) 类。

例如：

```

public enum PossibleResults
{
    [Description("成功")]
    OK = 1,
    [Description("文件未找到")]
    FileNotFound = 2,
    [Description("访问被拒绝")]
    AccessDenied = 3
}

```

现在，如果你想返回某个特定枚举值的描述，可以这样做：

```

public static string GetDescriptionAttribute(PossibleResults result)
{
    返回
((DescriptionAttribute)Attribute.GetCustomAttribute((result.GetType().GetField(result.ToString())),
typeof(DescriptionAttribute))).Description;
}

static void Main(string[] args)
{
    PossibleResults result = PossibleResults.FileNotFound;
    Console.WriteLine(result); // 输出 "FileNotFound"
    Console.WriteLine(GetDescriptionAttribute(result)); // 输出 "文件未找到"
}

```

这也可以很容易地转换为所有枚举的扩展方法：

```

static class EnumExtensions
{
    public static string GetDescription(this Enum enumValue)
    {
        返回
((DescriptionAttribute)Attribute.GetCustomAttribute((enumValue.GetType().GetField(enumValue.ToString())),
typeof(DescriptionAttribute))).Description;
    }
}

```

然后可以像这样轻松使用：Console.WriteLine(result.GetDescription());

```

    else
Console.WriteLine("Unknown");
}
}

```

Example: <https://dotnetfiddle.net/l5Rwie>

Section 22.9: Adding additional description information to an enum value

In some cases you might want to add an additional description to an enum value, for instance when the enum value itself is less readable than what you might want to display to the user. In such cases you can use the [System.ComponentModel.DescriptionAttribute](#) class.

For example:

```

public enum PossibleResults
{
    [Description("Success")]
    OK = 1,
    [Description("File not found")]
    FileNotFound = 2,
    [Description("Access denied")]
    AccessDenied = 3
}

```

Now, if you would like to return the description of a specific enum value you can do the following:

```

public static string GetDescriptionAttribute(PossibleResults result)
{
    return
((DescriptionAttribute)Attribute.GetCustomAttribute((result.GetType().GetField(result.ToString())),
typeof(DescriptionAttribute))).Description;
}

static void Main(string[] args)
{
    PossibleResults result = PossibleResults.FileNotFound;
    Console.WriteLine(result); // Prints "FileNotFound"
    Console.WriteLine(GetDescriptionAttribute(result)); // Prints "File not found"
}

```

This can also be easily transformed to an extension method for all enums:

```

static class EnumExtensions
{
    public static string GetDescription(this Enum enumValue)
    {
        return
((DescriptionAttribute)Attribute.GetCustomAttribute((enumValue.GetType().GetField(enumValue.ToString(),
typeof(DescriptionAttribute))).Description;
    }
}

```

And then easily used like this: Console.WriteLine(result.GetDescription());

第22.10节：获取枚举的所有成员值

```
enum MyEnum
{
    一,
    二,
    三
}

foreach(MyEnum e in Enum.GetValues(typeof(MyEnum)))
    Console.WriteLine(e);
```

这将打印：

```
一  
二  
三
```

第22.11节：使用枚举进行按位操作

当枚举表示一组标志而非单一值时，应使用FlagsAttribute。分配给每个枚举值的数值在使用按位运算符操作枚举时非常有用。

示例1：带有[Flags]

```
[Flags]
枚举 Colors
{
    红色=1,
    蓝色=2,
    绿色=4,
    黄色=8
}

var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

输出 红色,蓝色

示例2：不带[Flags]

```
enum Colors
{
    红色=1,
    蓝色=2,
    绿色=4,
    黄色=8
}

var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

打印 3

Section 22.10: Get all the members values of an enum

```
enum MyEnum
{
    One,
    Two,
    Three
}

foreach(MyEnum e in Enum.GetValues(typeof(MyEnum)))
    Console.WriteLine(e);
```

This will print:

```
One  
Two  
Three
```

Section 22.11: Bitwise Manipulation using enums

The [FlagsAttribute](#) should be used whenever the enumerable represents a collection of flags, rather than a single value. The numeric value assigned to each enum value helps when manipulating enums using bitwise operators.

Example 1 : With [Flags]

```
[Flags]
enum Colors
{
    Red=1,
    Blue=2,
    Green=4,
    Yellow=8
}

var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

prints Red,Blue

Example 2 : Without [Flags]

```
enum Colors
{
    Red=1,
    Blue=2,
    Green=4,
    Yellow=8
}

var color = Colors.Red | Colors.Blue;
Console.WriteLine(color.ToString());
```

prints 3

第23章：元组

第23.1节：访问元组元素

要访问元组元素，请使用Item1-Item8属性。只有索引号小于或等于元组大小的属性才可用（例如，不能访问Tuple<T1,T2>中的Item3属性）。

```
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
var item1 = tuple.Item1; // "foo"
var item2 = tuple.Item2; // 123
var item3 = tuple.Item3; // true
var item4 = tuple.Item4; // new MyClass()
```

第23.2节：创建元组

元组是使用泛型类型Tuple<T1>-Tuple<T1,T2,T3,T4,T5,T6,T7,T8>创建的。每种类型代表一个包含1到8个元素的元组。元素可以是不同类型。

```
// 具有4个元素的元组
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
```

元组也可以使用静态Tuple.Create方法创建。在这种情况下，元素的类型由C#编译器推断。

```
// 具有4个元素的元组
var tuple = Tuple.Create("foo", 123, true, new MyClass());
```

版本 ≥ 7.0

自C# 7.0起，可以使用ValueTuple轻松创建元组。

```
var tuple = ("foo", 123, true, new MyClass());
```

元素可以命名以便更容易地分解。

```
(int number, bool flag, MyClass instance) tuple = (123, true, new MyClass());
```

第23.3节：比较和排序元组

元组可以基于其元素进行比较。

例如，可以对元素类型为Tuple的可枚举集合，基于指定元素上定义的比较运算符进行排序：

```
List<Tuple<int, string>> list = new List<Tuple<int, string>>();
list.Add(new Tuple<int, string>(2, "foo"));
list.Add(new Tuple<int, string>(1, "bar"));
list.Add(new Tuple<int, string>(3, "qux"));

list.Sort((a, b) => a.Item2.CompareTo(b.Item2)); //根据字符串元素排序

foreach (var element in list) {
    Console.WriteLine(element);
}

// 输出：
```

Chapter 23: Tuples

Section 23.1: Accessing tuple elements

To access tuple elements use Item1-Item8 properties. Only the properties with index number less or equal to tuple size are going to be available (i.e. one cannot access Item3 property in Tuple<T1, T2>).

```
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
var item1 = tuple.Item1; // "foo"
var item2 = tuple.Item2; // 123
var item3 = tuple.Item3; // true
var item4 = tuple.Item4; // new MyClass()
```

Section 23.2: Creating tuples

Tuples are created using generic types Tuple<T1>-Tuple<T1,T2,T3,T4,T5,T6,T7,T8>. Each of the types represents a tuple containing 1 to 8 elements. Elements can be of different types.

```
// tuple with 4 elements
var tuple = new Tuple<string, int, bool, MyClass>("foo", 123, true, new MyClass());
```

Tuples can also be created using static Tuple.Create methods. In this case, the types of the elements are inferred by the C# Compiler.

```
// tuple with 4 elements
var tuple = Tuple.Create("foo", 123, true, new MyClass());
```

Since C# 7.0, Tuples can be easily created using ValueTuple.

```
var tuple = ("foo", 123, true, new MyClass());
```

Elements can be named for easier decomposition.

```
(int number, bool flag, MyClass instance) tuple = (123, true, new MyClass());
```

Section 23.3: Comparing and sorting Tuples

Tuples can be compared based on their elements.

As an example, an enumerable whose elements are of type Tuple can be sorted based on comparisons operators defined on a specified element:

```
List<Tuple<int, string>> list = new List<Tuple<int, string>>();
list.Add(new Tuple<int, string>(2, "foo"));
list.Add(new Tuple<int, string>(1, "bar"));
list.Add(new Tuple<int, string>(3, "qux"));

list.Sort((a, b) => a.Item2.CompareTo(b.Item2)); //sort based on the string element

foreach (var element in list) {
    Console.WriteLine(element);
}

// Output:
```

```
// (1, bar)  
// (2, foo)  
// (3, qux)
```

或者使用以下方式反转排序：

```
list.Sort((a, b) => b.Item2.CompareTo(a.Item2));
```

第23.4节：从方法返回多个值

元组可以用来从方法返回多个值，而无需使用 `out` 参数。在下面的示例中，`AddMultiply` 用于返回两个值（和，积）。

```
void Write()  
{  
    var result = AddMultiply(25, 28);  
    Console.WriteLine(result.Item1);  
    Console.WriteLine(result.Item2);  
}  
  
Tuple<int, int> AddMultiply(int a, int b)  
{  
    return new Tuple<int, int>(a + b, a * b);  
}
```

输出：

```
53  
700
```

```
// (1, bar)  
// (2, foo)  
// (3, qux)
```

Or to reverse the sort use:

```
list.Sort((a, b) => b.Item2.CompareTo(a.Item2));
```

Section 23.4: Return multiple values from a method

Tuples can be used to return multiple values from a method without using `out` parameters. In the following example `AddMultiply` is used to return two values (`sum, product`).

```
void Write()  
{  
    var result = AddMultiply(25, 28);  
    Console.WriteLine(result.Item1);  
    Console.WriteLine(result.Item2);  
}  
  
Tuple<int, int> AddMultiply(int a, int b)  
{  
    return new Tuple<int, int>(a + b, a * b);  
}
```

Output:

```
53  
700
```

Now C# 7.0 offers an alternative way to return multiple values from methods using value tuples [More info about ValueTuple struct](#).

第24章：Guid

GUID（或UUID）是“全局唯一标识符”（或“通用唯一标识符”）的缩写。它是一个128位的整数，用于标识资源。

第24.1节：获取Guid的字符串表示

可以使用内置的ToString方法获取Guid的字符串表示

```
string myGuidString = myGuid.ToString();
```

根据需要，也可以通过向ToString调用添加格式类型参数来格式化Guid。

```
var guid = new Guid("7feb16f-651b-43b0-a5e3-0da8da49e90d");

// None      "7feb16f651b43b0a5e30da8da49e90d"
Console.WriteLine(guid.ToString("N"));

// 连字符    "7feb16f-651b-43b0-a5e3-0da8da49e90d"
Console.WriteLine(guid.ToString("D"));

// 大括号    "{7feb16f-651b-43b0-a5e3-0da8da49e90d}"
Console.WriteLine(guid.ToString("B"));

// 圆括号    "(7feb16f-651b-43b0-a5e3-0da8da49e90d)"
Console.WriteLine(guid.ToString("P"));

// 十六进制   "{0x7feb16f,0x651b,0x43b0{0xa5,0xe3,0x0d,0xa8,0xda,0x49,0xe9,0x0d}}"
Console.WriteLine(guid.ToString("X"));
```

第24.2节：创建 Guid

以下是创建 Guid 实例的最常见方法：

- 创建一个空的 GUID (00000000-0000-0000-0000-000000000000) :

```
Guid g = Guid.Empty;
Guid g2 = new Guid();
```

- 创建一个新的（伪随机）Guid :

```
Guid g = Guid.NewGuid();
```

- 使用特定值创建Guid :

```
Guid g = new Guid("0b214de7-8958-4956-8eed-28f9ba2c47c6");
Guid g2 = new Guid("0b214de7895849568eed28f9ba2c47c6");
Guid g3 = Guid.Parse("0b214de7-8958-4956-8eed-28f9ba2c47c6");
```

第24.3节：声明可空的GUID

像其他值类型一样，GUID也有一个可空类型，可以取null值。

声明：

```
Guid? myGuidVar = null;
```

Chapter 24: Guid

GUID (or UUID) is an acronym for 'Globally Unique Identifier' (or 'Universally Unique Identifier'). It is a 128-bit integer number used to identify resources.

Section 24.1: Getting the string representation of a Guid

A string representation of a Guid can be obtained by using the built in ToString method

```
string myGuidString = myGuid.ToString();
```

Depending on your needs you can also format the Guid, by adding a format type argument to the ToString call.

```
var guid = new Guid("7feb16f-651b-43b0-a5e3-0da8da49e90d");

// None      "7feb16f651b43b0a5e30da8da49e90d"
Console.WriteLine(guid.ToString("N"));

// Hyphens   "7feb16f-651b-43b0-a5e3-0da8da49e90d"
Console.WriteLine(guid.ToString("D"));

// Braces    "{7feb16f-651b-43b0-a5e3-0da8da49e90d}"
Console.WriteLine(guid.ToString("B"));

// Parentheses "(7feb16f-651b-43b0-a5e3-0da8da49e90d)"
Console.WriteLine(guid.ToString("P"));

// Hex       "{0x7feb16f,0x651b,0x43b0{0xa5,0xe3,0x0d,0xa8,0xda,0x49,0xe9,0x0d}}"
Console.WriteLine(guid.ToString("X"));
```

Section 24.2: Creating a Guid

These are the most common ways to create an instance of Guid:

- Creating an empty guid (00000000-0000-0000-0000-000000000000):

```
Guid g = Guid.Empty;
Guid g2 = new Guid();
```

- Creating a new (pseudorandom) Guid:

```
Guid g = Guid.NewGuid();
```

- Creating Guids with a specific value:

```
Guid g = new Guid("0b214de7-8958-4956-8eed-28f9ba2c47c6");
Guid g2 = new Guid("0b214de7895849568eed28f9ba2c47c6");
Guid g3 = Guid.Parse("0b214de7-8958-4956-8eed-28f9ba2c47c6");
```

Section 24.3: Declaring a nullable GUID

Like other value types, GUID also has a nullable type which can take null value.

Declaration :

```
Guid? myGuidVar = null;
```

当从数据库检索数据时，如果表中的值可能为NULL，这尤其有用。

This is particularly useful when retrieving data from the data base when there is a possibility that value from a table is NULL.

第25章：大整数 (BigInteger)

第25.1节：计算第一个1000位的斐波那契数

包含usingSystem.Numerics，并向项目添加对System.Numerics的引用。

```
using System;
using System.Numerics;

namespace Euler_25
{
    class Program
    {
        static void Main(string[] args)
        {
            BigInteger l1=1;
            BigInteger l2=1;
            BigInteger current=l1+l2;
            while(current.ToString().Length<1000)
            {
                l2=l1;
                l1=current;
                current=l1+l2;
            }
            Console.WriteLine(current);
        }
    }
}
```

这个简单的算法迭代斐波那契数，直到找到一个至少有1000位十进制数字的数，然后打印出来。这个值远远大于任何ulong类型所能表示的范围。

理论上， BigInteger类唯一的限制是您的应用程序可以使用的内存大小。

注意： BigInteger 仅在 .NET 4.0 及更高版本中可用。

Chapter 25: BigInteger

Section 25.1: Calculate the First 1,000-Digit Fibonacci Number

Include `using System.Numerics` and add a reference to `System.Numerics` to the project.

```
using System;
using System.Numerics;

namespace Euler_25
{
    class Program
    {
        static void Main(string[] args)
        {
            BigInteger l1 = 1;
            BigInteger l2 = 1;
            BigInteger current = l1 + l2;
            while (current.ToString().Length < 1000)
            {
                l2 = l1;
                l1 = current;
                current = l1 + l2;
            }
            Console.WriteLine(current);
        }
    }
}
```

This simple algorithm iterates through Fibonacci numbers until it reaches one at least 1000 decimal digits in length, then prints it out. This value is significantly larger than even a `ulong` could hold.

Theoretically, the only limit on the `BigInteger` class is the amount of RAM your application can consume.

Note: `BigInteger` is only available in .NET 4.0 and higher.

第26章：集合初始化器

第26.1节：集合初始化器

使用值初始化集合类型：

```
var stringList = new List<string>
{
    "foo",
    "bar",
};
```

集合初始化器是对Add()调用的语法糖。上述代码等价于：

```
var temp = new List<string>();
temp.Add("foo");
temp.Add("bar");
var stringList = temp;
```

注意，初始化是通过临时变量原子完成的，以避免竞态条件。

对于Add()方法提供多个参数的类型，将逗号分隔的参数用大括号括起来：

```
var numberDictionary = new Dictionary<int, string>
{
    { 1, "One" },
    { 2, "Two" },
};
```

这等同于：

```
var temp = new Dictionary<int, string>();
temp.Add(1, "One");
temp.Add(2, "Two");
var numberDictionary = temp;
```

第26.2节：C# 6 索引初始化器

从C# 6开始，可以通过指定方括号中的索引，后跟等号，再后跟要赋的值，来初始化带有索引器的集合。

字典初始化

使用Dictionary的此语法示例：

```
var dict = new Dictionary<string, int>
{
    ["key1"] = 1,
    ["key2"] = 50
};
```

这等同于：

```
var dict = new Dictionary<string, int>();
```

Chapter 26: Collection Initializers

Section 26.1: Collection initializers

Initialize a collection type with values:

```
var stringList = new List<string>
{
    "foo",
    "bar",
};
```

Collection initializers are syntactic sugar for `Add()` calls. Above code is equivalent to:

```
var temp = new List<string>();
temp.Add("foo");
temp.Add("bar");
var stringList = temp;
```

Note that the initialization is done atomically using a temporary variable, to avoid race conditions.

For types that offer multiple parameters in their `Add()` method, enclose the comma-separated arguments in curly braces:

```
var numberDictionary = new Dictionary<int, string>
{
    { 1, "One" },
    { 2, "Two" },
};
```

This is equivalent to:

```
var temp = new Dictionary<int, string>();
temp.Add(1, "One");
temp.Add(2, "Two");
var numberDictionary = temp;
```

Section 26.2: C# 6 Index Initializers

Starting with C# 6, collections with indexers can be initialized by specifying the index to assign in square brackets, followed by an equals sign, followed by the value to assign.

Dictionary Initialization

An example of this syntax using a Dictionary:

```
var dict = new Dictionary<string, int>
{
    ["key1"] = 1,
    ["key2"] = 50
};
```

This is equivalent to:

```
var dict = new Dictionary<string, int>();
```

```
dict["key1"] = 1;  
dict["key2"] = 50
```

在 C# 6 之前，实现此功能的集合初始化语法是：

```
var dict = new Dictionary<string, int>  
{  
    { "key1", 1 },  
    { "key2", 50 }  
};
```

这相当于：

```
var dict = new Dictionary<string, int>();  
dict.Add("key1", 1);  
dict.Add("key2", 50);
```

因此，功能上存在显著差异，因为新语法使用被初始化对象的索引器来赋值，而不是使用其 `Add()` 方法。这意味着新语法只需要一个公开可用的索引器，且适用于任何拥有索引器的对象。

```
public class IndexableClass  
{  
    public int this[int index]  
    {  
        set  
        {  
            Console.WriteLine("{0} 被分配到索引 {1}", value, index);  
        }  
    }  
  
    var foo = new IndexableClass  
    {  
        [0] = 10,  
        [1] = 20  
    }  
}
```

这将输出：

```
10 被分配到索引 0  
20 被分配到索引 1
```

第26.3节：自定义类中的集合初始化器

为了使类支持集合初始化器，它必须实现 `IEnumerable` 接口并至少有一个 `Add` 方法。自 C# 6 起，任何实现了 `IEnumerable` 的集合都可以通过扩展方法添加自定义的 `Add` 方法。

```
class Program  
{  
    static void Main()  
    {  
        var col = new MyCollection {  
            "foo",  
            { "bar", 3 },  
        };  
    }  
}
```

```
dict["key1"] = 1;  
dict["key2"] = 50
```

The collection initializer syntax to do this before C# 6 was:

```
var dict = new Dictionary<string, int>  
{  
    { "key1", 1 },  
    { "key2", 50 }  
};
```

Which would correspond to:

```
var dict = new Dictionary<string, int>();  
dict.Add("key1", 1);  
dict.Add("key2", 50);
```

So there is a significant difference in functionality, as the new syntax uses the `indexer` of the initialized object to assign values instead of using its `Add()` method. This means the new syntax only requires a publicly available indexer, and works for any object that has one.

```
public class IndexableClass  
{  
    public int this[int index]  
    {  
        set  
        {  
            Console.WriteLine("{0} 被分配到索引 {1}", value, index);  
        }  
    }  
  
    var foo = new IndexableClass  
    {  
        [0] = 10,  
        [1] = 20  
    }  
}
```

This would output:

```
10 被分配到索引 0  
20 被分配到索引 1
```

Section 26.3: Collection initializers in custom classes

To make a class support collection initializers, it must implement `IEnumerable` interface and have at least one `Add` method. Since C# 6, any collection implementing `IEnumerable` can be extended with custom `Add` methods using extension methods.

```
class Program  
{  
    static void Main()  
    {  
        var col = new MyCollection {  
            "foo",  
            { "bar", 3 },  
        };  
    }  
}
```

```

        "baz",
123.45d,
    };
}
}

class MyCollection : IEnumerable
{
    private IList list = new ArrayList();

    public void Add(string item)
    {
        list.Add(item);
    }

    public void Add(string item, int count)
    {
        for(int i=0;i< count;i++) {
            list.Add(item);
        }
    }

    public IEnumerator GetEnumerator()
    {
        return list.GetEnumerator();
    }
}

静态类 MyCollectionExtensions
{
    public static void Add(this MyCollection @this, double value) =>
        @this.Add(value.ToString());
}

```

第26.4节：在对象初始化器中使用集合初始化器

```

public class Tag
{
    public IList<string> Synonyms { get; set; }
}

```

Synonyms 是一个集合类型的属性。当使用对象初始化器语法创建Tag对象时，Synonyms也可以使用集合初始化器语法进行初始化：

```

Tag t = new Tag
{
    Synonyms = new List<string> {"c#", "c-sharp"}
};

```

集合属性可以是只读的，但仍支持集合初始化器语法。考虑这个修改后的示例（Synonyms属性现在有一个私有的设置器）：

```

public class Tag
{
    public Tag()
    {
        同义词 = new List<string>();
    }
}

```

```

        "baz",
123.45d,
    };
}
}

class MyCollection : IEnumerable
{
    private IList list = new ArrayList();

    public void Add(string item)
    {
        list.Add(item);
    }

    public void Add(string item, int count)
    {
        for(int i=0;i< count;i++) {
            list.Add(item);
        }
    }

    public IEnumerator GetEnumerator()
    {
        return list.GetEnumerator();
    }
}

static class MyCollectionExtensions
{
    public static void Add(this MyCollection @this, double value) =>
        @this.Add(value.ToString());
}

```

Section 26.4: Using collection initializer inside object initializer

```

public class Tag
{
    public IList<string> Synonyms { get; set; }
}

```

Synonyms is a collection-type property. When the Tag object is created using object initializer syntax, Synonyms can also be initialized with collection initializer syntax:

```

Tag t = new Tag
{
    Synonyms = new List<string> {"c#", "c-sharp"}
};

```

The collection property can be readonly and still support collection initializer syntax. Consider this modified example (Synonyms property now has a private setter):

```

public class Tag
{
    public Tag()
    {
        Synonyms = new List<string>();
    }
}

```

```
public IList<string> 同义词 { get; private set; }  
}
```

可以这样创建一个新的Tag对象：

```
Tag t = new Tag  
{  
    同义词 = {"c#", "c-sharp"}  
};
```

这是可行的，因为集合初始化器只是对Add()调用的语法糖。这里并没有创建新的列表，编译器只是生成对现有对象的Add()调用。

第26.5节：带参数数组的集合初始化器

你可以混合使用普通参数和参数数组：

```
public class 彩票 : IEnumerable{  
    public int[] 幸运数字;  
    public string 用户名;  
  
    public void Add(string 用户名, params int[] 幸运数字){  
        用户名 = 用户名;  
        彩票 = 幸运数字;  
    }  
}
```

现在可以使用以下语法：

```
var Tickets = new List<LotteryTicket>{  
    {"酷先生" , 35663, 35732, 12312, 75685},  
    {"布鲁斯" , 26874, 66677, 24546, 36483, 46768, 24632, 24527},  
    {"约翰·塞纳", 25446, 83356, 65536, 23783, 24567, 89337}  
}
```

```
public IList<string> Synonyms { get; private set; }  
}
```

A new Tag object can be created like this:

```
Tag t = new Tag  
{  
    Synonyms = {"c#", "c-sharp"}  
};
```

This works because collection initializers are just syntactic sugar over calls to [Add\(\)](#). There's no new list being created here, the compiler is just generating calls to [Add\(\)](#) on the existing object.

Section 26.5: Collection Initializers with Parameter Arrays

You can mix normal parameters and parameter arrays:

```
public class LotteryTicket : IEnumerable{  
    public int[] LuckyNumbers;  
    public string UserName;  
  
    public void Add(string userName, params int[] luckyNumbers){  
        UserName = userName;  
        Lottery = luckyNumbers;  
    }  
}
```

This syntax is now possible:

```
var Tickets = new List<LotteryTicket>{  
    {"Mr Cool" , 35663, 35732, 12312, 75685},  
    {"Bruce" , 26874, 66677, 24546, 36483, 46768, 24632, 24527},  
    {"John Cena", 25446, 83356, 65536, 23783, 24567, 89337}  
}
```

第27章：C#集合概述

第27.1节：HashSet<T>

这是一个唯一的集合，查找时间复杂度为O(1)。

```
HashSet<int> validStoryPointValues = new HashSet<int>() { 1, 2, 3, 5, 8, 13, 21 };
bool containsEight = validStoryPointValues.Contains(8); // O(1)
```

相比之下，在List上执行Contains的性能较差：

```
List<int> validStoryPointValues = new List<int>() { 1, 2, 3, 5, 8, 13, 21 };
bool containsEight = validStoryPointValues.Contains(8); // O(n)
```

HashSet.Contains使用哈希表，因此无论集合中有多少项，查找速度都非常快。

第27.2节：Dictionary< TKey, TValue >

Dictionary< TKey, TValue > 是一个映射。对于给定的键，字典中只能有一个值。

```
using System.Collections.Generic;

var people = new Dictionary<string, int>
{
    { "John", 30 }, { "Mary", 35 }, { "Jack", 40 }
};

// 读取数据
Console.WriteLine(people["John"]); // 30
Console.WriteLine(people["George"]); // 抛出 KeyNotFoundException

int age;
if (people.TryGetValue("Mary", out age))
{
    Console.WriteLine(age); // 35
}

// 添加和修改数据
people["John"] = 40; // 这样覆盖值是可以的
people.Add("John", 40); // 抛出 ArgumentException, 因为"John"已存在

// 遍历内容
foreach(KeyValuePair<string, int> person in people)
{
    Console.WriteLine("Name={0}, Age={1}", person.Key, person.Value);
}

foreach(string name in people.Keys)
{
    Console.WriteLine("Name={0}", name);
}

foreach(int age in people.Values)
{
    Console.WriteLine("Age={0}", age);
}
```

Chapter 27: An overview of C# collections

Section 27.1: HashSet<T>

This is a collection of unique items, with O(1) lookup.

```
HashSet<int> validStoryPointValues = new HashSet<int>() { 1, 2, 3, 5, 8, 13, 21 };
bool containsEight = validStoryPointValues.Contains(8); // O(1)
```

By way of comparison, doing a Contains on a List yields poorer performance:

```
List<int> validStoryPointValues = new List<int>() { 1, 2, 3, 5, 8, 13, 21 };
bool containsEight = validStoryPointValues.Contains(8); // O(n)
```

HashSet.Contains uses a hash table, so that lookups are extremely fast, regardless of the number of items in the collection.

Section 27.2: Dictionary< TKey, TValue >

Dictionary< TKey, TValue > is a map. For a given key there can be one value in the dictionary.

```
using System.Collections.Generic;

var people = new Dictionary<string, int>
{
    { "John", 30 }, { "Mary", 35 }, { "Jack", 40 }
};

// Reading data
Console.WriteLine(people["John"]); // 30
Console.WriteLine(people["George"]); // throws KeyNotFoundException

int age;
if (people.TryGetValue("Mary", out age))
{
    Console.WriteLine(age); // 35
}

// Adding and changing data
people["John"] = 40; // Overwriting values this way is ok
people.Add("John", 40); // Throws ArgumentException since "John" already exists

// Iterating through contents
foreach(KeyValuePair<string, int> person in people)
{
    Console.WriteLine("Name={0}, Age={1}", person.Key, person.Value);
}

foreach(string name in people.Keys)
{
    Console.WriteLine("Name={0}", name);
}

foreach(int age in people.Values)
{
    Console.WriteLine("Age={0}", age);
}
```

使用集合初始化时的重复键

```
var people = new Dictionary<string, int>
{
    { "John", 30 }, {"Mary", 35}, {"Jack", 40}, {"Jack", 40}
}; // 抛出 ArgumentException, 因为"Jack"已存在
```

第27.3节 : SortedSet<T>

```
// 创建一个空集合
var mySet = new SortedSet<int>();

// 添加内容
// 注意我们先添加2再添加1
mySet.Add(2);
mySet.Add(1);

// 枚举集合中的元素
foreach(var item in mySet)
{
    Console.WriteLine(item);
}

// 输出:
// 1
// 2
```

第27.4节 : T[] (T类型数组)

```
// 创建一个包含2个元素的数组
var myArray = new [] { "one", "two" };

// 遍历数组
foreach(var item in myArray)
{
    Console.WriteLine(item);
}

// 输出:
// -
// -

// 交换第一个位置的元素
// 注意所有集合的索引从0开始
myArray[0] = "something else";

// 再次遍历数组
foreach(var item in myArray)
{
    Console.WriteLine(item);
}

// 输出:
// something else
// -
```

Duplicate key when using collection initialization

```
var people = new Dictionary<string, int>
{
    { "John", 30 }, {"Mary", 35}, {"Jack", 40}, {"Jack", 40}
}; // throws ArgumentException since "Jack" already exists
```

Section 27.3: SortedSet<T>

```
// create an empty set
var mySet = new SortedSet<int>();

// add something
// note that we add 2 before we add 1
mySet.Add(2);
mySet.Add(1);

// enumerate through the set
foreach(var item in mySet)
{
    Console.WriteLine(item);
}

// output:
// 1
// 2
```

Section 27.4: T[] (Array of T)

```
// create an array with 2 elements
var myArray = new [] { "one", "two" };

// enumerate through the array
foreach(var item in myArray)
{
    Console.WriteLine(item);
}

// output:
// one
// two

// exchange the element on the first position
// note that all collections start with the index 0
myArray[0] = "something else";

// enumerate through the array again
foreach(var item in myArray)
{
    Console.WriteLine(item);
}

// output:
// something else
// -
```

第27.5节 : List<T>

List<T> 是一种给定类型的列表。可以添加、插入、删除项目，并通过索引访问。

```
using System.Collections.Generic;

var list = new List<int>() { 1, 2, 3, 4, 5 };
list.Add(6);
Console.WriteLine(list.Count); // 6
list.RemoveAt(3);
Console.WriteLine(list.Count); // 5
Console.WriteLine(list[3]); // 5
```

List<T> 可以被看作是一个可以调整大小的数组。按顺序枚举集合很快，通过索引访问单个元素也很快。要根据元素的某些值或其他键来访问元素，Dictionary<T> 将提供更快的查找速度。

第27.6节 : Stack<T>

```
// 初始化一个整数类型的栈对象
var stack = new Stack<int>();

// 添加一些数据
stack.Push(3);
stack.Push(5);
stack.Push(8);

// 元素以“先进后出”的顺序存储。
// 栈从顶到底：8, 5, 3

// 我们可以使用 peek 查看栈顶元素。
Console.WriteLine(stack.Peek()); // 输出 8

// Pop 移除栈顶元素并返回它。
Console.WriteLine(stack.Pop()); // 输出 8
Console.WriteLine(stack.Pop()); // 输出 5
Console.WriteLine(stack.Pop()); // 输出 3
```

第27.7节 : LinkedList<T>

```
// 初始化一个整数类型的 LinkedList
LinkedList list = new LinkedList<int>();

// 向列表中添加一些数字。
list.AddLast(3);
list.AddLast(5);
list.AddLast(8);

// 当前列表为 3, 5, 8

list.AddFirst(2);
// 现在列表为 2, 3, 5, 8

list.RemoveFirst();
// 列表现在是 3, 5, 8

list.RemoveLast();
// 列表现在是 3, 5
```

Section 27.5: List<T>

List<T> is a list of a given type. Items can be added, inserted, removed and addressed by index.

```
using System.Collections.Generic;

var list = new List<int>() { 1, 2, 3, 4, 5 };
list.Add(6);
Console.WriteLine(list.Count); // 6
list.RemoveAt(3);
Console.WriteLine(list.Count); // 5
Console.WriteLine(list[3]); // 5
```

List<T> can be thought of as an array that you can resize. Enumerating over the collection in order is quick, as is access to individual elements via their index. To access elements based on some aspect of their value, or some other key, a Dictionary<T> will provide faster lookup.

Section 27.6: Stack<T>

```
// Initialize a stack object of integers
var stack = new Stack<int>();

// add some data
stack.Push(3);
stack.Push(5);
stack.Push(8);

// elements are stored with "first in, last out" order.
// stack from top to bottom is: 8, 5, 3

// We can use peek to see the top element of the stack.
Console.WriteLine(stack.Peek()); // prints 8

// Pop removes the top element of the stack and returns it.
Console.WriteLine(stack.Pop()); // prints 8
Console.WriteLine(stack.Pop()); // prints 5
Console.WriteLine(stack.Pop()); // prints 3
```

Section 27.7: LinkedList<T>

```
// initialize a LinkedList of integers
LinkedList list = new LinkedList<int>();

// add some numbers to our list.
list.AddLast(3);
list.AddLast(5);
list.AddLast(8);

// the list currently is 3, 5, 8

list.AddFirst(2);
// the list now is 2, 3, 5, 8

list.RemoveFirst();
// the list is now 3, 5, 8

list.RemoveLast();
// the list is now 3, 5
```

注意 `LinkedList<T>` 表示的是 双向 链表。所以，它只是节点的集合，每个节点包含一个类型为 `T` 的元素。每个节点都链接到前一个节点和后一个节点。

第27.8节：队列

```
// 初始化一个新的整数队列  
var queue = new Queue<int>();  
  
// 添加一些数据  
queue.Enqueue(6);  
queue.Enqueue(4);  
queue.Enqueue(9);  
  
// 队列中的元素按“先进先出”的顺序存储。  
// 队列从第一个到最后一个是：6, 4, 9  
  
// 查看队列中的下一个元素，但不移除它。  
Console.WriteLine(queue.Peek()); // 输出 6  
  
// 移除队列中的第一个元素，并返回它。  
Console.WriteLine(queue.Dequeue()); // 输出 6  
Console.WriteLine(queue.Dequeue()); // 输出 4  
Console.WriteLine(queue.Dequeue()); // 输出 9
```

线程安全提醒！在多线程环境中使用[ConcurrentQueue](#)。

Note that `LinkedList<T>` represents the *doubly* linked list. So, it's simply collection of nodes and each node contains an element of type `T`. Each node is linked to the preceding node and the following node.

Section 27.8: Queue

```
// Initialize a new queue of integers  
var queue = new Queue<int>();  
  
// Add some data  
queue.Enqueue(6);  
queue.Enqueue(4);  
queue.Enqueue(9);  
  
// Elements in a queue are stored in "first in, first out" order.  
// The queue from first to last is: 6, 4, 9  
  
// View the next element in the queue, without removing it.  
Console.WriteLine(queue.Peek()); // prints 6  
  
// Removes the first element in the queue, and returns it.  
Console.WriteLine(queue.Dequeue()); // prints 6  
Console.WriteLine(queue.Dequeue()); // prints 4  
Console.WriteLine(queue.Dequeue()); // prints 9
```

Thread safe heads up! Use [ConcurrentQueue](#) in multi-thread environments.

第28章：循环

第28.1节：For循环

For循环非常适合执行固定次数的操作。它类似于While循环，但增量包含在条件中。

For循环的结构如下：

```
for (初始化; 条件; 增量)
{
    // 代码
}
```

初始化 - 创建一个只能在循环中使用的新局部变量。
条件 - 循环仅在条件为真时运行。
增量 - 变量每次循环时的变化方式。

一个例子：

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

输出：

Chapter 28: Looping

Section 28.1: For Loop

A For Loop is great for doing things a certain amount of time. It's like a While Loop but the increment is included with the condition.

A For Loop is set up like this:

```
for (Initialization; Condition; Increment)
{
    // Code
}
```

Initialization - Makes a new local variable that can only be used in the loop.
Condition - The loop only runs when the condition is true.
Increment - How the variable changes every time the loop runs.

An example:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

Output:

```
0
1
2
3
4
```

You can also leave out spaces in the For Loop, but you have to have all semicolons for it to function.

```
int input = Console.ReadLine();

for ( ; input < 10; input + 2)
{
    Console.WriteLine(input);
}
```

Output for 3:

```
3
5
7
9
11
```

第 28.2 节：Do - While 循环

它类似于 `while` 循环，但在循环体的 末尾 测试条件。Do - While 循环无论条件是否为真，都会执行一次循环。

```
int[] numbers = new int[] { 6, 7, 8, 10 };

// 从数组中累加值，直到总和大于 10,
// 或者用完所有值。
int sum = 0;
int i = 0;
do
{
    sum += numbers[i];
    i++;
} while (sum <= 10 && i < numbers.Length);

System.Console.WriteLine(sum); // 13
```

第28.3节：foreach循环

`foreach`将遍历任何实现了`IEnumerable`接口的类的对象（注意`IEnumerable<T>`继承自该接口）。此类对象包括一些内置类型，但不限于：`List<T>`, `T[]`（任意类型的数组），`Dictionary< TKey, TSource >`，以及像`IQueryable`和`ICollection`等接口。

语法

```
foreach(ItemType itemVariable in enumerableObject)
    statement;
```

备注

1. 类型`ItemType`不需要与元素的精确类型匹配，只需能够从元素类型赋值即可
项目的类型
2. 可以使用`var`代替`ItemType`, `var`会从`enumerableObject`中推断出项的类型
通过检查`IEnumerable`实现的泛型参数
3. 语句可以是代码块、单条语句，甚至是空语句`();`
4. 如果`enumerableObject`没有实现`IEnumerable`，代码将无法编译
5. 在每次迭代中，当前项会被强制转换为`ItemType`（即使未指定而由编译器推断）
通过`var`，如果项无法转换，则会抛出`InvalidCastException`异常。

请考虑以下示例：

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
foreach(var name in list)
{
    Console.WriteLine("Hello " + name);
}
```

等同于：

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
```

Section 28.2: Do - While Loop

It is similar to a `while` loop, except that it tests the condition at the *end* of the loop body. The Do - While loop executes the loop once irrespective of whether the condition is true or not.

```
int[] numbers = new int[] { 6, 7, 8, 10 };

// Sum values from the array until we get a total that's greater than 10,
// or until we run out of values.
int sum = 0;
int i = 0;
do
{
    sum += numbers[i];
    i++;
} while (sum <= 10 && i < numbers.Length);

System.Console.WriteLine(sum); // 13
```

Section 28.3: Foreach Loop

`foreach` will iterate over any object of a class that implements `IEnumerable` (take note that `IEnumerable<T>` inherits from it). Such objects include some built-in ones, but not limit to: `List<T>`, `T[]` (arrays of any type), `Dictionary< TKey, TSource >`, as well as interfaces like `IQueryable` and `ICollection`, etc.

syntax

```
foreach(ItemType itemVariable in enumerableObject)
    statement;
```

remarks

1. The type `ItemType` does not need to match the precise type of the items, it just needs to be assignable from the type of the items
2. Instead of `ItemType`, alternatively `var` can be used which will infer the items type from the `enumerableObject` by inspecting the generic argument of the `IEnumerable` implementation
3. The statement can be a block, a single statement or even an empty statement`();`
4. If `enumerableObject` is not implementing `IEnumerable`, the code will not compile
5. During each iteration the current item is cast to `ItemType` (even if this is not specified but compiler-inferred via `var`) and if the item cannot be cast an `InvalidCastException` will be thrown.

Consider this example:

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
foreach(var name in list)
{
    Console.WriteLine("Hello " + name);
}
```

is equivalent to:

```
var list = new List<string>();
list.Add("Ion");
list.Add("Andrei");
```

```
IEnumerator 枚举器;
try
{
    枚举器 = list.GetEnumerator();
    while(枚举器.MoveNext())
    {
        string 名字 = (string)枚举器.Current;
        Console.WriteLine("Hello " + 名字);
    }
}
finally
{
    if (枚举器 != null)
        枚举器.Dispose();
}
```

第28.4节：循环风格

While

最简单的循环类型。唯一的缺点是没有内在的线索来知道你在循环中的位置。

```
/// 当条件满足时循环
while(condition)
{
    /// 做某事
}
```

做

类似于while，但条件在循环结束时而非开始时进行判断。这导致循环至少执行一次。

```
do
{
    /// 做某事
} while(条件) /// 当条件满足时循环
```

For

另一种简单的循环方式。在循环中索引 (i) 会递增，你可以使用它。通常用于处理数组。

```
for ( int i = 0; i < array.Count; i++ )
{
    var currentItem = array[i];
    /// 对"currentItem"做某事
}
```

Foreach

现代化的遍历`IEnumerable`对象的方式。好处是你不必考虑项目的索引或列表的项目数量。

```
foreach ( var item in someList )
{
    /// 对"item"执行某些操作
}
```

```
IEnumerator enumerator;
try
{
    enumerator = list.GetEnumerator();
    while(enumerator.MoveNext())
    {
        string name = (string)enumerator.Current;
        Console.WriteLine("Hello " + name);
    }
}
finally
{
    if (enumerator != null)
        enumerator.Dispose();
}
```

Section 28.4: Looping styles

While

The most trivial loop type. Only drawback is there is no intrinsic clue to know where you are in the loop.

```
/// loop while the condition satisfies
while(condition)
{
    /// do something
}
```

Do

Similar to `while`, but the condition is evaluated at the end of the loop instead of the beginning. This results in executing the loops at least once.

```
do
{
    /// do something
} while(condition) /// loop while the condition satisfies
```

For

Another trivial loop style. While looping an index (i) gets increased and you can use it. It is usually used for handling arrays.

```
for ( int i = 0; i < array.Count; i++ )
{
    var currentItem = array[i];
    /// do something with "currentItem"
}
```

Foreach

Modernized way of looping through `IEnumerable` objects. Good thing that you don't have to think about the index of the item or the item count of the list.

```
foreach ( var item in someList )
{
    /// do something with "item"
}
```

```
}
```

Foreach 方法

虽然其他样式用于选择或更新集合中的元素，这种样式通常用于直接调用集合中所有元素的方法。

```
list.ForEach(item => item.DoSomething());  
  
// 或者  
list.ForEach(item => DoSomething(item));  
  
// 或者使用方法组  
list.ForEach(Console.WriteLine);  
  
// 使用数组  
Array.ForEach(myArray, Console.WriteLine);
```

需要注意的是，该方法仅适用于List<T>实例，并且作为Array的静态方法存在——它不是Linq的一部分。

Linq 并行 Foreach

与 Linq Foreach 类似，只不过这个是以并行方式执行。也就是说，集合中的所有项目将同时执行给定的操作。

```
collection.AsParallel().ForAll(item => item.DoSomething());  
  
/// 或者  
collection.AsParallel().ForAll(item => DoSomething(item));
```

第28.5节：嵌套循环

```
// 打印乘法表，直到5  
for (int i = 1; i <= 5; i++)  
{  
    for (int j = 1; j <= 5; j++)  
    {  
        int product = i * j;  
        Console.WriteLine("{0} times {1} is {2}", i, j, product);  
    }  
}
```

第28.6节：continue

除了break之外，还有关键字continue。它不会完全跳出循环，而是跳过当前的迭代。如果你不想在某个特定值被设置时执行某些代码，这会很有用。

这里有一个简单的例子：

```
for (int i = 1; i <= 10; i++)  
{  
    if (i < 9)  
        continue;  
  
    Console.WriteLine(i);
```

```
}
```

Foreach Method

While the other styles are used for selecting or updating the elements in collections, this style is usually used for calling a method straight away for all elements in a collection.

```
list.ForEach(item => item.DoSomething());  
  
// or  
list.ForEach(item => DoSomething(item));  
  
// or using a method group  
list.ForEach(Console.WriteLine);  
  
// using an array  
Array.ForEach(myArray, Console.WriteLine);
```

It is important to note that this method is only available on List<T> instances and as a static method on Array - it is **not** part of Linq.

Linq Parallel Foreach

Just like Linq Foreach, except this one does the job in a parallel manner. Meaning that all the items in the collection will run the given action at the same time, simultaneously.

```
collection.AsParallel().ForAll(item => item.DoSomething());  
  
/// or  
collection.AsParallel().ForAll(item => DoSomething(item));
```

Section 28.5: Nested loops

```
// Print the multiplication table up to 5s  
for (int i = 1; i <= 5; i++)  
{  
    for (int j = 1; j <= 5; j++)  
    {  
        int product = i * j;  
        Console.WriteLine("{0} times {1} is {2}", i, j, product);  
    }  
}
```

Section 28.6: continue

In addition to break, there is also the keyword continue. Instead of breaking completely the loop, it will simply skip the current iteration. It could be useful if you don't want some code to be executed if a particular value is set.

Here's a simple example:

```
for (int i = 1; i <= 10; i++)  
{  
    if (i < 9)  
        continue;  
  
    Console.WriteLine(i);
```

}

将产生以下结果：

9
1
0
注意

: Continue通常在while或do-while循环中最为有用。对于具有明确定义退出条件的for循环，可能不会有太大益处。

第28.7节：while循环

```
int n = 0;
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
```

输出：

}

Will result in:

9
10

Note: Continue is often most useful in while or do-while loops. For-loops, with well-defined exit conditions, may not benefit as much.

Section 28.7: While loop

```
int n = 0;
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
```

Output:

0
1
2
3
4

IEnumerators can be iterated with a while loop:

```
// Call a custom method that takes a count, and returns an IEnumerator for a list
// of strings with the names of the largest city metro areas.
IEnumerator<string> largestMetroAreas = GetLargestMetroAreas(4);

while (largestMetroAreas.MoveNext())
{
    Console.WriteLine(largestMetroAreas.Current);
}
```

Sample output:

Tokyo/Yokohama
New York Metro
Sao Paulo
Seoul/Incheon

Section 28.8: break

Sometimes loop condition should be checked in the middle of the loop. The former is arguably more elegant than the latter:

```
for (;;)
{
```

```
// 可以改变should_end_loop表达式值的前置条件代码
```

```
if (should_end_loop)
    break;
```

```
// 执行某些操作
```

```
}
```

替代方案：

```
bool endLoop = false;
for (; !endLoop;)
{
    // 可以设置 endLoop 标志的前置条件代码

    if (!endLoop)
    {
        // 执行某些操作
    }
}
```

注意：在嵌套循环和/或 switch 中，必须使用比简单的 break 更多的控制方式。

```
// precondition code that can change the value of should_end_loop expression
```

```
if (should_end_loop)
    break;
```

```
// do something
```

```
}
```

Alternative:

```
bool endLoop = false;
for (; !endLoop;)
{
    // precondition code that can set endLoop flag

    if (!endLoop)
    {
        // do something
    }
}
```

Note: In nested loops and/or switch must use more than just a simple break.

第29章：迭代器

第29.1节：使用 Yield 创建迭代器

迭代器生成枚举器。在 C# 中，枚举器是通过定义包含yield语句的方法、属性或索引器来生成的。

大多数方法通过正常的return语句将控制权返回给调用者，这会释放该方法本地的所有状态。相比之下，使用yield语句的方法允许在请求时向调用者返回多个值，同时保留返回这些值之间的本地状态。这些返回的值构成一个序列。迭代器中使用了两种类型的yield语句：

- yield return，返回控制权给调用者但保留状态。当控制权传回给它时，被调用者将从这一行继续执行。
- yield break，功能类似于普通的return语句——表示序列的结束。
普通的return语句本身在迭代器块中是非法的。

下面的示例演示了一个可用于生成斐波那契数列的迭代器方法：

```
IEnumerable<int> Fibonacci(int count)
{
    int prev = 1;
    int curr = 1;

    for (int i = 0; i < count; i++)
    {
        yield return prev;
        int temp = prev + curr;
        prev = curr;
        curr = temp;
    }
}
```

然后可以使用此迭代器生成斐波那契数列的枚举器，供调用方法使用。下面的代码演示了如何枚举斐波那契数列中的前十项：

```
void Main()
{
    foreach (int term in Fibonacci(10))
    {
        Console.WriteLine(term);
    }
}
```

输出

13

Chapter 29: Iterators

Section 29.1: Creating Iterators Using Yield

Iterators produce enumerators. In C#, enumerators are produced by defining methods, properties or indexers that contain `yield` statements.

Most methods will return control to their caller through normal `return` statements, which disposes all state local to that method. In contrast, methods that use `yield` statements allow them to return multiple values to the caller on request while *preserving* local state in-between returning those values. These returned values constitute a sequence. There are two types of `yield` statements used within iterators:

- `yield return`, which returns control to the caller but preserves state. The callee will continue execution from this line when control is passed back to it.
- `yield break`, which functions similarly to a normal `return` statement - this signifies the end of the sequence.
Normal `return` statements themselves are illegal within an iterator block.

This example below demonstrates an iterator method that can be used to generate the [Fibonacci sequence](#):

```
IEnumerable<int> Fibonacci(int count)
{
    int prev = 1;
    int curr = 1;

    for (int i = 0; i < count; i++)
    {
        yield return prev;
        int temp = prev + curr;
        prev = curr;
        curr = temp;
    }
}
```

This iterator can then be used to produce an enumerator of the Fibonacci sequence that can be consumed by a calling method. The code below demonstrates how the first ten terms within the Fibonacci sequence can be enumerated:

```
void Main()
{
    foreach (int term in Fibonacci(10))
    {
        Console.WriteLine(term);
    }
}
```

Output

1
1
2
3
5
8
13
21

Section 29.2: Simple Numeric Iterator Example

A common use-case for iterators is to perform some operation over a collection of numbers. The example below demonstrates how each element within an array of numbers can be individually printed out to the console.

This is possible because arrays implement the `IEnumerable` interface, allowing clients to obtain an iterator for the array using the `GetEnumerator()` method. This method returns an *enumerator*, which is a read-only, forward-only cursor over each number in the array.

```
int[] numbers = { 1, 2, 3, 4, 5 };

IEnumerator iterator = numbers.GetEnumerator();

while (iterator.MoveNext())
{
    Console.WriteLine(iterator.Current);
}
```

Output

```
1  
2  
3  
4  
5
```

It's also possible to achieve the same results using a `foreach` statement:

```
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

第30章：IEnumerable

IEnumerable是所有非泛型集合（如ArrayList）可枚举的基础接口。

IEnumerator<T>是所有泛型枚举器（如List<>）的基础接口。

IEnumerable是一个接口，包含GetEnumerator方法。该GetEnumerator方法返回一个 IEnumerator，提供了类似foreach的遍历集合的选项。

第30.1节：带自定义枚举器的IEnumerable

实现IEnumerable接口允许类像BCL集合一样被枚举。这需要扩展跟踪枚举状态的Enumerator类。

除了遍历标准集合，示例还包括：

- 使用基于函数的数字范围而非对象集合
- 在集合上实现不同的迭代算法，如图集合上的深度优先搜索（DFS）或广度优先搜索（BFS）

```
public static void Main(string[] args) {  
  
    foreach (var coffee in new CoffeeCollection()) {  
        Console.WriteLine(coffee);  
    }  
  
}  
  
public class CoffeeCollection : IEnumerable {  
    private CoffeeEnumerator enumerator;  
  
    public CoffeeCollection() {  
        enumerator = new CoffeeEnumerator();  
    }  
  
    public IEnumerator GetEnumerator() {  
        return enumerator;  
    }  
  
    public class CoffeeEnumerator : IEnumerator {  
        string[] beverages = new string[3] { "espresso", "macchiato", "latte" };  
        int currentIndex = -1;  
  
        public object Current {  
            get {  
                return beverages[currentIndex];  
            }  
        }  
  
        public bool MoveNext() {  
            currentIndex++;  
  
            if (currentIndex < beverages.Length) {  
                return true;  
            }  
  
            return false;  
        }  
  
        public void Reset() {  
            currentIndex = 0;  
        }  
    }  
}
```

Chapter 30: IEnumerable

IEnumerable is the base interface for all non-generic collections like ArrayList that can be enumerated.

IEnumerator<T> is the base interface for all generic enumerators like List<>.

IEnumerable is an interface which implements the method GetEnumerator. The GetEnumerator method returns an IEnumerator which provides options to iterate through the collection like foreach.

Section 30.1: IEnumerable with custom Enumerator

Implementing the IEnumerable interface allows classes to be enumerated in the same way as BCL collections. This requires extending the Enumerator class which tracks the state of the enumeration.

Other than iterating over a standard collection, examples include:

- Using ranges of numbers based on a function rather than a collection of objects
- Implementing different iteration algorithms over collections, like DFS or BFS on a graph collection

```
public static void Main(string[] args) {  
  
    foreach (var coffee in new CoffeeCollection()) {  
        Console.WriteLine(coffee);  
    }  
  
}  
  
public class CoffeeCollection : IEnumerable {  
    private CoffeeEnumerator enumerator;  
  
    public CoffeeCollection() {  
        enumerator = new CoffeeEnumerator();  
    }  
  
    public IEnumerator GetEnumerator() {  
        return enumerator;  
    }  
  
    public class CoffeeEnumerator : IEnumerator {  
        string[] beverages = new string[3] { "espresso", "macchiato", "latte" };  
        int currentIndex = -1;  
  
        public object Current {  
            get {  
                return beverages[currentIndex];  
            }  
        }  
  
        public bool MoveNext() {  
            currentIndex++;  
  
            if (currentIndex < beverages.Length) {  
                return true;  
            }  
  
            return false;  
        }  
  
        public void Reset() {  
            currentIndex = 0;  
        }  
    }  
}
```

```
}
```

第30.2节：IEnumerable<int>

在最基本的形式中，实现 `IEnumerable` 的对象表示一系列对象。相关对象可以使用 C# 的 `foreach` 关键字进行迭代。

在下面的示例中，对象 `sequenceOfNumbers` 实现了 `IEnumerable`。它表示一系列整数。`foreach` 循环依次遍历每个元素。

```
int AddNumbers(IEnumerable<int> sequenceOfNumbers) {
    int returnValue = 0;
    foreach(int i in sequenceOfNumbers) {
        returnValue += i;
    }
    return returnValue;
}
```

```
}
```

Section 30.2: IEnumerable<int>

In its most basic form, an object that implements `IEnumerable` represents a series of objects. The objects in question can be iterated using the c# `foreach` keyword.

In the example below, the object `sequenceOfNumbers` implements `IEnumerable`. It represents a series of integers. The `foreach` loop iterates through each in turn.

```
int AddNumbers(IEnumerable<int> sequenceOfNumbers) {
    int returnValue = 0;
    foreach(int i in sequenceOfNumbers) {
        returnValue += i;
    }
    return returnValue;
}
```

第31章：值类型与引用类型

第31.1节：使用 ref 关键字按引用传递

来自文档：[_____](#)

在 C# 中，参数可以通过值传递或引用传递。通过引用传递使函数成员、方法、属性、索引器、运算符和构造函数能够改变参数的值，并使该更改在调用环境中持续存在。要通过引用传递参数，使用ref或out关键字。

ref和out的区别在于，out表示传入的参数必须在函数结束前被赋值。相比之下，通过ref传递的参数可以被修改也可以保持不变。

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("Inside Main - After Callee: a = {0}", a);

        Console.WriteLine("Inside Main - Before CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("Inside Main - After CalleeRef: a = {0}", a);

        Console.WriteLine("Inside Main - Before CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);

        Console.ReadLine();
    }

    static void Callee(int a)
    {
        a = 5;
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a = 6;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        a = 7;
        Console.WriteLine("Inside CalleeOut a : {0}", a);
    }
}
```

输出：

Chapter 31: Value type vs Reference type

Section 31.1: Passing by reference using ref keyword

From the [documentation](#) :

In C#, arguments can be passed to parameters either by value or by reference. Passing by reference enables function members, methods, properties, indexers, operators, and constructors to change the value of the parameters and have that change persist in the calling environment. To pass a parameter by reference, use the `ref` or `out` keyword.

The difference between `ref` and `out` is that `out` means that the passed parameter has to be assigned before the function ends.in contrast parameters passed with `ref` can be changed or left unchanged.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("Inside Main - After Callee: a = {0}", a);

        Console.WriteLine("Inside Main - Before CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("Inside Main - After CalleeRef: a = {0}", a);

        Console.WriteLine("Inside Main - Before CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);

        Console.ReadLine();
    }

    static void Callee(int a)
    {
        a = 5;
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a = 6;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        a = 7;
        Console.WriteLine("Inside CalleeOut a : {0}", a);
    }
}
```

Output :

```

主函数内部 - 调用 Callee 之前: a = 20
Callee 内部 a : 5
主函数内部 - 调用 Callee 之后: a = 20
主函数内部 - 调用 CalleeRef 之前: a = 20
CalleeRef 内部 a : 6
主函数内部-调用者引用后: a = 6
主函数内部-调用者输出前: a = 6
调用者输出内部 a : 7
内部主程序 - 调用者返回后: a = 7

```

第31.2节：在其他地方更改值

```

public static void Main(string[] args)
{
    var studentList = new List<Student>();
    studentList.Add(new Student("Scott", "Nuke"));
    studentList.Add(new Student("Vincent", "King"));
    studentList.Add(new Student("Craig", "Bertt"));

    // 创建一个单独的列表以便稍后打印
    var printingList = studentList; // 这是一个新的列表对象，但内部持有相同的学生对象

    // 哎呀，我们发现名字有拼写错误，所以修正它们
    studentList[0].LastName = "Duke";
    studentList[1].LastName = "Kong";
    studentList[2].LastName = "Brett";

    // 好的，现在我们打印列表
    PrintPrintingList(printingList);
}

private static void PrintPrintingList(List<Student> students)
{
    foreach (Student student in students)
    {
        Console.WriteLine(string.Format("{0} {1}", student.FirstName, student.LastName));
    }
}

```

你会注意到，尽管 printingList 列表是在修正学生姓名拼写错误之前创建的，PrintPrintingList 方法仍然打印出了更正后的姓名：

```

斯科特·杜克
文森特·孔
克雷格·布雷特

```

这是因为两个列表都持有对同一学生对象的引用。因此，修改底层的学生对象会影响两个列表中的使用。

下面是学生类的示例代码。

```

public class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Student(string firstName, string lastName)
    {

```

```

Inside Main - Before Callee: a = 20
Inside Callee a : 5
Inside Main - After Callee: a = 20
Inside Main - Before CalleeRef: a = 20
Inside CalleeRef a : 6
Inside Main - After CalleeRef: a = 6
Inside Main - Before CalleeOut: a = 6
Inside CalleeOut a : 7
Inside Main - After CalleeOut: a = 7

```

Section 31.2: Changing values elsewhere

```

public static void Main(string[] args)
{
    var studentList = new List<Student>();
    studentList.Add(new Student("Scott", "Nuke"));
    studentList.Add(new Student("Vincent", "King"));
    studentList.Add(new Student("Craig", "Bertt"));

    // make a separate list to print out later
    var printingList = studentList; // this is a new list object, but holding the same student
    objects inside it

    // oops, we've noticed typos in the names, so we fix those
    studentList[0].LastName = "Duke";
    studentList[1].LastName = "Kong";
    studentList[2].LastName = "Brett";

    // okay, we now print the list
    PrintPrintingList(printingList);
}

private static void PrintPrintingList(List<Student> students)
{
    foreach (Student student in students)
    {
        Console.WriteLine(string.Format("{0} {1}", student.FirstName, student.LastName));
    }
}

```

You'll notice that even though the printingList list was made before the corrections to student names after the typos, the PrintPrintingList method still prints out the corrected names:

```

Scott Duke
Vincent Kong
Craig Brett

```

This is because both lists hold a list of references to the same students. SO changing the underlying student object propagates to usages by either list.

Here's what the student class would look like.

```

public class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Student(string firstName, string lastName)
    {

```

```

    this.FirstName = firstName;
    this.LastName = lastName;
}
}

```

第31.3节：ref 与 out 参数

代码

```

class Program
{
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("主函数内 - 调用后: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("主函数内 - 调用前 CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("主函数内 - 调用后 CalleeRef: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("主函数内 - 调用前 CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);
        Console.ReadLine();
    }

    static void Callee(int a)
    {
        a += 5;
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a += 10;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        // 不能使用 a+=15, 因为对于此方法, 'a' 仅在方法声明中声明, 未初始化
        a = 25; // 必须初始化
        Console.WriteLine("CalleeOut 内的 a : {0}", a);
    }
}

```

输出

```

主函数内 - 调用前 Callee: a = 20
Callee 内的 a : 25
主函数内 - 调用后 Callee: a = 20

主函数内 - 调用前 CalleeRef: a = 20
CalleeRef 内的 a : 30
主函数内部-调用后引用: a = 30

```

```

    this.FirstName = firstName;
    this.LastName = lastName;
}
}

```

Section 31.3: ref vs out parameters

Code

```

class Program
{
    static void Main(string[] args)
    {
        int a = 20;
        Console.WriteLine("Inside Main - Before Callee: a = {0}", a);
        Callee(a);
        Console.WriteLine("Inside Main - After Callee: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("Inside Main - Before CalleeRef: a = {0}", a);
        CalleeRef(ref a);
        Console.WriteLine("Inside Main - After CalleeRef: a = {0}", a);
        Console.WriteLine();

        Console.WriteLine("Inside Main - Before CalleeOut: a = {0}", a);
        CalleeOut(out a);
        Console.WriteLine("Inside Main - After CalleeOut: a = {0}", a);
        Console.ReadLine();
    }

    static void Callee(int a)
    {
        a += 5;
        Console.WriteLine("Inside Callee a : {0}", a);
    }

    static void CalleeRef(ref int a)
    {
        a += 10;
        Console.WriteLine("Inside CalleeRef a : {0}", a);
    }

    static void CalleeOut(out int a)
    {
        // can't use a+=15 since for this method 'a' is not initialized only declared in the method
        // declaration
        a = 25; // has to be initialized
        Console.WriteLine("Inside CalleeOut a : {0}", a);
    }
}

```

Output

```

Inside Main - Before Callee: a = 20
Inside Callee a : 25
Inside Main - After Callee: a = 20

Inside Main - Before CalleeRef: a = 20
Inside CalleeRef a : 30
Inside Main - After CalleeRef: a = 30

```

```
主函数内部-调用前输出: a = 30  
调用输出内部 a : 25  
主函数内部-调用后输出: a = 25
```

第31.4节：赋值

```
var a = new List<int>();  
var b = a;  
a.Add(5);  
Console.WriteLine(a.Count); // 输出 1  
Console.WriteLine(b.Count); // 也输出 1
```

将值赋给一个 `List<int>` 类型的变量不会创建该 `List<int>` 的副本。相反，它会复制对该 `List<int>` 的引用。我们称这种行为的类型为引用类型。

第31.5节：与方法参数 `ref` 和 `out` 的区别

有两种方式可以通过引用传递值类型：`ref` 和 `out`。区别在于使用 `ref` 传递时，值必须被初始化，而使用 `out` 传递时则不需要。使用 `out` 可以确保变量在方法调用后有一个值：

```
public void ByRef(ref int value)  
{  
    Console.WriteLine(nameof(ByRef) + value);  
    value += 4;  
    Console.WriteLine(nameof(ByRef) + value);  
}  
  
public void ByOut(out int value)  
{  
    value += 4 // CS0269: 使用未赋值的 out 参数 'value'  
    Console.WriteLine(nameof(ByOut) + value); // CS0269: 使用未赋值的 out 参数 'value'  
  
    value = 4;  
    Console.WriteLine(nameof(ByOut) + value);  
}  
  
public void TestOut()  
{  
    int outValue1;  
    ByOut(out outValue1); // 输出 4  
  
    int outValue2 = 10; // 对 out 参数没有意义  
    ByOut(out outValue2); // 输出 4  
}  
  
public void TestRef()  
{  
    int refValue1;  
    ByRef(ref refValue1); // S0165 使用未赋值的局部变量 'refValue'  
  
    int refValue2 = 0;  
    ByRef(ref refValue2); // 输出 0 和 4  
  
    int refValue3 = 10;  
    ByRef(ref refValue3); // 输出 10 和 14  
}
```

关键在于使用 `out` 时，参数 must 在方法返回前必须被初始化，因此以下

```
Inside Main - Before CalleeOut: a = 30  
Inside CalleeOut a : 25  
Inside Main - After CalleeOut: a = 25
```

Section 31.4: Assignment

```
var a = new List<int>();  
var b = a;  
a.Add(5);  
Console.WriteLine(a.Count); // prints 1  
Console.WriteLine(b.Count); // prints 1 as well
```

Assigning to a variable of a `List<int>` does not create a copy of the `List<int>`. Instead, it copies the reference to the `List<int>`. We call types that behave this way *reference types*.

Section 31.5: Difference with method parameters `ref` and `out`

There are two possible ways to pass a value type by reference: `ref` and `out`. The difference is that by passing it with `ref` the value must be initialized but not when passing it with `out`. Using `out` ensures that the variable has a value after the method call:

```
public void ByRef(ref int value)  
{  
    Console.WriteLine(nameof(ByRef) + value);  
    value += 4;  
    Console.WriteLine(nameof(ByRef) + value);  
}  
  
public void ByOut(out int value)  
{  
    value += 4 // CS0269: Use of unassigned out parameter 'value'  
    Console.WriteLine(nameof(ByOut) + value); // CS0269: Use of unassigned out parameter 'value'  
  
    value = 4;  
    Console.WriteLine(nameof(ByOut) + value);  
}  
  
public void TestOut()  
{  
    int outValue1;  
    ByOut(out outValue1); // prints 4  
  
    int outValue2 = 10; // does not make any sense for out  
    ByOut(out outValue2); // prints 4  
}  
  
public void TestRef()  
{  
    int refValue1;  
    ByRef(ref refValue1); // S0165 Use of unassigned local variable 'refValue'  
  
    int refValue2 = 0;  
    ByRef(ref refValue2); // prints 0 and 4  
  
    int refValue3 = 10;  
    ByRef(ref refValue3); // prints 10 and 14  
}
```

The catch is that by using `out` the parameter must be initialized before leaving the method, therefore the following

方法使用 `ref` 是可行的，但使用 `out` 则不可：

```
public void EmtyRef(bool condition, ref int value)
{
    if (condition)
    {
        value += 10;
    }
}

public void EmtyOut(bool condition, out int value)
{
    if (condition)
    {
        value = 10;
    }
} //CS0177: out 参数 'value' 必须在当前方法控制流离开之前被赋值
```

这是因为如果 `condition` 不成立，`value` 就不会被赋值。

第31.6节：按引用传递

如果你想让值类型与引用类型在方法中的示例正常工作，请在方法签名中对你想按引用传递的参数使用 `ref` 关键字，并且在调用方法时也要使用该关键字。

```
public static void Main(string[] args)
{
    ...
    DoubleNumber(ref number); // 调用代码
    Console.WriteLine(number); // 输出 8
    ...
}

public void DoubleNumber(ref int number)
{
    number += number;
}
```

进行这些更改后，数字将按预期更新，意味着控制台输出的数字将是8。

method is possible with `ref` but not with `out`:

```
public void EmtyRef(bool condition, ref int value)
{
    if (condition)
    {
        value += 10;
    }
}

public void EmtyOut(bool condition, out int value)
{
    if (condition)
    {
        value = 10;
    }
} //CS0177: The out parameter 'value' must be assigned before control leaves the current method
```

This is because if `condition` does not hold, `value` goes unassigned.

Section 31.6: Passing by reference

If you want the Value Types vs Reference Types in methods example to work properly, use the `ref` keyword in your method signature for the parameter you want to pass by reference, as well as when you call the method.

```
public static void Main(string[] args)
{
    ...
    DoubleNumber(ref number); // calling code
    Console.WriteLine(number); // outputs 8
    ...
}

public void DoubleNumber(ref int number)
{
    number += number;
}
```

Making these changes would make the number update as expected, meaning the console output for `number` would be 8.

第32章：内置类型

第32.1节：装箱值类型的转换

装箱值类型只能拆箱回其原始类型，即使两种类型之间的转换是有效的，例如：

```
object boxedInt = (int)1; // 将int装箱到object中  
long unboxedInt1 = (long)boxedInt; // 无效的类型转换
```

可以通过先拆箱为原始类型来避免这种情况，例如：

```
long unboxedInt2 = (long)(int)boxedInt; // 有效
```

第32.2节：与装箱值类型的比较

如果值类型被赋值给类型为`object`的变量，则会被装箱——值存储在一个`System.Object`的实例中。这在使用`==`比较值时可能导致意想不到的结果，例如：

```
object left = (int)1; // 对象盒中的 int  
object right = (int)1; // 对象盒中的 int  
  
var comparison1 = left == right; // false
```

这可以通过使用重载的`Equals`方法来避免，该方法将给出预期的结果。

```
var comparison2 = left.Equals(right); // true
```

或者，也可以通过拆箱`left`和`right`变量，使得`int`值被

比较来实现相同的效果：

```
var comparison3 = (int)left == (int)right; // true
```

第32.3节：不可变引用类型 - 字符串

```
// 从字符串字面量赋值字符串  
string s = "hello";  
  
// 从字符数组赋值字符串  
char[] chars = new char[] { 'h', 'e', 'l', 'l', 'o' };  
string s = new string(chars, 0, chars.Length);  
  
// 从字符指针赋值字符串，该指针来源于字符串  
string s;  
unsafe  
{  
    fixed (char* charPointer = "hello")  
    {  
        s = new string(charPointer);  
    }  
}
```

Chapter 32: Built-in Types

Section 32.1: Conversion of boxed value types

`Boxed` value types can only be unboxed into their original Type, even if a conversion of the two Types is valid, e.g.:

```
object boxedInt = (int)1; // int boxed in an object  
long unboxedInt1 = (long)boxedInt; // invalid cast
```

This can be avoided by first unboxing into the original Type, e.g.:

```
long unboxedInt2 = (long)(int)boxedInt; // valid
```

Section 32.2: Comparisons with boxed value types

If value types are assigned to variables of type `object` they are `boxed` - the value is stored in an instance of a `System.Object`. This can lead to unintended consequences when comparing values with `==`, e.g.:

```
object left = (int)1; // int in an object box  
object right = (int)1; // int in an object box  
  
var comparison1 = left == right; // false
```

This can be avoided by using the overloaded `Equals` method, which will give the expected result.

```
var comparison2 = left.Equals(right); // true
```

Alternatively, the same could be done by unboxing the `left` and `right` variables so that the `int` values are compared:

```
var comparison3 = (int)left == (int)right; // true
```

Section 32.3: Immutable reference type - string

```
// assign string from a string literal  
string s = "hello";  
  
// assign string from an array of characters  
char[] chars = new char[] { 'h', 'e', 'l', 'l', 'o' };  
string s = new string(chars, 0, chars.Length);  
  
// assign string from a char pointer, derived from a string  
string s;  
unsafe  
{  
    fixed (char* charPointer = "hello")  
    {  
        s = new string(charPointer);  
    }  
}
```

第32.4节：值类型 - char

```
// 单个字符 s  
char c = 's';  
  
// 字符 s：从整数值强制转换  
char c = (char)115;  
  
// Unicode字符：单个字符 s  
char c = '\u0073';  
  
// Unicode字符：笑脸  
char c = '\u263a';
```

第32.5节：值类型 - short、int、long（有符号16位、32位、64位整数）

```
// 将有符号短整型赋值为其最小值  
short s = -32768;  
  
// 将有符号短整型赋值为其最大值  
short s = 32767;  
  
// 将有符号整型赋值为其最小值  
int i = -2147483648;  
  
// 将有符号整型赋值为其最大值  
int i = 2147483647;  
  
// 将有符号长整型赋值为其最小值 (注意long后缀)  
long l = -9223372036854775808L;  
  
// 将有符号长整型赋值为其最大值 (注意long后缀)  
long l = 9223372036854775807L;
```

也可以使这些类型可空，意味着除了通常的值外，还可以赋值为null。如果可空类型的变量未初始化，它将是null而不是0。可空类型通过在类型后添加问号 (?) 来标记。

```
int a; // 现在这是0。  
int? b; // 现在这是null。
```

第32.6节：值类型 - ushort、uint、ulong（无符号16位、32位、64位整数）

```
// 将无符号短整型赋值为其最小值  
ushort s = 0;  
  
// 将无符号短整型赋值为其最大值  
ushort s = 65535;  
  
// 将无符号整数赋值为其最小值  
uint i = 0;  
  
// 将无符号整数赋值为其最大值  
uint i = 4294967295;
```

Section 32.4: Value type - char

```
// single character s  
char c = 's';  
  
// character s: casted from integer value  
char c = (char)115;  
  
// unicode character: single character s  
char c = '\u0073';  
  
// unicode character: smiley face  
char c = '\u263a';
```

Section 32.5: Value type - short, int, long (signed 16 bit, 32 bit, 64 bit integers)

```
// assigning a signed short to its minimum value  
short s = -32768;  
  
// assigning a signed short to its maximum value  
short s = 32767;  
  
// assigning a signed int to its minimum value  
int i = -2147483648;  
  
// assigning a signed int to its maximum value  
int i = 2147483647;  
  
// assigning a signed long to its minimum value (note the long postfix)  
long l = -9223372036854775808L;  
  
// assigning a signed long to its maximum value (note the long postfix)  
long l = 9223372036854775807L;
```

It is also possible to make these types nullable, meaning that additionally to the usual values, null can be assigned, too. If a variable of a nullable type is not initialized, it will be null instead of 0. Nullable types are marked by adding a question mark (?) after the type.

```
int a; //This is now 0.  
int? b; //This is now null.
```

Section 32.6: Value type - ushort, uint, ulong (unsigned 16 bit, 32 bit, 64 bit integers)

```
// assigning an unsigned short to its minimum value  
ushort s = 0;  
  
// assigning an unsigned short to its maximum value  
ushort s = 65535;  
  
// assigning an unsigned int to its minimum value  
uint i = 0;  
  
// assigning an unsigned int to its maximum value  
uint i = 4294967295;
```

```
// 将无符号长整数赋值为其最小值 (注意无符号长整数后缀)
ulong l = 0UL;

// 将无符号长整数赋值为其最大值 (注意无符号长整数后缀)
ulong l = 18446744073709551615UL;
```

也可以使这些类型可空，意味着除了通常的值外，还可以赋值为null。如果可空类型的变量未初始化，它将是null而不是0。可空类型通过在类型后添加问号 (?) 来标记。

```
uint a; // 现在这是0。
uint? b; // 现在这是null。
```

第32.7节：值类型 - bool

```
// 布尔类型的默认值是 false
bool b;
// 可空布尔类型的默认值是 null
bool? z;
b = true;
if(b) {
    Console.WriteLine("Boolean 有 true 值");
}
```

bool 关键字是 System.Boolean 的别名。它用于声明变量以存储布尔值，true 和 false。

```
// assigning an unsigned long to its minimum value (note the unsigned long postfix)
ulong l = 0UL;

// assigning an unsigned long to its maximum value (note the unsigned long postfix)
ulong l = 18446744073709551615UL;
```

It is also possible to make these types nullable, meaning that additionally to the usual values, null can be assigned, too. If a variable of a nullable type is not initialized, it will be null instead of 0. Nullable types are marked by adding a question mark (?) after the type.

```
uint a; //This is now 0.
uint? b; //This is now null.
```

Section 32.7: Value type - bool

```
// default value of boolean is false
bool b;
//default value of nullable boolean is null
bool? z;
b = true;
if(b) {
    Console.WriteLine("Boolean has true value");
}
```

The bool keyword is an alias of System.Boolean. It is used to declare variables to store the Boolean values, true and false.

第33章：内置类型的别名

第33.1节：内置类型表

下表显示了内置C#类型的关键字，这些关键字是System命名空间中预定义类型的别名。

C# 类型 .NET 框架类型

bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
object	System.Object
short	System.Int16
ushort	System.UInt16
string	System.String

C# 类型关键字及其别名是可以互换的。例如，您可以通过以下任一声明来声明一个整数变量：

```
int number = 123;  
System.Int32 number = 123;
```

Chapter 33: Aliases of built-in types

Section 33.1: Built-In Types Table

The following table shows the keywords for built-in C# types, which are aliases of predefined types in the System namespaces.

C# Type .NET Framework Type

bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
object	System.Object
short	System.Int16
ushort	System.UInt16
string	System.String

The C# type keywords and their aliases are interchangeable. For example, you can declare an integer variable by using either of the following declarations:

```
int number = 123;  
System.Int32 number = 123;
```

第34章：匿名类型

第34.1节：匿名与动态

匿名类型允许创建对象而无需事先显式定义其类型，同时保持静态类型检查。

```
var anon = new { Value = 1 };
Console.WriteLine(anon.Id); // 编译时错误
```

相反，dynamic 具有动态类型检查，选择运行时错误而非编译时错误。

```
dynamic val = "foo";
Console.WriteLine(val.Id); // 编译通过，但运行时抛出错误
```

第34.2节：创建匿名类型

由于匿名类型没有名称，因此这些类型的变量必须是隐式类型 (var)。

```
var anon = new { Foo = 1, Bar = 2 };
// anon.Foo == 1
// anon.Bar == 2
```

如果未指定成员名称，则成员名称将设置为用于初始化

```
int foo = 1;
int bar = 2;
var anon2 = new { foo, bar };
// anon2.foo == 1
// anon2.bar == 2
```

请注意，只有当匿名类型声明中的表达式是简单的属性访问时，名称才能省略；对于方法调用或更复杂的表达式，必须指定属性名称。

```
string foo = "some string";
var anon3 = new { foo.Length };
// anon3.Length == 11
var anon4 = new { foo.Length <= 10 ? "short string" : "long string" };
// 编译错误 - 无效的匿名类型成员声明符。
var anon5 = new { Description = foo.Length <= 10 ? "short string" : "long string" };
// 正确
```

第34.3节：匿名类型的相等性

匿名类型的相等性由 Equals 实例方法决定。如果两个对象具有相同的类型，并且每个属性的值都相等（通过 `a.Prop.Equals(b.Prop)`），则这两个对象相等。

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 1, Bar = 2 };
var anon3 = new { Foo = 5, Bar = 10 };
var anon3 = new { Foo = 5, Bar = 10 };
var anon4 = new { Bar = 2, Foo = 1 };
// anon.Equals(anon2) == true
// anon.Equals(anon3) == false
```

Chapter 34: Anonymous types

Section 34.1: Anonymous vs dynamic

Anonymous types allow the creation of objects without having to explicitly define their types ahead of time, while maintaining static type checking.

```
var anon = new { Value = 1 };
Console.WriteLine(anon.Id); // compile time error
```

Conversely, dynamic has dynamic type checking, opting for runtime errors, instead of compile-time errors.

```
dynamic val = "foo";
Console.WriteLine(val.Id); // compiles, but throws runtime error
```

Section 34.2: Creating an anonymous type

Since anonymous types are not named, variables of those types must be implicitly typed (var).

```
var anon = new { Foo = 1, Bar = 2 };
// anon.Foo == 1
// anon.Bar == 2
```

If the member names are not specified, they are set to the name of the property/variable used to initialize the object.

```
int foo = 1;
int bar = 2;
var anon2 = new { foo, bar };
// anon2.foo == 1
// anon2.bar == 2
```

Note that names can only be omitted when the expression in the anonymous type declaration is a simple property access; for method calls or more complex expressions, a property name must be specified.

```
string foo = "some string";
var anon3 = new { foo.Length };
// anon3.Length == 11
var anon4 = new { foo.Length <= 10 ? "short string" : "long string" };
// compiler error - Invalid anonymous type member declarator.
var anon5 = new { Description = foo.Length <= 10 ? "short string" : "long string" };
// OK
```

Section 34.3: Anonymous type equality

Anonymous type equality is given by the Equals instance method. Two objects are equal if they have the same type and equal values (through `a.Prop.Equals(b.Prop)`) for every property.

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 1, Bar = 2 };
var anon3 = new { Foo = 5, Bar = 10 };
var anon3 = new { Foo = 5, Bar = 10 };
var anon4 = new { Bar = 2, Foo = 1 };
// anon.Equals(anon2) == true
// anon.Equals(anon3) == false
```

```
// anon.Equals(anon4) == false (anon 和 anon4 类型不同, 见下文)
```

当且仅当两个匿名类型的属性名称和类型相同且顺序一致时，才认为它们是相同的。

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 7, Bar = 1 };
var anon3 = new { Bar = 1, Foo = 3 };
var anon4 = new { Fa = 1, Bar = 2 };
// anon 和 anon2 类型相同
// anon 和 anon3 类型不同 (Bar 和 Foo 的顺序不同)
// anon 和 anon4 类型不同 (属性名称不同)
```

```
// anon.Equals(anon4) == false (anon and anon4 have different types, see below)
```

Two anonymous types are considered the same if and only if their properties have the same name and type and appear in the same order.

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 7, Bar = 1 };
var anon3 = new { Bar = 1, Foo = 3 };
var anon4 = new { Fa = 1, Bar = 2 };
// anon and anon2 have the same type
// anon and anon3 have different types (Bar and Foo appear in different orders)
// anon and anon4 have different types (property names are different)
```

第34.4节：带有匿名类型的泛型方法

泛型方法允许通过类型推断使用匿名类型。

```
void Log<T>(T obj) {
    // ...
}
Log(new { Value = 10 });
```

这意味着可以在 LINQ 表达式中使用匿名类型：

```
var products = new[] {
    new { Amount = 10, Id = 0 },
    new { Amount = 20, Id = 1 },
    new { Amount = 15, Id = 2 }
};
var idsByAmount = products.OrderBy(x => x.Amount).Select(x => x.Id);
// idsByAmount: 0, 2, 1
```

Section 34.4: Generic methods with anonymous types

Generic methods allow the use of anonymous types through type inference.

```
void Log<T>(T obj) {
    // ...
}
Log(new { Value = 10 });
```

This means LINQ expressions can be used with anonymous types:

```
var products = new[] {
    new { Amount = 10, Id = 0 },
    new { Amount = 20, Id = 1 },
    new { Amount = 15, Id = 2 }
};
var idsByAmount = products.OrderBy(x => x.Amount).Select(x => x.Id);
// idsByAmount: 0, 2, 1
```

第34.5节：使用匿名类型实例化泛型类型

使用泛型构造函数需要为匿名类型命名，这是不可能的。或者，可以使用泛型方法以允许类型推断发生。

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 5, Bar = 10 };
List<T> CreateList<T>(params T[] items) {
    return new List<T>(items);
}

var list1 = CreateList(anon, anon2);
```

对于List<T>，隐式类型数组可以通过ToList LINQ方法转换为List<T>：

```
var list2 = new[] {anon, anon2}.ToList();
```

Section 34.5: Instantiating generic types with anonymous types

Using generic constructors would require the anonymous types to be named, which is not possible. Alternatively, generic methods may be used to allow type inference to occur.

```
var anon = new { Foo = 1, Bar = 2 };
var anon2 = new { Foo = 5, Bar = 10 };
List<T> CreateList<T>(params T[] items) {
    return new List<T>(items);
}

var list1 = CreateList(anon, anon2);
```

In the case of List<T>, implicitly typed arrays may be converted to a List<T> through the ToList LINQ method:

```
var list2 = new[] {anon, anon2}.ToList();
```

第34.6节：隐式类型数组

可以使用隐式类型创建匿名类型数组。

```
var arr = new[] {
    new { Id = 0 },
```

Section 34.6: Implicitly typed arrays

Arrays of anonymous types may be created with implicit typing.

```
var arr = new[] {
    new { Id = 0 },
```

```
new { Id = 1 }  
};
```

```
new { Id = 1 }  
};
```

第35章：动态类型

第35.1节：创建带属性的动态对象

```
using System;
using System.Dynamic;

dynamic info = new ExpandoObject();
info.Id = 123;
info.Another = 456;

Console.WriteLine(info.Another);
// 456

Console.WriteLine(info.DoesntExist);
// 抛出 RuntimeBinderException
```

第35.2节：创建动态变量

```
dynamic foo = 123;
Console.WriteLine(foo + 234);
// 357 Console.WriteLine(foo.ToUpper())
// RuntimeBinderException, 因为 int 类型没有 ToUpper 方法

foo = "123";
Console.WriteLine(foo + 234);
// 123234
Console.WriteLine(foo.ToUpper());
// 现在是字符串
```

第35.3节：返回动态类型

```
using System;

public static void Main()
{
    var value = GetValue();
    Console.WriteLine(value);
    // 动态类型很有用！
}

private static dynamic GetValue()
{
    return "动态类型很有用！";
}
```

第35.4节：处理编译时未知的特定类型

以下输出结果等效：

```
class IfElseExample
{
    public string DebugToString(object a)
    {
        if (a is StringBuilder)
```

Chapter 35: Dynamic type

Section 35.1: Creating a dynamic object with properties

```
using System;
using System.Dynamic;

dynamic info = new ExpandoObject();
info.Id = 123;
info.Another = 456;

Console.WriteLine(info.Another);
// 456

Console.WriteLine(info.DoesntExist);
// Throws RuntimeBinderException
```

Section 35.2: Creating a dynamic variable

```
dynamic foo = 123;
Console.WriteLine(foo + 234);
// 357 Console.WriteLine(foo.ToUpper())
// RuntimeBinderException, since int doesn't have a ToUpper method

foo = "123";
Console.WriteLine(foo + 234);
// 123234
Console.WriteLine(foo.ToUpper());
// NOW A STRING
```

Section 35.3: Returning dynamic

```
using System;

public static void Main()
{
    var value = GetValue();
    Console.WriteLine(value);
    // dynamics are useful!
}

private static dynamic GetValue()
{
    return "dynamics are useful!";
}
```

Section 35.4: Handling Specific Types Unknown at Compile Time

The following output equivalent results:

```
class IfElseExample
{
    public string DebugToString(object a)
    {
        if (a is StringBuilder)
```

```

    {
        return DebugToStringInternal(a as StringBuilder);
    }
    else if (a is List<string>)
    {
        return DebugToStringInternal(a as List<string>);
    }
    else
    {
        return a.ToString();
    }
}

private string DebugToStringInternal(object a)
{
    // 备用方案
    return a.ToString();
}

private string DebugToStringInternal(StringBuilder sb)
{
    return $"StringBuilder - 容量: {sb.Capacity}, 最大容量: {sb.MaxCapacity}, 值: {sb.ToString()}";
}

private string DebugToStringInternal(List<string> list)
{
    return $"List<string> - Count: {list.Count}, Value: {Environment.NewLine + "" + string.Join(Environment.NewLine + "", list.ToArray())}";
}
}

class DynamicExample
{
    public string DebugToString(object a)
    {
        return DebugToStringInternal((dynamic)a);
    }

    private string DebugToStringInternal(object a)
    {
        // 备用方案
        return a.ToString();
    }

    private string DebugToStringInternal(StringBuilder sb)
    {
        return $"StringBuilder - 容量: {sb.Capacity}, 最大容量: {sb.MaxCapacity}, 值: {sb.ToString()}";
    }

    private string DebugToStringInternal(List<string> list)
    {
        return $"List<string> - Count: {list.Count}, Value: {Environment.NewLine + "" + string.Join(Environment.NewLine + "", list.ToArray())}";
    }
}

```

使用 dynamic 的优点是，添加一个新类型只需为新类型添加一个 DebugToStringInternal 的重载方法。同时也消除了手动将其强制转换为该类型的需求。

```

    {
        return DebugToStringInternal(a as StringBuilder);
    }
    else if (a is List<string>)
    {
        return DebugToStringInternal(a as List<string>);
    }
    else
    {
        return a.ToString();
    }
}

private string DebugToStringInternal(object a)
{
    // Fall Back
    return a.ToString();
}

private string DebugToStringInternal(StringBuilder sb)
{
    return $"StringBuilder - Capacity: {sb.Capacity}, MaxCapacity: {sb.MaxCapacity}, Value: {sb.ToString()}";
}

private string DebugToStringInternal(List<string> list)
{
    return $"List<string> - Count: {list.Count}, Value: {Environment.NewLine + "\t" + string.Join(Environment.NewLine + "\t", list.ToArray())}";
}

class DynamicExample
{
    public string DebugToString(object a)
    {
        return DebugToStringInternal((dynamic)a);
    }

    private string DebugToStringInternal(object a)
    {
        // Fall Back
        return a.ToString();
    }

    private string DebugToStringInternal(StringBuilder sb)
    {
        return $"StringBuilder - Capacity: {sb.Capacity}, MaxCapacity: {sb.MaxCapacity}, Value: {sb.ToString()}";
    }

    private string DebugToStringInternal(List<string> list)
    {
        return $"List<string> - Count: {list.Count}, Value: {Environment.NewLine + "\t" + string.Join(Environment.NewLine + "\t", list.ToArray())}";
    }
}

```

The advantage to the dynamic, is adding a new Type to handle just requires adding an overload of DebugToStringInternal of the new type. Also eliminates the need to manually cast it to the type as well.

第36章：类型转换

第36.1节：显式类型转换

```
using System;
namespace TypeConversionApplication
{
    class ExplicitConversion
    {
        static void Main(string[] args)
        {
            double d = 5673.74;
            int i;

            // 将 double 转换为 int.
            i = (int)d;
            Console.WriteLine(i);
            Console.ReadKey();
        }
    }
}
```

第36.2节：MSDN隐式运算符示例

```
类 Digit
{
    public Digit(double d) { val = d; }
    public double val;

    // 从Digit到double的用户自定义转换
    public static implicit operator double(Digit d)
    {
        Console.WriteLine("调用了Digit到double的隐式转换");
        return d.val;
    }

    // 从double到Digit的用户自定义转换
    public static implicit operator Digit(double d)
    {
        Console.WriteLine("调用了double到Digit的隐式转换");
        return new Digit(d);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Digit dig = new Digit(7);
        // 此调用触发隐式"double"操作符
        double num = dig;
        // 此调用触发隐式"Digit"操作符
        Digit dig2 = 12;
        Console.WriteLine("num = {0} dig2 = {1}", num, dig2.val);
        Console.ReadLine();
    }
}
```

输出：

Chapter 36: Type Conversion

Section 36.1: Explicit Type Conversion

```
using System;
namespace TypeConversionApplication
{
    class ExplicitConversion
    {
        static void Main(string[] args)
        {
            double d = 5673.74;
            int i;

            // cast double to int.
            i = (int)d;
            Console.WriteLine(i);
            Console.ReadKey();
        }
    }
}
```

Section 36.2: MSDN implicit operator example

```
class Digit
{
    public Digit(double d) { val = d; }
    public double val;

    // User-defined conversion from Digit to double
    public static implicit operator double(Digit d)
    {
        Console.WriteLine("Digit to double implicit conversion called");
        return d.val;
    }

    // User-defined conversion from double to Digit
    public static implicit operator Digit(double d)
    {
        Console.WriteLine("double to Digit implicit conversion called");
        return new Digit(d);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Digit dig = new Digit(7);
        // This call invokes the implicit "double" operator
        double num = dig;
        // This call invokes the implicit "Digit" operator
        Digit dig2 = 12;
        Console.WriteLine("num = {0} dig2 = {1}", num, dig2.val);
        Console.ReadLine();
    }
}
```

Output:

调用了Digit到double的隐式转换
调用了double到Digit的隐式转换
num = 7 dig2 = 12

[.NET Fiddle 在线演示](#)

Digit to double implicit conversion called
double to Digit implicit conversion called
num = 7 dig2 = 12

[Live Demo on .NET Fiddle](#)

第37章：类型转换

第37.1节：检查兼容性而不进行转换

如果你需要知道一个值的类型是否继承或实现了某个给定类型，但又不想实际将其强制转换为该类型，可以使用is操作符。

```
if(value is int)
{
    Console.WriteLine(value + "是一个 int");
}
```

第 37.2 节：将对象转换为基类型

给出以下定义：

```
public interface IMyInterface1
{
    string GetName();
}

public interface IMyInterface2
{
    string GetName();
}

public class MyClass : IMyInterface1, IMyInterface2
{
    string IMyInterface1.GetName()
    {
        return "IMyInterface1";
    }

    string IMyInterface2.GetName()
    {
        return "IMyInterface2";
    }
}
```

将对象转换为基类型的示例：

```
MyClass obj = new MyClass();

IMyInterface1 myClass1 = (IMyInterface1)obj;
IMyInterface2 myClass2 = (IMyInterface2)obj;

Console.WriteLine("我是 : {0}", myClass1.GetName());
Console.WriteLine("我是 : {0}", myClass2.GetName());

// 输出：
// 我是 : IMyInterface1
// 我是 : IMyInterface2
```

第37.3节：转换运算符

在C#中，类型可以定义自定义的转换运算符，这允许使用显式或隐式转换将值转换为其他类型或从其他类型转换。例如，考虑一个旨在表示JavaScript的类

Chapter 37: Casting

Section 37.1: Checking compatibility without casting

If you need to know whether a value's type extends or implements a given type, but you don't want to actually cast it as that type, you can use the is operator.

```
if(value is int)
{
    Console.WriteLine(value + " is an int");
}
```

Section 37.2: Cast an object to a base type

Given the following definitions :

```
public interface IMyInterface1
{
    string GetName();
}

public interface IMyInterface2
{
    string GetName();
}

public class MyClass : IMyInterface1, IMyInterface2
{
    string IMyInterface1.GetName()
    {
        return "IMyInterface1";
    }

    string IMyInterface2.GetName()
    {
        return "IMyInterface2";
    }
}
```

Casting an object to a base type example :

```
MyClass obj = new MyClass();

IMyInterface1 myClass1 = (IMyInterface1)obj;
IMyInterface2 myClass2 = (IMyInterface2)obj;

Console.WriteLine("I am : {0}", myClass1.GetName());
Console.WriteLine("I am : {0}", myClass2.GetName());

// Outputs :
// I am : IMyInterface1
// I am : IMyInterface2
```

Section 37.3: Conversion Operators

In C#, types can define custom *Conversion Operators*, which allow values to be converted to and from other types using either explicit or implicit casts. For example, consider a class that is meant to represent a JavaScript

表达式：

```
public class JsExpression
{
    private readonly string expression;
    public JsExpression(string rawExpression)
    {
        this.expression = rawExpression;
    }
    public override string ToString()
    {
        return this.expression;
    }
    public JsExpression IsEqualTo(JsExpression other)
    {
        return new JsExpression("(" + this + " == " + other + ")");
    }
}
```

如果我们想创建一个表示两个JavaScript值比较的JsExpression，我们可以这样做：

```
JsExpression intExpression = new JsExpression("-1");
JsExpression doubleExpression = new JsExpression("-1.0");
Console.WriteLine(intExpression.IsEqualTo(doubleExpression)); // (-1 == -1.0)
```

但是我们可以为JsExpression添加一些显式转换操作符，以便在使用显式强制转换时实现简单转换。

```
public static explicit operator JsExpression(int value)
{
    return new JsExpression(value.ToString());
}
public static explicit operator JsExpression(double value)
{
    return new JsExpression(value.ToString());
}

// Usage:
JsExpression intExpression = (JsExpression)(-1);
JsExpression doubleExpression = (JsExpression)(-1.0);
Console.WriteLine(intExpression.IsEqualTo(doubleExpression)); // (-1 == -1.0)
```

或者，我们可以将这些运算符改为implicit，使语法更简单。

```
public static implicit operator JsExpression(int value)
{
    return new JsExpression(value.ToString());
}
public static implicit operator JsExpression(double value)
{
    return new JsExpression(value.ToString());
}

// Usage:
JsExpression intExpression = -1;
Console.WriteLine(intExpression.IsEqualTo(-1.0)); // (-1 == -1.0)
```

expression:

```
public class JsExpression
{
    private readonly string expression;
    public JsExpression(string rawExpression)
    {
        this.expression = rawExpression;
    }
    public override string ToString()
    {
        return this.expression;
    }
    public JsExpression IsEqualTo(JsExpression other)
    {
        return new JsExpression("(" + this + " == " + other + ")");
    }
}
```

If we wanted to create a JsExpression representing a comparison of two JavaScript values, we could do something like this:

```
JsExpression intExpression = new JsExpression("-1");
JsExpression doubleExpression = new JsExpression("-1.0");
Console.WriteLine(intExpression.IsEqualTo(doubleExpression)); // (-1 == -1.0)
```

But we can add some *explicit conversion operators* to JsExpression, to allow a simple conversion when using explicit casting.

```
public static explicit operator JsExpression(int value)
{
    return new JsExpression(value.ToString());
}
public static explicit operator JsExpression(double value)
{
    return new JsExpression(value.ToString());
}

// Usage:
JsExpression intExpression = (JsExpression)(-1);
JsExpression doubleExpression = (JsExpression)(-1.0);
Console.WriteLine(intExpression.IsEqualTo(doubleExpression)); // (-1 == -1.0)
```

Or, we could change these operators to *implicit* to make the syntax much simpler.

```
public static implicit operator JsExpression(int value)
{
    return new JsExpression(value.ToString());
}
public static implicit operator JsExpression(double value)
{
    return new JsExpression(value.ToString());
}

// Usage:
JsExpression intExpression = -1;
Console.WriteLine(intExpression.IsEqualTo(-1.0)); // (-1 == -1.0)
```

第37.4节：LINQ类型转换操作

假设你有如下类型：

```
interface IThing { }
class Thing : IThing { }
```

LINQ 允许你通过 `Enumerable.Cast<>()` 和 `Enumerable.OfType<>()` 扩展方法创建一个投影，从而改变 `IEnumerable<>` 的编译时泛型类型。

```
IEnumerable<IThing> things = new IThing[] {new Thing()};
IEnumerable<Thing> things2 = things.Cast<Thing>();
IEnumerable<Thing> things3 = things.OfType<Thing>();
```

当对 `things2` 进行求值时，`Cast<>()` 方法会尝试将 `things` 中的所有值转换为 `Thing` 类型。如果遇到无法转换的值，将抛出 `InvalidCastException` 异常。

当对 `things3` 进行求值时，`OfType<>()` 方法也会执行相同操作，但如果遇到无法转换的值，它会简单地忽略该值，而不是抛出异常。

由于这些方法的泛型类型限制，它们无法调用转换运算符或执行数值转换。

```
double[] doubles = new[]{1,2,3}.Cast<double>().ToArray(); // 抛出 InvalidCastException
```

你可以简单地在 `.Select()` 中执行强制转换作为解决方法：

```
double[] doubles = new[]{1,2,3}.Select(i => (double)i).ToArray();
```

第37.5节：显式转换

如果你知道某个值是特定类型，可以显式地将其转换为该类型，以便在需要该类型的上下文中使用它。

```
object value = -1;
int number = (int) value;
Console.WriteLine(Math.Abs(number));
```

如果我们尝试将 `value` 直接传递给 `Math.Abs()`，会得到编译时异常，因为 `Math.Abs()` 没有接受 `object` 作为参数的重载。

如果 `value` 不能转换为 `int`，那么本例中的第二行将抛出 `InvalidCastException`

第37.6节：安全显式转换 (`as` 操作符)

如果你不确定某个值是否是你认为的类型，可以使用 `as` 操作符安全地转换它。如果值不是该类型，结果将是 `null`。

```
object value = "-1";
int? number = value as int?;
if(number != null)
{
    Console.WriteLine(Math.Abs(number.Value));
}
```

Section 37.4: LINQ Casting operations

Suppose you have types like the following:

```
interface IThing { }
class Thing : IThing { }
```

LINQ allows you to create a projection that changes the compile-time generic type of an `IEnumerable<>` via the `Enumerable.Cast<>()` and `Enumerable.OfType<>()` extension methods.

```
IEnumerable<IThing> things = new IThing[] {new Thing()};
IEnumerable<Thing> things2 = things.Cast<Thing>();
IEnumerable<Thing> things3 = things.OfType<Thing>();
```

When `things2` is evaluated, the `Cast<>()` method will try to cast all of the values in `things` into `Things`. If it encounters a value that cannot be cast, an `InvalidOperationException` will be thrown.

When `things3` is evaluated, the `OfType<>()` method will do the same, except that if it encounters a value that cannot be cast, it will simply omit that value rather than throw an exception.

Due to the generic type of these methods, they cannot invoke Conversion Operators or perform numeric conversions.

```
double[] doubles = new[]{1,2,3}.Cast<double>().ToArray(); // Throws InvalidOperationException
```

You can simply perform a cast inside a `.Select()` as a workaround:

```
double[] doubles = new[]{1,2,3}.Select(i => (double)i).ToArray();
```

Section 37.5: Explicit Casting

If you know that a value is of a specific type, you can explicitly cast it to that type in order to use it in a context where that type is needed.

```
object value = -1;
int number = (int) value;
Console.WriteLine(Math.Abs(number));
```

If we tried passing `value` directly to `Math.Abs()`，we would get a compile-time exception because `Math.Abs()` doesn't have an overload that takes an `object` as a parameter.

If `value` could not be cast to an `int`，then the second line in this example would throw an `InvalidOperationException`

Section 37.6: Safe Explicit Casting (`as` operator)

If you aren't sure whether a value is of the type you think it is, you can safely cast it using the `as` operator. If the value is not of that type, the resulting value will be `null`.

```
object value = "-1";
int? number = value as int?;
if(number != null)
{
    Console.WriteLine(Math.Abs(number.Value));
}
```

请注意，`null` 值没有类型，因此使用 `as` 关键字在转换任何 `null` 值时都会安全地返回 `null`。

第37.7节：隐式转换

如果编译器知道某个值总是可以转换为该类型，则该值会自动转换为适当的类型。

```
int number = -1;
object value = number;
Console.WriteLine(value);
```

在此示例中，我们不需要使用典型的显式转换语法，因为编译器知道所有 `int` 都可以转换为 `object`。实际上，我们可以避免创建变量，直接将 -1 作为参数传递给 `Console.WriteLine()`，该方法期望一个 `object` 类型的参数。

```
Console.WriteLine(-1);
```

第37.8节：显式数值转换

即使数值类型之间没有继承或实现关系，也可以使用显式转换运算符进行转换。

```
double value = -1.1;
int number = (int) value;
```

请注意，当目标类型的精度低于原始类型时，会丢失精度。例如，上述示例中作为 `double` 值的 -1.1 转换为整数值时变为 -1。

此外，数值转换依赖于编译时类型，因此如果数值类型已被“装箱”成对象，则无法进行转换。

```
object value = -1.1;
int number = (int) value; // 抛出 InvalidCastException
```

Note that `null` values have no type, so the `as` keyword will safely yield `null` when casting any `null` value.

Section 37.7: Implicit Casting

A value will automatically be cast to the appropriate type if the compiler knows that it can always be converted to that type.

```
int number = -1;
object value = number;
Console.WriteLine(value);
```

In this example, we didn't need to use the typical explicit casting syntax because the compiler knows all `ints` can be cast to `objects`. In fact, we could avoid creating variables and pass -1 directly as the argument of `Console.WriteLine()` that expects an `object`.

```
Console.WriteLine(-1);
```

Section 37.8: Explicit Numeric Conversions

Explicit casting operators can be used to perform conversions of numeric types, even though they don't extend or implement one another.

```
double value = -1.1;
int number = (int) value;
```

Note that in cases where the destination type has less precision than the original type, precision will be lost. For example, -1.1 as a `double` value in the above example becomes -1 as an integer value.

Also, numeric conversions rely on compile-time types, so they won't work if the numeric types have been "boxed" into objects.

```
object value = -1.1;
int number = (int) value; // throws InvalidCastException
```

第38章：可空类型

第38.1节：初始化可空类型

对于 null 值：

```
Nullable<int> i = null;
```

或者：

```
int? i = null;
```

或者：

```
var i = (int?)null;
```

对于非空值：

```
Nullable<int> i = 0;
```

或者：

```
int? i = 0;
```

第38.2节：检查Nullable是否有值

```
int? i = null;  
  
if (i != null)  
{  
    Console.WriteLine("i 不是 null");  
}  
else  
{  
    Console.WriteLine("i 是 null");  
}
```

这与以下代码相同：

```
if (i.HasValue)  
{  
    Console.WriteLine("i 不是 null");  
}  
else  
{  
    Console.WriteLine("i 是 null");  
}
```

第38.3节：获取nullable类型的值

给定以下 nullable int

```
int? i = 10;
```

Chapter 38: Nullable types

Section 38.1: Initialising a nullable

For null values:

```
Nullable<int> i = null;
```

Or:

```
int? i = null;
```

Or:

```
var i = (int?)null;
```

For non-null values:

```
Nullable<int> i = 0;
```

Or:

```
int? i = 0;
```

Section 38.2: Check if a Nullable has a value

```
int? i = null;  
  
if (i != null)  
{  
    Console.WriteLine("i is not null");  
}  
else  
{  
    Console.WriteLine("i is null");  
}
```

Which is the same as:

```
if (i.HasValue)  
{  
    Console.WriteLine("i is not null");  
}  
else  
{  
    Console.WriteLine("i is null");  
}
```

Section 38.3: Get the value of a nullable type

Given following nullable int

```
int? i = 10;
```

如果需要默认值，可以使用空合并运算符（null coalescing operator）、GetValueOrDefault方法，或者在赋值前检查可空int是否有值（HasValue）。

```
int j = i ?? 0;
int j = i.GetValueOrDefault(0);
int j = i.HasValue ? i.Value : 0;
```

以下用法始终是不安全的。如果运行时 i 为 null，将抛出 System.InvalidOperationException 异常。
设计时，如果未设置值，将出现 Use of unassigned local variable 'i' 错误。

```
int j = i.Value;
```

第38.4节：从可空类型获取默认值

.GetValueOrDefault() 方法即使在 .HasValue 属性为 false 时也会返回值（不同于 Value 属性，后者会抛出异常）。

```
class Program
{
    static void Main()
    {
        int? nullableExample = null;
        int result = nullableExample.GetValueOrDefault();
        Console.WriteLine(result); // 将输出 int 的默认值 - 0
        int secondResult = nullableExample.GetValueOrDefault(1);
        Console.WriteLine(secondResult) // 将输出我们指定的默认值 - 1
        int thirdResult = nullableExample ?? 1;
        Console.WriteLine(secondResult) // 与 GetValueOrDefault 相同，但更简短
    }
}
```

输出：

In case default value is needed, you can assign one using null coalescing operator, GetValueOrDefault method or check if nullable int HasValue before assignment.

```
int j = i ?? 0;
int j = i.GetValueOrDefault(0);
int j = i.HasValue ? i.Value : 0;
```

The following usage is always *unsafe*. If i is null at runtime, a System.InvalidOperationException will be thrown.
At design time, if a value is not set, you'll get a Use of unassigned local variable 'i' error.

```
int j = i.Value;
```

Section 38.4: Getting a default value from a nullable

The .GetValueOrDefault() method returns a value even if the .HasValue property is false (unlike the Value property, which throws an exception).

```
class Program
{
    static void Main()
    {
        int? nullableExample = null;
        int result = nullableExample.GetValueOrDefault();
        Console.WriteLine(result); // will output the default value for int - 0
        int secondResult = nullableExample.GetValueOrDefault(1);
        Console.WriteLine(secondResult) // will output our specified default - 1
        int thirdResult = nullableExample ?? 1;
        Console.WriteLine(secondResult) // same as the GetValueOrDefault but a bit shorter
    }
}
```

Output:

```
0
1
```

Section 38.5: Default value of nullable types is null

```
public class NullableTypesExample
{
    static int? _testValue;

    public static void Main()
    {
        if(_testValue == null)
            Console.WriteLine("null");
        else
            Console.WriteLine(_testValue.ToString());
    }
}
```

Output:

```
null
```

第38.6节：有效使用底层 Nullable<T> 参数

任何可空类型都是一个泛型类型。并且任何可空类型都是一个值类型。

有一些技巧可以有效使用 Nullable.GetUnderlyingType 方法的结果，当创建与反射/代码生成相关的代码时：

```
public static class TypesHelper {
    public static bool IsNullable(this Type type) {
        Type underlyingType;
        return IsNullable(type, out underlyingType);
    }
    public static bool IsNullable(this Type type, out Type underlyingType) {
        underlyingType = Nullable.GetUnderlyingType(type);
        return underlyingType != null;
    }
    public static Type GetNullable(Type type) {
        Type underlyingType;
        return IsNullable(type, out underlyingType) ? type : NullableTypesCache.Get(type);
    }
    public static bool IsExactOrNullable(this Type type, Func<Type, bool> predicate) {
        Type underlyingType;
        if(IsNullable(type, out underlyingType))
            return IsExactOrNullable(underlyingType, predicate);
        return predicate(type);
    }
    public static bool IsExactOrNullable<T>(this Type type)
        where T : struct {
            return IsExactOrNullable(type, t => Equals(t, typeof(T)));
    }
}
```

用法：

```
Type type = typeof(int).GetNullable();
Console.WriteLine(type.ToString());

if(type.IsDBNull())
    Console.WriteLine("类型是可空的。");
Type underlyingType;
if(type.IsDBNull(out underlyingType))
    Console.WriteLine("基础类型是 " + underlyingType.Name + ".");
if(type.IsExactOrNullable<int>())
    Console.WriteLine("类型是精确的或可空的 Int32。");
if(!type.IsExactOrNullable(t => t.IsEnum))
    Console.WriteLine("类型既不是精确的也不是可空的枚举。");
```

输出：

```
System.Nullable`1[System.Int32]
类型是可空的。
基础类型是 Int32。
类型是精确的或可空的 Int32。
类型既不是精确的也不是可空的枚举。
```

附注：NullableTypesCache 定义如下：

Section 38.6: Effective usage of underlying Nullable<T> argument

Any nullable type is a **generic** type. And any nullable type is a **value** type.

There are some tricks which allow to **effectively use** the result of the [Nullable.GetUnderlyingType](#) method when creating code related to reflection/code-generation purposes:

```
public static class TypesHelper {
    public static bool IsNullable(this Type type) {
        Type underlyingType;
        return IsNullable(type, out underlyingType);
    }
    public static bool IsNullable(this Type type, out Type underlyingType) {
        underlyingType = Nullable.GetUnderlyingType(type);
        return underlyingType != null;
    }
    public static Type GetNullable(Type type) {
        Type underlyingType;
        return IsNullable(type, out underlyingType) ? type : NullableTypesCache.Get(type);
    }
    public static bool IsExactOrNullable(this Type type, Func<Type, bool> predicate) {
        Type underlyingType;
        if(IsNullable(type, out underlyingType))
            return IsExactOrNullable(underlyingType, predicate);
        return predicate(type);
    }
    public static bool IsExactOrNullable<T>(this Type type)
        where T : struct {
            return IsExactOrNullable(type, t => Equals(t, typeof(T)));
    }
}
```

The usage:

```
Type type = typeof(int).GetNullable();
Console.WriteLine(type.ToString());

if(type.IsDBNull())
    Console.WriteLine("Type is nullable.");
Type underlyingType;
if(type.IsDBNull(out underlyingType))
    Console.WriteLine("The underlying type is " + underlyingType.Name + ".");
if(type.IsExactOrNullable<int>())
    Console.WriteLine("Type is either exact or nullable Int32.");
if(!type.IsExactOrNullable(t => t.IsEnum))
    Console.WriteLine("Type is neither exact nor nullable enum.");
```

Output:

```
System.Nullable`1[System.Int32]
Type is nullable.
The underlying type is Int32.
Type is either exact or nullable Int32.
Type is neither exact nor nullable enum.
```

PS. The NullableTypesCache is defined as follows:

```

static class NullableTypesCache {
    readonly static ConcurrentDictionary<Type, Type> cache = new ConcurrentDictionary<Type, Type>();
    static NullableTypesCache() {
        cache.TryAdd(typeof(byte), typeof(Nullable<byte>));
        cache.TryAdd(typeof(short), typeof(Nullable<short>));
        cache.TryAdd(typeof(int), typeof(Nullable<int>));
        cache.TryAdd(typeof(long), typeof(Nullable<long>));
        cache.TryAdd(typeof(float), typeof(Nullable<float>));
        cache.TryAdd(typeof(double), typeof(Nullable<double>));
        cache.TryAdd(typeof(decimal), typeof(Nullable<decimal>));
        cache.TryAdd(typeof(sbyte), typeof(Nullable<sbyte>));
        cache.TryAdd(typeof(ushort), typeof(Nullable<ushort>));
        cache.TryAdd(typeof(uint), typeof(Nullable<uint>));
        cache.TryAdd(typeof(ulong), typeof(Nullable<ulong>));
        //...
    }
    readonly static Type NullableBase = typeof(Nullable<>);
    internal static Type Get(Type type) {
        // 尽量避免调用开销较大的 MakeGenericType 方法
        return cache.GetOrAdd(type, t => NullableBase.MakeGenericType(t));
    }
}

```

第38.7节：检查泛型类型参数是否为可空类型

```

public bool IsTypeNullable<T>()
{
    return Nullable.GetUnderlyingType( typeof(T) )!=null;
}

```

```

static class NullableTypesCache {
    readonly static ConcurrentDictionary<Type, Type> cache = new ConcurrentDictionary<Type, Type>();
    static NullableTypesCache() {
        cache.TryAdd(typeof(byte), typeof(Nullable<byte>));
        cache.TryAdd(typeof(short), typeof(Nullable<short>));
        cache.TryAdd(typeof(int), typeof(Nullable<int>));
        cache.TryAdd(typeof(long), typeof(Nullable<long>));
        cache.TryAdd(typeof(float), typeof(Nullable<float>));
        cache.TryAdd(typeof(double), typeof(Nullable<double>));
        cache.TryAdd(typeof(decimal), typeof(Nullable<decimal>));
        cache.TryAdd(typeof(sbyte), typeof(Nullable<sbyte>));
        cache.TryAdd(typeof(ushort), typeof(Nullable<ushort>));
        cache.TryAdd(typeof(uint), typeof(Nullable<uint>));
        cache.TryAdd(typeof(ulong), typeof(Nullable<ulong>));
        //...
    }
    readonly static Type NullableBase = typeof(Nullable<>);
    internal static Type Get(Type type) {
        // Try to avoid the expensive MakeGenericType method call
        return cache.GetOrAdd(type, t => NullableBase.MakeGenericType(t));
    }
}

```

Section 38.7: Check if a generic type parameter is a nullable type

```

public bool IsTypeNullable<T>()
{
    return Nullable.GetUnderlyingType( typeof(T) )!=null;
}

```

第39章：构造函数和终结器

构造函数是类中的方法，当该类的实例被创建时会被调用。它们的主要职责是使新对象处于有用且一致的状态。

析构函数/终结器是类中的方法，当该类的实例被销毁时调用。在C#中，它们很少被显式编写/使用。

第39.1节：静态构造函数

静态构造函数是在类型的任何成员被初始化、静态类成员被调用或静态方法被调用的第一次执行。静态构造函数是线程安全的。静态构造函数通常用于：

- 初始化静态状态，即在同一类的不同实例之间共享的状态。
- 创建单例

示例：

```
类 Animal
{
    // * 静态构造函数只执行一次,
    //   当类首次被访问时。
    // * 静态构造函数不能有任何访问修饰符
    // * 静态构造函数不能有任何参数
    static Animal()
    {
        Console.WriteLine("Animal initialized");
    }

    // 实例构造函数, 每次创建类时都会执行
    public Animal()
    {
        Console.WriteLine("Animal created");
    }

    public static void Yawn()
    {
        Console.WriteLine("Yawn!");
    }
}

var turtle = new Animal();
var giraffe = new Animal();
```

输出：

```
Animal 初始化
Animal 创建
Animal 创建
```

[查看演示](#)

如果第一次调用的是静态方法，则会调用静态构造函数，而不会调用实例构造函数。这是可以的，因为静态方法无论如何都无法访问实例状态。

Chapter 39: Constructors and Finalizers

Constructors are methods in a class that are invoked when an instance of that class is created. Their main responsibility is to leave the new object in a useful and consistent state.

Destructors/Finalizers are methods in a class that are invoked when an instance of that is destroyed. In C# they are rarely explicitly written/used.

Section 39.1: Static constructor

A static constructor is called the first time any member of a type is initialized, a static class member is called or a static method. The static constructor is thread safe. A static constructor is commonly used to:

- Initialize static state, that is state which is shared across different instances of the same class.
- Create a singleton

Example:

```
class Animal
{
    // * A static constructor is executed only once,
    //   when a class is first accessed.
    // * A static constructor cannot have any access modifiers
    // * A static constructor cannot have any parameters
    static Animal()
    {
        Console.WriteLine("Animal initialized");
    }

    // Instance constructor, this is executed every time the class is created
    public Animal()
    {
        Console.WriteLine("Animal created");
    }

    public static void Yawn()
    {
        Console.WriteLine("Yawn!");
    }
}

var turtle = new Animal();
var giraffe = new Animal();
```

Output:

```
Animal initialized
Animal created
Animal created
```

[View Demo](#)

If the first call is to a static method, the static constructor is invoked without the instance constructor. This is OK, because the static method can't access instance state anyways.

```
Animal.Yawn();
```

这将输出：

```
动物初始化  
打哈欠！
```

另请参见静态构造函数中的异常和泛型静态构造函数。

单例示例：

```
public class SessionManager
{
    public static SessionManager Instance;

    static SessionManager()
    {
        Instance = new SessionManager();
    }
}
```

第39.2节：单例构造函数模式

```
public class SingletonClass
{
    public static SingletonClass Instance { get; } = new SingletonClass();

    private SingletonClass()
    {
        // 在此处放置自定义构造函数代码
    }
}
```

由于构造函数是私有的，使用代码无法创建新的SingletonClass实例。访问SingletonClass唯一实例的唯一方式是使用静态属性SingletonClass.Instance。

Instance属性由C#编译器生成的静态构造函数赋值。.NET运行时保证静态构造函数最多运行一次，并且在首次读取Instance之前运行。因此，所有的同步和初始化问题都由运行时处理。

注意，如果静态构造函数失败，Singleton类在整个AppDomain生命周期内将永久不可用。

此外，静态构造函数不保证在首次访问Instance时运行。相反，它会在此之前的某个时间点运行。这使得初始化发生的时间具有不确定性。在实际情况中，JIT通常会在引用Instance的方法编译（而非执行）期间调用静态构造函数。这是一种性能优化。

有关实现单例模式的其他方法，请参见单例实现页面。

第39.3节：默认构造函数

当类型定义时没有构造函数：

```
Animal.Yawn();
```

This will output:

```
Animal initialized  
Yawn!
```

See also Exceptions in static constructors and Generic Static Constructors.

Singleton example:

```
public class SessionManager
{
    public static SessionManager Instance;

    static SessionManager()
    {
        Instance = new SessionManager();
    }
}
```

Section 39.2: Singleton constructor pattern

```
public class SingletonClass
{
    public static SingletonClass Instance { get; } = new SingletonClass();

    private SingletonClass()
    {
        // Put custom constructor code here
    }
}
```

Because the constructor is private, no new instances of SingletonClass can be made by consuming code. The only way to access the single instance of SingletonClass is by using the static property SingletonClass.Instance.

The Instance property is assigned by a static constructor that the C# compiler generates. The .NET runtime guarantees that the static constructor is run at most once and is run before Instance is first read. Therefore, all synchronization and initialization concerns are carried out by the runtime.

Note, that if the static constructor fails the Singleton class becomes permanently unusable for the life of the AppDomain.

Also, the static constructor is not guaranteed to run at the time of the first access of Instance. Rather, it will run *at some point before that*. This makes the time at which initialization happens non-deterministic. In practical cases the JIT often calls the static constructor during *compilation* (not execution) of a method referencing Instance. This is a performance optimization.

See the Singleton Implementations page for other ways to implement the singleton pattern.

Section 39.3: Default Constructor

When a type is defined without a constructor:

```
public class Animal
{
}
```

那么编译器会生成一个等同于以下内容的默认构造函数：

```
public class Animal
{
    public Animal() {}
}
```

为该类型定义任何构造函数都会抑制默认构造函数的生成。如果该类型定义如下：

```
public class Animal
{
    public Animal(string name) {}
}
```

那么一个Animal只能通过调用声明的构造函数来创建。

```
// 这是有效的
var myAnimal = new Animal("Fluffy");
// 这将编译失败
var unnamedAnimal = new Animal();
```

对于第二个例子，编译器会显示错误信息：

'Animal' 不包含接受 0 个参数的构造函数

如果你希望一个类同时拥有无参构造函数和带参数的构造函数，可以通过显式实现这两个构造函数来实现。

```
public class Animal
{
    public Animal() {} // 等同于默认构造函数。
    public Animal(string name) {}
}
```

如果类继承了另一个没有无参构造函数的类，编译器将无法生成默认构造函数。例如，如果我们有一个类Creature：

```
public class Creature
{
    public Creature(Generics genus) {}
}
```

那么定义为class Animal : Creature {}的Animal类将无法编译。

第39.4节：强制调用静态构造函数

虽然静态构造函数总是在类型首次使用前调用，但有时强制调用它们是有用的，RuntimeHelpers类提供了一个辅助方法：

```
public class Animal
{
}
```

then the compiler generates a default constructor equivalent to the following:

```
public class Animal
{
    public Animal() {}
}
```

The definition of any constructor for the type will suppress the default constructor generation. If the type were defined as follows:

```
public class Animal
{
    public Animal(string name) {}
}
```

then an Animal could only be created by calling the declared constructor.

```
// This is valid
var myAnimal = new Animal("Fluffy");
// This fails to compile
var unnamedAnimal = new Animal();
```

For the second example, the compiler will display an error message:

'Animal' does not contain a constructor that takes 0 arguments

If you want a class to have both a parameterless constructor and a constructor that takes a parameter, you can do it by explicitly implementing both constructors.

```
public class Animal
{
    public Animal() {} //Equivalent to a default constructor.
    public Animal(string name) {}
}
```

The compiler will not be able to generate a default constructor if the class extends another class which doesn't have a parameterless constructor. For example, if we had a class Creature:

```
public class Creature
{
    public Creature(Generics genus) {}
}
```

then Animal defined as class Animal : Creature {} would not compile.

Section 39.4: Forcing a static constructor to be called

While static constructors are always called before the first usage of a type it's sometimes useful to be able to force them to be called and the RuntimeHelpers class provide an helper for it:

```
using System.Runtime.CompilerServices;
// ...
RuntimeHelpers.RunClassConstructor(typeof(Foo).TypeHandle);
```

备注：所有静态初始化（例如字段初始化器）都会执行，不仅仅是构造函数本身。

潜在用途：在UI应用的启动画面期间强制初始化，或确保静态构造函数在单元测试中不会失败。

第39.5节：从一个构造函数调用另一个构造函数

```
public class Animal
{
    public string Name { get; set; }

    public Animal() : this("Dog")
    {
    }

    public Animal(string name)
    {
        Name = name;
    }
}

var dog = new Animal(); // dog.Name 默认会被设置为 "Dog".
var cat = new Animal("Cat"); // cat.Name 是 "Cat"，不会调用无参构造函数。
```

```
using System.Runtime.CompilerServices;
// ...
RuntimeHelpers.RunClassConstructor(typeof(Foo).TypeHandle);
```

Remark: All static initialization (fields initializers for example) will run, not only the constructor itself.

Potential usages: Forcing initialization during the splash screen in an UI application or ensuring that a static constructor doesn't fail in a unit test.

Section 39.5: Calling a constructor from another constructor

```
public class Animal
{
    public string Name { get; set; }

    public Animal() : this("Dog")
    {
    }

    public Animal(string name)
    {
        Name = name;
    }
}

var dog = new Animal(); // dog.Name will be set to "Dog" by default.
var cat = new Animal("Cat"); // cat.Name is "Cat", the empty constructor is not called.
```

第39.6节：调用基类构造函数

基类的构造函数会在派生类的构造函数执行之前被调用。例如，如果哺乳动物 (Mammal) 继承自动物 (Animal)，那么在创建一个哺乳动物实例时，首先会调用动物 (Animal) 构造函数中的代码。

哺乳动物 (Mammal)。

如果派生类没有显式指定调用基类的哪个构造函数，编译器会默认调用无参构造函数。

```
public class Animal
{
    public Animal() { Console.WriteLine("一个未知的动物诞生了。"); }
    public Animal(string name) { Console.WriteLine(name + " 诞生了"); }
}

public class 哺乳动物 : 动物
{
    public 哺乳动物(string 名字)
    {
        Console.WriteLine(名字 + " 是一种哺乳动物。");
    }
}
```

在这种情况下，通过调用 `new 哺乳动物("乔治猫")` 实例化一个哺乳动物将打印

一个未知的动物诞生了。
乔治猫 是一种哺乳动物。

A constructor of a base class is called before a constructor of a derived class is executed. For example, if Mammal extends Animal, then the code contained in the constructor of Animal is called first when creating an instance of a Mammal.

If a derived class doesn't explicitly specify which constructor of the base class should be called, the compiler assumes the parameterless constructor.

```
public class Animal
{
    public Animal() { Console.WriteLine("An unknown animal gets born."); }
    public Animal(string name) { Console.WriteLine(name + " gets born"); }
}

public class Mammal : Animal
{
    public Mammal(string name)
    {
        Console.WriteLine(name + " is a mammal.");
    }
}
```

In this case, instantiating a Mammal by calling `new Mammal("George the Cat")` will print

An unknown animal gets born.
George the Cat is a mammal.

[查看演示](#)

调用基类的不同构造函数是通过在构造函数的

签名和其主体之间放置 : `base(参数)` 来完成的 :

```
public class 哺乳动物 : 动物
{
    public 哺乳动物(string 名字) : base(名字)
    {
        Console.WriteLine(名字 + " 是一种哺乳动物。");
    }
}
```

调用 `new 哺乳动物("乔治猫")` 现在将打印 :

猫乔治出生了。
乔治猫 是一种哺乳动物。

[查看演示](#)

第39.7节：派生类的终结器

当对象图被终结时，顺序与构造顺序相反。例如，超类型在基类型之前被终结，正如以下代码所示：

```
class 基类
{
    ~基类()
    {
        Console.WriteLine("基类已终结！");
    }
}

class 派生类 : 基类
{
    ~派生类()
    {
        Console.WriteLine("派生类已终结！");
    }
}

//不要赋值给变量
//以使对象变得不可达
new 派生类();

//仅为使示例可运行；
//否则不推荐这样做！
GC.Collect();

//派生类已终结！
//基类已终结！
```

第39.8节：静态构造函数中的异常

如果静态构造函数抛出异常，则不会重试。该类型在整个应用程序域（AppDomain）生命周期内不可用。对该类型的任何后续使用都会引发一个`TypeInitializationException`，该异常封装了原始

[View Demo](#)

Calling a different constructor of the base class is done by placing : `base(args)` between the constructor's signature and its body:

```
public class Mammal : Animal
{
    public Mammal(string name) : base(name)
    {
        Console.WriteLine(name + " is a mammal.");
    }
}
```

Calling `new Mammal("George the Cat")` will now print:

George the Cat gets born.
George the Cat is a mammal.

[View Demo](#)

Section 39.7: Finalizers on derived classes

When an object graph is finalized, the order is the reverse of the construction. E.g. the super-type is finalized before the base-type as the following code demonstrates:

```
class TheBaseClass
{
    ~TheBaseClass()
    {
        Console.WriteLine("Base class finalized!");
    }
}

class TheDerivedClass : TheBaseClass
{
    ~TheDerivedClass()
    {
        Console.WriteLine("Derived class finalized!");
    }
}

//Don't assign to a variable
//to make the object unreachable
new TheDerivedClass();

//Just to make the example work;
//this is otherwise NOT recommended!
GC.Collect();

//Derived class finalized!
//Base class finalized!
```

Section 39.8: Exceptions in static constructors

If a static constructor throws an exception, it is never retried. The type is unusable for the lifetime of the AppDomain. Any further usages of the type will raise a `TypeInitializationException` wrapped around the original

异常。

```
public class Animal
{
    static Animal()
    {
        Console.WriteLine("静态构造函数");
        throw new Exception();
    }

    public static void 打哈欠() {}

    try
    {
        Animal.Yawn();
    }
    catch (Exception e)
    {
        Console.WriteLine(e.ToString());
    }

    try
    {
        Animal.Yawn();
    }
    catch (Exception e)
    {
        Console.WriteLine(e.ToString());
    }
}
```

这将输出：

```
静态构造函数

System.TypeInitializationException : 'Animal' 类型的类型初始化器引发了异常。--->
System.Exception : 引发了类型为 'System.Exception' 的异常。
```

[...]

```
System.TypeInitializationException : 'Animal' 类型的类型初始化器引发了异常。--->
System.Exception : 引发了类型为 'System.Exception' 的异常。
```

你可以看到实际的构造函数只执行了一次，且异常被重复使用。

第39.9节：构造函数和属性初始化

属性值的赋值应在类的构造函数之前还是之后执行？

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
        if (TestProperty == 1)
```

exception.

```
public class Animal
{
    static Animal()
    {
        Console.WriteLine("Static ctor");
        throw new Exception();
    }

    public static void Yawn() {}

    try
    {
        Animal.Yawn();
    }
    catch (Exception e)
    {
        Console.WriteLine(e.ToString());
    }

    try
    {
        Animal.Yawn();
    }
    catch (Exception e)
    {
        Console.WriteLine(e.ToString());
    }
}
```

This will output:

```
Static ctor

System.TypeInitializationException: The type initializer for 'Animal' threw an exception.--->
System.Exception: Exception of type 'System.Exception' was thrown.
```

[...]

```
System.TypeInitializationException: The type initializer for 'Animal' threw an exception.--->
System.Exception: Exception of type 'System.Exception' was thrown.
```

where you can see that the actual constructor is only executed once, and the exception is re-used.

Section 39.9: Constructor and Property Initialization

Shall the property value's assignment be executed *before* or *after* the class' constructor?

```
public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
        if (TestProperty == 1)
```

```

{
Console.WriteLine("这应该被执行吗？");
}

if (TestProperty == 2)
{
Console.WriteLine("还是应该执行这个？");
}
}

var testInstance = new TestClass() { TestProperty = 1 };

```

在上面的例子中，`TestProperty` 的值是在类的构造函数中赋值，还是在类构造函数之后赋值？

像这样在实例创建时赋值属性：

```
var testInstance = new TestClass() {TestProperty = 1};
```

会在构造函数运行**之后**执行。然而，在 C# 6.0 中像这样在类的属性中初始化属性值：

```

public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
    }
}

```

将在构造函数运行之前完成。

将上述两个概念结合在一个示例中：

```

public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
        如果(TestProperty == 1)
        {
Console.WriteLine("这应该被执行吗？");
        }

        if (TestProperty == 2)
        {
Console.WriteLine("还是应该执行这个？");
        }
    }

    static void Main(string[] args)
    {
        var testInstance = new TestClass() { TestProperty = 1 };
        Console.WriteLine(testInstance.TestProperty); //结果为1
    }
}

```

```

{
    Console.WriteLine("Shall this be executed?");
}

if (TestProperty == 2)
{
    Console.WriteLine("Or shall this be executed");
}
}

var testInstance = new TestClass() { TestProperty = 1 };

```

In the example above, shall the `TestProperty` value be 1 in the class' constructor or after the class constructor?

Assigning property values in the instance creation like this:

```
var testInstance = new TestClass() {TestProperty = 1};
```

Will be executed **after** the constructor is run. However, initializing the property value in the class' property in C# 6.0 like this:

```

public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
    }
}

```

will be done **before** the constructor is run.

Combining the two concepts above in a single example:

```

public class TestClass
{
    public int TestProperty { get; set; } = 2;

    public TestClass()
    {
        if (TestProperty == 1)
        {
            Console.WriteLine("Shall this be executed?");
        }

        if (TestProperty == 2)
        {
            Console.WriteLine("Or shall this be executed");
        }
    }

    static void Main(string[] args)
    {
        var testInstance = new TestClass() { TestProperty = 1 };
        Console.WriteLine(testInstance.TestProperty); //resulting in 1
    }
}

```

最终结果：

“或者这将被执行”
“1”

解释：

TestProperty的值首先被赋为2，然后运行TestClass构造函数，结果打印出

“或者这将被执行”

然后由于new TestClass() { TestProperty = 1 }，TestProperty被赋值为1，使得最终由Console.WriteLine(testInstance.TestProperty)打印的TestProperty值为

1

第39.10节：通用静态构造函数

如果声明静态构造函数的类型是泛型，则静态构造函数将针对每个泛型参数的唯一组合调用一次。

```
class Animal<T>
{
    static Animal()
    {
        Console.WriteLine(typeof(T).FullName);
    }

    public static void Yawn() { }
}

Animal<Object>.Yawn();
Animal<String>.Yawn();
```

这将输出：

System.Object
System.String

另见 [泛型类型的静态构造函数如何工作？](#)

第39.11节：在构造函数中调用虚方法

与C++不同，在C#中你可以从类构造函数中调用虚方法（好吧，C++中也可以，但行为起初令人惊讶）。例如：

```
抽象类 Base
{
    受保护的 Base()
    {
        _obj = CreateAnother();
    }
}
```

Final result:

“Or shall this be executed”
“1”

Explanation:

The TestProperty value will first be assigned as 2, then the TestClass constructor will be run, resulting in printing of

“Or shall this be executed”

And then the TestProperty will be assigned as 1 due to new TestClass() { TestProperty = 1 }, making the final value for the TestProperty printed by Console.WriteLine(testInstance.TestProperty) to be

1

Section 39.10: Generic Static Constructors

If the type on which the static constructor is declared is generic, the static constructor will be called once for each unique combination of generic arguments.

```
class Animal<T>
{
    static Animal()
    {
        Console.WriteLine(typeof(T).FullName);
    }

    public static void Yawn() { }
}

Animal<Object>.Yawn();
Animal<String>.Yawn();
```

This will output:

System.Object
System.String

See also [How do static constructors for generic types work ?](#)

Section 39.11: Calling virtual methods in constructor

Unlike C++ in C# you can call a virtual method from class constructor (OK, you can also in C++ but behavior at first is surprising). For example:

```
抽象类 Base
{
    受保护的 Base()
    {
        _obj = CreateAnother();
    }
}
```

```

受保护的虚方法 AnotherBase CreateAnother()
{
    返回 new AnotherBase();
}

私有只读 AnotherBase _obj;
}

密封类 Derived : Base
{
    公共 Derived() { }

    受保护的重写 AnotherBase CreateAnother()
    {
        return new AnotherDerived();
    }
}

var test = new Derived();
// test._obj 是 AnotherDerived

```

如果你有 C++ 背景，这很令人惊讶，基类构造函数已经看到了派生类的虚方法表！

注意：派生类可能尚未完全初始化（其构造函数将在基类构造函数之后执行），这种技术很危险（对此还有 StyleCop 警告）。通常这被视为不良做法。

```

protected virtual AnotherBase CreateAnother()
{
    return new AnotherBase();
}

private readonly AnotherBase _obj;
}

sealed class Derived : Base
{
    public Derived() { }

    protected override AnotherBase CreateAnother()
    {
        return new AnotherDerived();
    }
}

var test = new Derived();
// test._obj is AnotherDerived

```

If you come from a C++ background this is surprising, base class constructor already sees derived class virtual method table!

Be careful: derived class may not be fully initialized yet (its constructor will be executed after base class constructor) and this technique is dangerous (there is also a StyleCop warning for this). Usually this is regarded as bad practice.

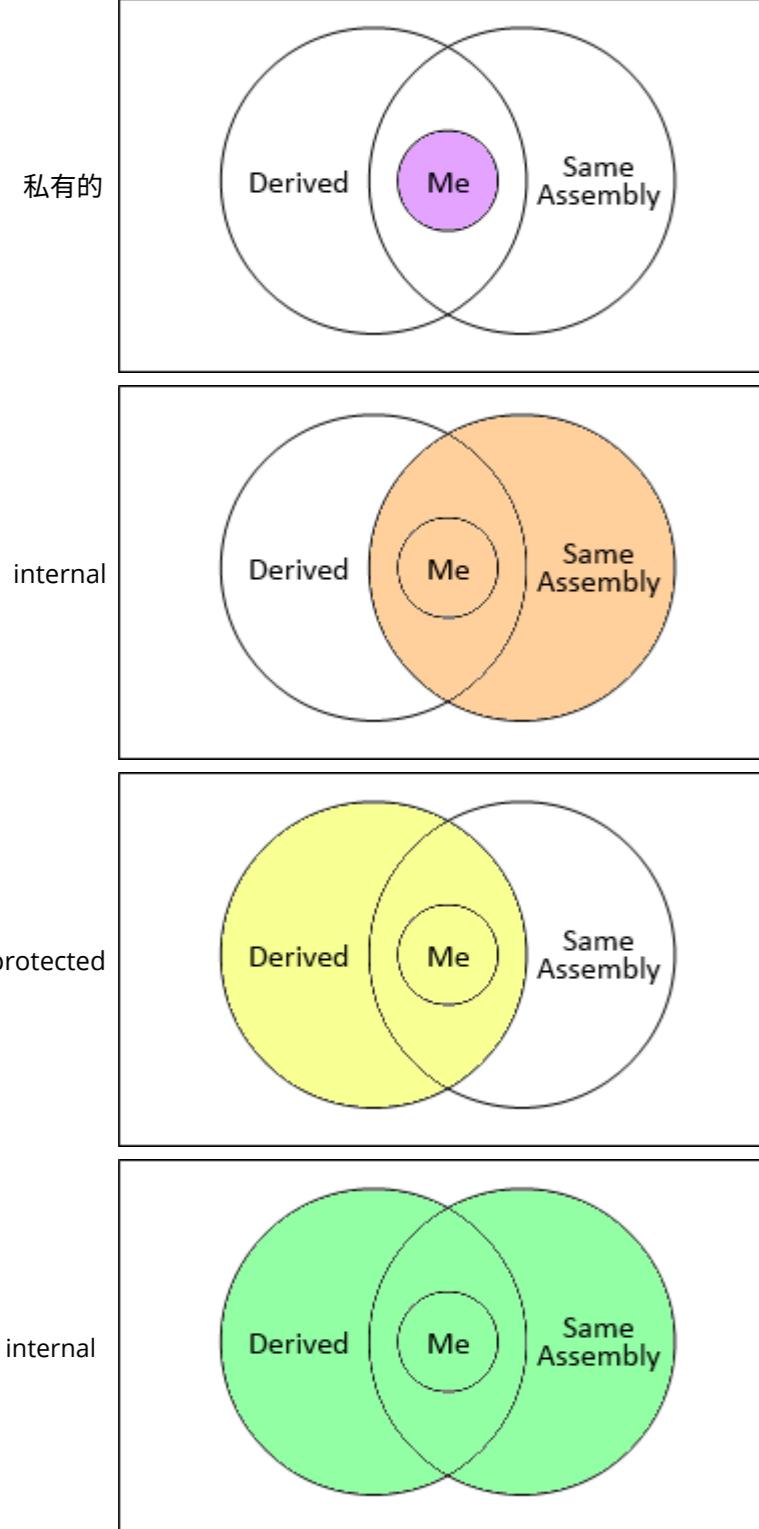
第40章：访问修饰符

第40.1节：访问修饰符图示

以下是所有访问修饰符的韦恩图，从限制更多到访问更广：

访问修饰符

图示



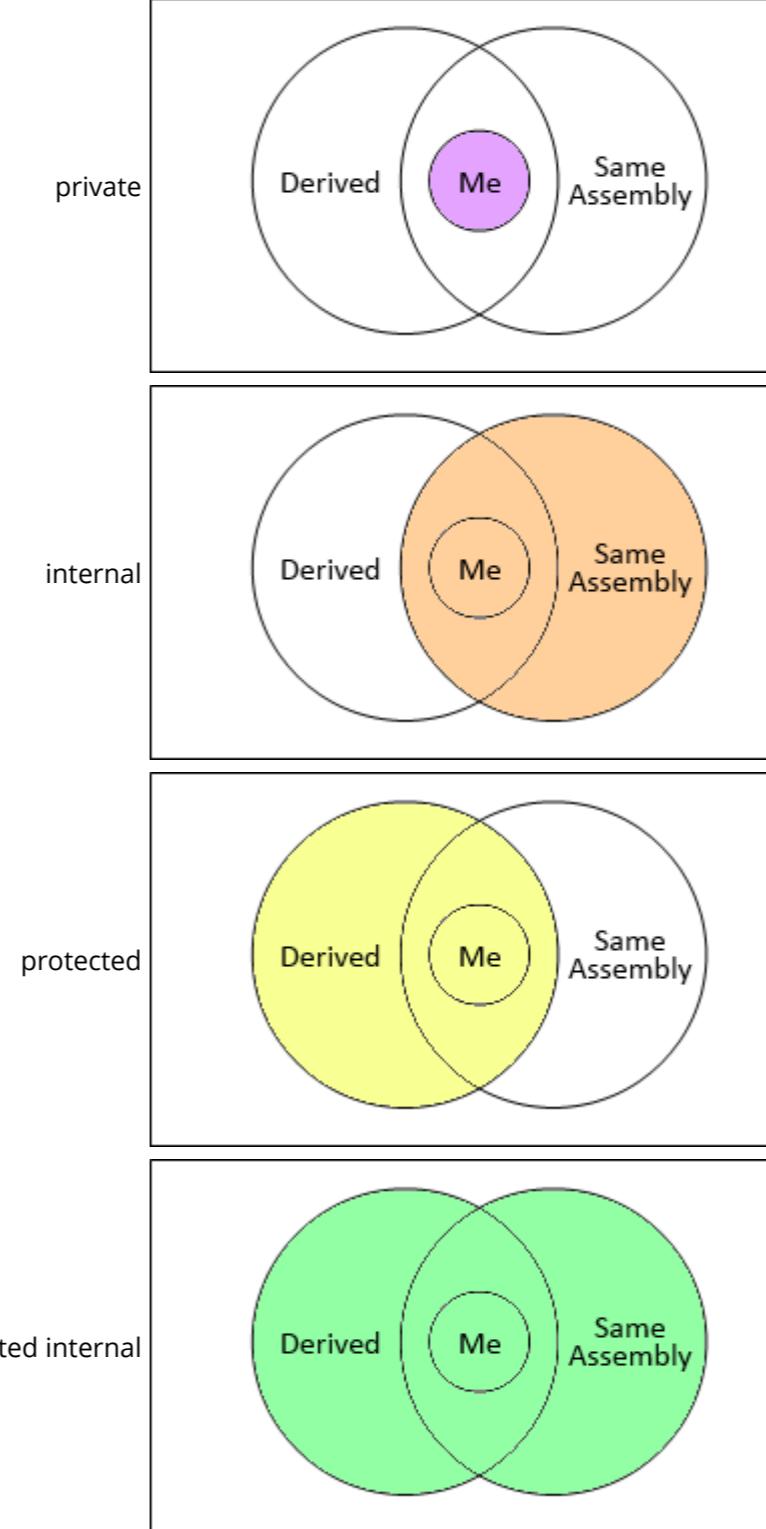
Chapter 40: Access Modifiers

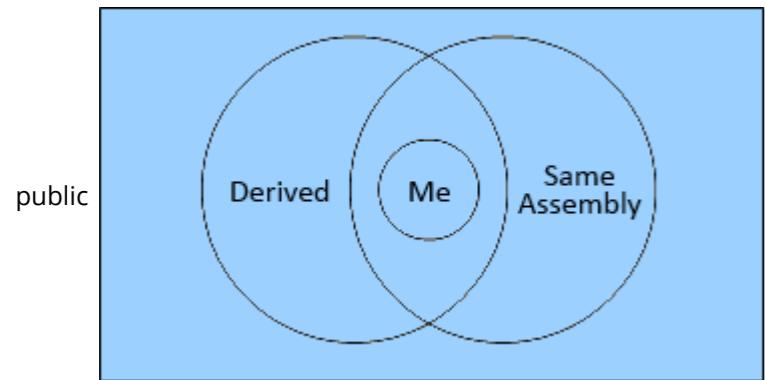
Section 40.1: Access Modifiers Diagrams

Here are all access modifiers in venn diagrams, from more limiting to more accessible:

Access Modifier

Diagram





public

以下您可以找到更多信息。

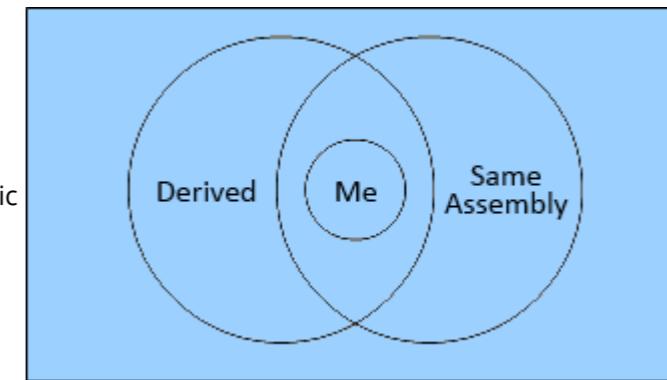
第40.2节 : public

`public`关键字使类（包括嵌套类）、属性、方法或字段对所有使用者可用：

```
public class Foo()
{
    public string SomeProperty { get; set; }

    public class Baz
    {
        public int Value { get; set; }
    }
}

public class Bar()
{
    public Bar()
    {
        var myInstance = new Foo();
        var someValue = foo.SomeProperty;
        var myNestedInstance = new Foo.Baz();
        var otherValue = myNestedInstance.Value;
    }
}
```



public

Below you could find more information.

Section 40.2: public

The `public` keyword makes a class (including nested classes), property, method or field available to every consumer:

```
public class Foo()
{
    public string SomeProperty { get; set; }

    public class Baz
    {
        public int Value { get; set; }
    }
}

public class Bar()
{
    public Bar()
    {
        var myInstance = new Foo();
        var someValue = foo.SomeProperty;
        var myNestedInstance = new Foo.Baz();
        var otherValue = myNestedInstance.Value;
    }
}
```

第40.3节 : private (私有)

`private` 关键字将属性、方法、字段和嵌套类标记为仅在类内部使用：

```
public class Foo()
{
    private string someProperty { get; set; }

    private class Baz
    {
        public string Value { get; set; }
    }

    public void Do()
    {
        var baz = new Baz { Value = 42 };
    }
}
```

Section 40.3: private

The `private` keyword marks properties, methods, fields and nested classes for use inside the class only:

```
public class Foo()
{
    private string someProperty { get; set; }

    private class Baz
    {
        public string Value { get; set; }
    }

    public void Do()
    {
        var baz = new Baz { Value = 42 };
    }
}
```

```

public class Bar()
{
    public Bar()
    {
        var myInstance = new Foo();

        // 编译错误 - 由于私有修饰符无法访问
        var someValue = foo.someProperty;
        // 编译错误 - 由于私有修饰符无法访问
        var baz = new Foo.Baz();
    }
}

```

第40.4节 : protected internal (受保护的内部)

关键字 `protected internal` 标记字段、方法、属性和嵌套类，可在同一程序集内或另一个程序集中的派生类中使用：

程序集1

```

public class Foo
{
    public string MyPublicProperty { get; set; }
    protected internal string MyProtectedInternalProperty { get; set; }

    protected internal class MyProtectedInternalNestedClass
    {
        private string blah;
        public int N { get; set; }
    }
}

public class Bar
{
    void MyMethod1()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }
}

```

程序集 2

```

public class Baz : Foo
{
    void MyMethod1()
    {
        var myPublicProperty = MyPublicProperty;
        var myProtectedInternalProperty = MyProtectedInternalProperty;
        var thing = new MyProtectedInternalNestedClass();
    }

    void MyMethod2()
}

```

```

public class Bar()
{
    public Bar()
    {
        var myInstance = new Foo();

        // Compile Error - not accessible due to private modifier
        var someValue = foo.someProperty;
        // Compile Error - not accessible due to private modifier
        var baz = new Foo.Baz();
    }
}

```

Section 40.4: protected internal

The `protected internal` keyword marks field, methods, properties and nested classes for use inside the same assembly or derived classes in another assembly:

Assembly 1

```

public class Foo
{
    public string MyPublicProperty { get; set; }
    protected internal string MyProtectedInternalProperty { get; set; }

    protected internal class MyProtectedInternalNestedClass
    {
        private string blah;
        public int N { get; set; }
    }
}

public class Bar
{
    void MyMethod1()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }
}

```

Assembly 2

```

public class Baz : Foo
{
    void MyMethod1()
    {
        var myPublicProperty = MyPublicProperty;
        var myProtectedInternalProperty = MyProtectedInternalProperty;
        var thing = new MyProtectedInternalNestedClass();
    }

    void MyMethod2()
}

```

```

{
    Foo foo = new Foo();
    var myPublicProperty = foo.MyPublicProperty;

    // 编译错误
    var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
    // 编译错误
    var myProtectedInternalNestedInstance =
        new Foo.MyProtectedInternalNestedClass();
}

public class Qux
{
    void MyMethod1()
    {
        Baz baz = new Baz();
        var myPublicProperty = baz.MyPublicProperty;

        // 编译错误
        var myProtectedInternalProperty = baz.MyProtectedInternalProperty;
        // 编译错误
        var myProtectedInternalNestedInstance =
            new Baz.MyProtectedInternalNestedClass();
    }

    void MyMethod2()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;

        // 编译错误
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        // 编译错误
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }
}

```

```

{
    Foo foo = new Foo();
    var myPublicProperty = foo.MyPublicProperty;

    // Compile Error
    var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
    // Compile Error
    var myProtectedInternalNestedInstance =
        new Foo.MyProtectedInternalNestedClass();
}

public class Qux
{
    void MyMethod1()
    {
        Baz baz = new Baz();
        var myPublicProperty = baz.MyPublicProperty;

        // Compile Error
        var myProtectedInternalProperty = baz.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Baz.MyProtectedInternalNestedClass();
    }

    void MyMethod2()
    {
        Foo foo = new Foo();
        var myPublicProperty = foo.MyPublicProperty;

        //Compile Error
        var myProtectedInternalProperty = foo.MyProtectedInternalProperty;
        // Compile Error
        var myProtectedInternalNestedInstance =
            new Foo.MyProtectedInternalNestedClass();
    }
}

```

第40.5节：internal

internal关键字使类（包括嵌套类）、属性、方法或字段对同一程序集中的所有使用者可用：

```

internal class Foo
{
    internal string SomeProperty {get; set;}
}

internal class Bar
{
    var myInstance = new Foo();
    internal string SomeField = foo.SomeProperty;

    internal class Baz
    {
        private string blah;
        public int N { get; set; }
    }
}

```

Section 40.5: internal

The internal keyword makes a class (including nested classes), property, method or field available to every consumer in the same assembly:

```

internal class Foo
{
    internal string SomeProperty {get; set;}
}

internal class Bar
{
    var myInstance = new Foo();
    internal string SomeField = foo.SomeProperty;

    internal class Baz
    {
        private string blah;
        public int N { get; set; }
    }
}

```

```
}
```

可以通过向 AssemblyInfo.cs 文件添加代码，允许测试程序集访问代码：

```
using System.Runtime.CompilerServices;  
[assembly:InternalsVisibleTo("MyTests")]
```

第40.6节：protected（受保护的）

protected 关键字标记字段、方法、属性和嵌套类，仅供同一类及其派生类内部使用：

```
public class Foo()  
{  
    protected void SomeFooMethod()  
    {  
        //执行某些操作  
    }  
  
    protected class Thing  
    {  
        private string blah;  
        public int N { get; set; }  
    }  
}  
  
public class Bar() : Foo  
{  
    private void someBarMethod()  
    {  
        SomeFooMethod(); // 在派生类内部  
        var thing = new Thing(); // 可以使用嵌套类  
    }  
}  
  
public class Baz()  
{  
    private void someBazMethod()  
    {  
        var foo = new Foo();  
        foo.SomeFooMethod(); // 由于受保护修饰符，无法访问  
    }  
}
```

```
}
```

This can be broken to allow a testing assembly to access the code via adding code to AssemblyInfo.cs file:

```
using System.Runtime.CompilerServices;  
[assembly:InternalsVisibleTo("MyTests")]
```

Section 40.6: protected

The **protected** keyword marks field, methods properties and nested classes for use inside the same class and derived classes only:

```
public class Foo()  
{  
    protected void SomeFooMethod()  
    {  
        //do something  
    }  
  
    protected class Thing  
    {  
        private string blah;  
        public int N { get; set; }  
    }  
}  
  
public class Bar() : Foo  
{  
    private void someBarMethod()  
    {  
        SomeFooMethod(); // inside derived class  
        var thing = new Thing(); // can use nested class  
    }  
}  
  
public class Baz()  
{  
    private void someBazMethod()  
    {  
        var foo = new Foo();  
        foo.SomeFooMethod(); //not accessible due to protected modifier  
    }  
}
```

第41章：接口

第41.1节：实现接口

接口用于强制任何“实现”它的类必须包含某个方法。接口通过关键字interface定义，类可以通过在类名后添加:接口名来“实现”它。一个类可以通过逗号分隔实现多个接口。

```
: InterfaceName, ISecondInterface
```

```
public interface INoiseMaker
{
    string MakeNoise();
}

public class Cat : INoiseMaker
{
    public string MakeNoise()
    {
        return "Nyan";
    }
}

public class Dog : INoiseMaker
{
    public string MakeNoise()
    {
        return "Woof";
    }
}
```

因为它们实现了INoiseMaker接口，cat和dog都必须包含string MakeNoise()方法，否则将无法编译。

第41.2节：显式接口实现

当你实现多个接口且这些接口定义了一个相同的方法时，但根据调用该方法的接口不同需要不同的实现，显式接口实现是必要的（注意，如果多个接口共享相同的方法且可以使用通用实现，则不需要显式实现）。

```
interface IChauffeur
{
    string Drive();
}

接口 IGolfPlayer
{
    字符串 Drive();
}

类 GolfingChauffeur : IChauffeur, IGolfPlayer
{
    公共字符串 Drive()
    {
        返回 "Vroom!";
    }
}
```

Chapter 41: Interfaces

Section 41.1: Implementing an interface

An interface is used to enforce the presence of a method in any class that 'implements' it. The interface is defined with the keyword `interface` and a class can 'implement' it by adding `: InterfaceName` after the class name. A class can implement multiple interfaces by separating each interface with a comma.

```
: InterfaceName, ISecondInterface
```

```
public interface INoiseMaker
{
    string MakeNoise();
}

public class Cat : INoiseMaker
{
    public string MakeNoise()
    {
        return "Nyan";
    }
}

public class Dog : INoiseMaker
{
    public string MakeNoise()
    {
        return "Woof";
    }
}
```

Because they implement INoiseMaker, both cat and dog are required to include the `string MakeNoise()` method and will fail to compile without it.

Section 41.2: Explicit interface implementation

Explicit interface implementation is necessary when you implement multiple interfaces who define a common method, but different implementations are required depending on which interface is being used to call the method (note that you don't need explicit implementations if multiple interfaces share the same method and a common implementation is possible).

```
interface IChauffeur
{
    string Drive();
}

interface IGolfPlayer
{
    string Drive();
}

class GolfingChauffeur : IChauffeur, IGolfPlayer
{
    public string Drive()
    {
        return "Vroom!";
    }
}
```

```
字符串 IGolfPlayer.Drive()
{
    返回 "挥杆了...";
}
}
```

```
GolfingChauffeur 对象 = 新建 GolfingChauffeur();
IChauffeur 司机 = 对象;
IGolfPlayer 高尔夫球手 = 对象;
```

```
控制台.写行(对象.Drive()); // Vroom!
Console.WriteLine(chauffeur.Drive()); // 嗡嗡声!
Console.WriteLine(golfer.Drive()); // 挥杆了.....
```

该实现只能通过接口调用，不能从其他任何地方调用：

```
public class 高尔夫球手 : IGolfPlayer
{
    字符串 IGolfPlayer.Drive()
    {
        返回 "用力挥杆...";
    }
    public void Swing()
    {
        Drive(); // 编译错误：没有此方法
    }
}
```

因此，将显式接口实现的复杂代码放入一个单独的私有方法中可能更有利。

显式接口实现当然只能用于该接口实际存在的方法：

```
public class 职业高尔夫球手 : IGolfPlayer
{
    string IGolfPlayer.Swear() // 错误
    {
        返回 "球在坑里";
    }
}
```

同样，使用显式接口实现而未在类上声明该接口也会导致错误。

提示：

显式实现接口也可以用来避免死代码。当某个方法不再需要并且从接口中移除时，编译器会对每个仍然存在的实现发出警告。

注意：

程序员期望契约在类型的任何上下文中都是相同的，显式实现不应在调用时表现出不同的行为。因此，与上面的例子不同，IGolfPlayer.Drive 和 Drive 在可能的情况下应执行相同的操作。

```
string IGolfPlayer.Drive()
{
    return "Took a swing...";
}
}
```

```
GolfingChauffeur obj = new GolfingChauffeur();
IChauffeur chauffeur = obj;
IGolfPlayer golfer = obj;
```

```
Console.WriteLine(obj.Drive()); // Vroom!
Console.WriteLine(chauffeur.Drive()); // Vroom!
Console.WriteLine(golfer.Drive()); // Took a swing...
```

The implementation cannot be called from anywhere else except by using the interface:

```
public class Golfer : IGolfPlayer
{
    string IGolfPlayer.Drive()
    {
        return "Swinging hard...";
    }
    public void Swing()
    {
        Drive(); // Compiler error: No such method
    }
}
```

Due to this, it may be advantageous to put complex implementation code of an explicitly implemented interface in a separate, private method.

An explicit interface implementation can of course only be used for methods that actually exist for that interface:

```
public class ProGolfer : IGolfPlayer
{
    string IGolfPlayer.Swear() // Error
    {
        返回 "The ball is in the pit";
    }
}
```

Similarly, using an explicit interface implementation without declaring that interface on the class causes an error, too.

Hint:

Implementing interfaces explicitly can also be used to avoid dead code. When a method is no longer needed and gets removed from the interface, the compiler will complain about each still existing implementation.

Note:

Programmers expect the contract to be the same regardless of the context of the type and explicit implementation should not expose different behavior when called. So unlike the example above, IGolfPlayer.Drive and Drive should do the same thing when possible.

第41.3节：接口基础

接口的功能被称为功能的“契约”。这意味着它声明属性和方法，但不实现它们。

所以与类不同，接口：

- 不能被实例化
- 不能有任何功能
- 只能包含方法 * (属性和事件在内部也是方法)
- 继承接口称为“实现”
- 你可以继承一个类，但可以“实现”多个接口

```
public interface ICanDoThis{
    void TheThingICanDo();
    int SomeValueProperty { get; set; }
}
```

需要注意的事项：

- “I”前缀是接口的命名约定。
- 函数体用分号“;”代替。
- 属性也是允许的，因为它们在内部也是方法

```
public class MyClass : ICanDoThis {
    public void TheThingICanDo(){
        // 执行该操作
    }

    public int SomeValueProperty { get; set; }
    public int SomeValueNotImplemtingAnything { get; set; }
}
```

```
ICanDoThis obj = new MyClass();

// 好的
obj.TheThingICanDo();

// 好的
obj.SomeValueProperty = 5;

// 错误，该成员在接口中不存在
obj.SomeValueNotImplemtingAnything = 5;

// 要访问类中的属性，必须进行“向下转换”
((MyClass)obj).SomeValueNotImplemtingAnything = 5; // 好的
```

这在使用诸如 WinForms 或 WPF 之类的 UI 框架时尤其有用，因为创建用户控件必须继承自基类，这会失去对不同控件类型进行抽象的能力。举个例子？马上展示：

```
public class MyTextBlock : TextBlock {
    public void SetText(string str){
        this.Text = str;
    }
}
```

Section 41.3: Interface Basics

An Interface's function known as a "contract" of functionality. It means that it declares properties and methods but it doesn't implement them.

So unlike classes Interfaces:

- Can't be instantiated
- Can't have any functionality
- Can only contain methods * (*Properties and Events are methods internally*)
- Inheriting an interface is called "Implementing"
- You can inherit from 1 class, but you can "Implement" multiple Interfaces

```
public interface ICanDoThis{
    void TheThingICanDo();
    int SomeValueProperty { get; set; }
}
```

Things to notice:

- The "I" prefix is a naming convention used for interfaces.
- The function body is replaced with a semicolon ";".
- Properties are also allowed because internally they are also methods

```
public class MyClass : ICanDoThis {
    public void TheThingICanDo(){
        // do the thing
    }

    public int SomeValueProperty { get; set; }
    public int SomeValueNotImplemtingAnything { get; set; }
}
```

```
ICanDoThis obj = new MyClass();

// ok
obj.TheThingICanDo();

// ok
obj.SomeValueProperty = 5;

// Error, this member doesn't exist in the interface
obj.SomeValueNotImplemtingAnything = 5;

// in order to access the property in the class you must "down cast" it
((MyClass)obj).SomeValueNotImplemtingAnything = 5; // ok
```

This is especially useful when you're working with UI frameworks such as WinForms or WPF because it's mandatory to inherit from a base class to create user control and you lose the ability to create abstraction over different control types. An example? Coming up:

```
public class MyTextBlock : TextBlock {
    public void SetText(string str){
        this.Text = str;
    }
}
```

```

    }

public class MyButton : Button {
    public void SetText(string str){
        this.Content = str;
    }
}

```

提出的问题是两者都包含某种“文本”概念，但属性名称不同。并且你不能创建一个抽象基类，因为它们必须分别继承自两个不同的类。接口可以缓解这个问题

```

public interface ITextControl{
    void SetText(string str);
}

public class MyTextBlock : TextBlock, ITextControl {
    public void SetText(string str){
        this.Text = str;
    }
}

public class MyButton : Button, ITextControl {
    public void SetText(string str){
        this.Content = str;
    }
}

public int Clicks { get; set; }
}

```

现在 MyButton 和 MyTextBlock 可以互换使用。

```

var controls = new List<ITextControls>{
    new MyTextBlock(),
    new MyButton()
};

foreach(var ctrl in controls){
    ctrl.SetText("尽管控件不同，这段文本将应用于两个控件");

    // 编译错误，接口中没有此成员
    ctrl.Clicks = 0;

    // 运行时错误，因为其中一个类实际上不是按钮，导致此转换无效
    ((MyButton)ctrl).Clicks = 0;

    /* 解决方案是先检查类型。
    这通常被认为是不良做法，因为
    它是抽象设计不良的表现 */
    var button = ctrl as MyButton;
    if(button != null)
        button.Clicks = 0; // 无错误
}

```

```

    }

public class MyButton : Button {
    public void SetText(string str){
        this.Content = str;
    }
}

```

The problem proposed is that both contain some concept of "Text" but the property names differ. And you can't create a *abstract base class* because they have a mandatory inheritance to 2 different classes. An interface can alleviate that

```

public interface ITextControl{
    void SetText(string str);
}

public class MyTextBlock : TextBlock, ITextControl {
    public void SetText(string str){
        this.Text = str;
    }
}

public class MyButton : Button, ITextControl {
    public void SetText(string str){
        this.Content = str;
    }
}

public int Clicks { get; set; }
}

```

Now MyButton and MyTextBlock is interchangeable.

```

var controls = new List<ITextControls>{
    new MyTextBlock(),
    new MyButton()
};

foreach(var ctrl in controls){
    ctrl.SetText("This text will be applied to both controls despite them being different");

    // Compiler Error, no such member in interface
    ctrl.Clicks = 0;

    // Runtime Error because 1 class is in fact not a button which makes this cast invalid
    ((MyButton)ctrl).Clicks = 0;

    /* the solution is to check the type first.
    This is usually considered bad practice since
    it's a symptom of poor abstraction */
    var button = ctrl as MyButton;
    if(button != null)
        button.Clicks = 0; // no errors
}

```

第41.4节：以IComparable<T>为例实现接口

接口看起来可能很抽象，直到你在实践中看到它们。IComparable和IComparable<T>是接口为何对我们有用的很好的例子。

假设在一个在线商店的程序中，我们有各种可以购买的商品。每个商品都有一个名称、一个ID号码和一个价格。

```
public class Item {  
  
    public string name; // 虽然公共变量通常是不好的做法,  
    public int idNumber; // 为了简化示例，我们将使用它们而不是属性。  
    public decimal price;  
  
    // 省略主体内容  
}
```

我们的Item存储在一个List<Item>中，在程序的某处，我们想按ID号码从小到大排序列表。我们不需要自己编写排序算法，而是可以使用List<T>已经有的Sort()方法。然而，当前的Item类无法让List<T>理解如何排序列表。这时IComparable接口就派上用场了。

为了正确实现CompareTo方法，CompareTo应当在参数“小于”当前对象时返回正数，参数等于当前对象时返回零，参数“大于”当前对象时返回负数。

```
Item apple = new Item();  
apple.idNumber = 15;  
Item banana = new Item();  
banana.idNumber = 4;  
Item cow = new Item();  
cow.idNumber = 15;  
Item diamond = new Item();  
diamond.idNumber = 18;  
  
Console.WriteLine(apple.CompareTo(banana)); // 11  
Console.WriteLine(apple.CompareTo(cow)); // 0  
Console.WriteLine(apple.CompareTo(diamond)); // -3
```

这是示例Item实现接口的代码：

```
public class Item : IComparable<Item> {  
  
    private string name;  
    private int idNumber;  
    private decimal price;  
  
    public int CompareTo(Item otherItem) {  
  
        return (this.idNumber - otherItem.idNumber);  
    }  
  
    // 其余代码为简洁起见省略  
}
```

Section 41.4: IComparable<T> as an Example of Implementing an Interface

Interfaces can seem abstract until you see them in practice. The IComparable and IComparable<T> are great examples of why interfaces can be helpful to us.

Let's say that in a program for an online store, we have a variety of items you can buy. Each item has a name, an ID number, and a price.

```
public class Item {  
  
    public string name; // though public variables are generally bad practice,  
    public int idNumber; // to keep this example simple we will use them instead  
    public decimal price; // of a property.  
  
    // body omitted for brevity  
}
```

We have our Items stored inside of a List<Item>, and in our program somewhere, we want to sort our list by ID number from smallest to largest. Instead of writing our own sorting algorithm, we can instead use the Sort() method that List<T> already has. However, as our Item class is right now, there is no way for the List<T> to understand what order to sort the list. Here is where the IComparable interface comes in.

To correctly implement the CompareTo method, CompareTo should return a positive number if the parameter is "less than" the current one, zero if they are equal, and a negative number if the parameter is "greater than".

```
Item apple = new Item();  
apple.idNumber = 15;  
Item banana = new Item();  
banana.idNumber = 4;  
Item cow = new Item();  
cow.idNumber = 15;  
Item diamond = new Item();  
diamond.idNumber = 18;  
  
Console.WriteLine(apple.CompareTo(banana)); // 11  
Console.WriteLine(apple.CompareTo(cow)); // 0  
Console.WriteLine(apple.CompareTo(diamond)); // -3
```

Here's the example Item's implementation of the interface:

```
public class Item : IComparable<Item> {  
  
    private string name;  
    private int idNumber;  
    private decimal price;  
  
    public int CompareTo(Item otherItem) {  
  
        return (this.idNumber - otherItem.idNumber);  
    }  
  
    // rest of code omitted for brevity  
}
```

在表面上，我们的项中的CompareTo方法只是返回它们ID号的差值，但上述代码在实际中是做什么的呢？

现在，当我们在List<Item>对象上调用Sort()时，List会在需要确定对象排序顺序时自动调用Item的CompareTo方法。此外，除了List<T>之外，任何需要比较两个对象能力的其他对象都可以使用Item，因为我们已经定义了两个不同Item之间的比较能力。

第41.5节：实现多个接口

```
public interface IAnimal
{
    string Name { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

public class Cat : IAnimal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
    }

    public string Name { get; set; }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

第41.6节：我们为什么使用接口

接口是接口的使用者与实现该接口的类之间契约的定义。理解接口的一种方式是将其视为声明一个对象可以执行某些功能。

假设我们定义了一个接口IShape来表示不同类型的形状，我们期望一个形状具有面积，因此我们将定义一个方法，强制接口的实现返回它们的面积：

```
public interface IShape
{
    double ComputeArea();
}
```

假设我们有以下两种形状：一个Rectangle和一个Circle

```
public class Rectangle : IShape
{
    private double length;
    private double width;

    public Rectangle(double length, double width)
    {
        this.length = length;
    }
}
```

On a surface level, the CompareTo method in our item simply returns the difference in their ID numbers, but what does the above do in practice?

Now, when we call Sort() on a List<Item> object, the List will automatically call the Item's CompareTo method when it needs to determine what order to put objects in. Furthermore, besides List<T>, any other objects that need the ability to compare two objects will work with the Item because we have defined the ability for two different Items to be compared with one another.

Section 41.5: Implementing multiple interfaces

```
public interface IAnimal
{
    string Name { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

public class Cat : IAnimal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
    }

    public string Name { get; set; }

    public string MakeNoise()
    {
        return "Nyan";
    }
}
```

Section 41.6: Why we use interfaces

An interface is a definition of a contract between the user of the interface and the class that implement it. One way to think of an interface is as a declaration that an object can perform certain functions.

Let's say that we define an interface IShape to represent different type of shapes, we expect a shape to have an area, so we will define a method to force the interface implementations to return their area :

```
public interface IShape
{
    double ComputeArea();
}
```

Let's that we have the following two shapes : a Rectangle and a Circle

```
public class Rectangle : IShape
{
    private double length;
    private double width;

    public Rectangle(double length, double width)
    {
        this.length = length;
    }
}
```

```

        this.width = width;
    }

    public double ComputeArea()
    {
        return length * width;
    }
}

public class Circle : IShape
{
    private double radius;

    public Circle(double radius)
    {
        this.radius = radius;
    }

    public double ComputeArea()
    {
        return Math.Pow(radius, 2.0) * Math.PI;
    }
}

```

它们每个都有自己的面积定义，但它们都是形状。因此，在我们的程序中将它们视为 IShape 是合乎逻辑的：

```

private static void Main(string[] args)
{
    var shapes = new List<IShape>() { new Rectangle(5, 10), new Circle(5) };
    ComputeArea(shapes);

    Console.ReadKey();
}

private static void ComputeArea(IEnumerable<IShape> shapes)
{
    foreach (shape in shapes)
    {
        Console.WriteLine("Area: {0:N}, shape.ComputeArea()");
    }
}

// 输出：
// Area : 50.00
// Area : 78.54

```

第41.7节：“隐藏”成员的显式实现

你是否讨厌接口给你的类带来太多你根本不关心的成员？我有一个解决方案！显式实现

```

public interface IMessageService {
    void OnMessageRecieve();
    void SendMessage();
    string Result { get; set; }
    int Encoding { get; set; }
    // yadda yadda
}

```

```

        this.width = width;
    }

    public double ComputeArea()
    {
        return length * width;
    }
}

public class Circle : IShape
{
    private double radius;

    public Circle(double radius)
    {
        this.radius = radius;
    }

    public double ComputeArea()
    {
        return Math.Pow(radius, 2.0) * Math.PI;
    }
}

```

Each one of them have its own definition of its area, but both of them are shapes. So it's only logical to see them as IShape in our program :

```

private static void Main(string[] args)
{
    var shapes = new List<IShape>() { new Rectangle(5, 10), new Circle(5) };
    ComputeArea(shapes);

    Console.ReadKey();
}

private static void ComputeArea(IEnumerable<IShape> shapes)
{
    foreach (shape in shapes)
    {
        Console.WriteLine("Area: {0:N}, shape.ComputeArea()");
    }
}

// Output:
// Area : 50.00
// Area : 78.54

```

Section 41.7: "Hiding" members with Explicit Implementation

Don't you hate it when interfaces pollute your class with too many members you don't even care about? Well I got a solution! Explicit Implementations

```

public interface IMessageService {
    void OnMessageRecieve();
    void SendMessage();
    string Result { get; set; }
    int Encoding { get; set; }
    // yadda yadda
}

```

通常你会这样实现这个类。

```
public class MyObjectWithMessages : IMessageService {
    public void OnMessageRecieve(){

    }

    public void SendMessage(){

    }

    public string Result { get; set; }
    public int Encoding { get; set; }
}
```

每个成员都是公共的。

```
var obj = new MyObjectWithMessages();

// 我为什么要调用这个函数？
obj.OnMessageRecieve();
```

回答：我不这样做。所以它也不应该被声明为 `public`，只需将成员声明为 `private`，编译器就会报错

解决方案是使用显式实现：

```
public class MyObjectWithMessages : IMessageService{
    void IMessageService.OnMessageRecieve() {

    }

    void IMessageService.SendMessage() {

    }

    string IMessageService.Result { get; set; }
    int IMessageService.Encoding { get; set; }
}
```

所以现在你已经按要求实现了成员，并且它们不会以公共成员的形式暴露。

```
var obj = new MyObjectWithMessages();

/* 错误：成员在类型 MyObjectWithMessages 上不存在。
 * 我们已经成功将其设为“私有” */
obj.OnMessageRecieve();
```

如果你真的还想访问该成员，尽管它是显式实现的，你所要做的就是将对象强制转换为接口类型，就可以使用了。

```
((IMessageService)obj).OnMessageRecieve();
```

Normally you'd implement the class like this.

```
public class MyObjectWithMessages : IMessageService {
    public void OnMessageRecieve(){

    }

    public void SendMessage(){

    }

    public string Result { get; set; }
    public int Encoding { get; set; }
}
```

Every member is public.

```
var obj = new MyObjectWithMessages();

// why would i want to call this function?
obj.OnMessageRecieve();
```

Answer: I don't. So neither should it be declared public but simply declaring the members as private will make the compiler throw an error

The solution is to use explicit implementation:

```
public class MyObjectWithMessages : IMessageService{
    void IMessageService.OnMessageRecieve() {

    }

    void IMessageService.SendMessage() {

    }

    string IMessageService.Result { get; set; }
    int IMessageService.Encoding { get; set; }
}
```

So now you have implemented the members as required and they won't expose any members in as public.

```
var obj = new MyObjectWithMessages();

/* error member does not exist on type MyObjectWithMessages.
 * We've successfully made it "private" */
obj.OnMessageRecieve();
```

If you seriously still want to access the member even though is explicitly implement all you have to do is cast the object to the interface and you good to go.

```
((IMessageService)obj).OnMessageRecieve();
```

第42章：静态类

第42.1节：静态类

当提到类时，“static”关键字有三个作用：

1. 你不能创建静态类的实例（这甚至移除了默认构造函数）
2. 类中的所有属性和方法也必须是静态的。
3. 一个static类是一个sealed类，意味着它不能被继承。

```
public static class Foo
{
    //注意这里没有构造函数，因为不能创建实例
    public static int Counter { get; set; }
    public static int GetCount()
    {
        return Counter;
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Foo.Counter++;
        Console.WriteLine(Foo.GetCount()); //这将打印1

        //var foo1 = new Foo();
        //这行代码会导致错误，因为Foo类没有构造函数
    }
}
```

第42.2节：静态类的生命周期

一个静态类在成员访问时延迟初始化，并在应用程序域的整个生命周期内存在。

```
void Main()
{
    Console.WriteLine("静态类是延迟初始化的");
    Console.WriteLine("静态构造函数仅在首次访问类时调用");
    Foo.SayHi();

    Console.WriteLine("反射一个类型不会触发它的静态构造函数");
    var barType = typeof(Bar);

    Console.WriteLine("但是，你可以通过
System.Runtime.CompilerServices.RuntimeHelpers手动触发它");
    RuntimeHelpers.RunClassConstructor(barType.TypeHandle);
}

// 在这里定义其他方法和类
public static class Foo
{
    static Foo()
    {
        Console.WriteLine("static Foo.ctor");
    }
    public static void SayHi()
```

Chapter 42: Static Classes

Section 42.1: Static Classes

The "static" keyword when referring to a class has three effects:

1. You **cannot** create an instance of a static class (this even removes the default constructor)
2. All properties and methods in the class **must** be static as well.
3. A **static** class is a **sealed** class, meaning it cannot be inherited.

```
public static class Foo
{
    //Notice there is no constructor as this cannot be an instance
    public static int Counter { get; set; }
    public static int GetCount()
    {
        return Counter;
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Foo.Counter++;
        Console.WriteLine(Foo.GetCount()); //this will print 1

        //var foo1 = new Foo();
        //this line would break the code as the Foo class does not have a constructor
    }
}
```

Section 42.2: Static class lifetime

A **static** class is lazily initialized on member access and lives for the duration of the application domain.

```
void Main()
{
    Console.WriteLine("Static classes are lazily initialized");
    Console.WriteLine("The static constructor is only invoked when the class is first accessed");
    Foo.SayHi();

    Console.WriteLine("Reflecting on a type won't trigger its static .ctor");
    var barType = typeof(Bar);

    Console.WriteLine("However, you can manually trigger it with
System.Runtime.CompilerServices.RuntimeHelpers");
    RuntimeHelpers.RunClassConstructor(barType.TypeHandle);
}

// Define other methods and classes here
public static class Foo
{
    static Foo()
    {
        Console.WriteLine("static Foo.ctor");
    }
    public static void SayHi()
```

```

    {
    Console.WriteLine("Foo: Hi");
    }
}
public static class Bar
{
    static Bar()
    {
        Console.WriteLine("static Bar.ctor");
    }
}

```

第42.3节：static关键字

static关键字有两个含义：

1. 该值不会因对象而异，而是作为整个类的属性变化
2. 静态属性和方法不需要实例。

```

public class Foo
{
    public Foo{
        Counter++;
        NonStaticCounter++;
    }

    public static int Counter { get; set; }
    public int NonStaticCounter { get; set; }
}

public class Program
{
    static void Main(string[] args)
    {
        //创建一个实例
        var foo1 = new Foo();
        Console.WriteLine(foo1.NonStaticCounter); //这将打印 "1"

        //注意下面这次调用不是通过实例访问，而是通过类名调用。
        Console.WriteLine(Foo.Counter); //这也将打印 "1"

        //创建第二个实例
        var foo2 = new Foo();

        Console.WriteLine(foo2.NonStaticCounter); //这将打印 "1"

        Console.WriteLine(Foo.Counter); //这现在将打印 "2"
        // 静态属性在两个实例上都会递增，并且可以在整个类中持续存在
    }
}

```

```

    {
        Console.WriteLine("Foo: Hi");
    }
}
public static class Bar
{
    static Bar()
    {
        Console.WriteLine("static Bar.ctor");
    }
}

```

Section 42.3: Static keyword

The static keyword means 2 things:

1. This value does not change from object to object but rather changes on a class as a whole
2. Static properties and methods don't require an instance.

```

public class Foo
{
    public Foo{
        Counter++;
        NonStaticCounter++;
    }

    public static int Counter { get; set; }
    public int NonStaticCounter { get; set; }
}

public class Program
{
    static void Main(string[] args)
    {
        //Create an instance
        var foo1 = new Foo();
        Console.WriteLine(foo1.NonStaticCounter); //this will print "1"

        //Notice this next call doesn't access the instance but calls by the class name.
        Console.WriteLine(Foo.Counter); //this will also print "1"

        //Create a second instance
        var foo2 = new Foo();

        Console.WriteLine(foo2.NonStaticCounter); //this will print "1"

        Console.WriteLine(Foo.Counter); //this will now print "2"
        //The static property incremented on both instances and can persist for the whole class
    }
}

```

第43章：单例模式实现

第43.1节：静态初始化单例

```
public class 单例
{
    private readonly static 单例 instance = new 单例();
    private 单例() { }
    public static 单例 Instance => instance;
}
```

该实现是线程安全的，因为在这种情况下`instance`对象是在静态构造函数中初始化的。CLR已经确保所有静态构造函数的执行是线程安全的。

修改`instance`不是线程安全的操作，因此`readonly`属性保证了初始化后的不可变性。

第43.2节：延迟加载的线程安全单例（使用Lazy<T>）

.Net 4.0的Lazy类型保证了线程安全的对象初始化，因此该类型可以用来实现单例模式。

```
public class LazySingleton
{
    private static readonly Lazy<LazySingleton> _instance =
        new Lazy<LazySingleton>(() => new LazySingleton());

    public static LazySingleton Instance
    {
        get { return _instance.Value; }
    }

    private LazySingleton() { }
}
```

使用`Lazy<T>`将确保对象仅在调用代码中某处使用时才被实例化。

一个简单的用法如下：

```
using System;

public class Program
{
    public static void Main()
    {
        var instance = LazySingleton.Instance;
    }
}
```

[.NET Fiddle 在线演示](#)

第43.3节：使用双重检查锁定的延迟线程安全单例

这个线程安全的单例版本在早期的.NET版本中是必要的，因为`static`初始化不保证线程安全。在更现代的框架版本中，通常更推荐使用静态初始化的单例，

Chapter 43: Singleton Implementation

Section 43.1: Statically Initialized Singleton

```
public class Singleton
{
    private readonly static Singleton instance = new Singleton();
    private Singleton() { }
    public static Singleton Instance => instance;
}
```

This implementation is thread-safe because in this case `instance` object is initialized in the static constructor. The CLR already ensures that all static constructors are executed thread-safe.

Mutating `instance` is not a thread-safe operation, therefore the `readonly` attribute guarantees immutability after initialization.

Section 43.2: Lazy, thread-safe Singleton (using Lazy<T>)

.Net 4.0 type `Lazy` guarantees thread-safe object initialization, so this type could be used to make Singletons.

```
public class LazySingleton
{
    private static readonly Lazy<LazySingleton> _instance =
        new Lazy<LazySingleton>(() => new LazySingleton());

    public static LazySingleton Instance
    {
        get { return _instance.Value; }
    }

    private LazySingleton() { }
}
```

Using `Lazy<T>` will make sure that the object is only instantiated when it is used somewhere in the calling code.

A simple usage will be like:

```
using System;

public class Program
{
    public static void Main()
    {
        var instance = LazySingleton.Instance;
    }
}
```

[Live Demo on .NET Fiddle](#)

Section 43.3: Lazy, thread-safe Singleton (using Double Checked Locking)

This thread-safe version of a singleton was necessary in the early versions of .NET where `static` initialization was not guaranteed to be thread-safe. In more modern versions of the framework a statically initialized singleton is

因为在以下模式中很容易出现实现错误。

```
public sealed class ThreadSafeSingleton
{
    private static volatile ThreadSafeSingleton 实例;
    private static object lockObject = new Object();

    private ThreadSafeSingleton()
    {}

    public static ThreadSafeSingleton 实例
    {
        get
        {
            if (instance == null)
            {
                lock (lockObject)
                {
                    if (instance == null)
                    {
                        instance = new ThreadSafeSingleton();
                    }
                }
            }
            return instance;
        }
    }
}
```

注意这里对 `if (instance == null)` 的检查做了两次：一次是在获取锁之前，一次是在之后。即使没有第一次的 `null` 检查，这个实现仍然是线程安全的。然而，这意味着每次请求实例时都会获取锁，这会导致性能下降。第一次的 `null` 检查是为了避免不必要的锁。第二次的 `null` 检查确保只有第一个获取锁的线程创建实例，其他线程会发现实例已被创建，从而跳过创建过程。

第43.4节：延迟加载的线程安全单例（适用于.NET 3.5或更早版本，替代实现）

因为在 .NET 3.5 及更早版本中没有 `Lazy<T>` 类，所以你可以使用以下模式：

```
public class 单例
{
    private Singleton() // 防止公共实例化
    {}

    public static Singleton 实例
    {
        get
        {
            return Nested.instance;
        }
    }

    private class Nested
    {
    }
}
```

usually preferred because it is very easy to make implementation mistakes in the following pattern.

```
public sealed class ThreadSafeSingleton
{
    private static volatile ThreadSafeSingleton instance;
    private static object lockObject = new Object();

    private ThreadSafeSingleton()
    {}

    public static ThreadSafeSingleton Instance
    {
        get
        {
            if (instance == null)
            {
                lock (lockObject)
                {
                    if (instance == null)
                    {
                        instance = new ThreadSafeSingleton();
                    }
                }
            }
            return instance;
        }
    }
}
```

Notice that the `if (instance == null)` check is done twice: once before the lock is acquired, and once afterwards. This implementation would still be thread-safe even without the first null check. However, that would mean that a lock would be acquired *every time* the instance is requested, and that would cause performance to suffer. The first null check is added so that the lock is not acquired unless it's necessary. The second null check makes sure that only the first thread to acquire the lock then creates the instance. The other threads will find the instance to be populated and skip ahead.

Section 43.4: Lazy, thread safe singleton (for .NET 3.5 or older, alternate implementation)

Because in .NET 3.5 and older you don't have `Lazy<T>` class you use the following pattern:

```
public class Singleton
{
    private Singleton() // prevents public instantiation
    {}

    public static Singleton Instance
    {
        get
        {
            return Nested.instance;
        }
    }

    private class Nested
    {
    }
}
```

```
// 显式静态构造函数，告诉 C# 编译器  
// 不要将类型标记为 beforefieldinit  
static Nested()  
{  
}  
  
internal static readonly Singleton instance = new Singleton();  
}
```

此方法灵感来源于 [Jon Skeet 的博客文章](#)。

因为Nested类是嵌套且私有的，单例实例的创建不会因为访问Singleton类的其他成员（例如公共只读属性）而被触发。

```
// Explicit static constructor to tell C# compiler  
// not to mark type as beforefieldinit  
static Nested()  
{  
}  
  
internal static readonly Singleton instance = new Singleton();  
}
```

This is inspired from [Jon Skeet's blog post](#).

Because the Nested class is nested and private the instantiation of the singleton instance will not be triggered by accessing other members of the Singleton class (such as a public readonly property, for example).

第44章：依赖注入

第44.1节：使用Unity的C#和ASP.NET依赖注入

首先，为什么我们应该在代码中使用依赖注入？我们希望将程序中其他组件与其他类解耦。例如，我们有一个AnimalController类，代码如下：

```
public class AnimalController()
{
    private SantaAndHisReindeer _SantaAndHisReindeer = new SantaAndHisReindeer();

    public AnimalController()
    {
        Console.WriteLine("");
    }
}
```

我们看这段代码，觉得一切正常，但现在我们的AnimalController依赖于对象_SantaAndHisReindeer。这样我的Controller在测试和代码复用方面会非常困难。

非常好的解释了为什么我们应该使用依赖注入和接口there。

如果我们想让Unity处理依赖注入，实现的路径非常简单 :) 通过NuGet（包管理器），我们可以轻松地将Unity导入到我们的代码中。

在 Visual Studio 工具 -> NuGet 包管理器 -> 管理解决方案的包 -> 在搜索框中输入 unity -> 选择我们的项目 -> 点击安装

现在将创建两个带有良好注释的文件。

在 App-Data 文件夹中 UnityConfig.cs 和 UnityMvcActivator.cs

UnityConfig - 在 RegisterTypes 方法中，我们可以看到将注入到构造函数中的类型。

```
namespace Vegan.WebUi.App_Start
{
    public class UnityConfig
    {
        #region Unity 容器
        private static Lazy<IUnityContainer> container = new Lazy<IUnityContainer>(() =>
        {
            var container = new UnityContainer();
            RegisterTypes(container);
            return container;
        });

        /// <summary>
        /// 获取已配置的 Unity 容器。
        /// </summary>
        public static IUnityContainer 获取已配置的容器()
        {
            return container.Value;
        }
    }
}
```

Chapter 44: Dependency Injection

Section 44.1: Dependency Injection C# and ASP.NET with Unity

First why we should use dependency injection in our code ? We want to decouple other components from other classes in our program. For example we have class AnimalController which have code like this :

```
public class AnimalController()
{
    private SantaAndHisReindeer _SantaAndHisReindeer = new SantaAndHisReindeer();

    public AnimalController()
    {
        Console.WriteLine("");
    }
}
```

We look at this code and we think everything is ok but now our AnimalController is reliant on object _SantaAndHisReindeer. Automatically my Controller is bad to testing and reusability of my code will be very hard.

Very good explanation why we should use Dependency Injection and interfaces [here](#).

If we want Unity to handle DI, the road to achieve this is very simple :) With NuGet(package manager) we can easily import unity to our code.

in Visual Studio Tools -> NuGet Package Manager -> Manage Packages for Solution -> in search input write unity -> choose our project-> click install

Now two files with nice comments will be created.

in App-Data folder UnityConfig.cs and UnityMvcActivator.cs

UnityConfig - in RegisterTypes method, we can see type that will be injection in our constructors.

```
namespace Vegan.WebUi.App_Start
{
    public class UnityConfig
    {
        #region Unity Container
        private static Lazy<IUnityContainer> container = new Lazy<IUnityContainer>(() =>
        {
            var container = new UnityContainer();
            RegisterTypes(container);
            return container;
        });

        /// <summary>
        /// Gets the configured Unity container.
        /// </summary>
        public static IUnityContainer GetConfiguredContainer()
        {
            return container.Value;
        }
    }
}
```

```

}
#endregion

/// <summary>在 Unity 容器中注册类型映射。</summary>
/// <param name="container">要配置的 Unity 容器。</param>
/// <remarks>无需注册具体类型，如控制器或 API 控制器
(除非你想
    /// 更改默认设置)，因为即使未事先注册，Unity 也允许解析具体类型。</remarks>

public static void 注册类型(IUnityContainer container)
{
    // 注意：要从 web.config 加载，请取消注释下面的行。确保在 using 语句中添加
Microsoft.Practices.Unity.Configuration.
    // container.LoadConfiguration();

    // TODO: 在此注册你的类型
    // container.RegisterType<IProductRepository, ProductRepository>();

container.RegisterType<ISanta, SantaAndHisReindeer>();

}

}

```

UnityMvcActivator - > 还有很好的注释说明该类将 Unity 与 ASP.NET

MVC 集成

```

using System.Linq;
using System.Web.Mvc;
using Microsoft.Practices.Unity.Mvc;

[assembly:
WebActivatorEx.PreApplicationStartMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator), "Start")]
[assembly:
WebActivatorEx.ApplicationShutdownMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator),
"Shutdown")]

namespace Vegan.WebUi.App_Start
{
    /// <summary>提供将 Unity 与 ASP.NET MVC 集成的引导程序。</summary>
    public static class UnityWebActivator
    {
        /// <summary>在应用程序启动时集成Unity。</summary>
        public static void Start()
        {
            var container = UnityConfig.GetConfiguredContainer();

            FilterProviders.Providers.Remove(FilterProviders.Providers.OfType<FilterAttributeFilterProvider>().
First());
            FilterProviders.Providers.Add(new UnityFilterAttributeFilterProvider(container));

            DependencyResolver.SetResolver(new UnityDependencyResolver(container));

            // TODO: 如果想使用PerRequestLifetimeManager, 请取消注释
            //
Microsoft.Web.Infrastructure.DynamicModuleHelper.DynamicModuleUtility.RegisterModule(typeof(UnityPerRequestHttpModule));
        }
    }
}

```

```

}
#endregion

/// <summary>Registers the type mappings with the Unity container.</summary>
/// <param name="container">The unity container to configure.</param>
/// <remarks>There is no need to register concrete types such as controllers or API controllers
(unless you want to
    /// change the defaults), as Unity allows resolving a concrete type even if it was not previously
registered.</remarks>
public static void RegisterTypes(IUnityContainer container)
{
    // NOTE: To load from web.config uncomment the line below. Make sure to add a
Microsoft.Practices.Unity.Configuration to the using statements.
    // container.LoadConfiguration();

    // TODO: Register your types here
    // container.RegisterType<IProductRepository, ProductRepository>();

    container.RegisterType<ISanta, SantaAndHisReindeer>();

}
}

```

UnityMvcActivator - > also with nice comments which say that this class integrates Unity with ASP.NET MVC

```

using System.Linq;
using System.Web.Mvc;
using Microsoft.Practices.Unity.Mvc;

[assembly:
WebActivatorEx.PreApplicationStartMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator), "Start")]
[assembly:
WebActivatorEx.ApplicationShutdownMethod(typeof(Vegan.WebUi.App_Start.UnityWebActivator),
"Shutdown")]

namespace Vegan.WebUi.App_Start
{
    /// <summary>Provides the bootstrapping for integrating Unity with ASP.NET MVC.</summary>
    public static class UnityWebActivator
    {
        /// <summary>Integrates Unity when the application starts.</summary>
        public static void Start()
        {
            var container = UnityConfig.GetConfiguredContainer();

            FilterProviders.Providers.Remove(FilterProviders.Providers.OfType<FilterAttributeFilterProvider>().
First());
            FilterProviders.Providers.Add(new UnityFilterAttributeFilterProvider(container));

            DependencyResolver.SetResolver(new UnityDependencyResolver(container));

            // TODO: Uncomment if you want to use PerRequestLifetimeManager
            //
Microsoft.Web.Infrastructure.DynamicModuleHelper.DynamicModuleUtility.RegisterModule(typeof(UnityPerRequestHttpModule));
        }
    }
}

```

```

/// <summary>在应用程序关闭时释放Unity容器。</summary>
public static void Shutdown()
{
    var container = UnityConfig.GetConfiguredContainer();
    container.Dispose();
}

```

在运行我们的应用程序之前，还有最后一件事必须做。

在Global.asax.cs中，我们必须添加新行：UnityWebActivator.Start()，它将启动、配置Unity并注册我们的类型。

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using Vegan.WebUi.App_Start;

namespace Vegan.WebUi
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
BundleConfig.RegisterBundles(BundleTable.Bundles);
            UnityWebActivator.Start();
        }
    }
}

```

第44.2节：使用MEF的依赖注入

```

public interface ILogger
{
    void Log(string message);
}

[Export(typeof(ILogger))]

```

```

/// <summary>Disposes the Unity container when the application is shut down.</summary>
public static void Shutdown()
{
    var container = UnityConfig.GetConfiguredContainer();
    container.Dispose();
}

```

Now we can decouple our Controller from class SantAndHisReindeer :)

```

public class AnimalController()
{
    private readonly SantaAndHisReindeer _SantaAndHisReindeer;

    public AnimalController(SantaAndHisReindeer SantaAndHisReindeer)
    {
        _SantaAndHisReindeer = SantaAndHisReindeer;
    }
}

```

There is one final thing we must do before running our application.

In Global.asax.cs we must add new line: UnityWebActivator.Start() which will start, configure Unity and register our types.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using Vegan.WebUi.App_Start;

namespace Vegan.WebUi
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
            UnityWebActivator.Start();
        }
    }
}

```

Section 44.2: Dependency injection using MEF

```

public interface ILogger
{
    void Log(string message);
}

[Export(typeof(ILogger))]

```

```

[ExportMetadata("Name", "Console")]
public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}

[Export(typeof(ILogger))]
[ExportMetadata("Name", "File")]
public class FileLogger : ILogger
{
    public void Log(string message)
    {
        //将消息写入文件
    }
}

public class User
{
    private readonly ILogger logger;
    public User(ILogger logger)
    {
        this.logger = logger;
    }
    public void LogUser(string message)
    {
        logger.Log(message) ;
    }
}

public interface ILoggerMetaData
{
    string Name { get; }
}

internal class Program
{
    private CompositionContainer _container;

    [ImportMany]
    private IEnumerable<Lazy<ILogger, ILoggerMetaData>> _loggers;

    private static void Main()
    {
        组合日志记录器();
        Lazy<ILogger, ILoggerMetaData> loggerNameAndLoggerMapping = _loggers.First((n) =>
        ((n.Metadata.Name.ToUpper() == "Console")));
        ILogger logger= loggerNameAndLoggerMapping.Value
        var user = new User(logger);
        user.LogUser("user name");
    }

    private void 组合日志记录器()
    {
        //一个组合多个目录的聚合目录
        var 目录 = new AggregateCatalog();
        string loggersDll目录 = Path.Combine(Utilities.GetApplicationDirectory(), "Loggers");
        if (!Directory.Exists(loggersDll目录 ))
        {
            Directory.CreateDirectory(loggersDll目录 );
        }
    }
}

```

```

[ExportMetadata("Name", "Console")]
public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}

[Export(typeof(ILogger))]
[ExportMetadata("Name", "File")]
public class FileLogger : ILogger
{
    public void Log(string message)
    {
        //Write the message to file
    }
}

public class User
{
    private readonly ILogger logger;
    public User(ILogger logger)
    {
        this.logger = logger;
    }
    public void LogUser(string message)
    {
        logger.Log(message) ;
    }
}

public interface ILoggerMetaData
{
    string Name { get; }
}

internal class Program
{
    private CompositionContainer _container;

    [ImportMany]
    private IEnumerable<Lazy<ILogger, ILoggerMetaData>> _loggers;

    private static void Main()
    {
        ComposeLoggers();
        Lazy<ILogger, ILoggerMetaData> loggerNameAndLoggerMapping = _loggers.First((n) =>
        ((n.Metadata.Name.ToUpper() == "Console")));
        ILogger logger= loggerNameAndLoggerMapping.Value
        var user = new User(logger);
        user.LogUser("user name");
    }

    private void ComposeLoggers()
    {
        //An aggregate catalog that combines multiple catalogs
        var catalog = new AggregateCatalog();
        string loggersDllDirectory = Path.Combine(Utilities.GetApplicationDirectory(), "Loggers");
        if (!Directory.Exists(loggersDllDirectory ))
        {
            Directory.CreateDirectory(loggersDllDirectory );
        }
    }
}

```

```

    }

    //添加与PluginManager类位于同一程序集中的所有部件
    catalog.Catalogs.Add(new AssemblyCatalog(typeof(Program).Assembly));
    catalog.Catalogs.Add(new DirectoryCatalog(loggersDllDirectory));

    //使用目录中的部件创建CompositionContainer
    _container = new CompositionContainer(catalog);

    //填充此对象的导入部分
    try
    {
        this._container.ComposeParts(this);
    }
    catch (CompositionException compositionException)
    {
        throw new CompositionException(compositionException.Message);
    }
}

```

```

    }

    //Adds all the parts found in the same assembly as the PluginManager class
    catalog.Catalogs.Add(new AssemblyCatalog(typeof(Program).Assembly));
    catalog.Catalogs.Add(new DirectoryCatalog(loggersDllDirectory));

    //Create the CompositionContainer with the parts in the catalog
    _container = new CompositionContainer(catalog);

    //Fill the imports of this object
    try
    {
        this._container.ComposeParts(this);
    }
    catch (CompositionException compositionException)
    {
        throw new CompositionException(compositionException.Message);
    }
}

```

第45章：部分类和方法

部分类为我们提供了将类拆分为多个部分并放在多个源文件中的选项。所有部分在编译时会合并为一个单一的类。所有部分都应包含关键字partial，且应具有相同的访问权限。所有部分应存在于同一程序集内，才能在编译时被包含。

第45.1节：部分类

部分类提供了将类声明拆分（通常拆分到不同文件）的方法。部分类可以解决的一个常见问题是允许用户修改自动生成的代码，而不必担心代码重新生成时他们的更改会被覆盖。此外，多个开发者可以同时在同一个类或方法上协作开发。

```
using System;

namespace PartialClassAndMethods
{
    public partial class PartialClass
    {
        public void ExampleMethod() {
            Console.WriteLine("来自第一个声明的方法调用。");
        }
    }

    public partial class PartialClass
    {
        public void AnotherExampleMethod()
        {
            Console.WriteLine("来自第二个声明的方法调用。");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            PartialClass partial = new PartialClass();
            partial.ExampleMethod(); // 输出 "来自第一个声明的方法调用。"
            partial.AnotherExampleMethod(); // 输出 "来自第二个声明的方法调用。"
        }
    }
}
```

第45.2节：继承自基类的部分类

当继承自任何基类时，只需在一个部分类中指定基类。

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {}

// PartialClass2.cs
public partial class PartialClass {}
```

您可以在多个部分类中指定相同的基类。某些IDE工具会将其标记为冗余，但它确实可以正确编译。

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {}
```

Chapter 45: Partial class and methods

Partial classes provides us an option to split classes into multiple parts and in multiple source files. All parts are combined into one single class during compile time. All parts should contain the keyword `partial`, should be of the same accessibility. All parts should be present in the same assembly for it to be included during compile time.

Section 45.1: Partial classes

Partial classes provide an ability to split class declaration (usually into separate files). A common problem that can be solved with partial classes is allowing users to modify auto-generated code without fearing that their changes will be overwritten if the code is regenerated. Also multiple developers can work on same class or methods.

```
using System;

namespace PartialClassAndMethods
{
    public partial class PartialClass
    {
        public void ExampleMethod() {
            Console.WriteLine("Method call from the first declaration.");
        }
    }

    public partial class PartialClass
    {
        public void AnotherExampleMethod()
        {
            Console.WriteLine("Method call from the second declaration.");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            PartialClass partial = new PartialClass();
            partial.ExampleMethod(); // outputs "Method call from the first declaration."
            partial.AnotherExampleMethod(); // outputs "Method call from the second declaration."
        }
    }
}
```

Section 45.2: Partial classes inheriting from a base class

When inheriting from any base class, only one partial class needs to have the base class specified.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {}

// PartialClass2.cs
public partial class PartialClass {}
```

You *can* specify the *same* base class in more than one partial class. It will get flagged as redundant by some IDE tools, but it does compile correctly.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {}
```

```
// PartialClass2.cs
public partial class PartialClass : BaseClass {} // 这里的基类是冗余的
```

您不能在多个部分类中指定不同的基类，这将导致编译错误。

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {} // 编译错误
```

```
// PartialClass2.cs
public partial class PartialClass : OtherBaseClass {} // 编译错误
```

第45.3节：部分方法

部分方法由一个部分类声明中的定义组成（常见场景是在自动生成的部分中），而实现则在另一个部分类声明中完成。

```
using System;

namespace PartialClassAndMethods
{
    public partial class PartialClass // 自动生成的
    {
        partial void PartialMethod();
    }

    public partial class PartialClass // 人工编写的
    {
        public void PartialMethod()
        {
            Console.WriteLine("调用了部分方法。");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
PartialClass partial = new PartialClass();
            partial.PartialMethod(); // 输出 "调用了部分方法。"
        }
    }
}
```

```
// PartialClass2.cs
public partial class PartialClass : BaseClass {} // base class here is redundant
```

You cannot specify different base classes in multiple partial classes, it will result in a compiler error.

```
// PartialClass1.cs
public partial class PartialClass : BaseClass {} // compiler error
```

```
// PartialClass2.cs
public partial class PartialClass : OtherBaseClass {} // compiler error
```

Section 45.3: Partial methods

Partial method consists of the definition in one partial class declaration (as a common scenario - in the auto-generated one) and the implementation in another partial class declaration.

```
using System;

namespace PartialClassAndMethods
{
    public partial class PartialClass // Auto-generated
    {
        partial void PartialMethod();
    }

    public partial class PartialClass // Human-written
    {
        public void PartialMethod()
        {
            Console.WriteLine("Partial method called.");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
PartialClass partial = new PartialClass();
            partial.PartialMethod(); // outputs "Partial method called."
        }
    }
}
```

第46章：对象初始化器

第46.1节：简单用法

当你需要创建一个对象并立即设置几个属性，但现有的构造函数不足以满足需求时，对象初始化器非常方便。假设你有一个类

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }

    // 类定义的其余部分
}
```

使用初始化器初始化类的新实例：

```
Book theBook = new Book { Title = "堂吉诃德", Author = "米格尔·德·塞万提斯" };
```

这等同于

```
Book theBook = new Book();
theBook.Title = "堂吉诃德";
theBook.Author = "米格尔·德·塞万提斯";
```

第46.2节：带非默认构造函数的用法

如果需要，可以将对象初始化器与构造函数结合使用来初始化类型。例如，定义如下类：

```
public class Book {
    public string Title { get; set; }
    public string Author { get; set; }

    public Book(int id) {
        // 执行操作
    }

    // 类定义的其余部分
}

var someBook = new Book(16) { Title = "堂吉诃德", Author = "米格尔·德·塞万提斯" }
```

这将首先使用Book(int)构造函数实例化一个Book对象，然后在初始化器中设置每个属性。它等同于：

```
var someBook = new Book();
someBook.Title = "堂吉诃德";
someBook.Author = "米格尔·德·塞万提斯";
```

第46.3节：匿名类型的使用

对象初始化器是初始化匿名类型的唯一方式，匿名类型是由编译器生成的类型。

```
var album = new { Band = "披头士", Title = "艾比路" };
```

Chapter 46: Object initializers

Section 46.1: Simple usage

Object initializers are handy when you need to create an object and set a couple of properties right away, but the available constructors are not sufficient. Say you have a class

```
public class Book
{
    public string Title { get; set; }
    public string Author { get; set; }

    // the rest of class definition
}
```

To initialize a new instance of the class with an initializer:

```
Book theBook = new Book { Title = "Don Quixote", Author = "Miguel de Cervantes" };
```

This is equivalent to

```
Book theBook = new Book();
theBook.Title = "Don Quixote";
theBook.Author = "Miguel de Cervantes";
```

Section 46.2: Usage with non-default constructors

You can combine object initializers with constructors to initialize types if necessary. Take for example a class defined as such:

```
public class Book {
    public string Title { get; set; }
    public string Author { get; set; }

    public Book(int id) {
        // do things
    }

    // the rest of class definition
}

var someBook = new Book(16) { Title = "Don Quixote", Author = "Miguel de Cervantes" }
```

This will first instantiate a Book with the Book(int) constructor, then set each property in the initializer. It is equivalent to:

```
var someBook = new Book();
someBook.Title = "Don Quixote";
someBook.Author = "Miguel de Cervantes";
```

Section 46.3: Usage with anonymous types

Object initializers are the only way to initialize anonymous types, which are types generated by the compiler.

```
var album = new { Band = "Beatles", Title = "Abbey Road" };
```

因此，对象初始化器在LINQ的select查询中被广泛使用，因为它们提供了一种方便的方式来指定你感兴趣的查询对象的部分内容。

```
var albumTitles = from a in albums
                  select new
                  {
                      标题 = a.标题,
                      艺术家 = a.乐队
                  };
```

For that reason object initializers are widely used in LINQ select queries, since they provide a convenient way to specify which parts of a queried object you are interested in.

```
var albumTitles = from a in albums
                  select new
                  {
                      Title = a.Title,
                      Artist = a.Band
                  };
```

第47章：方法

第47.1节：调用方法

调用静态方法：

```
// 单个参数  
System.Console.WriteLine("Hello World");  
  
// 多个参数  
string name = "User";  
System.Console.WriteLine("Hello, {0}!", name);
```

调用静态方法并存储其返回值：

```
string input = System.Console.ReadLine();
```

调用实例方法：

```
int x = 42;  
// 这里调用的实例方法是 Int32.ToString()  
string xAsString = x.ToString();
```

调用泛型方法

```
// 假设有一个方法 'T[] CreateArray<T>(int size)'  
DateTime[] dates = CreateArray<DateTime>(8);
```

第47.2节：匿名方法

匿名方法提供了一种将代码块作为委托参数传递的技术。它们是有主体但没有名称的方法。

```
delegate int IntOp(int lhs, int rhs);  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        // C# 2.0 定义  
        IntOp add = delegate(int lhs, int rhs)  
        {  
            return lhs + rhs;  
        };  
  
        // C# 3.0 定义  
        IntOp mul = (lhs, rhs) =>  
        {  
            return lhs * rhs;  
        };  
  
        // C# 3.0 定义 - 简写  
        IntOp sub = (lhs, rhs) => lhs - rhs;  
  
        // 调用每个方法  
        Console.WriteLine("2 + 3 = " + add(2, 3));  
        Console.WriteLine("2 * 3 = " + mul(2, 3));
```

Chapter 47: Methods

Section 47.1: Calling a Method

Calling a static method:

```
// Single argument  
System.Console.WriteLine("Hello World");  
  
// Multiple arguments  
string name = "User";  
System.Console.WriteLine("Hello, {0}!", name);
```

Calling a static method and storing its return value:

```
string input = System.Console.ReadLine();
```

Calling an instance method:

```
int x = 42;  
// The instance method called here is Int32.ToString()  
string xAsString = x.ToString();
```

Calling a generic method

```
// Assuming a method 'T[] CreateArray<T>(int size)'  
DateTime[] dates = CreateArray<DateTime>(8);
```

Section 47.2: Anonymous method

Anonymous methods provide a technique to pass a code block as a delegate parameter. They are methods with a body, but no name.

```
delegate int IntOp(int lhs, int rhs);  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        // C# 2.0 definition  
        IntOp add = delegate(int lhs, int rhs)  
        {  
            return lhs + rhs;  
        };  
  
        // C# 3.0 definition  
        IntOp mul = (lhs, rhs) =>  
        {  
            return lhs * rhs;  
        };  
  
        // C# 3.0 definition - shorthand  
        IntOp sub = (lhs, rhs) => lhs - rhs;  
  
        // Calling each method  
        Console.WriteLine("2 + 3 = " + add(2, 3));  
        Console.WriteLine("2 * 3 = " + mul(2, 3));
```

```
Console.WriteLine("2 - 3 = " + sub(2, 3));
}
}
```

第47.3节：声明方法

每个方法都有一个唯一的签名，由访问修饰符（public、private等）、可选修饰符（abstract）、方法名以及（如有需要）方法参数组成。注意，返回类型不属于签名的一部分。方法原型如下所示：

```
访问修饰符 可选修饰符 返回类型 方法名(输入参数)
{
    //方法体
}
```

访问修饰符可以是public、protected、private，或者默认的internal。

可选修饰符可以是static abstract virtual override new或sealed。

返回类型可以是void表示无返回值，也可以是从基本类型如int到复杂类的任意类型。

方法可以有一些输入参数，也可以没有。要为方法设置参数，应像声明普通变量一样声明每个参数（如int a），多个参数之间用逗号分隔（如int a, int b）。

参数可以有默认值。为此，应为参数设置一个值（如int a = 0）。如果参数有默认值，传入该参数值是可选的。

下面的方法示例返回两个整数的和：

```
private int Sum(int a, int b)
{
    return a + b;
}
```

第47.4节：参数和实参

一个方法可以声明任意数量的参数（在此示例中，i、s和o是参数）：

```
static void DoSomething(int i, string s, object o) {
    Console.WriteLine(String.Format("i={0}, s={1}, o={2}", i, s, o));
}
```

参数可以用来向方法传递值，以便方法可以使用它们。这可以是各种工作，比如打印这些值，或者修改参数所引用的对象，或者存储这些值。

当你调用方法时，需要为每个参数传递一个实际的值。此时，你实际传递给方法调用的值称为实参：

```
DoSomething(x, "hello", new object());
```

第47.5节：返回类型

一个方法可以返回空（void），或者返回指定类型的值：

```
Console.WriteLine("2 - 3 = " + sub(2, 3));
}
}
```

Section 47.3: Declaring a Method

Every method has a unique signature consisting of a accessor (public, private, ...), optional modifier (abstract), a name and if needed method parameters. Note, that the return type is not part of the signature. A method prototype looks like the following:

```
AccessModifier OptionalModifier ReturnType MethodName(InputParameters)
{
    //Method body
}
```

AccessModifier can be public, protected, private or by default internal.

OptionalModifier can be static abstract virtual override new or sealed.

ReturnType can be void for no return or can be any type from the basic ones, as int to complex classes.

A Method may have some or no input parameters. To set parameters for a method, you should declare each one like normal variable declarations (like int a), and for more than one parameter you should use comma between them (like int a, int b).

Parameters may have default values. For this you should set a value for the parameter (like int a = 0). If a parameter has a default value, setting the input value is optional.

The following method example returns the sum of two integers:

```
private int Sum(int a, int b)
{
    return a + b;
}
```

Section 47.4: Parameters and Arguments

A method can declare any number of parameters (in this example, i, s and o are the parameters):

```
static void DoSomething(int i, string s, object o) {
    Console.WriteLine(String.Format("i={0}, s={1}, o={2}", i, s, o));
}
```

Parameters can be used to pass values into a method, so that the method can work with them. This can be every kind of work like printing the values, or making modifications to the object referenced by a parameter, or storing the values.

When you call the method, you need to pass an actual value for every parameter. At this point, the values that you actually pass to the method call are called Arguments:

```
DoSomething(x, "hello", new object());
```

Section 47.5: Return Types

A method can return either nothing (void), or a value of a specified type:

```
// 如果你不返回值，使用 void 作为返回类型。
static void ReturnsNothing() {
    Console.WriteLine("Returns nothing");
}

// 如果你想返回一个值，需要指定它的类型。
static string ReturnsHelloWorld() {
    return "Hello World";
}
```

如果你的方法指定了返回值，方法必须返回一个值。你可以使用return语句来实现。
一旦执行到return语句，它会返回指定的值，之后的任何代码都不会再执行
(例外是finally块，它们会在方法返回前仍然被执行)。

如果你的方法不返回任何值 (void)，你仍然可以使用不带值的return语句来立即从方法返回。
不过在这种方法的末尾，return语句是多余的。

有效的return语句示例：

```
return;
return 0;
return x * 2;
return Console.ReadLine();
```

抛出异常可能会导致方法执行结束且不返回值。此外，还有迭代器块，其中通过使用 yield 关键字生成返回值，但这些是特殊情况，此处不会进行解释。

第47.6节：默认参数

如果您想提供省略参数的选项，可以使用默认参数：

```
static void SaySomething(string what = "ehh") {
    Console.WriteLine(what);
}

static void Main() {
    // 输出 "hello"
    SaySomething("hello");
    // 输出 "ehh"
    SaySomething(); // 编译器会将此编译为 SaySomething("ehh")
}
```

当您调用这样的方法并省略了提供默认值的参数时，编译器会为您插入该默认值。

请记住，带有默认值的参数需要写在没有默认值的参数之后。

```
static void SaySomething(string say, string what = "ehh") {
    // 正确
    Console.WriteLine(say + what);
}

static void SaySomethingElse(string what = "ehh", string say) {
    // 错误
    Console.WriteLine(say + what);
}
```

```
// If you don't want to return a value, use void as return type.
static void ReturnsNothing() {
    Console.WriteLine("Returns nothing");
}

// If you want to return a value, you need to specify its type.
static string ReturnsHelloWorld() {
    return "Hello World";
}
```

If your method specifies a return value, the method *must* return a value. You do this using the `return` statement.
Once a `return` statement has been reached, it returns the specified value and any code after it will not be run anymore (exceptions are `finally` blocks, which will still be executed before the method returns).

If your method returns nothing (`void`), you can still use the `return` statement without a value if you want to return from the method immediately. At the end of such a method, a `return` statement would be unnecessary though.

Examples of valid `return` statements:

```
return;
return 0;
return x * 2;
return Console.ReadLine();
```

Throwing an exception can end method execution without returning a value. Also, there are iterator blocks, where return values are generated by using the `yield` keyword, but those are special cases that will not be explained at this point.

Section 47.6: Default Parameters

You can use default parameters if you want to provide the option to leave out parameters:

```
static void SaySomething(string what = "ehh") {
    Console.WriteLine(what);
}

static void Main() {
    // prints "hello"
    SaySomething("hello");
    // prints "ehh"
    SaySomething(); // The compiler compiles this as if we had typed SaySomething("ehh")
}
```

When you call such a method and omit a parameter for which a default value is provided, the compiler inserts that default value for you.

Keep in mind that parameters with default values need to be written **after** parameters without default values.

```
static void SaySomething(string say, string what = "ehh") {
    // Correct
    Console.WriteLine(say + what);
}

static void SaySomethingElse(string what = "ehh", string say) {
    // Incorrect
    Console.WriteLine(say + what);
}
```

警告：由于它是这样工作的，默认值在某些情况下可能会有问题。如果你更改了方法参数的默认值，但没有重新编译所有调用该方法的代码，这些调用者仍将使用它们编译时存在的默认值，可能导致不一致。

第47.7节：方法重载

定义：当多个同名方法声明了不同的参数时，称为方法重载。方法重载通常表示功能相同但接受不同数据类型参数的函数。

影响因素

- 参数数量
- 参数类型
- 返回类型**

考虑一个名为Area的方法，它将执行计算功能，接受各种参数并返回结果。

示例

```
public string Area(int value1)
{
    return String.Format("正方形的面积是 {0}", value1 * value1);
}
```

该方法接受一个参数并返回一个字符串，如果我们用一个整数（比如5）调用该方法，输出将是“正方形的面积是 25”。

```
public double Area(double value1, double value2)
{
    return value1 * value2;
}
```

同样，如果我们传递两个双精度值给该方法，输出将是这两个值的乘积，类型为double。该方法既可用于乘法，也可用于计算矩形的面积。

```
public double Area(double value1)
{
    return 3.14 * Math.Pow(value1, 2);
}
```

这可以专门用于计算圆的面积，它接受一个双精度值（半径）并返回另一个双精度值作为其面积。

这些方法中的每一个都可以正常调用且不会冲突——编译器会检查每个方法调用的参数，以确定需要使用哪个版本的Area。

```
string squareArea = Area(2);
double rectangleArea = Area(32.0, 17.5);
double circleArea = Area(5.0); // 这些都是有效的并且可以编译。
```

**注意，仅凭返回类型无法区分两个方法。例如，如果我们有两个参数相同的Area定义，如下所示：

WARNING: Because it works that way, default values can be problematic in some cases. If you change the default value of a method parameter and don't recompile all callers of that method, those callers will still use the default value that was present when they were compiled, possibly causing inconsistencies.

Section 47.7: Method overloading

Definition : When multiple methods with the same name are declared with different parameters, it is referred to as method overloading. Method overloading typically represents functions that are identical in their purpose but are written to accept different data types as their parameters.

Factors affecting

- Number of Arguments
- Type of arguments
- Return Type**

Consider a method named Area that will perform calculation functions, which will accept various arguments and return the result.

Example

```
public string Area(int value1)
{
    return String.Format("Area of Square is {0}", value1 * value1);
}
```

This method will accept one argument and return a string, if we call the method with an integer(say 5) the output will be “Area of Square is 25”.

```
public double Area(double value1, double value2)
{
    return value1 * value2;
}
```

Similarly if we pass two double values to this method the output will be the product of the two values and are of type double. This can be used of multiplication as well as finding the Area of rectangles

```
public double Area(double value1)
{
    return 3.14 * Math.Pow(value1, 2);
}
```

This can be used specially for finding the area of circle, which will accept a double value(radius) and return another double value as its Area.

Each of these methods can be called normally without conflict - the compiler will examine the parameters of each method call to determine which version of Area needs to be used.

```
string squareArea = Area(2);
double rectangleArea = Area(32.0, 17.5);
double circleArea = Area(5.0); // all of these are valid and will compile.
```

**Note that return type alone cannot differentiate between two methods. For instance, if we had two definitions for Area that had the same parameters, like so:

```
public string Area(double width, double height) { ... }
public double Area(double width, double height) { ... }
// 这将无法编译。
```

如果我们需要让类使用相同的方法名但返回不同的值，可以通过实现接口并显式定义其用法来消除歧义问题。

```
public interface IAreaCalculatorString {
    public string Area(double width, double height);
}

public class AreaCalculator : IAreaCalculatorString {
    public string IAreaCalculatorString.Area(double width, double height) { ... }
    // 注意该方法调用现在明确表示当通过
    // IAreaCalculatorString 接口调用时使用，这使我们能够解决歧义。
    public double Area(double width, double height) { ... }
}
```

第47.8节：访问权限

```
// static: 即使未创建类的实例，也可以在类上调用
public static void MyMethod()
```

```
// virtual: 可以在继承类中调用或重写
public virtual void MyMethod()
```

```
// internal: 访问限制在当前程序集内
internal void MyMethod()
```

```
// private: 访问权限仅限于同一类内部
private void MyMethod()
```

```
// public: 所有类/程序集均可访问权限
public void MyMethod()
```

```
// protected: 访问权限限于包含类或从其派生的类型
protected void MyMethod()
```

```
// protected internal: 访问权限限于当前程序集或包含类派生的类型
protected internal void MyMethod()
```

```
public string Area(double width, double height) { ... }
public double Area(double width, double height) { ... }
// This will NOT compile.
```

If we need to have our class use the same method names that return different values, we can remove the issues of ambiguity by implementing an interface and explicitly defining its usage.

```
public interface IAreaCalculatorString {
    public string Area(double width, double height);
}

public class AreaCalculator : IAreaCalculatorString {
    public string IAreaCalculatorString.Area(double width, double height) { ... }
    // Note that the method call now explicitly says it will be used when called through
    // the IAreaCalculatorString interface, allowing us to resolve the ambiguity.
    public double Area(double width, double height) { ... }
}
```

Section 47.8: Access rights

```
// static: is callable on a class even when no instance of the class has been created
public static void MyMethod()
```

```
// virtual: can be called or overridden in an inherited class
public virtual void MyMethod()
```

```
// internal: access is limited within the current assembly
internal void MyMethod()
```

```
// private: access is limited only within the same class
private void MyMethod()
```

```
// public: access right from every class / assembly
public void MyMethod()
```

```
// protected: access is limited to the containing class or types derived from it
protected void MyMethod()
```

```
// protected internal: access is limited to the current assembly or types derived from the containing
// class.
protected internal void MyMethod()
```

第48章：扩展方法

参数	详细信息
这个	扩展方法的第一个参数应始终以this关键字开头，随后是用于引用所扩展对象“当前”实例的标识符

第48.1节：扩展方法 - 概述

扩展方法是在C# 3.0中引入的。扩展方法扩展并添加现有类型的行为，而无需创建新的派生类型、重新编译或以其他方式修改原始类型。当你无法修改想要增强的类型的源代码时，它们尤其有用。扩展方法可以为系统类型、第三方定义的类型以及你自己定义的类型创建。扩展方法可以像原始类型的成员方法一样调用。这允许使用方法链来实现流畅接口。

扩展方法是通过向一个与被扩展的原始类型不同的静态类添加静态方法来创建的。承载扩展方法的静态类通常是专门为承载扩展方法而创建的。

扩展方法的第一个参数是一个特殊参数，用于指定被扩展的原始类型。这个第一个参数带有关键字this（这是C#中this的特殊且独特用法——应理解为不同于允许引用当前对象实例成员的this用法）。

在下面的示例中，被扩展的原始类型是类string。通过方法Shorten()扩展了String，提供了缩短字符串的附加功能。静态类StringExtensions被创建来承载扩展方法。扩展方法Shorten()通过特别标记的第一个参数表明它是对string的扩展。为了表明Shorten()方法扩展了string，第一个参数被标记为this。因此，第一个参数的完整签名是this string text，其中string是被扩展的原始类型，text是所选的参数名。

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}

class Program
{
    static void Main()
    {
        // 这调用了方法 String.ToUpper()
        var myString = "Hello World!".ToUpper();

        // 这调用了扩展方法 StringExtensions.Shorten()
        var newString = myString.Shorten(5);

        // 值得注意的是，上述调用纯粹是语法糖
        // 下面的赋值在功能上是等价的
        var newString2 = StringExtensions.Shorten(myString, 5);
    }
}
```

[.NET Fiddle 在线演示](#)

Chapter 48: Extension Methods

Parameter	Details
this	The first parameter of an extension method should always be preceded by the <code>this</code> keyword, followed by the identifier with which to refer to the "current" instance of the object you are extending

Section 48.1: Extension methods - overview

Extension methods were introduced in C# 3.0. Extension methods extend and add behavior to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. *They are especially helpful when you cannot modify the source of a type you are looking to enhance.* Extension methods may be created for system types, types defined by third parties, and types that you have defined yourself. The extension method can be invoked as though it were a member method of the original type. This allows for **Method Chaining** used to implement a **Fluent Interface**.

An extension method is created by adding a **static method** to a **static class** which is distinct from the original type being extended. The static class holding the extension method is often created for the sole purpose of holding extension methods.

Extension methods take a special first parameter that designates the original type being extended. This first parameter is decorated with the keyword `this` (which constitutes a special and distinct use of `this` in C#—it should be understood as different from the use of `this` which allows referring to members of the current object instance).

In the following example, the original type being extended is the class `string`. `String` has been extended by a method `Shorten()`, which provides the additional functionality of shortening. The static class `StringExtensions` has been created to hold the extension method. The extension method `Shorten()` shows that it is an extension of `string` via the specially marked first parameter. To show that the `Shorten()` method extends `string`, the first parameter is marked with `this`. Therefore, the full signature of the first parameter is `this string text`, where `string` is the original type being extended and `text` is the chosen parameter name.

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}

class Program
{
    static void Main()
    {
        // This calls method String.ToUpper()
        var myString = "Hello World!".ToUpper();

        // This calls the extension method StringExtensions.Shorten()
        var newString = myString.Shorten(5);

        // It is worth noting that the above call is purely syntactic sugar
        // and the assignment below is functionally equivalent
        var newString2 = StringExtensions.Shorten(myString, 5);
    }
}
```

[Live Demo on .NET Fiddle](#)

作为扩展方法第一个参数传入的对象（伴随有this关键字）是调用该扩展方法的实例。

例如，当执行以下代码时：

```
"some string".Shorten(5);
```

参数的值如下：

```
text: "some string"  
length: 5
```

请注意，扩展方法只有在与其定义位于同一命名空间中、代码显式导入该命名空间，或扩展类无命名空间时才可使用。.NET框架指南建议将扩展类放在它们自己的命名空间中。然而，这可能导致发现问题。

这就导致扩展方法与所使用的库之间不会发生冲突，除非显式引入可能冲突的命名空间。例如 LINQ 扩展：

```
using System.Linq; // 允许使用 System.Linq 命名空间中的扩展方法  
  
class Program  
{  
    static void Main()  
    {  
        var ints = new int[] {1, 2, 3, 4};  
  
        // 调用来自 System.Linq 命名空间的 Where() 扩展方法  
        var even = ints.Where(x => x % 2 == 0);  
    }  
}
```

[.NET Fiddle 在线演示](#)

自 C# 6.0 起，也可以对包含扩展方法的class使用using static指令。例如，using static System.Linq.Enumerable;。这样可以使该特定类中的扩展方法可用，而无需将同一命名空间中的其他类型引入作用域。

当存在具有相同签名的类方法时，编译器会优先调用该类方法而非扩展方法。例如：

```
class Test  
{  
    public void Hello()  
    {  
        Console.WriteLine("From Test");  
    }  
}  
  
static class TestExtensions  
{  
    public static void Hello(this Test test)  
    {  
        Console.WriteLine("From extension method");  
    }  
}
```

The object passed as the *first argument of an extension method* (which is accompanied by the `this` keyword) is the instance the extension method is called upon.

For example, when this code is executed:

```
"some string".Shorten(5);
```

The values of the arguments are as below:

```
text: "some string"  
length: 5
```

Note that extension methods are only usable if they are in the same namespace as their definition, if the namespace is imported explicitly by the code using the extension method, or if the extension class is namespace-less. The .NET framework guidelines recommend putting extension classes in their own namespace. However, this may lead to discovery issues.

This results in no conflicts between the extension methods and the libraries being used, unless namespaces which might conflict are explicitly pulled in. For example LINQ Extensions:

```
using System.Linq; // Allows use of extension methods from the System.Linq namespace  
  
class Program  
{  
    static void Main()  
    {  
        var ints = new int[] {1, 2, 3, 4};  
  
        // Call Where() extension method from the System.Linq namespace  
        var even = ints.Where(x => x % 2 == 0);  
    }  
}
```

[Live Demo on .NET Fiddle](#)

Since C# 6.0, it is also possible to put a `using static` directive to the `class` containing the extension methods. For example, `using static System.Linq.Enumerable;`. This makes extension methods from that particular class available without bringing other types from the same namespace into scope.

When a class method with the same signature is available, the compiler prioritizes it over the extension method call. For example:

```
class Test  
{  
    public void Hello()  
    {  
        Console.WriteLine("From Test");  
    }  
}  
  
static class TestExtensions  
{  
    public static void Hello(this Test test)  
    {  
        Console.WriteLine("From extension method");  
    }  
}
```

```

class Program
{
    static void Main()
    {
        Test t = new Test();
        t.Hello(); // 输出 "From Test"
    }
}

```

[.NET Fiddle 在线演示](#)

请注意，如果存在两个签名相同的扩展函数，其中一个在相同的命名空间中，那么该函数将被优先使用。另一方面，如果两个函数都通过using访问，则会出现编译时错误，错误信息为：

调用在以下方法或属性之间存在歧义

请注意，通过originalTypeInstance.ExtensionMethod()语法调用扩展方法的便利性是可选的。该方法也可以用传统方式调用，这样特殊的第一个参数将作为方法的参数使用。

即，以下两者均适用：

```

//调用方式如同方法属于字符串——它无缝扩展了字符串
String s = "Hello World";
s.Shorten(5);

//以传统静态方法调用，带两个参数
StringExtensions.Shorten(s, 5);

```

第48.2节：空值检查

扩展方法是静态方法，但表现得像实例方法。然而，与调用实例方法时遇到null引用不同，当扩展方法以null引用调用时，它不会抛出NullReferenceException。这在某些场景下非常有用。

例如，考虑以下静态类：

```

public static class StringExtensions
{
    public static string EmptyIfNull(this string text)
    {
        return text ?? String.Empty;
    }

    public static string NullIfEmpty(this string text)
    {
        return String.Empty == text ? null : text;
    }
}

string nullString = null;
string emptyString = nullString.EmptyIfNull(); // 将返回 ""
string anotherNullString = emptyString.NullIfEmpty(); // 将返回 null

```

[.NET Fiddle 在线演示](#)

```

class Program
{
    static void Main()
    {
        Test t = new Test();
        t.Hello(); // Prints "From Test"
    }
}

```

[Live demo on .NET Fiddle](#)

Note that if there are two extension functions with the same signature, and one of them is in the same namespace, then that one will be prioritized. On the other hand, if both of them are accessed by using, then a compile time error will ensue with the message:

The call is ambiguous between the following methods or properties

Note that the syntactic convenience of calling an extension method via originalTypeInstance.ExtensionMethod() is an optional convenience. The method can also be called in the traditional manner, so that the special first parameter is used as a parameter to the method.

I.e., both of the following work:

```

//Calling as though method belongs to string--it seamlessly extends string
String s = "Hello World";
s.Shorten(5);

//Calling as a traditional static method with two parameters
StringExtensions.Shorten(s, 5);

```

Section 48.2: Null checking

Extension methods are static methods which behave like instance methods. However, unlike what happens when calling an instance method on a null reference, when an extension method is called with a null reference, it does not throw a [NullReferenceException](#). This can be quite useful in some scenarios.

For example, consider the following static class:

```

public static class StringExtensions
{
    public static string EmptyIfNull(this string text)
    {
        return text ?? String.Empty;
    }

    public static string NullIfEmpty(this string text)
    {
        return String.Empty == text ? null : text;
    }
}

string nullString = null;
string emptyString = nullString.EmptyIfNull(); // will return ""
string anotherNullString = emptyString.NullIfEmpty(); // will return null

```

[Live Demo on .NET Fiddle](#)

第48.3节：显式使用扩展方法

扩展方法也可以像普通静态类方法一样使用。这种调用扩展方法的方式更为冗长，但在某些情况下是必要的。

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}
```

用法：

```
var newString = StringExtensions.Shorten("Hello World", 5);
```

何时将扩展方法作为静态方法调用

仍有一些场景需要将扩展方法作为静态方法使用：

- 解决与成员方法的冲突。如果库的新版本引入了具有相同签名的新成员方法，编译器将优先选择成员方法。
- 解决与另一个具有相同签名的扩展方法的冲突。如果两个库包含类似的扩展方法，且两个扩展方法类的命名空间都在同一文件中使用，就可能发生这种情况。
- 将扩展方法作为方法组传递给委托参数。
- 通过反射自行绑定。
- 在Visual Studio的立即窗口中使用扩展方法。

使用 static

如果使用`using static`指令将静态类的静态成员引入全局作用域，则扩展方法会被跳过。示例：

```
using static OurNamespace.StringExtensions; // 指向前面示例中的类

// 正确：扩展方法语法仍然有效。
"Hello World".Shorten(5);
// 正确：静态方法语法仍然有效。
OurNamespace.StringExtensions.Shorten("Hello World", 5);
// 编译时错误：扩展方法不能作为静态方法调用，除非指定类名。
Shorten("Hello World", 5);
```

如果你从`Shorten`方法的第一个参数中移除`this`修饰符，最后一行将能够编译。

第48.4节：泛型扩展方法

就像其他方法一样，扩展方法也可以使用泛型。例如：

```
static class Extensions
{
    public static bool HasMoreThanThreeElements<T>(this IEnumerable<T> enumerable)
    {
        return enumerable.Take(4).Count() > 3;
    }
}
```

Section 48.3: Explicitly using an extension method

Extension methods can also be used like ordinary static class methods. This way of calling an extension method is more verbose, but is necessary in some cases.

```
static class StringExtensions
{
    public static string Shorten(this string text, int length)
    {
        return text.Substring(0, length);
    }
}
```

Usage:

```
var newString = StringExtensions.Shorten("Hello World", 5);
```

When to call extension methods as static methods

There are still scenarios where you would need to use an extension method as a static method:

- Resolving conflict with a member method. This can happen if a new version of a library introduces a new member method with the same signature. In this case, the member method will be preferred by the compiler.
- Resolving conflicts with another extension method with the same signature. This can happen if two libraries include similar extension methods and namespaces of both classes with extension methods are used in the same file.
- Passing extension method as a method group into delegate parameter.
- Doing your own binding through Reflection.
- Using the extension method in the Immediate window in Visual Studio.

Using static

If a `using static` directive is used to bring static members of a static class into global scope, extension methods are skipped. Example:

```
using static OurNamespace.StringExtensions; // refers to class in previous example

// OK: extension method syntax still works.
"Hello World".Shorten(5);
// OK: static method syntax still works.
OurNamespace.StringExtensions.Shorten("Hello World", 5);
// Compile time error: extension methods can't be called as static without specifying class.
Shorten("Hello World", 5);
```

If you remove the `this` modifier from the first argument of the `Shorten` method, the last line will compile.

Section 48.4: Generic Extension Methods

Just like other methods, extension methods can use generics. For example:

```
static class Extensions
{
    public static bool HasMoreThanThreeElements<T>(this IEnumerable<T> enumerable)
    {
        return enumerable.Take(4).Count() > 3;
    }
}
```

```
}
```

调用方式如下：

```
IEnumerable<int> numbers = new List<int> {1, 2, 3, 4, 5, 6};  
var hasMoreThanThreeElements = numbers.HasMoreThanThreeElements();
```

[查看演示](#)

同样适用于多个类型参数：

```
public static TU GenericExt<T, TU>(this T obj)  
{  
    TU ret = default(TU);  
    // 对 obj 做一些操作  
    return ret;  
}
```

调用它的方式如下：

```
IEnumerable<int> numbers = new List<int> {1, 2, 3, 4, 5, 6};  
var result = numbers.GenericExt<IEnumerable<int>, String>();
```

[查看演示](#)

你也可以为多泛型类型中部分绑定的类型创建扩展方法：

```
class MyType<T1, T2>  
{  
  
    static class Extensions  
    {  
        public static void Example<T>(this MyType<int, T> test)  
        {  
        }  
    }  
}
```

调用它的方式如下：

```
MyType<int, string> t = new MyType<int, string>();  
t.Example();
```

[查看演示](#)

你也可以使用 `where` 指定类型约束：

```
public static bool IsDefault<T>(this T obj) where T : struct, IEquatable<T>  
{  
    return EqualityComparer<T>.Default.Equals(obj, default(T));  
}
```

调用代码：

```
int number = 5;  
var IsDefault = number.IsDefault();
```

```
}
```

and calling it would be like:

```
IEnumerable<int> numbers = new List<int> {1, 2, 3, 4, 5, 6};  
var hasMoreThanThreeElements = numbers.HasMoreThanThreeElements();
```

[View Demo](#)

Likewise for multiple Type Arguments:

```
public static TU GenericExt<T, TU>(this T obj)  
{  
    TU ret = default(TU);  
    // do some stuff with obj  
    return ret;  
}
```

Calling it would be like:

```
IEnumerable<int> numbers = new List<int> {1, 2, 3, 4, 5, 6};  
var result = numbers.GenericExt<IEnumerable<int>, String>();
```

[View Demo](#)

You can also create extension methods for partially bound types in multi generic types:

```
class MyType<T1, T2>  
{  
  
    static class Extensions  
    {  
        public static void Example<T>(this MyType<int, T> test)  
        {  
        }  
    }  
}
```

Calling it would be like:

```
MyType<int, string> t = new MyType<int, string>();  
t.Example();
```

[View Demo](#)

You can also specify type constraints with `where` :

```
public static bool IsDefault<T>(this T obj) where T : struct, IEquatable<T>  
{  
    return EqualityComparer<T>.Default.Equals(obj, default(T));  
}
```

Calling code:

```
int number = 5;  
var IsDefault = number.IsDefault();
```

第48.5节：扩展方法只能访问被扩展类的公共（或内部）成员

```
public class SomeClass
{
    public void DoStuff()
    {
    }

    protected void DoMagic()
    {
    }
}

public static class SomeClassExtensions
{
    public static void DoStuffWrapper(this SomeClass someInstance)
    {
        someInstance.DoStuff(); // 正常
    }

    public static void DoMagicWrapper(this SomeClass someInstance)
    {
        someInstance.DoMagic(); // 编译错误
    }
}
```

扩展方法只是语法糖，实际上并不是它们所扩展类的成员。这意味着它们不能破坏封装—它们只能访问`public`（或者当在同一程序集内实现时，`internal`）字段、属性和方法。

第48.6节：用于链式调用的扩展方法

当扩展方法返回的值与其`this`参数类型相同时，可以用来“链式”调用一个或多个具有兼容签名的方法。这对于密封类和/或原始类型非常有用，并且如果方法名像自然语言一样易读，还可以创建所谓的“流畅”API。

```
void Main()
{
    int result = 5.Increment().Decrement().Increment();
    // 结果现在是6
}

public static class IntExtensions
{
    public static int Increment(this int number) {
        return ++number;
    }

    public static int Decrement(this int number) {
        return --number;
    }
}
```

Section 48.5: Extension methods can only see public (or internal) members of the extended class

```
public class SomeClass
{
    public void DoStuff()
    {
    }

    protected void DoMagic()
    {
    }
}

public static class SomeClassExtensions
{
    public static void DoStuffWrapper(this SomeClass someInstance)
    {
        someInstance.DoStuff(); // ok
    }

    public static void DoMagicWrapper(this SomeClass someInstance)
    {
        someInstance.DoMagic(); // compilation error
    }
}
```

Extension methods are just a syntactic sugar, and are not actually members of the class they extend. This means that they cannot break encapsulation—they only have access to `public` (or when implemented in the same assembly, `internal`) fields, properties and methods.

Section 48.6: Extension methods for chaining

When an extension method returns a value that has the same type as its `this` argument, it can be used to "chain" one or more method calls with a compatible signature. This can be useful for sealed and/or primitive types, and allows the creation of so-called "fluent" APIs if the method names read like natural human language.

```
void Main()
{
    int result = 5.Increment().Decrement().Increment();
    // result is now 6
}

public static class IntExtensions
{
    public static int Increment(this int number) {
        return ++number;
    }

    public static int Decrement(this int number) {
        return --number;
    }
}
```

或者像这样

```
void Main()
{
    int[] ints = new[] { 1, 2, 3, 4, 5, 6 };
    int[] a = ints.WhereEven();
    //a 是 { 2, 4, 6 };
    int[] b = ints.WhereEven().WhereGreaterThan(2);
    //b 是 { 4, 6 };
}

public static class IntArrayExtensions
{
    public static int[] WhereEven(this int[] array)
    {
        //Enumerable.* 扩展方法使用流式调用方式
        return array.Where(i => (i%2) == 0).ToArray();
    }

    public static int[] WhereGreaterThan(this int[] array, int value)
    {
        return array.Where(i => i > value).ToArray();
    }
}
```

第48.7节：带枚举的扩展方法

扩展方法对于为枚举添加功能非常有用。

一个常见的用法是实现转换方法。

```
public enum 是否
{
    是,
    否,
}

public static class 枚举扩展
{
    public static bool 转换为布尔值(this 是否 yn)
    {
        return yn == 是否.是;
    }

    public static 是否 转换为是否(this bool yn)
    {
        return yn ? 是否.Yes : 是否.No;
    }
}
```

现在你可以快速地将枚举值转换为不同的类型。在这个例子中是转换为bool类型。

```
bool yesNoBool = 是否.Yes.ToBoolean(); // yesNoBool == true
是No yesNoEnum = false.ToBoolean(); // yesNoEnum == 是No.No
```

或者可以使用扩展方法来添加类似属性的方法。

```
public enum 元素
{
    氢,
```

Or like this

```
void Main()
{
    int[] ints = new[] { 1, 2, 3, 4, 5, 6 };
    int[] a = ints.WhereEven();
    //a is { 2, 4, 6 };
    int[] b = ints.WhereEven().WhereGreaterThan(2);
    //b is { 4, 6 };
}

public static class IntArrayExtensions
{
    public static int[] WhereEven(this int[] array)
    {
        //Enumerable.* extension methods use a fluent approach
        return array.Where(i => (i%2) == 0).ToArray();
    }

    public static int[] WhereGreaterThan(this int[] array, int value)
    {
        return array.Where(i => i > value).ToArray();
    }
}
```

Section 48.7: Extension methods with Enumeration

Extension methods are useful for adding functionality to enumerations.

One common use is to implement a conversion method.

```
public enum 是否
{
    是,
    否,
}

public static class EnumExtentions
{
    public static bool ToBool(this 是否 yn)
    {
        return yn == 是否.是;
    }

    public static 是否 ToYesNo(this bool yn)
    {
        return yn ? 是否.Yes : 是否.No;
    }
}
```

Now you can quickly convert your enum value to a different type. In this case a bool.

```
bool yesNoBool = 是否.Yes.ToBoolean(); // yesNoBool == true
是No yesNoEnum = false.ToBoolean(); // yesNoEnum == 是No.No
```

Alternatively extension methods can be used to add property like methods.

```
public enum 元素
{
    氢,
```

```

        氢,
        锂,
        镁,
        硼,
        碳,
        氮,
        氧
        //等等
    }

    public static class 元素扩展
    {
        public static double 原子质量(this 元素 element)
        {
            switch(元素)
            {
                case 元素.氢:   返回 1.00794;
                case 元素.氦:   返回 4.002602;
                case 元素.锂:   返回 6.941;
                case 元素.镁:   返回 9.012182;
                case 元素.硼:   返回 10.811;
                case 元素.碳:   返回 12.0107;
                case 元素.氮:   返回 14.0067;
                case 元素.氧:   返回 15.9994;
                //等等
            }
            返回 double.Nan;
        }
    }

    var 水的质量 = 2*元素.氢.原子质量() + 元素.氧.原子质量();

```

```

        Helium,
        Lithium,
        Beryllium,
        Boron,
        Carbon,
        Nitrogen,
        Oxygen
        //Etc
    }

    public static class ElementExtensions
    {
        public static double AtomicMass(this Element element)
        {
            switch(element)
            {
                case Element.Hydrogen: return 1.00794;
                case Element.Helium:   return 4.002602;
                case Element.Lithium:  return 6.941;
                case Element.Beryllium: return 9.012182;
                case Element.Boron:   return 10.811;
                case Element.Carbon:  return 12.0107;
                case Element.Nitrogen: return 14.0067;
                case Element.Oxygen:  return 15.9994;
                //Etc
            }
            return double.Nan;
        }
    }

    var massWater = 2*Element.Hydrogen.AtomicMass() + Element.Oxygen.AtomicMass();

```

第48.8节：基于静态类型的扩展方法分派

使用静态（编译时）类型而非动态（运行时）类型来匹配参数。

```

public class 基类
{
    public virtual string 获取名称()
    {
        返回 "基类";
    }
}

public class 派生类 : 基类
{
    public override string 获取名称()
    {
        return "Derived";
    }
}

public static class 扩展
{
    public static string GetNameByExtension(this Base 项目)
    {
        return "Base";
    }

    public static string GetNameByExtension(this Derived 项目)

```

Section 48.8: Extension methods dispatch based on static type

The static (compile-time) type is used rather than the dynamic (run-time type) to match parameters.

```

public class Base
{
    public virtual string GetName()
    {
        return "Base";
    }
}

public class Derived : Base
{
    public override string GetName()
    {
        return "Derived";
    }
}

public static class Extensions
{
    public static string GetNameByExtension(this Base item)
    {
        return "Base";
    }

    public static string GetNameByExtension(this Derived item)

```

```

    {
        return "Derived";
    }

}

public static class 程序
{
    public static void Main()
    {
        Derived 派生 = new Derived();
        Base @基类 = 派生;

        // 使用实例方法 "GetName"
        Console.WriteLine(派生.GetName()); // 输出 "Derived"
        Console.WriteLine(@基类.GetName()); // 输出 "Derived"

        // 使用静态扩展方法 "GetNameByExtension"
        Console.WriteLine(derived.GetNameByExtension()); // 输出 "Derived"
        Console.WriteLine(@base.GetNameByExtension()); // 输出 "Base"
    }
}

```

[.NET Fiddle 在线演示](#)

基于静态类型的调度也不允许在dynamic对象上调用扩展方法：

```

public class Person
{
    public string Name { get; set; }
}

public static class ExtensionPerson
{
    public static string GetPersonName(this Person person)
    {
        return person.Name;
    }
}

dynamic person = new Person { Name = "Jon" };
var name = person.GetPersonName(); // 抛出 RuntimeBinderException

```

第48.9节：接口上的扩展方法

扩展方法的一个有用特性是你可以为接口创建通用方法。通常接口不能有共享的实现，但通过扩展方法它们可以。

```

public interface IVehicle
{
    int MilesDriven { get; set; }
}

public static class 扩展
{
    public static int FeetDriven(this IVehicle vehicle)
    {
        return vehicle.MilesDriven * 5028;
    }
}

```

```

    {
        return "Derived";
    }

}

public static class Program
{
    public static void Main()
    {
        Derived derived = new Derived();
        Base @base = derived;

        // Use the instance method "GetName"
        Console.WriteLine(derived.GetName()); // Prints "Derived"
        Console.WriteLine(@base.GetName()); // Prints "Derived"

        // Use the static extension method "GetNameByExtension"
        Console.WriteLine(derived.GetNameByExtension()); // Prints "Derived"
        Console.WriteLine(@base.GetNameByExtension()); // Prints "Base"
    }
}

```

[Live Demo on .NET Fiddle](#)

Also the dispatch based on static type does not allow an extension method to be called on a `dynamic` object:

```

public class Person
{
    public string Name { get; set; }
}

public static class ExtensionPerson
{
    public static string GetPersonName(this Person person)
    {
        return person.Name;
    }
}

dynamic person = new Person { Name = "Jon" };
var name = person.GetPersonName(); // RuntimeBinderException is thrown

```

Section 48.9: Extension methods on Interfaces

One useful feature of extension methods is that you can create common methods for an interface. Normally an interface cannot have shared implementations, but with extension methods they can.

```

public interface IVehicle
{
    int MilesDriven { get; set; }
}

public static class Extensions
{
    public static int FeetDriven(this IVehicle vehicle)
    {
        return vehicle.MilesDriven * 5028;
    }
}

```

在此示例中，方法FeetDriven可以用于任何IVehicle。该方法中的逻辑适用于所有IVehicle，因此可以这样实现，这样就不必在IVehicle定义中包含一个FeetDriven，该方法对所有子类的实现方式都是相同的。

第48.10节：扩展方法与接口的结合使用

将扩展方法与接口一起使用非常方便，因为实现可以存储在类之外，只需将类装饰为接口即可为类添加某些功能。

```
public interface IInterface
{
    string Do();
}

public static class ExtensionMethods{
    public static string DoWith(this IInterface obj){
        //对IInterface实例执行某些操作
    }
}

public class Classy : IInterface
{
    // 这是一个包装方法；你也可以直接在 Classy 实例上调用 DoWith(),
    // 前提是你导入了包含该扩展方法的命名空间
    public Do(){
        return this.DoWith();
    }
}
```

用法示例：

```
var classy = new Classy();
classy.Do(); // 将调用扩展方法
classy.DoWith(); // Classy 实现了 IInterface，因此也可以这样调用
```

第48.11节：扩展方法不支持动态代码

```
static class Program
{
    static void Main()
    {
        dynamic dynamicObject = new ExpandoObject();

        string awesomeString = "Awesome";

        // 输出 True
        Console.WriteLine(awesomeString.IsTrueAwesome());

        dynamicObject.StringValue = awesomeString;

        // 输出 True
        Console.WriteLine(StringExtensions.IsTrueAwesome(dynamicObject.StringValue));

        // 编译时无错误或警告，但运行时抛出 RuntimeBinderException
        Console.WriteLine(dynamicObject.StringValue.IsTrueAwesome());
    }
}
```

In this example, the method FeetDriven can be used on any IVehicle. This logic in this method would apply to all IVehicles, so it can be done this way so that there doesn't have to be a FeetDriven in the IVehicle definition which would be implemented the same way for all children.

Section 48.10: Extension methods in combination with interfaces

It is very convenient to use extension methods with interfaces as implementation can be stored outside of class and all it takes to add some functionality to class is to decorate class with interface.

```
public interface IInterface
{
    string Do();
}

public static class ExtensionMethods{
    public static string DoWith(this IInterface obj){
        //does something with IInterface instance
    }
}

public class Classy : IInterface
{
    // this is a wrapper method; you could also call DoWith() on a Classy instance directly,
    // provided you import the namespace containing the extension method
    public Do(){
        return this.DoWith();
    }
}
```

use like:

```
var classy = new Classy();
classy.Do(); // will call the extension
classy.DoWith(); // Classy implements IInterface so it can also be called this way
```

Section 48.11: Extension methods aren't supported by dynamic code

```
static class Program
{
    static void Main()
    {
        dynamic dynamicObject = new ExpandoObject();

        string awesomeString = "Awesome";

        // Prints True
        Console.WriteLine(awesomeString.IsTrueAwesome());

        dynamicObject.StringValue = awesomeString;

        // Prints True
        Console.WriteLine(StringExtensions.IsTrueAwesome(dynamicObject.StringValue));

        // No compile time error or warning, but on runtime throws RuntimeBinderException
        Console.WriteLine(dynamicObject.StringValue.IsTrueAwesome());
    }
}
```

```

static class StringExtensions
{
    public static bool IsThisAwesome(this string value)
    {
        return value.Equals("Awesome");
    }
}

```

调用扩展方法失败的原因是，因为在常规的非动态代码中，扩展方法通过对编译器已知的所有类进行全面搜索来实现，寻找包含匹配扩展方法的静态类。搜索顺序基于命名空间的嵌套关系以及每个命名空间中可用的using指令。

这意味着，为了让动态扩展方法调用正确解析，DLR必须在运行时知道你源代码中所有的命名空间嵌套和using指令。我们没有现成的机制将所有这些信息编码到调用点。我们曾考虑发明这样一种机制，但认为其成本过高且带来太多进度风险，不值得实施。

[来源](#)

第48.12节：扩展和接口共同实现了DRY代码和类似 mixin 的功能

扩展方法使您能够简化接口定义，仅在接口本身中包含核心必需的功能，并允许您将便捷方法和重载定义为扩展方法。方法较少的接口更容易在新类中实现。将重载作为扩展而不是直接包含在接口中，可以避免将样板代码复制到每个实现中，帮助你保持代码的干燥（DRY）。这实际上类似于 C# 不支持的混入（mixin）模式。

System.Linq.Enumerable 对 IEnumerable<T> 的扩展是一个很好的例子。 IEnumerable<T> 只要求实现类实现两个方法：泛型和非泛型的 GetGetEnumerator()。但是 System.Linq.Enumerable 提供了无数有用的扩展工具，使得对 IEnumerable<T> 的使用简洁明了。

下面是一个非常简单的接口，带有作为扩展提供的便捷重载。

```

public interface ITimeFormatter
{
    string Format(TimeSpan span);
}

public static class TimeFormatter
{
    // 为所有 ITimeFormatter 的实现者提供一个重载。
    public static string Format(
        this ITimeFormatter formatter,
        int millisecondsSpan)
    => formatter.Format(TimeSpan.FromMilliseconds(millisecondsSpan));
}

// 实现只需提供一个方法。编写额外的实现非常简单。

```

```

}
}

static class StringExtensions
{
    public static bool IsThisAwesome(this string value)
    {
        return value.Equals("Awesome");
    }
}

```

The reason [calling extension methods from dynamic code] doesn't work is because in regular, non-dynamic code extension methods work by doing a full search of all the classes known to the compiler for a static class that has an extension method that matches. The search goes in order based on the namespace nesting and available `using` directives in each namespace.

That means that in order to get a dynamic extension method invocation resolved correctly, somehow the DLR has to know *at runtime* what all the namespace nestings and `using` directives were *in your source code*. We do not have a mechanism handy for encoding all that information into the call site. We considered inventing such a mechanism, but decided that it was too high cost and produced too much schedule risk to be worth it.

[Source](#)

Section 48.12: Extensions and interfaces together enable DRY code and mixin-like functionality

Extension methods enable you to simplify your interface definitions by only including core required functionality in the interface itself and allowing you to define convenience methods and overloads as extension methods. Interfaces with fewer methods are easier to implement in new classes. Keeping overloads as extensions rather than including them in the interface directly saves you from copying boilerplate code into every implementation, helping you keep your code DRY. This in fact is similar to the mixin pattern which C# does not support.

System.Linq.Enumerable's extensions to IEnumerable<T> is a great example of this. IEnumerable<T> only requires the implementing class to implement two methods: generic and non-generic GetGetEnumerator(). But System.Linq.Enumerable provides countless useful utilities as extensions enabling concise and clear consumption of IEnumerable<T>.

The following is a very simple interface with convenience overloads provided as extensions.

```

public interface ITimeFormatter
{
    string Format(TimeSpan span);
}

public static class TimeFormatter
{
    // Provide an overload to *all* implementers of ITimeFormatter.
    public static string Format(
        this ITimeFormatter formatter,
        int millisecondsSpan)
    => formatter.Format(TimeSpan.FromMilliseconds(millisecondsSpan));

    // Implementations only need to provide one method. Very easy to
    // write additional implementations.
}

```

```

public class SecondsTimeFormatter : ITimeFormatter
{
    public string Format(TimeSpan span)
    {
        return $"{(int)span.TotalSeconds}s";
    }
}

class Program
{
    static void Main(string[] args)
    {
        var formatter = new SecondsTimeFormatter();
        // 调用者获得两个方法重载！
        Console.WriteLine($"4500ms 大约是 {formatter.Format(4500)}");
        var span = TimeSpan.FromSeconds(5);
        Console.WriteLine($"{span} 格式化为 {formatter.Format(span)}");
    }
}

```

第48.13节：IList<T> 扩展方法示例：比较 2 个列表

您可以使用以下扩展方法来比较两个相同类型的 IList<T> 实例的内容。

默认情况下，项目是根据它们在列表中的顺序以及项目本身进行比较的，向 isOrdered 参数传递 false 将只比较项目本身，而不考虑它们的顺序。

为了使此方法生效，泛型类型 (T) 必须重写 Equals 和 GetHashCode 方法。

用法：

```

List<string> list1 = new List<string> {"a1", "a2", null, "a3"};
List<string> list2 = new List<string> {"a1", "a2", "a3", null};

list1.Compare(list2); // 这将返回 false
list1.Compare(list2, false); // 这将返回 true。当忽略顺序时，它们是相等的

```

方法：

```

public static bool Compare<T>(this IList<T> list1, IList<T> list2, bool isOrdered = true)
{
    if (list1 == null && list2 == null)
        return true;
    if (list1 == null || list2 == null || list1.Count != list2.Count)
        return false;

    if (isOrdered)
    {
        for (int i = 0; i < list2.Count; i++)
        {
            var l1 = list1[i];
            var l2 = list2[i];
            if (
                (l1 == null && l2 != null) ||
                (l1 != null && l2 == null) ||
                (!l1.Equals(l2)))
            {

```

```

public class SecondsTimeFormatter : ITimeFormatter
{
    public string Format(TimeSpan span)
    {
        return $"{(int)span.TotalSeconds}s";
    }
}

class Program
{
    static void Main(string[] args)
    {
        var formatter = new SecondsTimeFormatter();
        // Callers get two method overloads!
        Console.WriteLine($"4500ms is roughly {formatter.Format(4500)}");
        var span = TimeSpan.FromSeconds(5);
        Console.WriteLine($"{span} is formatted as {formatter.Format(span)}");
    }
}

```

Section 48.13: IList<T> Extension Method Example: Comparing 2 Lists

You can use the following extension method for comparing the contents of two IList<T> instances of the same type.

By default the items are compared based on their order within the list and the items themselves, passing false to the isOrdered parameter will compare only the items themselves regardless of their order.

For this method to work, the generic type (T) must override both Equals and GetHashCode methods.

Usage:

```

List<string> list1 = new List<string> {"a1", "a2", null, "a3"};
List<string> list2 = new List<string> {"a1", "a2", "a3", null};

list1.Compare(list2); // this gives false
list1.Compare(list2, false); // this gives true. they are equal when the order is disregarded

```

Method:

```

public static bool Compare<T>(this IList<T> list1, IList<T> list2, bool isOrdered = true)
{
    if (list1 == null && list2 == null)
        return true;
    if (list1 == null || list2 == null || list1.Count != list2.Count)
        return false;

    if (isOrdered)
    {
        for (int i = 0; i < list2.Count; i++)
        {
            var l1 = list1[i];
            var l2 = list2[i];
            if (
                (l1 == null && l2 != null) ||
                (l1 != null && l2 == null) ||
                (!l1.Equals(l2)))
            {

```

```

        return false;
    }
    return true;
}
else
{
List<T> list2Copy = new List<T>(list2);
//可以用字典实现，避免O(n^2)复杂度
for (int i = 0; i < list1.Count; i++)
{
    if (!list2Copy.Remove(list1[i]))
        return false;
}
return true;
}
}

```

第48.14节：作为强类型包装器的扩展方法

扩展方法可用于为类似字典的对象编写强类型包装器。例如缓存，`HttpContext.Items` 等等...

```

public static class CacheExtensions
{
    public static void SetUserInfo(this Cache cache, UserInfo data) =>
        cache["UserInfo"] = data;

    public static UserInfo GetUserInfo(this Cache cache) =>
        cache["UserInfo"] as UserInfo;
}

```

这种方法消除了在整个代码库中使用字符串字面量作为键的需求，也避免了读取操作时需要进行类型转换。总体来说，它为与诸如字典这类松散类型对象的交互创建了一种更安全、强类型的方式。

第48.15节：使用扩展方法创建优雅的映射器类

我们可以通过扩展方法创建更好的映射器类，假设我有一些DTO类，比如

```

public class UserDTO
{
    public AddressDTO Address { get; set; }
}

public class AddressDTO
{
    public string Name { get; set; }
}

```

并且我需要映射到相应的视图模型类

```

public class UserViewModel
{
    public AddressViewModel Address { get; set; }
}

```

```

        return false;
    }
    return true;
}
else
{
List<T> list2Copy = new List<T>(list2);
//Can be done with Dictionary without O(n^2)
for (int i = 0; i < list1.Count; i++)
{
    if (!list2Copy.Remove(list1[i]))
        return false;
}
return true;
}
}

```

Section 48.14: Extension methods as strongly typed wrappers

Extension methods can be used for writing strongly typed wrappers for dictionary-like objects. For example a cache, `HttpContext.Items` at cetera...

```

public static class CacheExtensions
{
    public static void SetUserInfo(this Cache cache, UserInfo data) =>
        cache["UserInfo"] = data;

    public static UserInfo GetUserInfo(this Cache cache) =>
        cache["UserInfo"] as UserInfo;
}

```

This approach removes the need of using string literals as keys all over the codebase as well as the need of casting to the required type during the read operation. Overall it creates a more secure, strongly typed way of interacting with such loosely typed objects as Dictionaries.

Section 48.15: Using Extension methods to create beautiful mapper classes

We can create a better mapper classes with extension methods, Suppose if i have some DTO classes like

```

public class UserDTO
{
    public AddressDTO Address { get; set; }
}

public class AddressDTO
{
    public string Name { get; set; }
}

```

and i need to map to corresponding view model classes

```

public class UserViewModel
{
    public AddressViewModel Address { get; set; }
}

```

```
public class AddressViewModel
{
    public string Name { get; set; }
}
```

然后我可以像下面这样创建我的映射器类

```
public static class ViewModelMapper
{
    public static UserViewModel ToViewModel(this UserDTO user)
    {
        return user == null ?
            null :
            new UserViewModel()
        {
            Address = user.Address.ToViewModel()
                // Job = user.Job.ToViewModel(),
                // Contact = user.Contact.ToViewModel() .. 等等
        };
    }

    public static AddressViewModel ToViewModel(this AddressDTO userAddr)
    {
        return userAddr == null ?
            null :
            new AddressViewModel()
        {
            Name = userAddr.Name
        };
    }
}
```

然后我最终可以像下面这样调用我的映射器

```
UserDTO userDTOObj = new UserDTO() {
    Address = new AddressDTO() {
        Name = "用户的地址"
    }
};

UserViewModel user = userDTOObj.ToViewModel(); // 我的DTO映射到ViewModel
```

这里的优点是所有映射方法都有一个通用名称 (ToViewModel) , 我们可以多种方式重用它

第48.16节：使用扩展方法构建新的集合类型（例如 DictList）

你可以创建扩展方法来提升嵌套集合的可用性，比如带有List<T>值的Dictionary。

考虑以下扩展方法：

```
public static class DictListExtensions
{
    public static void Add< TKey, TValue, TCollection>(this Dictionary< TKey, TCollection> dict, TKey key, TValue value)
        where TCollection : ICollection< TValue >, new()
    {
        TCollection list;
```

```
public class AddressViewModel
{
    public string Name { get; set; }
}
```

then I can create my mapper class like below

```
public static class ViewModelMapper
{
    public static UserViewModel ToViewModel(this UserDTO user)
    {
        return user == null ?
            null :
            new UserViewModel()
        {
            Address = user.Address.ToViewModel()
                // Job = user.Job.ToViewModel(),
                // Contact = user.Contact.ToViewModel() .. and so on
        };
    }

    public static AddressViewModel ToViewModel(this AddressDTO userAddr)
    {
        return userAddr == null ?
            null :
            new AddressViewModel()
        {
            Name = userAddr.Name
        };
    }
}
```

Then finally i can invoke my mapper like below

```
UserDTO userDT0bj = new UserDTO() {
    Address = new AddressDTO() {
        Name = "Address of the user"
    }
};

UserViewModel user = userDT0bj.ToViewModel(); // My DTO mapped to Viewmodel
```

The beauty here is all the mapping method have a common name (ToViewModel) and we can reuse it several ways

Section 48.16: Using Extension methods to build new collection types (e.g. DictList)

You can create extension methods to improve usability for nested collections like a Dictionary with a List<T> value.

Consider the following extension methods:

```
public static class DictListExtensions
{
    public static void Add< TKey, TValue, TCollection>(this Dictionary< TKey, TCollection> dict, TKey key, TValue value)
        where TCollection : ICollection< TValue >, new()
    {
        TCollection list;
```

```

    if (!dict.TryGetValue(key, out list))
    {
        list = new TCollection();
        dict.Add(key, list);
    }

    list.Add(value);
}

public static bool Remove<TKey, TValue, TCollection>(this Dictionary<TKey, TCollection> dict,
TKey key, TValue value)
    where TCollection : ICollection<TValue>
{
    TCollection list;
    if (!dict.TryGetValue(key, out list))
    {
        return false;
    }

    var ret = list.Remove(value);
    if (list.Count == 0)
    {
        dict.Remove(key);
    }
    return ret;
}
}

```

你可以按如下方式使用扩展方法：

```

var dictList = new Dictionary<string, List<int>>();

dictList.Add("example", 5);
dictList.Add("example", 10);
dictList.Add("example", 15);

Console.WriteLine(String.Join(", ", dictList["example"])); // 5, 10, 15

dictList.Remove("example", 5);
dictList.Remove("example", 10);

Console.WriteLine(String.Join(", ", dictList["example"])); // 15

dictList.Remove("example", 15);

Console.WriteLine(dictList.ContainsKey("example")); // False

```

[查看演示](#)

第48.17节：处理特殊情况的扩展方法

扩展方法可以用来“隐藏”那些笨拙的业务规则处理，否则这些规则会使调用函数充满if/then语句。这类似于使用扩展方法处理null的方式。例如，

```

public static class CakeExtensions
{
    public static Cake EnsureTrueCake(this Cake cake)
    {
        //如果蛋糕是谎言，则用奶奶的蛋糕替代，奶奶的蛋糕虽然味道不如人意，

```

```

    if (!dict.TryGetValue(key, out list))
    {
        list = new TCollection();
        dict.Add(key, list);
    }

    list.Add(value);
}

public static bool Remove<TKey, TValue, TCollection>(this Dictionary<TKey, TCollection> dict,
TKey key, TValue value)
    where TCollection : ICollection<TValue>
{
    TCollection list;
    if (!dict.TryGetValue(key, out list))
    {
        return false;
    }

    var ret = list.Remove(value);
    if (list.Count == 0)
    {
        dict.Remove(key);
    }
    return ret;
}
}

```

you can use the extension methods as follows:

```

var dictList = new Dictionary<string, List<int>>();

dictList.Add("example", 5);
dictList.Add("example", 10);
dictList.Add("example", 15);

Console.WriteLine(String.Join(", ", dictList["example"])); // 5, 10, 15

dictList.Remove("example", 5);
dictList.Remove("example", 10);

Console.WriteLine(String.Join(", ", dictList["example"])); // 15

dictList.Remove("example", 15);

Console.WriteLine(dictList.ContainsKey("example")); // False

```

[View Demo](#)

Section 48.17: Extension methods for handling special cases

Extension methods can be used to "hide" processing of inelegant business rules that would otherwise require cluttering up a calling function with if/then statements. This is similar to and analogous to handling nulls with extension methods. For example,

```

public static class CakeExtensions
{
    public static Cake EnsureTrueCake(this Cake cake)
    {
        //If the cake is a lie, substitute a cake from grandma, whose cakes aren't as tasty but are

```

但众所周知绝不会是谎言。如果蛋糕不是谎言，则不做任何处理，直接返回。

```
    return CakeVerificationService.IsCakeLie(cake) ? GrandmasKitchen.Get1950sCake() : cake;
}

Cake myCake = Bakery.GetNextCake().EnsureTrueCake();
myMouth.Eat(myCake); //吃蛋糕，确信它不是真假蛋糕。
```

第48.18节：将扩展方法与静态方法和回调一起使用

考虑将扩展方法用作包装其他代码的函数，下面是一个很好的示例，它同时使用了静态方法和扩展方法来包装Try Catch结构。让你的代码坚不可摧.....

```
using System;
using System.Diagnostics;

namespace Samples
{
    /// <summary>
    /// 将 try catch 语句封装为一个静态辅助方法，使用
    /// 异常的扩展方法
    /// </summary>
    public static class Bullet
    {
        /// <summary>
        /// Try Catch 语句的封装器
        /// </summary>
        /// <param name="code">代码回调</param>
        /// <param name="error">已处理并记录的异常</param>
        public static void Proof(Action code, Action<Exception> error)
        {
            try
            {
                code();
            }
            catch (Exception iox)
            {
                // 此处使用扩展方法
                iox.Log("BP2200-ERR-Unexpected Error");
                // 回调，异常已处理并记录
                error(iox);
            }
        }
        /// <summary>
        /// 日志方法辅助示例，这是扩展方法
        ///
        /// <param name="error">要记录的异常</param>
        /// <param name="messageID">唯一的错误ID头</param>
        public static void Log(this Exception error, string messageID)
        {
            Trace.WriteLine(messageID);
            Trace.WriteLine(error.Message);
            Trace.WriteLine(error.StackTrace);
            Trace.WriteLine("");
        }
    }
    /// <summary>
    /// 展示如何使用包装器和扩展方法。
    /// </summary>
    public class UseBulletProofing
```

known never to be lies. If the cake isn't a lie, don't do anything and return it.

```
    return CakeVerificationService.IsCakeLie(cake) ? GrandmasKitchen.Get1950sCake() : cake;
}

Cake myCake = Bakery.GetNextCake().EnsureTrueCake();
myMouth.Eat(myCake); //Eat the cake, confident that it is not a lie.
```

Section 48.18: Using Extension methods with Static methods and Callbacks

Consider using Extension Methods as Functions which wrap other code, here's a great example that uses both a static method and an extension method to wrap the Try Catch construct. Make your code Bullet Proof...

```
using System;
using System.Diagnostics;

namespace Samples
{
    /// <summary>
    /// Wraps a try catch statement as a static helper which uses
    /// Extension methods for the exception
    /// </summary>
    public static class Bullet
    {
        /// <summary>
        /// Wrapper for Try Catch Statement
        /// </summary>
        /// <param name="code">Call back for code</param>
        /// <param name="error">Already handled and logged exception</param>
        public static void Proof(Action code, Action<Exception> error)
        {
            try
            {
                code();
            }
            catch (Exception iox)
            {
                //extension method used here
                iox.Log("BP2200-ERR-Unexpected Error");
                //callback, exception already handled and logged
                error(iox);
            }
        }
        /// <summary>
        /// Example of a logging method helper, this is the extension method
        /// </summary>
        /// <param name="error">The Exception to log</param>
        /// <param name="messageID">A unique error ID header</param>
        public static void Log(this Exception error, string messageID)
        {
            Trace.WriteLine(messageID);
            Trace.WriteLine(error.Message);
            Trace.WriteLine(error.StackTrace);
            Trace.WriteLine("");
        }
    }
    /// <summary>
    /// Shows how to use both the wrapper and extension methods.
    /// </summary>
    public class UseBulletProofing
```

```

{
    public UseBulletProofing()
    {
        var ok = false;
        var result = DoSomething();
        if (!result.Contains("ERR"))
        {
            ok = true;
            DoSomethingElse();
        }
    }

    /// <summary>
    /// 如何在代码中使用防弹处理。
    /// </summary>
    /// <returns>返回一个字符串</returns>
    public string DoSomething()
    {
        string result = string.Empty;
        //注意 Bullet.Proof 方法强制使用此结构。
        Bullet.Proof(() =>
        {
            //这是代码回调
            result = "DST5900-INF-此代码中无异常";
        }, error =>
        {
            //error 是已记录和处理的异常
            //确定基础结果
            result = "DTS6200-ERR-发生异常, 请查看控制台日志";
            if (error.Message.Contains("SomeMarker"))
            {
                //筛选异常消息中包含"Something"的结果
                result = "DST6500-ERR-在异常中发现某个标记";
            }
        });
        return result;
    }

    /// <summary>
    /// 工作流的下一步
    /// </summary>
    public void DoSomethingElse()
    {
        //仅在之前未抛出异常时调用
    }
}

```

```

{
    public UseBulletProofing()
    {
        var ok = false;
        var result = DoSomething();
        if (!result.Contains("ERR"))
        {
            ok = true;
            DoSomethingElse();
        }
    }

    /// <summary>
    /// How to use Bullet Proofing in your code.
    /// </summary>
    /// <returns>A string</returns>
    public string DoSomething()
    {
        string result = string.Empty;
        //Note that the Bullet.Proof method forces this construct.
        Bullet.Proof(() =>
        {
            //this is the code callback
            result = "DST5900-INF-No Exceptions in this code";
        }, error =>
        {
            //error is the already logged and handled exception
            //determine the base result
            result = "DTS6200-ERR-An exception happened look at console log";
            if (error.Message.Contains("SomeMarker"))
            {
                //filter the result for Something within the exception message
                result = "DST6500-ERR-Some marker was found in the exception";
            }
        });
        return result;
    }

    /// <summary>
    /// Next step in workflow
    /// </summary>
    public void DoSomethingElse()
    {
        //Only called if no exception was thrown before
    }
}

```

第49章：命名参数

第49.1节：参数顺序不是必须的

你可以按任何顺序放置命名参数。

示例方法：

```
public static string Sample(string left, string right)
{
    return string.Join("-",left,right);
}
```

调用示例：

```
Console.WriteLine (Sample(left:"A",right:"B"));
Console.WriteLine (Sample(right:"A",left:"B"));
```

结果：

```
A-B
B-A
```

第49.2节：命名参数和可选参数

您可以将命名参数与可选参数结合使用。

来看这个方法：

```
public sealed class SmsUtil
{
    public static bool SendMessage(string from, string to, string message, int retryCount = 5,
object attachment = null)
    {
        // 一些代码
    }
}
```

当你想调用此方法不带设置retryCount参数时：

```
var result = SmsUtil.SendMessage(
    from      : "Cihan",
to       : "Yakar",
    message   : "Hello there!",
attachment : new object());
```

第49.3节：命名参数可以让你的代码更清晰

考虑这个简单的类：

```
class SmsUtil
{
    public bool SendMessage(string from, string to, string message, int retryCount, object
attachment)
```

Chapter 49: Named Arguments

Section 49.1: Argument order is not necessary

You can place named arguments in any order you want.

Sample Method:

```
public static string Sample(string left, string right)
{
    return string.Join("-",left,right);
}
```

Call Sample:

```
Console.WriteLine (Sample(left:"A",right:"B"));
Console.WriteLine (Sample(right:"A",left:"B"));
```

Results:

```
A-B
B-A
```

Section 49.2: Named arguments and optional parameters

You can combine named arguments with optional parameters.

Let see this method:

```
public sealed class SmsUtil
{
    public static bool SendMessage(string from, string to, string message, int retryCount = 5,
object attachment = null)
    {
        // Some code
    }
}
```

When you want to call this method *without* set retryCount argument :

```
var result = SmsUtil.SendMessage(
    from      : "Cihan",
to       : "Yakar",
    message   : "Hello there!",
attachment : new object());
```

Section 49.3: Named Arguments can make your code more clear

Consider this simple class:

```
class SmsUtil
{
    public bool SendMessage(string from, string to, string message, int retryCount, object
attachment)
```

```
{  
    // 一些代码  
}  
}
```

C# 3.0之前是：

```
var result = SmsUtil.SendMessage("Mehran", "Maryam", "Hello there!", 12, null);
```

你可以通过命名参数使这个方法调用更加清晰：

```
var result = SmsUtil.SendMessage(  
    from: "Mehran",  
    to: "Maryam",  
    message "Hello there!",  
    retryCount: 12,  
    attachment: null);
```

```
{  
    // Some code  
}
```

Before C# 3.0 it was:

```
var result = SmsUtil.SendMessage("Mehran", "Maryam", "Hello there!", 12, null);
```

you can make this method call even more clear with **named arguments**:

```
var result = SmsUtil.SendMessage(  
    from: "Mehran",  
    to: "Maryam",  
    message "Hello there!",  
    retryCount: 12,  
    attachment: null);
```

第50章：命名参数和可选参数

第50.1节：可选参数

考虑前面是我们带有可选参数的函数定义。

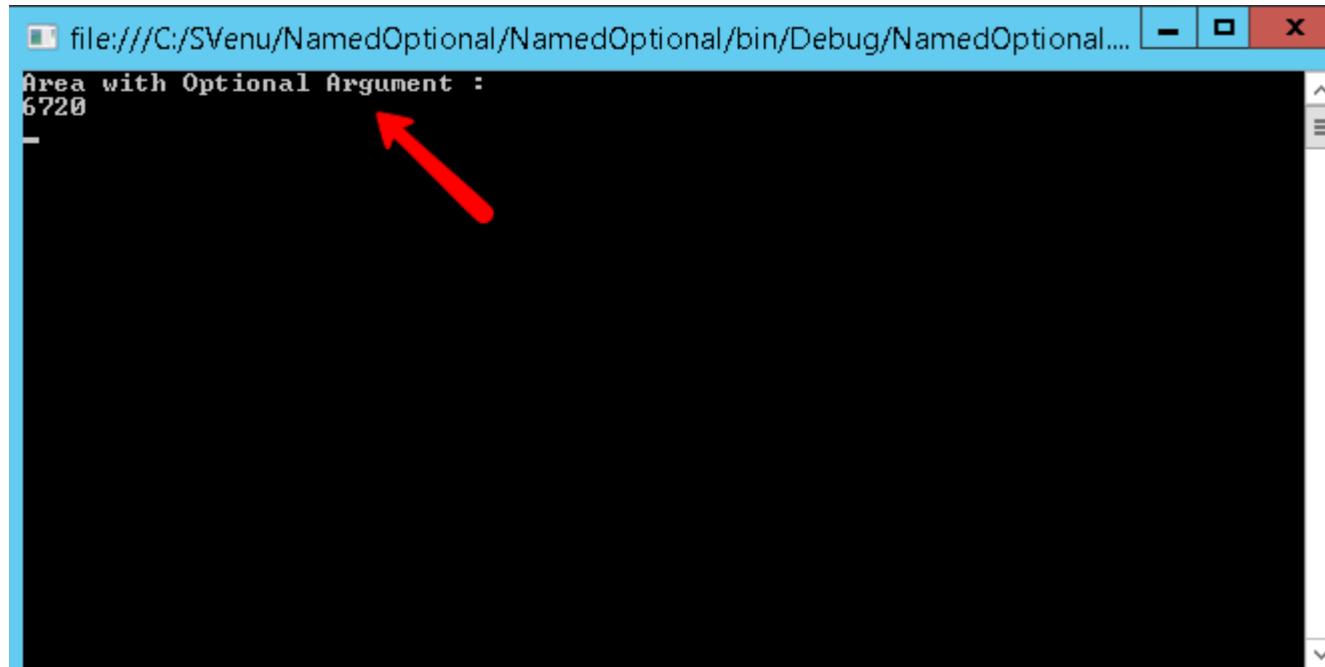
```
private static double FindAreaWithOptional(int length, int width=56)
{
    try
    {
        return (length * width);
    }
    catch (Exception)
    {
        throw new NotImplementedException();
    }
}
```

这里我们将宽度的值设置为可选，并赋值为56。如果你注意到，IntelliSense本身会显示可选参数，如下图所示。

```
-area=FindAreaWithOptional(
    double Program.FindAreaWithOptional(int length, [int width = 56])
```

```
Console.WriteLine("带可选参数的面积 : ");
area = FindAreaWithOptional(120);
Console.WriteLine(area);
Console.Read();
```

注意，我们在编译时没有收到任何错误，输出结果如下。



```
file:///C:/SVenu/NamedOptional/NamedOptional/bin/Debug/NamedOptional...
Area with Optional Argument :
6720
```

使用可选属性。

实现可选参数的另一种方法是使用[Optional]关键字。如果你不传递

Chapter 50: Named and Optional Arguments

Section 50.1: Optional Arguments

Consider preceding is our function definition with optional arguments.

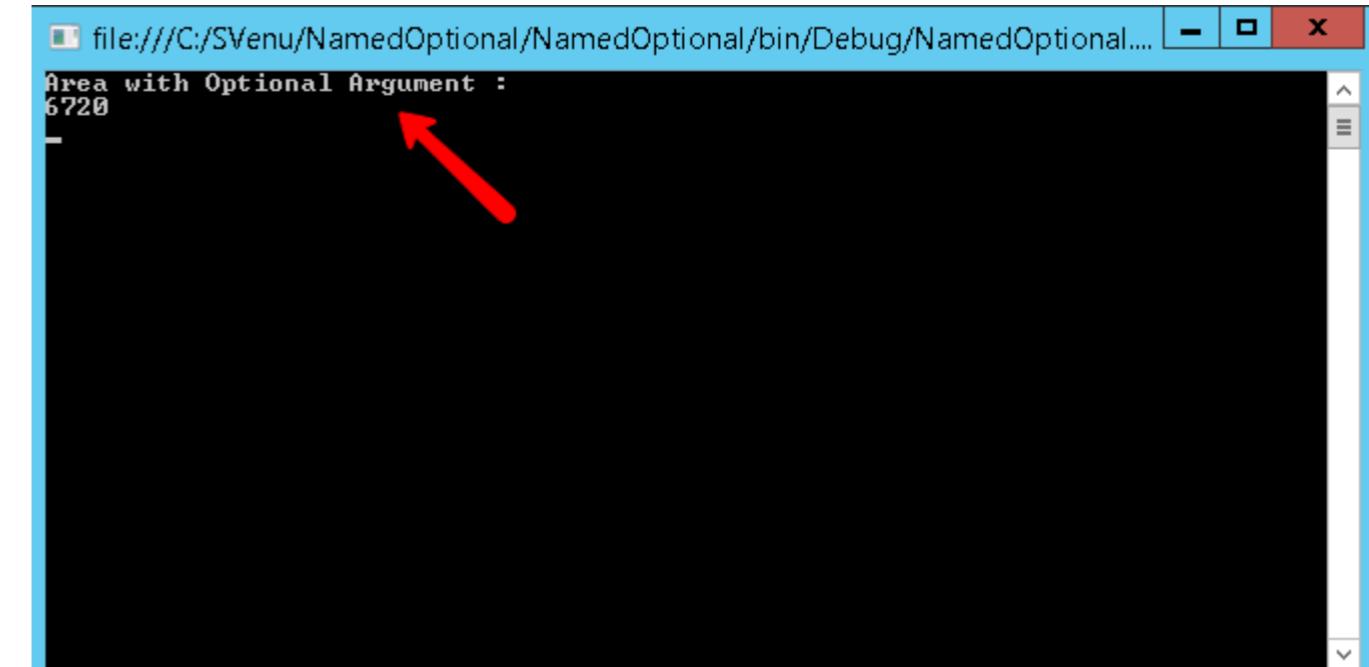
```
private static double FindAreaWithOptional(int length, int width=56)
{
    try
    {
        return (length * width);
    }
    catch (Exception)
    {
        throw new NotImplementedException();
    }
}
```

Here we have set the value for width as optional and gave value as 56. If you note, the IntelliSense itself shows you the optional argument as shown in the below image.

```
-area=FindAreaWithOptional(
    double Program.FindAreaWithOptional(int length, [int width = 56])
```

```
Console.WriteLine("Area with Optional Argument : ");
area = FindAreaWithOptional(120);
Console.WriteLine(area);
Console.Read();
```

Note that we did not get any error while compiling and it will give you an output as follows.



```
file:///C:/SVenu/NamedOptional/NamedOptional/bin/Debug/NamedOptional...
Area with Optional Argument :
6720
```

Using Optional Attribute.

Another way of implementing the optional argument is by using the [Optional] keyword. If you do not pass the

可选参数的值，则该参数会被赋予该数据类型的默认值。 Optional 关键字位于“Runtime.InteropServices”命名空间中。

```
using System.Runtime.InteropServices;
private static double FindAreaWithOptional(int length, [Optional]int width)
{
    try
    {
        return (length * width);
    }
    catch (Exception)
    {
        throw new NotImplementedException();
    }
}

area = FindAreaWithOptional(120); //area=0
```

当我们调用该函数时，得到的结果是0，因为第二个参数没有传入，int类型的默认值是0，所以乘积为0。

第50.2节：命名参数

考虑以下是我们函数调用。

```
FindArea(120, 56);
```

在这里，我们的第一个参数是长度（即120），第二个参数是宽度（即56）。我们通过该函数计算面积。以下是函数定义。

```
private static double FindArea(int length, int width)
{
    try
    {
        return (length * width);
    }
    catch (Exception)
    {
        throw new NotImplementedException();
    }
}
```

所以在第一个函数调用中，我们只是按位置传递了参数。对吗？

```
double area;
Console.WriteLine("使用位置参数计算的面积是: ");
area = FindArea(120, 56);
Console.WriteLine(area);
Console.Read();
```

如果你运行这个，你将得到如下输出。

value for the optional argument, the default value of that datatype is assigned to that argument. The Optional keyword is present in “Runtime.InteropServices” namespace.

```
using System.Runtime.InteropServices;
private static double FindAreaWithOptional(int length, [Optional]int width)
{
    try
    {
        return (length * width);
    }
    catch (Exception)
    {
        throw new NotImplementedException();
    }
}

area = FindAreaWithOptional(120); //area=0
```

And when we call the function, we get 0 because the second argument is not passed and the default value of int is 0 and so the product is 0.

Section 50.2: Named Arguments

Consider following is our function call.

```
FindArea(120, 56);
```

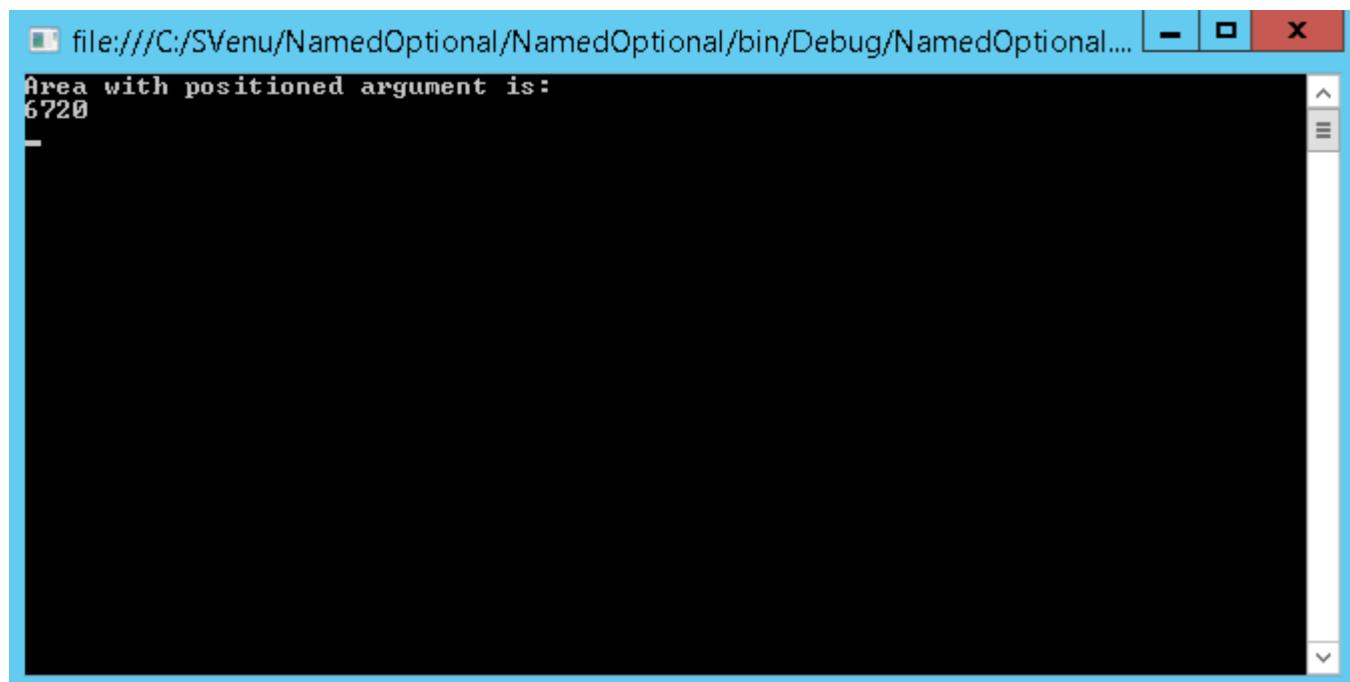
In this our first argument is length (ie 120) and second argument is width (ie 56). And we are calculating the area by that function. And following is the function definition.

```
private static double FindArea(int length, int width)
{
    try
    {
        return (length * width);
    }
    catch (Exception)
    {
        throw new NotImplementedException();
    }
}
```

So in the first function call, we just passed the arguments by its position. Right?

```
double area;
Console.WriteLine("Area with positioned argument is: ");
area = FindArea(120, 56);
Console.WriteLine(area);
Console.Read();
```

If you run this, you will get an output as follows.



现在介绍命名参数的特性。请参见前面的函数调用。

```
Console.WriteLine("使用命名参数计算的面积是: ");
area = FindArea(length: 120, width: 56);
Console.WriteLine(area);
Console.Read();
```

这里我们在方法调用中使用了命名参数。

```
area = FindArea(length: 120, width: 56);
```

现在如果运行这个程序，你会得到相同的结果。如果使用命名参数，我们可以在方法调用中交换参数名的位置。

```
Console.WriteLine("使用命名参数交换位置计算的面积是: ");
area = FindArea(width: 120, length: 56);
Console.WriteLine(area);
Console.Read();
```

命名参数的一个重要用途是，当你在程序中使用它时，可以提高代码的可读性。它清楚地说明了参数的含义或用途。

你也可以同时使用位置参数。这意味着可以结合使用位置参数和命名参数。

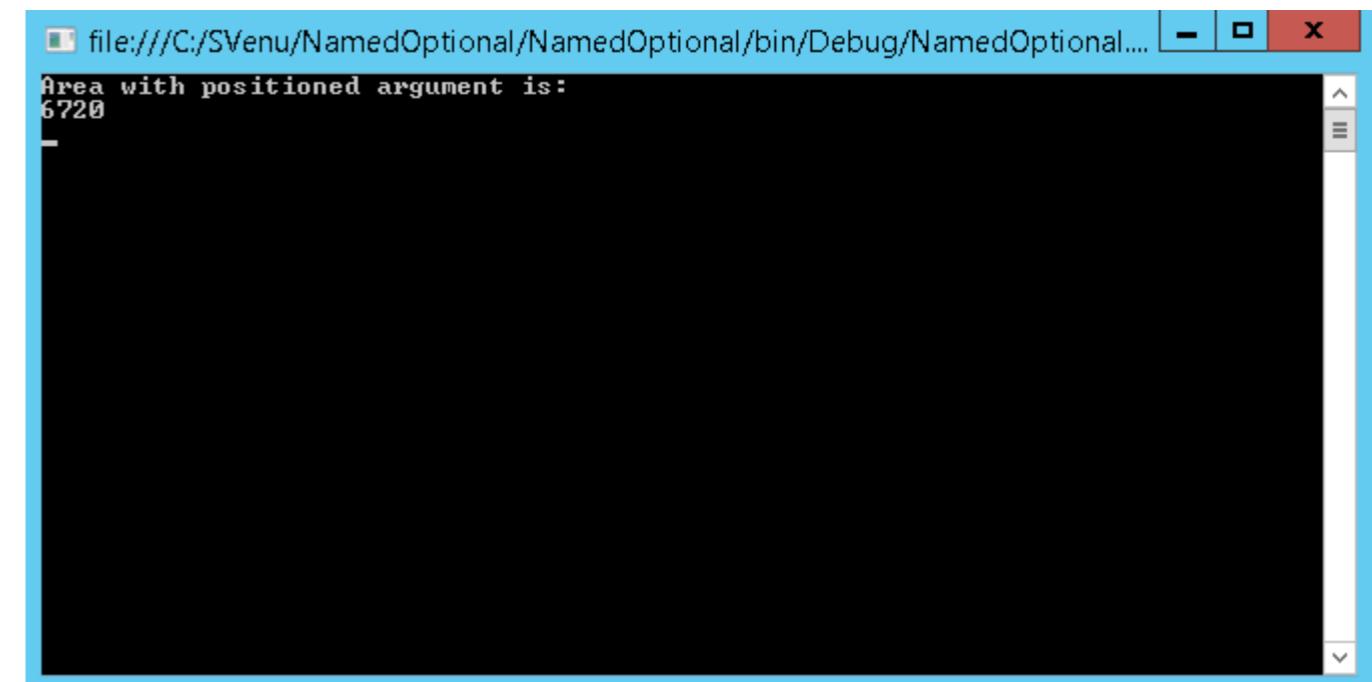
```
Console.WriteLine("使用命名参数和位置参数计算的面积是: ");
area = FindArea(120, width: 56);
Console.WriteLine(area);
Console.Read();
```

在上述示例中，我们将120作为长度传递，并将56作为参数宽度的命名参数传递。

也存在一些限制。我们现在将讨论命名参数的限制。

使用命名参数的限制

命名参数的指定必须出现在所有固定参数指定之后。



Now here it comes the features of a named arguments. Please see the preceding function call.

```
Console.WriteLine("Area with Named argument is: ");
area = FindArea(length: 120, width: 56);
Console.WriteLine(area);
Console.Read();
```

Here we are giving the named arguments in the method call.

```
area = FindArea(length: 120, width: 56);
```

Now if you run this program, you will get the same result. We can give the names vice versa in the method call if we are using the named arguments.

```
Console.WriteLine("Area with Named argument vice versa is: ");
area = FindArea(width: 120, length: 56);
Console.WriteLine(area);
Console.Read();
```

One of the important use of a named argument is, when you use this in your program it improves the readability of your code. It simply says what your argument is meant to be, or what it is?.

You can give the positional arguments too. That means, a combination of both positional argument and named argument.

```
Console.WriteLine("Area with Named argument Positional Argument : ");
area = FindArea(120, width: 56);
Console.WriteLine(area);
Console.Read();
```

In the above example we passed 120 as the length and 56 as a named argument for the parameter width.

There are some limitations too. We will discuss the limitation of a named arguments now.

Limitation of using a Named Argument

Named argument specification must appear after all fixed arguments have been specified.

如果在固定参数之前使用命名参数，将会出现如下编译时错误。

```
...area = FindArea(length:120, ...);
...
}
private static double FindArea(i
{
    try
{
```

struct System.Int32
Represents a 32-bit signed integer.

Error:
Named argument specifications must appear after all fixed arguments have been specified



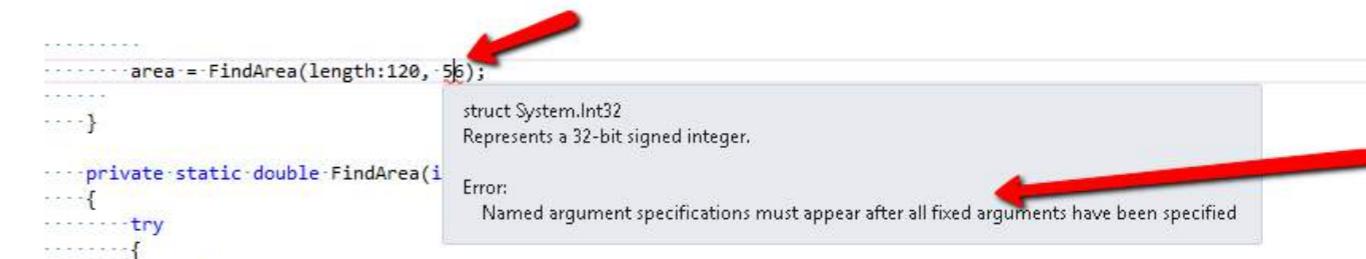
命名参数的指定必须出现在所有固定参数指定之后

If you use a named argument before a fixed argument you will get a compile time error as follows.

```
...area = FindArea(length:120, ...);
...
}
private static double FindArea(i
{
    try
{
```

struct System.Int32
Represents a 32-bit signed integer.

Error:
Named argument specifications must appear after all fixed arguments have been specified



Named argument specification must appear after all fixed arguments have been specified

第51章：数据注解

第51.1节：数据注解基础

数据注解是一种向类或类成员添加更多上下文信息的方式。注解主要分为三类：

- 验证属性：为数据添加验证条件
- 显示属性：指定数据应如何显示给用户
- 建模属性：添加有关使用和与其他类关系的信息

用法

下面是一个示例，其中使用了两个ValidationAttribute和一个DisplayAttribute：

```
类 Kid
{
    [Range(0, 18)] // 年龄不能超过18岁且不能为负数
    public int Age { get; set; }

    [StringLength(MaximumLength = 50, MinimumLength = 3)] // 名字长度不能少于3个字符或超过50个字符
    public string Name { get; set; }

    [DataType(DataType.Date)] // 生日将仅显示为日期（不含时间）
    public DateTime Birthday { get; set; }
}
```

数据注解主要用于如ASP.NET等框架中。例如，在ASP.NET MVC中，当模型被控制器方法接收时，可以使用ModelState.IsValid()来判断接收的模型是否符合所有的ValidationAttribute。DisplayAttribute也用于ASP.NET MVC中，以确定网页上值的显示方式。

第51.2节：创建自定义验证属性

可以通过继承ValidationAttribute基类来创建自定义验证属性，然后根据需要重写虚方法。

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited = false)]
public class NotABananaAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        var inputValue = value as string;
        var isValid = true;

        if (!string.IsNullOrEmpty(inputValue))
        {
            isValid = inputValue.ToUpperInvariant() != "BANANA";
        }

        return isValid;
    }
}
```

该属性可以这样使用：

```
public class Model
```

Chapter 51: Data Annotation

Section 51.1: Data Annotation Basics

Data annotations are a way of adding more contextual information to classes or members of a class. There are three main categories of annotations:

- Validation Attributes: add validation criteria to data
- Display Attributes: specify how the data should be displayed to the user
- Modelling Attributes: add information on usage and relationship with other classes

Usage

Here is an example where two ValidationAttribute and one DisplayAttribute are used:

```
class Kid
{
    [Range(0, 18)] // The age cannot be over 18 and cannot be negative
    public int Age { get; set; }

    [StringLength(MaximumLength = 50, MinimumLength = 3)] // The name cannot be under 3 chars or
    more than 50 chars
    public string Name { get; set; }

    [DataType(DataType.Date)] // The birthday will be displayed as a date only (without the time)
    public DateTime Birthday { get; set; }
}
```

Data annotations are mostly used in frameworks such as ASP.NET. For example, in ASP.NET MVC, when a model is received by a controller method, ModelState.IsValid() can be used to tell if the received model respects all its ValidationAttribute. DisplayAttribute is also used in ASP.NET MVC to determine how to display values on a web page.

Section 51.2: Creating a custom validation attribute

Custom validation attributes can be created by deriving from the ValidationAttribute base class, then overriding `virtual` methods as needed.

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited = false)]
public class NotABananaAttribute : ValidationAttribute
{
    public override bool IsValid(object value)
    {
        var inputValue = value as string;
        var isValid = true;

        if (!string.IsNullOrEmpty(inputValue))
        {
            isValid = inputValue.ToUpperInvariant() != "BANANA";
        }

        return isValid;
    }
}
```

This attribute can then be used like this:

```
public class Model
```

```
{  
    [NotABanana(ErrorMessage = "不允许使用香蕉。")]  
    public string FavoriteFruit { get; set; }  
}
```

第51.3节：手动执行验证属性

大多数情况下，验证属性是在框架（如ASP.NET）内部使用的。这些框架负责执行验证属性。但如果你想手动执行验证属性呢？只需使用验证器类（无需反射）。

验证上下文

任何验证都需要一个上下文来提供有关被验证内容的信息。这可以包括各种信息，例如要验证的对象、某些属性、错误消息中显示的名称等。

```
ValidationContext vc = new ValidationContext(objectToValidate); // 最简单形式的验证  
上下文。它仅包含对被验证对象的引用。
```

上下文创建后，有多种方式进行验证。

验证对象及其所有属性

```
ICollection<ValidationResult> results = new List<ValidationResult>(); // 将包含验证的结果  
  
bool isValid = Validator.TryValidateObject(objectToValidate, vc, results, true); // 使用之前创建的上下文验证  
对象及其属性。  
// 如果所有内容有效，变量 isValid 将为 true  
// 变量 results 包含验证结果
```

验证对象的某个属性

```
ICollection<ValidationResult> results = new List<ValidationResult>(); // 将包含验证的结果  
  
bool isValid = Validator.TryValidateProperty(objectToValidate.PropertyToValidate, vc, results,  
true); // 使用之前创建的上下文验证该属性。  
// 如果所有内容有效，变量 isValid 将为 true  
// 变量 results 包含验证结果
```

更多内容

要了解更多关于手动验证的信息，请参见：

- [ValidationContext 类文档](#)
- [Validator 类文档](#)

第 51.4 节：验证属性

验证属性用于以声明式方式对类或类成员强制执行各种验证规则。所有验证属性都派生自ValidationAttribute基类。

示例：RequiredAttribute

通过ValidationAttribute.Validate方法验证时，如果Name属性为 null 或仅包含空白字符，该属性将返回错误。

```
{  
    [NotABanana(ErrorMessage = "Bananas are not allowed.")]  
    public string FavoriteFruit { get; set; }  
}
```

Section 51.3: Manually Execute Validation Attributes

Most of the times, validation attributes are used inside frameworks (such as ASP.NET). Those frameworks take care of executing the validation attributes. But what if you want to execute validation attributes manually? Just use the Validator class (no reflection needed).

Validation Context

Any validation needs a context to give some information about what is being validated. This can include various information such as the object to be validated, some properties, the name to display in the error message, etc.

```
ValidationContext vc = new ValidationContext(objectToValidate); // The simplest form of validation  
context. It contains only a reference to the object being validated.
```

Once the context is created, there are multiple ways of doing validation.

Validate an Object and All of its Properties

```
ICollection<ValidationResult> results = new List<ValidationResult>(); // Will contain the results  
of the validation  
bool isValid = Validator.TryValidateObject(objectToValidate, vc, results, true); // Validates the  
object and its properties using the previously created context.  
// The variable isValid will be true if everything is valid  
// The results variable contains the results of the validation
```

Validate a Property of an Object

```
ICollection<ValidationResult> results = new List<ValidationResult>(); // Will contain the results  
of the validation  
bool isValid = Validator.TryValidateProperty(objectToValidate.PropertyToValidate, vc, results,  
true); // Validates the property using the previously created context.  
// The variable isValid will be true if everything is valid  
// The results variable contains the results of the validation
```

And More

To learn more about manual validation see:

- [ValidationContext Class Documentation](#)
- [Validator Class Documentation](#)

Section 51.4: Validation Attributes

Validation attributes are used to enforce various validation rules in a declarative fashion on classes or class members. All validation attributes derive from the [ValidationAttribute](#) base class.

Example: RequiredAttribute

When validated through the ValidationAttribute.Validate method, this attribute will return an error if the Name property is null or contains only whitespace.

```
public class ContactModel
{
    [Required(ErrorMessage = "请提供一个名称。")]
    public string Name { get; set; }
}
```

示例：StringLengthAttribute

StringLengthAttribute 用于验证字符串是否小于字符串的最大长度。它可以选择性地指定最小长度。两个值均包含在内。

```
public class ContactModel
{
    [StringLength(20, MinimumLength = 5, ErrorMessage = "名称必须在五到二十个
    字符之间。")]
    public string Name { get; set; }
}
```

示例：RangeAttribute

RangeAttribute 用于指定数值字段的最大值和最小值。

```
public class Model
{
    [Range(0.01, 100.00,ErrorMessage = "价格必须在0.01到100.00之间")]
    public decimal Price { get; set; }
}
```

示例：CustomValidationAttribute

CustomValidationAttribute 类允许调用自定义的static方法进行验证。自定义方法必须是static ValidationResult [MethodName] (object input)。

```
public class Model
{
    [CustomValidation(typeof(MyCustomValidation), "IsNotAnApple")]
    public string FavoriteFruit { get; set; }
}
```

方法声明：

```
public static class MyCustomValidation
{
    public static ValidationResult IsNotAnApple(object input)
    {
        var result = ValidationResult.Success;

        if (input?.ToString()?.ToUpperInvariant() == "APPLE")
        {
            result = new ValidationResult("不允许使用苹果。");
        }

        return result;
    }
}
```

```
public class ContactModel
{
    [Required(ErrorMessage = "Please provide a name.")]
    public string Name { get; set; }
}
```

Example: StringLengthAttribute

The StringLengthAttribute validates if a string is less than the maximum length of a string. It can optionally specify a minimum length. Both values are inclusive.

```
public class ContactModel
{
    [StringLength(20, MinimumLength = 5, ErrorMessage = "A name must be between five and twenty
    characters.")]
    public string Name { get; set; }
}
```

Example: RangeAttribute

The RangeAttribute gives the maximum and minimum value for a numeric field.

```
public class Model
{
    [Range(0.01, 100.00,ErrorMessage = "Price must be between 0.01 and 100.00")]
    public decimal Price { get; set; }
}
```

Example: CustomValidationAttribute

The CustomValidationAttribute class allows a custom `static` method to be invoked for validation. The custom method must be `static ValidationResult [MethodName] (object input)`.

```
public class Model
{
    [CustomValidation(typeof(MyCustomValidation), "IsNotAnApple")]
    public string FavoriteFruit { get; set; }
}
```

Method declaration:

```
public static class MyCustomValidation
{
    public static ValidationResult IsNotAnApple(object input)
    {
        var result = ValidationResult.Success;

        if (input?.ToString()?.ToUpperInvariant() == "APPLE")
        {
            result = new ValidationResult("Apples are not allowed.");
        }

        return result;
    }
}
```

第51.5节：EditableAttribute (数据建模属性)

EditableAttribute 设置用户是否可以更改类属性的值。

```
public class Employee
{
    [Editable(false)]
    public string FirstName { get; set; }
}
```

XAML 应用中的简单使用示例

```
<Window x:Class="WpfApplication.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:wpfApplication="clr-namespace:WpfApplication"
        Height="70" Width="360" Title="显示名称示例">

    <Window.Resources>
        <wpfApplication:EditableConverter x:Key="EditableConverter"/>
    </Window.Resources>

    <StackPanel Margin="5">
        <!-- 文本框文本 (FirstName 属性值) -->
        <!-- 文本框是否启用 (Editable 属性) -->
        <TextBox Text="{Binding Employee.FirstName, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}"
                 IsEnabled="{Binding Employee, Converter={StaticResource EditableConverter},
ConverterParameter=FirstName}"/>
    </StackPanel>

</Window >
```

```
namespace WpfApplication
{
    /// <summary>
    /// MainWindow.xaml 的交互逻辑
    /// </summary>
    public partial class MainWindow : Window
    {
        private Employee _employee = new Employee() { FirstName = "此项不可编辑" };

        public MainWindow()
        {
            InitializeComponent();
            DataContext = this;
        }

        public Employee Employee
        {
            get { return _employee; }
            set { _employee = value; }
        }
    }
}
```

```
namespace WpfApplication
{
    public class EditableConverter : IValueConverter
```

Section 51.5: EditableAttribute (data modeling attribute)

EditableAttribute sets whether users should be able to change the value of the class property.

```
public class Employee
{
    [Editable(false)]
    public string FirstName { get; set; }
}
```

Simple usage example in XAML application

```
<Window x:Class="WpfApplication.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:wpfApplication="clr-namespace:WpfApplication"
        Height="70" Width="360" Title="Display name example">

    <Window.Resources>
        <wpfApplication:EditableConverter x:Key="EditableConverter"/>
    </Window.Resources>

    <StackPanel Margin="5">
        <!-- TextBox Text (FirstName property value) -->
        <!-- TextBox IsEnabled (Editable attribute) -->
        <TextBox Text="{Binding Employee.FirstName, Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged}"
                 IsEnabled="{Binding Employee, Converter={StaticResource EditableConverter},
ConverterParameter=FirstName}"/>
    </StackPanel>

</Window >
```

```
namespace WpfApplication
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private Employee _employee = new Employee() { FirstName = "This is not editable" };

        public MainWindow()
        {
            InitializeComponent();
            DataContext = this;
        }

        public Employee Employee
        {
            get { return _employee; }
            set { _employee = value; }
        }
    }
}
```

```
namespace WpfApplication
{
    public class EditableConverter : IValueConverter
```

```

{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        // 返回给定实例属性的可编辑属性值,
        // 如果未找到则默认为 true
        var attribute = value.GetType()
            .GetProperty(parameter.ToString())
            .GetCustomAttributes(false)
            .OfType<EditableAttribute>()
            .FirstOrDefault();

        return attribute != null ? attribute.AllowEdit : true;
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}

```



```

{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        // return editable attribute's value for given instance property,
        // defaults to true if not found
        var attribute = value.GetType()
            .GetProperty(parameter.ToString())
            .GetCustomAttributes(false)
            .OfType<EditableAttribute>()
            .FirstOrDefault();

        return attribute != null ? attribute.AllowEdit : true;
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}

```



第52章：关键字

关键字是预定义的、保留的标识符，对编译器具有特殊意义。没有 @ 前缀，它们不能作为程序中的标识符使用。例如，@if 是合法的标识符，但不是关键字 if。

第52.1节：as

as 关键字是一个类似于 cast 的操作符。如果无法进行强制转换，使用 as 会返回 null，而不是抛出 InvalidCastException 异常。

表达式 as 类型 等价于 表达式 is 类型 ? (类型)表达式 : (类型)null，但需要注意的是，as 仅适用于引用转换、可空转换和装箱转换。不支持用户自定义转换；必须使用常规强制转换。

对于上述展开，编译器生成的代码使得表达式只会被计算一次，并且只使用一次动态类型检查（不同于上面示例中的两次）。

as 在期望参数支持多种类型时非常有用。具体来说，它为用户提供了多种选项——而不是在转换前用 is 检查每种可能性，或者直接转换后捕获异常。最佳实践是在转换/检查对象时使用 'as'，这样只会产生一次拆箱开销。使用 is 先检查再转换会产生两次拆箱开销。

如果期望参数是特定类型的实例，建议使用常规转换，因为这样对读者来说目的更明确。

由于调用 as 可能返回 null，务必检查结果以避免 NullReferenceException。

示例用法

```
object something = "Hello";
Console.WriteLine(something as string);      //Hello
Console.Writeline(something as Nullable<int>); //null
Console.WriteLine(something as int?);         //null

//这段代码无法编译：
//目标类型必须是引用类型（或可空值类型）
Console.WriteLine(something as int);
```

[.NET Fiddle 在线演示](#)

不使用 as 的等效示例：

```
Console.WriteLine(something is string ? (string)something : (string)null);
```

这在重写自定义类中的 Equals 函数时非常有用。

```
class MyCustomClass
{
    public override bool Equals(object obj)
    {
        MyCustomClass customObject = obj as MyCustomClass;
        // 如果它是 null，可能是真的 null
        // 或者它可能是不同的类型
    }
}
```

Chapter 52: Keywords

[Keywords](#) are predefined, reserved identifiers with special meaning to the compiler. They cannot be used as identifiers in your program without the @ prefix. For example @if is a legal identifier but not the keyword if.

Section 52.1: as

The as keyword is an operator similar to a cast. If a cast is not possible, using as produces `null` rather than resulting in an `InvalidCastException`.

`expression as type` is equivalent to `expression is type ? (type)expression : (type)null` with the caveat that as is only valid on reference conversions, nullable conversions, and boxing conversions. User-defined conversions are *not* supported; a regular cast must be used instead.

For the expansion above, the compiler generates code such that expression will only be evaluated once and use single dynamic type check (unlike the two in the sample above).

as can be useful when expecting an argument to facilitate several types. Specifically it grants the user multiple options - rather than checking every possibility with is before casting, or just casting and catching exceptions. It is best practice to use 'as' when casting/checking an object which will cause only one unboxing penalty. Using is to check, then casting will cause two unboxing penalties.

If an argument is expected to be an instance of a specific type, a regular cast is preferred as its purpose is more clear to the reader.

Because a call to as may produce `null`, always check the result to avoid a `NullReferenceException`.

Example usage

```
object something = "Hello";
Console.WriteLine(something as string);      //Hello
Console.Writeline(something as Nullable<int>); //null
Console.WriteLine(something as int?);         //null

//This does NOT compile:
//destination type must be a reference type (or a nullable value type)
Console.WriteLine(something as int);
```

[Live Demo on .NET Fiddle](#)

Equivalent example without using as:

```
Console.WriteLine(something is string ? (string)something : (string)null);
```

This is useful when overriding the Equals function in custom classes.

```
class MyCustomClass
{
    public override bool Equals(object obj)
    {
        MyCustomClass customObject = obj as MyCustomClass;
        // if it is null it may be really null
        // or it may be of a different type
    }
}
```

```

if (Object.ReferenceEquals(null, customObject))
{
    // 如果它是 null, 则不等于此实例。
    return false;
}

// 类特定的其他相等性控制
}

```

第 52.2 节：goto

`goto` 可用于跳转到代码中的特定行，由标签指定。

`goto` 作为：

标签：

```

void InfiniteHello()
{
    sayHello:
    Console.WriteLine("Hello!");
    goto sayHello;
}

```

[.NET Fiddle 在线演示](#)

`Case` 语句：

```

enum Permissions { 读取, 写入 };

switch (GetRequestedPermission())
{
    case Permissions.读取:
        授予读取权限();
        break;

    case Permissions.写入:
        授予写入权限();
        goto case Permissions.读取; //拥有写入权限的人也获得读取权限
}

```

[.NET Fiddle 在线演示](#)

这在执行 `switch` 语句中的多重行为时特别有用，因为 C# 不支持 [case 块的贯穿执行 \(fall-through\)](#)

```

if (Object.ReferenceEquals(null, customObject))
{
    // If it is null then it is not equal to this instance.
    return false;
}

// Other equality controls specific to class
}

```

Section 52.2: goto

`goto` can be used to jump to a specific line inside the code, specified by a label.

`goto as a:`

`Label:`

```

void InfiniteHello()
{
    sayHello:
    Console.WriteLine("Hello!");
    goto sayHello;
}

```

[Live Demo on .NET Fiddle](#)

`Case statement:`

```

enum Permissions { Read, Write };

switch (GetRequestedPermission())
{
    case Permissions.Read:
        GrantReadAccess();
        break;

    case Permissions.Write:
        GrantWriteAccess();
        goto case Permissions.Read; //People with write access also get read
}

```

[Live Demo on .NET Fiddle](#)

This is particularly useful in executing multiple behaviors in a switch statement, as C# does not support [fall-through](#) `case` blocks.

`Exception Retry`

```

var exCount = 0;
retry:
try
{
    //执行操作
}
catch (IOException)
{
    exCount++;
    if (exCount < 3)
    {
        Thread.Sleep(100);
    }
}

```

`异常重试`

```

var exCount = 0;
retry:
try
{
    //执行操作
}
catch (IOException)
{
    exCount++;
    if (exCount < 3)
    {
        Thread.Sleep(100);
    }
}

```

```
    goto retry;
}
throw;
}
```

[.NET Fiddle 在线演示](#)

与许多语言类似，除以下情况外，不建议使用goto关键字。

C#中有效的goto用法：

- switch语句中的贯穿情况。
- 多级跳出。通常可以使用LINQ代替，但性能通常较差。
- 处理未封装的底层对象时的资源释放。在C#中，底层对象通常应封装在单独的类中。
- 有限状态机，例如解析器；由编译器生成的异步/等待状态机内部使用。

```
    goto retry;
}
throw;
}
```

[Live Demo on .NET Fiddle](#)

Similar to many languages, use of goto keyword is discouraged except the cases below.

[Valid usages of goto which apply to C#:](#)

- Fall-through case in switch statement.
- Multi-level break. LINQ can often be used instead, but it usually has worse performance.
- Resource deallocation when working with unwrapped low-level objects. In C#, low-level objects should usually be wrapped in separate classes.
- Finite state machines, for example, parsers; used internally by compiler generated async/await state machines.

第52.3节：volatile

向字段添加volatile关键字表示该字段的值可能被多个独立线程更改。volatile关键字的主要目的是防止编译器假设只有单线程访问而进行的优化。使用volatile确保字段的值是最新的值，并且该值不会像非volatile值那样被缓存。

将每个可能被多个线程使用的变量标记为volatile是良好实践，以防止由于幕后优化导致的意外行为。考虑以下代码块：

```
public class Example
{
    public int x;

    public void DoStuff()
    {
        x = 5;

        // 编译器会将此优化为 y = 15
        var y = x + 10;

        /* x 的值始终是当前值，但 y 始终是"15" */
        Debug.WriteLine("x = " + x + ", y = " + y);
    }
}
```

在上述代码块中，编译器读取语句 `x = 5` 和 `y = x + 10`，并确定 `y` 的值最终总是 15。因此，它会将最后一条语句优化为 `y = 15`。然而，变量 `x` 实际上是一个public 字段，且 `x` 的值可能会在运行时被另一个线程单独修改。现在考虑这个修改后的代码块。请注意，字段 `x` 现在声明为 volatile。

```
public class Example
{
    public volatile int x;

    public void DoStuff()
```

Section 52.3: volatile

Adding the `volatile` keyword to a field indicates to the compiler that the field's value may be changed by multiple separate threads. The primary purpose of the `volatile` keyword is to prevent compiler optimizations that assume only single-threaded access. Using `volatile` ensures that the value of the field is the most recent value that is available, and the value is not subject to the caching that non-volatile values are.

It is good practice to mark *every variable* that may be used by multiple threads as `volatile` to prevent unexpected behavior due to behind-the-scenes optimizations. Consider the following code block:

```
public class Example
{
    public int x;

    public void DoStuff()
    {
        x = 5;

        // the compiler will optimize this to y = 15
        var y = x + 10;

        /* the value of x will always be the current value, but y will always be "15" */
        Debug.WriteLine("x = " + x + ", y = " + y);
    }
}
```

In the above code-block, the compiler reads the statements `x = 5` and `y = x + 10` and determines that the value of `y` will always end up as 15. Thus, it will optimize the last statement as `y = 15`. However, the variable `x` is in fact a `public` field and the value of `x` may be modified at runtime through a different thread acting on this field separately. Now consider this modified code-block. Do note that the field `x` is now declared as `volatile`.

```
public class Example
{
    public volatile int x;

    public void DoStuff()
```

```

{
    x = 5;

    // 编译器不再优化此语句
    var y = x + 10;

    /* x 和 y 的值始终是正确的 */
    Debug.WriteLine("x = " + x + ", y = " + y);
}

```

现在，编译器会查找字段 x 的 read 用法，并确保始终获取该字段的当前值。这保证了即使多个线程同时读写该字段，也能始终获取 x 的当前值。

`volatile` 只能用于 `class` 或 `struct` 中的字段。以下用法 is not 有效：

```
public void MyMethod() { volatile int x; }
```

`volatile` 只能应用于以下类型的字段：

- 引用类型或已知为引用类型的泛型类型参数
- 原始类型，如 `sbyte`、`byte`、`short`、`ushort`、`int`、`uint`、`char`、`float` 和 `bool`
- 基于 `byte`、`sbyte`、`short`、`ushort`、`int` 或 `uint` 的枚举类型
- `IntPtr` 和 `UIntPtr`

备注：

- `volatile` 修饰符通常用于多个线程访问且未使用锁语句序列化访问的字段。
- `volatile` 关键字可以应用于引用类型的字段 `volatile` 关键字不会使在 32 位平台上对 64 位原始类型的操作变为原子操作。在这些平台上，仍必须使用诸如 `Interlocked.Read` 和 `Interlocked.Exchange` 之类的互锁操作以保证多线程访问的安全。

第 52.4 节：checked, unchecked

`checked` 和 `unchecked` 关键字定义了操作如何处理数学溢出。在 `checked` 和 `unchecked` 关键字的上下文中，“溢出”是指整数算术运算结果的数值大小超过目标数据类型所能表示的范围。

当溢出发生在 `checked` 块内（或编译器设置为全局使用 `checked` 算术时），会抛出异常以警告不期望的行为。与此同时，在 `unchecked` 块中，溢出是静默的：不会抛出异常，数值将简单地绕回到相反的边界。这可能导致细微且难以发现的错误。

由于大多数算术操作是在不会导致溢出的数值上进行的，因此大多数情况下无需显式定义 `checked` 块。在对可能导致溢出的无界输入进行算术运算时需要小心，例如在递归函数中进行算术运算或处理用户输入时。

无论是`checked`还是`unchecked`都不会影响浮点数算术运算。

当一个代码块或表达式被声明为`unchecked`时，内部的任何算术运算允许溢出而不会引发错误。一个需要这种行为的例子是校验和的计算，其中

```

{
    x = 5;

    // the compiler no longer optimizes this statement
    var y = x + 10;

    /* the value of x and y will always be the correct values */
    Debug.WriteLine("x = " + x + ", y = " + y);
}

```

Now, the compiler looks for *read* usages of the field x and ensures that the current value of the field is always retrieved. This ensures that even if multiple threads are reading and writing to this field, the current value of x is always retrieved.

`volatile` can only be used on fields within `classes` or `structs`. The following is not valid:

```
public void MyMethod() { volatile int x; }
```

`volatile` can only be applied to fields of following types:

- reference types or generic type parameters known to be reference types
- primitive types such as `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float`, and `bool`
- enums types based on `byte`, `sbyte`, `short`, `ushort`, `int` or `uint`
- `IntPtr` and `UIntPtr`

Remarks:

- The `volatile` modifier is usually used for a field that is accessed by multiple threads without using the lock statement to serialize access.
- The `volatile` keyword can be applied to fields of reference types
- The `volatile` keyword will not make operating on 64-bit primitives on a 32-bit platform atomic. `Interlocked` operations such as `Interlocked.Read` and `Interlocked.Exchange` must still be used for safe multi-threaded access on these platforms.

Section 52.4: checked, unchecked

The `checked` and `unchecked` keywords define how operations handle mathematical overflow. "Overflow" in the context of the `checked` and `unchecked` keywords is when an integer arithmetic operation results in a value which is greater in magnitude than the target data type can represent.

When overflow occurs within a `checked` block (or when the compiler is set to globally use checked arithmetic), an exception is thrown to warn of undesired behavior. Meanwhile, in an `unchecked` block, overflow is silent: no exceptions are thrown, and the value will simply wrap around to the opposite boundary. This can lead to subtle, hard to find bugs.

Since most arithmetic operations are done on values that are not large or small enough to overflow, most of the time, there is no need to explicitly define a block as `checked`. Care needs to be taken when doing arithmetic on unbounded input that may cause overflow, for example when doing arithmetic in recursive functions or while taking user input.

Neither `checked` nor `unchecked` affect floating point arithmetic operations.

When a block or expression is declared as `unchecked`, any arithmetic operations inside it are allowed to overflow without causing an error. An example where this behavior is desired would be the calculation of a checksum, where

计算过程中允许值“回绕”：

```
byte Checksum(byte[] data) {
    byte result = 0;
    for (int i = 0; i < data.Length; i++) {
        result = unchecked(result + data[i]); // unchecked 表达式
    }
    return result;
}
```

unchecked 最常见的用途之一是实现对object.GetHashCode()的自定义重写，这是一种校验和。你可以在这个问题的回答中看到该关键字的用法：[What is the best algorithm for an overridden System.Object.GetHashCode?](#)

当一个代码块或表达式被声明为checked时，任何导致溢出的算术运算都会抛出OverflowException异常。

```
int SafeSum(int x, int y) {
    checked { // checked 块
        return x + y;
    }
}
```

checked 和 unchecked 都可以以代码块和表达式的形式存在。

选中和未选中的代码块不会影响被调用的方法，只影响当前方法中直接调用的操作符。

例如，`Enum.ToObject()`，`Convert.ToInt32()`，以及用户定义的运算符不受自定义的checked/unchecked上下文影响。

注意：默认的溢出行为（checked与unchecked）可以在项目属性中更改，或通过/checked[+/-]命令行开关设置。通常调试版本默认使用checked操作，发布版本默认使用unchecked操作。此时，checked和unchecked关键字仅在默认方式不适用且需要显式行为以确保正确性时使用。

第52.5节：virtual、override、new

virtual和override

virtual关键字允许方法、属性、索引器或事件被派生类重写，并呈现多态行为。（在C#中成员默认是非虚的）

```
public class BaseClass
{
    public virtual void Foo()
    {
        Console.WriteLine("来自BaseClass的Foo");
    }
}
```

为了重写成员，派生类中使用override关键字。（注意成员的签名必须完全相同）

```
public class DerivedClass: BaseClass
{
    public override void Foo()
    {
    }
```

the value is allowed to "wrap around" during calculation:

```
byte Checksum(byte[] data) {
    byte result = 0;
    for (int i = 0; i < data.Length; i++) {
        result = unchecked(result + data[i]); // unchecked expression
    }
    return result;
}
```

One of the most common uses for `unchecked` is implementing a custom override for `object.GetHashCode()`, a type of checksum. You can see the keyword's use in the answers to this question: [What is the best algorithm for an overridden System.Object.GetHashCode?](#).

When a block or expression is declared as `checked`, any arithmetic operation that causes an overflow results in an `OverflowException` being thrown.

```
int SafeSum(int x, int y) {
    checked { // checked block
        return x + y;
    }
}
```

Both checked and unchecked may be in block and expression form.

Checked and unchecked blocks do not affect called methods, only operators called directly in the current method. For example, `Enum.ToObject()`, `Convert.ToInt32()`, and user-defined operators are not affected by custom checked/unchecked contexts.

Note: The default overflow default behavior (checked vs. unchecked) may be changed in the **Project Properties** or through the /checked[+|-] command line switch. It is common to default to checked operations for debug builds and unchecked for release builds. The `checked` and `unchecked` keywords would then be used only where a default approach does not apply and you need an explicit behavior to ensure correctness.

Section 52.5: virtual, override, new

virtual and override

The `virtual` keyword allows a method, property, indexer or event to be overridden by derived classes and present polymorphic behavior. (Members are non-virtual by default in C#)

```
public class BaseClass
{
    public virtual void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}
```

In order to override a member, the `override` keyword is used in the derived classes. (Note the signature of the members must be identical)

```
public class DerivedClass: BaseClass
{
    public override void Foo()
    {
    }
```

```

Console.WriteLine("Foo from DerivedClass");
}
}

```

虚拟成员的多态行为意味着在调用时，实际执行的成员是在运行时而非编译时确定的。将执行的是最派生类中该对象实例所重写的成员。

简而言之，对象可以在编译时声明为BaseClass类型，但如果在运行时它是DerivedClass的实例，则会执行被重写的成员：

```

BaseClass obj1 = new BaseClass();
obj1.Foo(); //输出 "Foo from BaseClass"

```

```

obj1 = new DerivedClass();
obj1.Foo(); //输出 "Foo from DerivedClass"

```

重写方法是可选的：

```

public class SecondDerivedClass: DerivedClass {}

var obj1 = new SecondDerivedClass();
obj1.Foo(); //输出 "Foo from DerivedClass"

```

新

由于只有被定义为`virtual`的成员才是可重写和多态的，派生类重新定义非虚拟成员可能会导致意想不到的结果。

```

public class BaseClass
{
    public void Foo()
    {
        Console.WriteLine("来自BaseClass的Foo");
    }
}

public class DerivedClass: BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}

BaseClass obj1 = new BaseClass();
obj1.Foo(); //输出 "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); // 也输出"BaseClass的Foo ! "

```

当发生这种情况时，执行的成员总是在编译时根据对象的类型确定。

- 如果对象被声明为BaseClass类型（即使运行时是派生类），则执行BaseClass的方法
- 如果对象被声明为DerivedClass类型，则执行DerivedClass的方法。

这通常是一个意外（当在基类中添加成员后，派生类中已经添加了相同的成员），

```

Console.WriteLine("Foo from DerivedClass");
}
}

```

The polymorphic behavior of virtual members means that when invoked, the actual member being executed is determined at runtime instead of at compile time. The overriding member in the most derived class the particular object is an instance of will be the one executed.

In short, object can be declared of type BaseClass at compile time but if at runtime it is an instance of DerivedClass then the overridden member will be executed:

```

BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

```

```

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from DerivedClass"

```

Overriding a method is optional:

```

public class SecondDerivedClass: DerivedClass {}

var obj1 = new SecondDerivedClass();
obj1.Foo(); //Outputs "Foo from DerivedClass"

```

new

Since only members defined as `virtual` are overridable and polymorphic, a derived class redefining a non `virtual` member might lead to unexpected results.

```

public class BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}

public class DerivedClass: BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}

BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from BaseClass" too!

```

When this happens, the member executed is always determined at compile time based on the type of the object.

- If the object is declared of type BaseClass (even if at runtime is of a derived class) then the method of BaseClass is executed
- If the object is declared of type DerivedClass then the method of DerivedClass is executed.

This is usually an accident (When a member is added to the base type after an identical one was added to the

在这些情况下会生成编译器警告CS0108。

如果这是故意的，那么使用new关键字来抑制编译器警告（并告知其他开发者你的意图！）。行为保持不变，new关键字只是抑制编译器警告。

```
public class BaseClass
{
    public void Foo()
    {
        Console.WriteLine("来自BaseClass的Foo");
    }
}

public class DerivedClass: BaseClass
{
    public new void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}

BaseClass obj1 = new BaseClass();
obj1.Foo(); //输出 "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); // 也输出"BaseClass 的 Foo"!
```

override 的使用不是可选的

与 C++ 不同，override关键字的使用不是可选的：

```
public class A
{
    public virtual void Foo()
    {
    }
}

public class B : A
{
    public void Foo() // 生成 CS0108
    {
    }
}
```

上述示例也会导致警告CS0108，因为B.Foo()并未自动覆盖A.Foo()。当意图覆盖基类并产生多态行为时，添加override；当你想要非多态行为并使用静态类型解析调用时，添加new。后者应谨慎使用，因为它可能导致严重混淆。

以下代码甚至会导致错误：

```
public class A
{
    public void Foo()
    {
    }
}

public class B : A
```

derived type) and a compiler warning **CS0108** is generated in those scenarios.

If it was intentional, then the `new` keyword is used to suppress the compiler warning (And inform other developers of your intentions!). the behavior remains the same, the `new` keyword just suppresses the compiler warning.

```
public class BaseClass
{
    public void Foo()
    {
        Console.WriteLine("Foo from BaseClass");
    }
}

public class DerivedClass: BaseClass
{
    public new void Foo()
    {
        Console.WriteLine("Foo from DerivedClass");
    }
}

BaseClass obj1 = new BaseClass();
obj1.Foo(); //Outputs "Foo from BaseClass"

obj1 = new DerivedClass();
obj1.Foo(); //Outputs "Foo from BaseClass" too!
```

The usage of override is not optional

Unlike in C++, the usage of the `override` keyword is *not* optional:

```
public class A
{
    public virtual void Foo()
    {
    }
}

public class B : A
{
    public void Foo() // Generates CS0108
    {
    }
}
```

The above example also causes warning **CS0108**, because `B.Foo()` is not automatically overriding `A.Foo()`. Add `override` when the intention is to override the base class and cause polymorphic behavior, add `new` when you want non-polymorphic behavior and resolve the call using the static type. The latter should be used with caution, as it may cause severe confusion.

The following code even results in an error:

```
public class A
{
    public void Foo()
    {
    }
}

public class B : A
```

```

{
    public override void Foo() // 错误：没有可重写的内容
    {
    }
}

```

派生类可以引入多态性

下面的代码是完全有效的（尽管很少见）：

```

public class A
{
    public void Foo()
    {
        Console.WriteLine("A");
    }
}

public class B : A
{
    public new virtual void Foo()
    {
        Console.WriteLine("B");
    }
}

```

现在所有静态引用为 B（及其派生类）的对象都使用多态性来解析 Foo()，而 A 的引用则使用 A.Foo()。

```

A a = new A();
a.Foo(); // 输出 "A";
a = new B();
a.Foo(); // 输出 "A";
B b = new B();
b.Foo(); // 输出 "B";

```

虚方法不能是私有的

C# 编译器严格防止无意义的构造。标记为 virtual 的方法不能是私有的。因为私有方法不能被派生类型看到，所以也无法被重写。以下代码无法编译：

```

public class A
{
    private virtual void Foo() // 错误：虚方法不能是私有的
    {
    }
}

```

第52.6节：stackalloc

stackalloc 关键字在栈上创建一块内存区域，并返回指向该内存起始位置的指针。栈分配的内存会在其创建的作用域退出时自动释放。

```

//分配1024字节。这将返回指向第一个字节的指针。
byte* ptr = stackalloc byte[1024];

//赋值...
ptr[0] = 109;
ptr[1] = 13;

```

```

{
    public override void Foo() // Error: Nothing to override
    {
    }
}

```

Derived classes can introduce polymorphism

The following code is perfectly valid (although rare):

```

public class A
{
    public void Foo()
    {
        Console.WriteLine("A");
    }
}

public class B : A
{
    public new virtual void Foo()
    {
        Console.WriteLine("B");
    }
}

```

Now all objects with a static reference of B (and its derivatives) use polymorphism to resolve Foo(), while references of A use A.Foo().

```

A a = new A();
a.Foo(); // Prints "A";
a = new B();
a.Foo(); // Prints "A";
B b = new B();
b.Foo(); // Prints "B";

```

Virtual methods cannot be private

The C# compiler is strict in preventing senseless constructs. Methods marked as `virtual` cannot be private. Because a private method cannot be seen from a derived type, it couldn't be overwritten either. This fails to compile:

```

public class A
{
    private virtual void Foo() // Error: virtual methods cannot be private
    {
    }
}

```

Section 52.6: stackalloc

The `stackalloc` keyword creates a region of memory on the stack and returns a pointer to the start of that memory. Stack allocated memory is automatically removed when the scope it was created in is exited.

```

//Allocate 1024 bytes. This returns a pointer to the first byte.
byte* ptr = stackalloc byte[1024];

//Assign some values...
ptr[0] = 109;
ptr[1] = 13;

```

```
ptr[2] = 232;
```

...

在不安全的上下文中使用。

与 C# 中所有指针一样，读取和赋值时没有边界检查。读取超出分配内存范围的内容将产生不可预测的结果——可能访问内存中的任意位置，也可能导致访问冲突异常。

```
//分配 1 字节
byte* ptr = stackalloc byte[1];

//不可预测的结果...
ptr[10] = 1;
ptr[-1] = 2;
```

栈分配的内存会在其创建的作用域退出时自动释放。这意味着你绝不应该返回用 `stackalloc` 创建的内存，或将其存储超出该作用域的生命周期。

```
unsafe IntPtr Leak() {
    //在线上分配一些内存
    var ptr = stackalloc byte[1024];

    //返回指向该内存的指针 (这将退出"Leak"的作用域)
    return new IntPtr(ptr);
}

unsafe void Bad() {
    //ptr 现在一个无效指针，任何使用它的方式都会有//不可预测的结果。这与访问指针边界之外的内容完全相同。
    var ptr = Leak();
}
```

`stackalloc` 只能在声明和初始化变量时使用。以下用法无效：

```
byte* ptr;
...
ptr = stackalloc byte[1024];
```

备注：

`stackalloc` 应仅用于性能优化（无论是计算还是互操作）。这是因为

- 垃圾回收器不需要介入，因为内存分配在栈上而非堆上——变量一旦超出作用域，内存即被释放
- 在栈上分配内存比在堆上分配更快
- 由于数据的局部性，增加了CPU缓存命中机会

第52.7节：break语句

在循环（`for`、`foreach`、`do`、`while`）中，`break`语句中止最内层循环的执行，并返回到其后的代码。它也可以与 `yield` 一起使用，表示迭代器已结束。

```
for (var i = 0; i < 10; i++)
{
```

```
ptr[2] = 232;
```

...

Used in an unsafe context.

As with all pointers in C# there is no bounds checking on reads and assignments. Reading beyond the bounds of the allocated memory will have unpredictable results - it may access some arbitrary location within memory or it may cause an access violation exception.

```
//Allocate 1 byte
byte* ptr = stackalloc byte[1];

//Unpredictable results...
ptr[10] = 1;
ptr[-1] = 2;
```

Stack allocated memory is automatically removed when the scope it was created in is exited. This means that you should never return the memory created with `stackalloc` or store it beyond the lifetime of the scope.

```
unsafe IntPtr Leak() {
    //Allocate some memory on the stack
    var ptr = stackalloc byte[1024];

    //Return a pointer to that memory (this exits the scope of "Leak")
    return new IntPtr(ptr);
}

unsafe void Bad() {
    //ptr is now an invalid pointer, using it in any way will have
    //unpredictable results. This is exactly the same as accessing beyond
    //the bounds of the pointer.
    var ptr = Leak();
}
```

`stackalloc` 只能在声明和初始化变量时使用。以下用法无效：

```
byte* ptr;
...
ptr = stackalloc byte[1024];
```

Remarks:

`stackalloc` 应仅用于性能优化（无论是计算还是互操作）。这是由于事实：

- 垃圾回收器不需要介入，因为内存分配在栈上而非堆上——内存即被释放
- 在栈上分配内存比在堆上分配更快
- 增加了CPU缓存命中机会

Section 52.7: break

In a loop (`for`, `foreach`, `do`, `while`) the `break` statement aborts the execution of the innermost loop and returns to the code after it. Also it can be used with `yield` in which it specifies that an iterator has come to an end.

```
for (var i = 0; i < 10; i++)
{
```

```

if (i == 5)
{
    break;
}
Console.WriteLine("这将只显示5次，因为break会停止循环。");
}

```

[.NET Fiddle 在线演示](#)

```

foreach (var stuff in stuffCollection)
{
    if (stuff.SomeStringProp == null)
        break;
    // 如果任何"stuff"的stuff.SomeStringProp为null，循环将被中止。
    Console.WriteLine(stuff.SomeStringProp);
}

```

break语句也用于switch-case结构中，用于跳出某个case或default段。

```

switch(a)
{
    case 5:
        Console.WriteLine("a是5！");
        break;

    default:
        Console.WriteLine("a是其他值！");
        break;
}

```

在switch语句中，每个case语句末尾都必须使用'break'关键字。这与某些允许“贯穿”到下一个case语句的语言不同。解决方法包括使用'goto'语句或顺序堆叠'case'语句。

以下代码将输出数字0、1、2、...、9，最后一行不会被执行。yield break表示函数的结束（不仅仅是循环）。

```

public static IEnumerable<int> GetNumbers()
{
    int i = 0;
    while (true) {
        if (i < 10) {
            yield return i++;
        } else {
            yield break;
        }
    }
    Console.WriteLine("这行代码将不会被执行");
}

```

[.NET Fiddle 在线演示](#)

请注意，与其他一些语言不同，C# 中没有办法为特定的 break 添加标签。这意味着在嵌套循环的情况下，只有最内层的循环会被终止：

```

foreach (var outerItem in outerList)
{
    foreach (var innerItem in innerList)
}

```

```

if (i == 5)
{
    break;
}
Console.WriteLine("This will appear only 5 times, as the break will stop the loop.");
}

```

[Live Demo on .NET Fiddle](#)

```

foreach (var stuff in stuffCollection)
{
    if (stuff.SomeStringProp == null)
        break;
    // If stuff.SomeStringProp for any "stuff" is null, the loop is aborted.
    Console.WriteLine(stuff.SomeStringProp);
}

```

The break-statement is also used in switch-case constructs to break out of a case or default segment.

```

switch(a)
{
    case 5:
        Console.WriteLine("a was 5!");
        break;

    default:
        Console.WriteLine("a was something else!");
        break;
}

```

In switch statements, the 'break' keyword is required at the end of each case statement. This is contrary to some languages that allow for 'falling through' to the next case statement in the series. Workarounds for this would include 'goto' statements or stacking the 'case' statements sequentially.

Following code will give numbers 0, 1, 2, ..., 9 and the last line will not be executed. `yield break` signifies the end of the function (not just a loop).

```

public static IEnumerable<int> GetNumbers()
{
    int i = 0;
    while (true) {
        if (i < 10) {
            yield return i++;
        } else {
            yield break;
        }
    }
    Console.WriteLine("This line will not be executed");
}

```

[Live Demo on .NET Fiddle](#)

Note that unlike some other languages, there is no way to label a particular break in C#. This means that in the case of nested loops, only the innermost loop will be stopped:

```

foreach (var outerItem in outerList)
{
    foreach (var innerItem in innerList)
}

```

```

{
    if (innerItem.ShouldBreakForWhateverReason)
        // 这只会跳出内层循环，外层循环将继续执行：
        break;
}

```

如果你想在这里跳出外层循环，可以使用几种不同的策略，例如：

- 使用`goto`语句跳出整个循环结构。
- 使用一个特定的标志变量（以下示例中的`shouldBreak`），可以在外层循环的每次迭代结束时检查。
- 重构代码，在最内层循环体中使用`return`语句，或者完全避免嵌套循环结构。

```

bool shouldBreak = false;
while(comeCondition)
{
    while(otherCondition)
    {
        if (conditionToBreak)
        {
            // 或者将控制流转移到下面的标签...
            goto endAllLooping;

            // 或者使用一个标志，可以在外层循环中检查：
        shouldBreak = true;
        }

        if(shouldBreakNow)
        {
            break; // 如果标志被设置为true，则跳出外层循环
        }
    }

    endAllLooping: // 控制流将从此标签继续
}

```

第52.8节：const

`const` 用于表示在程序生命周期内永远不会改变的值。它的值在编译时就是常量，这与`readonly`关键字不同，后者的值是在运行时才确定为常量的。

例如，由于光速永远不会改变，我们可以将其存储为常量。

```

const double c = 299792458; // 光速

double CalculateEnergy(double mass)
{
    return mass * c * c;
}

```

这本质上等同于`return mass * 299792458 * 299792458`，因为编译器会直接用常量值替换`c`。

因此，`c` 一旦声明后就不能更改。以下代码将产生编译时错误：

```
const double c = 299792458; // 光速
```

```

{
    if (innerItem.ShouldBreakForWhateverReason)
        // This will only break out of the inner loop, the outer will continue:
        break;
}

```

If you want to break out of the *outer* loop here, you can use one of several different strategies, such as:

- A `goto` statement to jump out of the whole looping structure.
- A specific flag variable (`shouldBreak` in the following example) that can be checked at the end of each iteration of the outer loop.
- Refactoring the code to use a `return` statement in the innermost loop body, or avoid the whole nested loop structure altogether.

```

bool shouldBreak = false;
while(comeCondition)
{
    while(otherCondition)
    {
        if (conditionToBreak)
        {
            // Either tranfer control flow to the label below...
            goto endAllLooping;

            // OR use a flag, which can be checked in the outer loop:
            shouldBreak = true;
        }

        if(shouldBreakNow)
        {
            break; // Break out of outer loop if flag was set to true
        }
    }

    endAllLooping: // label from where control flow will continue
}

```

Section 52.8: const

`const` is used to represent values that **will never change** throughout the lifetime of the program. Its value is constant from **compile-time**, as opposed to the `readonly` keyword, whose value is constant from run-time.

For example, since the speed of light will never change, we can store it in a constant.

```

const double c = 299792458; // Speed of light

double CalculateEnergy(double mass)
{
    return mass * c * c;
}

```

This is essentially the same as having `return mass * 299792458 * 299792458`, as the compiler will directly substitute `c` with its constant value.

As a result, `c` cannot be changed once declared. The following will produce a compile-time error:

```
const double c = 299792458; // Speed of light
```

```
c = 500; // 编译时错误
```

常量可以使用与方法相同的访问修饰符作为前缀：

```
private const double c = 299792458;
public const double c = 299792458;
internal const double c = 299792458;
```

const 成员本质上是 static 的，但不允许显式使用 static 关键字。

你也可以定义方法局部常量：

```
double CalculateEnergy(double mass)
{
    const c = 299792458;
    return mass * c * c;
}
```

这些常量不能加 private 或 public 关键字前缀，因为它们隐式地是定义所在方法的局部常量。

并非所有类型都可以用于const声明。允许的值类型包括预定义类型sbyte、byte、short、ushort、int、uint、long、ulong、char、float、double、decimal、bool，以及所有enum类型。尝试使用其他值类型（例如TimeSpan或Guid）声明const成员将导致编译时失败。

对于特殊的预定义引用类型string，可以声明任意值的常量。对于所有其他引用类型，可以声明常量，但其值必须始终为null。

由于const值在编译时已知，因此它们被允许作为switch语句中的case标签，作为可选参数的标准参数，作为属性规范的参数，等等。

如果const值在不同的程序集之间使用，则必须注意版本控制。例如，如果程序集A定义了public const int MaxRetries = 3；而程序集B使用该常量，那么如果程序集A中MaxRetries的值后来更改为5（并重新编译），该更改在程序集B中不会生效，除非程序集B也重新编译（并引用了程序集A的新版本）。

因此，如果某个值可能在程序的未来版本中更改，且该值需要公开可见，除非你确定所有依赖程序集在任何更改时都会重新编译，否则不要将该值声明为const。另一种选择是使用static readonly代替const，这将在运行时解析。

```
c = 500; //compile-time error
```

A constant can be prefixed with the same access modifiers as methods:

```
private const double c = 299792458;
public const double c = 299792458;
internal const double c = 299792458;
```

const members are static by nature. However using static explicitly is not permitted.

You can also define method-local constants:

```
double CalculateEnergy(double mass)
{
    const c = 299792458;
    return mass * c * c;
}
```

These can not be prefixed with a private or public keyword, since they are implicitly local to the method they are defined in.

Not all types can be used in a const declaration. The value types that are allowed, are the pre-defined types sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, and all enum types. Trying to declare const members with other value types (such as TimeSpan or Guid) will fail at compile-time.

For the special pre-defined reference type string, constants can be declared with any value. For all other reference types, constants can be declared but must always have the value null.

Because const values are known at compile-time, they are allowed as case labels in a switch statement, as standard arguments for optional parameters, as arguments to attribute specifications, and so on.

If const values are used across different assemblies, care must be taken with versioning. For example, if assembly A defines a public const int MaxRetries = 3; , and assembly B uses that constant, then if the value of MaxRetries is later changed to 5 in assembly A (which is then re-compiled), that change will not be effective in assembly B unless assembly B is also re-compiled (with a reference to the new version of A).

For that reason, if a value might change in future revisions of the program, and if the value needs to be publicly visible, do not declare that value const unless you know that all dependent assemblies will be re-compiled whenever something is changed. The alternative is using static readonly instead of const, which is resolved at runtime.

Section 52.9: async, await

await关键字作为C# 5.0版本的一部分引入，从Visual Studio 2012开始支持。它利用了任务并行库（Task Parallel Library, TPL），使多线程变得相对简单。下面示例中，async和await关键字成对使用。使用await关键字可以暂停当前异步方法的执行，直到被等待的异步任务完成并/或返回结果。为了使用await关键字，使用它的方法必须标记为async关键字。

强烈不建议将async与void一起使用。更多信息可以查看[这里](#)。

示例：

```
public async Task DoSomethingAsync()
{
```

The await keyword was added as part of C# 5.0 release which is supported from Visual Studio 2012 onwards. It leverages Task Parallel Library (TPL) which made the multi-threading relatively easier. The async and await keywords are used in pair in the same function as shown below. The await keyword is used to pause the current asynchronous method's execution until the awaited asynchronous task is completed and/or its results returned. In order to use the await keyword, the method that uses it must be marked with the async keyword.

Using async with void is strongly discouraged. For more info you can look [here](#).

Example:

```
public async Task DoSomethingAsync()
{
```

```

Console.WriteLine("开始一个无用的进程...");
Stopwatch stopwatch = Stopwatch.StartNew();
int delay = await UselessProcessAsync(1000);
stopwatch.Stop();
Console.WriteLine("一个无用的进程执行了 {0} 毫秒。",
stopwatch.ElapsedMilliseconds);
}

public async Task<int> 无用进程异步(int x)
{
    await Task.Delay(x);
    return x;
}

```

输出：

```

"开始一个无用的进程..."

**... 1 秒延迟... **

"一个无用的进程执行了 1000 毫秒。"

```

如果一个返回 Task 或 Task<T> 的方法仅返回单个异步操作，则可以省略关键字对 `async` 和 `await`。

而不是这样：

```

public async Task 打印并延迟异步(string message, int delay)
{
    Debug.WriteLine(message);
    await Task.Delay(x);
}

```

建议这样做：

```

public Task PrintAndDelayAsync(string message, int delay)
{
    Debug.WriteLine(message);
    return Task.Delay(x);
}

```

版本 = 5.0

在 C# 5.0 中，`await` 不能用于 `catch` 和 `finally`。

版本 ≥ 6.0

在 C# 6.0 中，`await` 可以用于 `catch` 和 `finally`。

第 52.10 节：for

语法：for (初始化器; 条件; 迭代器)

- 当已知迭代次数时，通常使用 `for` 循环。
- 初始化器 部分的语句只执行一次，在进入循环之前执行。
- 条件 部分包含一个布尔表达式，该表达式在每次循环迭代结束时进行评估，以

```

Console.WriteLine("Starting a useless process...");
Stopwatch stopwatch = Stopwatch.StartNew();
int delay = await UselessProcessAsync(1000);
stopwatch.Stop();
Console.WriteLine("A useless process took {0} milliseconds to execute.",
stopwatch.ElapsedMilliseconds);
}

public async Task<int> UselessProcessAsync(int x)
{
    await Task.Delay(x);
    return x;
}

```

Output:

```

"Starting a useless process..."

**... 1 second delay... **

"A useless process took 1000 milliseconds to execute."

```

The keyword pairs `async` and `await` can be omitted if a Task or Task<T> returning method only returns a single asynchronous operation.

Rather than this:

```

public async Task PrintAndDelayAsync(string message, int delay)
{
    Debug.WriteLine(message);
    await Task.Delay(x);
}

```

It is preferred to do this:

```

public Task PrintAndDelayAsync(string message, int delay)
{
    Debug.WriteLine(message);
    return Task.Delay(x);
}

```

Version = 5.0

In C# 5.0 `await` cannot be used in `catch` and `finally`.

Version ≥ 6.0

With C# 6.0 `await` can be used in `catch` and `finally`.

Section 52.10: for

Syntax: `for (initializer; condition; iterator)`

- The `for` loop is commonly used when the number of iterations is known.
- The statements in the `initializer` section run only once, before you enter the loop.
- The `condition` section contains a boolean expression that's evaluated at the end of every loop iteration to

确定循环是否应该退出或继续运行。

- 迭代器部分定义了循环体每次迭代后发生的操作。

此示例展示了如何使用for来遍历字符串的字符：

```
string str = "Hello";
for (int i = 0; i < str.Length; i++)
{
    Console.WriteLine(str[i]);
}
```

输出：

```
H  
e  
l  
l  
o
```

[.NET Fiddle 在线演示](#)

定义for语句的所有表达式都是可选的；例如，以下语句用于创建一个无限循环：

```
for( ; ; )
{
    // 你的代码写在这里
}
```

初始化器部分可以包含多个变量，只要它们是相同类型。条件部分可以由任何可以求值为bool的表达式组成。迭代器部分可以执行由逗号分隔的多个操作：

```
string hello = "hello";
for (int i = 0, j = 1, k = 9; i < 3 && k > 0; i++, hello += i) {
    Console.WriteLine(hello);
}
```

输出：

```
hello
hello1
hello12
```

[.NET Fiddle 在线演示](#)

第52.11节：摘要

用关键字abstract标记的类不能被实例化。

如果一个类包含抽象成员，或者继承了未实现的抽象成员，则该类必须被标记为抽象。即使没有抽象成员，类也可以被标记为抽象。

determine whether the loop should exit or should run again.

- The iterator section defines what happens after each iteration of the body of the loop.

This example shows how `for` can be used to iterate over the characters of a string:

```
string str = "Hello";
for (int i = 0; i < str.Length; i++)
{
    Console.WriteLine(str[i]);
}
```

Output:

```
H  
e  
l  
l  
o
```

[Live Demo on .NET Fiddle](#)

All of the expressions that define a `for` statement are optional; for example, the following statement is used to create an infinite loop:

```
for( ; ; )
{
    // Your code here
}
```

The initializer section can contain multiple variables, so long as they are of the same type. The condition section can consist of any expression which can be evaluated to a `bool`. And the iterator section can perform multiple actions separated by comma:

```
string hello = "hello";
for (int i = 0, j = 1, k = 9; i < 3 && k > 0; i++, hello += i) {
    Console.WriteLine(hello);
}
```

Output:

```
hello
hello1
hello12
```

[Live Demo on .NET Fiddle](#)

Section 52.11: abstract

A class marked with the keyword `abstract` cannot be instantiated.

A class *must* be marked as abstract if it contains abstract members or if it inherits abstract members that it doesn't implement. A class *may* be marked as abstract even if no abstract members are involved.

抽象类通常用作基类，当某部分实现需要由另一个组件指定时使用。

```
abstract class Animal
{
    string Name { get; set; }
    public abstract void MakeSound();
}

public class 猫 : 动物
{
    public override void 发出声音()
    {
        Console.WriteLine("Meow meow");
    }
}

public class 狗 : 动物
{
    public override void 发出声音()
    {
        Console.WriteLine("Bark bark");
    }
}

动物 猫 = new 猫(); // 由于猫继承自动物，因此允许
猫.发出声音(); // 将打印 "Meow meow"

动物 狗 = new 狗(); // 由于狗继承自动物，因此允许
狗.发出声音(); // 将打印 "Bark bark"

动物 动物 = new 动物(); // 由于是抽象类，不允许
```

使用关键字 `abstract` 标记的方法、属性或事件表示该成员的实现应由子类提供。如上所述，抽象成员只能出现在抽象类中。

```
abstract class Animal
{
    public abstract string 名称 { get; set; }
}

public class 猫 : 动物
{
    public override string Name { get; set; }
}

public class 狗 : Animal
{
    public override string Name { get; set; }
}
```

第52.12节：fixed

`fixed`语句将内存固定在一个位置。内存中的对象通常会移动，这使得垃圾回收成为可能。但当我们使用指向内存地址的不安全指针时，该内存不能被移动。

- 我们使用`fixed`语句来确保垃圾回收器不会重新定位字符串数据。

Abstract classes are usually used as base classes when some part of the implementation needs to be specified by another component.

```
abstract class Animal
{
    string Name { get; set; }
    public abstract void MakeSound();
}

public class Cat : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Meow meow");
    }
}

public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Bark bark");
    }
}

Animal cat = new Cat(); // Allowed due to Cat deriving from Animal
cat.MakeSound(); // will print out "Meow meow"

Animal dog = new Dog(); // Allowed due to Dog deriving from Animal
dog.MakeSound(); // will print out "Bark bark"

Animal animal = new Animal(); // Not allowed due to being an abstract class
```

A method, property, or event marked with the keyword `abstract` indicates that the implementation for that member is expected to be provided in a subclass. As mentioned above, abstract members can only appear in abstract classes.

```
abstract class Animal
{
    public abstract string Name { get; set; }
}

public class Cat : Animal
{
    public override string Name { get; set; }
}

public class Dog : Animal
{
    public override string Name { get; set; }
}
```

Section 52.12: fixed

The `fixed` statement fixes memory in one location. Objects in memory are usually moving around, this makes garbage collection possible. But when we use unsafe pointers to memory addresses, that memory must not be moved.

- We use the `fixed` statement to ensure that the garbage collector does not relocate the string data.

固定变量

```
var myStr = "Hello world!";

fixed (char* ptr = myStr)
{
    // myStr现在被固定（不会被垃圾回收器[重新]移动）。
    // 我们现在可以使用ptr做一些操作。
}
```

在不安全的上下文中使用。

固定数组大小

```
unsafe struct Example
{
    public fixed byte SomeField[8];
    public fixed char AnotherField[64];
}
```

fixed 只能用于 struct 中的字段（必须在不安全上下文中使用）。

第52.13节：默认值

对于类、接口、委托、数组、可空类型（如 int?）和指针类型，default(TheType) 返回 null：

```
class MyClass {}
Debug.Assert(default(MyClass) == null);
Debug.Assert(default(string) == null);
```

对于结构体和枚举，default(TheType) 返回与 new TheType() 相同的值：

```
struct Coordinates
{
    public int X { get; set; }
    public int Y { get; set; }
}

struct MyStruct
{
    public string Name { get; set; }
    public Coordinates Location { get; set; }
    public Coordinates? SecondLocation { get; set; }
    public TimeSpan Duration { get; set; }
}

var defaultStruct = default(MyStruct);
Debug.Assert(defaultStruct.Equals(new MyStruct()));
Debug.Assert(defaultStruct.Location.Equals(new Coordinates()));
Debug.Assert(defaultStruct.Location.X == 0);
Debug.Assert(defaultStruct.Location.Y == 0);
Debug.Assert(defaultStruct.SecondLocation == null);
Debug.Assert(defaultStruct.Name == null);
Debug.Assert(defaultStruct.Duration == TimeSpan.Zero);
```

default(T) 在 T 是一个泛型参数且没有约束来决定 T 是引用类型还是值类型时特别有用，例如：

Fixed Variables

```
var myStr = "Hello world!";

fixed (char* ptr = myStr)
{
    // myStr is now fixed (won't be [re]moved by the Garbage Collector).
    // We can now do something with ptr.
}
```

Used in an unsafe context.

Fixed Array Size

```
unsafe struct Example
{
    public fixed byte SomeField[8];
    public fixed char AnotherField[64];
}
```

fixed 只能用于 struct 中的字段（必须在不安全上下文中使用）。

Section 52.13: default

For classes, interfaces, delegate, array, nullable (such as int?) and pointer types, default(TheType) returns null:

```
class MyClass {}
Debug.Assert(default(MyClass) == null);
Debug.Assert(default(string) == null);
```

For structs and enums, default(TheType) returns the same as new TheType():

```
struct Coordinates
{
    public int X { get; set; }
    public int Y { get; set; }
}

struct MyStruct
{
    public string Name { get; set; }
    public Coordinates Location { get; set; }
    public Coordinates? SecondLocation { get; set; }
    public TimeSpan Duration { get; set; }
}
```

```
var defaultStruct = default(MyStruct);
Debug.Assert(defaultStruct.Equals(new MyStruct()));
Debug.Assert(defaultStruct.Location.Equals(new Coordinates()));
Debug.Assert(defaultStruct.Location.X == 0);
Debug.Assert(defaultStruct.Location.Y == 0);
Debug.Assert(defaultStruct.SecondLocation == null);
Debug.Assert(defaultStruct.Name == null);
Debug.Assert(defaultStruct.Duration == TimeSpan.Zero);
```

default(T) 可以特别有用，当 T 是一个泛型参数且没有约束来决定 T 是引用类型还是值类型时，例如：

```

public T GetResourceOrDefault<T>(string resourceName)
{
    if (ResourceExists(resourceName))
    {
        return (T)GetResource(resourceName);
    }
    else
    {
        return default(T);
    }
}

```

第52.14节 : sealed (密封)

当应用于类时, `sealed` 修饰符阻止其他类继承该类。

```

class A {}
sealed class B : A {}
class C : B {} //错误：无法从密封类继承

```

当应用于`virtual`方法 (或虚拟属性) 时, `sealed`修饰符阻止该方法 (属性) 在派生类中被重写。

```

public class A
{
    public sealed override string ToString() // 从类 Object 继承的虚拟方法
    {
        return "不要重写我!";
    }
}

public class B: A
{
    public override string ToString() // 编译时错误
    {
        return "尝试重写";
    }
}

```

第 52.15 节 : is

检查对象是否与给定类型兼容, 即对象是否是`BaseInterface`类型的实例, 或是从`BaseInterface`派生的类型:

```

interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
Console.WriteLine(d is DerivedClass); // True
Console.WriteLine(d is BaseClass); // True
Console.WriteLine(d is BaseInterface); // True
Console.WriteLine(d is object); // True
Console.WriteLine(d is string); // False

var b = new BaseClass();
Console.WriteLine(b is DerivedClass); // False
Console.WriteLine(b is BaseClass); // True

```

```

public T GetResourceOrDefault<T>(string resourceName)
{
    if (ResourceExists(resourceName))
    {
        return (T)GetResource(resourceName);
    }
    else
    {
        return default(T);
    }
}

```

Section 52.14: sealed

When applied to a class, the `sealed` modifier prevents other classes from inheriting from it.

```

class A {}
sealed class B : A {}
class C : B {} //error : Cannot derive from the sealed class

```

When applied to a `virtual` method (or virtual property), the `sealed` modifier prevents this method (property) from being *overridden* in derived classes.

```

public class A
{
    public sealed override string ToString() // Virtual method inherited from class Object
    {
        return "Do not override me!";
    }
}

public class B: A
{
    public override string ToString() // Compile time error
    {
        return "An attempt to override";
    }
}

```

Section 52.15: is

Checks if an object is compatible with a given type, i.e. if an object is an instance of the `BaseInterface` type, or a type that derives from `BaseInterface`:

```

interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
Console.WriteLine(d is DerivedClass); // True
Console.WriteLine(d is BaseClass); // True
Console.WriteLine(d is BaseInterface); // True
Console.WriteLine(d is object); // True
Console.WriteLine(d is string); // False

var b = new BaseClass();
Console.WriteLine(b is DerivedClass); // False
Console.WriteLine(b is BaseClass); // True

```

```
Console.WriteLine(b is BaseInterface); // True
Console.WriteLine(b is object); // True
Console.WriteLine(b is string); // False
```

如果转换的目的是使用该对象，最佳实践是使用as关键字

```
interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
Console.WriteLine(d is DerivedClass); // True - 合法使用'is'
Console.WriteLine(d is BaseClass); // True - 合法使用'is'

if(d is BaseClass){
    var castedD = (BaseClass)d;
    castedD.Method(); // 合法，但不是最佳实践
}

var asD = d as BaseClass;

if(asD!=null){
    asD.Method(); // 推荐的方法，因为只产生一次拆箱开销
}
```

但是，从 C# 7 开始，模式匹配功能扩展了 is 操作符，可以同时检查类型并声明新变量。C# 7 中的相同代码部分：

版本 ≥ 7.0

```
if(d is BaseClass asD ){
    asD.Method();
}
```

第52.16节：this

关键字this指当前类（对象）的实例。这样可以区分两个同名变量，一个是类级别的（字段），另一个是方法的参数（或局部变量）。

```
public MyClass {
    int a;

    void set_a(int a)
    {
        //this.a 指的是方法外定义的变量,
        //而 a 指传入的参数。
        this.a = a;
    }
}
```

关键字的其他用法包括链式调用非静态构造函数重载：

```
public MyClass(int arg) : this(arg, null)
{ }
```

以及编写索引器：

```
public string this[int idx1, string idx2]
```

```
Console.WriteLine(b is BaseInterface); // True
Console.WriteLine(b is object); // True
Console.WriteLine(b is string); // False
```

If the intent of the cast is to use the object, it is best practice to use the as keyword'

```
interface BaseInterface {}
class BaseClass : BaseInterface {}
class DerivedClass : BaseClass {}

var d = new DerivedClass();
Console.WriteLine(d is DerivedClass); // True - valid use of 'is'
Console.WriteLine(d is BaseClass); // True - valid use of 'is'

if(d is BaseClass){
    var castedD = (BaseClass)d;
    castedD.Method(); // valid, but not best practice
}

var asD = d as BaseClass;

if(asD!=null){
    asD.Method(); // preferred method since you incur only one unboxing penalty
}
```

But, from C# 7 pattern matching feature extends the is operator to check for a type and declare a new variable at the same time. Same code part with C# 7:

Version ≥ 7.0

```
if(d is BaseClass asD ){
    asD.Method();
}
```

Section 52.16: this

The **this** keyword refers to the current instance of class(object). That way two variables with the same name, one at the class-level (a field) and one being a parameter (or local variable) of a method, can be distinguished.

```
public MyClass {
    int a;

    void set_a(int a)
    {
        //this.a refers to the variable defined outside of the method,
        //while a refers to the passed parameter.
        this.a = a;
    }
}
```

Other usages of the keyword are chaining non-static constructor overloads:

```
public MyClass(int arg) : this(arg, null)
{ }
```

and writing indexers:

```
public string this[int idx1, string idx2]
```

```
{  
    get { /* ... */ }  
    set { /* ... */ }  
}
```

以及声明扩展方法：

```
public static int Count<TItem>(this IEnumerable<TItem> source)  
{  
    // ...  
}
```

如果与局部变量或参数没有冲突，是否使用`this`是风格问题，因此
`this.MemberOfType` 和 `MemberOfType` 在这种情况下是等价的。另请参见`base`关键字。

注意，如果要在当前实例上调用扩展方法，则必须使用`this`。例如，如果你在实现了`IEnumerable<>`的类的非静态方法中，并且想调用之前的扩展方法`Count`，则必须使用：

```
this.Count() // 作用类似于 StaticClassForExtensionMethod.Count(this)
```

此处不能省略`this`。

第52.17节：readonly

`readonly`关键字是字段修饰符。当字段声明包含`readonly`修饰符时，对该字段的赋值只能在声明时或同一类的构造函数中进行。

`readonly`关键字不同于`const`关键字。一个`const`字段只能在声明时初始化。一个`readonly`字段可以在声明时或构造函数中初始化。因此，`readonly`字段的值可以根据所使用的构造函数不同而不同。

`readonly`关键字通常在注入依赖时使用。

```
class Person  
{  
    readonly string _name;  
    readonly string _surname = "姓氏";  
  
    Person(string name)  
    {  
        _name = name;  
    }  
    void ChangeName()  
    {  
        _name = "另一个名字"; // 编译错误  
        _surname = "另一个姓氏"; // 编译错误  
    }  
}
```

注意：声明字段为`readonly`并不意味着不可变性。如果字段是引用类型，则对象的内容可以被更改。
通常`readonly`用于防止对象被覆盖，只能在对象实例化时赋值。

```
{  
    get { /* ... */ }  
    set { /* ... */ }  
}
```

and declaring extension methods:

```
public static int Count<TItem>(this IEnumerable<TItem> source)  
{  
    // ...  
}
```

If there is no conflict with a local variable or parameter, it is a matter of style whether to use `this` or not, so
`this.MemberOfType` and `MemberOfType` would be equivalent in that case. Also see `base` keyword.

Note that if an extension method is to be called on the current instance, `this` is required. For example if you are inside a non-static method of a class which implements `IEnumerable<>` and you want to call the extension `Count` from before, you must use:

```
this.Count() // works like StaticClassForExtensionMethod.Count(this)
```

and `this` cannot be omitted there.

Section 52.17: readonly

The `readonly` keyword is a field modifier. When a field declaration includes a `readonly` modifier, assignments to that field can only occur as part of the declaration or in a constructor in the same class.

The `readonly` keyword is different from the `const` keyword. A `const` field can only be initialized at the declaration of the field. A `readonly` field can be initialized either at the declaration or in a constructor. Therefore, `readonly` fields can have different values depending on the constructor used.

The `readonly` keyword is often used when injecting dependencies.

```
class Person  
{  
    readonly string _name;  
    readonly string _surname = "Surname";  
  
    Person(string name)  
    {  
        _name = name;  
    }  
    void ChangeName()  
    {  
        _name = "another name"; // Compile error  
        _surname = "another surname"; // Compile error  
    }  
}
```

Note: Declaring a field `readonly` does not imply *immutability*. If the field is a *reference type* then the **content** of the object can be changed. `readonly` is typically used to prevent having the object being **overwritten** and assigned only during **instantiation** of that object.

注意：在构造函数内部，readonly字段可以被重新赋值

```
public class Car
{
    public double Speed {get; set;}
}

//代码中

private readonly Car car = new Car();

private void SomeMethod()
{
    car.Speed = 100;
}
```

第52.18节：typeof

返回对象的Type类型，无需实例化对象。

```
Type type = typeof(string);
Console.WriteLine(type.FullName); //System.String
Console.WriteLine("Hello".GetType() == type); //True
Console.WriteLine("Hello".GetType() == typeof(string)); //True
```

第52.19节：foreach

foreach用于遍历数组的元素或实现了

IEnumerable的集合中的项。

```
var lines = new string[] {
    "你好，世界！",
    "你今天怎么样？",
    "再见"
};

foreach (string line in lines)
{
    Console.WriteLine(line);
}
```

这将输出

```
"你好，世界！"
"你今天怎么样？"
"再见"
```

[.NET Fiddle 在线演示](#)

你可以随时使用 break 关键字退出 foreach 循环，或者使用

continue 关键字跳过当前迭代继续下一次循环。

```
var numbers = new int[] {1, 2, 3, 4, 5, 6};

foreach (var number in numbers)
```

Note: Inside the constructor a readonly field can be reassigned

```
public class Car
{
    public double Speed {get; set;}
}

//In code

private readonly Car car = new Car();

private void SomeMethod()
{
    car.Speed = 100;
}
```

Section 52.18: typeof

Returns the Type of an object, without the need to instantiate it.

```
Type type = typeof(string);
Console.WriteLine(type.FullName); //System.String
Console.WriteLine("Hello".GetType() == type); //True
Console.WriteLine("Hello".GetType() == typeof(string)); //True
```

Section 52.19: foreach

foreach is used to iterate over the elements of an array or the items within a collection which implements [IEnumerable](#).

```
var lines = new string[] {
    "Hello world!",
    "How are you doing today?",
    "Goodbye"
};

foreach (string line in lines)
{
    Console.WriteLine(line);
}
```

This will output

```
"Hello world!"
"How are you doing today?"
"Goodbye"
```

[Live Demo on .NET Fiddle](#)

You can exit the foreach loop at any point by using the break keyword or move on to the next iteration using the continue keyword.

```
var numbers = new int[] {1, 2, 3, 4, 5, 6};

foreach (var number in numbers)
```

```

{
    // 如果是 2 则跳过
    if (number == 2)
        continue;

    // 如果等于5则停止迭代
    if (number == 5)
        break;

    Console.WriteLine(number + ", ");
}

// 输出: 1, 3, 4

```

[.NET Fiddle 在线演示](#)

请注意，只有某些集合如数组和List保证迭代顺序，而许多其他集合则不保证。

† 虽然 I Enumerable 通常用于表示可枚举集合，foreach 仅要求集合公开暴露 object GetEnumerator() 方法，该方法应返回一个对象，该对象暴露 bool MoveNext() 方法和 object Current { get; } 属性。

第52.20节：dynamic

dynamic 关键字用于动态类型对象。声明为 dynamic 的对象放弃编译时的静态检查，而是在运行时进行评估。

```

using System;
using System.Dynamic;

dynamic info = new ExpandoObject();
info.Id = 123;
info.Another = 456;

Console.WriteLine(info.Another);
// 456

Console.WriteLine(info.DoesNotExist);
// 抛出 RuntimeBinderException

```

下面的示例使用 dynamic 和 Newtonsoft 的 Json.NET 库，方便地读取反序列化的 JSON 文件中的数据。

```

try
{
    string json = @"{ x : 10, y : ""ho""}";
    dynamic deserializedJson = JsonConvert.DeserializeObject(json);
    int x = deserializedJson.x;
    string y = deserializedJson.y;
    // int z = deserializedJson.z; // 抛出 RuntimeBinderException
}
catch (RuntimeBinderException e)
{
    // 当使用未赋值给动态变量的属性时，会抛出此异常
}

```

```

{
    // Skip if 2
    if (number == 2)
        continue;

    // Stop iteration if 5
    if (number == 5)
        break;

    Console.WriteLine(number + ", ");
}

// Prints: 1, 3, 4,

```

[Live Demo on .NET Fiddle](#)

Notice that the order of iteration is guaranteed *only* for certain collections such as arrays and List, but **not** guaranteed for many other collections.

† While I Enumerable is typically used to indicate enumerable collections, foreach only requires that the collection expose publicly the object GetEnumerator() method, which should return an object that exposes the bool MoveNext() method and the object Current { get; } property.

Section 52.20: dynamic

The dynamic keyword is used with dynamically typed objects. Objects declared as dynamic forego compile-time static checks, and are instead evaluated at runtime.

```

using System;
using System.Dynamic;

dynamic info = new ExpandoObject();
info.Id = 123;
info.Another = 456;

Console.WriteLine(info.Another);
// 456

Console.WriteLine(info.DoesNotExist);
// Throws RuntimeBinderException

```

The following example uses dynamic with Newtonsoft's library Json.NET, in order to easily read data from a deserialized JSON file.

```

try
{
    string json = @"{ x : 10, y : ""ho""}";
    dynamic deserializedJson = JsonConvert.DeserializeObject(json);
    int x = deserializedJson.x;
    string y = deserializedJson.y;
    // int z = deserializedJson.z; // throws RuntimeBinderException
}
catch (RuntimeBinderException e)
{
    // This exception is thrown when a property
    // that wasn't assigned to a dynamic variable is used
}

```

dynamic 关键字存在一些限制。其中之一是扩展方法的使用。
下面的示例为字符串添加了一个扩展方法：SayHello。

```
static class StringExtensions
{
    public static string SayHello(this string s) => $"Hello {s}!";
}
```

第一种方法是像通常调用字符串方法那样调用它：

```
var person = "Person";
Console.WriteLine(person.SayHello());

dynamic manager = "Manager";
Console.WriteLine(manager.SayHello()); // RuntimeBinderException
```

没有编译错误，但在运行时会出现RuntimeBinderException。解决方法是通过静态类调用扩展方法：

```
var helloManager = StringExtensions.SayHello(manager);
Console.WriteLine(helloManager);
```

第52.21节：try、catch、finally、throw

try、catch、finally和throw允许你处理代码中的异常。

```
var processor = new InputProcessor();

// try块中的代码将被执行。如果在执行过程中发生异常，  
// 执行将转到与异常类型对应的catch块。
try
{
    processor.Process(input);
}
// 如果在try块中抛出FormatException，则会执行此catch块。

catch (FormatException ex)
{
    // throw是一个关键字，用于手动抛出异常，触发任何等待该异常类型的catch块。
    throw new InvalidOperationException("Invalid input", ex);
}
// catch 可以用来捕获所有或任何特定的异常。这个 catch 块，// 没有指定类型，捕获任何之前的 cat  
ch 块未捕获的异常。

catch
{
    LogUnexpectedException();
    throw; // 重新抛出原始异常。
}
// finally 块在所有 try-catch 块执行完后执行；无论是 try 块成功执行所有命令，还是所有异常都被捕获之后。

finally
{
    processor.Dispose();
}
```

注意：return 关键字可以在 try 块中使用，finally 块仍然会被执行（就在返回之前）。

There are some limitations associated with the dynamic keyword. One of them is the use of extension methods.
The following example adds an extension method for string: SayHello.

```
static class StringExtensions
{
    public static string SayHello(this string s) => $"Hello {s}!";
}
```

The first approach will be to call it as usual (as for a string):

```
var person = "Person";
Console.WriteLine(person.SayHello());

dynamic manager = "Manager";
Console.WriteLine(manager.SayHello()); // RuntimeBinderException
```

No compilation error, but at runtime you get a RuntimeBinderException. The workaround for this will be to call the extension method via the static class:

```
var helloManager = StringExtensions.SayHello(manager);
Console.WriteLine(helloManager);
```

Section 52.21: try, catch, finally, throw

try, catch, finally, and throw 允许你处理代码中的异常。

```
var processor = new InputProcessor();

// The code within the try block will be executed. If an exception occurs during execution of  
// this code, execution will pass to the catch block corresponding to the exception type.
try
{
    processor.Process(input);
}
// If a FormatException is thrown during the try block, then this catch block  
// will be executed.
catch (FormatException ex)
{
    // Throw is a keyword that will manually throw an exception, triggering any catch block that is  
// waiting for that exception type.
    throw new InvalidOperationException("Invalid input", ex);
}
// catch can be used to catch all or any specific exceptions. This catch block,  
// with no type specified, catches any exception that hasn't already been caught  
// in a prior catch block.

catch
{
    LogUnexpectedException();
    throw; // Re-throws the original exception.
}
// The finally block is executed after all try-catch blocks have been; either after the try has  
// succeeded in running all commands or after all exceptions have been caught.

finally
{
    processor.Dispose();
}
```

Note: The return keyword can be used in try block, and the finally block will still be executed (just before

例如：

```
try
{
    connection.Open();
    return connection.Get(query);
}
finally
{
    connection.Close();
}
```

语句connection.Close()将在connection.Get(query)的结果返回之前执行。

第52.22节：void

保留字"void"是System.Void类型的别名，有两种用法：

1. 声明一个没有返回值的方法：

```
public void DoSomething()
{
    // 执行一些操作，不向调用者返回任何值。
}
```

返回类型为void的方法体内仍然可以使用return关键字。当你想退出方法执行并将控制权返回给调用者时，这很有用：

```
public void DoSomething()
{
    // 执行一些操作...

    if (condition)
        return;

    // 如果条件为假，则执行更多操作。
}
```

2. 在不安全上下文中声明一个指向未知类型的指针。

在不安全的上下文中，类型可以是指针类型、值类型或引用类型。指针类型声明通常是类型*标识符，其中类型是已知类型——例如int*myInt，但也可以是void*标识符，其中类型未知。

请注意，微软不建议声明void指针类型。

第52.23节：命名空间

namespace关键字是一种组织结构，帮助我们理解代码库的排列方式。C#中的命名空间是虚拟空间，而不是物理文件夹。

```
namespace StackOverflow
{
    namespace Documentation
    {
        namespace CSharp.Keywords
        {
        }
    }
}
```

returning). For example:

```
try
{
    connection.Open();
    return connection.Get(query);
}
finally
{
    connection.Close();
}
```

The statement connection.Close() will execute before the result of connection.Get(query) is returned.

Section 52.22: void

The reserved word "void" is an alias of System.Void type, and has two uses:

1. Declare a method that doesn't have a return value:

```
public void DoSomething()
{
    // Do some work, don't return any value to the caller.
}
```

A method with a return type of void can still have the return keyword in its body. This is useful when you want to exit the method's execution and return the flow to the caller:

```
public void DoSomething()
{
    // Do some work...

    if (condition)
        return;

    // Do some more work if the condition evaluated to false.
}
```

2. Declare a pointer to an unknown type in an unsafe context.

In an unsafe context, a type may be a pointer type, a value type, or a reference type. A pointer type declaration is usually type* identifier, where the type is a known type - i.e int* myInt, but can also be void* identifier, where the type is unknown.

Note that declaring a void pointer type is [discouraged by Microsoft](#).

Section 52.23: namespace

The namespace keyword is an organization construct that helps us understand how a codebase is arranged. Namespaces in C# are virtual spaces rather than being in a physical folder.

```
namespace StackOverflow
{
    namespace Documentation
    {
        namespace CSharp.Keywords
        {
        }
    }
}
```

```

public class Program
{
    public static void Main()
    {
        Console.WriteLine(typeof(Program).Namespace);
        //StackOverflow.Documentation.CSharp.Keywords
    }
}
}

```

C#中的命名空间也可以使用链式语法编写。以下内容与上述等效：

```

namespace StackOverflow.Documentation.CSharp.Keywords
{
    public class Program
    {
        public static void Main()
        {
            Console.WriteLine(typeof(Program).Namespace);
            //StackOverflow.Documentation.CSharp.Keywords
        }
    }
}

```

第52.24节：ref, out

关键字 `ref` 和 `out` 使参数以引用方式传递，而非值传递。对于值类型，这意味着被调用者可以更改变量的值。

```

int x = 5;
ChangeX(ref x);
// 现在x的值可能不同了

```

对于引用类型，变量中的实例不仅可以被修改（如无 `ref` 时的情况），还可以被完全替换：

```

Address a = new Address();
ChangeFieldInAddress(a);
// a将是之前相同的实例，即使它被修改了
CreateANewInstance(ref a);
// a 现在可能是一个全新的实例

```

“out”和“ref”关键字的主要区别在于，“ref”要求变量由调用者初始化，而“out”则将此责任交给被调用者。

要使用“out”参数，方法定义和调用方法都必须显式使用“out”关键字。

```

int number = 1;
Console.WriteLine("AddByRef 之前: " + number); // number = 1
AddOneByRef(ref number);
Console.WriteLine("AddByRef 之后: " + number); // number = 2
SetByOut(out number);
Console.WriteLine("SetByOut 之后: " + number); // number = 34

void AddOneByRef(ref int value)
{

```

```

public class Program
{
    public static void Main()
    {
        Console.WriteLine(typeof(Program).Namespace);
        //StackOverflow.Documentation.CSharp.Keywords
    }
}
}

```

Namespaces in C# can also be written in chained syntax. The following is equivalent to above:

```

namespace StackOverflow.Documentation.CSharp.Keywords
{
    public class Program
    {
        public static void Main()
        {
            Console.WriteLine(typeof(Program).Namespace);
            //StackOverflow.Documentation.CSharp.Keywords
        }
    }
}

```

Section 52.24: ref, out

The `ref` and `out` keywords cause an argument to be passed by reference, not by value. For value types, this means that the value of the variable can be changed by the callee.

```

int x = 5;
ChangeX(ref x);
// The value of x could be different now

```

For reference types, the instance in the variable can not only be modified (as is the case without `ref`), but it can also be replaced altogether:

```

Address a = new Address();
ChangeFieldInAddress(a);
// a will be the same instance as before, even if it is modified
CreateANewInstance(ref a);
// a could be an entirely new instance now

```

The main difference between the `out` and `ref` keyword is that `ref` requires the variable to be initialized by the caller, while `out` passes that responsibility to the callee.

To use an `out` parameter, both the method definition and the calling method must explicitly use the `out` keyword.

```

int number = 1;
Console.WriteLine("Before AddByRef: " + number); // number = 1
AddOneByRef(ref number);
Console.WriteLine("After AddByRef: " + number); // number = 2
SetByOut(out number);
Console.WriteLine("After SetByOut: " + number); // number = 34

void AddOneByRef(ref int value)
{

```

```

        value++;
    }

    void SetByOut(out int value)
    {
        value = 34;
    }

```

[.NET Fiddle 在线演示](#)

以下代码无法编译，因为输出参数必须在方法返回之前被赋值（如果使用ref则可以编译）：

```

void PrintByOut(out int value)
{
    Console.WriteLine("Hello!");
}

```

使用 out 关键字作为泛型修饰符

out 关键字也可以用于定义泛型接口和委托时的泛型类型参数。在这种情况下，out 关键字指定该类型参数是协变的。

协变使您可以使用比泛型参数指定的类型更派生的类型。这允许实现变体接口的类的隐式转换以及委托类型的隐式转换。协变和逆变支持引用类型，但不支持值类型。 - MSDN

```
//如果我们有如下接口
interface ICovariant<out R> { }
```

```
//以及两个变量，如
ICovariant<Object> iobj = new Sample<Object>();
ICovariant<String> istr = new Sample<String>();

// 那么以下语句是有效的
// 如果没有 out 关键字，这将会报错
iobj = istr; // 这里发生了隐式转换
```

第52.25节：base

关键字base用于访问基类的成员。它通常用于调用虚方法的基类实现，或指定应调用哪个基类构造函数。

选择构造函数

```

public class Child : SomeBaseClass {
    public Child() : base("some string for the base class")
    {
    }
}

public class SomeBaseClass {
    public SomeBaseClass()
    {
        // new Child() 不会调用此构造函数，因为它没有参数
    }
}

```

```

        value++;
    }

    void SetByOut(out int value)
    {
        value = 34;
    }

```

[Live Demo on .NET Fiddle](#)

The following does *not* compile, because `out` parameters must have a value assigned before the method returns (it would compile using `ref` instead):

```

void PrintByOut(out int value)
{
    Console.WriteLine("Hello!");
}

```

using out keyword as Generic Modifier

out keyword can also be used in generic type parameters when defining generic interfaces and delegates. In this case, the out keyword specifies that the type parameter is covariant.

Covariance enables you to use a more derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement variant interfaces and implicit conversion of delegate types. Covariance and contravariance are supported for reference types, but they are not supported for value types. - MSDN

```
//if we have an interface like this
interface ICovariant<out R> { }

//and two variables like
ICovariant<Object> iobj = new Sample<Object>();
ICovariant<String> istr = new Sample<String>();

// then the following statement is valid
// without the out keyword this would have thrown error
iobj = istr; // implicit conversion occurs here
```

Section 52.25: base

The `base` keyword is used to access members from a base class. It is commonly used to call base implementations of virtual methods, or to specify which base constructor should be called.

Choosing a constructor

```

public class Child : SomeBaseClass {
    public Child() : base("some string for the base class")
    {
    }
}

public class SomeBaseClass {
    public SomeBaseClass()
    {
        // new Child() will not call this constructor, as it does not have a parameter
    }
}

```

```

    }
    public SomeBaseClass(string message)
    {
        // new Child() 会使用此基类构造函数，因为Child的构造函数中指定了参数
        Console.WriteLine(message);
    }
}

```

调用虚方法的基类实现

```

public override void SomeVirtualMethod()
{
    // 执行某些操作，然后调用基类实现
    base.SomeVirtualMethod();
}

```

可以使用 `base` 关键字从任何方法调用基类实现。这将方法调用直接绑定到基类实现，这意味着即使新的子类重写了虚方法，基类实现仍然会被调用，因此需要谨慎使用。

```

public class Parent
{
    public virtual int VirtualMethod()
    {
        return 1;
    }
}

public class Child : Parent
{
    public override int VirtualMethod() {
        return 11;
    }

    public int NormalMethod()
    {
        return base.VirtualMethod();
    }

    public void CallMethods()
    {
        Assert.AreEqual(11, VirtualMethod());
        Assert.AreEqual(1, NormalMethod());
        Assert.AreEqual(1, base.VirtualMethod());
    }
}

public class GrandChild : Child
{
    public override int VirtualMethod()
    {
        return 21;
    }

    public void CallAgain()
    {
        Assert.AreEqual(21, VirtualMethod());
        Assert.AreEqual(11, base.VirtualMethod());
    }
}

```

```

    }
    public SomeBaseClass(string message)
    {
        // new Child() will use this base constructor because of the specified parameter in Child's
        // constructor
        Console.WriteLine(message);
    }
}

```

Calling base implementation of virtual method

```

public override void SomeVirtualMethod()
{
    // Do something, then call base implementation
    base.SomeVirtualMethod();
}

```

It is possible to use the `base` keyword to call a base implementation from any method. This ties the method call directly to the base implementation, which means that even if new child classes override a virtual method, the base implementation will still be called so this needs to be used with caution.

```

public class Parent
{
    public virtual int VirtualMethod()
    {
        return 1;
    }
}

public class Child : Parent
{
    public override int VirtualMethod() {
        return 11;
    }

    public int NormalMethod()
    {
        return base.VirtualMethod();
    }

    public void CallMethods()
    {
        Assert.AreEqual(11, VirtualMethod());
        Assert.AreEqual(1, NormalMethod());
        Assert.AreEqual(1, base.VirtualMethod());
    }
}

public class GrandChild : Child
{
    public override int VirtualMethod()
    {
        return 21;
    }

    public void CallAgain()
    {
        Assert.AreEqual(21, VirtualMethod());
        Assert.AreEqual(11, base.VirtualMethod());
    }
}

```

```
// 注意下面对 NormalMethod 的调用仍然返回极基类中的值// 尽管该方法已在子类中被重写。
```

```
Assert.AreEqual(1, NormalMethod());  
}  
}
```

```
// Notice that the call to NormalMethod below still returns the value  
// from the extreme base class even though the method has been overridden  
// in the child class.  
Assert.AreEqual(1, NormalMethod());  
}  
}
```

第52.26节 : float, double, decimal

float

float 是 .NET 数据类型 System.Single 的别名。它允许存储 IEEE 754 单精度浮点数。该数据类型存在于 mscorelib.dll 中，每个 C# 项目创建时都会隐式引用该文件。

近似范围： -3.4×10^{38} 到 3.4×10^{38}

十进制精度：6-9 位有效数字

表示法：

```
float f = 0.1259;  
var f1 = 0.7895f; // f 是表示 float 值的字面量后缀
```

需要注意的是，float 类型常常会导致显著的舍入误差。在对精度要求较高的应用中，应考虑使用其他数据类型。

double

double 是 .NET 数据类型 System.Double 的别名。它表示双精度 64 位浮点数。该数据类型存在于 mscorelib.dll 中，任何 C# 项目都会隐式引用该文件。

范围： $\pm 5.0 \times 10^{-324}$ 到 $\pm 1.7 \times 10^{308}$

十进制精度：15-16 位有效数字

表示法：

```
double distance = 200.34; // 一个 double 类型的值  
double salary = 245; // 一个隐式类型转换为 double 的整数值  
var marks = 123.764D; // D 是表示 double 值的字面量后缀
```

decimal

decimal 是 .NET 数据类型 System.Decimal 的别名。它表示一个关键字，指示一个 128 位的数据类型。与浮点类型相比，decimal 类型具有更高的精度和更小的范围，这使得它适用于财务和货币计算。该数据类型存在于 mscorelib.dll 中，且在任何 C# 项目中都会被隐式引用。

范围： -7.9×10^{28} 到 7.9×10^{28}

Decimal 精度：28-29 位有效数字

Section 52.26: float, double, decimal

float

float 是 .NET 数据类型 System.Single 的别名。它允许 IEEE 754 单精度浮点数存储。该数据类型存在于 mscorelib.dll 中，所有 C# 项目都会隐式引用该文件。

近似范围： -3.4×10^{38} 到 3.4×10^{38}

十进制精度：6-9 位有效数字

Notation:

```
float f = 0.1259;  
var f1 = 0.7895f; // f is literal suffix to represent float values
```

需要注意的是，float 类型常常会导致显著的舍入误差。在对精度要求较高的应用中，应考虑使用其他数据类型。

double

double 是 .NET 数据类型 System.Double 的别名。它表示双精度 64 位浮点数。该数据类型存在于 mscorelib.dll 中，所有 C# 项目都会隐式引用该文件。

范围： $\pm 5.0 \times 10^{-324}$ 到 $\pm 1.7 \times 10^{308}$

十进制精度：15-16 位有效数字

Notation:

```
double distance = 200.34; // a double value  
double salary = 245; // an integer implicitly type-casted to double value  
var marks = 123.764D; // D is literal suffix to represent double values
```

decimal

decimal 是 .NET 数据类型 System.Decimal 的别名。它表示一个关键字，指示一个 128 位的数据类型。与浮点类型相比，decimal 类型具有更高的精度和更小的范围，这使得它适用于财务和货币计算。该数据类型存在于 mscorelib.dll 中，所有 C# 项目都会隐式引用该文件。

范围： -7.9×10^{28} 到 7.9×10^{28}

Decimal 精度：28-29 位有效数字

表示法：

```
decimal payable = 152.25m; // 一个 decimal 类型的值
var marks = 754.24m; // m 是表示 decimal 值的字面量后缀
```

第 52.27 节：运算符

大多数 [内置运算符](#)（包括转换运算符）可以通过使用 `operator` 关键字以及 `public` 和 `static` 修饰符来重载。

运算符有三种形式：一元运算符、二元运算符和转换运算符。

一元和二元运算符至少需要一个与包含类型相同的参数，有些还需要一个互补匹配的运算符。

转换运算符必须转换为或从封闭类型。

```
public struct Vector32
{
    public Vector32(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; }
    public int Y { get; }

    public static bool operator ==(Vector32 left, Vector32 right)
        => left.X == right.X && left.Y == right.Y;

    public static bool operator !=(Vector32 left, Vector32 right)
        => !(left == right);

    public static Vector32 operator +(Vector32 left, Vector32 right)
        => new Vector32(left.X + right.X, left.Y + right.Y);

    public static Vector32 operator +(Vector32 left, int right)
        => new Vector32(left.X + right, left.Y + right);

    public static Vector32 operator +(int left, Vector32 right)
        => right + left;

    public static Vector32 operator -(Vector32 left, Vector32 right)
        => new Vector32(left.X - right.X, left.Y - right.Y);

    public static Vector32 operator -(Vector32 left, int right)
        => new Vector32(left.X - right, left.Y - right);

    public static Vector32 operator -(int left, Vector32 right)
        => right - left;

    public static implicit operator Vector64(Vector32 vector)
        => new Vector64(vector.X, vector.Y);

    public override string ToString() => $"{{{{X}, {Y}}}}";
}
```

Notation:

```
decimal payable = 152.25m; // a decimal value
var marks = 754.24m; // m is literal suffix to represent decimal values
```

Section 52.27: operator

Most of the [built-in operators](#) (including conversion operators) can be overloaded by using the `operator` keyword along with the `public` and `static` modifiers.

The operators comes in three forms: unary operators, binary operators and conversion operators.

Unary and binary operators requires at least one parameter of same type as the containing type, and some requires a complementary matching operator.

Conversion operators must convert to or from the enclosing type.

```
public struct Vector32
{
    public Vector32(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; }
    public int Y { get; }

    public static bool operator ==(Vector32 left, Vector32 right)
        => left.X == right.X && left.Y == right.Y;

    public static bool operator !=(Vector32 left, Vector32 right)
        => !(left == right);

    public static Vector32 operator +(Vector32 left, Vector32 right)
        => new Vector32(left.X + right.X, left.Y + right.Y);

    public static Vector32 operator +(Vector32 left, int right)
        => new Vector32(left.X + right, left.Y + right);

    public static Vector32 operator +(int left, Vector32 right)
        => right + left;

    public static Vector32 operator -(Vector32 left, Vector32 right)
        => new Vector32(left.X - right.X, left.Y - right.Y);

    public static Vector32 operator -(Vector32 left, int right)
        => new Vector32(left.X - right, left.Y - right);

    public static Vector32 operator -(int left, Vector32 right)
        => right - left;

    public static implicit operator Vector64(Vector32 vector)
        => new Vector64(vector.X, vector.Y);

    public override string ToString() => $"{{{{X}, {Y}}}}";
}
```

```

public struct Vector64
{
    public Vector64(long x, long y)
    {
        X = x;
        Y = y;
    }

    public long X { get; }
    public long Y { get; }

    public override string ToString() => $"{{X}, {Y}}";
}

```

示例

```

var vector1 = new Vector32(15, 39);
var vector2 = new Vector32(87, 64);

Console.WriteLine(vector1 == vector2); // false
Console.WriteLine(vector1 != vector2); // true
Console.WriteLine(vector1 + vector2); // {102, 103}
Console.WriteLine(vector1 - vector2); // {-72, -25}

```

第52.28节：字符

字符 (char) 是存储在变量中的单个字母。它是一种内置值类型，占用两个字节的内存空间。它表示System.Char数据类型，该类型位于mscorlib.dll中，每个C#项目在创建时都会隐式引用该dll。

有多种方法可以做到这一点。

1. `char c = 'c';`
2. `char c = '\u0063'; //Unicode`
3. `char c = '63'; //十六进制`
4. `char c = (char)99; //整数`

字符 (char) 可以隐式转换为ushort、int、uint、long、ulong、float、double或decimal类型，并返回该字符的整数值。

```
ushort u = c;
```

返回99等。

但是，没有其他类型到char的隐式转换。相反，必须进行强制类型转换。

```
ushort u = 99;
char c = (char)u;
```

第52.29节：params

params 允许方法参数接收可变数量的参数，即该参数可以接受零个、一个或多个参数。

```

public struct Vector64
{
    public Vector64(long x, long y)
    {
        X = x;
        Y = y;
    }

    public long X { get; }
    public long Y { get; }

    public override string ToString() => $"{{X}, {Y}}";
}

```

Example

```

var vector1 = new Vector32(15, 39);
var vector2 = new Vector32(87, 64);

Console.WriteLine(vector1 == vector2); // false
Console.WriteLine(vector1 != vector2); // true
Console.WriteLine(vector1 + vector2); // {102, 103}
Console.WriteLine(vector1 - vector2); // {-72, -25}

```

Section 52.28: char

A char is single letter stored inside a variable. It is built-in value type which takes two bytes of memory space. It represents System.Char data type found in mscorlib.dll which is implicitly referenced by every C# project when you create them.

There are multiple ways to do this.

1. `char c = 'c';`
2. `char c = '\u0063'; //Unicode`
3. `char c = 'x0063'; //Hex`
4. `char c = (char)99; //Integral`

A char can be implicitly converted to ushort, int, uint, long, ulong, float, double, or decimal and it will return the integer value of that char.

```
ushort u = c;
```

returns 99 etc.

However, there are no implicit conversions from other types to char. Instead you must cast them.

```
ushort u = 99;
char c = (char)u;
```

Section 52.29: params

params allows a method parameter to receive a variable number of arguments, i.e. zero, one or multiple arguments are allowed for that parameter.

```

static int AddAll(params int[] numbers)
{
    int total = 0;
    foreach (int number in numbers)
    {
        total += number;
    }

    return total;
}

```

该方法现在可以使用典型的int参数列表或一个int数组来调用。

```

AddAll(5, 10, 15, 20);           // 50
AddAll(new int[] { 5, 10, 15, 20 }); // 50

```

params 必须最多出现一次，且如果使用，必须是参数列表中的最后一个，即使后续类型与数组类型不同。

使用params关键字重载函数时要小心。C#更倾向于先匹配更具体的重载，然后才尝试使用带有params的重载。
例如，如果你有两个方法：

```

static double Add(params double[] numbers)
{
    Console.WriteLine("使用双精度数组的Add");
    double total = 0.0;
    foreach (double number in numbers)
    {
        total += number;
    }

    return total;
}

static int Add(int a, int b)
{
    Console.WriteLine("使用两个整数的Add");
    return a + b;
}

```

那么，具体的两个参数重载会优先于尝试params重载。

```

Add(2, 3);      //输出 "使用两个整数的Add"
Add(2, 3.0);    //输出 "使用双精度数组的Add" (double不是int)
Add(2, 3, 4);   //输出 "使用双精度数组的Add" (没有三个参数的重载)

```

第52.30节：while

while操作符会迭代执行一段代码块，直到条件查询为假或代码被goto、return、break或throw语句中断。

while关键字的语法：

```
while( condition ) { 代码块; }
```

示例：

```

static int AddAll(params int[] numbers)
{
    int total = 0;
    foreach (int number in numbers)
    {
        total += number;
    }

    return total;
}

```

This method can now be called with a typical list of int arguments, or an array of ints.

```

AddAll(5, 10, 15, 20);           // 50
AddAll(new int[] { 5, 10, 15, 20 }); // 50

```

params 必须出现最多一次且如果使用，它必须是参数列表中的最后一个，即使后续类型与数组类型不同。

Be careful when overloading functions when using the params keyword. C# prefers matching more specific overloads before resorting to trying to use overloads with params. For example if you have two methods:

```

static double Add(params double[] numbers)
{
    Console.WriteLine("Add with array of doubles");
    double total = 0.0;
    foreach (double number in numbers)
    {
        total += number;
    }

    return total;
}

static int Add(int a, int b)
{
    Console.WriteLine("Add with 2 ints");
    return a + b;
}

```

那么，具体的两个参数重载会优先于尝试params重载。

```

Add(2, 3);      //prints "Add with 2 ints"
Add(2, 3.0);    //prints "Add with array of doubles" (doubles are not ints)
Add(2, 3, 4);   //prints "Add with array of doubles" (no 3 argument overload)

```

Section 52.30: while

The while operator iterates over a block of code until the conditional query equals false or the code is interrupted with a goto, return, break or throw statement.

Syntax for while keyword:

```
while( condition ) { code block; }
```

Example:

```
int i = 0;
while (i++ < 5)
{
    Console.WriteLine("While 是第 {0} 次循环。", i);
}
```

输出：

```
"While 是第 1 次循环。"
"While 是第 2 次循环。"
"While 是第 3 次循环。"
"While 是第 4 次循环。"
"While 是第 5 次循环。"
```

[.NET Fiddle 在线演示](#)

while 循环是入口控制的，因为条件在执行包含的代码块之前被检查。这意味着如果条件为假，while 循环不会执行其语句。

```
bool a = false;

while (a == true)
{
    Console.WriteLine("这条信息永远不会被打印。");
}
```

给出一个while条件而不预设其在某个时刻变为假，将导致无限或无尽的循环。尽可能应避免这种情况，然而，有些特殊情况下你可能需要这样做。

你可以按如下方式创建这样的循环：

```
while (true)
{
//...
}
```

请注意，C# 编译器会将如下循环转换为

```
while (true)
{
// ...
}
```

或

```
for(;;)
{
// ...
}
```

转换为

```
{
label
```

```
int i = 0;
while (i++ < 5)
{
    Console.WriteLine("While is on loop number {0}.", i);
}
```

Output:

```
"While is on loop number 1."
"While is on loop number 2."
"While is on loop number 3."
"While is on loop number 4."
"While is on loop number 5."
```

[Live Demo on .NET Fiddle](#)

A while loop is **Entry Controlled**, as the condition is checked **before** the execution of the enclosed code block. This means that the while loop wouldn't execute its statements if the condition is false.

```
bool a = false;

while (a == true)
{
    Console.WriteLine("This will never be printed.");
}
```

Giving a `while` condition without provisioning it to become false at some point will result in an infinite or endless loop. As far as possible, this should be avoided, however, there may be some exceptional circumstances when you need this.

You can create such a loop as follows:

```
while (true)
{
//...
}
```

Note that the C# compiler will transform loops such as

```
while (true)
{
// ...
}
```

or

```
for(;;)
{
// ...
}
```

into

```
{
:label
```

```
// ...
goto label;
}
```

请注意，while 循环的条件可以是任意复杂的表达式，只要它计算（或返回）一个布尔值（bool）。它也可以包含返回布尔值的函数（因为这样的函数计算结果与表达式如 `a==x` 的类型相同）。例如，

```
while (AgriculturalService.MoreCornToPick(myFarm.GetAddress()))
{
    myFarm.PickCorn();
}
```

第52.31节：null

引用类型的变量可以保存对实例的有效引用或空引用。空引用是引用类型变量以及可空值类型的默认值。

null 是表示空引用的关键字。

作为表达式，它可以用于将空引用赋值给上述类型的变量：

```
object a = null;
string b = null;
int? c = null;
List<int> d = null;
```

不可空值类型不能被赋值为空引用。以下所有赋值都是无效的：

```
int a = null;
float b = null;
decimal c = null;
```

空引用不应与各种类型的有效实例混淆，例如：

- 一个空列表 (`new List<int>()`)
- 一个空字符串 (`" "`)
- 数字零 (`0, 0f, 0m`)
- 空字符 (`'\0'`)

有时，检查某个值是否为null或空/默认对象是有意义的。可以使用`System.String.IsNullOrEmpty(string)`方法来进行检查，或者你也可以实现自己的等效方法。

```
private void GreetUser(string userName)
{
    if (String.IsNullOrEmpty(userName))
    {
        //调用我们的方法要么传入了空字符串，要么传入了null引用。
        //无论哪种情况，都需要报告这个问题。
        throw new InvalidOperationException("userName may not be null or empty.");
    }
    else
    {
        //userName 是可接受的。
        Console.WriteLine("Hello, " + userName + "!");
    }
}
```

```
// ...
goto label;
}
```

Note that a while loop may have any condition, no matter how complex, as long as it evaluates to (or returns) a boolean value (bool). It may also contain a function that returns a boolean value (as such a function evaluates to the same type as an expression such as `a==x'). For example,

```
while (AgriculturalService.MoreCornToPick(myFarm.GetAddress()))
{
    myFarm.PickCorn();
}
```

Section 52.31: null

A variable of a reference type can hold either a valid reference to an instance or a null reference. The null reference is the default value of reference type variables, as well as nullable value types.

`null` is the keyword that represents a null reference.

As an expression, it can be used to assign the null reference to variables of the aforementioned types:

```
object a = null;
string b = null;
int? c = null;
List<int> d = null;
```

Non-nullable value types cannot be assigned a null reference. All the following assignments are invalid:

```
int a = null;
float b = null;
decimal c = null;
```

The null reference should *not* be confused with valid instances of various types such as:

- an empty list (`new List<int>()`)
- an empty string (`" "`)
- the number zero (`0, 0f, 0m`)
- the null character (`'\0'`)

Sometimes, it is meaningful to check if something is either null or an empty/default object. The `System.String.IsNullOrEmpty(string)` method may be used to check this, or you may implement your own equivalent method.

```
private void GreetUser(string userName)
{
    if (String.IsNullOrEmpty(userName))
    {
        //The method that called us either sent in an empty string, or they sent us a null reference.
        //Either way, we need to report the problem.
        throw new InvalidOperationException("userName may not be null or empty.");
    }
    else
    {
        //userName is acceptable.
        Console.WriteLine("Hello, " + userName + "!");
    }
}
```

```
}
```

第52.32节：继续

立即将控制权传递给包含循环结构（for、foreach、do、while）的下一次迭代：

```
for (var i = 0; i < 10; i++)
{
    if (i < 5)
    {
        continue;
    }
    Console.WriteLine(i);
}
```

输出：

```
5
```

[.NET Fiddle 在线演示](#)

```
var stuff = new [] {"a", "b", null, "c", "d"};

foreach (var s in stuff)
{
    if (s == null)
    {
        continue;
    }
    Console.WriteLine(s);
}
```

输出：

String，允许存储文本（字符序列）。

符号说明：

```
string a = "Hello";
var b = "world";
```

```
}
```

Section 52.32: continue

Immediately pass control to the next iteration of the enclosing loop construct (for, foreach, do, while):

```
for (var i = 0; i < 10; i++)
{
    if (i < 5)
    {
        continue;
    }
    Console.WriteLine(i);
}
```

Output:

```
5
6
7
8
9
```

[Live Demo on .NET Fiddle](#)

```
var stuff = new [] {"a", "b", null, "c", "d"};

foreach (var s in stuff)
{
    if (s == null)
    {
        continue;
    }
    Console.WriteLine(s);
}
```

Output:

```
a
b
c
d
```

[Live Demo on .NET Fiddle](#)

Section 52.33: string

`string` is an alias to the .NET datatype `System.String`, which allows text (sequences of characters) to be stored.

Notation:

```
string a = "Hello";
var b = "world";
```

```
var f = new string(new []{ 'h', 'i', '!' }); // hi!
```

字符串中的每个字符都采用 UTF-16 编码，这意味着每个字符至少需要 2 字节的存储空间。

第 52.34 节 : return

MSDN : return 语句终止其所在方法的执行，并将控制权返回给调用该方法的代码。它还可以返回一个可选值。如果方法是 void 类型，则可以省略 return 语句。

```
public int Sum(int valueA, int valueB)
{
    return valueA + valueB;
}
```

```
public void Terminate(bool terminateEarly)
{
    if (terminateEarly) return; // 如果传入 true, 方法返回调用者
    else Console.WriteLine("Not early"); // 仅当 terminateEarly 为 false 时打印
}
```

```
var f = new string(new []{ 'h', 'i', '!' }); // hi!
```

Each character in the string is encoded in UTF-16, which means that each character will require a minimum 2 bytes of storage space.

Section 52.34: return

MSDN: The return statement terminates execution of the method in which it appears and returns control to the calling method. It can also return an optional value. If the method is a void type, the return statement can be omitted.

```
public int Sum(int valueA, int valueB)
{
    return valueA + valueB;
}
```

```
public void Terminate(bool terminateEarly)
{
    if (terminateEarly) return; // method returns to caller if true was passed in
    else Console.WriteLine("Not early"); // prints only if terminateEarly was false
}
```

第 52.35 节 : unsafe

关键字unsafe可以用于类型或方法声明，或声明内联代码块。

该关键字的目的是启用该代码块中C#的unsafe子集的使用。unsafe 子集包括指针、栈分配、类似C的数组等特性。

不安全代码不可验证，因此不建议使用。编译不安全代码需要向C#编译器传递一个开关。此外，CLR要求运行的程序集具有完全信任权限。

尽管有这些限制，不安全代码在某些操作中仍有合理用途，使其更高效（例如数组索引）或更简便（例如与某些非托管库的互操作）。

作为一个非常简单的示例

```
// 使用 /unsafe 编译
class UnsafeTest
{
    unsafe static void SquarePtrParam(int* p)
    {
        *p *= *p; // '*' 用于解引用指针。
        // 由于我们传入了“i 的地址”，这就变成了“i *= i”
    }

    unsafe static void Main()
    {
        int i = 5;
        // 不安全的方法：使用取地址操作符 (&)：
        SquarePtrParam(&i); // "&i" 表示 "i 的地址"。行为类似于 "ref i"
        Console.WriteLine(i); // 输出: 25
    }
}
```

Section 52.35: unsafe

The unsafe keyword can be used in type or method declarations or to declare an inline block.

The purpose of this keyword is to enable the use of the unsafe subset of C# for the block in question. The unsafe subset includes features like pointers, stack allocation, C-like arrays, and so on.

Unsafe code is not verifiable and that's why its usage is discouraged. Compilation of unsafe code requires passing a switch to the C# compiler. Additionally, the CLR requires that the running assembly has full trust.

Despite these limitations, unsafe code has valid usages in making some operations more performant (e.g. array indexing) or easier (e.g. interop with some unmanaged libraries).

As a very simple example

```
// compile with /unsafe
class UnsafeTest
{
    unsafe static void SquarePtrParam(int* p)
    {
        *p *= *p; // the '*' dereferences the pointer.
        // Since we passed in "the address of i", this becomes "i *= i"
    }

    unsafe static void Main()
    {
        int i = 5;
        // Unsafe method: uses address-of operator (&):
        SquarePtrParam(&i); // "&i" means "the address of i". The behavior is similar to "ref i"
        Console.WriteLine(i); // Output: 25
    }
}
```

在使用指针时，我们可以直接更改内存位置的值，而不必通过名称来访问它们。请注意，这通常需要使用 `fixed` 关键字以防止垃圾回收器移动对象时可能导致的内存损坏（否则，可能会出现 `error CS0212`）。由于被“固定”的变量不能被写入，我们通常还需要第二个指针，起初指向与第一个指针相同的位置。

```
void Main()
{
    int[] intArray = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    UnsafeSquareArray(intArray);
    foreach(int i in intArray)
        Console.WriteLine(i);
}

unsafe static void UnsafeSquareArray(int[] pArr)
{
    int len = pArr.Length;

    // 在 C 或 C++ 中，我们可以写成
    // int* a = &(pArr[0])
    // 但是，C# 要求你先“固定”变量
    fixed(int* fixedPointer = &(pArr[0]))
    {
        // 声明一个新的 int 指针，因为“fixedPointer”不能被写入。
        // “p”指向相同的地址空间，但我们可以修改它
        int* p = fixedPointer;

        for (int i = 0; i < len; i++)
        {
            *p *= *p; // 将值平方，就像上面 SquarePtrParam 中做的那样
            p++; // 将指针移动到下一个内存空间。
            // 注意指针将移动 4 个字节，因为“p”是一个
            // int 指针，而 int 占用 4 个字节

            // 上面两行代码也可以写成一行，像这样：
            // "*p *= *p++;""
        }
    }
}
```

输出：

```
1  
4  
9  
1  
6  
2  
5  
3  
6  
4  
9  
6  
4  
8  
1  
1  
0  
0  
u  
n  
s  
a  
f  
e
```

还允许使用 `stackalloc`，它将在栈上分配内存，类似于 C 运行时库中的 `_alloca`。

我们可以将上述示例修改为使用 `stackalloc`，如下所示：

```
unsafe void Main()
{
```

While working with pointers, we can change the values of memory locations directly, rather than having to address them by name. Note that this often requires the use of the `fixed` keyword to prevent possible memory corruption as the garbage collector moves things around (otherwise, you may get [error CS0212](#)). Since a variable that has been “fixed” cannot be written to, we also often have to have a second pointer that starts out pointing to the same location as the first.

```
void Main()
{
    int[] intArray = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    UnsafeSquareArray(intArray);
    foreach(int i in intArray)
        Console.WriteLine(i);
}

unsafe static void UnsafeSquareArray(int[] pArr)
{
    int len = pArr.Length;

    //in C or C++, we could say
    // int* a = &(pArr[0])
    // however, C# requires you to "fix" the variable first
    fixed(int* fixedPointer = &(pArr[0]))
    {
        //Declare a new int pointer because "fixedPointer" cannot be written to.
        // "p" points to the same address space, but we can modify it
        int* p = fixedPointer;

        for (int i = 0; i < len; i++)
        {
            *p *= *p; //square the value, just like we did in SquarePtrParam, above
            p++; //move the pointer to the next memory space.
            // NOTE that the pointer will move 4 bytes since "p" is an
            // int pointer and an int takes 4 bytes

            //the above 2 lines could be written as one, like this:
            // "*p *= *p++;""
        }
    }
}
```

Output:

```
1  
4  
9  
16  
25  
36  
49  
64  
81  
100
```

`unsafe` also allows the use of `stackalloc` which will allocate memory on the stack like `_alloca` in the C run-time library. We can modify the above example to use `stackalloc` as follows:

```
unsafe void Main()
{
```

```

const int len=10;
int* seedArray = stackalloc int[len];

//我们不能再像以前那样使用初始化器 "{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}"。
// 我们至少有两种选项来填充数组。无论选择哪种
// 选项，最终结果都是相同的（这里两者都做也一样）。

//第一种选项：
int* p = seedArray; // 我们不想丢失数组的起始位置，所以
                     // 创建了指针的影子副本
for(int i=1; i<=len; i++)
    *p++ = i;
//第一种选项结束

//第二种选项：
for(int i=0; i<len; i++)
    seedArray[i] = i+1;
//第二个选项结束

UnsafeSquareArray(seedArray, len);
for(int i=0; i< len; i++)
Console.WriteLine(seedArray[i]);
}

//既然我们直接使用指针，就不需要使用
// "fixed"，这大大简化了代码
unsafe static void UnsafeSquareArray(int* p, int len)
{
    for (int i = 0; i < len; i++)
        *p *= *p++;
}

```

(输出与上述相同)

第52.36节：switch语句

switch语句是一种控制语句，用于从候选列表中选择一个switch部分执行。一个switch语句包含一个或多个switch部分。每个switch部分包含一个或多个case标签，后跟一个或多个语句。如果没有case标签匹配值，则控制权转移到default部分（如果存在）。严格来说，C#不支持case穿透。然而，如果一个或多个case标签为空，执行将继续到下一个包含代码的case块。这允许将多个case标签分组使用相同的实现。在下面的示例中，如果month等于12，将执行case 2中的代码，因为case标签12 1和2被分组。如果case块不为空，必须在下一个case标签之前有一个break，否则编译器会报错。

```

int month = DateTime.Now.Month; // 这里预期是1-12，代表1月至12月

switch (month)
{
    case 12:
    case 1:
    case 2:
        Console.WriteLine("冬季");
        break;
    case 3:
    case 4:
    case 5:
        Console.WriteLine("春季");
        break;
}

```

```

const int len=10;
int* seedArray = stackalloc int[len];

//We can no longer use the initializer "{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}" as before.
// We have at least 2 options to populate the array. The end result of either
// option will be the same (doing both will also be the same here).

//FIRST OPTION:
int* p = seedArray; // we don't want to lose where the array starts, so we
                     // create a shadow copy of the pointer
for(int i=1; i<=len; i++)
    *p++ = i;
//end of first option

//SECOND OPTION:
for(int i=0; i<len; i++)
    seedArray[i] = i+1;
//end of second option

UnsafeSquareArray(seedArray, len);
for(int i=0; i< len; i++)
Console.WriteLine(seedArray[i]);
}

//Now that we are dealing directly in pointers, we don't need to mess around with
// "fixed", which dramatically simplifies the code
unsafe static void UnsafeSquareArray(int* p, int len)
{
    for (int i = 0; i < len; i++)
        *p *= *p++;
}

```

(Output is the same as above)

Section 52.36: switch

The `switch` statement is a control statement that selects a switch section to execute from a list of candidates. A switch statement includes one or more switch sections. Each switch section contains one or more `case` labels followed by one or more statements. If no case label contains a matching value, control is transferred to the `default` section, if there is one. Case fall-through is not supported in C#, strictly speaking. However, if 1 or more `case` labels are empty, execution will follow the code of the next `case` block which contains code. This allows grouping of multiple `case` labels with the same implementation. In the following example, if `month` equals 12, the code in `case 2` will be executed since the `case` labels 12 1 and 2 are grouped. If a `case` block is not empty, a `break` must be present before the next `case` label, otherwise the compiler will flag an error.

```

int month = DateTime.Now.Month; // this is expected to be 1-12 for Jan-Dec

switch (month)
{
    case 12:
    case 1:
    case 2:
        Console.WriteLine("Winter");
        break;
    case 3:
    case 4:
    case 5:
        Console.WriteLine("Spring");
        break;
}

```

```

case 6:
case 7:
case 8:
Console.WriteLine("夏季");
break;
case 9:
case 10:
case 11:
Console.WriteLine("秋天");
break;
default:
Console.WriteLine("月份索引不正确");
break;
}

```

一个case只能由编译时已知的值标记（例如1、"str"、Enum.A），因此变量不是有效的case标签，但const或Enum值是有效的（以及任何字面值）。

第52.37节：var

一种隐式类型的局部变量，其类型与用户显式声明的类型完全相同。与其他变量声明不同，编译器根据赋给它的值确定该变量的类型。

```

var i = 10; // 隐式类型，编译器必须确定该变量的类型
int i = 10; // 显式类型，变量类型明确告知编译器

// 注意，这两者都表示相同类型的变量 (int) 且值相同 (10)。

```

与其他类型的变量不同，使用此关键字定义的变量在声明时必须初始化。这是因为var关键字表示隐式类型变量。

```

var i;
i = 10;

// 这段代码不会运行，因为声明时未初始化。

```

关键字var也可以用来动态创建新数据类型。这些新数据类型被称为匿名类型。它们非常有用，因为允许用户定义一组属性，而无需先显式声明任何对象类型。

普通匿名类型

```
var a = new { number = 1, text = "hi" };
```

返回匿名类型的 LINQ 查询

```

public class Dog
{
    public string Name { get; set; }
    public int Age { get; set; }
}

public class DogWithBreed
{
    public Dog Dog { get; set; }
    public string BreedName { get; set; }
}

```

```

case 6:
case 7:
case 8:
    Console.WriteLine("Summer");
    break;
case 9:
case 10:
case 11:
    Console.WriteLine("Autumn");
    break;
default:
    Console.WriteLine("Incorrect month index");
    break;
}

```

A case can only be labeled by a value known at compile time (e.g. 1, "str", Enum.A), so a variable isn't a valid case label, but a const or an Enum value is (as well as any literal value).

Section 52.37: var

An implicitly-typed local variable that is strongly typed just as if the user had declared the type. Unlike other variable declarations, the compiler determines the type of variable that this represents based on the value that is assigned to it.

```

var i = 10; // implicitly typed, the compiler must determine what type of variable this is
int i = 10; // explicitly typed, the type of variable is explicitly stated to the compiler

// Note that these both represent the same type of variable (int) with the same value (10).

```

Unlike other types of variables, variable definitions with this keyword need to be initialized when declared. This is due to the var keyword representing an implicitly-typed variable.

```

var i;
i = 10;

// This code will not run as it is not initialized upon declaration.

```

The var keyword can also be used to create new datatypes on the fly. These new datatypes are known as *anonymous types*. They are quite useful, as they allow a user to define a set of properties without having to explicitly declare any kind of object type first.

Plain anonymous type

```
var a = new { number = 1, text = "hi" };
```

LINQ query that returns an anonymous type

```

public class Dog
{
    public string Name { get; set; }
    public int Age { get; set; }
}

public class DogWithBreed
{
    public Dog Dog { get; set; }
    public string BreedName { get; set; }
}

```

```

    }

    public void GetDogsWithBreedNames()
    {
        var db = new DogDataContext(ConnectionString);
        var result = from d in db.Dogs
                     join b in db.Breeds on d.BreedId equals b.BreedId
                     select new
                     {
                         DogName = d.Name,
                         BreedName = b.BreedName
                     };
        DoStuff(result);
    }
}

```

你可以在 foreach 语句中使用 var 关键字

```

public bool hasItemInList(List<String> list, string stringToSearch)
{
    foreach(var item in list)
    {
        if( ( (string)item ).equals(stringToSearch) )
            return true;
    }

    return false;
}

```

第 52.38 节 : when

when 是在 C# 6 中新增的关键字，用于异常过滤。

在引入when关键字之前，你可以为每种异常类型设置一个catch子句；随着该关键字的加入，现在可以实现更细粒度的控制。

当一个when表达式附加到一个catch分支时，只有当when条件为true时，catch子句才会被执行。可以有多个catch子句具有相同的异常类类型，但不同的when条件。

```

private void CatchException(Action action)
{
    try
    {
        action.Invoke();
    }

    // 异常过滤器
    catch (Exception ex) when (ex.Message.Contains("when"))
    {
        Console.WriteLine("捕获了带有 when 的异常");
    }

    catch (Exception ex)
    {
        Console.WriteLine("捕获了不带 when 的异常");
    }
}

```

```

    }

    public void GetDogsWithBreedNames()
    {
        var db = new DogDataContext(ConnectionString);
        var result = from d in db.Dogs
                     join b in db.Breeds on d.BreedId equals b.BreedId
                     select new
                     {
                         DogName = d.Name,
                         BreedName = b.BreedName
                     };
        DoStuff(result);
    }
}

```

You can use var keyword in foreach statement

```

public bool hasItemInList(List<String> list, string stringToSearch)
{
    foreach(var item in list)
    {
        if( ( (string)item ).equals(stringToSearch) )
            return true;
    }

    return false;
}

```

Section 52.38: when

The when is a keyword added in **C# 6**, and it is used for exception filtering.

Before the introduction of the when keyword, you could have had one catch clause for each type of exception; with the addition of the keyword, a more fine-grained control is now possible.

A when expression is attached to a **catch** branch, and only if the when condition is **true**, the **catch** clause will be executed. It is possible to have several **catch** clauses with the same exception class types, and different when conditions.

```

private void CatchException(Action action)
{
    try
    {
        action.Invoke();
    }

    // exception filter
    catch (Exception ex) when (ex.Message.Contains("when"))
    {
        Console.WriteLine("Caught an exception with when");
    }

    catch (Exception ex)
    {
        Console.WriteLine("Caught an exception without when");
    }
}

```

```
private void Method1() { throw new Exception("带有 when 的异常消息"); }
private void Method2() { throw new Exception("一般异常消息"); }
```

```
CatchException(Method1);
CatchException(Method2);
```

第52.39节：锁

lock 为一段代码块提供线程安全，确保同一

进程中只有一个线程可以访问该代码块。示例：

```
private static object _lockObj = new object();
static void Main(string[] args)
{
    Task.Run(() => TaskWork());
    Task.Run(() => TaskWork());
    Task.Run(() => TaskWork());

    Console.ReadKey();
}

private static void TaskWork()
{
    lock(_lockObj)
    {
        Console.WriteLine("Entered");

        Task.Delay(3000);
        Console.WriteLine("Done Delaying");

        // 安全访问共享资源

        Console.WriteLine("Leaving");
    }
}
```

输出：

```
进入
完成延迟
离开
进入
完成延迟
离开
进入
完成延迟
离开
```

用例：

每当你有一段代码块，如果被多个线程同时执行可能会产生副作用时，可以使用**lock**关键字和共享同步对象（示例中的`_objLock`）来防止这种情况。

注意`_objLock`不能为null，且执行该代码的多个线程必须使用相同的对象实例（要么将其设为static字段，要么两个线程使用相同的类实例）

从编译器角度看，**lock**关键字是语法糖，会被替换为`Monitor.Enter(_lockObj)`；和

```
private void Method1() { throw new Exception("message for exception with when"); }
private void Method2() { throw new Exception("message for general exception"); }
```

```
CatchException(Method1);
CatchException(Method2);
```

Section 52.39: lock

lock 提供线程安全性，确保在同一进程中只能有一个线程访问该代码块。示例：

```
private static object _lockObj = new object();
static void Main(string[] args)
{
    Task.Run(() => TaskWork());
    Task.Run(() => TaskWork());
    Task.Run(() => TaskWork());

    Console.ReadKey();
}

private static void TaskWork()
{
    lock(_lockObj)
    {
        Console.WriteLine("Entered");

        Task.Delay(3000);
        Console.WriteLine("Done Delaying");

        // Access shared resources safely

        Console.WriteLine("Leaving");
    }
}
```

Output:

```
Entered
Done Delaying
Leaving
Entered
Done Delaying
Leaving
Entered
Done Delaying
Leaving
```

Use cases:

每当你有一段代码块，如果被多个线程同时执行可能会产生副作用时，可以使用**lock**关键字和共享同步对象（示例中的`_objLock`）来防止这种情况。

Note that `_objLock` can't be null and multiple threads executing the code must use the same object instance (either by making it a static field, or by using the same class instance for both threads)

从编译器角度看，**lock**关键字是语法糖，会被替换为`Monitor.Enter(_lockObj)`；和

Monitor.Exit(_lockObj);。因此，如果你用这两个方法包围代码块替代lock，结果是一样的。你可以在“Syntactic sugar in C# - lock example”中看到实际代码

第52.40节 : uint

无符号整数，或称uint，是一种只能存储正整数的数值数据类型。顾名思义，它表示一个无符号的32位整数。 uint关键字本身是公共类型系统类型System.UInt32的别名。该数据类型存在于mscorlib.dll中，每个C#项目创建时都会隐式引用它。它占用四个字节的内存空间。

无符号整数可以存储从0到4,294,967,295的任何值。

如何声明无符号整数的示例及错误示范

```
uint i = 425697; // 有效表达式, 明确告知编译器
var i1 = 789247U; // 有效表达式, 后缀使编译器能确定数据类型
uint x = 3.0; // 错误, 没有隐式转换
```

请注意：根据 Microsoft 的建议，尽可能使用 int 数据类型，因为 uint 数据类型不符合 CLS 规范。

第52.41节 : if, if...else, if... else if

if 语句用于控制程序流程。 if 语句根据 Boolean 表达式的值确定执行哪个语句。

对于单条语句，大括号{}是可选的，但建议使用。

```
int a = 4;
if(a % 2 == 0)
{
    Console.WriteLine("a 包含一个偶数");
}
// 输出: "a 包含一个偶数"
```

if 语句也可以有一个 else 子句，当条件为假时执行该子句：

```
int a = 5;
if(a % 2 == 0)
{
    Console.WriteLine("a 包含一个偶数");
}
else
{
    Console.WriteLine("a contains an odd number");
}
// 输出: "a contains an odd number"
```

if...else if 结构允许你指定多个条件：

```
int a = 9;
if(a % 2 == 0)
{
    Console.WriteLine("a 包含一个偶数");
}
else if(a % 3 == 0)
```

Monitor.Exit(_lockObj);。So if you replace the lock by surrounding the block of code with these two methods, you would get the same results. You can see actual code in Syntactic sugar in C# - lock example

Section 52.40: uint

An **unsigned integer**, or **uint**, is a numeric datatype that only can hold positive integers. Like its name suggests, it represents an unsigned 32-bit integer. The **uint** keyword itself is an alias for the Common Type System type **System.UInt32**. This datatype is present in **mscorlib.dll**, which is implicitly referenced by every C# project when you create them. It occupies four bytes of memory space.

Unsigned integers can hold any value from 0 to 4,294,967,295.

Examples on how and now not to declare unsigned integers

```
uint i = 425697; // Valid expression, explicitly stated to compiler
var i1 = 789247U; // Valid expression, suffix allows compiler to determine datatype
uint x = 3.0; // Error, there is no implicit conversion
```

Please note: According to [Microsoft](#), it is recommended to use the **int** datatype wherever possible as the **uint** datatype is not CLS-compliant.

Section 52.41: if, if...else, if... else if

The if statement is used to control the flow of the program. An if statement identifies which statement to run based on the value of a Boolean expression.

For a single statement, the braces{} are optional but recommended.

```
int a = 4;
if(a % 2 == 0)
{
    Console.WriteLine("a contains an even number");
}
// output: "a contains an even number"
```

The if can also have an else clause, that will be executed in case the condition evaluates to false:

```
int a = 5;
if(a % 2 == 0)
{
    Console.WriteLine("a contains an even number");
}
else
{
    Console.WriteLine("a contains an odd number");
}
// output: "a contains an odd number"
```

The if...else if construct lets you specify multiple conditions:

```
int a = 9;
if(a % 2 == 0)
{
    Console.WriteLine("a contains an even number");
}
else if(a % 3 == 0)
```

```

{
Console.WriteLine("a contains an odd number that is a multiple of 3");
}
else
{
Console.WriteLine("a contains an odd number");
}
// 输出: "a contains an odd number that is a multiple of 3"

```

重要的是要注意，在上述示例中，如果某个条件满足，控制流会跳过其他测试，直接跳到该 `if .. else` 结构的末尾。因此，如果使用 `if .. else if` 结构，测试的顺序非常重要

C# 布尔表达式使用短路求值。这在评估条件可能有副作用的情况下非常重要：

```

if (someBooleanMethodWithSideEffects() && someOtherBooleanMethodWithSideEffects()) {
    //...
}

```

不能保证 `someOtherBooleanMethodWithSideEffects` 实际上会被执行。

在前面的条件确保“安全”评估后续条件的情况下，这一点也很重要。例如：

```

if (someCollection != null && someCollection.Count > 0) {
    // ..
}

```

在这种情况下，顺序非常重要，因为如果我们颠倒顺序：

```

if (someCollection.Count > 0 && someCollection != null) {
}

```

如果 `someCollection` 是 `null`，将会抛出 `NullReferenceException`。

第52.42节：static

`static` 修饰符用于声明静态成员，该成员无需实例化即可访问，而是通过其名称直接访问，例如 `DateTime.Now`。

`static` 可以用于类、字段、方法、属性、运算符、事件和构造函数。

虽然一个类的实例包含该类所有实例字段的独立副本，但每个静态字段只有一个副本。

```

class A
{
    static public int count = 0;

    public A()
    {
        count++;
    }
}

class Program
{
    static void Main(string[] args)
}

```

```

{
    Console.WriteLine("a contains an odd number that is a multiple of 3");
}
else
{
    Console.WriteLine("a contains an odd number");
}
// output: "a contains an odd number that is a multiple of 3"

```

Important to note that if a condition is met in the above example , the control skips other tests and jumps to the end of that particular if else construct. So, the order of tests is important if you are using if .. else if construct

C# Boolean expressions use [short-circuit evaluation](#). This is important in cases where evaluating conditions may have side effects:

```

if (someBooleanMethodWithSideEffects() && someOtherBooleanMethodWithSideEffects()) {
    //...
}

```

There's no guarantee that `someOtherBooleanMethodWithSideEffects` will actually run.

It's also important in cases where earlier conditions ensure that it's "safe" to evaluate later ones. For example:

```

if (someCollection != null && someCollection.Count > 0) {
    // ..
}

```

The order is very important in this case because, if we reverse the order:

```

if (someCollection.Count > 0 && someCollection != null) {
}

```

it will throw a `NullReferenceException` if `someCollection` is `null`.

Section 52.42: static

The `static` modifier is used to declare a static member, which does not need to be instantiated in order to be accessed, but instead is accessed simply through its name, i.e. `DateTime.Now`.

`static` can be used with classes, fields, methods, properties, operators, events, and constructors.

While an instance of a class contains a separate copy of all instance fields of the class, there is only one copy of each static field.

```

class A
{
    static public int count = 0;

    public A()
    {
        count++;
    }
}

class Program
{
    static void Main(string[] args)
}

```

```

{
A a = new A();
A b = new A();
A c = new A();

Console.WriteLine(A.count); // 3
}

```

count 等于 A 类的实例总数。

`static` 修饰符也可用于声明类的静态构造函数，用于初始化静态数据或运行只需调用一次的代码。
静态构造函数在类首次被引用之前调用。

```

class A
{
    static public DateTime InitializationTime;

    // 静态构造函数
    static A()
    {
        InitializationTime = DateTime.Now;
        // 保证只运行一次
        Console.WriteLine(InitializationTime.ToString());
    }
}

```

一个静态类用`static`关键字标记，可以作为一组方法的有益容器使用，这些方法处理参数，但不一定需要绑定到实例。由于该类的`static`特性，它不能被实例化，但可以包含一个`static`构造函数。静态类的一些特性包括：

- 不能被继承
- 不能继承除`Object`之外的任何类
- 可以包含静态构造函数，但不能包含实例构造函数
- 只能包含静态成员
- 已密封

编译器也很友好，会让开发者知道类中是否存在任何实例成员。一个例子是一个在美国和加拿大度量单位之间转换的静态类：

```

静态类 ConversionHelper {
    私有静态双精度浮点数 oneGallonPerLitreRate = 0.264172;

    public static double litreToGallonConversion(int litres) {
        return litres * oneGallonPerLitreRate;
    }
}

```

当类被声明为静态时：

```

public static class Functions
{
    public static int Double(int value)
    {
        return value + value;
    }
}

```

```

{
A a = new A();
A b = new A();
A c = new A();

Console.WriteLine(A.count); // 3
}

```

count equals to the total number of instances of A class.

The `static` modifier can also be used to declare a static constructor for a class, to initialize static data or run code that only needs to be called once. Static constructors are called before the class is referenced for the first time.

```

class A
{
    static public DateTime InitializationTime;

    // Static constructor
    static A()
    {
        InitializationTime = DateTime.Now;
        // Guaranteed to only run once
        Console.WriteLine(InitializationTime.ToString());
    }
}

```

A `static class` is marked with the `static` keyword, and can be used as a beneficial container for a set of methods that work on parameters, but don't necessarily require being tied to an instance. Because of the `static` nature of the class, it cannot be instantiated, but it can contain a `static` constructor. Some features of a `static class` include:

- Can't be inherited
- Can't inherit from anything other than `Object`
- Can contain a static constructor but not an instance constructor
- Can only contain static members
- Is sealed

The compiler is also friendly and will let the developer know if any instance members exist within the class. An example would be a static class that converts between US and Canadian metrics:

```

static class ConversionHelper {
    private static double oneGallonPerLitreRate = 0.264172;

    public static double litreToGallonConversion(int litres) {
        return litres * oneGallonPerLitreRate;
    }
}

```

When classes are declared static:

```

public static class Functions
{
    public static int Double(int value)
    {
        return value + value;
    }
}

```

```
}
```

类中的所有函数、属性或成员也需要声明为静态。本质上，静态类允许你创建逻辑上分组在一起的函数集合。

自 C#6 起，static 也可以与 using 一起使用来导入静态成员和方法。这样就可以在不使用类名的情况下调用它们。

旧方法，不使用 using static：

```
using System;

public class ConsoleApplication
{
    public static void Main()
    {
        Console.WriteLine("Hello World!"); //WriteLine 是属于静态类 Console 的方法
    }
}
```

使用 using static 的示例

```
using static System.Console;

public class ConsoleApplication
{
    public static void Main()
    {
        WriteLine("Hello World!"); //WriteLine 是属于静态类 Console 的方法
    }
}
```

缺点

虽然静态类非常有用，但它们也有一些注意事项：

- 一旦调用了静态类，该类会被加载到内存中，直到包含该静态类的应用程序域（AppDomain）被卸载之前，垃圾回收器无法回收它。
- 静态类不能实现接口。

第 52.43 节：internal

internal 关键字是类型和类型成员的访问修饰符。internal 类型或成员仅在同一程序集中的文件内可访问

用法：

```
public class BaseClass
{
    // 仅在同一程序集中可访问
    internal static int x = 0;
}
```

不同访问修饰符之间的区别在此处进行了说明

```
}
```

all function, properties or members within the class also need to be declared static. No instance of the class can be created. In essence a static class allows you to create bundles of functions that are grouped together logically.

Since C#6 static can also be used alongside using to import static members and methods. They can be used then without class name.

Old way, without using static:

```
using System;

public class ConsoleApplication
{
    public static void Main()
    {
        Console.WriteLine("Hello World!"); //Writeline is method belonging to static class Console
    }
}
```

Example with using static

```
using static System.Console;

public class ConsoleApplication
{
    public static void Main()
    {
        WriteLine("Hello World!"); //Writeline is method belonging to static class Console
    }
}
```

Drawbacks

While static classes can be incredibly useful, they do come with their own caveats:

- Once the static class has been called, the class is loaded into memory and cannot be run through the garbage collector until the AppDomain housing the static class is unloaded.
- A static class cannot implement an interface.

Section 52.43: internal

The internal keyword is an access modifier for types and type members. Internal types or members are **accessible only within files in the same assembly**

usage:

```
public class BaseClass
{
    // Only accessible within the same assembly
    internal static int x = 0;
}
```

The difference between different access modifiers is clarified [here](#)

访问修饰符

public

该类型或成员可以被同一程序集中或引用该程序集的另一个程序集中的任何代码访问。

私有的

该类型或成员只能被同一类或结构体中的代码访问。

protected

该类型或成员只能被同一类或结构体中的代码访问，或者被派生类访问。

internal

该类型或成员可以被同一程序集中任何代码访问，但不能被另一个程序集访问。

protected internal

该类型或成员可以被同一程序集中任何代码访问，或者被另一个程序集中的任何派生类访问。

当未设置访问修饰符时，将使用默认访问修饰符。因此，即使未设置，也总会有某种形式的访问修饰符。

第52.44节：using

"using"关键字有两种用法，分别是"using语句"和"using指令"：

1. using 语句:

using 关键字确保实现 IDisposable 接口的对象在使用后被正确释放。using 语句有单独的专题介绍。

2.using 指令

using 指令有三种用法，详见msdn 页面关于 using 指令的说明。using 指令有单独的专题介绍。

Access modifiers

public

The type or member can be accessed by any other code in the same assembly or another assembly that references it.

private

The type or member can only be accessed by code in the same class or struct.

protected

The type or member can only be accessed by code in the same class or struct, or in a derived class.

internal

The type or member can be accessed by any code in the same assembly, but not from another assembly.

protected internal

The type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.

When **no access modifier** is set, a default access modifier is used. So there is always some form of access modifier even if it's not set.

Section 52.44: using

There are two types of **using** keyword usage, **using statement** and **using directive**:

1. using statement:

The **using** keyword ensures that objects that implement the **IDisposable** interface are properly disposed after usage. There is a separate topic for the using statement

2. using directive

The **using** directive has three usages, see the [msdn page for the using directive](#). There is a separate topic for the using directive.

第 52.45 节 : where

where 在 C# 中有两个用途：泛型参数的类型约束，以及 LINQ 查询的过滤。

Section 52.45: where

where can serve two purposes in C#: type constraining in a generic argument, and filtering LINQ queries.

在泛型类中，考虑如下情况

```
public class Cup<T>
{
    // ...
}
```

T 被称为类型参数。类定义可以对实际传入的类型施加约束
T.

可以应用以下几种约束：

- 值类型
- 引用类型
- 默认构造函数
- 继承和实现

值类型

在这种情况下，只能提供struct（这包括“原始”数据类型，如int、boolean等）

```
public class Cup<T> 其中 T : struct
{
    // ...
}
```

引用类型

在这种情况下，只能提供类类型

```
public class Cup<T> 其中 T : class
{
    // ...
}
```

混合值/引用类型

有时希望将类型参数限制为数据库中可用的类型，这些类型通常映射为
值类型和字符串。由于必须满足所有类型限制，因此无法指定`where T : struct`或
`string`（这不是有效语法）。一种解决方法是将类型参数限制为[IConvertible](#)，该接口内置了
“... Boolean、SByte、Byte、Int16、UInt16、Int32、UInt32、Int64、UInt64、Single、Double、Decimal、DateTime、
Char和String。”其他对象也可能实现IConvertible，尽管在实际中这种情况较少见。

```
public class Cup<T> where T : IConvertible
{
    // ...
}
```

默认构造函数

只允许包含默认构造函数的类型。这包括值类型和包含默认（无参数）构造函数的类

```
public class Cup<T> where T : new
{
    // ...
}
```

In a generic class, let's consider

```
public class Cup<T>
{
    // ...
}
```

T is called a type parameter. The class definition can impose constraints on the actual types that can be supplied for
T.

The following kinds of constraints can be applied:

- value type
- reference type
- default constructor
- inheritance and implementation

value type

In this case only `structs` (this includes 'primitive' data types such as `int`, `boolean` etc) can be supplied

```
public class Cup<T> where T : struct
{
    // ...
}
```

reference type

In this case only class types can be supplied

```
public class Cup<T> where T : class
{
    // ...
}
```

hybrid value/reference type

Occasionally it is desired to restrict type arguments to those available in a database, and these will usually map to
value types and strings. As all type restrictions must be met, it is not possible to specify `where T : struct` or
`string` (this is not valid syntax). A workaround is to restrict type arguments to [IConvertible](#) which has built in
types of "... Boolean, SByte, Byte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double, Decimal, DateTime,
Char, and String." It is possible other objects will implement IConvertible, though this is rare in practice.

```
public class Cup<T> where T : IConvertible
{
    // ...
}
```

default constructor

Only types that contain a default constructor will be allowed. This includes value types and classes that contain a
default (parameterless) constructor

```
public class Cup<T> where T : new
{
    // ...
}
```

```
}
```

继承和实现

只允许继承自某个基类或实现某个接口的类型

```
public class Cup<T> where T : Beverage
{
    // ...
}
```

```
public class Cup<T> where T : IBeer
{
    // ...
}
```

约束甚至可以引用另一个类型参数：

```
public class Cup<T, U> where U : T
{
    // ...
}
```

类型参数可以指定多个约束：

```
public class Cup<T> where T : class, new()
{
    // ...
}
```

前面的示例展示了类定义上的泛型约束，但约束可以用于任何提供类型参数的地方：类、结构体、接口、方法等。

where 也可以是 LINQ 子句。在这种情况下，它类似于 SQL 中的 WHERE：

```
int[] nums = { 5, 2, 1, 3, 9, 8, 6, 7, 2, 0 };

var query =
    from num in nums
    where num < 5
    select num;

foreach (var n in query)
{
    Console.WriteLine(n + " ");
}
// 输出 2 1 3 2 0
```

第52.46节：int

`int` 是 `System.Int32` 的别名，它是有符号32位整数的数据类型。该数据类型位于 `mscorlib.dll` 中，每个C#项目创建时都会隐式引用该文件。

范围：-2,147,483,648 到 2,147,483,647

```
int int1 = -10007;
```

```
}
```

inheritance and implementation

Only types that inherit from a certain base class or implement a certain interface can be supplied.

```
public class Cup<T> where T : Beverage
{
    // ...
}
```

```
public class Cup<T> where T : IBeer
{
    // ...
}
```

The constraint can even reference another type parameter:

```
public class Cup<T, U> where U : T
{
    // ...
}
```

Multiple constraints can be specified for a type argument:

```
public class Cup<T> where T : class, new()
{
    // ...
}
```

The previous examples show generic constraints on a class definition, but constraints can be used anywhere a type argument is supplied: classes, structs, interfaces, methods, etc.

`where` 也可以是 LINQ 子句。在这种情况下，它类似于 SQL 中的 WHERE：

```
int[] nums = { 5, 2, 1, 3, 9, 8, 6, 7, 2, 0 };

var query =
    from num in nums
    where num < 5
    select num;

foreach (var n in query)
{
    Console.WriteLine(n + " ");
}
// prints 2 1 3 2 0
```

Section 52.46: int

`int` is an alias for `System.Int32`, which is a data type for signed 32-bit integers. This data type can be found in `mscorlib.dll` which is implicitly referenced by every C# project when you create them.

Range: -2,147,483,648 to 2,147,483,647

```
int int1 = -10007;
```

```
var int2 = 2132012521;
```

第52.47节 : ulong

用于无符号64位整数的关键字。它表示System.UInt64数据类型，位于mscorlib.dll中，每个C#项目创建时都会隐式引用该dll。

范围：0 到 18,446,744,073,709,551,615

```
ulong veryLargeInt = 18446744073609451315;
var anotherVeryLargeInt = 15446744063609451315UL;
```

第52.48节 : true, false

true 和 false 关键字有两种用法：

1. 作为布尔字面值

```
var myTrueBool = true;
var myFalseBool = false;
```

2. 作为可以重载的运算符

```
public static bool operator true(MyClass x)
{
    return x.value >= 0;
}

public static bool operator false(MyClass x)
{
    return x.value < 0;
}
```

在引入Nullable类型之前，重载false运算符是有用的，尤其是在C# 2.0之前。

重载true运算符的类型，必须同时重载false运算符。

第52.49节 : 结构体

结构体 (struct) 类型是一种值类型，通常用于封装一小组相关变量，例如矩形的坐标或库存中某个物品的特征。

类 (Classes) 是引用类型，结构体 (structs) 是值类型。

```
using static System.Console;

namespace ConsoleApplication1
{
    struct Point
    {
        public int X;
        public int Y;

        public override string ToString()
        {
            return $"X = {X}, Y = {Y}";
        }
}
```

```
var int2 = 2132012521;
```

Section 52.47: ulong

Keyword used for unsigned 64-bit integers. It represents System.UInt64 data type found in mscorlib.dll which is implicitly referenced by every C# project when you create them.

Range: 0 to 18,446,744,073,709,551,615

```
ulong veryLargeInt = 18446744073609451315;
var anotherVeryLargeInt = 15446744063609451315UL;
```

Section 52.48: true, false

The `true` and `false` keywords have two uses:

1. As literal Boolean values

```
var myTrueBool = true;
var myFalseBool = false;
```

2. As operators that can be overloaded

```
public static bool operator true(MyClass x)
{
    return x.value >= 0;
}

public static bool operator false(MyClass x)
{
    return x.value < 0;
}
```

Overloading the `false` operator was useful prior to C# 2.0, before the introduction of Nullable types.

A type that overloads the `true` operator, must also overload the `false` operator.

Section 52.49: struct

A `struct` type is a value type that is typically used to encapsulate small groups of related variables, such as the coordinates of a rectangle or the characteristics of an item in an inventory.

`Classes` are reference types, `structs` are value types.

```
using static System.Console;

namespace ConsoleApplication1
{
    struct Point
    {
        public int X;
        public int Y;

        public override string ToString()
        {
            return $"X = {X}, Y = {Y}";
        }
}
```

```

    }

    public void Display(string name)
    {
        WriteLine(name + " : " + ToString());
    }
}

class Program
{
    static void Main()
    {
        var point1 = new Point {X = 10, Y = 20};
        // it's not a reference but value type
        var point2 = point1;
        point2.X = 777;
        point2.Y = 888;
        point1.Display(nameof(point1)); // point1: X = 10, Y = 20
        point2.Display(nameof(point2)); // point2: X = 777, Y = 888

        ReadKey();
    }
}

```

结构体也可以包含构造函数、常量、字段、方法、属性、索引器、运算符、事件和嵌套类型，尽管如果需要多个此类成员，应该考虑将你的类型定义为类。

微软关于何时使用结构体、何时使用类的一些建议：

考虑

如果类型的实例较小且通常生命周期较短，或者通常嵌入在其他对象中，考虑定义结构体而非类。

避免

除非类型具备以下所有特征，否则避免定义结构体：

- 它在逻辑上表示单一值，类似于基本类型 (int、double 等)
- 它的实例大小小于16字节。
- 它是不可变的。
- 不需要频繁进行装箱操作。

第52.50节：extern

extern 关键字用于声明在外部实现的方法。它可以与 DllImport 属性结合使用，通过互操作服务调用非托管代码，在这种情况下会带有static 修饰符

例如：

```

using System.Runtime.InteropServices;
public class MyClass
{
    [DllImport("User32.dll")]
    private static extern int SetForegroundWindow(IntPtr point);
}

```

```

    }

    public void Display(string name)
    {
        WriteLine(name + " : " + ToString());
    }
}

class Program
{
    static void Main()
    {
        var point1 = new Point {X = 10, Y = 20};
        // it's not a reference but value type
        var point2 = point1;
        point2.X = 777;
        point2.Y = 888;
        point1.Display(nameof(point1)); // point1: X = 10, Y = 20
        point2.Display(nameof(point2)); // point2: X = 777, Y = 888

        ReadKey();
    }
}

```

Structs can also contain constructors, constants, fields, methods, properties, indexers, operators, events, and nested types, although if several such members are required, you should consider making your type a class instead.

Some suggestions from MS on when to use struct and when to use class:

CONSIDER

defining a struct instead of a class if instances of the type are small and commonly short-lived or are commonly embedded in other objects.

AVOID

defining a struct unless the type has all of the following characteristics:

- It logically represents a single value, similar to primitive types (int, double, etc.)
- It has an instance size under 16 bytes.
- It is immutable.
- It will not have to be boxed frequently.

Section 52.50: extern

The **extern** keyword is used to declare methods that are implemented externally. This can be used in conjunction with the **DllImport** attribute to call into unmanaged code using Interop services. which in this case it will come with **static** modifier

For Example:

```

using System.Runtime.InteropServices;
public class MyClass
{
    [DllImport("User32.dll")]
    private static extern int SetForegroundWindow(IntPtr point);
}

```

```
public void ActivateProcessWindow(Process p)
{
    SetForegroundWindow(p.MainWindowHandle);
}
```

这使用了从 User32.dll 库导入的 SetForegroundWindow 方法。这也同样可以用来定义外部程序集别名，使我们能够从单个程序集引用同一组件的不同版本。

要引用具有相同完全限定类型名称的两个程序集，必须在命令提示符下指定别名，方法如下：

```
/r:GridV1=grid.dll
/r:GridV2=grid20.dll
```

这会创建外部别名 GridV1 和 GridV2。要在程序中使用这些别名，请使用 `extern` 关键字引用它们。例如：

```
extern alias GridV1;
extern alias GridV2;
```

第 52.51 节：bool

用于存储布尔值 true 和 false 的关键字。bool 是 System.Boolean 的别名。

bool 的默认值是 false。

```
bool b; // 默认值是 false
b = true; // true
b = ((5 + 2) == 6); // false
```

若要使 bool 允许空值，必须将其初始化为 bool?。

bool? 的默认值是 null。

```
bool? a // 默认值为 null
```

第 52.52 节：接口

一个接口包含方法、属性和事件的签名。派生类定义成员，因为接口只包含成员的声明。

接口使用 interface 关键字声明。

```
interface IProduct
{
    decimal Price { get; }

    class Product : IProduct
    {
        const decimal vat = 0.2M;

        public Product(decimal price)
```

```
public void ActivateProcessWindow(Process p)
{
    SetForegroundWindow(p.MainWindowHandle);
}
```

This uses the SetForegroundWindow method imported from the User32.dll library.

This can also be used to define an external assembly alias, which let us to reference different versions of same components from single assembly.

To reference two assemblies with the same fully-qualified type names, an alias must be specified at a command prompt, as follows:

```
/r:GridV1=grid.dll
/r:GridV2=grid20.dll
```

This creates the external aliases GridV1 and GridV2. To use these aliases from within a program, reference them by using the `extern` keyword. For example:

```
extern alias GridV1;
extern alias GridV2;
```

Section 52.51: bool

Keyword for storing the Boolean values `true` and `false`. bool is an alias of System.Boolean.

The default value of a bool is false.

```
bool b; // default value is false
b = true; // true
b = ((5 + 2) == 6); // false
```

For a bool to allow null values it must be initialized as a bool?.

The default value of a bool? is null.

```
bool? a // default value is null
```

Section 52.52: interface

An `interface` contains the `signatures` of methods, properties and events. The derived classes defines the members as the interface contains only the declaration of the members.

An interface is declared using the `interface` keyword.

```
interface IProduct
{
    decimal Price { get; }

    class Product : IProduct
    {
        const decimal vat = 0.2M;

        public Product(decimal price)
```

```

    {
        _price = price;
    }

    private decimal _price;
    public decimal Price { get { return _price * (1 + vat); } }
}

```

第52.53节：委托

委托是表示对方法引用的类型。它们用于将方法作为参数传递给其他方法。

委托可以持有静态方法、实例方法、匿名方法或lambda表达式。

```

class DelegateExample
{
    public void Run()
    {
        //使用类方法
        InvokeDelegate( WriteToConsole );

        //使用匿名方法
        DelegateInvoker di = delegate ( string input )
        {
            Console.WriteLine( string.Format( "di: {0}" , input ) );
            return true;
        };
        调用委托( di );

        //使用 lambda 表达式
        调用委托( input => false );
    }

    public delegate bool 委托调用器( string input );

    public void 调用委托(委托调用器 func)
    {
        var ret = func( "hello world" );
        Console.WriteLine( string.Format( " > 委托返回了 {0}" , ret ) );
    }

    public bool 写入控制台( string input )
    {
        Console.WriteLine( string.Format( "写入控制台: '{0}'" , input ) );
        return true;
    }
}

```

将方法赋值给委托时，重要的是方法必须具有相同的返回类型和参数。这与“普通”方法重载不同，后者仅由参数定义方法签名。

事件是建立在委托之上的。

第52.54节：未检查

未检查（unchecked）关键字阻止编译器检查溢出/下溢。

```

    {
        _price = price;
    }

    private decimal _price;
    public decimal Price { get { return _price * (1 + vat); } }
}

```

Section 52.53: delegate

Delegates are types that represent a reference to a method. They are used for passing methods as arguments to other methods.

Delegates can hold static methods, instance methods, anonymous methods, or lambda expressions.

```

class DelegateExample
{
    public void Run()
    {
        //using class method
        InvokeDelegate( WriteToConsole );

        //using anonymous method
        DelegateInvoker di = delegate ( string input )
        {
            Console.WriteLine( string.Format( "di: {0}" , input ) );
            return true;
        };
        InvokeDelegate( di );

        //using lambda expression
        InvokeDelegate( input => false );
    }

    public delegate bool DelegateInvoker( string input );

    public void InvokeDelegate(DelegateInvoker func)
    {
        var ret = func( "hello world" );
        Console.WriteLine( string.Format( " > delegate returned {0}" , ret ) );
    }

    public bool WriteToConsole( string input )
    {
        Console.WriteLine( string.Format( "WriteToConsole: '{0}'" , input ) );
        return true;
    }
}

```

When assigning a method to a delegate it is important to note that the method must have the same return type as well as parameters. This differs from 'normal' method overloading, where only the parameters define the signature of the method.

Events are built on top of delegates.

Section 52.54: unchecked

The `unchecked` keyword prevents the compiler from checking for overflows/underflows.

例如：

```
const int ConstantMax = int.MaxValue;
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

没有unchecked关键字，这两个加法操作都无法编译。

什么时候有用？

这很有用，因为它可以加快那些绝对不会溢出的计算速度，因为检查溢出需要时间，或者当溢出/下溢是期望的行为时（例如，在生成哈希码时）。

第52.55节：ushort

一种用于存储16位正整数的数值类型。ushort是System.UInt16的别名，占用2字节内存。

有效范围是0到65535。

```
ushort a = 50; // 50
ushort b = 65536; // 错误，无法转换
ushort c = unchecked((ushort)65536); // 溢出 (回绕到0)
```

第52.56节：sizeof

用于获取非托管类型的字节大小

```
int byteSize = sizeof(byte) // 1
int sbyteSize = sizeof(sbyte) // 1
int shortSize = sizeof(short) // 2
int ushortSize = sizeof(ushort) // 2
int intSize = sizeof(int) // 4
int uintSize = sizeof(uint) // 4
int longSize = sizeof(long) // 8
int ulongSize = sizeof(ulong) // 8
int charSize = sizeof(char) // 2 (Unicode)
int floatSize = sizeof(float) // 4
int doubleSize = sizeof(double) // 8
int decimalSize = sizeof(decimal) // 16
int boolSize = sizeof(bool) // 1
```

第52.57节：in

in 关键字有三种用法：

a) 作为foreach语句语法的一部分，或作为LINQ查询语法的一部分

```
foreach (var member in sequence)
{
    // ...
}
```

For example:

```
const int ConstantMax = int.MaxValue;
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

Without the `unchecked` keyword, neither of the two addition operations will compile.

When is this useful?

This is useful as it may help speed up calculations that definitely will not overflow since checking for overflow takes time, or when an overflow/underflow is desired behavior (for instance, when generating a hash code).

Section 52.55: ushort

A numeric type used to store 16-bit positive integers. `ushort` is an alias for `System.UInt16`, and takes up 2 bytes of memory.

Valid range is 0 to 65535.

```
ushort a = 50; // 50
ushort b = 65536; // Error, cannot be converted
ushort c = unchecked((ushort)65536); // Overflows (wraps around to 0)
```

Section 52.56: sizeof

Used to obtain the size in bytes for an unmanaged type

```
int byteSize = sizeof(byte) // 1
int sbyteSize = sizeof(sbyte) // 1
int shortSize = sizeof(short) // 2
int ushortSize = sizeof(ushort) // 2
int intSize = sizeof(int) // 4
int uintSize = sizeof(uint) // 4
int longSize = sizeof(long) // 8
int ulongSize = sizeof(ulong) // 8
int charSize = sizeof(char) // 2 (Unicode)
int floatSize = sizeof(float) // 4
int doubleSize = sizeof(double) // 8
int decimalSize = sizeof(decimal) // 16
int boolSize = sizeof(bool) // 1
```

Section 52.57: in

The `in` keyword has three uses:

a) As part of the syntax in a `foreach` statement or as part of the syntax in a LINQ query

```
foreach (var member in sequence)
{
    // ...
}
```

b) 在泛型接口和泛型委托类型的上下文中，表示该类型参数的逆变：

```
public interface IComparer<in T>
{
    // ...
}
```

c) 在 LINQ 查询的上下文中，指被查询的集合

```
var query = from x in source select new { x.Name, x.ID, };
```

第 52.58 节 : implicit

关键字 `implicit` 用于重载转换运算符。例如，你可以声明一个 `Fraction` 类，当需要时它应自动转换为 `double` 类型，并且可以自动从

`int` 转换：

```
class Fraction(int numerator, int denominator)
{
    public int Numerator { get; } = numerator;
    public int Denominator { get; } = denominator;
    // ...
    public static implicit operator double(Fraction f)
    {
        return f.Numerator / (double) f.Denominator;
    }
    public static implicit operator Fraction(int i)
    {
        return new Fraction(i, 1);
    }
}
```

b) In the context of generic interfaces and generic delegate types signifies *contravariance* for the type parameter in question:

```
public interface IComparer<in T>
{
    // ...
}
```

c) In the context of LINQ query refers to the collection that is being queried

```
var query = from x in source select new { x.Name, x.ID, };
```

Section 52.58: implicit

The `implicit` keyword is used to overload a conversion operator. For example, you may declare a `Fraction` class that should automatically be converted to a `double` when needed, and that can be automatically converted from `int`:

```
class Fraction(int numerator, int denominator)
{
    public int Numerator { get; } = numerator;
    public int Denominator { get; } = denominator;
    // ...
    public static implicit operator double(Fraction f)
    {
        return f.Numerator / (double) f.Denominator;
    }
    public static implicit operator Fraction(int i)
    {
        return new Fraction(i, 1);
    }
}
```

第52.59节 : do

do 操作符会重复执行一段代码块，直到条件查询为假。do-while 循环也可以被 `goto`、`return`、`break` 或 `throw` 语句中断。

do 关键字的语法是：

```
do { 代码块; } while( 条件 );
```

示例：

```
int i = 0;

do
{
    Console.WriteLine("Do is on loop number {0}.", i);
} while (i++ < 5);
```

输出：

```
"Do is on loop number 1."
```

Section 52.59: do

The `do` operator iterates over a block of code until a conditional query equals false. The `do-while` loop can also be interrupted by a `goto`, `return`, `break` or `throw` statement.

The syntax for the `do` keyword is:

```
do { code block; } while( condition );
```

Example:

```
int i = 0;

do
{
    Console.WriteLine("Do is on loop number {0}.", i);
} while (i++ < 5);
```

Output:

```
"Do is on loop number 1."
```

```
"Do是第2次循环。"  
"Do是第3次循环。"  
"Do是第4次循环。"  
"Do是第5次循环。"
```

与 `while` 循环不同，`do-while` 循环是 **退出控制的**。这意味着即使条件第一次不满足，`do-while` 循环也会至少执行一次其语句。

```
bool a = false;  
  
do  
{  
    Console.WriteLine("即使 a 为 false, 这也只会打印一次。");  
} while (a == true);
```

第52.60节：long

`long` 关键字用于表示有符号的64位整数。它是 `System.Int64` 数据类型的别名，该数据类型存在于 `mscorlib.dll` 中，每个C#项目创建时都会隐式引用该dll。

任何 `long` 变量都可以显式或隐式声明：

```
long long1 = 9223372036854775806; // 显式声明, 使用 long 关键字  
var long2 = -9223372036854775806L; // 隐式声明, 使用 'L' 后缀
```

`long` 变量可以保存从 -9,223,372,036,854,775,808 到 9,223,372,036,854,775,807 之间的任意值，在变量必须保存超出其他变量（如 `int` 变量）范围的情况下非常有用。

第52.61节：enum

`enum` 关键字告诉编译器该类继承自抽象类 `Enum`，程序员无需显式继承。`Enum` 是 `ValueType` 的子类，用于表示一组独特的命名常量。

```
public enum DaysOfWeek  
{  
    Monday,  
    Tuesday,  
}
```

您可以选择为每个枚举值（或其中一些）指定一个具体的值：

```
public enum 重要年份  
{  
    第一次世界大战结束 = 1918;  
    第二次世界大战结束 = 1945,  
}
```

在此示例中，我省略了值为0的枚举项，这通常是不好的做法。枚举类型(`enum`)总会有一个默认值由显式转换(`YourEnumType`) 0产生，其中`YourEnumType`是您声明的枚举类型。若未定义值为0的枚举项，枚举在初始化时将没有定义的值。

枚举的默认底层类型是 `int`，您可以将底层类型更改为任何整型，包括 `byte`、

```
"Do is on loop number 2."  
"Do is on loop number 3."  
"Do is on loop number 4."  
"Do is on loop number 5."
```

Unlike the `while` loop, the `do-while` loop is **Exit Controlled**. This means that the `do-while` loop would execute its statements at least once, even if the condition fails the first time.

```
bool a = false;  
  
do  
{  
    Console.WriteLine("This will be printed once, even if a is false.");  
} while (a == true);
```

Section 52.60: long

The `long` keyword is used to represent signed 64-bit integers. It is an alias for the `System.Int64` datatype present in `mscorlib.dll`, which is implicitly referenced by every C# project when you create them.

Any `long` variable can be declared both explicitly and implicitly:

```
long long1 = 9223372036854775806; // explicit declaration, long keyword used  
var long2 = -9223372036854775806L; // implicit declaration, 'L' suffix used
```

A `long` variable can hold any value from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, and can be useful in situations which a variable must hold a value that exceeds the bounds of what other variables (such as the `int` variable) can hold.

Section 52.61: enum

The `enum` keyword tells the compiler that this class inherits from the abstract class `Enum`, without the programmer having to explicitly inherit it. `Enum` is a descendant of `ValueType`, which is intended for use with distinct set of named constants.

```
public enum DaysOfWeek  
{  
    Monday,  
    Tuesday,  
}
```

You can optionally specify a specific value for each one (or some of them):

```
public enum NotableYear  
{  
    EndOfWwI = 1918;  
    EndOfWwII = 1945,  
}
```

In this example I omitted a value for 0, this is usually a bad practice. An `enum` will always have a default value produced by explicit conversion (`YourEnumType`) 0, where `YourEnumType` is your declared enum type. Without a value of 0 defined, an `enum` will not have a defined value at initiation.

The default underlying type of `enum` is `int`, you can change the underlying type to any integral type including `byte`,

sbyte、short、ushort、int、uint、long和ulong。以下是一个底层类型为byte的枚举示例：

```
enum 星期 : byte
{
    星期日 = 0,
    星期一,
    星期二,
    星期三,
    星期四,
    星期五,
    星期六
};
```

还要注意，您可以通过强制转换轻松地在枚举类型和底层类型之间转换：

```
int value = (int)星期.第一次世界大战结束;
```

基于这些原因，当你调用库函数时，最好始终检查一个enum是否有效：

```
void PrintNotes(NotableYear year)
{
    if (!Enum.IsDefined(typeof(NotableYear), year))
        throw InvalidEnumArgumentException("year", (int)year, typeof(NotableYear));

    // ...
}
```

第52.62节：partial

关键字partial可以在类、结构体或接口的类型定义中使用，以允许类型定义被拆分到多个文件中。这对于在自动生成的代码中加入新功能非常有用。

File1.cs

```
namespace A
{
    public partial class Test
    {
        public string Var1 {get;set;}
    }
}
```

File2.cs

```
namespace A
{
    public partial class Test
    {
        public string Var2 {get;set;}
    }
}
```

注意：一个类可以拆分成任意数量的文件。但是，所有声明必须在相同的命名空间和相同的程序集下。

方法也可以使用partial关键字声明为部分方法。在这种情况下，一个文件只包含方法定义，另一个文件包含实现。

sbyte, short, ushort, int, uint, long and ulong. Below is an enum with underlying type byte:

```
enum Days : byte
{
    Sunday = 0,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};
```

Also note that you can convert to/from underlying type simply with a cast:

```
int value = (int)NotableYear.EndOfWwI;
```

For these reasons you'd better always check if an enum is valid when you're exposing library functions:

```
void PrintNotes(NotableYear year)
{
    if (!Enum.IsDefined(typeof(NotableYear), year))
        throw InvalidEnumArgumentException("year", (int)year, typeof(NotableYear));

    // ...
}
```

Section 52.62: partial

The keyword partial can be used during type definition of class, struct, or interface to allow the type definition to be split into several files. This is useful to incorporate new features in auto generated code.

File1.cs

```
namespace A
{
    public partial class Test
    {
        public string Var1 {get;set;}
    }
}
```

File2.cs

```
namespace A
{
    public partial class Test
    {
        public string Var2 {get;set;}
    }
}
```

Note: A class can be split into any number of files. However, all declaration must be under same namespace and the same assembly.

Methods can also be declared partial using the partial keyword. In this case one file will contain only the method definition and another file will contain the implementation.

部分方法在部分类型的一个部分中定义其签名，在类型的另一个部分中定义其实现。部分方法使类设计者能够提供方法钩子，类似于事件处理器，开发者可以选择是否实现。如果开发者没有提供实现，编译器将在编译时移除该签名。部分方法适用以下条件：

- 部分类型的两个部分中的签名必须匹配。
- 该方法必须返回void。
- 不允许访问修饰符。部分方法隐式为私有。

-- MSDN

File1.cs

```
namespace A
{
    public partial class Test
    {
        public string Var1 {get;set;}
        public partial Method1(string str);
    }
}
```

File2.cs

```
namespace A
{
    public partial class Test
    {
        public string Var2 {get;set;}
        public partial Method1(string str)
        {
            Console.WriteLine(str);
        }
    }
}
```

注意：包含部分方法的类型也必须声明为部分类型。

第52.63节：事件

事件 (event) 允许开发者实现通知模式。

简单示例

```
public class 服务器
{
    // 定义事件
    public event EventHandler 数据变更事件;

    void 触发事件()
    {
        var ev = 数据变更事件;
        if(ev != null)
        {
            ev(this, EventArgs.Empty);
        }
    }
}
```

A partial method has its signature defined in one part of a partial type, and its implementation defined in another part of the type. Partial methods enable class designers to provide method hooks, similar to event handlers, that developers may decide to implement or not. If the developer does not supply an implementation, the compiler removes the signature at compile time. The following conditions apply to partial methods:

- Signatures in both parts of the partial type must match.
- The method must return void.
- No access modifiers are allowed. Partial methods are implicitly private.

-- MSDN

File1.cs

```
namespace A
{
    public partial class Test
    {
        public string Var1 {get;set;}
        public partial Method1(string str);
    }
}
```

File2.cs

```
namespace A
{
    public partial class Test
    {
        public string Var2 {get;set;}
        public partial Method1(string str)
        {
            Console.WriteLine(str);
        }
    }
}
```

Note: The type containing the partial method must also be declared partial.

Section 52.63: event

An **event** allows the developer to implement a notification pattern.

simple example

```
public class Server
{
    // defines the event
    public event EventHandler DataChangeEvent;

    void RaiseEvent()
    {
        var ev = DataChangeEvent;
        if(ev != null)
        {
            ev(this, EventArgs.Empty);
        }
    }
}
```

```
}
```

```
}
```

```
public class 客户端
```

```
{
```

```
    public void 客户端(服务器 server)
```

```
{
```

```
        // 客户端订阅服务器的 DataChangeEvent 事件
```

```
server.DataChangeEvent += server_DataChanged;
```

```
}
```

```
    private void server_DataChanged(object sender, EventArgs args)
```

```
{
```

```
        // 当服务器触发 DataChangeEvent 事件时收到通知
```

```
}
```

```
}
```

[MSDN 参考](#)

第 52.64 节 : sbyte

一种用于存储 8 位有符号整数的数值类型。sbyte是System.SByte的别名，占用 1 字节内存。无符号对应类型使用byte。

有效范围是-127到127（剩余部分用于存储符号）。

```
sbyte a = 127; // 127
sbyte b = -127; // -127
sbyte c = 200; // 错误, 无法转换
sbyte d = unchecked((sbyte)129); // -127 (溢出)
```

```
}
```

```
}
```

```
public class Client
```

```
{
```

```
    public void Client(Server server)
```

```
{
```

```
        // client subscribes to the server's DataChangeEvent
```

```
        server.DataChangeEvent += server_DataChanged;
```

```
}
```

```
    private void server_DataChanged(object sender, EventArgs args)
```

```
{
```

```
        // notified when the server raises the DataChangeEvent
```

```
}
```

```
}
```

[MSDN reference](#)

Section 52.64: sbyte

A numeric type used to store 8-bit *signed* integers. sbyte is an alias for System.SByte, and takes up 1 byte of memory. For the unsigned equivalent, use byte.

Valid range is -127 to 127 (the leftover is used to store the sign).

```
sbyte a = 127; // 127
sbyte b = -127; // -127
sbyte c = 200; // Error, cannot be converted
sbyte d = unchecked((sbyte)129); // -127 (overflows)
```

第53章：面向对象编程 在C#中

本主题试图告诉我们如何基于面向对象编程（OOP）方法编写程序。但我们不打算教授面向对象编程范式。我们将涵盖以下主题：
类、属性、继承、多态、接口等。

第53.1节：类：

声明类的骨架是：

<>:必需

[]:可选

```
[private/public/protected/internal] class <期望的类名> [:[继承的类][,][[接口  
名称1],[接口名称2],...]  
{  
    //你的代码  
}
```

如果你不能理解整个语法也别担心，我们会熟悉所有部分。第一个例子考虑以下类：

```
class MyClass  
{  
    int i = 100;  
    public void getMyValue()  
    {  
        Console.WriteLine(this.i); //将在输出中打印数字100  
    }  
}
```

在这个类中，我们创建了一个变量 `i`，类型为 `int`，默认访问修饰符为私有，`getMyValue()`方法的访问修饰符为公共。

Chapter 53: Object Oriented Programming In C#

This topic try to tell us how we can write programs based on OOP approach. But we don't try to teach Object Oriented Programming paradigm. We'll be covering following topics:
Classes,Properties,Inheritance,Polymorphism,Interfaces and so on.

Section 53.1: Classes:

Skeleton of declaring class is:

<>:Required

[]:Optional

```
[private/public/protected/internal] class <Desired Class Name> [:[Inherited class][,][[Interface  
Name 1],[Interface Name 2],...]  
{  
    //Your code  
}
```

Don't worry if you can't understand whole syntax, We'll be get familiar with all part of that. for first example consider following class:

```
class MyClass  
{  
    int i = 100;  
    public void getMyValue()  
    {  
        Console.WriteLine(this.i); //Will print number 100 in output  
    }  
}
```

in this class we create variable `i` with `int` type and with default private [Access Modifiers](#) and `getMyValue()` method with public access modifiers.

第54章：递归

第54.1节：通俗易懂的递归

递归可以定义为：

一个方法调用自身，直到满足特定条件为止。

递归的一个优秀且简单的例子是计算给定数字的阶乘的方法：

```
public int Factorial(int number)
{
    return number == 0 ? 1 : n * Factorial(number - 1);
}
```

在此方法中，我们可以看到该方法将接受一个参数，number。

逐步说明：

以示例为例，执行阶乘(4)

1. 数字(4) 是否等于1？
2. 不等于？返回 $4 * \text{阶乘}(数字-1)(3)$
3. 因为该方法再次被调用，它现在使用阶乘(3)作为新的参数。
4. 这个过程会持续，直到执行阶乘(1)且数字(1) == 1返回1。
5. 总体上，计算过程“累积”了 $4 * 3 * 2 * 1$ ，最终返回24。

理解递归的关键在于该方法调用了自身的新实例。返回后，调用实例的执行继续进行。

第54.2节：斐波那契数列

你可以使用递归计算斐波那契数列中的某个数字。

根据数学理论 $F(n) = F(n-2) + F(n-1)$ ，对于任意 $i > 0$ ，

```
// 返回第 i 个斐波那契数
public int fib(int i) {
    if(i <= 2) {
        // 递归函数的基本情况。
        // i 是 1 或 2，对应的斐波那契数列数字都是 1。
        return 1;
    }
    // 递归情况。返回前两个斐波那契数的和。
    // 这是可行的，因为斐波那契数列的定义规定
    // 两个相邻元素的和等于下一个元素。
    return fib(i - 2) + fib(i - 1);
}

fib(10); // 返回55
```

Chapter 54: Recursion

Section 54.1: Recursion in plain English

Recursion can be defined as:

A method that calls itself until a specific condition is met.

An excellent and simple example of recursion is a method that will get the factorial of a given number:

```
public int Factorial(int number)
{
    return number == 0 ? 1 : n * Factorial(number - 1);
}
```

In this method, we can see that the method will take an argument, number.

Step by step:

Given the example, executing Factorial(4)

1. Is number (4) == 1?
2. No? return $4 * \text{Factorial}(number-1)$ (3)
3. Because the method is called once again, it now repeats the first step using Factorial(3) as the new argument.
4. This continues until Factorial(1) is executed and number (1) == 1 returns 1.
5. Overall, the calculation "builds up" $4 * 3 * 2 * 1$ and finally returns 24.

The key to understanding recursion is that the method calls a *new instance* of itself. After returning, the execution of the calling instance continues.

Section 54.2: Fibonacci Sequence

You can calculate a number in the Fibonacci sequence using recursion.

Following the math theory of $F(n) = F(n-2) + F(n-1)$, for any $i > 0$,

```
// Returns the i'th Fibonacci number
public int fib(int i) {
    if(i <= 2) {
        // Base case of the recursive function.
        // i is either 1 or 2, whose associated Fibonacci sequence numbers are 1 and 1.
        return 1;
    }
    // Recursive case. Return the sum of the two previous Fibonacci numbers.
    // This works because the definition of the Fibonacci sequence specifies
    // that the sum of two adjacent elements equals the next element.
    return fib(i - 2) + fib(i - 1);
}

fib(10); // Returns 55
```

第54.3节：幂的计算

计算给定数字的幂也可以递归完成。给定底数 n 和指数 e，我们需要确保通过减少指数 e 将问题分解成若干部分。

理论示例：

- $2^2 = 2 \times 2$
- $2^3 = 2 \times 2 \times 2$ 或者, $2^3 = 2^2 \times 2$

这就是我们递归算法的秘密所在（见下面的代码）。这就是将问题拆分成更小、更易解决的部分。

• 注意事项

- 当底数为0时，我们必须注意返回0，因为 $0^3 = 0 \times 0 \times 0$
- 当指数为0时，我们必须注意始终返回1，因为这是数学规则。

代码示例：

```
public int CalcPowerOf(int b, int e) {  
    if (b == 0) { return 0; } // 当底数为0时，无论如何，结果总是返回0  
    if (e == 0) { return 1; } // 数学规则，指数为0时结果总是返回1  
    return b * CalcPowerOf(b, e - 1); // 实际的递归逻辑，我们将问题拆分，即： $2^3 = 2 \times 2^2$  等等..  
}
```

在xUnit中进行测试以验证逻辑：

虽然这不是必须的，但编写测试来验证逻辑总是好的。我在这里包含了使用[xUnit框架](#)编写的测试。

```
[Theory]  
[MemberData(nameof(PowerOfTestData))]  
public void PowerOfTest(int @base, int exponent, int expected) {  
    Assert.Equal(expected, CalcPowerOf(@base, exponent));  
}  
  
public static IEnumerable<object[]> PowerOfTestData() {  
    yield return new object[] { 0, 0, 0 };  
    yield return new object[] { 0, 1, 0 };  
    yield return new object[] { 2, 0, 1 };  
    yield return new object[] { 2, 1, 2 };  
    yield return new object[] { 2, 2, 4 };  
    yield return new object[] { 5, 2, 25 };  
    yield return new object[] { 5, 3, 125 };  
    yield return new object[] { 5, 4, 625 };  
}
```

第54.4节：递归描述对象结构

递归是指一个方法调用自身。理想情况下，它会一直调用直到满足特定条件，然后正常退出方法，返回到调用该方法的地方。如果不这样，可能会因为递归调用过多而导致堆栈溢出异常。

```
/// <summary>  
/// 创建一个对象结构，代码可以递归描述  
///  
public class Root  
{
```

Section 54.3: PowerOf calculation

Calculating the power of a given number can be done recursively as well. Given a base number n and exponent e, we need to make sure to split the problem in chunks by decreasing the exponent e.

Theoretical Example:

- $2^2 = 2 \times 2$
- $2^3 = 2 \times 2 \times 2$ or, $2^3 = 2^2 \times 2$

In there lies the secret of our recursive algorithm (see the code below). This is about taking the problem and separating it into smaller and simpler to solve chunks.

• Notes

- when the base number is 0, we have to be aware to return 0 as $0^3 = 0 \times 0 \times 0$
- when the exponent is 0, we have to be aware to always return 1, as this is a mathematical rule.

Code Example:

```
public int CalcPowerOf(int b, int e) {  
    if (b == 0) { return 0; } // when base is 0, it doesn't matter, it will always return 0  
    if (e == 0) { return 1; } // math rule, exponent 0 always returns 1  
    return b * CalcPowerOf(b, e - 1); // actual recursive logic, where we split the problem, aka:  $2^3 = 2 \times 2^2$  etc..  
}
```

Tests in xUnit to verify the logic:

Although this is not necessary, it's always good to write tests to verify your logic. I include those here written in the [xUnit framework](#).

```
[Theory]  
[MemberData(nameof(PowerOfTestData))]  
public void PowerOfTest(int @base, int exponent, int expected) {  
    Assert.Equal(expected, CalcPowerOf(@base, exponent));  
}  
  
public static IEnumerable<object[]> PowerOfTestData() {  
    yield return new object[] { 0, 0, 0 };  
    yield return new object[] { 0, 1, 0 };  
    yield return new object[] { 2, 0, 1 };  
    yield return new object[] { 2, 1, 2 };  
    yield return new object[] { 2, 2, 4 };  
    yield return new object[] { 5, 2, 25 };  
    yield return new object[] { 5, 3, 125 };  
    yield return new object[] { 5, 4, 625 };  
}
```

Section 54.4: Recursively describe an object structure

Recursion is when a method calls itself. Preferably it will do so until a specific condition is met and then it will exit the method normally, returning to the point from which the method was called. If not, a stack overflow exception might occur due to too many recursive calls.

```
/// <summary>  
/// Create an object structure the code can recursively describe  
/// </summary>  
public class Root  
{
```

```

public string Name { get; set; }
public ChildOne Child { get; set; }
}
public class ChildOne
{
    public string ChildOneName { get; set; }
    public ChildTwo Child { get; set; }
}
public class ChildTwo
{
    public string ChildTwoName { get; set; }
}
/// <summary>
/// 带有递归函数 DescribeTypeOfObject 的控制台应用程序
///
public class Program
{
    static void Main(string[] args)
    {
        // 点 A, 我们用类型 'Root' 调用该函数
DescribeTypeOfObject(typeof(Root));
        Console.WriteLine("按任意键退出");
        Console.ReadKey();
    }

    static void DescribeTypeOfObject(Type type)
    {
        // 获取此类型的所有属性
Console.WriteLine($"描述类型 {type.Name}");
        PropertyInfo[] propertyInfos = type.GetProperties();
        foreach (PropertyInfo pi in propertyInfos)
        {
            Console.WriteLine($"具有属性 {pi.Name}, 类型为 {pi.PropertyType.Name}");
            // 是否为自定义类类型? 也描述它
            if (pi.PropertyType.IsClass && !pi.PropertyType.FullName.StartsWith("System."))
            {
                // 点B, 我们调用该属性的类型函数
DescribeTypeOfObject(pi.PropertyType);
            }
        }
        // 所有属性处理完毕
        // 返回到调用点
        // 第一次调用的点A
        // 自定义类类型所有属性的点B
    }
}

```

第54.5节：使用递归获取目录树

递归的一个用途是遍历层级数据结构，比如文件系统目录树，而无需知道树有多少层或每层有多少对象。在这个例子中，您将看到如何在目录树上使用递归来查找指定目录的所有子目录，并将整个树打印到控制台。

```

internal class Program
{
    internal const int RootLevel = 0;
    internal const char Tab = '\t';

    internal static void Main()

```

```

public string Name { get; set; }
public ChildOne Child { get; set; }
}
public class ChildOne
{
    public string ChildOneName { get; set; }
    public ChildTwo Child { get; set; }
}
public class ChildTwo
{
    public string ChildTwoName { get; set; }
}
/// <summary>
/// The console application with the recursive function DescribeTypeOfObject
/// </summary>
public class Program
{
    static void Main(string[] args)
    {
        // point A, we call the function with type 'Root'
DescribeTypeOfObject(typeof(Root));
        Console.WriteLine("Press a key to exit");
        Console.ReadKey();
    }

    static void DescribeTypeOfObject(Type type)
    {
        // get all properties of this type
Console.WriteLine($"Describing type {type.Name}");
        PropertyInfo[] propertyInfos = type.GetProperties();
        foreach (PropertyInfo pi in propertyInfos)
        {
            Console.WriteLine($"Has property {pi.Name} of type {pi.PropertyType.Name}");
            // is a custom class type? describe it too
            if (pi.PropertyType.IsClass && !pi.PropertyType.FullName.StartsWith("System."))
            {
                // point B, we call the function type this property
                DescribeTypeOfObject(pi.PropertyType);
            }
        }
        // done with all properties
        // we return to the point where we were called
        // point A for the first call
        // point B for all properties of type custom class
    }
}

```

Section 54.5: Using Recursion to Get Directory Tree

One of the uses of recursion is to navigate through a hierarchical data structure, like a file system directory tree, without knowing how many levels the tree has or the number of objects on each level. In this example, you will see how to use recursion on a directory tree to find all sub-directories of a specified directory and print the whole tree to the console.

```

internal class Program
{
    internal const int RootLevel = 0;
    internal const char Tab = '\t';

    internal static void Main()

```

```

{
Console.WriteLine("请输入根目录的路径:");
    var rootDirectorypath = Console.ReadLine();

Console.WriteLine(
$"正在获取目录树 '{rootDirectorypath}'");

PrintDirectoryTree(rootDirectorypath);
Console.WriteLine("按 'Enter' 键退出...");
    Console.ReadLine();
}

internal static void PrintDirectoryTree(string rootDirectoryPath)
{
    try
    {
        if (!Directory.Exists(rootDirectoryPath))
        {
            throw new DirectoryNotFoundException(
                $"未找到目录 '{rootDirectoryPath}'。");
        }

        var rootDirectory = new DirectoryInfo(rootDirectoryPath);
        PrintDirectoryTree(rootDirectory, RootLevel);
    }
    catch (DirectoryNotFoundException e)
    {
        Console.WriteLine(e.Message);
    }
}

private static void PrintDirectoryTree(
    DirectoryInfo directory, int currentLevel)
{
    var indentation = string.Empty;
    for (var i = RootLevel; i < currentLevel; i++)
    {
        indentation += Tab;
    }

Console.WriteLine($"{indentation}-{directory.Name}");
    var nextLevel = currentLevel + 1;
    try
    {
        foreach (var subDirectory in directory.GetDirectories())
        {
PrintDirectoryTree(subDirectory, nextLevel);
        }
    }
    catch (UnauthorizedAccessException e)
    {
        Console.WriteLine($"{indentation}-{e.Message}");
    }
}
}

```

这段代码比完成此任务所需的最基本代码稍微复杂一些，因为它包含异常检查来处理获取目录时的任何问题。下面您将看到代码被分解成更小的部分，并附有每部分的解释。

主要：

```

{
Console.WriteLine("Enter the path of the root directory:");
    var rootDirectorypath = Console.ReadLine();

Console.WriteLine(
    $"Getting directory tree of '{rootDirectorypath}'");

PrintDirectoryTree(rootDirectorypath);
Console.WriteLine("Press 'Enter' to quit...");
    Console.ReadLine();
}

internal static void PrintDirectoryTree(string rootDirectoryPath)
{
    try
    {
        if (!Directory.Exists(rootDirectoryPath))
        {
            throw new DirectoryNotFoundException(
                $"Directory '{rootDirectoryPath}' not found.");
        }

        var rootDirectory = new DirectoryInfo(rootDirectoryPath);
        PrintDirectoryTree(rootDirectory, RootLevel);
    }
    catch (DirectoryNotFoundException e)
    {
        Console.WriteLine(e.Message);
    }
}

private static void PrintDirectoryTree(
    DirectoryInfo directory, int currentLevel)
{
    var indentation = string.Empty;
    for (var i = RootLevel; i < currentLevel; i++)
    {
        indentation += Tab;
    }

Console.WriteLine($"{indentation}-{directory.Name}");
    var nextLevel = currentLevel + 1;
    try
    {
        foreach (var subDirectory in directory.GetDirectories())
        {
PrintDirectoryTree(subDirectory, nextLevel);
        }
    }
    catch (UnauthorizedAccessException e)
    {
        Console.WriteLine($"{indentation}-{e.Message}");
    }
}
}

```

This code is somewhat more complicated than the bare minimum to complete this task, as it includes exception checking to handle any issues with getting the directories. Below you will find a break-down of the code into smaller segments with explanations of each.

Main:

主方法接收用户输入的字符串，该字符串用作根目录的路径。然后它调用PrintDirectoryTree方法，并将该字符串作为参数传入。

PrintDirectoryTree(string):

这是处理实际目录树打印的两种方法中的第一种。该方法以表示根目录路径的字符串作为参数。它会检查该路径是否为实际目录，如果不是，则抛出一个DirectoryNotFoundException异常，该异常随后在catch块中处理。如果路径是一个真实的目录，从路径创建了 DirectoryInfo对象rootDirectory，随后调用第二个PrintDirectoryTree方法，传入rootDirectory对象和RootLevel，后者是一个值为零的整数常量。

PrintDirectoryTree(DirectoryInfo, int):

第二种方法处理了大部分工作。它以一个DirectoryInfo和一个整数作为参数。该 DirectoryInfo是当前目录，整数表示该目录相对于根目录的深度。为了便于阅读，输出会根据当前目录的深度进行缩进，使输出看起来像这样：

```
- 根目录
  - 子节点 1
  - 子目录 2
    - 子目录 2.1
  - 子目录 3
```

当前目录打印完成后，会获取其子目录，然后对每个子目录调用此方法，传入的深度值比当前深度大一。这部分就是递归：方法调用自身。程序将以这种方式运行，直到访问完目录树中的每个目录。当遇到没有子目录的目录时，方法会自动返回。

此方法还捕获了UnauthorizedAccessException异常，如果当前目录的任何子目录受系统保护而无法访问，则会抛出该异常。错误信息会以当前缩进级别打印，以保持一致性。

下面的方法提供了一个更基础的解决方案：

```
internal static void PrintDirectoryTree(string directoryName)
{
    try
    {
        if (!Directory.Exists(directoryName)) return;
        Console.WriteLine(directoryName);
        foreach (var d in Directory.GetDirectories(directoryName))
        {
            PrintDirectoryTree(d);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

这不包括第一种方法中特定的错误检查或输出格式，但它实际上完成了相同的功能。由于它只使用字符串而不是DirectoryInfo，因此无法访问诸如权限等其他目录属性。

The main method takes an input from a user as a string, which is to be used as the path to the root directory. It then calls the PrintDirectoryTree method with this string as the parameter.

PrintDirectoryTree(string):

This is the first of two methods that handle the actual directory tree printing. This method takes a string representing the path to the root directory as a parameter. It checks if the path is an actual directory, and if not, throws a DirectoryNotFoundException which is then handled in the catch block. If the path is a real directory, a DirectoryInfo object rootDirectory is created from the path, and the second PrintDirectoryTree method is called with the rootDirectory object and RootLevel, which is an integer constant with a value of zero.

PrintDirectoryTree(DirectoryInfo, int):

This second method handles the brunt of the work. It takes a DirectoryInfo and an integer as parameters. The DirectoryInfo is the current directory, and the integer is the depth of the directory relative to the root. For ease of reading, the output is indented for each level deep the current directory is, so that the output looks like this:

```
-Root
  -Child 1
  -Child 2
    -Grandchild 2.1
  -Child 3
```

Once the current directory is printed, its sub directories are retrieved, and this method is then called on each of them with a depth level value of one more than the current. That part is the recursion: the method calling itself. The program will run in this manner until it has visited every directory in the tree. When it reached a directory with no sub directories, the method will return automatically.

This method also catches an UnauthorizedAccessException, which is thrown if any of the sub directories of the current directory are protected by the system. The error message is printed at the current indentation level for consistency.

The method below provides a more basic approach to this problem:

```
internal static void PrintDirectoryTree(string directoryName)
{
    try
    {
        if (!Directory.Exists(directoryName)) return;
        Console.WriteLine(directoryName);
        foreach (var d in Directory.GetDirectories(directoryName))
        {
            PrintDirectoryTree(d);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

This does not include the specific error checking or output formatting of the first approach, but it effectively does the same thing. Since it only uses strings as opposed to DirectoryInfo, it cannot provide access to other directory properties like permissions.

第54.6节：阶乘计算

一个数的阶乘（用“！”表示，例如 $9!$ ）是该数与比它小一的数的阶乘相乘。因此，例如， $9! = 9 \times 8! = 9 \times 8 \times 7! = 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$ 。

所以在代码中，这用递归实现如下：

```
long Factorial(long x)
{
    if (x < 1)
    {
        throw new OutOfRangeException("Factorial can only be used with positive numbers.");
    }

    if (x == 1)
    {
        return 1;
    } else {
        return x * Factorial(x - 1);
    }
}
```

Section 54.6: Factorial calculation

The factorial of a number (denoted with $!$, as for instance $9!$) is the multiplication of that number with the factorial of one lower. So, for instance, $9! = 9 \times 8! = 9 \times 8 \times 7! = 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$.

So in code that becomes, using recursion:

```
long Factorial(long x)
{
    if (x < 1)
    {
        throw new OutOfRangeException("Factorial can only be used with positive numbers.");
    }

    if (x == 1)
    {
        return 1;
    } else {
        return x * Factorial(x - 1);
    }
}
```

第55章：命名约定

本节介绍在编写C#语言时使用的一些基本命名约定。像所有约定一样，它们不会被编译器强制执行，但能确保开发者之间的可读性。

有关全面的 .NET 框架设计指南，请参见 docs.microsoft.com/dotnet/standard/design-guidelines。

第 55.1 节：大小写约定

以下术语描述了标识符的不同大小写方式。

帕斯卡命名法 (Pascal Casing)

标识符的第一个字母以及每个后续连接单词的首字母均大写。您可以对三个或更多字符的标识符使用帕斯卡命名法。例如：BackColor

驼峰命名法 (Camel Casing)

标识符的第一个字母小写，每个后续连接单词的首字母大写。
例如：backColor

全部大写

标识符中的所有字母均大写。例如：IO

规则

当标识符由多个单词组成时，不要在单词之间使用分隔符，如下划线（"_"）或连字符（"-"）。应使用大小写来表示每个单词的开头。

下表总结了标识符的大小写规则，并提供了不同类型标识符的示例：

标识符	大小写	示例
局部变量	驼峰式	carName
类	帕斯卡式	AppDomain
枚举类型	帕斯卡式	ErrorLevel
枚举值	帕斯卡式	FatalError
Event	Pascal	值更改
异常类	Pascal	网络异常
只读静态字段	Pascal	RedValue
接口	Pascal	可释放接口 (IDisposable)
方法	Pascal	转字符串 (ToString)
命名空间	Pascal	系统绘图 (System.Drawing)
参数	驼峰命名类型名	
属性	Pascal	背景色

更多信息可在MSDN上找到。

Chapter 55: Naming Conventions

This topic outlines some basic naming conventions used when writing in the C# language. Like all conventions, they are not enforced by the compiler, but will ensure readability between developers.

For comprehensive .NET framework design guidelines, see docs.microsoft.com/dotnet/standard/design-guidelines.

Section 55.1: Capitalization conventions

The following terms describe different ways to case identifiers.

Pascal Casing

The first letter in the identifier and the first letter of each subsequent concatenated word are capitalized. You can use Pascal case for identifiers of three or more characters. For example: BackColor

Camel Casing

The first letter of an identifier is lowercase and the first letter of each subsequent concatenated word is capitalized. For example: backColor

Uppercase

All letters in the identifier are capitalized. For example: IO

Rules

When an identifier consists of multiple words, do not use separators, such as underscores ("_") or hyphens ("-"), between words. Instead, use casing to indicate the beginning of each word.

The following table summarizes the capitalization rules for identifiers and provides examples for the different types of identifiers:

Identifier	Case	Example
Local variable	Camel	carName
Class	Pascal	AppDomain
Enumeration type	Pascal	ErrorLevel
Enumeration values	Pascal	FatalError
Event	Pascal	ValueChanged
Exception class	Pascal	WebException
Read-only static field	Pascal	RedValue
Interface	Pascal	IDisposable
Method	Pascal	ToString
Namespace	Pascal	System.Drawing
Parameter	Camel	typeName
Property	Pascal	BackColor

More information can be found on [MSDN](#).

第55.2节：枚举

大多数枚举使用单数名称

```
public enum Volume
{
    低,
    中,
    高
}
```

位字段的枚举类型使用复数名称

```
[Flags]
public enum MyColors
{
    黄色 = 1,
    绿色 = 2,
    红色 = 4,
    蓝色 = 8
}
```

注意：始终为位字段枚举类型添加`FlagsAttribute`。

不要在后缀中添加'enum'

```
public enum VolumeEnum // 错误示例
```

不要在每个条目中使用枚举名

```
public enum Color
{
    ColorBlue, // 去掉Color, 没必要
    ColorGreen,
}
```

第55.3节：接口

接口名称应使用名词或名词短语，或描述行为的形容词。例如

`IComponent` 使用描述性名词，`ICustomAttributeProvider` 使用名词短语，`IPersistable` 使用形容词。

接口名称应以字母I为前缀，以表示该类型是接口，并且应使用帕斯卡命名法。

以下是命名正确的接口：

```
公共接口 IServiceProvider
公共接口 IFormattable
```

第55.4节：异常

添加“exception”作为后缀

自定义异常名称应以“-Exception”结尾。

下面是命名正确的异常：

Section 55.2: Enums

Use a singular name for most Enums

```
public enum Volume
{
    Low,
    Medium,
    High
}
```

Use a plural name for Enum types that are bit fields

```
[Flags]
public enum MyColors
{
    Yellow = 1,
    Green = 2,
    Red = 4,
    Blue = 8
}
```

Note: Always add the `FlagsAttribute` to a bit field Enum type.

Do not add 'enum' as a suffix

```
public enum VolumeEnum // Incorrect
```

Do not use the enum name in each entry

```
public enum Color
{
    ColorBlue, // Remove Color, unnecessary
    ColorGreen,
}
```

Section 55.3: Interfaces

Interfaces should be named with nouns or noun phrases, or adjectives that describe behaviour. For example `IComponent` uses a descriptive noun, `ICustomAttributeProvider` uses a noun phrase and `IPersistable` uses an adjective.

Interface names should be prefixed with the letter I, to indicate that the type is an interface, and Pascal case should be used.

Below are correctly named interfaces:

```
public interface IServiceProvider
public interface IFormattable
```

Section 55.4: Exceptions

Add 'exception' as a suffix

Custom exception names should be suffixed with "-Exception".

Below are correctly named exceptions:

```
public class MyCustomException : Exception  
public class FooException : Exception
```

第55.5节：私有字段

私有字段有两种常见的命名约定：camelCase和_camelCaseWithLeadingUnderscore。

驼峰式命名

```
public class Rational  
{  
    private readonly int numerator;  
    private readonly int denominator;  
  
    public Rational(int numerator, int denominator)  
    {  
        // "this" 关键字用于引用类作用域内的字段  
        this.分子 = 分子;  
        this.分母 = 分母;  
    }  
}
```

带下划线的驼峰命名法

```
public class Rational  
{  
    private readonly int _分子;  
    private readonly int _分母;  
  
    public 有理数(int 分子, int 分母)  
    {  
        // 名称唯一, 因此不需要"this"关键字  
        _分子 = 分子;  
        _分母 = 分母;  
    }  
}
```

```
public class MyCustomException : Exception  
public class FooException : Exception
```

Section 55.5: Private fields

There are two common conventions for private fields: camelCase and _camelCaseWithLeadingUnderscore.

Camel case

```
public class Rational  
{  
    private readonly int numerator;  
    private readonly int denominator;  
  
    public Rational(int numerator, int denominator)  
    {  
        // "this" keyword is required to refer to the class-scope field  
        this.numerator = numerator;  
        this.denominator = denominator;  
    }  
}
```

Camel case with underscore

```
public class Rational  
{  
    private readonly int _numerator;  
    private readonly int _denominator;  
  
    public Rational(int numerator, int denominator)  
    {  
        // Names are unique, so "this" keyword is not required  
        _numerator = numerator;  
        _denominator = denominator;  
    }  
}
```

第55.6节：命名空间

命名空间的一般格式为：

```
<公司>.(<产品>|<技术>)[.<特性>][.<子命名空间>].
```

示例包括：

```
Fabrikam.数学  
Litware.安全
```

在命名空间名前加上公司名称可以防止不同公司的命名空间名称相同。

Section 55.6: Namespaces

The general format for namespaces is:

```
<Company> . (<Product>|<Technology>) [ .<Feature> ] [ .<Subnamespace> ].
```

Examples include:

```
Fabrikam.Math  
Litware.Security
```

Prefixing namespace names with a company name prevents namespaces from different companies from having the same name.

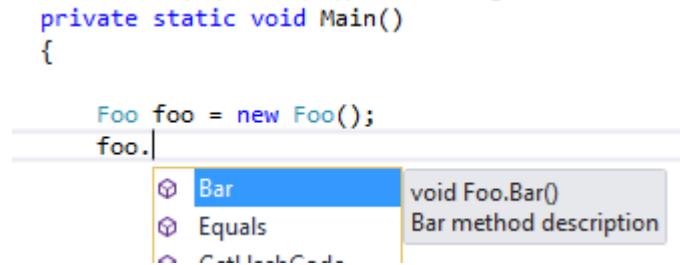
第56章：XML文档注释

第56.1节：简单的方法注释

文档注释直接放置在它们描述的方法或类的上方。它们以三个斜杠///开头，允许通过XML存储元信息。

```
/// <summary>
/// Bar方法描述
/// </summary>
public void Bar()
{}
```

标签内的信息可以被Visual Studio和其他工具使用，以提供如IntelliSense等服务：

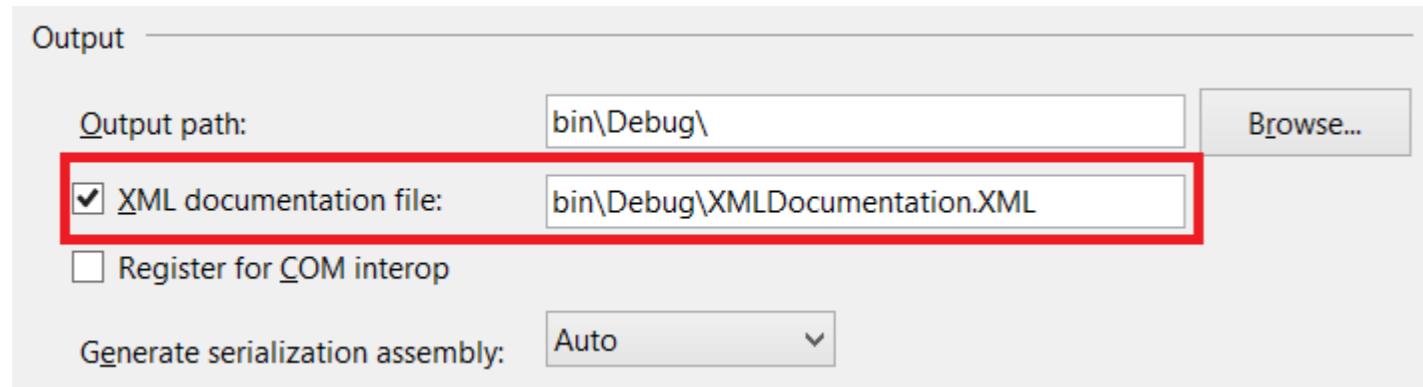


另见微软的常用文档标签列表。

第56.2节：从文档注释生成XML

要从代码中的文档注释生成XML文档文件，请使用/doc选项配合
csc.exe C#编译器。

在 Visual Studio 2013/2015 中，进入项目 -> 属性 -> 生成 -> 输出，勾选XML 文档文件复选框：



当你构建项目时，编译器会生成一个与项目名称对应的 XML 文件（例如XMLDocumentation.dll -> XMLDocumentation.xml）。

当你在另一个项目中使用该程序集时，确保 XML 文件与被引用的 DLL 位于同一目录下。

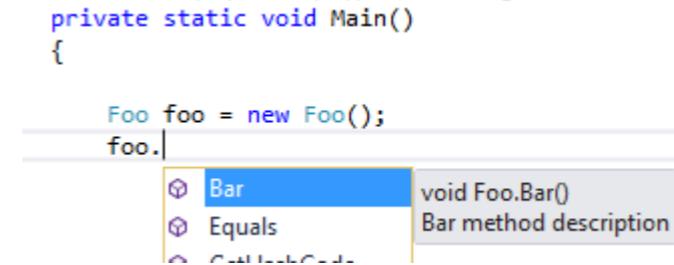
Chapter 56: XML Documentation Comments

Section 56.1: Simple method annotation

Documentation comments are placed directly above the method or class they describe. They begin with three forward slashes `///`, and allow meta information to be stored via XML.

```
/// <summary>
/// Bar method description
/// </summary>
public void Bar()
{}
```

Information inside the tags can be used by Visual Studio and other tools to provide services such as IntelliSense:

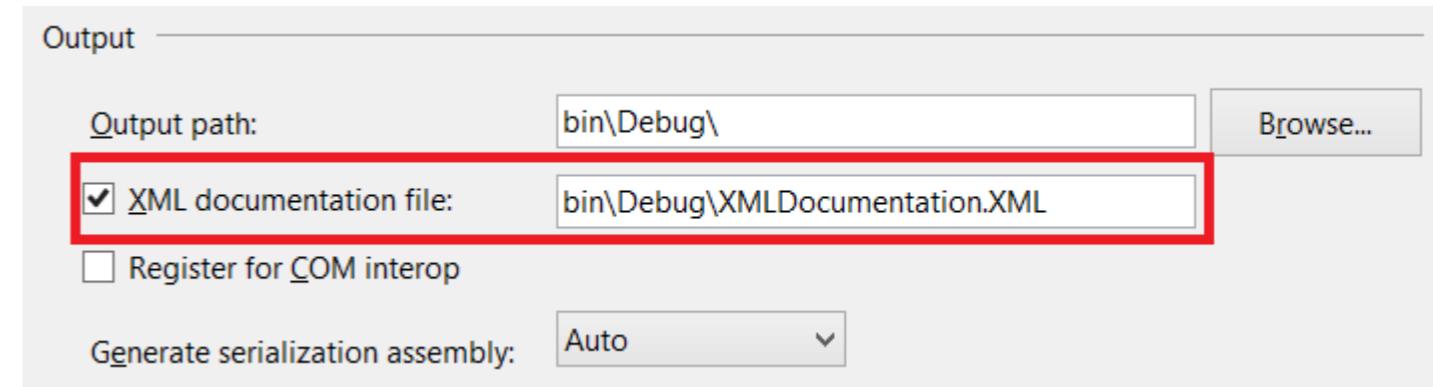


See also [Microsoft's list of common documentation tags](#).

Section 56.2: Generating XML from documentation comments

To generate an XML documentation file from documentation comments in the code, use the /doc option with the csc.exe C# compiler.

In Visual Studio 2013/2015, In **Project -> Properties -> Build -> Output**, check the XML documentation file checkbox:



When you build the project, an XML file will be produced by the compiler with a name corresponding to the project name (e.g. XMLDocumentation.dll -> XMLDocumentation.xml).

When you use the assembly in another project, make sure that the XML file is in the same directory as the DLL being referenced.

此示例：

```
/// <summary>
/// 数据类描述
/// </summary>
public class DataClass
{
    /// <summary>
    /// 名称属性描述
    /// </summary>
    public string Name { get; set; }
}

/// <summary>
/// Foo 函数
/// </summary>
public class Foo
{
    /// <summary>
    /// 此方法返回一些数据
    /// </summary>
    /// <param name="id">Id 参数</param>
    /// <param name="time">时间参数</param>
    /// <returns>将返回数据</returns>
    public DataClass GetData(int id, DateTime time)
    {
        return new DataClass();
    }
}
```

生成此构建时的 XML：

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>XMLDocumentation</name>
    </assembly>
    <members>
        <member name="T:XMLDocumentation.DataClass">
            <summary>
                数据类描述
            </summary>
            </member>
            <member name="P:XMLDocumentation.DataClass.Name">
                <summary>
                    名称属性描述
                </summary>
                </member>
                <member name="T:XMLDocumentation.Foo">
                    <summary>
                        Foo 函数
                    </summary>
                    </member>
                    <member name="M:XMLDocumentation.Foo.GetData(System.Int32, System.DateTime)">
                        <summary>
                            此方法返回一些数据
                        </summary>
                        <param name="id">Id 参数</param>
                        <param name="time">时间参数</param>
                        <returns>将返回数据</returns>

```

This example:

```
/// <summary>
/// Data class description
/// </summary>
public class DataClass
{
    /// <summary>
    /// Name property description
    /// </summary>
    public string Name { get; set; }
}

/// <summary>
/// Foo function
/// </summary>
public class Foo
{
    /// <summary>
    /// This method returning some data
    /// </summary>
    /// <param name="id">Id parameter</param>
    /// <param name="time">Time parameter</param>
    /// <returns>Data will be returned</returns>
    public DataClass GetData(int id, DateTime time)
    {
        return new DataClass();
    }
}
```

Produces this xml on build:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>XMLDocumentation</name>
    </assembly>
    <members>
        <member name="T:XMLDocumentation.DataClass">
            <summary>
                Data class description
            </summary>
            </member>
            <member name="P:XMLDocumentation.DataClass.Name">
                <summary>
                    Name property description
                </summary>
                </member>
                <member name="T:XMLDocumentation.Foo">
                    <summary>
                        Foo function
                    </summary>
                    </member>
                    <member name="M:XMLDocumentation.Foo.GetData(System.Int32, System.DateTime)">
                        <summary>
                            This method returning some data
                        </summary>
                        <param name="id">Id parameter</param>
                        <param name="time">Time parameter</param>
                        <returns>Data will be returned</returns>

```

```
</member>
</members>
</doc>
```

第56.3节：带有param和returns元素的方法文档注释

```
/// <summary>
/// 返回指定ID和时间戳的数据。
/// </summary>
/// <param name="id">要获取数据的ID。</param>
/// <param name="time">要获取数据的日期时间。</param>
/// <returns>包含结果的DataClass实例。</returns>
public DataClass GetData(int id, DateTime time)
{
    // ...
}
```

IntelliSense 显示每个参数的描述：

```
obj.GetData(3, DateTime.Now);
DataClass Foo.GetData(int id, DateTime time)
This method returning some data
id: Id parameter
```

提示：如果Visual Studio中Intellisense未显示，删除第一个括号或逗号后重新输入。

第56.4节：接口和类的文档注释

```
/// <summary>
/// 此接口可以执行Foo
/// </summary>
public interface ICanDoFoo
{
    // ...
}

/// <summary>
/// 此Bar类实现了ICanDoFoo接口
/// </summary>
public class Bar : ICanDoFoo
{
    // ...
}
```

结果

接口摘要

```
ICanDoFoo bar = new Bar();
} ← ICanDoFoo interface ConsoleApplication1.ICanDoFoo
This interface can do Foo
```

类摘要

```
</member>
</members>
</doc>
```

Section 56.3: Method documentation comment with param and returns elements

```
/// <summary>
/// Returns the data for the specified ID and timestamp.
/// </summary>
/// <param name="id">The ID for which to get data. </param>
/// <param name="time">The DateTime for which to get data. </param>
/// <returns>A DataClass instance with the result. </returns>
public DataClass GetData(int id, DateTime time)
{
    // ...
}
```

IntelliSense shows you the description for each parameter:

```
obj.GetData(3, DateTime.Now);
DataClass Foo.GetData(int id, DateTime time)
This method returning some data
id: Id parameter
```

Tip: If Intellisense doesn't display in Visual Studio, delete the first bracket or comma and then type it again.

Section 56.4: Interface and class documentation comments

```
/// <summary>
/// This interface can do Foo
/// </summary>
public interface ICanDoFoo
{
    // ...
}

/// <summary>
/// This Bar class implements ICanDoFoo interface
/// </summary>
public class Bar : ICanDoFoo
{
    // ...
}
```

Result

Interface summary

```
ICanDoFoo bar = new Bar();
} ← ICanDoFoo interface ConsoleApplication1.ICanDoFoo
This interface can do Foo
```

Class summary

```
ICanDoFoo bar = new Bar();
}
} class ConsoleApplication1.Bar
This is a Bar class implements ICanDoFoo interface
```

第56.5节：文档中引用另一个类

<see> 标签可用于链接到另一个类。它包含 cref 成员，应该包含要引用的类的名称。Visual Studio 在编写此标签时会提供智能感知 (Intellisense) ，并且在重命名被引用的类时也会处理这些引用。

```
/// <summary>
/// 你可能还想查看 <see cref="SomeOtherClass"/>。
/// </summary>
public class SomeClass
{ }
```

在 Visual Studio 的智能感知弹出窗口中，这些引用也会以彩色显示在文本中。

要引用泛型类，请使用类似以下的写法：

```
/// <summary>
/// <see cref="List{T}"> 的增强版本。
/// </summary>
public class SomeGenericClass<T>
{ }
```

```
ICanDoFoo bar = new Bar();
}
} class ConsoleApplication1.Bar
This is a Bar class implements ICanDoFoo interface
```

Section 56.5: Referencing another class in documentation

The <see> tag can be used to link to another class. It contains the cref member which should contain the name of the class that is to be referenced. Visual Studio will provide Intellisense when writing this tag and such references will be processed when renaming the referenced class, too.

```
/// <summary>
/// You might also want to check out <see cref="SomeOtherClass"/>.
/// </summary>
public class SomeClass
{ }
```

In Visual Studio Intellisense popups such references will also be displayed colored in the text.

To reference a generic class, use something similar to the following:

```
/// <summary>
/// An enhanced version of <see cref="List{T}">.
/// </summary>
public class SomeGenericClass<T>
{ }
```

第57章：注释和区域

第57.1节：注释

在项目中使用注释是一种方便的方式，用于解释你的设计选择，目的是在维护或添加代码时让你（或其他人）的工作更容易。

有两种方法可以向代码中添加注释。

单行注释

放在//后面的任何文本都会被视为注释。

```
public class Program
{
    // 这是我的程序入口点。
    public static void Main()
    {
        // 向控制台打印一条消息。 - 这是一个注释！
        System.Console.WriteLine("Hello, World!");

        // System.Console.WriteLine("Hello, World again!"); // 你甚至可以注释掉代码。
        System.Console.ReadLine();
    }
}
```

多行或分隔注释

位于/*和*/之间的任何文本都会被视为注释。

```
public class Program
{
    public static void Main()
    {
        /*
        这是一个多行注释
        编译器会忽略它。
        */
        System.Console.WriteLine("Hello, World!");

        // 也可以使用 /* */ 来写内联注释
        // 虽然在实际中很少使用
        System.Console.WriteLine(/* 内联注释 */ "Hello, World!");

        System.Console.ReadLine();
    }
}
```

第57.2节：区域

区域是一个可折叠的代码块，有助于提高代码的可读性和组织性。

注意：StyleCop的规则SA1124 DoNotUseRegions不鼓励使用区域。它们通常是代码组织不良的标志，因为C#包含部分类和其他特性，使得区域变得多余。

你可以按以下方式使用区域：

```
class Program
```

Chapter 57: Comments and regions

Section 57.1: Comments

Using comments in your projects is a handy way of leaving explanations of your design choices, and should aim to make your (or someone else's) life easier when maintaining or adding to the code.

There are two ways of adding a comment to your code.

Single line comments

Any text placed after // will be treated as a comment.

```
public class Program
{
    // This is the entry point of my program.
    public static void Main()
    {
        // Prints a message to the console. - This is a comment!
        System.Console.WriteLine("Hello, World!");

        // System.Console.WriteLine("Hello, World again!"); // You can even comment out code.
        System.Console.ReadLine();
    }
}
```

Multi line or delimited comments

Any text between /* and */ will be treated as a comment.

```
public class Program
{
    public static void Main()
    {
        /*
        This is a multi line comment
        it will be ignored by the compiler.
        */
        System.Console.WriteLine("Hello, World!");

        // It's also possible to make an inline comment with /* */
        // although it's rarely used in practice
        System.Console.WriteLine(/* Inline comment */ "Hello, World!");

        System.Console.ReadLine();
    }
}
```

Section 57.2: Regions

A region is a collapsible block of code, that can help with the readability and organisation of your code.

NOTE: StyleCop's rule SA1124 DoNotUseRegions discourages use of regions. They are usually a sign of badly organized code, as C# includes partial classes and other features which make regions obsolete.

You can use regions in the following way:

```
class Program
```

```

{
    #region 应用程序入口点
    static void Main(string[] args)
    {
        PrintHelloWorld();
        System.Console.ReadLine();
    }
    #endregion

    #region 我的方法
    private static void PrintHelloWorld()
    {
        System.Console.WriteLine("Hello, World!");
    }
    #endregion
}

```

当在集成开发环境（IDE）中查看上述代码时，您可以使用 + 和 - 符号来折叠和展开代码。

展开

```

class Program
{
    #region Application entry point
    static void Main(string[] args)
    {
        PrintHelloWorld();
        System.Console.ReadLine();
    }
    #endregion

    #region My method
    private static void PrintHelloWorld()
    {
        System.Console.WriteLine("Hello, World!");
    }
    #endregion
}

```

折叠

```

class Program
{
    [Application entry point]
    [My method]
}

```

第57.3节：文档注释

XML文档注释可用于提供API文档，便于工具进行处理：

```

/// <summary>
/// 一个用于验证方法参数的辅助类。
/// </summary>
public static class Precondition
{
    /// <summary>
    ///     抛出一个带有参数的 <see cref="ArgumentOutOfRangeException"/> 异常
}

```

```

{
    #region Application entry point
    static void Main(string[] args)
    {
        PrintHelloWorld();
        System.Console.ReadLine();
    }
    #endregion

    #region My method
    private static void PrintHelloWorld()
    {
        System.Console.WriteLine("Hello, World!");
    }
    #endregion
}

```

When the above code is view in an IDE, you will be able to collapse and expand the code using the + and - symbols.

Expanded

```

class Program
{
    #region Application entry point
    static void Main(string[] args)
    {
        PrintHelloWorld();
        System.Console.ReadLine();
    }
    #endregion

    #region My method
    private static void PrintHelloWorld()
    {
        System.Console.WriteLine("Hello, World!");
    }
    #endregion
}

```

Collapsed

```

class Program
{
    [Application entry point]
    [My method]
}

```

Section 57.3: Documentation comments

XML documentation comments can be used to provide API documentation that can be easily processed by tools:

```

/// <summary>
/// A helper class for validating method arguments.
/// </summary>
public static class Precondition
{
    /// <summary>
    ///     Throws an <see cref="ArgumentOutOfRangeException"/> with the parameter
}

```

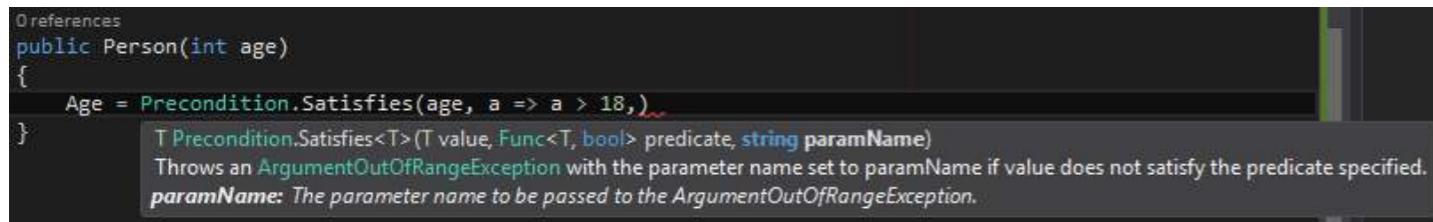
```

///<summary>
///<typeparam name="T">
///    被检查参数的类型
///</typeparam>
///<param name="value">
///    要检查的参数
///</param>
///<param name="predicate">
///    该值必须满足的谓词
///</param>
///<param name="paramName">
///    传递给
///    <see cref="ArgumentOutOfRangeException"/> 的参数名。
///</param>
///<returns>指定的值</returns>
public static T Satisfies<T>(T value, Func<T, bool> predicate, string paramName)
{
    if (!predicate(value))
        throw new ArgumentOutOfRangeException(paramName);

    return value;
}

```

文档会被 IntelliSense 即时捕获：



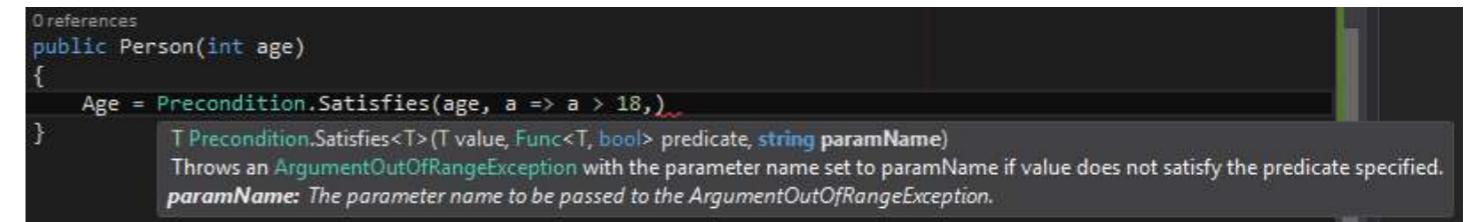
```

///      name set to <c>paramName</c> if <c>value</c> does not satisfy the
///      <c>predicate</c> specified.
///</summary>
///<typeparam name="T">
///    The type of the argument checked
///</typeparam>
///<param name="value">
///    The argument to be checked
///</param>
///<param name="predicate">
///    The predicate the value is required to satisfy
///</param>
///<param name="paramName">
///    The parameter name to be passed to the
///    <see cref="ArgumentOutOfRangeException"/>.
///</param>
///<returns>The value specified</returns>
public static T Satisfies<T>(T value, Func<T, bool> predicate, string paramName)
{
    if (!predicate(value))
        throw new ArgumentOutOfRangeException(paramName);

    return value;
}

```

Documentation is instantly picked up by IntelliSense:



第58章：继承

第58.1节：继承。构造函数调用顺序

假设我们有一个类Animal，它有一个子类Dog

```
类 Animal
{
    public Animal()
    {
        Console.WriteLine("在 Animal 的构造函数中");
    }
}

class Dog : Animal
{
    public Dog()
    {
        Console.WriteLine("在 Dog 的构造函数中");
    }
}
```

默认情况下，每个类都隐式继承Object类。

这与上面的代码相同。

```
class Animal : Object
{
    public Animal()
    {
        Console.WriteLine("在 Animal 的构造函数中");
    }
}
```

当创建Dog类的实例时，如果没有显式调用父类的其他构造函数，基类的默认构造函数（无参数）将被调用。在我们的例子中，首先会调用Object的构造函数，然后是Animal的构造函数，最后是Dog的构造函数。

```
public class Program
{
    public static void Main()
    {
        Dog dog = new Dog();
    }
}
```

输出将是

在 Animal 的构造函数中
在 Dog 的构造函数中

[查看演示](#)

显式调用父类的构造函数。

Chapter 58: Inheritance

Section 58.1: Inheritance. Constructors' calls sequence

Consider we have a class Animal which has a child class Dog

```
class Animal
{
    public Animal()
    {
        Console.WriteLine("In Animal's constructor");
    }
}

class Dog : Animal
{
    public Dog()
    {
        Console.WriteLine("In Dog's constructor");
    }
}
```

By default every class implicitly inherits the `Object` class.

This is same as the above code.

```
class Animal : Object
{
    public Animal()
    {
        Console.WriteLine("In Animal's constructor");
    }
}
```

When creating an instance of Dog class, the **base classes's default constructor (without parameters) will be called if there is no explicit call to another constructor in the parent class**. In our case, first will be called `Object`'s constructor, then `Animal`'s and at the end `Dog`'s constructor.

```
public class Program
{
    public static void Main()
    {
        Dog dog = new Dog();
    }
}
```

Output will be

In Animal's constructor
In Dog's constructor

[View Demo](#)

Call parent's constructor explicitly.

在上述示例中，我们的Dog类构造函数调用了Animal类的默认构造函数。如果需要，您可以指定调用哪个构造函数：可以调用父类中定义的任何构造函数。

假设我们有以下两个类。

```
类 Animal
{
    protected string name;

    public Animal()
    {
        Console.WriteLine("动物的默认构造函数");
    }

    public Animal(string name)
    {
        this.name = name;
    }
    Console.WriteLine("带有1个参数的动物构造函数");
    Console.WriteLine(this.name);
}

class Dog : Animal
{
    public Dog() : base()
    {
        Console.WriteLine("狗的默认构造函数");
    }

    public Dog(string name) : base(name)
    {
        Console.WriteLine("带有1个参数的狗构造函数");
        Console.WriteLine(this.name);
    }
}
```

这里发生了什么？

我们在每个类中都有两个构造函数。

base是什么意思？

base是对父类的引用。在我们的例子中，当我们像这样创建Dog类的实例时

```
Dog dog = new Dog();
```

运行时首先调用无参数构造函数Dog()，但其主体不会立即执行。

在构造函数的括号后面，我们有这样一个调用：`:base()`，这意味着当我们调用默认的Dog构造函数时，它会依次调用父类的默认构造函数。父类的构造函数执行完毕后，会返回然后，最后执行Dog()构造函数体。

所以输出将会是：

Animal的默认构造函数
Dog的默认构造函数

In the above examples, our Dog class constructor calls the **default** constructor of the Animal class. If you want, you can specify which constructor should be called: it is possible to call any constructor which is defined in the parent class.

Consider we have these two classes.

```
class Animal
{
    protected string name;

    public Animal()
    {
        Console.WriteLine("Animal's default constructor");
    }

    public Animal(string name)
    {
        this.name = name;
    }
    Console.WriteLine("Animal's constructor with 1 parameter");
    Console.WriteLine(this.name);
}

class Dog : Animal
{
    public Dog() : base()
    {
        Console.WriteLine("Dog's default constructor");
    }

    public Dog(string name) : base(name)
    {
        Console.WriteLine("Dog's constructor with 1 parameter");
        Console.WriteLine(this.name);
    }
}
```

What is going here?

We have 2 constructors in each class.

What does `base` mean?

`base` is a reference to the parent class. In our case, when we create an instance of Dog class like this

```
Dog dog = new Dog();
```

The runtime first calls the `Dog()`, which is the parameterless constructor. But its body doesn't work immediately. After the parentheses of the constructor we have a such call: `base()`, which means that when we call the default Dog constructor, it will in turn call the parent's **default** constructor. After the parent's constructor runs, it will return and then, finally, run the `Dog()` constructor body.

So output will be like this:

Animal's default constructor
Dog's default constructor

那么如果我们调用带参数的Dog构造函数会怎样？

```
Dog dog = new Dog("Rex");
```

你知道父类中非私有的成员会被子类继承，这意味着Dog也会有name字段。

在这种情况下，我们向构造函数传递了一个参数。它会将该参数传递给父类的带参数构造函数，后者初始化了name字段。

输出将是

带有1个参数的Animal构造函数

雷克斯

带有1个参数的Dog构造函数

雷克斯

摘要：

每个对象的创建都始于基类。在继承中，层级中的类是链式连接的。由于所有类都派生自Object，创建任何对象时首先调用的构造函数是Object类的构造函数；然后调用链中的下一个构造函数，只有当所有构造函数都被调用后，对象才被创建

base关键字

1. base关键字用于在派生类中访问基类的成员：
2. 调用基类中被另一个方法重写的方法。指定在创建派生类实例时应调用哪个基类构造函数。

第58.2节：从基类继承

为了避免代码重复，在通用类中定义公共方法和属性作为基类：

```
public class Animal
{
    public string Name { get; set; }
    // 所有动物共有的方法和属性
    public void Eat(Object dinner)
    {
        // ...
    }
    public void 凝视()
    {
        // ...
    }
    public void 翻滚()
    {
        // ...
    }
}
```

既然你已经有了一个表示动物（Animal）的一般类，你就可以定义一个描述特定动物特性的类：

Now what if we call the Dog's constructor with a parameter?

```
Dog dog = new Dog("Rex");
```

You know that members in the parent class which are not private are inherited by the child class, meaning that Dog will also have the name field.

In this case we passed an argument to our constructor. It in turn passes the argument to the parent class' **constructor with a parameter**, which initializes the name field.

Output will be

```
Animal's constructor with 1 parameter
Rex
Dog's constructor with 1 parameter
Rex
```

Summary:

Every object creation starts from the base class. In the inheritance, the classes which are in the hierarchy are chained. As all classes derive from Object, the first constructor to be called when any object is created is the Object class constructor; Then the next constructor in the chain is called and only after all of them are called the object is created

base keyword

1. The base keyword is used to access members of the base class from within a derived class:
2. Call a method on the base class that has been overridden by another method. Specify which base-class constructor should be called when creating instances of the derived class.

Section 58.2: Inheriting from a base class

To avoid duplicating code, define common methods and attributes in a general class as a base:

```
public class Animal
{
    public string Name { get; set; }
    // Methods and attributes common to all animals
    public void Eat(Object dinner)
    {
        // ...
    }
    public void Stare()
    {
        // ...
    }
    public void Roll()
    {
        // ...
    }
}
```

Now that you have a class that represents Animal in general, you can define a class that describes the peculiarities of specific animals:

```

public class 猫 : 动物
{
    public Cat()
    {
        Name = "Cat";
    }
    // 抓家具和无视主人的方法
    public void 抓挠(Object 家具)
    {
        // ...
    }
}

```

Cat 类不仅可以访问其定义中明确描述的方法，还可以访问一般动物基类中定义的所有方法。任何动物（无论是否是猫）都可以吃、凝视或翻滚。然而，除非它也是猫，否则动物无法抓挠。你还可以定义其他描述其他动物的类。（例如带有破坏花园方法的地鼠类和没有额外方法的树懒类。）

第58.3节：继承类并实现接口

```

public class Animal
{
    public string Name { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

// 注意在 C# 中，基类名称必须位于接口名称之前
public class 猫 : 动物, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
    }

    public string MakeNoise()
    {
        return "Nyan";
    }
}

```

第58.4节：继承类并实现多个接口

```

public class LivingBeing
{
    string Name { get; set; }
}

public interface IAnimal
{
    bool HasHair { get; set; }
}

```

```

public class Cat : Animal
{
    public Cat()
    {
        Name = "Cat";
    }
    // Methods for scratching furniture and ignoring owner
    public void Scratch(Object furniture)
    {
        // ...
    }
}

```

The Cat class gets access to not only the methods described in its definition explicitly, but also all the methods defined in the general Animal base class. Any Animal (whether or not it was a Cat) could Eat, Stare, or Roll. An Animal would not be able to Scratch, however, unless it was also a Cat. You could then define other classes describing other animals. (Such as Gopher with a method for destroying flower gardens and Sloth with no extra methods at all.)

Section 58.3: Inheriting from a class and implementing an interface

```

public class Animal
{
    public string Name { get; set; }
}

public interface INoiseMaker
{
    string MakeNoise();
}

// Note that in C#, the base class name must come before the interface names
public class Cat : Animal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
    }

    public string MakeNoise()
    {
        return "Nyan";
    }
}

```

Section 58.4: Inheriting from a class and implementing multiple interfaces

```

public class LivingBeing
{
    string Name { get; set; }
}

public interface IAnimal
{
    bool HasHair { get; set; }
}

```

```

public interface INoiseMaker
{
    string MakeNoise();
}

// 注意在 C# 中，基类名称必须位于接口名称之前
public class Cat : LivingBeing, IAnimal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
        HasHair = true;
    }

    public bool HasHair { get; set; }

    public string Name { get; set; }

    public string MakeNoise()
    {
        return "Nyan";
    }
}

```

第58.5节：子类中的构造函数

当你创建一个基类的子类时，可以通过在子类构造函数的参数后使用: `base`来构造基类。

```

class Instrument
{
    string type;
    bool clean;

    public Instrument (string type, bool clean)
    {
        this.类型 = 类型;
        this.清理 = 清理;
    }
}

类 小号 : 乐器
{
    布尔型 上油;

    公共 小号(字符串 类型, 布尔型 清理, 布尔型 上油) : 基类(类型, 清理)
    {
        this.上油 = 上油;
    }
}

```

第58.6节：继承反模式

不当继承

假设有两个类，类Foo和Bar。Foo有两个功能Do1和Do2。Bar需要使用Foo的Do1，但它不需要Do2，或者需要一个等同于Do2但功能完全不同的特性。

错误方式：将Do2()设为Foo的虚方法，然后在Bar中重写它，或者在Bar中直接为Do2()抛出异常Exception

```

public interface INoiseMaker
{
    string MakeNoise();
}

// Note that in C#, the base class name must come before the interface names
public class Cat : LivingBeing, IAnimal, INoiseMaker
{
    public Cat()
    {
        Name = "Cat";
        HasHair = true;
    }

    public bool HasHair { get; set; }

    public string Name { get; set; }

    public string MakeNoise()
    {
        return "Nyan";
    }
}

```

Section 58.5: Constructors In A Subclass

When you make a subclass of a base class, you can construct the base class by using : `base` after the subclass constructor's parameters.

```

class Instrument
{
    string type;
    bool clean;

    public Instrument (string type, bool clean)
    {
        this.type = type;
        this.clean = clean;
    }
}

class Trumpet : Instrument
{
    bool oiled;

    public Trumpet(string type, bool clean, bool oiled) : base(type, clean)
    {
        this.oiled = oiled;
    }
}

```

Section 58.6: Inheritance Anti-patterns

Improper Inheritance

Lets say there are 2 classes class Foo and Bar. Foo has two features Do1 and Do2. Bar needs to use Do1 from Foo, but it doesn't need Do2 or needs feature that is equivalent to Do2 but does something completely different.

Bad way: make Do2() on Foo virtual then override it in Bar or just `throw` Exception in Bar for Do2()

```
公共类 Bar : Foo
{
    public override void Do2()
    {
        //做一些你完全不会期望Foo去做的事情
        //或者简单地抛出新的异常
    }
}
```

好方法

将Do1()从Foo中取出，放入新类Baz中，然后让Foo和Bar都继承Baz，并分别实现Do2()

```
public class Baz
{
    public void Do1()
    {
        // 魔法代码
    }
}

public class Foo : Baz
{
    public void Do2()
    {
        // foo的方式
    }
}

public class Bar : Baz
{
    public void Do2()
    {
        // bar方式或者根本没有Do2
    }
}
```

现在为什么第一个例子不好而第二个例子好：当开发者nr2需要修改Foo时，很可能会破坏Bar的实现，因为Bar现在与Foo密不可分。使用后一个例子时，Foo和Bar的公共部分已被移到Baz，它们彼此不影响（正如它们不应该影响彼此一样）。

第58.7节：扩展抽象基类

与可以被描述为实现契约的接口不同，抽象类充当扩展的契约。

抽象类不能被实例化，必须被继承，继承后的类（或派生类）才能被实例化。

抽象类用于提供通用实现

```
public abstract class Car
{
    public void HonkHorn()
    {
        // 喇叭鸣响的实现
    }
}
```

```
public class Bar : Foo
{
    public override void Do2()
    {
        //Does something completely different that you would expect Foo to do
        //or simply throws new Exception
    }
}
```

Good way

Take out Do1() from Foo and put it into new class Baz then inherit both Foo and Bar from Baz and implement Do2() separately

```
public class Baz
{
    public void Do1()
    {
        // magic
    }
}

public class Foo : Baz
{
    public void Do2()
    {
        // foo way
    }
}

public class Bar : Baz
{
    public void Do2()
    {
        // bar way or not have Do2 at all
    }
}
```

Now why first example is bad and second is good: When developer nr2 has to do a change in Foo, chances are he will break implementation of Bar because Bar is now inseparable from Foo. When doing it by latter example Foo and Bar commonality has been moved to Baz and they do not affect each other (like they shouldn't).

Section 58.7: Extending an abstract base class

Unlike interfaces, which can be described as contracts for implementation, abstract classes act as contracts for extension.

An abstract class cannot be instantiated, it must be extended and the resulting class (or derived class) can then be instantiated.

Abstract classes are used to provide generic implementations

```
public abstract class Car
{
    public void HonkHorn()
    {
        // Implementation of horn being honked
    }
}
```

```

public class Mustang : Car
{
    // 仅仅通过继承抽象类 Car, Mustang 就可以调用 HonkHorn()// 如果 Car 是一个接口, HonkHorn 方法需要包含在每个实现该接口的类中。
}

```

上面的例子展示了任何继承自汽车 (Car) 类的子类都会自动获得带有实现的HonkHorn方法。这意味着任何开发者创建新的汽车类时，不需要担心它如何鸣喇叭。

第58.8节：测试和导航继承

```

interface BaseInterface {}
class BaseClass : BaseInterface {}

interface DerivedInterface {}
class DerivedClass : BaseClass, DerivedInterface {}

var baseInterfaceType = typeof(BaseInterface);
var derivedInterfaceType = typeof(DerivedInterface);
var baseType = typeof(BaseClass);
var derivedType = typeof(DerivedClass);

var baseInstance = new BaseClass();
var derivedInstance = new DerivedClass();

Console.WriteLine(derivedInstance is DerivedClass); //True
Console.WriteLine(derivedInstance is DerivedInterface); //True
Console.WriteLine(derivedInstance is BaseClass); //True
Console.WriteLine(derivedInstance is BaseInterface); //True
Console.WriteLine(derivedInstance is object); //True

Console.WriteLine(derivedType.BaseType.Name); //BaseClass
Console.WriteLine(baseType.BaseType.Name); //Object
Console.WriteLine(typeof(object).BaseType); //null

Console.WriteLine(baseType.IsInstanceOfType(derivedInstance)); //True
Console.WriteLine(derivedType.IsInstanceOfType(baseInstance)); //False

Console.WriteLine(
    string.Join(",",
    derivedType.GetInterfaces().Select(t => t.Name).ToArray()));
//BaseInterface,DerivedInterface

Console.WriteLine(baseInterfaceType.IsAssignableFrom(derivedType)); //True
Console.WriteLine(derivedInterfaceType.IsAssignableFrom(derivedType)); //True
Console.WriteLine(derivedInterfaceType.IsAssignableFrom(baseType)); //False

```

第58.9节：继承方法

方法可以通过多种方式继承

```

public abstract class Car
{
    public void HonkHorn()
        // 喇叭鸣响的实现
}

```

```

public class Mustang : Car
{
    // Simply by extending the abstract class Car, the Mustang can HonkHorn()
    // If Car were an interface, the HonkHorn method would need to be included
    // in every class that implemented it.
}

```

The above example shows how any class extending Car will automatically receive the HonkHorn method with the implementation. This means that any developer creating a new Car will not need to worry about how it will honk its horn.

Section 58.8: Testing and navigating inheritance

```

interface BaseInterface {}
class BaseClass : BaseInterface {}

interface DerivedInterface {}
class DerivedClass : BaseClass, DerivedInterface {}

var baseInterfaceType = typeof(BaseInterface);
var derivedInterfaceType = typeof(DerivedInterface);
var baseType = typeof(BaseClass);
var derivedType = typeof(DerivedClass);

var baseInstance = new BaseClass();
var derivedInstance = new DerivedClass();

Console.WriteLine(derivedInstance is DerivedClass); //True
Console.WriteLine(derivedInstance is DerivedInterface); //True
Console.WriteLine(derivedInstance is BaseClass); //True
Console.WriteLine(derivedInstance is BaseInterface); //True
Console.WriteLine(derivedInstance is object); //True

Console.WriteLine(derivedType.BaseType.Name); //BaseClass
Console.WriteLine(baseType.BaseType.Name); //Object
Console.WriteLine(typeof(object).BaseType); //null

Console.WriteLine(baseType.IsInstanceOfType(derivedInstance)); //True
Console.WriteLine(derivedType.IsInstanceOfType(baseInstance)); //False

Console.WriteLine(
    string.Join(",",
    derivedType.GetInterfaces().Select(t => t.Name).ToArray()));
//BaseInterface,DerivedInterface

Console.WriteLine(baseInterfaceType.IsAssignableFrom(derivedType)); //True
Console.WriteLine(derivedInterfaceType.IsAssignableFrom(derivedType)); //True
Console.WriteLine(derivedInterfaceType.IsAssignableFrom(baseType)); //False

```

Section 58.9: Inheriting methods

There are several ways methods can be inherited

```

public abstract class Car
{
    public void HonkHorn()
        // Implementation of horn being honked
}

```

```

// 虚方法可以在派生类中被重写
public virtual void ChangeGear() {
    // 换挡的实现
}

// 抽象方法必须在派生类中被重写
public abstract void Accelerate();

}

public class Mustang : Car
{
    // 在向Mustang类添加任何代码之前，它已经包含了
    // HonkHorn和ChangeGear的实现。

    // 为了能够编译，必须为Accelerate提供实现，
    // 这通过使用override关键字完成
    public override void Accelerate() {
        // Mustang加速的实现
    }

    // 如果Mustang换挡方式与Car中的实现不同
    // 可以使用上面相同的override关键字进行重写
    public override void ChangeGear() {
        // Mustang换挡的实现
    }
}

```

```

// virtual methods CAN be overridden in derived classes
public virtual void ChangeGear() {
    // Implementation of gears being changed
}

// abstract methods MUST be overridden in derived classes
public abstract void Accelerate();

}

public class Mustang : Car
{
    // Before any code is added to the Mustang class, it already contains
    // implementations of HonkHorn and ChangeGear.

    // In order to compile, it must be given an implementation of Accelerate,
    // this is done using the override keyword
    public override void Accelerate() {
        // Implementation of Mustang accelerating
    }

    // If the Mustang changes gears differently to the implementation in Car
    // this can be overridden using the same override keyword as above
    public override void ChangeGear() {
        // Implementation of Mustang changing gears
    }
}

```

第58.10节：带递归类型指定的基类

泛型基类的递归类型指定的一次性定义。每个节点有一个父节点和多个子节点。

```

/// <summary>
/// 树结构的泛型基类
/// </summary>
/// <typeparam name="T">树的节点类型</typeparam>
public abstract class Tree<T> where T : Tree<T>
{
    /// <summary>
    /// 构造函数设置父节点并将此节点添加到父节点的子节点中
    /// </summary>
    /// <param name="parent">父节点，如果是根节点则为null</param>
    protected Tree(T parent)
    {
        this.Parent=parent;
        this.Children=new List<T>();
        if(parent!=null)
        {
            parent.Children.Add(this as T);
        }
    }

    public T Parent { get; private set; }
    public List<T> Children { get; private set; }
    public bool IsRoot { get { return Parent==null; } }
    public bool IsLeaf { get { return Children.Count==0; } }

    /// <summary>
    /// 返回到根对象的跳数
    /// </summary>
    public int Level { get { return IsRoot ? 0 : Parent.Level+1; } }
}

```

Section 58.10: Base class with recursive type specification

One time definition of a generic base class with recursive type specifier. Each node has one parent and multiple children.

```

/// <summary>
/// Generic base class for a tree structure
/// </summary>
/// <typeparam name="T">The node type of the tree</typeparam>
public abstract class Tree<T> where T : Tree<T>
{
    /// <summary>
    /// Constructor sets the parent node and adds this node to the parent's child nodes
    /// </summary>
    /// <param name="parent">The parent node or null if a root</param>
    protected Tree(T parent)
    {
        this.Parent=parent;
        this.Children=new List<T>();
        if(parent!=null)
        {
            parent.Children.Add(this as T);
        }
    }

    public T Parent { get; private set; }
    public List<T> Children { get; private set; }
    public bool IsRoot { get { return Parent==null; } }
    public bool IsLeaf { get { return Children.Count==0; } }

    /// <summary>
    /// Returns the number of hops to the root object
    /// </summary>
    public int Level { get { return IsRoot ? 0 : Parent.Level+1; } }
}

```

上述代码可以在每次需要定义对象的树形层级时重复使用。树中的节点对象必须继承自带有以下内容的基类

```
public class MyNode : Tree<MyNode>
{
    // stuff
}
```

每个节点类都知道它在层级中的位置，知道父对象是什么，以及子对象是什么。几个内置类型使用树结构，比如Control或XmlElement，上述的Tree<T>可以用作代码中任何类型的基类。

例如，要创建一个零件层级，其中总重量是从所有子零件的重量计算得出，请执行以下操作：

```
public class Part : Tree<Part>
{
    public static readonly Part Empty = new Part(null) { Weight=0 };
    public Part(Part parent) : base(parent) { }
    public Part Add(float weight)
    {
        return new Part(this) { Weight=weight };
    }
    public float Weight { get; set; }

    public float TotalWeight { get { return Weight+Children.Sum((part) => part.TotalWeight); } }
}
```

用法示例

```
// [Q:2.5] -- [P:4.2] -- [R:0.4]
//   |
//   - [Z:0.8]
var Q = Part.Empty.Add(2.5f);
var P = Q.Add(4.2f);
var R = P.Add(0.4f);
var Z = Q.Add(0.9f);

// 2.5+(4.2+0.4)+0.9 = 8.0
float weight = Q.TotalWeight;
```

另一个例子是在定义相对坐标系时。在这种情况下，坐标系的真实位置取决于所有父坐标系的位置。

```
public class RelativeCoordinate : Tree<RelativeCoordinate>
{
    public static readonly RelativeCoordinate Start = new RelativeCoordinate(null, PointF.Empty) {
    };
    public RelativeCoordinate(RelativeCoordinate parent, PointF local_position)
        : base(parent)
    {
        this.LocalPosition=local_position;
    }
    public PointF 本地位置 { 获取; 设置; }
    public PointF 全局位置
    {
        获取
        {
            如果(IsRoot) 返回 LocalPosition;
        }
    }
}
```

The above can be re-used every time a tree hierarchy of objects needs to be defined. The node object in the tree has to inherit from the base class with

```
public class MyNode : Tree<MyNode>
{
    // stuff
}
```

each node class knows where it is in the hierarchy, what the parent object is as well as what the children objects are. Several built in types use a tree structure, like Control or XmlElement and the above Tree<T> can be used as a base class of *any* type in your code.

For example, to create a hierarchy of parts where the total weight is calculated from the weight of *all* the children, do the following:

```
public class Part : Tree<Part>
{
    public static readonly Part Empty = new Part(null) { Weight=0 };
    public Part(Part parent) : base(parent) { }
    public Part Add(float weight)
    {
        return new Part(this) { Weight=weight };
    }
    public float Weight { get; set; }

    public float TotalWeight { get { return Weight+Children.Sum((part) => part.TotalWeight); } }
}
```

to be used as

```
// [Q:2.5] -- [P:4.2] -- [R:0.4]
//   \
//   - [Z:0.8]
var Q = Part.Empty.Add(2.5f);
var P = Q.Add(4.2f);
var R = P.Add(0.4f);
var Z = Q.Add(0.9f);

// 2.5+(4.2+0.4)+0.9 = 8.0
float weight = Q.TotalWeight;
```

Another example would be in the definition of relative coordinate frames. In this case the true position of the coordinate frame depends on the positions of *all* the parent coordinate frames.

```
public class RelativeCoordinate : Tree<RelativeCoordinate>
{
    public static readonly RelativeCoordinate Start = new RelativeCoordinate(null, PointF.Empty) {
    };
    public RelativeCoordinate(RelativeCoordinate parent, PointF local_position)
        : base(parent)
    {
        this.LocalPosition=local_position;
    }
    public PointF LocalPosition { get; set; }
    public PointF GlobalPosition
    {
        get
        {
            if(IsRoot) return LocalPosition;
        }
    }
}
```

```

        变量 parent_pos = Parent.GlobalPosition;
        返回 新的 PointF(parent_pos.X+LocalPosition.X, parent_pos.Y+LocalPosition.Y);
    }

    公共 浮点数 TotalDistance
    {
        get
        {
            浮点数 dist =
(浮点数)Math.Sqrt(LocalPosition.X*LocalPosition.X+LocalPosition.Y*LocalPosition.Y);
            返回 IsRoot ? dist : Parent.TotalDistance+dist;
        }
    }

    公共 RelativeCoordinate Add(PointF local_position)
    {
        返回 新的 RelativeCoordinate(this, local_position);
    }

    public RelativeCoordinate Add(float x, float y)
    {
        return Add(new PointF(x, y));
    }
}

```

用法示例

```

// 定义以下坐标系层级
//
// o--> [A1] ---+--> [B1] -----> [C1]
//           |
//           +--> [B2] ---+--> [C2]
//           |
//           +--> [C3]

var A1 = RelativeCoordinate.Start;
var B1 = A1.Add(100, 20);
var B2 = A1.Add(160, 10);

var C1 = B1.Add(120, -40);
var C2 = B2.Add(80, -20);
var C3 = B2.Add(60, -30);

double dist1 = C1.TotalDistance;

```

```

        var parent_pos = Parent.GlobalPosition;
        return new PointF(parent_pos.X+LocalPosition.X, parent_pos.Y+LocalPosition.Y);
    }

    public float TotalDistance
    {
        get
        {
            float dist =
(float)Math.Sqrt(LocalPosition.X*LocalPosition.X+LocalPosition.Y*LocalPosition.Y);
            return IsRoot ? dist : Parent.TotalDistance+dist;
        }
    }

    public RelativeCoordinate Add(PointF local_position)
    {
        return new RelativeCoordinate(this, local_position);
    }

    public RelativeCoordinate Add(float x, float y)
    {
        return Add(new PointF(x, y));
    }
}

```

to be used as

```

// Define the following coordinate system hierarchy
//
// o--> [A1] ---+--> [B1] -----> [C1]
//           |
//           +--> [B2] ---+--> [C2]
//           |
//           +--> [C3]

var A1 = RelativeCoordinate.Start;
var B1 = A1.Add(100, 20);
var B2 = A1.Add(160, 10);

var C1 = B1.Add(120, -40);
var C2 = B2.Add(80, -20);
var C3 = B2.Add(60, -30);

double dist1 = C1.TotalDistance;

```

第59章：泛型

参数	描述
T, V	泛型声明的类型占位符

第59.1节：隐式类型推断（方法）

在向泛型方法传递形式参数时，相关的泛型类型参数通常可以隐式推断。如果所有泛型类型都能被推断，则在语法中指定它们是可选的。

考虑以下泛型方法。它有一个形式参数和一个泛型类型参数。它们之间有一个非常明显的关系——传递给泛型类型参数的类型必须与传递给形式参数的参数的编译时类型相同。

```
void M<T>(T obj)
{ }
```

这两个调用是等价的：

```
M<object>(new object());
M(new object());
```

这两个调用也是等价的：

```
M<string>("");
M("");
```

这三个调用也是如此：

```
M<object>("");
M((object ""));
M("") as object);
```

注意，如果至少有一个类型参数无法推断，那么所有类型参数都必须被指定。

考虑以下泛型方法。第一个泛型类型参数与形式参数的类型相同。但第二个泛型类型参数没有这样的关系。因此，编译器无法推断对该方法的任何调用中的第二个泛型类型参数。

```
void X<T1, T2>(T1 obj)
{ }
```

这不再起作用：

```
X("");
```

这也行，因为编译器不确定我们是在指定第一个还是第二个泛型参数（两者作为object都是有效的）：

```
X<object>("");
```

Chapter 59: Generics

Parameter(s)	Description
T, V	Type placeholders for generic declarations

Section 59.1: Implicit type inference (methods)

When passing formal arguments to a generic method, relevant generic type arguments can usually be inferred implicitly. If all generic type can be inferred, then specifying them in the syntax is optional.

Consider the following generic method. It has one formal parameter and one generic type parameter. There is a very obvious relationship between them -- the type passed as an argument to the generic type parameter must be the same as the compile-time type of the argument passed to the formal parameter.

```
void M<T>(T obj)
{ }
```

These two calls are equivalent:

```
M<object>(new object());
M(new object());
```

These two calls are also equivalent:

```
M<string>("");
M("");
```

And so are these three calls:

```
M<object>("");
M((object ""));
M("") as object);
```

Notice that if at least one type argument cannot be inferred, then all of them have to be specified.

Consider the following generic method. The first generic type argument is the same as the type of the formal argument. But there is no such relationship for the second generic type argument. Therefore, the compiler has no way of inferring the second generic type argument in any call to this method.

```
void X<T1, T2>(T1 obj)
{ }
```

This doesn't work anymore:

```
X("");
```

This doesn't work either, because the compiler isn't sure if we are specifying the first or the second generic parameter (both would be valid as object):

```
X<object>("");
```

我们需要同时写出它们两个，像这样：

```
X<string, object>("");
```

第59.2节：类型推断（类）

开发者可能会因为类型推断对构造函数不起作用而感到困惑：

```
类 Tuple<T1,T2>
{
    公共 Tuple(T1 value1, T2 value2)
    {
    }
}

var x = new Tuple(2, "two");           // 这将不起作用...
var y = new Tuple<int, string>(2, "two"); // 即使显式形式可以。
```

第一种不显式指定类型参数的实例创建方式会导致编译时错误，错误信息为：

使用泛型类型 'Tuple<T1, T2>' 需要 2 个类型参数

一个常见的解决方法是在静态类中添加一个辅助方法：

```
静态类 Tuple
{
    公共 static Tuple<T1, T2> Create<T1, T2>(T1 value1, T2 value2)
    {
        return new Tuple<T1, T2>(value1, value2);
    }
}

var x = Tuple.Create(2, "two"); // 这将会生效...
```

第59.3节：使用带有接口作为约束类型的泛型方法

这是一个示例，展示如何在Animal类的Eat方法中使用泛型类型IFood

```
public interface IFood
{
    void EatenBy(Animal animal);
}

public class Grass: IFood
{
    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Grass 被以下对象吃掉了: {0}", animal.Name);
    }
}

public class Animal
{
    public string Name { get; set; }
```

We are required to type out both of them, like this:

```
X<string, object>("") ;
```

Section 59.2: Type inference (classes)

Developers can be caught out by the fact that type inference *doesn't work* for constructors:

```
class Tuple<T1,T2>
{
    public Tuple(T1 value1, T2 value2)
    {
    }
}

var x = new Tuple(2, "two");           // This WON'T work...
var y = new Tuple<int, string>(2, "two"); // even though the explicit form will.
```

The first way of creating instance without explicitly specifying type parameters will cause compile time error which would say:

Using the generic type 'Tuple<T1, T2>' requires 2 type arguments

A common workaround is to add a helper method in a static class:

```
static class Tuple
{
    public static Tuple<T1, T2> Create<T1, T2>(T1 value1, T2 value2)
    {
        return new Tuple<T1, T2>(value1, value2);
    }
}

var x = Tuple.Create(2, "two"); // This WILL work...
```

Section 59.3: Using generic method with an interface as a constraint type

This is an example of how to use the generic type IFood inside Eat method on the class Animal

```
public interface IFood
{
    void EatenBy(Animal animal);
}

public class Grass: IFood
{
    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Grass was eaten by: {0}", animal.Name);
    }
}

public class Animal
{
    public string Name { get; set; }
```

```

public void Eat<TFood>(TFood food)
    where TFood : IFood
{
    food.EatenBy(this);
}

public class 食肉动物 : 动物
{
    public 食肉动物()
    {
        Name = "食肉动物";
    }
}

public class 食草动物 : 动物, IFood
{
    public 食草动物()
    {
        Name = "食草动物";
    }

    public void EatenBy(Animal animal)
    {
        Console.WriteLine("食草动物被以下动物吃掉了: {0}", animal.Name);
    }
}

```

你可以这样调用 Eat 方法：

```

var 草 = new 草();
var 羊 = new 食草动物();
var 狮子 = new 食肉动物();

羊.Eat(草);
//输出：草被吃掉了：草食动物

狮子.吃(羊);
//输出：草食动物被吃掉了：肉食动物

```

在这种情况下，如果你尝试调用：

```
羊.吃(狮子);
```

这是不可能的，因为对象狮子没有实现接口IFood。尝试进行上述调用将产生编译错误：“类型 'Carnivore' 不能用作泛型类型或方法 'Animal.Eat(TFood)' 中的类型参数 'TFood'。没有从 'Carnivore' 到 'IFood' 的隐式引用转换。”

第59.4节：类型约束 (new关键字)

通过使用 new() 约束，可以强制类型参数定义一个空的（默认）构造函数。

```

类 Foo
{
    公共 Foo () { }
}

class Bar
{

```

```

public void Eat<TFood>(TFood food)
    where TFood : IFood
{
    food.EatenBy(this);
}

public class Carnivore : Animal
{
    public Carnivore()
    {
        Name = "Carnivore";
    }
}

public class Herbivore : Animal, IFood
{
    public Herbivore()
    {
        Name = "Herbivore";
    }

    public void EatenBy(Animal animal)
    {
        Console.WriteLine("Herbivore was eaten by: {0}", animal.Name);
    }
}

```

You can call the Eat method like this:

```

var grass = new Grass();
var sheep = new Herbivore();
var lion = new Carnivore();

sheep.Eat(grass);
//Output: Grass was eaten by: Herbivore

lion.Eat(sheep);
//Output: Herbivore was eaten by: Carnivore

```

In this case if you try to call:

```
sheep.Eat(lion);
```

It won't be possible because the object lion does not implement the interface IFood. Attempting to make the above call will generate a compiler error: "The type 'Carnivore' cannot be used as type parameter 'TFood' in the generic type or method 'Animal.Eat(TFood)'. There is no implicit reference conversion from 'Carnivore' to 'IFood'."

Section 59.4: Type constraints (new-keyword)

By using the new() constraint, it is possible to enforce type parameters to define an empty (default) constructor.

```

class Foo
{
    public Foo () { }
}

class Bar
{

```

```

    public Bar (string s) { ... }

}

class Factory<T>
    where T : new()
{
    public T Create()
    {
        return new T();
    }
}

Foo f = new Factory<Foo>().Create(); // 有效。
Bar b = new Factory<Bar>().Create(); // 无效, Bar 没有定义默认/无参构造函数。

```

第二次调用 Create() 会产生编译时错误，错误信息如下：

“Bar” 必须是一个非抽象类型，并且具有公共的无参构造函数，才能作为泛型类型或方法“Factory”中的参数 “T”。

构造函数不支持带参数的约束，只支持无参构造函数。

第59.5节：类型约束（类和接口）

类型约束能够强制类型参数实现某个接口或类。

```

interface IType;
interface IAnotherType;

// T 必须是 IType 的子类型
interface IGeneric<T>
    where T : IType
{ }

// T 必须是 IType 的子类型
class Generic<T>
    where T : IType
{ }

class NonGeneric
{
    // T 必须是 IType 的子类型
    public void DoSomething<T>(T arg)
        where T : IType
    {
    }
}

// 有效的定义和表达式：
class Type : IType { }
class Sub : IGeneric<Type> { }
class Sub : Generic<Type> { }
new NonGeneric().DoSomething(new Type());

// 无效的定义和表达式：
class AnotherType : IAnotherType { }

```

```

    public Bar (string s) { ... }

}

class Factory<T>
    where T : new()
{
    public T Create()
    {
        return new T();
    }
}

Foo f = new Factory<Foo>().Create(); // Valid.
Bar b = new Factory<Bar>().Create(); // Invalid, Bar does not define a default/empty constructor.

```

The second call to to Create() will give compile time error with following message:

'Bar' must be a non-abstract type with a public parameterless constructor in order to use it as parameter 'T' in the generic type or method 'Factory'

There is no constraint for a constructor with parameters, only parameterless constructors are supported.

Section 59.5: Type constraints (classes and interfaces)

Type constraints are able to force a type parameter to implement a certain interface or class.

```

interface IType;
interface IAnotherType;

// T must be a subtype of IType
interface IGeneric<T>
    where T : IType
{ }

// T must be a subtype of IType
class Generic<T>
    where T : IType
{ }

class NonGeneric
{
    // T must be a subtype of IType
    public void DoSomething<T>(T arg)
        where T : IType
    {
    }
}

// Valid definitions and expressions:
class Type : IType { }
class Sub : IGeneric<Type> { }
class Sub : Generic<Type> { }
new NonGeneric().DoSomething(new Type());

// Invalid definitions and expressions:
class AnotherType : IAnotherType { }

```

```
class Sub : IGeneric<AnotherType> { }
class Sub : Generic<AnotherType> { }
new NonGeneric().DoSomething(new AnotherType());
```

多重约束的语法：

```
class Generic<T, T1>
    where T : IType
    where T1 : Base, new()
{ }
```

类型约束的工作方式与继承相同，可以指定多个接口作为泛型类型的约束，但只能有一个类：

```
class A { /* ... */ }
class B { /* ... */ }

interface I1 { }
interface I2 { }

class Generic<T>
    where T : A, I1, I2
{ }

class Generic2<T>
    where T : A, B //编译错误
{ }
```

另一条规则是类必须作为第一个约束添加，然后才是接口：

```
class Generic<T>
    where T : A, I1
{ }

class Generic2<T>
    where T : I1, A //编译错误
{ }
```

所有声明的约束必须同时满足，特定的泛型实例才能生效。无法指定两个或多个备选的约束集合。

第59.6节：检查泛型值的相等性

如果泛型类或方法的逻辑需要检查具有泛型类型的值的相等性，请使用 EqualityComparer<TType>.Default 属性：

```
public void Foo<TBar>(TBar arg1, TBar arg2)
{
    var comparer = EqualityComparer<TBar>.Default;
    if (comparer.Equals(arg1, arg2))
    {
        ...
    }
}
```

```
class Sub : IGeneric<AnotherType> { }
class Sub : Generic<AnotherType> { }
new NonGeneric().DoSomething(new AnotherType());
```

Syntax for multiple constraints:

```
class Generic<T, T1>
    where T : IType
    where T1 : Base, new()
{ }
```

Type constraints works in the same way as inheritance, in that it is possible to specify multiple interfaces as constraints on the generic type, but only one class:

```
class A { /* ... */ }
class B { /* ... */ }

interface I1 { }
interface I2 { }

class Generic<T>
    where T : A, I1, I2
{ }

class Generic2<T>
    where T : A, B //Compilation error
{ }
```

Another rule is that the class must be added as the first constraint and then the interfaces:

```
class Generic<T>
    where T : A, I1
{ }

class Generic2<T>
    where T : I1, A //Compilation error
{ }
```

All declared constraints must be satisfied simultaneously for a particular generic instantiation to work. There is no way to specify two or more alternative sets of constraints.

Section 59.6: Checking equality of generic values

If logic of generic class or method requires checking equality of values having generic type, use EqualityComparer<TType>.Default property:

```
public void Foo<TBar>(TBar arg1, TBar arg2)
{
    var comparer = EqualityComparer<TBar>.Default;
    if (comparer.Equals(arg1, arg2))
    {
        ...
    }
}
```

```
}
```

这种方法优于简单调用Object.Equals()方法，因为默认的比较器实现会检查TBar类型是否实现了IEquatable<TBar>接口，如果是，则调用IEquatable<TBar>.Equals(TBarother)方法。这样可以避免值类型的装箱/拆箱操作。

第59.7节：反思类型参数

typeof操作符适用于类型参数。

```
类 NameGetter<T>
{
    公共字符串 GetTypeName()
    {
        返回 typeof(T).Name;
    }
}
```

第59.8节：协变

什么时候 IEnumerable<T> 是不同 IEnumerable<T1> 的子类型？当 T 是 T1 的子类型时。IEnumerable 在其 T 参数中是协变的，这意味着 IEnumerable 的子类型关系与 T 的方向相同。

```
类 Animal { /* ... */ }
类 Dog : Animal { /* ... */ }
```

```
IEnumerable<Dog> dogs = Enumerable.Empty<Dog>();
IEnumerable<Animal> animals = dogs; // IEnumerable<Dog> 是 IEnumerable<Animal> 的子类型
// dogs = animals; // 编译错误 - IEnumerable<Animal> 不是 IEnumerable<Dog> 的子类型
```

具有给定类型参数的协变泛型类型实例可以隐式转换为具有更基类型参数的相同泛型类型。

这种关系成立是因为 IEnumerable 产生 T，但不消费它们。一个产生 Dog 的对象可以被当作产生 Animal 的对象来使用。

协变类型参数使用 `out` 关键字声明，因为该参数必须仅用作输出。

```
interface IEnumerable<out T> { /* ... */ }
```

声明为协变的类型参数不得作为输入出现。

```
interface Bad<out T>
{
    void SetT(T t); // 类型错误
}
```

这是一个完整的示例：

```
using NUnit.Framework;

namespace ToyStore
{
    enum Taste { Bitter, Sweet };
```

```
}
```

This approach is better than simply calling `Object.Equals()` method, because default comparer implementation checks, whether TBar type implements `IEquatable<TBar>` interface and if yes, calls `IEquatable<TBar>.Equals(TBar other)` method. This allows to avoid boxing/unboxing of value types.

Section 59.7: Reflecting on type parameters

The `typeof` operator works on type parameters.

```
class NameGetter<T>
{
    public string GetTypeName()
    {
        return typeof(T).Name;
    }
}
```

Section 59.8: Covariance

When is an `IEnumerable<T>` a subtype of a different `IEnumerable<T1>`? When T is a subtype of T1. `IEnumerable` is covariant in its T parameter, which means that `IEnumerable`'s subtype relationship goes in the same direction as T's.

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }
```

```
IEnumerable<Dog> dogs = Enumerable.Empty<Dog>();
IEnumerable<Animal> animals = dogs; // IEnumerable<Dog> is a subtype of IEnumerable<Animal>
// dogs = animals; // Compilation error - IEnumerable<Animal> is not a subtype of IEnumerable<Dog>
```

An instance of a covariant generic type with a given type parameter is implicitly convertible to the same generic type with a less derived type parameter.

This relationship holds because `IEnumerable` produces Ts but doesn't consume them. An object that produces Dogs can be used as if it produces Animals.

Covariant type parameters are declared using the `out` keyword, because the parameter must be used only as an output.

```
interface IEnumerable<out T> { /* ... */ }
```

A type parameter declared as covariant may not appear as an input.

```
interface Bad<out T>
{
    void SetT(T t); // type error
}
```

Here's a complete example:

```
using NUnit.Framework;

namespace ToyStore
{
    enum Taste { Bitter, Sweet };
```

```

interface IWidget
{
    int Weight { get; }
}

interface IFactory<out TWidget>
    where TWidget : IWidget
{
    TWidget Create();
}

class Toy : IWidget
{
    public int Weight { get; set; }
    public Taste Taste { get; set; }
}

class ToyFactory : IFactory<Toy>
{
    public const int StandardWeight = 100;
    public const Taste StandardTaste = Taste.Sweet;

    public Toy Create() { return new Toy { Weight = StandardWeight, Taste = StandardTaste }; }
}

[TestFixture]
public class GivenAToyFactory
{
    [Test]
    public static void WhenUsingToyFactoryToMakeWidgets()
    {
        var toyFactory = new ToyFactory();

        // 没有 out 关键字，注意冗长的显式转换：
        // IFactory<IWidget> rustBeltFactory = (IFactory<IWidget>)toyFactory;

        // 协变：具体类型赋值给抽象类型（崭新且闪亮）
        IFactory<IWidget> widgetFactory = toyFactory;
        IWidget anotherToy = widgetFactory.Create();
        Assert.That(anotherToy.Weight, Is.EqualTo(toyFactory.StandardWeight)); // 抽象
    }

    // 契约
    Assert.That(((Toy)anotherToy).Taste, Is.EqualTo(toyFactory.StandardTaste)); // 具体
    // 契约
}
}

```

第59.9节：逆变

当一个IComparer<T>是另一个IComparer<T1>的子类型时？当T1是T的子类型时。IComparer在其T参数中是逆变的，这意味着IComparer的子类型关系与

```

类 Animal { /* ... */ }
类 Dog : Animal { /* ... */ }

IComparer<Animal> animalComparer = /* ... */;
IComparer<Dog> dogComparer = animalComparer; // IComparer<Animal> 是 IComparer<Dog> 的子类型
// animalComparer = dogComparer; // 编译错误 - IComparer<Dog> 不是 IComparer<Animal> 的子类型
IComparer<Animal>

```

```

interface IWidget
{
    int Weight { get; }
}

interface IFactory<out TWidget>
    where TWidget : IWidget
{
    TWidget Create();
}

class Toy : IWidget
{
    public int Weight { get; set; }
    public Taste Taste { get; set; }
}

class ToyFactory : IFactory<Toy>
{
    public const int StandardWeight = 100;
    public const Taste StandardTaste = Taste.Sweet;

    public Toy Create() { return new Toy { Weight = StandardWeight, Taste = StandardTaste }; }
}

[TestFixture]
public class GivenAToyFactory
{
    [Test]
    public static void WhenUsingToyFactoryToMakeWidgets()
    {
        var toyFactory = new ToyFactory();

        // Without out keyword, note the verbose explicit cast:
        // IFactory<IWidget> rustBeltFactory = (IFactory<IWidget>)toyFactory;

        // covariance: concrete being assigned to abstract (shiny and new)
        IFactory<IWidget> widgetFactory = toyFactory;
        IWidget anotherToy = widgetFactory.Create();
        Assert.That(anotherToy.Weight, Is.EqualTo(toyFactory.StandardWeight)); // abstract
    }

    contract
        Assert.That(((Toy)anotherToy).Taste, Is.EqualTo(toyFactory.StandardTaste)); // concrete
    contract
    }
}

```

Section 59.9: Contravariance

When is an IComparer<T> a subtype of a different IComparer<T1>? When T1 is a subtype of T. IComparer is *contravariant* in its T parameter, which means that IComparer's subtype relationship goes in the *opposite direction* as T's.

```

class Animal { /* ... */ }
class Dog : Animal { /* ... */ }

IComparer<Animal> animalComparer = /* ... */;
IComparer<Dog> dogComparer = animalComparer; // IComparer<Animal> is a subtype of IComparer<Dog>
// animalComparer = dogComparer; // Compilation error - IComparer<Dog> is not a subtype of IComparer<Animal>

```

具有给定类型参数的逆变泛型类型的实例可以隐式转换为具有更派生类型参数的相同泛型类型。

这种关系成立是因为 `IComparer` 消费 `T`, 但不产生它们。能够比较任意两个 `Animal` 的对象可以用来比较两个 `Dog`。

逆变类型参数使用 `in` 关键字声明, 因为该参数必须仅用作输入。

```
interface IComparer<in T> { /* ... */ }
```

声明为逆变的类型参数不得作为输出出现。

```
interface Bad<in T>
{
    T GetT(); // 类型错误
}
```

第59.10节：不变性

`IList<T>` 永远不是不同 `IList<T1>` 的子类型。 `IList` 在其类型参数中是不变的。

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }
```

```
IList<Dog> dogs = new List<Dog>();
IList<Animal> animals = dogs; // 类型错误
```

列表之间不存在子类型关系, 因为你既可以向列表中放入值, 也可以从列表中取出值。

如果 `IList` 是协变的, 你就可以向给定列表中添加错误子类型的项。

```
IList<Animal> animals = new List<Dog>(); // 假设这是允许的...
animals.Add(new Giraffe()); // ... 那么这也会被允许, 这是错误的!
```

如果 `IList` 是逆变的, 你就可以从给定列表中提取错误子类型的值。

```
IList<Dog> dogs = new List<Animal> { new Dog(), new Giraffe() }; // 如果这是允许的...
Dog dog = dogs[1]; // ... 那么这将被允许, 这是错误的!
```

不变类型参数通过省略`in`和`out`关键字来声明。

```
接口 IList<T> { /* ... */ }
```

第59.11节：变体接口

接口可以有变体类型参数。

```
接口 IEnumerable<out T>
{
    // ...
}
接口 IComparer<in T>
{
    // ...
}
```

An instance of a contravariant generic type with a given type parameter is implicitly convertible to the same generic type with a more derived type parameter.

This relationship holds because `IComparer` consumes `Ts` but doesn't produce them. An object which can compare any two `Animals` can be used to compare two `Dogs`.

Contravariant type parameters are declared using the `in` keyword, because the parameter must be used only as an *input*.

```
interface IComparer<in T> { /* ... */ }
```

A type parameter declared as contravariant may not appear as an output.

```
interface Bad<in T>
{
    T GetT(); // type error
}
```

Section 59.10: Invariance

`IList<T>` is never a subtype of a different `IList<T1>`. `IList` is *invariant* in its type parameter.

```
class Animal { /* ... */ }
class Dog : Animal { /* ... */ }
```

```
IList<Dog> dogs = new List<Dog>();
IList<Animal> animals = dogs; // type error
```

There is no subtype relationship for lists because you can put values into a list *and* take values out of a list.

If `IList` was covariant, you'd be able to add items of the *wrong subtype* to a given list.

```
IList<Animal> animals = new List<Dog>(); // supposing this were allowed...
animals.Add(new Giraffe()); // ... then this would also be allowed, which is bad!
```

If `IList` was contravariant, you'd be able to extract values of the wrong subtype from a given list.

```
IList<Dog> dogs = new List<Animal> { new Dog(), new Giraffe() }; // if this were allowed...
Dog dog = dogs[1]; // ... then this would be allowed, which is bad!
```

Invariant type parameters are declared by omitting both the `in` and `out` keywords.

```
interface IList<T> { /* ... */ }
```

Section 59.11: Variant interfaces

Interfaces may have variant type parameters.

```
interface IEnumerable<out T>
{
    // ...
}
interface IComparer<in T>
{
    // ...
}
```

```
}
```

但类和结构体不允许

```
类 BadClass<in T1, out T2> // 不允许
{
}
```

```
结构体 BadStruct<in T1, out T2> // 不允许
{
}
```

泛型方法声明也不允许

```
class MyClass
{
    public T Bad<out T, in T1>(T1 t1) // 不允许
    {
        // ...
    }
}
```

下面的示例展示了在同一接口上声明多个变体类型参数

```
interface IFoo<in T1, out T2, T3>
// T1 : 逆变类型
// T2 : 协变类型
// T3 : 不变类型
{
    // ...
}

IFoo<Animal, Dog, int> foo1 = /* ... */;
IFoo<Dog, Animal, int> foo2 = foo1;
// IFoo<Animal, Dog, int> 是 IFoo<Dog, Animal, int> 的子类型
```

第59.12节：变体委托

委托可以具有变体类型参数。

```
delegate void Action<in T>(T t); // T 是输入
delegate T Func<out T>(); // T 是输出
delegate T2 Func<in T1, out T2>(); // T1 是输入, T2 是输出
```

这遵循了里氏替换原则 ([Liskov Substitution Principle](#))，该原则指出（除其他外）如果满足以下条件，则方法D可以被视为比方法B更派生：

- D 的返回类型与 B 相同或更派生
- D 的对应参数类型与 B 相同或更通用

因此，以下赋值都是类型安全的：

```
Func<object, string> original = SomeMethod;
Func<object, object> d1 = original;
Func<string, string> d2 = original;
Func<string, object> d3 = original;
```

```
}
```

but classes and structures may not

```
class BadClass<in T1, out T2> // not allowed
{
}
```

```
struct BadStruct<in T1, out T2> // not allowed
{
}
```

nor do generic method declarations

```
class MyClass
{
    public T Bad<out T, in T1>(T1 t1) // not allowed
    {
        // ...
    }
}
```

The example below shows multiple variance declarations on the same interface

```
interface IFoo<in T1, out T2, T3>
// T1 : Contravariant type
// T2 : Covariant type
// T3 : Invariant type
{
    // ...
}

IFoo<Animal, Dog, int> foo1 = /* ... */;
IFoo<Dog, Animal, int> foo2 = foo1;
// IFoo<Animal, Dog, int> is a subtype of IFoo<Dog, Animal, int>
```

Section 59.12: Variant delegates

Delegates may have variant type parameters.

```
delegate void Action<in T>(T t); // T is an input
delegate T Func<out T>(); // T is an output
delegate T2 Func<in T1, out T2>(); // T1 is an input, T2 is an output
```

This follows from the [Liskov Substitution Principle](#), which states (among other things) that a method D can be considered more derived than a method B if:

- D has an equal or more derived return type than B
- D has equal or more general corresponding parameter types than B

Therefore the following assignments are all type safe:

```
Func<object, string> original = SomeMethod;
Func<object, object> d1 = original;
Func<string, string> d2 = original;
Func<string, object> d3 = original;
```

第59.13节：作为参数和返回值的变体类型

如果协变类型作为输出出现，则包含类型是协变的。生成一个T的生产者就像生成T。

```
接口 IReturnCovariant<out T>
{
    IEnumerable<T> GetTs();
}
```

如果逆变类型作为输出出现，则包含类型是逆变的。生成一个T的消费者就像消费T。

```
接口 IReturnContravariant<in T>
{
    IComparer<T> GetTComparer();
}
```

如果协变类型作为输入出现，则包含类型是逆变的。消费一个T的生产者就像消费T。

```
接口 IAcceptCovariant<in T>
{
    void ProcessTs(IEnumerable<T> ts);
}
```

如果逆变类型作为输入出现，则包含该类型的是协变的。使用一个T的消费者就像是生产T。

```
interface IAcceptContravariant<out T>
{
    void CompareTs(IComparer<T> tComparer);
}
```

第59.14节：类型参数（接口）

声明：

```
interface IMyGenericInterface<T1, T2, T3, ...> { ... }
```

用法（继承中）：

```
class ClassA<T1, T2, T3> : IMyGenericInterface<T1, T2, T3> { ... }

class ClassB<T1, T2> : IMyGenericInterface<T1, T2, int> { ... }

class ClassC<T1> : IMyGenericInterface<T1, char, int> { ... }

class ClassD : IMyGenericInterface<bool, char, int> { ... }
```

用法（作为参数类型）：

```
void SomeMethod(IMyGenericInterface<int, char, bool> 参数) { ... }
```

Section 59.13: Variant types as parameters and return values

If a covariant type appears as an output, the containing type is covariant. Producing a producer of Ts is like producing Ts.

```
interface IReturnCovariant<out T>
{
    IEnumerable<T> GetTs();
}
```

If a contravariant type appears as an output, the containing type is contravariant. Producing a consumer of Ts is like consuming Ts.

```
interface IReturnContravariant<in T>
{
    IComparer<T> GetTComparer();
}
```

If a covariant type appears as an input, the containing type is contravariant. Consuming a producer of Ts is like consuming Ts.

```
interface IAcceptCovariant<in T>
{
    void ProcessTs(IEnumerable<T> ts);
}
```

If a contravariant type appears as an input, the containing type is covariant. Consuming a consumer of Ts is like producing Ts.

```
interface IAcceptContravariant<out T>
{
    void CompareTs(IComparer<T> tComparer);
}
```

Section 59.14: Type Parameters (Interfaces)

Declaration:

```
interface IMyGenericInterface<T1, T2, T3, ...> { ... }
```

Usage (in inheritance):

```
class ClassA<T1, T2, T3> : IMyGenericInterface<T1, T2, T3> { ... }

class ClassB<T1, T2> : IMyGenericInterface<T1, T2, int> { ... }

class ClassC<T1> : IMyGenericInterface<T1, char, int> { ... }

class ClassD : IMyGenericInterface<bool, char, int> { ... }
```

Usage (as the type of a parameter):

```
void SomeMethod(IMyGenericInterface<int, char, bool> arg) { ... }
```

第59.15节：类型约束（类和结构体）

可以通过使用相应的约束class或struct来指定类型参数是否应为引用类型或值类型。如果使用这些约束，必须在所有其他约束（例如父类型或new()）之前定义。

```
// TRef必须是引用类型，使用Int32、Single等无效。  
// 接口是有效的，因为它们是引用类型  
class AcceptsRefType<TRef>  
    where TRef : class  
{  
    // TStruct必须是值类型。  
    public void AcceptStruct<TStruct>()  
        where TStruct : struct  
    {  
    }  
  
    // 如果与class/struct一起使用多个约束  
    // 则class或struct约束必须首先指定  
    public void Foo<TComparableClass>()  
        where TComparableClass : class, IComparable  
    {  
    }  
}
```

Section 59.15: Type constraints (class and struct)

It is possible to specify whether or not the type argument should be a reference type or a value type by using the respective constraints `class` or `struct`. If these constraints are used, they *must* be defined *before all other* constraints (for example a parent type or `new()`) can be listed.

```
// TRef must be a reference type, the use of Int32, Single, etc. is invalid.  
// Interfaces are valid, as they are reference types  
class AcceptsRefType<TRef>  
    where TRef : class  
{  
    // TStruct must be a value type.  
    public void AcceptStruct<TStruct>()  
        where TStruct : struct  
    {  
    }  
  
    // If multiple constraints are used along with class/struct  
    // then the class or struct constraint MUST be specified first  
    public void Foo<TComparableClass>()  
        where TComparableClass : class, IComparable  
    {  
    }  
}
```

第59.16节：显式类型参数

在某些情况下，必须为泛型方法显式指定类型参数。在以下两种情况下，编译器无法从指定的方法参数中推断出所有类型参数。

一种情况是没有参数时：

```
public void SomeMethod<T, V>()  
{  
    // 为简洁起见，无代码  
}  
  
SomeMethod(); // 无法编译  
SomeMethod<int, bool>(); // 可以编译
```

Section 59.16: Explicit type parameters

There are different cases where you must Explicitly specify the type parameters for a generic method. In both of the below cases, the compiler is not able to infer all of the type parameters from the specified method parameters.

One case is when there are no parameters:

```
public void SomeMethod<T, V>()  
{  
    // No code for simplicity  
}  
  
SomeMethod(); // doesn't compile  
SomeMethod<int, bool>(); // compiles
```

第二种情况是当一个（或多个）类型参数不属于方法参数时：

```
public K SomeMethod<K, V>(V input)  
{  
    return default(K);  
}  
  
int num1 = SomeMethod(3); // 无法编译  
int num2 = SomeMethod<int>("3"); // 无法编译  
int num3 = SomeMethod<int, string>("3"); // 可以编译。
```

Second case is when one (or more) of the type parameters is not part of the method parameters:

```
public K SomeMethod<K, V>(V input)  
{  
    return default(K);  
}  
  
int num1 = SomeMethod(3); // doesn't compile  
int num2 = SomeMethod<int>("3"); // doesn't compile  
int num3 = SomeMethod<int, string>("3"); // compiles.
```

第59.17节：类型参数（类）

声明：

```
class MyGenericClass<T1, T2, T3, ...>  
{
```

Section 59.17: Type Parameters (Classes)

Declaration:

```
class MyGenericClass<T1, T2, T3, ...>  
{
```

```
// 对类型参数进行操作。  
}
```

初始化：

```
var x = new MyGenericClass<int, char, bool>();
```

用法（作为参数类型）：

```
void AnotherMethod(MyGenericClass<float, byte, char> arg) { ... }
```

第59.18节：类型参数（方法）

声明：

```
void MyGenericMethod<T1, T2, T3>(T1 a, T2 b, T3 c)  
{  
    // 对类型参数进行操作。  
}
```

调用：

不需要为泛型方法提供类型参数，因为编译器可以隐式推断类型。

```
int x = 10;  
int y = 20;  
string z = "test";  
MyGenericMethod(x, y, z);
```

但是，如果存在歧义，泛型方法需要使用类型参数调用，如

```
MyGenericMethod<int, int, string>(x, y, z);
```

第59.19节：泛型类型转换

```
/// <summary>  
/// 将一种数据类型转换为另一种数据类型。  
/// </summary>  
public static class Cast  
{  
    /// <summary>  
    /// 将输入转换为默认值的类型或指定的类型参数T  
    /// </summary>  
    /// <typeparam name="T">typeparam是返回值的类型，可以是任何类型，例如int、string、bool、decimal等。</t  
ypeparam>/// <param name="input">需要转换为指定类型的输入</pa  
ram>/// <param name="defaultValue">如果值为null或发生任何异常，将返回defaultValue</param>  
  
    /// <returns>输入被转换为默认值的类型或指定的类型参数T并返回</returns>  
  
    public static T To<T>(object input, T defaultValue)  
    {  
        var result = defaultValue;  
        try  
        {  
            if (input == null || input == DBNull.Value) return result;  
            if (typeof(T).IsEnum)
```

```
        // Do something with the type parameters.  
    }
```

Initialisation:

```
var x = new MyGenericClass<int, char, bool>();
```

Usage (as the type of a parameter):

```
void AnotherMethod(MyGenericClass<float, byte, char> arg) { ... }
```

Section 59.18: Type Parameters (Methods)

Declaration:

```
void MyGenericMethod<T1, T2, T3>(T1 a, T2 b, T3 c)  
{  
    // Do something with the type parameters.  
}
```

Invocation:

There is no need to supply type arguments to a generic method, because the compiler can implicitly infer the type.

```
int x = 10;  
int y = 20;  
string z = "test";  
MyGenericMethod(x, y, z);
```

However, if there is an ambiguity, generic methods need to be called with type arguments as

```
MyGenericMethod<int, int, string>(x, y, z);
```

Section 59.19: Generic type casting

```
/// <summary>  
/// Converts a data type to another data type.  
/// </summary>  
public static class Cast  
{  
    /// <summary>  
    /// Converts input to Type of default value or given as typeparam T  
    /// </summary>  
    /// <typeparam name="T">typeparam is the type in which value will be returned, it could be  
any type eg. int, string, bool, decimal etc.</typeparam>  
    /// <param name="input">Input that need to be converted to specified type</param>  
    /// <param name="defaultValue">defaultValue will be returned in case of value is null or any  
exception occurs</param>  
    /// <returns>Input is converted in Type of default value or given as typeparam T and  
returned</returns>  
    public static T To<T>(object input, T defaultValue)  
    {  
        var result = defaultValue;  
        try  
        {  
            if (input == null || input == DBNull.Value) return result;  
            if (typeof(T).IsEnum)
```

```

        {
result = (T) Enum.ToObject(typeof(T), To(input,
Convert.ToInt32(defaultValue)));
    }
    else
    {
result = (T) Convert.ChangeType(input, typeof(T));
    }
}
catch (Exception ex)
{
Tracer.Current.LogException(ex);
}

return result;
}

/// <summary>
/// 将输入转换为类型参数 T 的类型
/// </summary>
/// <typeparam name="T">typeparam 是返回值的类型，可以是任何类型，例如 int、string、bool、decimal 等。
</typeparam>/// <param name="input">需要转换为指定类型的输入</param>/// <returns>输入被转换为默认值的类型或指定的类型参数 T 并返回</returns>

public static T To<T>(object input)
{
    return To(input, default(T));
}
}

```

用法：

```

std.Name = Cast.ToString(drConnection["Name"]);
std.Age = Cast.ToInt(drConnection["Age"]);
std.IsPassed = Cast.ToBoolean(drConnection["IsPassed"]);

// 使用默认值进行类型转换
// 以下两种方式均正确
// 方式1 (以下样式中输入被转换为默认值的类型)
std.Name = Cast.To(drConnection["Name"], "");
std.Marks = Cast.To(drConnection["Marks"], 0);
// 方式2
std.Name = Cast.ToString(drConnection["Name"], "");
std.Marks = Cast.ToInt(drConnection["Marks"], 0);

```

第59.20节：带有泛型类型转换的配置读取器

```

/// <summary>
/// 从app.config读取配置值并转换为指定类型
/// </summary>
public static class ConfigurationReader
{
    /// <summary>
    /// 通过键从AppSettings获取值，转换为默认值的类型或类型参数 T 并返回
    ///

```

```

        {
            result = (T) Enum.ToObject(typeof(T), To(input,
Convert.ToInt32(defaultValue)));
        }
        else
        {
            result = (T) Convert.ChangeType(input, typeof(T));
        }
    }
    catch (Exception ex)
    {
        Tracer.Current.LogException(ex);
    }

    return result;
}

/// <summary>
/// Converts input to Type of typeparam T
/// </summary>
/// <typeparam name="T">typeparam is the type in which value will be returned, it could be
any type eg. int, string, bool, decimal etc.</typeparam>
/// <param name="input">Input that need to be converted to specified type</param>
/// <returns>Input is converted in Type of default value or given as typeparam T and
returned</returns>
public static T To<T>(object input)
{
    return To(input, default(T));
}
}

```

Usages:

```

std.Name = Cast.ToString(drConnection["Name"]);
std.Age = Cast.ToInt(drConnection["Age"]);
std.IsPassed = Cast.ToBoolean(drConnection["IsPassed"]);

// Casting type using default value
// Following both ways are correct
// Way 1 (In following style input is converted into type of default value)
std.Name = Cast.To(drConnection["Name"], "");
std.Marks = Cast.To(drConnection["Marks"], 0);
// Way 2
std.Name = Cast.ToString(drConnection["Name"], "");
std.Marks = Cast.ToInt(drConnection["Marks"], 0);

```

Section 59.20: Configuration reader with generic type casting

```

/// <summary>
/// Read configuration values from app.config and convert to specified types
/// </summary>
public static class ConfigurationReader
{
    /// <summary>
    /// Get value from AppSettings by key, convert to Type of default value or typeparam T and
    return

```

```

/// </summary>
/// <typeparam name="T">typeparam 是返回值的类型，可以是任何类型，例如 int、string、bool、decimal 等。
</typeparam>/// <param name="strKey">用于从 AppSettings 中查
找值的键</param>/// <param name="defaultValue">当值为 null 或发生异常时返回
的默认值</param>

/// <returns>返回与键对应的 AppSettings 值，类型为默认值的类型或指定的 typeparam T</returns>

public static T GetConfigKeyValue<T>(string strKey, T defaultValue)
{
    var result = defaultValue;
    try
    {
        if (ConfigurationManager.AppSettings[strKey] != null)
            result = (T)Convert.ChangeType(ConfigurationManager.AppSettings[strKey],
typeof(T));
    }
    catch (Exception ex)
    {
        Tracer.Current.LogException(ex);
    }

    return result;
}
/// <summary>
/// 通过键从 AppSettings 获取值，转换为默认值的类型或类型参数 T 并返回
/// </summary>
/// <typeparam name="T">typeparam 是返回值的类型，可以是任何类型，例如 int、string、bool、decimal 等。
</typeparam>/// <param name="strKey">用于从 AppSettings 中查
找值的键</param>/// <returns>返回与键对应的 AppSettings 值，类型为指定的 type
param T</returns>
public static T GetConfigKeyValue<T>(string strKey)
{
    return GetConfigKeyValue(strKey, default(T));
}

```

用法：

```

var timeOut = ConfigurationReader.GetConfigKeyValue("RequestTimeout", 2000);
var url = ConfigurationReader.GetConfigKeyValue("URL", "www.someurl.com");
var enabled = ConfigurationReader.GetConfigKeyValue(".IsEnabled", false);

```

```

/// </summary>
/// <typeparam name="T">typeparam is the type in which value will be returned, it could be
any type eg. int, string, bool, decimal etc.</typeparam>
/// <param name="strKey">key to find value from AppSettings</param>
/// <param name="defaultValue">defaultValue will be returned in case of value is null or any
exception occurs</param>
/// <returns>AppSettings value against key is returned in Type of default value or given as
typeparam T</returns>

public static T GetConfigKeyValue<T>(string strKey, T defaultValue)
{
    var result = defaultValue;
    try
    {
        if (ConfigurationManager.AppSettings[strKey] != null)
            result = (T)Convert.ChangeType(ConfigurationManager.AppSettings[strKey],
typeof(T));
    }
    catch (Exception ex)
    {
        Tracer.Current.LogException(ex);
    }

    return result;
}
/// <summary>
/// Get value from AppSettings by key, convert to Type of default value or typeparam T and
return
/// </summary>
/// <typeparam name="T">typeparam is the type in which value will be returned, it could be
any type eg. int, string, bool, decimal etc.</typeparam>
/// <param name="strKey">key to find value from AppSettings</param>
/// <returns>AppSettings value against key is returned in Type given as typeparam T</returns>
public static T GetConfigKeyValue<T>(string strKey)
{
    return GetConfigKeyValue(strKey, default(T));
}

```

Usages:

```

var timeOut = ConfigurationReader.GetConfigKeyValue("RequestTimeout", 2000);
var url = ConfigurationReader.GetConfigKeyValue("URL", "www.someurl.com");
var enabled = ConfigurationReader.GetConfigKeyValue(".IsEnabled", false);

```

第60章：Using语句

提供了一种方便的语法，确保正确使用IDisposable对象。

第60.1节：using语句基础

using 是一种语法糖，允许你保证资源被清理，而无需显式的try-finally块。这意味着你的代码会更简洁，并且不会泄漏非托管资源。

标准的Dispose清理模式，适用于实现了IDisposable接口的对象（例如FileStream的基类Stream在.NET中实现了该接口）：

```
int Foo()
{
    var fileName = "file.txt";

    {
        FileStream disposable = null;

        try
        {
            disposable = File.Open(fileName, FileMode.Open);

            return disposable.ReadByte();
        }
        finally
        {
            // finally 块总是会执行
            if (disposable != null) disposable.Dispose();
        }
    }
}
```

using 简化了你的语法，通过隐藏显式的 try-finally：

```
int Foo()
{
    var fileName = "file.txt";

    using (var disposable = File.Open(fileName, FileMode.Open))
    {
        return disposable.ReadByte();
    }
    // 即使提前返回，也会调用 disposable.Dispose
}
```

就像 finally 块无论是否发生错误或返回都会执行一样，using 总是调用 Dispose()，即使在发生错误的情况下：

```
int Foo()
{
    var fileName = "file.txt";

    using (var disposable = File.Open(fileName, FileMode.Open))
    {
        throw new InvalidOperationException();
    }
    // 即使提前抛出异常，也会调用 disposable.Dispose
}
```

Chapter 60: Using Statement

Provides a convenient syntax that ensures the correct use of [IDisposable](#) objects.

Section 60.1: Using Statement Basics

using is syntactic sugar that allows you to guarantee that a resource is cleaned up without needing an explicit try-finally block. This means your code will be much cleaner, and you won't leak non-managed resources.

Standard Dispose cleanup pattern, for objects that implement the IDisposable interface (which the FileStream's base class Stream does in .NET):

```
int Foo()
{
    var fileName = "file.txt";

    {
        FileStream disposable = null;

        try
        {
            disposable = File.Open(fileName, FileMode.Open);

            return disposable.ReadByte();
        }
        finally
        {
            // finally blocks are always run
            if (disposable != null) disposable.Dispose();
        }
    }
}
```

using simplifies your syntax by hiding the explicit try-finally:

```
int Foo()
{
    var fileName = "file.txt";

    using (var disposable = File.Open(fileName, FileMode.Open))
    {
        return disposable.ReadByte();
    }
    // disposable.Dispose is called even if we return earlier
}
```

Just like finally blocks always execute regardless of errors or returns, using always calls Dispose()，even in the event of an error:

```
int Foo()
{
    var fileName = "file.txt";

    using (var disposable = File.Open(fileName, FileMode.Open))
    {
        throw new InvalidOperationException();
    }
    // disposable.Dispose is called even if we throw an exception earlier
}
```

```
}
```

注意：由于 Dispose 保证无论代码流程如何都会被调用，建议在实现 IDisposable 时确保Dispose 永远不会抛出异常。否则，实际异常会被新异常覆盖，导致调试困难。

从 using 块返回

```
使用 ( var disposable = new DisposableItem() )
{
    return disposable.SomeProperty;
}
```

由于 using 代码块会被转换为 try..finally 语义，return 语句的行为符合预期——返回值会在 finally 块执行并释放资源之前被计算。计算顺序如下：

1. 执行 try 体
2. 计算并缓存返回值
3. 执行 finally 块
4. 返回缓存的返回值

但是，不能直接返回变量 disposable 本身，因为它会包含无效的、已释放的引用——详见相关示例。

第60.2节：注意事项：返回你正在释放的资源

下面的做法是不好的，因为它会在返回之前释放 db 变量。

```
public IDBContext GetDBContext()
{
    使用 (var db = new DBContext())
    {
        return db;
    }
}
```

这也可能导致更微妙的错误：

```
public IEnumerable<Person> GetPeople(int age)
{
    使用 (var db = new DBContext())
    {
        return db.People.Where(p => p.Age == age);
    }
}
```

这看起来没问题，但关键是 LINQ 表达式的求值是延迟的，可能会在底层的DBContext已经被释放后才执行。

简而言之，表达式不会在离开using之前被求值。解决这个问题的一种方法是，仍然使用using，但通过调用一个会枚举结果的方法来立即求值表达式。例如ToList()、ToArray()等。如果你使用的是最新版本的 Entity Framework，可以使用异步版本，如ToListAsync()或ToListAsyncAsync()。

```
}
```

Note: Since Dispose is guaranteed to be called irrespective of the code flow, it's a good idea to make sure that Dispose never throws an exception when you implement IDisposable. Otherwise an actual exception would get overridden by the new exception resulting in a debugging nightmare.

Returning from using block

```
using ( var disposable = new DisposableItem() )
{
    return disposable.SomeProperty;
}
```

Because of the semantics of `try..finally` to which the `using` block translates, the `return` statement works as expected - the return value is evaluated before `finally` block is executed and the value disposed. The order of evaluation is as follows:

1. Evaluate the `try` body
2. Evaluate and cache the returned value
3. Execute finally block
4. Return the cached return value

However, you may not return the variable `disposable` itself, as it would contain invalid, disposed reference - see related example.

Section 60.2: Gotcha: returning the resource which you are disposing

The following is a bad idea because it would dispose the db variable before returning it.

```
public IDBContext GetDBContext()
{
    using (var db = new DBContext())
    {
        return db;
    }
}
```

This can also create more subtle mistakes:

```
public IEnumerable<Person> GetPeople(int age)
{
    using (var db = new DBContext())
    {
        return db.People.Where(p => p.Age == age);
    }
}
```

This looks ok, but the catch is that the LINQ expression evaluation is lazy, and will possibly only be executed later when the underlying DBContext has already been disposed.

So in short the expression isn't evaluated before leaving the `using`. One possible solution to this problem, which still makes use of `using`, is to cause the expression to evaluate immediately by calling a method that will enumerate the result. For example `ToList()`, `ToArray()`, etc. If you are using the newest version of Entity Framework you could use the `async` counterparts like `ToToListAsync()` or `ToToListAsyncAsync()`.

下面是示例代码：

```
public IEnumerable<Person> GetPeople(int age)
{
    使用 (var db = new DbContext())
    {
        return db.Persons.Where(p => p.Age == age).ToList();
    }
}
```

需要注意的是，调用ToList()或ToArray()会使表达式被立即求值，这意味着即使你不遍历结果，所有指定年龄的人员也会被加载到内存中。

第60.3节：多个using语句与一个代码块

可以使用多个嵌套的using语句，而无需增加多层嵌套的大括号。例如：

```
using (var input = File.OpenRead("input.txt"))
{
    using (var output = File.OpenWrite("output.txt"))
    {
        input.CopyTo(output);
    } // 这里释放 output
} // 这里释放 input
```

另一种写法是：

```
using (var input = File.OpenRead("input.txt"))
using (var output = File.OpenWrite("output.txt"))
{
    input.CopyTo(output);
} // 这里先释放 output 然后释放 input
```

这与第一个示例完全等价。

注意：嵌套的using语句可能会触发微软代码分析规则CS2002（详见[this answer](#)以获得说明），并产生警告。如链接回答中所述，通常嵌套using语句是安全的。

当using语句中的类型相同时，可以用逗号分隔它们，并且只需指定一次类型（尽管这种用法不常见）：

```
使用 (FileStream 文件 = File.Open("MyFile.txt"), file2 = File.Open("MyFile2.txt"))
{ }
```

当类型具有共享层次结构时，也可以使用此方法：

```
using (Stream file = File.Open("MyFile.txt"), data = new MemoryStream())
{ }
```

上述示例中不能使用var关键字。会发生编译错误。当声明的变量类型来自不同层次结构时，即使是逗号分隔的声明也无法工作。

Below you find the example in action:

```
public IEnumerable<Person> GetPeople(int age)
{
    using (var db = new DbContext())
    {
        return db.Persons.Where(p => p.Age == age).ToList();
    }
}
```

It is important to note, though, that by calling `ToList()` or `ToArray()`, the expression will be eagerly evaluated, meaning that all the persons with the specified age will be loaded to memory even if you do not iterate on them.

Section 60.3: Multiple using statements with one block

It is possible to use multiple nested `using` statements without added multiple levels of nested braces. For example:

```
using (var input = File.OpenRead("input.txt"))
{
    using (var output = File.OpenWrite("output.txt"))
    {
        input.CopyTo(output);
    } // output is disposed here
} // input is disposed here
```

An alternative is to write:

```
using (var input = File.OpenRead("input.txt"))
using (var output = File.OpenWrite("output.txt"))
{
    input.CopyTo(output);
} // output and then input are disposed here
```

Which is exactly equivalent to the first example.

Note: Nested `using` statements might trigger Microsoft Code Analysis rule [CS2002](#) (see [this answer](#) for clarification) and generate a warning. As explained in the linked answer, it is generally safe to nest `using` statements.

When the types within the `using` statement are of the same type you can comma-delimit them and specify the type only once (though this is uncommon):

```
using (FileStream file = File.Open("MyFile.txt"), file2 = File.Open("MyFile2.txt"))
{ }
```

This can also be used when the types have a shared hierarchy:

```
using (Stream file = File.Open("MyFile.txt"), data = new MemoryStream())
{ }
```

The `var` keyword *cannot* be used in the above example. A compilation error would occur. Even the comma separated declaration won't work when the declared variables have types from different hierarchies.

第60.4节：注意事项：Dispose方法中的异常掩盖Using块中的其他错误

考虑以下代码块。

```
try
{
    using (var disposable = new MyDisposable())
    {
        throw new Exception("无法执行操作。");
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

class MyDisposable : IDisposable
{
    public void Dispose()
    {
        throw new Exception("无法成功释放资源。");
    }
}
```

你可能期望在控制台看到“无法执行操作”的打印，但实际上你会看到“无法成功释放资源。”，因为即使在第一次异常抛出后，Dispose 方法仍然会被调用。

了解这一细微差别是值得的，因为它可能掩盖了阻止对象被释放的真正错误，从而使调试变得更加困难。

第60.5节：using语句是空安全的

你不必检查IDisposable对象是否为null。 using不会抛出异常，且Dispose()不会被调用：

```
DisposableObject TryOpenFile()
{
    return null;
}

// 此处 disposable 为 null, 但不会抛出异常
using (var disposable = TryOpenFile())
{
    // 这将抛出一个 NullReferenceException, 因为 disposable 为 null
    disposable.DoSomething();

    if(disposable != null)
    {
        // 这里是安全的, 因为 disposable 已经被检查是否为 null
        disposable.DoSomething();
    }
}
```

第60.6节：使用 Dispose 语法定义自定义作用域

对于某些用例，您可以使用using语法来帮助定义自定义作用域。例如，您可以定义以下类以在特定文化环境中执行代码。

Section 60.4: Gotcha: Exception in Dispose method masking other errors in Using blocks

Consider the following block of code.

```
try
{
    using (var disposable = new MyDisposable())
    {
        throw new Exception("Couldn't perform operation.");
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}

class MyDisposable : IDisposable
{
    public void Dispose()
    {
        throw new Exception("Couldn't dispose successfully.");
    }
}
```

You may expect to see "Couldn't perform operation" printed to the Console but you would actually see "Couldn't dispose successfully." as the Dispose method is still called even after the first exception is thrown.

It is worth being aware of this subtlety as it may be masking the real error that prevented the object from being disposed and make it harder to debug.

Section 60.5: Using statements are null-safe

You don't have to check the IDisposable object for null. using will not throw an exception and Dispose() will not be called:

```
DisposableObject TryOpenFile()
{
    return null;
}

// disposable is null here, but this does not throw an exception
using (var disposable = TryOpenFile())
{
    // this will throw a NullReferenceException because disposable is null
    disposable.DoSomething();

    if(disposable != null)
    {
        // here we are safe because disposable has been checked for null
        disposable.DoSomething();
    }
}
```

Section 60.6: Using Dispose Syntax to define custom scope

For some use cases, you can use the using syntax to help define a custom scope. For example, you can define the following class to execute code in a specific culture.

```

public class CultureContext : IDisposable
{
    private readonly CultureInfo originalCulture;

    public CultureContext(string culture)
    {
        originalCulture = CultureInfo.CurrentCulture;
        Thread.CurrentThread.CurrentCulture = new CultureInfo(culture);
    }

    public void Dispose()
    {
        Thread.CurrentThread.CurrentCulture = originalCulture;
    }
}

```

然后，您可以使用此类来定义在特定文化环境下执行的代码块。

```

Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");

using (new CultureContext("nl-NL"))
{
    // 此代码块中的代码使用"nl-NL"文化
    Console.WriteLine(new DateTime(2016, 12, 25)); // 输出: 25-12-2016 00:00:00
}

using (new CultureContext("es-ES"))
{
    // 此代码块中的代码使用"es-ES"文化
    Console.WriteLine(new DateTime(2016, 12, 25)); // 输出: 25/12/2016 0:00:00
}

// 已恢复到原始文化
Console.WriteLine(new DateTime(2016, 12, 25)); // 输出: 12/25/2016 12:00:00 AM

```

注意：由于我们没有使用创建的CultureContext实例，因此没有为其分配变量。

该技术由ASP.NET MVC中的BeginForm 助手使用。

第60.7节：使用语句和数据库连接

using关键字确保语句中定义的资源仅在语句本身的作用域内存在。语句中定义的任何资源必须实现IDisposable接口。

当处理实现了IDisposable接口的连接时，这一点非常重要，因为它不仅能确保连接被正确关闭，还能确保在using语句超出作用域后释放其资源。

常见的IDisposable数据类

下面许多是实现了IDisposable接口的数据相关类，非常适合用于using语句：

- SqlConnection、SqlCommand、SqlDataReader等。
- OleDbConnection、OleDbCommand、OleDbDataReader等。
- MySqlConnection、MySqlCommand、MySqlDbDataReader等。
- DbContext

```

public class CultureContext : IDisposable
{
    private readonly CultureInfo originalCulture;

    public CultureContext(string culture)
    {
        originalCulture = CultureInfo.CurrentCulture;
        Thread.CurrentThread.CurrentCulture = new CultureInfo(culture);
    }

    public void Dispose()
    {
        Thread.CurrentThread.CurrentCulture = originalCulture;
    }
}

```

You can then use this class to define blocks of code that execute in a specific culture.

```

Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");

using (new CultureContext("nl-NL"))
{
    // Code in this block uses the "nl-NL" culture
    Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 25-12-2016 00:00:00
}

using (new CultureContext("es-ES"))
{
    // Code in this block uses the "es-ES" culture
    Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 25/12/2016 0:00:00
}

// Reverted back to the original culture
Console.WriteLine(new DateTime(2016, 12, 25)); // Output: 12/25/2016 12:00:00 AM

```

Note: as we don't use the CultureContext instance we create, we don't assign a variable for it.

This technique is used by the BeginForm [helper](#) in ASP.NET MVC.

Section 60.7: Using Statements and Database Connections

The **using** keyword ensures that the resource defined within the statement only exists within the scope of the statement itself. Any resources defined within the statement must implement the **IDisposable** interface.

These are incredibly important when dealing with any connections that implement the **IDisposable** interface as it can ensure the connections are not only properly closed but that their resources are freed after the **using** statement is out of scope.

Common IDisposable Data Classes

Many of the following are data-related classes that implement the **IDisposable** interface and are perfect candidates for a **using** statement :

- SqlConnection, SqlCommand, SqlDataReader, etc.
- OleDbConnection, OleDbCommand, OleDbDataReader, etc.
- MySqlConnection, MySqlCommand, MySqlDbDataReader, etc.
- DbContext

所有这些都是通过C#访问数据时常用的，并且在构建以数据为中心的应用程序时会经常遇到。许多其他未提及但实现相同接口的类 FooConnection,FooCommand,FooDataReader 类也可以预期具有相同的行为。

ADO.NET连接的常见访问模式

通过ADO.NET连接访问数据时可以使用的常见模式可能如下所示：

```
// 这限定了连接的作用域 (您的具体类可能有所不同)
using(var connection = new SqlConnection("{your-connection-string}"))
{
    // 构建查询
    var query = "SELECT * FROM YourTable WHERE Property = @property";
    // 限定执行命令的作用域
    using(var command = new SqlCommand(query, connection))
    {
        // 打开您的连接
        connection.Open();

        // 如有必要, 在此添加参数

        // 以读取器方式执行查询 (同样使用 using 语句块)
        using(var reader = command.ExecuteReader())
        {
            // 在此遍历查询结果
        }
    }
}
```

或者如果只是执行简单的更新且不需要读取器，基本思路同样适用：

```
using(var connection = new SqlConnection("{your-connection-string}"))
{
    var query = "UPDATE YourTable SET Property = Value WHERE Foo = @foo";
    using(var command = new SqlCommand(query, connection))
    {
        connection.Open();

        // 在此添加参数

        // 执行更新操作
        command.ExecuteNonQuery();
    }
}
```

使用带有数据上下文的using语句

许多ORM（如实体框架）提供抽象类，用于以类似DbContext的形式与底层数据库交互。这些上下文通常也实现了IDisposable接口，且应尽可能通过using语句来利用这一点：

```
using(var context = new YourDbContext())
{
    // 访问你的上下文并执行查询
    var data = context.Widgets.ToList();
}
```

All of these are commonly used to access data through C# and will be commonly encountered throughout building data-centric applications. Many other classes that are not mentioned that implement the same FooConnection,FooCommand,FooDataReader classes can be expected to behave the same way.

Common Access Pattern for ADO.NET Connections

A common pattern that can be used when accessing your data through an ADO.NET connection might look as follows :

```
// This scopes the connection (your specific class may vary)
using(var connection = new SqlConnection("{your-connection-string}"))
{
    // Build your query
    var query = "SELECT * FROM YourTable WHERE Property = @property";
    // Scope your command to execute
    using(var command = new SqlCommand(query, connection))
    {
        // Open your connection
        connection.Open();

        // Add your parameters here if necessary

        // Execute your query as a reader (again scoped with a using statement)
        using(var reader = command.ExecuteReader())
        {
            // Iterate through your results here
        }
    }
}
```

Or if you were just performing a simple update and didn't require a reader, the same basic concept would apply :

```
using(var connection = new SqlConnection("{your-connection-string}"))
{
    var query = "UPDATE YourTable SET Property = Value WHERE Foo = @foo";
    using(var command = new SqlCommand(query, connection))
    {
        connection.Open();

        // Add parameters here

        // Perform your update
        command.ExecuteNonQuery();
    }
}
```

Using Statements with DataContexts

Many ORMs such as Entity Framework expose abstraction classes that are used to interact with underlying databases in the form of classes like DbContext. These contexts generally implement the IDisposable interface as well and should take advantage of this through using statements when possible :

```
using(var context = new YourDbContext())
{
    // Access your context and perform your query
    var data = context.Widgets.ToList();
}
```

第60.8节：在约束上下文中执行代码

如果你有代码（一个routine）想在特定的（约束）上下文中执行，可以使用依赖注入。

下面的示例展示了在开放SSL连接下执行的约束。第一部分位于你的库或框架中，不会暴露给客户端代码。

```
public static class SSLContext
{
    // 定义要注入的委托
    public delegate void TunnelRoutine(BinaryReader sslReader, BinaryWriter sslWriter);

    // 这允许该例程在 SSL 下执行
    public static void ClientTunnel(TcpClient tcpClient, TunnelRoutine routine)
    {
        using (SslStream sslStream = new SslStream(tcpClient.GetStream(), true, _validate))
        {
            sslStream.AuthenticateAsClient(HOSTNAME, null, SslProtocols.Tls, false);

            if (!sslStream.IsAuthenticated)
            {
                throw new SecurityException("SSL 隧道未认证");
            }

            if (!sslStream.IsEncrypted)
            {
                throw new SecurityException("SSL 隧道未加密");
            }

            using (BinaryReader sslReader = new BinaryReader(sslStream))
            using (BinaryWriter sslWriter = new BinaryWriter(sslStream))
            {
                routine(sslReader, sslWriter);
            }
        }
    }
}
```

现在客户端代码想在SSL下执行某些操作，但不想处理所有SSL细节。你现在可以在SSL隧道内做任何你想做的事情，例如交换对称密钥：

```
public void ExchangeSymmetricKey(BinaryReader sslReader, BinaryWriter sslWriter)
{
    byte[] bytes = new byte[8];
    (new RNGCryptoServiceProvider()).GetNonZeroBytes(bytes);
    sslWriter.Write(BitConverter.ToInt64(bytes, 0));
}
```

你可以如下执行此例程：

```
SSLContext.ClientTunnel(tcpClient, this.ExchangeSymmetricKey);
```

为此，你需要使用using()语句，因为这是唯一一种方式（除了try..finally块之外）可以保证客户端代码（ExchangeSymmetricKey）在退出时正确释放可释放资源。没有using()语句，你永远无法确定某个例程是否会破坏上下文对这些资源释放的约束。

Section 60.8: Executing code in constraint context

If you have code (*a routine*) you want to execute under a specific (constraint) context, you can use dependency injection.

The following example shows the constraint of executing under an open SSL connection. This first part would be in your library or framework, which you won't expose to the client code.

```
public static class SSLContext
{
    // define the delegate to inject
    public delegate void TunnelRoutine(BinaryReader sslReader, BinaryWriter sslWriter);

    // this allows the routine to be executed under SSL
    public static void ClientTunnel(TcpClient tcpClient, TunnelRoutine routine)
    {
        using (SslStream sslStream = new SslStream(tcpClient.GetStream(), true, _validate))
        {
            sslStream.AuthenticateAsClient(HOSTNAME, null, SslProtocols.Tls, false);

            if (!sslStream.IsAuthenticated)
            {
                throw new SecurityException("SSL tunnel not authenticated");
            }

            if (!sslStream.IsEncrypted)
            {
                throw new SecurityException("SSL tunnel not encrypted");
            }

            using (BinaryReader sslReader = new BinaryReader(sslStream))
            using (BinaryWriter sslWriter = new BinaryWriter(sslStream))
            {
                routine(sslReader, sslWriter);
            }
        }
    }
}
```

Now the client code which wants to do something under SSL but does not want to handle all the SSL details. You can now do whatever you want inside the SSL tunnel, for example exchange a symmetric key:

```
public void ExchangeSymmetricKey(BinaryReader sslReader, BinaryWriter sslWriter)
{
    byte[] bytes = new byte[8];
    (new RNGCryptoServiceProvider()).GetNonZeroBytes(bytes);
    sslWriter.Write(BitConverter.ToInt64(bytes, 0));
}
```

You execute this routine as follows:

```
SSLContext.ClientTunnel(tcpClient, this.ExchangeSymmetricKey);
```

To do this, you need the `using()` clause because it is the only way (apart from a `try..finally` block) you can guarantee the client code (`ExchangeSymmetricKey`) never exits without properly disposing of the disposable resources. Without `using()` clause, you would never know if a routine could break the context's constraint to dispose of those resources.

第61章：Using指令

第61.1节：关联别名以解决冲突

如果你使用多个可能包含同名类的命名空间（例如System.Random和UnityEngine.Random），你可以使用别名来指定Random来自哪个命名空间，而无需在调用时使用完整的命名空间。

例如：

```
using UnityEngine;
using System;

Random rnd = new Random();
```

这将导致编译器无法确定应将新的变量视为哪个Random。相反，你可以这样做：

```
using UnityEngine;
using System;
using Random = System.Random;

Random rnd = new Random();
```

这并不妨碍你通过其完整限定的命名空间来调用另一个，例如：

```
using UnityEngine;
using System;
using Random = System.Random;

Random rnd = new Random();
int unityRandom = UnityEngine.Random.Range(0, 100);
```

rnd 将是一个 System.Random 类型的变量，而 unityRandom 将是一个 UnityEngine.Random 类型的变量。

第61.2节：使用别名指令

你可以使用 using 来为命名空间或类型设置别名。更多细节可以在 [here](#) 找到。

语法：

```
using <identifier> = <namespace-or-type-name>;
```

示例：

```
using NewType = Dictionary<string, Dictionary<string, int>>;
NewType multiDictionary = new NewType();
//使用实例就像使用原始类型一样
multiDictionary.Add("test", new Dictionary<string,int>());
```

第61.3节：访问类的静态成员

版本 ≥ 6.0

允许你导入特定类型并使用该类型的静态成员，而无需使用类型名进行限定。
下面展示了使用静态方法的示例：

Chapter 61: Using Directive

Section 61.1: Associate an Alias to Resolve Conflicts

If you are using multiple namespaces that may have same-name classes(such as System.Random and UnityEngine.Random), you can use an alias to specify that Random comes from one or the other without having to use the entire namespace in the call.

For instance:

```
using UnityEngine;
using System;

Random rnd = new Random();
```

This will cause the compiler to be unsure which Random to evaluate the new variable as. Instead, you can do:

```
using UnityEngine;
using System;
using Random = System.Random;

Random rnd = new Random();
```

This doesn't preclude you from calling the other by its fully qualified namespace, like this:

```
using UnityEngine;
using System;
using Random = System.Random;

Random rnd = new Random();
int unityRandom = UnityEngine.Random.Range(0, 100);
```

rnd will be a System.Random variable and unityRandom will be a UnityEngine.Random variable.

Section 61.2: Using alias directives

You can use using in order to set an alias for a namespace or type. More detail can be found in [here](#).

Syntax:

```
using <identifier> = <namespace-or-type-name>;
```

Example:

```
using NewType = Dictionary<string, Dictionary<string, int>>;
NewType multiDictionary = new NewType();
//Use instances as you are using the original one
multiDictionary.Add("test", new Dictionary<string,int>());
```

Section 61.3: Access Static Members of a Class

Version ≥ 6.0

Allows you to import a specific type and use the type's static members without qualifying them with the type name.
This shows an example using static methods:

```
using static System.Console;
// ...
string GetName()
{
    WriteLine("请输入你的名字。");
    return ReadLine();
}
```

这展示了一个使用静态属性和方法的示例：

```
using static System.Math;
namespace 几何
{
    public class 圆
    {
        public double 半径 { get; set; }

        public double 面积 => PI * Pow(半径, 2);
    }
}
```

第61.4节：基本用法

```
using System;
using BasicStuff = System;
using Sayer = System.Console;
using static System.Console; //从C# 6开始

class 程序
{
    public static void Main()
    {
        System.Console.WriteLine("忽略using并指定完整类型名");
        Console.WriteLine("感谢 'using System' 指令");
        BasicStuff.Console.WriteLine("命名空间别名");
        Sayer.WriteLine("类型别名");
        WriteLine("感谢 'using static' 指令 (来自C# 6)");
    }
}
```

第61.5节：引用命名空间

```
using System.Text;
//允许你访问该命名空间内的类，如StringBuilder
//而无需在前面加上命名空间前缀。例如：

//...
var sb = new StringBuilder();
//代替
var sb = new System.Text.StringBuilder();
```

第61.6节：为命名空间关联别名

```
using st = System.Text;
//允许你访问该命名空间内的类，如StringBuilder
```

```
using static System.Console;
// ...
string GetName()
{
    WriteLine("Enter your name.");
    return ReadLine();
}
```

And this shows an example using static properties and methods:

```
using static System.Math;
namespace Geometry
{
    public class Circle
    {
        public double Radius { get; set; }

        public double Area => PI * Pow(Radius, 2);
    }
}
```

Section 61.4: Basic Usage

```
using System;
using BasicStuff = System;
using Sayer = System.Console;
using static System.Console; //From C# 6

class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Ignoring usings and specifying full type name");
        Console.WriteLine("Thanks to the 'using System' directive");
        BasicStuff.Console.WriteLine("Namespace aliasing");
        Sayer.WriteLine("Type aliasing");
        WriteLine("Thanks to the 'using static' directive (from C# 6)");
    }
}
```

Section 61.5: Reference a Namespace

```
using System.Text;
//allows you to access classes within this namespace such as StringBuilder
//without prefixing them with the namespace. i.e:

//...
var sb = new StringBuilder();
//instead of
var sb = new System.Text.StringBuilder();
```

Section 61.6: Associate an Alias with a Namespace

```
using st = System.Text;
//allows you to access classes within this namespace such as StringBuilder
```

//只需使用定义的别名前缀，而非完整命名空间。例如：

```
//...
var sb = new st.StringBuilder();
//代替
var sb = new System.Text.StringBuilder();
```

//prefixing them with only the defined alias and not the full namespace. i.e:

```
//...
var sb = new st.StringBuilder();
//instead of
var sb = new System.Text.StringBuilder();
```

第62章：IDisposable接口

第62.1节：在仅包含托管资源的类中

托管资源是运行时的垃圾回收器已知并控制的资源。例如，BCL 中有许多可用的类，如 SqlConnection，它是非托管资源的包装类。这些类已经实现了 IDisposable 接口——由你的代码在使用完成后负责清理它们。

如果你的类只包含托管资源，则不必实现终结器。

```
public class 仅含托管资源的对象 : IDisposable
{
    private SqlConnection sqlConnection = new SqlConnection();

    public void Dispose()
    {
        sqlConnection.Dispose();
    }
}
```

第62.2节：包含托管和非托管资源的类

让终结器忽略托管资源非常重要。终结器在另一个线程上运行——当终结器运行时，托管对象可能已经不存在了。实现一个受保护的 Dispose(bool) 方法是常见做法，以确保托管资源不会从终结器中调用它们的 Dispose 方法。

```
public class 托管和非托管对象 : IDisposable
{
    private SqlConnection sqlConnection = new SqlConnection();
    private UnmanagedHandle unmanagedHandle = Win32.SomeUnmanagedResource();
    private bool disposed;

    public void Dispose()
    {
        Dispose(true); // 客户端调用了 dispose
        GC.SuppressFinalize(this); // 告诉垃圾回收器不要执行终结器
    }

    受保护的虚方法 Dispose(bool disposeManaged)
    {
        if (!disposed)
        {
            if (disposeManaged)
            {
                if (sqlConnection != null)
                {
                    sqlConnection.Dispose();
                }
            }
        }
    }

    unmanagedHandle.Release();

    disposed = true;
}
```

Chapter 62: IDisposable interface

Section 62.1: In a class that contains only managed resources

Managed resources are resources that the runtime's garbage collector is aware and under control of. There are many classes available in the BCL, for example, such as a SqlConnection that is a wrapper class for an unmanaged resource. These classes already implement the IDisposable interface -- it's up to your code to clean them up when you are done.

It's not necessary to implement a finalizer if your class only contains managed resources.

```
public class ObjectWithManagedResourcesOnly : IDisposable
{
    private SqlConnection sqlConnection = new SqlConnection();

    public void Dispose()
    {
        sqlConnection.Dispose();
    }
}
```

Section 62.2: In a class with managed and unmanaged resources

It's important to let finalization ignore managed resources. The finalizer runs on another thread -- it's possible that the managed objects don't exist anymore by the time the finalizer runs. Implementing a protected Dispose(bool) method is a common practice to ensure managed resources do not have their Dispose method called from a finalizer.

```
public class ManagedAndUnmanagedObject : IDisposable
{
    private SqlConnection sqlConnection = new SqlConnection();
    private UnmanagedHandle unmanagedHandle = Win32.SomeUnmanagedResource();
    private bool disposed;

    public void Dispose()
    {
        Dispose(true); // client called dispose
        GC.SuppressFinalize(this); // tell the GC to not execute the Finalizer
    }

    protected virtual void Dispose(bool disposeManaged)
    {
        if (!disposed)
        {
            if (disposeManaged)
            {
                if (sqlConnection != null)
                {
                    sqlConnection.Dispose();
                }
            }
        }
    }

    unmanagedHandle.Release();

    disposed = true;
}
```

```
}

~ManagedAndUnmanagedObject()
{
    Dispose(false);
}
}
```

第62.3节：IDisposable, Dispose

.NET Framework 定义了一个需要清理方法的类型接口：

```
public interface IDisposable
{
    void Dispose();
}
```

Dispose() 主要用于清理资源，如非托管引用。然而，它也可以用于强制释放其他托管资源。与其等待垃圾回收器最终清理数据库连接，不如确保在你自己的Dispose()实现中完成清理。

```
public void Dispose()
{
    if (null != this.CurrentDatabaseConnection)
    {
        this.CurrentDatabaseConnection.Dispose();
        this.CurrentDatabaseConnection = null;
    }
}
```

当你需要直接访问非托管资源，如非托管指针或 Win32 资源时，创建一个继承自SafeHandle的类，并使用该类的约定/工具来操作。

第62.4节：using 关键字

当一个对象实现了IDisposable接口时，可以在using语法中创建该对象：

```
using (var foo = new Foo())
{
    // 执行 foo 相关操作
} // 当执行到这里时，会调用 foo.Dispose()

public class Foo : IDisposable
{
    public void Dispose()
    {
        Console.WriteLine("dispose called");
    }
}
```

[查看演示](#)

using 是 try/finally 代码块的语法糖；上述用法大致可翻译为：

```
{
    var foo = new Foo();
    try
```

```
}

~ManagedAndUnmanagedObject()
{
    Dispose(false);
}
}
```

Section 62.3: IDisposable, Dispose

.NET Framework defines a interface for types requiring a tear-down method:

```
public interface IDisposable
{
    void Dispose();
}
```

Dispose() is primarily used for cleaning up resources, like unmanaged references. However, it can also be useful to force the disposing of other resources even though they are managed. Instead of waiting for the GC to eventually also clean up your database connection, you can make sure it's done in your own Dispose() implementation.

```
public void Dispose()
{
    if (null != this.CurrentDatabaseConnection)
    {
        this.CurrentDatabaseConnection.Dispose();
        this.CurrentDatabaseConnection = null;
    }
}
```

When you need to directly access unmanaged resources such as unmanaged pointers or win32 resources, create a class inheriting from SafeHandle and use that class's conventions/tools to do so.

Section 62.4: using keyword

When an object implements the IDisposable interface, it can be created within the using syntax:

```
using (var foo = new Foo())
{
    // do foo stuff
} // when it reaches here foo.Dispose() will get called

public class Foo : IDisposable
{
    public void Dispose()
    {
        Console.WriteLine("dispose called");
    }
}
```

[View demo](#)

using is syntactic sugar for a try/finally block; the above usage would roughly translate into:

```
{
    var foo = new Foo();
    try
```

```

    {
        // 执行 foo 相关操作
    }
    finally
    {
        if (foo != null)
            ((IDisposable)foo).Dispose();
    }
}

```

第62.5节：在具有托管资源的继承类中

你可能经常会创建一个实现了 IDisposable 的类，然后派生出也包含托管资源的类。建议将 Dispose 方法标记为 virtual，以便客户端能够清理它们可能拥有的任何资源。

```

public class Parent : IDisposable
{
    private ManagedResource parentManagedResource = new ManagedResource();

    public virtual void Dispose()
    {
        if (parentManagedResource != null)
        {
            parentManagedResource.Dispose();
        }
    }
}

public class Child : Parent
{
    private ManagedResource childManagedResource = new ManagedResource();

    public override void Dispose()
    {
        if (childManagedResource != null)
        {
            childManagedResource.Dispose();
        }
        //清理父资源
        base.Dispose();
    }
}

```

```

    {
        // do foo stuff
    }
    finally
    {
        if (foo != null)
            ((IDisposable)foo).Dispose();
    }
}

```

Section 62.5: In an inherited class with managed resources

It's fairly common that you may create a class that implements IDisposable, and then derive classes that also contain managed resources. It is recommended to mark the Dispose method with the `virtual` keyword so that clients have the ability to cleanup any resources they may own.

```

public class Parent : IDisposable
{
    private ManagedResource parentManagedResource = new ManagedResource();

    public virtual void Dispose()
    {
        if (parentManagedResource != null)
        {
            parentManagedResource.Dispose();
        }
    }
}

public class Child : Parent
{
    private ManagedResource childManagedResource = new ManagedResource();

    public override void Dispose()
    {
        if (childManagedResource != null)
        {
            childManagedResource.Dispose();
        }
        //clean up the parent's resources
        base.Dispose();
    }
}

```

第63章：反射

反射是C#语言在运行时访问动态对象属性的机制。通常，反射用于获取动态对象类型和对象属性值的信息。例如，在REST应用中，反射可以用来遍历序列化的响应对象。

备注：根据微软指南，性能关键代码应避免使用反射。详见
<https://msdn.microsoft.com/en-us/library/ff647790.aspx>

第63.1节：获取某个类型的成员

```
using System;
using System.Reflection;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var members = typeof(object)
            .GetMembers(BindingFlags.Public |
                BindingFlags.Static |
                BindingFlags.Instance);

        foreach (var member in members)
        {
            bool inherited = member.DeclaringType.Equals( typeof(object).Name );
            Console.WriteLine($"{member.Name} 是一个 {member.MemberType}, " +
                $"它{(inherited ? ":"not")} 被继承。");
        }
    }
}
```

输出（关于输出顺序的说明见下文）：

GetType 是一个方法，它没有被继承。 GetHashCode 是一个方法，它没有被继承。 ToString 是一个方法，它没有被继承。 Equals 是一个方法，它没有被继承。 Equals 是一个方法，它没有被继承。 ReferenceEquals 是一个方法，它没有被继承。 .ctor 是一个构造函数，它没有被继承。

我们也可以使用 GetMembers()，而不传入任何 BindingFlags。这将返回该

特定类型的所有公共成员。

需要注意的是，GetMembers不会以任何特定顺序返回成员，因此绝不要依赖GetMembers返回的顺序。

[查看演示](#)

第63.2节：获取方法并调用它

获取实例方法并调用它

```
using System;

public class Program
{
    public static void Main()
```

Chapter 63: Reflection

Reflection is a C# language mechanism for accessing dynamic object properties on runtime. Typically, reflection is used to fetch the information about dynamic object type and object attribute values. In REST application, for example, reflection could be used to iterate through serialized response object.

Remark: According to MS guidelines performance critical code should avoid reflection. See
<https://msdn.microsoft.com/en-us/library/ff647790.aspx>

Section 63.1: Get the members of a type

```
using System;
using System.Reflection;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var members = typeof(object)
            .GetMembers(BindingFlags.Public |
                BindingFlags.Static |
                BindingFlags.Instance);

        foreach (var member in members)
        {
            bool inherited = member.DeclaringType.Equals( typeof(object).Name );
            Console.WriteLine($"{member.Name} is a {member.MemberType}, " +
                $"it has {(inherited ? ":"not")} been inherited.");
        }
    }
}
```

Output (see note about output order further down):

GetType is a Method, it has not been inherited. GetHashCode is a Method, it has not been inherited. ToString is a Method, it has not been inherited. Equals is a Method, it has not been inherited. Equals is a Method, it has not been inherited. ReferenceEquals is a Method, it has not been inherited. .ctor is a Constructor, it has not been inherited.

We can also use the GetMembers() without passing any BindingFlags. This will return all public members of that specific type.

One thing to note that GetMembers does not return the members in any particular order, so never rely on the order that GetMembers returns you.

[View Demo](#)

Section 63.2: Get a method and invoke it

Get Instance method and invoke it

```
using System;

public class Program
{
    public static void Main()
```

```

{
    var theString = "hello";
    var method = theString
        .GetType()
        .GetMethod("Substring",
            new[] {typeof(int), typeof(int)}); //方法参数的类型

    var result = method.Invoke(theString, new object[] {0, 4});
    Console.WriteLine(result);
}

```

输出：

hell

[查看演示](#)

获取静态方法并调用它

另一方面，如果方法是静态的，则不需要实例即可调用它。

```

var method = typeof(Math).GetMethod("Exp");
var result = method.Invoke(null, new object[] {2}); //第一个参数传入 null (不需要实例)

Console.WriteLine(result); //你将得到 e^2

```

输出：

7.38905609893065

[查看演示](#)

第63.3节：创建类型的实例

最简单的方法是使用Activator类。

然而，尽管自 .NET 3.5 以来Activator的性能有所提升，使用 Activator.CreateInstance() 有时仍然不是好的选择，因为性能（相对）较低：[测试1](#), [测试2](#), [测试3...](#)

使用Activator类

```

Type type = typeof(BigInteger);
object result = Activator.CreateInstance(type); //需要无参构造函数。
Console.WriteLine(result); //输出: 0
result = Activator.CreateInstance(type, 123); //需要一个可以接收'int'类型参数的构造函数。

Console.WriteLine(result); //输出: 123

```

如果有多个参数，可以将对象数组传递给Activator.CreateInstance。

```

// 使用类似 MyClass(int, int, string) 的构造函数
Activator.CreateInstance(typeof(MyClass), new object[] {1, 2, "Hello World"});

```

```

{
    var theString = "hello";
    var method = theString
        .GetType()
        .GetMethod("Substring",
            new[] {typeof(int), typeof(int)}); //The types of the method arguments

    var result = method.Invoke(theString, new object[] {0, 4});
    Console.WriteLine(result);
}

```

Output:

hell

[View Demo](#)

Get Static method and invoke it

On the other hand, if the method is static, you do not need an instance to call it.

```

var method = typeof(Math).GetMethod("Exp");
var result = method.Invoke(null, new object[] {2}); //Pass null as the first argument (no need for an instance)
Console.WriteLine(result); //You'll get e^2

```

Output:

7.38905609893065

[View Demo](#)

Section 63.3: Creating an instance of a Type

The simplest way is to use the Activator class.

However, even though Activator performance have been improved since .NET 3.5, using Activator.CreateInstance() is bad option sometimes, due to (relatively) low performance: [Test 1](#), [Test 2](#), [Test 3...](#)

With Activator class

```

Type type = typeof(BigInteger);
object result = Activator.CreateInstance(type); //Requires parameterless constructor.
Console.WriteLine(result); //Output: 0
result = Activator.CreateInstance(type, 123); //Requires a constructor which can receive an 'int' compatible argument.
Console.WriteLine(result); //Output: 123

```

You can pass an object array to Activator.CreateInstance if you have more than one parameter.

```

// With a constructor such as MyClass(int, int, string)
Activator.CreateInstance(typeof(MyClass), new object[] {1, 2, "Hello World"});

```

```
Type type = typeof(someObject);
var instance = Activator.CreateInstance(type);
```

对于泛型类型

MakeGenericType 方法通过应用类型参数，将一个开放的泛型类型（如 List<>）转换为具体类型（如 List<string>）。

```
// 无参数的泛型 List
类型 openType = typeof(List<>);

// 创建一个 List<string>
类型[] tArgs = { typeof(string) };
类型 target = openType.MakeGenericType(tArgs);

// 创建实例 - Activator.CreateInstance 会调用默认构造函数。
// 这相当于调用 new List<string>().
List<string> result = (List<string>)Activator.CreateInstance(target);
```

List<> 语法不允许出现在 typeof 表达式之外。

不使用 Activator 类

使用 new 关键字（适用于无参构造函数）

```
T GetInstance<T>() where T : new()
{
    T 实例 = new T();
    return 实例;
}
```

使用 Invoke 方法

```
// 获取所需构造函数的实例（这里构造函数以字符串作为参数）。
ConstructorInfo c = typeof(T).GetConstructor(new[] { typeof(string) });
// 别忘了检查该构造函数是否存在
if (c == null)
    throw new InvalidOperationException(string.Format("未找到类型 '{0}' 的构造函数。", typeof(T)));

T instance = (T)c.Invoke(new object[] { "test" });
```

使用表达式树

表达式树以树状数据结构表示代码，其中每个节点都是一个表达式。正如 MSDN 所解释的：

表达式是一串一个或多个操作数和零个或多个运算符，可以被计算为单一的值、对象、方法或命名空间。表达式可以由字面值、方法调用、运算符及其操作数，或简单名称组成。简单名称可以是变量名、类型成员、方法参数、命名空间或类型的名称。

```
public class GenericFactory< TKey, TType >
{
    private readonly Dictionary< TKey, Func< object[], TType > > _registeredTypes; // 字典,
    保存构造函数。
    private object _locker = new object(); // 用于锁定字典的对象，保证线程
```

```
Type type = typeof(someObject);
var instance = Activator.CreateInstance(type);
```

For a generic type

The MakeGenericType method turns an open generic type (like List<>) into a concrete type (like List<string>) by applying type arguments to it.

```
// generic List with no parameters
Type openType = typeof(List<>);

// To create a List<string>
Type[] tArgs = { typeof(string) };
Type target = openType.MakeGenericType(tArgs);

// Create an instance - Activator.CreateInstance will call the default constructor.
// This is equivalent to calling new List<string>().
List<string> result = (List<string>)Activator.CreateInstance(target);
```

The List<> syntax is not permitted outside of a typeof expression.

Without Activator class

Using new keyword (will do for parameterless constructors)

```
T GetInstance<T>() where T : new()
{
    T instance = new T();
    return instance;
}
```

Using Invoke method

```
// Get the instance of the desired constructor (here it takes a string as a parameter).
ConstructorInfo c = typeof(T).GetConstructor(new[] { typeof(string) });
// Don't forget to check if such constructor exists
if (c == null)
    throw new InvalidOperationException(string.Format("A constructor for type '{0}' was not
found.", typeof(T)));
T instance = (T)c.Invoke(new object[] { "test" });
```

Using Expression trees

Expression trees represent code in a tree-like data structure, where each node is an expression. As [MSDN](#) explains:

Expression is a sequence of one or more operands and zero or more operators that can be evaluated to a single value, object, method, or namespace. Expressions can consist of a literal value, a method invocation, an operator and its operands, or a simple name. Simple names can be the name of a variable, type member, method parameter, namespace or type.

```
public class GenericFactory< TKey, TType >
{
    private readonly Dictionary< TKey, Func< object[], TType > > _registeredTypes; // dictionary,
    that holds constructor functions.
    private object _locker = new object(); // object for locking dictionary, to guarantee thread
```

```

public GenericFactory()
{
    _registeredTypes = new Dictionary<TKey, Func<object[], TType>>();
}

/// <summary>
/// 查找并注册适合的类型构造函数
/// </summary>
/// <typeparam name="TType"></typeparam>
/// <param name="key">此构造函数的键</param>
/// <param name="parameters">参数</param>
public void Register(TKey key, params Type[] parameters)
{
    ConstructorInfo ci = typeof(TType).GetConstructor(BindingFlags.Public |
    BindingFlags.Instance, null, CallingConventions.HasThis, parameters, new ParameterModifier[] { });
    // 获取构造函数的实例。
    if (ci == null)
        throw new InvalidOperationException(string.Format("未找到类型 '{0}' 的构造函数。", typeof(TType)));
}

Func<object[], TType> ctor;

lock (_locker)
{
    if (!_registeredTypes.TryGetValue(key, out ctor)) // 检查是否已注册该构造函数
    {
        var pExp = Expression.Parameter(typeof(object[]), "arguments"); // 创建参数表达式
        var ctorParams = ci.GetParameters(); // 从构造函数获取参数信息

        var argExpressions = new Expression[ctorParams.Length]; // 将包含参数表达式的数组
        for (var i = 0; i < parameters.Length; i++)
        {
            var indexedAccess = Expression.ArrayIndex(pExp, Expression.Constant(i));

            if (!parameters[i].IsClass && !parameters[i].IsInterface) // 检查是否
                参数是值类型
            {
                var localVariable = Expression.Variable(parameters[i],
                "localVariable"); // 如果是 - 我们应该创建一个局部变量来存储参数值

                var block = Expression.Block(new[] { localVariable },
                    Expression.IfThenElse(Expression.Equal(indexedAccess,
                    Expression.Constant(null)),
                    Expression.Assign(localVariable,
                    Expression.Default(parameters[i])),
                    Expression.Assign(localVariable,
                    Expression.Convert(indexedAccess, parameters[i]))),
                localVariable
            );
            argExpressions[i] = block;
        }
        else
            argExpressions[i] = Expression.Convert(indexedAccess, parameters[i]);
    }
}

```

```

public GenericFactory()
{
    _registeredTypes = new Dictionary<TKey, Func<object[], TType>>();
}

/// <summary>
/// Find and register suitable constructor for type
/// </summary>
/// <typeparam name="TType"></typeparam>
/// <param name="key">Key for this constructor</param>
/// <param name="parameters">Parameters</param>
public void Register(TKey key, params Type[] parameters)
{
    ConstructorInfo ci = typeof(TType).GetConstructor(BindingFlags.Public |
    BindingFlags.Instance, null, CallingConventions.HasThis, parameters, new ParameterModifier[] { });
    // Get the instance of ctor.
    if (ci == null)
        throw new InvalidOperationException(string.Format("Constructor for type '{0}' was
        not found.", typeof(TType)));

    Func<object[], TType> ctor;

    lock (_locker)
    {
        if (!_registeredTypes.TryGetValue(key, out ctor)) // check if such ctor already
            been registered
        {
            var pExp = Expression.Parameter(typeof(object[]), "arguments"); // create
            parameter Expression
            var ctorParams = ci.GetParameters(); // get parameter info from constructor

            var argExpressions = new Expression[ctorParams.Length]; // array that will
            contains parameter expressions
            for (var i = 0; i < parameters.Length; i++)
            {
                var indexedAcccess = Expression.ArrayIndex(pExp, Expression.Constant(i));

                if (!parameters[i].IsClass && !parameters[i].IsInterface) // check if
                    parameter is a value type
                {
                    var localVariable = Expression.Variable(parameters[i],
                    "localVariable"); // if so - we should create local variable that will store paraameter value

                    var block = Expression.Block(new[] { localVariable },
                        Expression.IfThenElse(Expression.Equal(indexedAcccess,
                        Expression.Constant(null)),
                        Expression.Assign(localVariable,
                        Expression.Default(parameters[i])),
                        Expression.Assign(localVariable,
                        Expression.Convert(indexedAcccess, parameters[i]))),
                    localVariable
                );
                argExpressions[i] = block;
            }
            else
                argExpressions[i] = Expression.Convert(indexedAcccess, parameters[i]);
        }
    }
}

```

```

    }
    var newExpr = Expression.New(ci, argExpressions); // 创建表示调用指定构造函数及其指定参数
    的表达式。
    _registeredTypes.Add(key, Expression.Lambda(newExpr, new[] { pExp }).Compile()
    as Func<object[], TType>); // 编译表达式以创建委托，并将函数添加到字典中
}
}

/// <summary>
/// 通过键返回已注册类型的实例。
/// </summary>
/// <typeparam name="TType"></typeparam>
/// <param name="key"></param>
/// <param name="args"></param>
/// <returns></returns>
public TType Create(TKey key, params object[] args)
{
    Func<object[], TType> foo;
    if (_registeredTypes.TryGetValue(key, out foo))
    {
        return (TType)foo(args);
    }

    throw new ArgumentException("此键未注册类型。");
}
}

```

可以这样使用：

```

public class TestClass
{
    public TestClass(string parameter)
    {
        Console.WriteLine(parameter);
    }
}

public void TestMethod()
{
    var factory = new GenericFactory<string, TestClass>();
    factory.Register("key", typeof(string));
    TestClass newInstance = factory.Create("key", "testParameter");
}

```

使用 `FormatterServices.GetUninitializedObject`

```
T 实例 = (T)FormatterServices.GetUninitializedObject(typeof(T));
```

使用 `FormatterServices.GetUninitializedObject` 时，构造函数和字段初始化器不会被调用。它旨在用于序列化器和远程调用引擎

第63.4节：通过反射获取方法或属性的强类型委托

当性能成为关注点时，通过反射调用方法（即通过 `MethodInfo.Invoke` 方法）并不理想。然而，使用

```

    }
    var newExpr = Expression.New(ci, argExpressions); // create expression that
    represents call to specified ctor with the specified arguments.

    _registeredTypes.Add(key, Expression.Lambda(newExpr, new[] { pExp }).Compile()
    as Func<object[], TType>); // compile expression to create delegate, and add function to dictionary
}
}

/// <summary>
/// Returns instance of registered type by key.
/// </summary>
/// <typeparam name="TType"></typeparam>
/// <param name="key"></param>
/// <param name="args"></param>
/// <returns></returns>
public TType Create(TKey key, params object[] args)
{
    Func<object[], TType> foo;
    if (_registeredTypes.TryGetValue(key, out foo))
    {
        return (TType)foo(args);
    }

    throw new ArgumentException("No type registered for this key.");
}
}

```

Could be used like this:

```

public class TestClass
{
    public TestClass(string parameter)
    {
        Console.WriteLine(parameter);
    }
}

public void TestMethod()
{
    var factory = new GenericFactory<string, TestClass>();
    factory.Register("key", typeof(string));
    TestClass newInstance = factory.Create("key", "testParameter");
}

```

Using `FormatterServices.GetUninitializedObject`

```
T instance = (T)FormatterServices.GetUninitializedObject(typeof(T));
```

In case of using `FormatterServices.GetUninitializedObject` constructors and field initializers will not be called. It is meant to be used in serializers and remoting engines

Section 63.4: Get a Strongly-Typed Delegate to a Method or Property via Reflection

When performance is a concern, invoking a method via reflection (i.e. via the `MethodInfo.Invoke` method) is not ideal. However, it is relatively straightforward to obtain a more performant strongly-typed delegate using the

`Delegate.CreateDelegate` 函数获取更高性能的强类型委托相对简单。使用反射的性能损失仅在委托创建过程中产生。一旦委托创建完成，调用它几乎没有性能损失：

```
// 获取 Math.Max(int, int) 方法的 MethodInfo...
var maxMethod = typeof(Math).GetMethod("Max", new Type[] { typeof(int), typeof(int) });
// 现在获取一个针对 Math.Max(int, int) 的强类型委托...
var stronglyTypedDelegate = (Func<int, int, int>)Delegate.CreateDelegate(typeof(Func<int, int, int>), null, maxMethod);
// 使用强类型委托调用 Math.Max(int, int) 方法...
Console.WriteLine("3 和 5 中的最大值是: {0}", stronglyTypedDelegate(3, 5));
```

此技术也可以扩展到属性。如果我们有一个名为`MyClass`的类，且有一个名为`MyIntProperty`的`int`属性，获取强类型 getter 的代码如下（以下示例假设“target”是`MyClass`的有效实例）：

```
// 获取 MyClass.MyIntProperty getter 的 MethodInfo...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
var theGetter = theProperty.GetGetMethod();
// 现在获取一个针对 MyIntProperty 的强类型委托，可对任何 MyClass 实例执行...

var stronglyTypedGetter = (Func<MyClass, int>)Delegate.CreateDelegate(typeof(Func<MyClass, int>), theGetter);
// 对 MyClass 实例“target”调用 MyIntProperty getter...
Console.WriteLine("target.MyIntProperty 是: {0}", stronglyTypedGetter(target));
```

...同样的方法也可以用于设置器：

```
// 获取 MyClass.MyIntProperty 设置器的 MethodInfo...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
var theSetter = theProperty.GetSetMethod();
// 现在获取一个针对 MyIntProperty 的强类型委托，可对任何 MyClass 实例执行...

var stronglyTypedSetter = (Action<MyClass, int>)Delegate.CreateDelegate(typeof(Action<MyClass, int>), theSetter);
// 将 MyIntProperty 设置为 5...
stronglyTypedSetter(target, 5);
```

第63.5节：获取泛型方法并调用它

假设你有一个包含泛型方法的类。你需要通过反射调用它的函数。

```
public class Sample
{
    public void GenericMethod<T>()
    {
        // ...
    }

    public static void StaticMethod<T>()
    {
        //...
    }
}
```

假设我们想用类型 `string` 调用 `GenericMethod`。

```
Sample sample = new Sample(); //或者你也可以通过反射获取实例
```

`Delegate.CreateDelegate` function. The performance penalty for using reflection is incurred only during the delegate-creation process. Once the delegate is created, there is little-to-no performance penalty for invoking it:

```
// Get a MethodInfo for the Math.Max(int, int) method...
var maxMethod = typeof(Math).GetMethod("Max", new Type[] { typeof(int), typeof(int) });
// Now get a strongly-typed delegate for Math.Max(int, int)...
var stronglyTypedDelegate = (Func<int, int, int>)Delegate.CreateDelegate(typeof(Func<int, int, int>), null, maxMethod);
// Invoke the Math.Max(int, int) method using the strongly-typed delegate...
Console.WriteLine("Max of 3 and 5 is: {0}", stronglyTypedDelegate(3, 5));
```

This technique can be extended to properties as well. If we have a class named `MyClass` with an `int` property named `MyIntProperty`, the code to get a strongly-typed getter would be (the following example assumes 'target' is a valid instance of `MyClass`):

```
// Get a MethodInfo for the MyClass.MyIntProperty getter...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
var theGetter = theProperty.GetGetMethod();
// Now get a strongly-typed delegate for MyIntProperty that can be executed against any MyClass instance...
var stronglyTypedGetter = (Func<MyClass, int>)Delegate.CreateDelegate(typeof(Func<MyClass, int>), theGetter);
// Invoke the MyIntProperty getter against MyClass instance 'target',...
Console.WriteLine("target.MyIntProperty is: {0}", stronglyTypedGetter(target));
```

...and the same can be done for the setter:

```
// Get a MethodInfo for the MyClass.MyIntProperty setter...
var theProperty = typeof(MyClass).GetProperty("MyIntProperty");
var theSetter = theProperty.GetSetMethod();
// Now get a strongly-typed delegate for MyIntProperty that can be executed against any MyClass instance...
var stronglyTypedSetter = (Action<MyClass, int>)Delegate.CreateDelegate(typeof(Action<MyClass, int>), theSetter);
// Set MyIntProperty to 5...
stronglyTypedSetter(target, 5);
```

Section 63.5: Get a generic method and invoke it

Let's say you have class with generic methods. And you need to call its functions with reflection.

```
public class Sample
{
    public void GenericMethod<T>()
    {
        // ...
    }

    public static void StaticMethod<T>()
    {
        //...
    }
}
```

Let's say we want to call the `GenericMethod` with type `string`.

```
Sample sample = new Sample(); //or you can get an instance via reflection
```

```
MethodInfo method = typeof(Sample).GetMethod("GenericMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(sample, null); //因为没有参数，所以传入 null
```

对于静态方法，你不需要实例。因此第一个参数也将是 null。

```
MethodInfo method = typeof(Sample).GetMethod("StaticMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(null, null);
```

第63.6节：获取 System.Type

对于某个类型的实例：

```
var theString = "hello";
var theType = theString.GetType();
```

从类型本身：

```
var theType = typeof(string);
```

第63.7节：获取和设置属性

基本用法：

```
 PropertyInfo prop = myInstance.GetType().GetProperty("myProperty");
// 获取 myInstance.myProperty 的值
object value = prop.GetValue(myInstance);

int newValue = 1;
// 将 myInstance.myProperty 的值设置为 newValue
prop.SetValue(myInstance, newValue);
```

通过其支持字段可以设置只读的自动实现属性（在 .NET Framework 中支持字段的名称是 "k_BackingField"）：

```
// 获取支持字段信息
FieldInfo fieldInfo = myInstance.GetType()
    .GetField("<myProperty>k__BackingField", BindingFlags.Instance | BindingFlags.NonPublic);

int newValue = 1;
// 将 myInstance.myProperty 的 backing 字段值设置为 newValue
fieldInfo.SetValue(myInstance, newValue);
```

第63.8节：创建泛型类型的实例并调用其方法

```
var baseType = typeof(List<>);
var genericType = baseType.MakeGenericType(typeof(String));
var instance = Activator.CreateInstance(genericType);
var method = genericType.GetMethod("GetHashCode");
var result = method.Invoke(instance, new object[] { });
```

```
MethodInfo method = typeof(Sample).GetMethod("GenericMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(sample, null); //Since there are no arguments, we are passing null
```

For the static method you do not need an instance. Therefore the first argument will also be null.

```
MethodInfo method = typeof(Sample).GetMethod("StaticMethod");
MethodInfo generic = method.MakeGenericMethod(typeof(string));
generic.Invoke(null, null);
```

Section 63.6: Get a System.Type

For an instance of a type:

```
var theString = "hello";
var theType = theString.GetType();
```

From the type itself:

```
var theType = typeof(string);
```

Section 63.7: Getting and setting properties

Basic usage:

```
 PropertyInfo prop = myInstance.GetType().GetProperty("myProperty");
// get the value myInstance.myProperty
object value = prop.GetValue(myInstance);

int newValue = 1;
// set the value myInstance.myProperty to newValue
prop.SetValue(myInstance, newValue);
```

Setting read-only automatically-implemented properties can be done through its backing field (in .NET Framework name of backing field is "k_BackingField"):

```
// get backing field info
FieldInfo fieldInfo = myInstance.GetType()
    .GetField("<myProperty>k__BackingField", BindingFlags.Instance | BindingFlags.NonPublic);

int newValue = 1;
// set the value of myInstance.myProperty backing field to newValue
fieldInfo.SetValue(myInstance, newValue);
```

Section 63.8: Create an instance of a Generic Type and invoke its method

```
var baseType = typeof(List<>);
var genericType = baseType.MakeGenericType(typeof(String));
var instance = Activator.CreateInstance(genericType);
var method = genericType.GetMethod("GetHashCode");
var result = method.Invoke(instance, new object[] { });
```

第63.9节：自定义特性

查找带有自定义特性 - MyAttribute 的属性

```
var props = t.GetProperties(BindingFlags.NonPublic | BindingFlags.Public |  
    BindingFlags.Instance).Where(  
    prop => Attribute.IsDefined(prop, typeof(MyAttribute)));
```

查找给定属性上的所有自定义属性

```
var attributes = typeof(t).GetProperty("Name").GetCustomAttributes(false);
```

枚举所有带有自定义属性 - MyAttribute的类

```
static IEnumerable<Type> GetTypesWithAttribute(Assembly assembly) {  
    foreach(Type type in assembly.GetTypes()) {  
        if (type.GetCustomAttributes(typeof(MyAttribute), true).Length > 0) {  
            yield return type;  
        }  
    }  
}
```

在运行时读取自定义属性的值

```
public static class AttributeExtensions  
{  
  
    /// <summary>  
    /// 返回类中任意成员属性的值。  
    ///     (成员可以是字段、属性、方法等...)  
    /// <remarks>  
    /// 如果类中有多个同名成员，将返回第一个  
一个 (这适用于重载方法)  
    /// </remarks>  
    /// <example>  
    /// 从类  
  
    ///     var Attribute = typeof(MyClass).GetAttribute("MyMethodName", (DescriptionAttribute d)  
=> d.Description);  
    /// </example>  
    /// <param name="type">包含该成员的类类型</param>  
    /// <param name="MemberName">类中成员的名称</param>  
    /// <param name="valueSelector">要获取的属性类型和属性 (如果存在多个相同类型的属性，将返回第一个实  
例) </param>/// <param name="inherit">是否搜索该成员的继承链以查找属性；否则  
为 false。此参数对属性和事件无效</param>/// </summary>  
  
    public static TValue GetAttribute<TAttribute, TValue>(this Type type, string MemberName,  
Func<TAttribute, TValue> valueSelector, bool inherit = false) where TAttribute : Attribute  
{  
    var att =  
type.GetMember(MemberName).FirstOrDefault().GetCustomAttributes(typeof(TAttribute),  
inherit).FirstOrDefault() as TAttribute;  
    if (att != null)  
    {  
        return valueSelector(att);  
    }  
    return default(TValue);  
}
```

Section 63.9: Custom Attributes

Find properties with a custom attribute - MyAttribute

```
var props = t.GetProperties(BindingFlags.NonPublic | BindingFlags.Public |  
    BindingFlags.Instance).Where(  
    prop => Attribute.IsDefined(prop, typeof(MyAttribute)));
```

Find all custom attributes on a given property

```
var attributes = typeof(t).GetProperty("Name").GetCustomAttributes(false);
```

Enumerate all classes with custom attribute - MyAttribute

```
static IEnumerable<Type> GetTypesWithAttribute(Assembly assembly) {  
    foreach(Type type in assembly.GetTypes()) {  
        if (type.GetCustomAttributes(typeof(MyAttribute), true).Length > 0) {  
            yield return type;  
        }  
    }  
}
```

Read value of a custom attribute at runtime

```
public static class AttributeExtensions  
{  
  
    /// <summary>  
    /// Returns the value of a member attribute for any member in a class.  
    ///     (a member is a Field, Property, Method, etc...)  
    /// <remarks>  
    /// If there is more than one member of the same name in the class, it will return the first  
one (this applies to overloaded methods)  
    /// </remarks>  
    /// <example>  
    /// Read System.ComponentModel.DescriptionAttribute from method 'MyMethodName' in class  
'MyClass':  
    ///     var Attribute = typeof(MyClass).GetAttribute("MyMethodName", (DescriptionAttribute d)  
=> d.Description);  
    /// </example>  
    /// <param name="type">The class that contains the member as a type</param>  
    /// <param name="MemberName">Name of the member in the class</param>  
    /// <param name="valueSelector">Attribute type and property to get (will return first  
instance if there are multiple attributes of the same type)</param>  
    /// <param name="inherit">true to search this member's inheritance chain to find the  
attributes; otherwise, false. This parameter is ignored for properties and events</param>  
    /// </summary>  
    public static TValue GetAttribute<TAttribute, TValue>(this Type type, string MemberName,  
Func<TAttribute, TValue> valueSelector, bool inherit = false) where TAttribute : Attribute  
{  
    var att =  
type.GetMember(MemberName).FirstOrDefault().GetCustomAttributes(typeof(TAttribute),  
inherit).FirstOrDefault() as TAttribute;  
    if (att != null)  
    {  
        return valueSelector(att);  
    }  
    return default(TValue);  
}
```

}

用法

```
// 从类 'MyClass' 中的方法 'MyMethodName' 读取 System.ComponentModel.Description 属性
var Attribute = typeof(MyClass).GetAttribute("MyMethodName", (DescriptionAttribute d) =>
d.Description);
```

第63.10节：实例化实现接口的类（例如插件激活）

如果您希望您的应用程序支持插件系统，例如从位于

plugins 文件夹加载插件：

```
interface IPlugin
{
    string PluginDescription { get; }
    void DoWork();
}
```

此类将位于单独的dll中

```
class HelloPlugin : IPlugin
{
    public string PluginDescription => "一个会说你好插件";
    public void DoWork()
    {
        Console.WriteLine("你好");
    }
}
```

您的应用程序的插件加载器会找到dll文件，获取那些程序集内实现了IPlugin的所有类型，并创建它们的实例。

```
public IEnumerable<IPlugin> InstantiatePlugins(string directory)
{
    var pluginAssemblyNames = Directory.GetFiles(directory, "*addin.dll").Select(name => new
    FileInfo(name).FullName).ToArray();
    //将程序集加载到当前AppDomain中，以便稍后实例化类型
    foreach (var fileName in pluginAssemblyNames)
        AppDomain.CurrentDomain.Load(File.ReadAllBytes(fileName));
    var assemblies = pluginAssemblyNames.Select(System.Reflection.Assembly.LoadFile);
    var typesInAssembly = assemblies.SelectMany(asm => asm.GetTypes());
    var pluginTypes = typesInAssembly.Where(type => typeof(IPlugin).IsAssignableFrom(type));
    return pluginTypes.Select(Activator.CreateInstance).Cast<IPlugin>();
}
```

第63.11节：通过带命名空间的名称获取类型

为此，您需要引用包含该类型的程序集。如果您有另一个类型，并且知道它与您想要的类型在同一程序集中，可以这样做：

```
typeof(KnownType).Assembly.GetType(typeName);
```

- 其中 typeName 是你要查找的类型名称（包括命名空间），KnownType 是你知道位于同一程序集中的类型。

}

Usage

```
//Read System.ComponentModel.Description Attribute from method 'MyMethodName' in class 'MyClass'
var Attribute = typeof(MyClass).GetAttribute("MyMethodName", (DescriptionAttribute d) =>
d.Description);
```

Section 63.10: Instantiating classes that implement an interface (e.g. plugin activation)

If you want your application to support a plug-in system, for example to load plug-ins from assemblies located in plugins folder:

```
interface IPlugin
{
    string PluginDescription { get; }
    void DoWork();
}
```

This class would be located in a separate dll

```
class HelloPlugin : IPlugin
{
    public string PluginDescription => "A plugin that says Hello";
    public void DoWork()
    {
        Console.WriteLine("Hello");
    }
}
```

Your application's plugin loader would find the dll files, get all types in those assemblies that implement IPlugin, and create instances of those.

```
public IEnumerable<IPlugin> InstantiatePlugins(string directory)
{
    var pluginAssemblyNames = Directory.GetFiles(directory, "*addin.dll").Select(name => new
    FileInfo(name).FullName).ToArray();
    //load the assemblies into the current AppDomain, so we can instantiate the types later
    foreach (var fileName in pluginAssemblyNames)
        AppDomain.CurrentDomain.Load(File.ReadAllBytes(fileName));
    var assemblies = pluginAssemblyNames.Select(System.Reflection.Assembly.LoadFile);
    var typesInAssembly = assemblies.SelectMany(asm => asm.GetTypes());
    var pluginTypes = typesInAssembly.Where(type => typeof(IPlugin).IsAssignableFrom(type));
    return pluginTypes.Select(Activator.CreateInstance).Cast<IPlugin>();
}
```

Section 63.11: Get a Type by name with namespace

To do this you need a reference to the assembly which contains the type. If you have another type available which you know is in the same assembly as the one you want you can do this:

```
typeof(KnownType).Assembly.GetType(typeName);
```

- where typeName is the name of the type you are looking for (including the namespace), and KnownType is the type you know is in the same assembly.

效率较低但更通用的方法如下：

```
Type t = null;
foreach (Assembly ass in AppDomain.CurrentDomain.GetAssemblies())
{
    if (ass.FullName.StartsWith("System."))
        continue;
    t = ass.GetType(typeName);
    if (t != null)
        break;
}
```

注意这里检查并排除了 System 命名空间的程序集扫描，以加快搜索速度。如果你的类型可能实际上是 CLR 类型，则需要删除这两行代码。

如果你恰好拥有包含程序集的完全限定类型名称，可以直接使用

```
Type.GetType(fullyQualifiedName);
```

第63.12节：确定泛型类型实例的泛型参数

如果你有一个泛型类型的实例，但由于某种原因不知道具体的类型，你可能想确定用于创建该实例的泛型参数。

假设有人创建了一个List<T>的实例，并将其传递给一个方法：

```
var myList = new List<int>();
ShowGenericArguments(myList);
```

其中ShowGenericArguments具有以下签名：

```
public void ShowGenericArguments(object o)
```

因此在编译时你无法知道用于创建o的泛型参数。反射（Reflection）提供了许多方法来检查泛型类型。[首先](#)，我们可以确定o的类型是否为泛型类型：

```
public void ShowGenericArguments(object o)
{
    if (o == null) return;

    Type t = o.GetType();
    if (!t.IsGenericType) return;
    ...
}
```

Type.IsGenericType如果类型是泛型类型则返回true，否则返回false。

但这还不是我们想了解的全部。List<>本身也是一个泛型类型。但我们只想检查特定构造的泛型类型的实例。构造的泛型类型例如List<int>，它为其所有泛型参数提供了具体的类型参数。

Type类提供了另外两个属性，IsConstructedGenericType和IsGenericTypeDefinition，用于区分这些构造的泛型类型和泛型类型定义：

```
typeof(List<>).IsGenericType // true
typeof(List<>).IsGenericTypeDefinition // true
```

Less efficient but more general is as follows:

```
Type t = null;
foreach (Assembly ass in AppDomain.CurrentDomain.GetAssemblies())
{
    if (ass.FullName.StartsWith("System."))
        continue;
    t = ass.GetType(typeName);
    if (t != null)
        break;
}
```

Notice the check to exclude scanning System namespace assemblies to speed up the search. If your type may actually be a CLR type, you will have to delete these two lines.

If you happen to have the fully assembly-qualified type name including the assembly you can simply get it with

```
Type.GetType(fullyQualifiedName);
```

Section 63.12: Determining generic arguments of instances of generic types

If you have an instance of a generic type but for some reason don't know the specific type, you might want to determine the generic arguments that were used to create this instance.

Let's say someone created an instance of List<T> like that and passes it to a method:

```
var myList = new List<int>();
ShowGenericArguments(myList);
```

where ShowGenericArguments has this signature:

```
public void ShowGenericArguments(object o)
```

so at compile time you don't have any idea what generic arguments have been used to create o. [Reflection](#) provides a lot of methods to inspect generic types. At first, we can determine if the type of o is a generic type at all:

```
public void ShowGenericArguments(object o)
{
    if (o == null) return;

    Type t = o.GetType();
    if (!t.IsGenericType) return;
    ...
}
```

Type.IsGenericType returns true if the type is a generic type and false if not.

But this is not all we want to know. List<> itself is a generic type, too. But we only want to examine instances of specific *constructed generic* types. A constructed generic type is for example a List<int> that has a specific type argument for all its generic parameters.

The Type class provides two more properties, [IsConstructedGenericType](#) and [IsGenericTypeDefinition](#), to distinguish these constructed generic types from generic type definitions:

```
typeof(List<>).IsGenericType // true
typeof(List<>).IsGenericTypeDefinition // true
```

```

typeof(List<>).IsConstructedGenericType// false
typeof(List<int>).IsGenericType // true
typeof(List<int>).IsGenericTypeDefinition // false
typeof(List<int>).IsConstructedGenericType// true

```

要枚举实例的泛型参数，我们可以使用GetGenericArguments()方法，该方法返回一个包含泛型类型参数的Type数组：

```

public void ShowGenericArguments(object o)
{
    if (o == null) return;
    Type t = o.GetType();
    if (!t.IsConstructedGenericType) return;

    foreach(Type genericTypeArgument in t.GetGenericArguments())
        Console.WriteLine(genericTypeArgument.Name);
}

```

因此，上面调用的ShowGenericArguments(myList)会产生如下输出：

Int32

第63.13节：遍历一个类的所有属性

```

Type type = obj.GetType();
// 限制返回属性。如果需要所有属性，则不提供标志。
BindingFlags flags = BindingFlags.Public | BindingFlags.Instance;
 PropertyInfo[] properties = type.GetProperties(flags);

foreach (PropertyInfo property in properties)
{
    Console.WriteLine("Name: " + property.Name + ", Value: " + property.GetValue(obj, null));
}

```

```

typeof(List<>).IsConstructedGenericType// false

```

```

typeof(List<int>).IsGenericType // true
typeof(List<int>).IsGenericTypeDefinition // false
typeof(List<int>).IsConstructedGenericType// true

```

To enumerate the generic arguments of an instance, we can use the [GetGenericArguments\(\)](#) method that returns an Type array containing the generic type arguments:

```

public void ShowGenericArguments(object o)
{
    if (o == null) return;
    Type t = o.GetType();
    if (!t.IsConstructedGenericType) return;

    foreach(Type genericTypeArgument in t.GetGenericArguments())
        Console.WriteLine(genericTypeArgument.Name);
}

```

So the call from above (ShowGenericArguments(myList)) results in this output:

Int32

Section 63.13: Looping through all the properties of a class

```

Type type = obj.GetType();
// To restrict return properties. If all properties are required don't provide flag.
BindingFlags flags = BindingFlags.Public | BindingFlags.Instance;
 PropertyInfo[] properties = type.GetProperties(flags);

foreach (PropertyInfo property in properties)
{
    Console.WriteLine("Name: " + property.Name + ", Value: " + property.GetValue(obj, null));
}

```

第64章：IQueryable接口

第64.1节：将LINQ查询转换为SQL查询

IQueryable和IQueryable<T>接口允许开发者将LINQ查询（“语言集成”查询）转换为特定的数据源，例如关系数据库。以下是用C#编写的LINQ查询：

```
var query = from book in books
            where book.Author == "Stephen King"
            select book;
```

如果变量books的类型实现了IQueryable<Book>接口，那么上述查询将以表达式树的形式传递给提供程序（设置在IQueryable.Provider属性上），表达式树是一种反映代码结构的数据结构。

提供程序可以在运行时检查表达式树以确定：

- 存在针对Book类的Author属性的谓词；
- 使用的比较方法是“等于”（==）；
- 它应该等于的值是"Stephen King"。

有了这些信息，提供程序可以在运行时将C#查询转换为SQL查询，并将该查询传递给关系数据库，以仅获取符合谓词的书籍：

```
SELECT *
FROM Books
WHERE Author = 'Stephen King'
```

当对query变量进行迭代时会调用提供程序（IQueryable实现了IEnumerable）。

（本示例中使用的提供程序需要一些额外的元数据来确定查询哪个表以及如何将C#类的属性映射到表的列，但此类元数据超出了 IQueryable接口的范围。）

Chapter 64: IQueryable interface

Section 64.1: Translating a LINQ query to a SQL query

The IQueryable and IQueryable<T> interfaces allows developers to translate a LINQ query (a 'language-integrated' query) to a specific datasource, for example a relational database. Take this LINQ query written in C#:

```
var query = from book in books
            where book.Author == "Stephen King"
            select book;
```

If the variable books is of a type that implements IQueryable<Book> then the query above gets passed to the provider (set on the IQueryable.Provider property) in the form of an expression tree, a data structure that reflects the structure of the code.

The provider can inspect the expression tree at runtime to determine:

- that there is a predicate for the Author property of the Book class;
- that the comparison method used is 'equals' (==);
- that the value it should equal is "Stephen King".

With this information the provider can translate the C# query to a SQL query at runtime and pass that query to a relational database to fetch only those books that match the predicate:

```
SELECT *
FROM Books
WHERE Author = 'Stephen King'
```

The provider gets called when the query variable is iterated over (IQueryable implements IEnumerable).

(The provider used in this example would require some extra metadata to know which table to query and to know how to match properties of the C# class to columns of the table, but such metadata is outside of the scope of the IQueryable interface.)

第65章：对象的Linq

LINQ to Objects 指的是在任何 `IEnumerable` 集合上使用 LINQ 查询。

第65.1节：在 C# 中使用 LINQ to Objects

一个简单的 Linq SELECT 查询

```
static void Main(string[] args)
{
    string[] cars = { "大众高尔夫",
                      "欧宝雅特",
                      "奥迪 A4",
                      "福特福克斯",
                      "西雅特利昂",
                      "大众帕萨特",
                      "大众波罗",
                      "奔驰 C 级" };

    var list = from car in cars
              select car;

    StringBuilder sb = new StringBuilder();

    foreach (string entry in list)
    {
        sb.Append(entry + "\n");
    }

    Console.WriteLine(sb.ToString());
    Console.ReadLine();
}
```

在上面的示例中，使用字符串数组（cars）作为要使用LINQ查询的对象集合。在LINQ查询中，`from`子句首先出现，用于引入数据源（cars）和范围变量（car）。当查询执行时，范围变量将作为对cars中每个连续元素的引用。因为编译器可以推断car的类型，所以不必显式指定。

当上述代码被编译并执行时，会产生以下结果：

```
VW Golf
Opel Astra
Audi A4
Ford Focus
Seat Leon
VW Passat
VW Polo
Mercedes C-Class
```

带有WHERE子句的SELECT

```
var list = from car in cars
           where car.Contains("VW")
           select car;
```

WHERE子句用于查询字符串数组（cars），以查找并返回满足条件的数组子集

Chapter 65: Linq to Objects

LINQ to Objects refers to the use of LINQ queries with any `IEnumerable` collection.

Section 65.1: Using LINQ to Objects in C#

A simple SELECT query in Linq

```
static void Main(string[] args)
{
    string[] cars = { "VW Golf",
                      "Opel Astra",
                      "Audi A4",
                      "Ford Focus",
                      "Seat Leon",
                      "VW Passat",
                      "VW Polo",
                      "Mercedes C-Class" };

    var list = from car in cars
              select car;

    StringBuilder sb = new StringBuilder();

    foreach (string entry in list)
    {
        sb.Append(entry + "\n");
    }

    Console.WriteLine(sb.ToString());
    Console.ReadLine();
}
```

In the example above, an array of strings (cars) is used as a collection of objects to be queried using LINQ. In a LINQ query, the `from` clause comes first in order to introduce the data source (cars) and the range variable (car). When the query is executed, the range variable will serve as a reference to each successive element in cars. Because the compiler can infer the type of car, you do not have to specify it explicitly

When the above code is compiled and executed, it produces the following result:

```
VW Golf
Opel Astra
Audi A4
Ford Focus
Seat Leon
VW Passat
VW Polo
Mercedes C-Class
```

SELECT with a WHERE Clause

```
var list = from car in cars
           where car.Contains("VW")
           select car;
```

The WHERE clause is used to query the string array (cars) to find and return a subset of array which satisfies the

WHERE 子句。

当上述代码被编译并执行时，会产生以下结果：

```
VW Golf  
VW Passat  
VW Polo
```

生成有序列表

```
var list = from car in cars  
          orderby car ascending  
          select car;
```

有时对返回的数据进行排序是有用的。orderby 子句将根据被排序类型的默认比较器对元素进行排序。

当上述代码被编译并执行时，会产生以下结果：

```
Audi A4  
Ford Focus  
Mercedes C-Class  
Opel Astra  
Seat Leon  
VW Golf  
VW Passat  
VW Polo
```

使用自定义类型

在此示例中，创建了一个类型化列表，填充数据，然后进行查询

```
public class Car  
{  
    public String Name { get; private set; }  
    public int UnitsSold { get; private set; }  
  
    public Car(string name, int unitsSold)  
    {  
        Name = name;  
        UnitsSold = unitsSold;  
    }  
}  
  
class Program  
{  
    static void Main(string[] args)  
    {  
  
        var car1 = new Car("大众高尔夫", 270952);  
        var car2 = new Car("欧宝雅特", 56079);  
        var car3 = new Car("奥迪A4", 52493);  
        var car4 = new Car("福特福克斯", 51677);  
        var car5 = new Car("西雅特利昂", 42125);  
    }  
}
```

WHERE clause.

When the above code is compiled and executed, it produces the following result:

```
VW Golf  
VW Passat  
VW Polo
```

Generating an Ordered List

```
var list = from car in cars  
          orderby car ascending  
          select car;
```

Sometimes it is useful to sort the returned data. The orderby clause will cause the elements to be sorted according to the default comparer for the type being sorted.

When the above code is compiled and executed, it produces the following result:

```
Audi A4  
Ford Focus  
Mercedes C-Class  
Opel Astra  
Seat Leon  
VW Golf  
VW Passat  
VW Polo
```

Working with a custom type

In this example, a typed list is created, populated, and then queried

```
public class Car  
{  
    public String Name { get; private set; }  
    public int UnitsSold { get; private set; }  
  
    public Car(string name, int unitsSold)  
    {  
        Name = name;  
        UnitsSold = unitsSold;  
    }  
}  
  
class Program  
{  
    static void Main(string[] args)  
    {  
  
        var car1 = new Car("VW Golf", 270952);  
        var car2 = new Car("Opel Astra", 56079);  
        var car3 = new Car("Audi A4", 52493);  
        var car4 = new Car("Ford Focus", 51677);  
        var car5 = new Car("Seat Leon", 42125);  
    }  
}
```

```

var car6 = new Car("大众帕萨特", 97586);
var car7 = new Car("大众波罗", 69867);
var car8 = new Car("奔驰C级", 67549);

var cars = new List<Car> {
    car1, car2, car3, car4, car5, car6, car7, car8 };
    var list = from car in cars
        select car.Name;

    foreach (var entry in list)
    {
Console.WriteLine(entry);
    }
Console.ReadLine();
}
}

```

当上述代码被编译并执行时，会产生以下结果：

```

VW Golf
Opel Astra
Audi A4
Ford Focus
Seat Leon
VW Passat
VW Polo
Mercedes C-Class

```

到目前为止，这些示例看起来并不惊艳，因为人们可以直接遍历数组来完成基本相同的操作。然而，通过下面的几个示例，你可以看到如何使用 LINQ to Objects 创建更复杂的查询，并用更少的代码实现更多功能。

在下面的示例中，我们可以选择销量超过 60000 辆的汽车，并按销量数量进行排序：

```

var list = from car in cars
    where car.UnitsSold > 60000
    orderby car.UnitsSold descending
    select car;

StringBuilder sb = new StringBuilder();

foreach (var entry in list)
{
    sb.AppendLine($"{entry.Name} - {entry.UnitsSold}");
}
Console.WriteLine(sb.ToString());

```

当上述代码被编译并执行时，会产生以下结果：

```

var car6 = new Car("VW Passat", 97586);
var car7 = new Car("VW Polo", 69867);
var car8 = new Car("Mercedes C-Class", 67549);

var cars = new List<Car> {
    car1, car2, car3, car4, car5, car6, car7, car8 };
    var list = from car in cars
        select car.Name;

    foreach (var entry in list)
    {
        Console.WriteLine(entry);
    }
    Console.ReadLine();
}
}

```

When the above code is compiled and executed, it produces the following result:

```

VW Golf
Opel Astra
Audi A4
Ford Focus
Seat Leon
VW Passat
VW Polo
Mercedes C-Class

```

Until now the examples don't seem amazing as one can just iterate through the array to do basically the same. However, with the few examples below you can see how to create more complex queries with LINQ to Objects and achieve more with a lot less of code.

In the example below we can select cars that have been sold over 60000 units and sort them over the number of units sold:

```

var list = from car in cars
    where car.UnitsSold > 60000
    orderby car.UnitsSold descending
    select car;

StringBuilder sb = new StringBuilder();

foreach (var entry in list)
{
    sb.AppendLine($"{entry.Name} - {entry.UnitsSold}");
}
Console.WriteLine(sb.ToString());

```

When the above code is compiled and executed, it produces the following result:

```
VW Golf - 270952  
VW Passat - 97586  
VW Polo - 69867  
Mercedes C-Class - 67549
```

在下面的示例中，我们可以选择销售数量为奇数的汽车，并按名称字母顺序排序：

```
var list = from car in cars  
           where car.UnitsSold % 2 != 0  
           orderby car.Name ascending  
           select car;
```

当上述代码被编译并执行时，会产生以下结果：

```
Audi A4 - 52493  
Ford Focus - 51677  
Mercedes C-Class - 67549  
Opel Astra - 56079  
Seat Leon - 42125  
VW Polo - 69867
```

```
VW Golf - 270952  
VW Passat - 97586  
VW Polo - 69867  
Mercedes C-Class - 67549
```

In the example below we can select cars that have sold an odd number of units and order them alphabetically over its name:

```
var list = from car in cars  
           where car.UnitsSold % 2 != 0  
           orderby car.Name ascending  
           select car;
```

When the above code is compiled and executed, it produces the following result:

```
Audi A4 - 52493  
Ford Focus - 51677  
Mercedes C-Class - 67549  
Opel Astra - 56079  
Seat Leon - 42125  
VW Polo - 69867
```

第66章：LINQ查询

LINQ是“语言集成查询”（Language Integrated Query）的缩写。它是一种概念，通过提供一个一致的模型来处理各种数据源和格式的数据查询语言；你可以使用相同的基本编码模式来查询和转换XML文档、SQL数据库、ADO.NET数据集、.NET集合以及任何有LINQ提供程序支持的其他格式的数据。

第66.1节：方法链

许多LINQ函数既作用于`IEnumerable<TSource>`，也返回`IEnumerable<TResult>`。类型参数`TSource`和`TResult`可能相同，也可能不同，这取决于具体的方法以及传入的方法参数。

以下是一些示例

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector
)

public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate
)

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector
)
```

虽然某些方法链可能需要先处理整个集合才能继续，但LINQ利用了延迟执行，通过使用`yield return` MSDN，在后台创建了一个`Enumerable`和一个`Enumerator`。LINQ中的链式调用过程本质上是为原始集合构建一个可枚举对象（迭代器）——这是延迟的——直到通过枚举该可枚举对象时才会被实际执行。

这使得这些函数可以流畅地链式调用wiki，其中一个函数可以直接作用于另一个函数的结果。这种代码风格可以用来在单个语句中执行许多基于序列的操作。

例如，可以将`SELECT`、`Where`和`OrderBy`组合起来，在单个语句中对序列进行转换、过滤和排序。

```
var someNumbers = { 4, 3, 2, 1 };

var processed = someNumbers
    .选择(n => n * 2) // 将每个数字乘以2
    .筛选(n => n != 6) // 保留所有结果，除了6
    .排序(n => n); // 按升序排序
```

输出：

2
4
8

Chapter 66: LINQ Queries

LINQ is an acronym which stands for **L**anguage **I**ntegrated **Q**uery. It is a concept which integrates a query language by offering a consistent model for working with data across various kinds of data sources and formats; you use the same basic coding patterns to query and transform data in XML documents, SQL databases, ADO.NET Datasets, .NET collections, and any other format for which a LINQ provider is available.

Section 66.1: Chaining methods

Many LINQ functions both operate on an `IEnumerable<TSource>` and also return an `IEnumerable<TResult>`. The type parameters `TSource` and `TResult` may or may not refer to the same type, depending on the method in question and any functions passed to it.

A few examples of this are

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector
)

public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate
)

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector
)
```

While some method chaining may require an entire set to be worked prior to moving on, LINQ takes advantage of deferred execution by using `yield return` MSDN which creates an `Enumerable` and an `Enumerator` behind the scenes. The process of chaining in LINQ is essentially building an enumerable (iterator) for the original set -- which is deferred -- until materialized by enumerating the enumerable.

This allows these functions to be [fluently chained](#) wiki, where one function can act directly on the result of another. This style of code can be used to perform many sequence based operations in a single statement.

For example, it's possible to combine `SELECT`, `Where` and `OrderBy` to transform, filter and sort a sequence in a single statement.

```
var someNumbers = { 4, 3, 2, 1 };

var processed = someNumbers
    .Select(n => n * 2) // Multiply each number by 2
    .Where(n => n != 6) // Keep all the results, except for 6
    .OrderBy(n => n); // Sort in ascending order
```

Output:

2
4
8

任何既扩展又返回泛型`IEnumerable<T>`类型的函数都可以作为链式子句在单个语句中使用。这种流畅的编程风格非常强大，创建自己的扩展方法时应予以考虑。

第66.2节：First、FirstOrDefault、Last、LastOrDefault、Single和SingleOrDefault

这六个方法都返回序列类型的单个值，可以带有或不带有谓词调用。

根据匹配predicate的元素数量，或者如果未提供predicate，则根据源序列中的元素数量，它们的行为如下：

First()

- 返回序列的第一个元素，或匹配提供的predicate的第一个元素。
- 如果序列不包含任何元素，则抛出`InvalidOperationException`，消息为：“序列不包含任何元素”。
- 如果序列中没有匹配提供的predicate的元素，则抛出`InvalidOperationException`，消息为“序列不包含匹配的元素”。

示例

```
// 返回 "a":  
new[] { "a" }.First();  
  
// 返回 "a":  
new[] { "a", "b" }.First();  
  
// 返回 "b":  
new[] { "a", "b" }.First(x => x.Equals("b"));  
  
// 返回 "ba":  
new[] { "ba", "be" }.First(x => x.Contains("b"));  
  
// 抛出 InvalidOperationException 异常:  
new[] { "ca", "ce" }.First(x => x.Contains("b"));  
  
// 抛出 InvalidOperationException 异常:  
new string[0].First();
```

FirstOrDefault()

- 返回序列的第一个元素，或匹配提供的predicate的第一个元素。
- 如果序列不包含任何元素，或没有任何元素符合提供的谓词，则使用`default(T)`返回序列类型的默认值。

示例

```
// 返回 "a":  
new[] { "a" }.FirstOrDefault();  
  
// 返回 "a":  
new[] { "a", "b" }.FirstOrDefault();
```

Any functions that both extend and return the generic `IEnumerable<T>` type can be used as chained clauses in a single statement. This style of fluent programming is powerful, and should be considered when creating your own extension methods.

Section 66.2: First, FirstOrDefault, Last, LastOrDefault, Single, and SingleOrDefault

All six methods return a single value of the sequence type, and can be called with or without a predicate.

Depending on the number of elements that match the predicate or, if no predicate is supplied, the number of elements in the source sequence, they behave as follows:

First()

- Returns the first element of a sequence, or the first element matching the provided predicate.
- If the sequence contains no elements, an `InvalidOperationException` is thrown with the message: “Sequence contains no elements”.
- If the sequence contains no elements matching the provided predicate, an `InvalidOperationException` is thrown with the message “Sequence contains no matching element”.

Example

```
// Returns "a":  
new[] { "a" }.First();  
  
// Returns "a":  
new[] { "a", "b" }.First();  
  
// Returns "b":  
new[] { "a", "b" }.First(x => x.Equals("b"));  
  
// Returns "ba":  
new[] { "ba", "be" }.First(x => x.Contains("b"));  
  
// Throws InvalidOperationException:  
new[] { "ca", "ce" }.First(x => x.Contains("b"));  
  
// Throws InvalidOperationException:  
new string[0].First();
```

FirstOrDefault()

- Returns the first element of a sequence, or the first element matching the provided predicate.
- If the sequence contains no elements, or no elements matching the provided predicate, returns the default value of the sequence type using `default(T)`.

Example

```
// Returns "a":  
new[] { "a" }.FirstOrDefault();  
  
// Returns "a":  
new[] { "a", "b" }.FirstOrDefault();
```

```
// 返回 "b":  
new[] { "a", "b" }.FirstOrDefault(x => x.Equals("b"));  
  
// 返回 "ba":  
new[] { "ba", "be" }.FirstOrDefault(x => x.Contains("b"));  
  
// 返回 null:  
new[] { "ca", "ce" }.FirstOrDefault(x => x.Contains("b"));  
  
// 返回 null:  
new string[0].FirstOrDefault();
```

[.NET Fiddle 在线演示](#)

Last()

- 返回序列的最后一个元素，或匹配提供的predicate的最后一个元素。
- 如果序列不包含任何元素，则会抛出带有消息“序列不包含任何元素”的InvalidOperationException异常。
- 如果序列中没有匹配提供的predicate的元素，则抛出InvalidOperationException，消息为“序列不包含匹配的元素”。

示例

```
// 返回 "a":  
new[] { "a" }.Last();  
  
// 返回 "b":  
new[] { "a", "b" }.Last();  
  
// 返回 "a":  
new[] { "a", "b" }.Last(x => x.Equals("a"));  
  
// 返回 "be":  
new[] { "ba", "be" }.Last(x => x.Contains("b"));  
  
// 抛出 InvalidOperationException 异常:  
new[] { "ca", "ce" }.Last(x => x.Contains("b"));  
  
// 抛出 InvalidOperationException 异常:  
new string[0].Last();
```

LastOrDefault()

- 返回序列的最后一个元素，或匹配提供的predicate的最后一个元素。
- 如果序列不包含任何元素，或没有任何元素符合提供的谓词，则使用default(T)返回序列类型的默认值。

示例

```
// 返回 "a":  
new[] { "a" }.LastOrDefault();  
  
// 返回 "b":  
new[] { "a", "b" }.LastOrDefault();  
  
// 返回 "a":  
new[] { "a", "b" }.LastOrDefault(x => x.Equals("a"));  
  
// 返回 "be":
```

```
// Returns "b":  
new[] { "a", "b" }.FirstOrDefault(x => x.Equals("b"));  
  
// Returns "ba":  
new[] { "ba", "be" }.FirstOrDefault(x => x.Contains("b"));  
  
// Returns null:  
new[] { "ca", "ce" }.FirstOrDefault(x => x.Contains("b"));  
  
// Returns null:  
new string[0].FirstOrDefault();
```

[Live Demo on .NET Fiddle](#)

Last()

- Returns the last element of a sequence, or the last element matching the provided predicate.
- If the sequence contains no elements, an InvalidOperationException is thrown with the message "Sequence contains no elements."
- If the sequence contains no elements matching the provided predicate, an InvalidOperationException is thrown with the message "Sequence contains no matching element".

Example

```
// Returns "a":  
new[] { "a" }.Last();  
  
// Returns "b":  
new[] { "a", "b" }.Last();  
  
// Returns "a":  
new[] { "a", "b" }.Last(x => x.Equals("a"));  
  
// Returns "be":  
new[] { "ba", "be" }.Last(x => x.Contains("b"));  
  
// Throws InvalidOperationException:  
new[] { "ca", "ce" }.Last(x => x.Contains("b"));  
  
// Throws InvalidOperationException:  
new string[0].Last();
```

LastOrDefault()

- Returns the last element of a sequence, or the last element matching the provided predicate.
- If the sequence contains no elements, or no elements matching the provided predicate, returns the default value of the sequence type using default(T).

Example

```
// Returns "a":  
new[] { "a" }.LastOrDefault();  
  
// Returns "b":  
new[] { "a", "b" }.LastOrDefault();  
  
// Returns "a":  
new[] { "a", "b" }.LastOrDefault(x => x.Equals("a"));  
  
// Returns "be":
```

```

new[] { "ba", "be" }.LastOrDefault(x => x.Contains("b"));

// 返回 null:
new[] { "ca", "ce" }.LastOrDefault(x => x.Contains("b"));

// 返回 null:
new string[0].LastOrDefault();

```

Single()

- 如果序列恰好包含一个元素，或者恰好包含一个匹配所提供predicate的元素，则返回该元素。
- 如果序列不包含任何元素，或者不包含任何匹配所提供predicate的元素，则抛出带有消息“Sequence contains no elements”的InvalidOperationException异常。
- 如果序列包含多个元素，或者包含多个匹配所提供predicate的元素，则抛出带有消息“Sequence contains more than one element”。
- 注意：为了判断序列是否恰好包含一个元素，最多只需枚举两个元素。**

示例

```

// 返回 "a":
new[] { "a" }.Single();

// 抛出 InvalidOperationException, 因为序列包含多个元素:
new[] { "a", "b" }.Single();

// 返回 "b":
new[] { "a", "b" }.Single(x => x.Equals("b"));

// 抛出 InvalidOperationException 异常:
new[] { "a", "b" }.Single(x => x.Equals("c"));

// 抛出 InvalidOperationException 异常:
new string[0].Single();

// 抛出 InvalidOperationException, 因为序列包含多个元素:
new[] { "a", "a" }.Single();

```

SingleOrDefault()

- 如果序列恰好包含一个元素，或者恰好包含一个匹配所提供predicate的元素，则返回该元素。
- 如果序列不包含任何元素，或者没有元素匹配提供的predicate，则返回default(T)。
- 如果序列包含多个元素，或者包含多个匹配所提供predicate的元素，则抛出带有消息“Sequence contains more than one element”。
- 如果序列中不包含与提供的谓词匹配的元素，则使用default(T)返回序列类型的默认值。
- 注意：为了判断序列是否恰好包含一个元素，最多只需枚举两个元素。**

示例

```

// 返回 "a":
new[] { "a" }.SingleOrDefault();

```

```

new[] { "ba", "be" }.LastOrDefault(x => x.Contains("b"));

// Returns null:
new[] { "ca", "ce" }.LastOrDefault(x => x.Contains("b"));

// Returns null:
new string[0].SingleOrDefault();

```

Single()

- If the sequence contains exactly one element, or exactly one element matching the provided predicate, that element is returned.
- If the sequence contains no elements, or no elements matching the provided predicate, an InvalidOperationException is thrown with the message "Sequence contains no elements".
- If the sequence contains more than one element, or more than one element matching the provided predicate, an InvalidOperationException is thrown with the message "Sequence contains more than one element".
- Note:** in order to evaluate whether the sequence contains exactly one element, at most two elements has to be enumerated.

Example

```

// Returns "a":
new[] { "a" }.Single();

// Throws InvalidOperationException because sequence contains more than one element:
new[] { "a", "b" }.Single();

// Returns "b":
new[] { "a", "b" }.Single(x => x.Equals("b"));

// Throws InvalidOperationException:
new[] { "a", "b" }.Single(x => x.Equals("c"));

// Throws InvalidOperationException:
new string[0].Single();

// Throws InvalidOperationException because sequence contains more than one element:
new[] { "a", "a" }.Single();

```

SingleOrDefault()

- If the sequence contains exactly one element, or exactly one element matching the provided predicate, that element is returned.
- If the sequence contains no elements, or no elements matching the provided predicate, default(T) is returned.
- If the sequence contains more than one element, or more than one element matching the provided predicate, an InvalidOperationException is thrown with the message "Sequence contains more than one element".
- If the sequence contains no elements matching the provided predicate, returns the default value of the sequence type using default(T).
- Note:** in order to evaluate whether the sequence contains exactly one element, at most two elements has to be enumerated.

Example

```

// Returns "a":
new[] { "a" }.SingleOrDefault();

```

```
// 返回 "a"
new[] { "a", "b" }.SingleOrDefault(x => x == "a");

// 返回 null:
new[] { "a", "b" }.SingleOrDefault(x => x == "c");

// 抛出 InvalidOperationException 异常:
new[] { "a", "a" }.SingleOrDefault(x => x == "a");

// 抛出 InvalidOperationException 异常:
new[] { "a", "b" }.SingleOrDefault();

// 返回 null:
new string[0].SingleOrDefault();
```

建议

- 虽然你可以使用FirstOrDefault、LastOrDefault或SingleOrDefault来检查序列中是否包含任何元素，Any或Count更为可靠。原因是这三种方法返回的default(T)值并不能证明序列为空，因为序列的第一个/最后一个/唯一元素的值也可能是default(T)
- 根据你的代码目的决定使用哪种方法。例如，仅当你必须确保集合中有且只有一个元素满足谓词时，才使用Single；否则使用First；因为如果序列中有多个匹配元素，Single会抛出异常。当然，这同样适用于其“*OrDefault”对应方法。
- 关于效率：虽然通常确保查询只返回一个项 (Single) 或仅返回一个或零个项 (SingleOrDefault) 是合适的，但这两种方法都需要检查更多，通常是整个集合，以确保查询中没有第二个匹配项。这与例如First方法的行为不同，后者在找到第一个匹配项后即可满足条件。

```
// returns "a"
new[] { "a", "b" }.SingleOrDefault(x => x == "a");

// Returns null:
new[] { "a", "b" }.SingleOrDefault(x => x == "c");

// Throws InvalidOperationException:
new[] { "a", "a" }.SingleOrDefault(x => x == "a");

// Throws InvalidOperationException:
new[] { "a", "b" }.SingleOrDefault();

// Returns null:
new string[0].SingleOrDefault();
```

Recommendations

- Although you can use FirstOrDefault, LastOrDefault or SingleOrDefault to check whether a sequence contains any items, Any or Count are more reliable. This is because a return value of `default(T)` from one of these three methods doesn't prove that the sequence is empty, as the value of the first / last / single element of the sequence could equally be `default(T)`
- Decide on which methods fits your code's purpose the most. For instance, use Single only if you must ensure that there is a single item in the collection matching your predicate — otherwise use First; as Single throw an exception if the sequence has more than one matching element. This of course applies to the “*OrDefault”-counterparts as well.
- Regarding efficiency: Although it's often appropriate to ensure that there is only one item (Single) or, either only one or zero (SingleOrDefault) items, returned by a query, both of these methods require more, and often the entirety, of the collection to be examined to ensure there is no second match to the query. This is unlike the behavior of, for example, the First method, which can be satisfied after finding the first match.

第66.3节：除外

Except 方法返回包含在第一个集合中但不包含在第二个集合中的项集合。默认使用IEqualityComparer来比较两个集合中的项。该方法有一个重载，接受一个IEqualityComparer作为参数。

示例：

```
int[] first = { 1, 2, 3, 4 };
int[] second = { 0, 2, 3, 5 };

IEnumerable<int> inFirstButNotInSecond = first.Except(second);
// inFirstButNotInSecond = { 1, 4 }
```

输出：

1
4

在这种情况下.Except(second)排除包含在数组second中的元素，即2和3（0和5不在其中）
包含在第一个数组中并被跳过）。

Section 66.3: Except

The Except method returns the set of items which are contained in the first collection but are not contained in the second. The default `IEqualityComparer` is used to compare the items within the two sets. There is an overload which accepts an `IEqualityComparer` as an argument.

Example:

```
int[] first = { 1, 2, 3, 4 };
int[] second = { 0, 2, 3, 5 };

IEnumerable<int> inFirstButNotInSecond = first.Except(second);
// inFirstButNotInSecond = { 1, 4 }
```

Output:

1
4

[Live Demo on .NET Fiddle](#)

In this case .Except (second) excludes elements contained in the array second, namely 2 and 3 (0 and 5 are not contained in the first array and are skipped).

注意，Except 表示 Distinct（即去除重复元素）。例如：

```
int[] third = { 1, 1, 1, 2, 3, 4 };

IEnumerable<int> inThirdButNotInSecond = third.Except(second);
// inThirdButNotInSecond = { 1, 4 }
```

输出：

```
1  
4
```

[IEqualityComparer](#) 将允许使用不同的方法来比较元素。注意，[GetHashCode](#) 方法也应被重写，以便对根据 [IEquatable](#) 实现判定为相同的 object 返回相同的哈希码。

使用 [IEquatable](#) 的示例：

```
class Holiday : IEquatable<Holiday>
{
    public string Name { get; set; }

    public bool Equals(Holiday other)
    {
        return Name == other.Name;
    }

    // 当 Equals 返回 true 时，GetHashCode 必须返回相同的值。
    public override int GetHashCode()
    {
        //如果Name字段不为空，则获取其哈希码。
        return Name?.GetHashCode() ?? 0;
    }
}

public class Program
{
    public static void Main()
    {
        List<Holiday> holidayDifference = new List<Holiday>();

        List<Holiday> remoteHolidays = new List<Holiday>
        {
            new Holiday { Name = "圣诞节" },
            new Holiday { Name = "光明节" },
            new Holiday { Name = "斋月" }
        };

        List<Holiday> localHolidays = new List<Holiday>
        {
            new Holiday { Name = "圣诞节" },
            new Holiday { Name = "斋月" }
        };
    }
}
```

Note that Except implies Distinct (i.e., it removes repeated elements). For example:

```
int[] third = { 1, 1, 1, 2, 3, 4 };

IEnumerable<int> inThirdButNotInSecond = third.Except(second);
// inThirdButNotInSecond = { 1, 4 }
```

Output:

```
1  
4
```

[Live Demo on .NET Fiddle](#)

In this case, the elements 1 and 4 are returned only once.

Implementing [IEquatable](#) or providing the function an [IEqualityComparer](#) will allow using a different method to compare the elements. Note that the [GetHashCode](#) method should also be overridden so that it will return an identical hash code for [object](#) that are identical according to the [IEquatable](#) implementation.

Example With IEquatable:

```
class Holiday : IEquatable<Holiday>
{
    public string Name { get; set; }

    public bool Equals(Holiday other)
    {
        return Name == other.Name;
    }

    // GetHashCode must return true whenever Equals returns true.
    public override int GetHashCode()
    {
        //Get hash code for the Name field if it is not null.
        return Name?.GetHashCode() ?? 0;
    }
}

public class Program
{
    public static void Main()
    {
        List<Holiday> holidayDifference = new List<Holiday>();

        List<Holiday> remoteHolidays = new List<Holiday>
        {
            new Holiday { Name = "Xmas" },
            new Holiday { Name = "Hanukkah" },
            new Holiday { Name = "Ramadan" }
        };

        List<Holiday> localHolidays = new List<Holiday>
        {
            new Holiday { Name = "Xmas" },
            new Holiday { Name = "Ramadan" }
        };
    }
}
```

```

holidayDifference = remoteHolidays
    .Except(localHolidays)
    .ToList();

holidayDifference.ForEach(x => Console.WriteLine(x.Name));
}
}

```

输出：

光明节

[.NET Fiddle 在线演示](#)

第66.4节：SelectMany

SelectMany LINQ 方法将一个 `IEnumerable<IEnumerable<T>>` “扁平化”为一个 `IEnumerable<T>`。源 `IEnumerable` 中包含的所有 `IEnumerable` 实例内的 `T` 元素将被合并为一个单一的 `IEnumerable`。

```

var words = new [] { "a,b,c", "d,e", "f" };
var splitAndCombine = words.SelectMany(x => x.Split(','));
// 返回 { "a", "b", "c", "d", "e", "f" }

```

如果你使用一个选择器函数将输入元素转换为序列，结果将是这些序列中的元素逐一返回。

注意，与 `SELECT()` 不同，输出中的元素数量不需要与输入中的元素数量相同。

更多实际示例

```

class 学校
{
    public 学生[] 学生们 { get; set; }
}

class 学生
{
    public string Name { get; set; }
}

var 学校们 = new [] {
    new School(){ Students = new [] { new Student { Name="Bob"}, new Student { Name="Jack"} } },
    new School(){ Students = new [] { new Student { Name="Jim"}, new Student { Name="John"} } }
};

var allStudents = schools.SelectMany(s=> s.Students);

foreach(var student in allStudents)
{
    Console.WriteLine(student.Name);
}

```

输出：

Bob

```

holidayDifference = remoteHolidays
    .Except(localHolidays)
    .ToList();

```

```

holidayDifference.ForEach(x => Console.WriteLine(x.Name));
}
}

```

Output:

Hanukkah

[Live Demo on .NET Fiddle](#)

Section 66.4: SelectMany

The `SelectMany` linq method 'flattens' an `IEnumerable<IEnumerable<T>>` into an `IEnumerable<T>`. All of the `T` elements within the `IEnumerable` instances contained in the source `IEnumerable` will be combined into a single `IEnumerable`.

```

var words = new [] { "a,b,c", "d,e", "f" };
var splitAndCombine = words.SelectMany(x => x.Split(','));
// returns { "a", "b", "c", "d", "e", "f" }

```

If you use a selector function which turns input elements into sequences, the result will be the elements of those sequences returned one by one.

Note that, unlike `SELECT()`, the number of elements in the output doesn't need to be the same as were in the input.

More real-world example

```

class School
{
    public Student[] Students { get; set; }
}

class Student
{
    public string Name { get; set; }
}

var schools = new [] {
    new School(){ Students = new [] { new Student { Name="Bob"}, new Student { Name="Jack"} } },
    new School(){ Students = new [] { new Student { Name="Jim"}, new Student { Name="John"} } }
};

var allStudents = schools.SelectMany(s=> s.Students);

foreach(var student in allStudents)
{
    Console.WriteLine(student.Name);
}

```

Output:

Bob

Jack
Jim
John

[.NET Fiddle 在线演示](#)

第66.5节 : Any

Any 用于检查集合中的任意元素是否满足某个条件。

另见：*All*、*Any* 和 *FirstOrDefault*：最佳实践

1. 空参数

Any：如果集合中有任何元素则返回true，如果集合为空则返回false：

```
var numbers = new List<int>();
bool result = numbers.Any(); // false

var numbers = new List<int>(){ 1, 2, 3, 4, 5};
bool result = numbers.Any(); // true
```

2. 作为参数的 Lambda 表达式

Any: 如果集合中有一个或多个元素满足 Lambda 表达式中的条件，则返回true：

```
var arrayOfStrings = new string[] { "a", "b", "c" };
arrayOfStrings.Any(item => item == "a"); // true
arrayOfStrings.Any(item => item == "d"); // false
```

3. 空集合

Any: 如果集合为空且提供了 Lambda 表达式，则返回false：

```
var numbers = new List<int>();
bool result = numbers.Any(i => i >= 0); // false
```

注意：Any 会在找到第一个满足条件的元素后立即停止遍历集合。这意味着集合不一定会被完全枚举；它只会被枚举到找到第一个满足条件的元素为止。

[.NET Fiddle 在线演示](#)

第66.6节：连接 (JOINS)

连接用于通过公共键合并持有数据的不同列表或表。

与 SQL 中类似，LINQ 支持以下几种连接类型：

内连接、左连接、右连接、交叉连接和全外连接。

以下两个列表用于下面的示例：

```
var first = new List<string>(){ "a", "b", "c"}; // 左侧数据
var second = new List<string>(){ "a", "c", "d"}; // 右侧数据
```

Jack
Jim
John

[Live Demo on .NET Fiddle](#)

Section 66.5: Any

Any is used to check if **any** element of a collection matches a condition or not.

see also: *All*, *Any* and *FirstOrDefault*: best practice

1. Empty parameter

Any: Returns **true** if the collection has any elements and **false** if the collection is empty:

```
var numbers = new List<int>();
bool result = numbers.Any(); // false

var numbers = new List<int>(){ 1, 2, 3, 4, 5};
bool result = numbers.Any(); // true
```

2. Lambda expression as parameter

Any: Returns **true** if the collection has one or more elements that meet the condition in the lambda expression:

```
var arrayOfStrings = new string[] { "a", "b", "c" };
arrayOfStrings.Any(item => item == "a"); // true
arrayOfStrings.Any(item => item == "d"); // false
```

3. Empty collection

Any: Returns **false** if the collection is empty and a lambda expression is supplied:

```
var numbers = new List<int>();
bool result = numbers.Any(i => i >= 0); // false
```

Note: Any will stop iteration of the collection as soon as it finds an element matching the condition. This means that the collection will not necessarily be fully enumerated; it will only be enumerated far enough to find the first item matching the condition.

[Live Demo on .NET Fiddle](#)

Section 66.6: JOINS

Joins are used to combine different lists or tables holding data via a common key.

Like in SQL, the following kinds of Joins are supported in LINQ:

Inner, Left, Right, Cross and Full Outer Joins.

The following two lists are used in the examples below:

```
var first = new List<string>(){ "a", "b", "c"}; // Left data
var second = new List<string>(){ "a", "c", "d"}; // Right data
```

(内) 连接

```
var result = from f in first
    join s in second on f equals s
    select new { f, s };

var result = first.Join(second,
    f => f,
    s => s,
    (f, s) => new { f, s });

// 结果: {"a","a"}
//      {"c", "c"}
```

左外连接

```
var leftOuterJoin = from f in first
    join s in second on f equals s into temp
    from t in temp.DefaultIfEmpty()
    select new { First = f, Second = t};

// 或者也可以这样做：
var leftOuterJoin = from f in first
    from s in second.Where(x => x == f).DefaultIfEmpty()
    select new { First = f, Second = s };

// 结果: {"a","a"}
//      {"b", null}
//      {"c", "c"}
```

```
// 左外连接方法语法
var leftOuterJoinFluentSyntax = first.GroupJoin(second,
    f => f,
    s => s,
    (f, s) => new { First = f, Second = s })
    .SelectMany(temp => temp.Second.DefaultIfEmpty(),
    (f, s) => new { First = f.First, Second = s });
```

右外连接

```
var rightOuterJoin = from s in second
    join f in first on s equals f into temp
    from t in temp.DefaultIfEmpty()
    select new {First=t,Second=s};

// 结果: {"a","a"}
//      {"c", "c"}
//      {null, "d"}
```

笛卡尔积连接

```
var CrossJoin = from f in first
    from s in second
    select new { f, s };

// 结果: {"a","a"}
//      {"a", "c"}
//      {"a", "d"}
//      {"b", "a"}
//      {"b", "c"}
//      {"b", "d"}
```

(Inner) Join

```
var result = from f in first
    join s in second on f equals s
    select new { f, s };

var result = first.Join(second,
    f => f,
    s => s,
    (f, s) => new { f, s });

// Result: {"a", "a"}
//      {"c", "c"}
```

Left outer join

```
var leftOuterJoin = from f in first
    join s in second on f equals s into temp
    from t in temp.DefaultIfEmpty()
    select new { First = f, Second = t};

// Or can also do:
var leftOuterJoin = from f in first
    from s in second.Where(x => x == f).DefaultIfEmpty()
    select new { First = f, Second = s };

// Result: {"a", "a"}
//      {"b", null}
//      {"c", "c"}
```

```
// Left outer join method syntax
var leftOuterJoinFluentSyntax = first.GroupJoin(second,
    f => f,
    s => s,
    (f, s) => new { First = f, Second = s })
    .SelectMany(temp => temp.Second.DefaultIfEmpty(),
    (f, s) => new { First = f.First, Second = s });
```

Right Outer Join

```
var rightOuterJoin = from s in second
    join f in first on s equals f into temp
    from t in temp.DefaultIfEmpty()
    select new {First=t,Second=s};

// Result: {"a", "a"}
//      {"c", "c"}
//      {null, "d"}
```

Cross Join

```
var CrossJoin = from f in first
    from s in second
    select new { f, s };

// Result: {"a", "a"}
//      {"a", "c"}
//      {"a", "d"}
//      {"b", "a"}
//      {"b", "c"}
//      {"b", "d"}
```

```
// {"c", "a"}  
// {"c", "c"}  
// {"c", "d"}
```

全外连接

```
var fullOuterjoin = leftOuterJoin.Union(rightOuterJoin);
```

```
// 结果: {"a", "a"}  
// {"b", null}  
// {"c", "c"}  
// {null, "d"}
```

实际示例

上述示例的数据结构较为简单，便于您专注于技术上理解不同的LINQ连接，但在实际应用中，您会有需要连接的带有多列的表。

在以下示例中，仅使用了一个类Region，实际上您会连接两个或更多具有相同键的不同表（在本例中，first和second通过公共键ID进行连接）。

示例：考虑以下数据结构：

```
public class Region  
{  
    public Int32 ID;  
    public string RegionDescription;  
  
    public Region(Int32 pRegionID, string pRegionDescription=null)  
    {  
        ID = pRegionID; RegionDescription = pRegionDescription;  
    }  
}
```

现在准备数据（即填充数据）：

```
// 左侧数据  
var first = new List<Region>()  
    { new Region(1), new Region(3), new Region(4) };  
// 正确的数据  
var second = new List<Region>()  
    {  
        new Region(1, "东部"), new Region(2, "西部"),  
        new Region(3, "北部"), new Region(4, "南部")  
    };
```

你可以看到在这个例子中first不包含任何区域描述，所以你想从second中连接它们。然后内连接看起来像这样：

```
// 执行内连接  
var result = from f in first  
            join s in second on f.ID equals s.ID  
            select new { f.ID, s.RegionDescription };  
  
// 结果: {1, "东部"}  
// {3, "北部"}  
// {4, "南部"}
```

```
// {"c", "a"}  
// {"c", "c"}  
// {"c", "d"}
```

Full Outer Join

```
var fullOuterjoin = leftOuterJoin.Union(rightOuterJoin);
```

```
// Result: {"a", "a"}  
// {"b", null}  
// {"c", "c"}  
// {null, "d"}
```

Practical example

The examples above have a simple data structure so you can focus on understanding the different LINQ joins technically, but in the real world you would have tables with columns you need to join.

In the following example, there is just one class Region used, in reality you would join two or more different tables which hold the same key (in this example first and second are joined via the common key ID).

Example: Consider the following data structure:

```
public class Region  
{  
    public Int32 ID;  
    public string RegionDescription;  
  
    public Region(Int32 pRegionID, string pRegionDescription=null)  
    {  
        ID = pRegionID; RegionDescription = pRegionDescription;  
    }  
}
```

Now prepare the data (i.e. populate with data):

```
// Left data  
var first = new List<Region>()  
    { new Region(1), new Region(3), new Region(4) };  
// Right data  
var second = new List<Region>()  
    {  
        new Region(1, "Eastern"), new Region(2, "Western"),  
        new Region(3, "Northern"), new Region(4, "Southern")  
    };
```

You can see that in this example first doesn't contain any region descriptions so you want to join them from second. Then the inner join would look like:

```
// do the inner join  
var result = from f in first  
            join s in second on f.ID equals s.ID  
            select new { f.ID, s.RegionDescription };  
  
// Result: {1, "Eastern"}  
// {3, "Northern"}  
// {4, "Southern"}
```

这个结果动态创建了匿名对象，这没问题，但我们可以创建了一个合适的类——所以我们可以指定它：不使用`SELECT NEW { f.ID, s.RegionDescription };`，而是说`SELECT NEW Region(f.ID, s.RegionDescription);`，这将返回相同的数据，但会创建Region类型的对象——这将保持与其他对象的兼容性。

[.NET fiddle 上的实时演示](#)

第66.7节：Skip和Take

Skip方法返回一个集合，该集合排除了源集合开头指定数量的元素。排除的元素数量由参数指定。如果集合中的元素少于参数指定的数量，则返回一个空集合。

Take方法返回一个集合，包含源集合开头指定数量的元素。

包含的元素数量由参数指定。如果集合中的元素少于参数指定的数量，则返回的集合将包含与源集合相同的元素。

```
var values = new [] { 5, 4, 3, 2, 1 };

var skipTwo      = values.Skip(2);          // { 3, 2, 1 }
var takeThree    = values.Take(3);          // { 5, 4, 3 }
var skipOneTakeTwo = values.Skip(1).Take(2); // { 4, 3 }
var takeZero     = values.Take(0);          // 一个包含0个元素的IEnumerable<int>
```

[.NET Fiddle 在线演示](#)

Skip和Take通常一起使用来实现结果的分页，例如：

```
IEnumerable<T> GetPage<T>(IEnumerable<T> collection, int pageNumber, int resultsPerPage) {
    int startIndex = (pageNumber - 1) * resultsPerPage;
    return collection.Skip(startIndex).Take(resultsPerPage);
}
```

警告：LINQ to Entities仅支持对有序查询使用Skip。如果尝试在未排序的情况下使用Skip，将会抛出NotSupportedException，错误信息为“方法'Skip'仅支持LINQ to Entities中排序后的输入。必须先调用'OrderBy'方法，然后才能调用'Skip'方法。”

第66.8节：在Linq查询中定义变量（let关键字）

为了在linq表达式中定义变量，可以使用let关键字。通常这样做是为了存储中间子查询的结果，例如：

```
int[] 数组 = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

var 高于平均值 = from 数字 in 数组
let 平均值 = 数组.Average()
let 平方值 = Math.Pow(数字,2)
where 平方值 > 平均值
select 数字;

Console.WriteLine("数字的平均值是 {0}." , 数组.Average());
foreach (int n in 高于平均值)
```

This result has created anonymous objects on the fly, which is fine, but we have already created a proper class - so we can specify it: Instead of `SELECT NEW { f.ID, s.RegionDescription };` we can say `SELECT NEW Region(f.ID, s.RegionDescription);`, which will return the same data but will create objects of type Region - that will maintain compatibility with the other objects.

[Live demo on .NET fiddle](#)

Section 66.7: Skip and Take

The Skip method returns a collection excluding a number of items from the beginning of the source collection. The number of items excluded is the number given as an argument. If there are less items in the collection than specified in the argument then an empty collection is returned.

The Take method returns a collection containing a number of elements from the beginning of the source collection. The number of items included is the number given as an argument. If there are less items in the collection than specified in the argument then the collection returned will contain the same elements as the source collection.

```
var values = new [] { 5, 4, 3, 2, 1 };

var skipTwo      = values.Skip(2);          // { 3, 2, 1 }
var takeThree    = values.Take(3);          // { 5, 4, 3 }
var skipOneTakeTwo = values.Skip(1).Take(2); // { 4, 3 }
var takeZero     = values.Take(0);          // An IEnumerable<int> with 0 items
```

[Live Demo on .NET Fiddle](#)

Skip and Take通常一起使用来实现结果的分页，例如：

```
IEnumerable<T> GetPage<T>(IEnumerable<T> collection, int pageNumber, int resultsPerPage) {
    int startIndex = (pageNumber - 1) * resultsPerPage;
    return collection.Skip(startIndex).Take(resultsPerPage);
}
```

Warning: LINQ to Entities only supports Skip on ordered queries. If you try to use Skip without ordering you will get a **NotSupportedException** with the message "The method 'Skip' is only supported for sorted input in LINQ to Entities. The method 'OrderBy' must be called before the method 'Skip'."

Section 66.8: Defining a variable inside a Linq query (let keyword)

In order to define a variable inside a linq expression, you can use the let keyword. This is usually done in order to store the results of intermediate sub-queries, for example:

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

var aboveAverages = from number in numbers
let average = numbers.Average()
let nSquared = Math.Pow(number,2)
where nSquared > average
select number;

Console.WriteLine("The average of the numbers is {0}." , numbers.Average());
foreach (int n in aboveAverages)
```

```
{  
Console.WriteLine("查询结果包含数字 {0}，其平方是 {1}." , n, Math.Pow(n,2));  
}
```

输出：

```
数字的平均值是4.5。  
查询结果包含数字3，其平方是9。  
查询结果包括数字4，其平方为16。  
查询结果包含数字5，其平方为25。  
查询结果包含数字6，其平方为36。  
查询结果包含数字7，其平方为49。  
查询结果包含数字8，其平方为64。  
查询结果包含数字9，其平方为81。
```

[查看演示](#)

第66.9节：Zip

Zip扩展方法作用于两个集合。它根据位置将两个序列中的每个元素配对。使用Func实例时，我们用Zip来成对处理两个C#集合中的元素。如果序列大小不同，较大序列的多余元素将被忽略。

举《C#简明教程》一书中的例子，

```
int[] numbers = { 3, 5, 7 };  
string[] words = { "three", "five", "seven", "ignored" };  
IEnumerable<string> zip = numbers.Zip(words, (n, w) => n + "=" + w);
```

输出：

```
3=三  
5=五  
7=七
```

[查看演示](#)

第66.10节：范围与重复

Enumerable上的Range和Repeat静态方法可用于生成简单序列。

范围

Enumerable.Range()根据起始值和计数生成整数序列。

```
// 生成包含数字1到100的集合 ([1, 2, 3, ..., 98, 99, 100])  
var range = Enumerable.Range(1,100);
```

[.NET Fiddle 在线演示](#)

重复

```
{  
Console.WriteLine("Query result includes number {0} with square of {1}." , n, Math.Pow(n,2));  
}
```

Output:

```
The average of the numbers is 4.5.  
Query result includes number 3 with square of 9.  
Query result includes number 4 with square of 16.  
Query result includes number 5 with square of 25.  
Query result includes number 6 with square of 36.  
Query result includes number 7 with square of 49.  
Query result includes number 8 with square of 64.  
Query result includes number 9 with square of 81.
```

[View Demo](#)

Section 66.9: Zip

The Zip extension method acts upon two collections. It pairs each element in the two series together based on position. With a Func instance, we use Zip to handle elements from the two C# collections in pairs. If the series differ in size, the extra elements of the larger series will be ignored.

To take an example from the book "C# in a Nutshell",

```
int[] numbers = { 3, 5, 7 };  
string[] words = { "three", "five", "seven", "ignored" };  
IEnumerable<string> zip = numbers.Zip(words, (n, w) => n + "=" + w);
```

Output:

```
3=three  
5=five  
7=seven
```

[View Demo](#)

Section 66.10: Range and Repeat

The Range and Repeat static methods on Enumerable can be used to generate simple sequences.

Range

Enumerable.Range() generates a sequence of integers given a starting value and a count.

```
// Generate a collection containing the numbers 1-100 ([1, 2, 3, ..., 98, 99, 100])  
var range = Enumerable.Range(1,100);
```

[Live Demo on .NET Fiddle](#)

Repeat

Enumerable.Repeat()根据元素和所需重复次数生成重复元素序列。

```
// 生成包含“a”三次的集合 ([“a”, “a”, “a”])
var repeatedValues = Enumerable.Repeat("a", 3);
```

[.NET Fiddle 在线演示](#)

第66.11节：基础知识

LINQ在查询集合（或数组）方面非常有用。

例如，给定以下示例数据：

```
var classroom = new Classroom
{
    new Student { Name = "Alice", Grade = 97, HasSnack = true },
    new Student { Name = "Bob", Grade = 82, HasSnack = false },
    new Student { Name = "Jimmy", Grade = 71, HasSnack = true },
    new Student { Name = "Greg", Grade = 90, HasSnack = false },
    new Student { Name = "Joe", Grade = 59, HasSnack = false }
}
```

我们可以使用LINQ语法对这些数据进行“查询”。例如，检索今天带零食的所有学生：

```
var studentsWithSnacks = from s in classroom.Students
                           where s.HasSnack
                           select s;
```

或者，检索成绩90分及以上的学生，并且只返回他们的名字，而不是完整的Student对象：

```
var topStudentNames = from s in classroom.Students
                           where s.Grade >= 90
                           select s.Name;
```

LINQ 功能包含两种语法，它们执行相同的功能，性能几乎相同，但写法差异很大。上面示例中的语法称为查询语法。下面的示例则展示了方法语法。返回的数据与上面示例相同，但查询的写法不同。

```
var topStudentNames = classroom.Students
                           .Where(s => s.Grade >= 90)
                           .Select(s => s.Name);
```

第66.12节：全部

All用于检查集合中的所有元素是否都满足某个条件。

另见：[.Any](#)

1. 空参数

All：不允许使用空参数。

2. 作为参数的 Lambda 表达式

Enumerable.Repeat() generates a sequence of repeating elements given an element and the number of repetitions required.

```
// Generate a collection containing "a", three times ([“a”, “a”, “a”])
var repeatedValues = Enumerable.Repeat("a", 3);
```

[Live Demo on .NET Fiddle](#)

Section 66.11: Basics

LINQ is largely beneficial for querying collections (or arrays).

For example, given the following sample data:

```
var classroom = new Classroom
{
    new Student { Name = "Alice", Grade = 97, HasSnack = true },
    new Student { Name = "Bob", Grade = 82, HasSnack = false },
    new Student { Name = "Jimmy", Grade = 71, HasSnack = true },
    new Student { Name = "Greg", Grade = 90, HasSnack = false },
    new Student { Name = "Joe", Grade = 59, HasSnack = false }
}
```

We can "query" on this data using LINQ syntax. For example, to retrieve all students who have a snack today:

```
var studentsWithSnacks = from s in classroom.Students
                           where s.HasSnack
                           select s;
```

Or, to retrieve students with a grade of 90 or above, and only return their names, not the full Student object:

```
var topStudentNames = from s in classroom.Students
                           where s.Grade >= 90
                           select s.Name;
```

The LINQ feature is comprised of two syntaxes that perform the same functions, have nearly identical performance, but are written very differently. The syntax in the example above is called **query syntax**. The following example, however, illustrates **method syntax**. The same data will be returned as in the example above, but the way the query is written is different.

```
var topStudentNames = classroom.Students
                           .Where(s => s.Grade >= 90)
                           .Select(s => s.Name);
```

Section 66.12: All

All is used to check, if all elements of a collection match a condition or not.

see also: [.Any](#)

1. Empty parameter

All: is not allowed to be used with empty parameter.

2. Lambda expression as parameter

All：如果集合中的所有元素都满足 lambda 表达式，则返回true，否则返回false：

```
var numbers = new List<int>(){ 1, 2, 3, 4, 5};  
bool result = numbers.All(i => i < 10); // true  
bool result = numbers.All(i => i >= 3); // false
```

3. 空集合

All：如果集合为空且提供了 lambda 表达式，则返回true：

```
var numbers = new List<int>();  
bool result = numbers.All(i => i >= 0); // true
```

注意：All 会在找到第一个不符合条件的元素时停止对集合的迭代。这意味着集合不一定会被完全枚举；它只会被枚举到找到第一个不符合条件的元素为止。

第66.13节：Aggregate (聚合)

Aggregate 对序列应用累加器函数。

```
int[] intList = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
int sum = intList.Aggregate((prevSum, current) => prevSum + current);  
// sum = 55
```

- 第一步时 prevSum = 1
- 第二步时 prevSum = 第一步时的 prevSum + 2
- 第i步时 prevSum = 第 (i-1) 步时的 prevSum + 数组的第i个元素

```
string[] stringList = { "Hello", "World", "!" };  
string joinedString = stringList.Aggregate((prev, current) => prev + " " + current);  
// joinedString = "Hello World !"
```

Aggregate 的第二个重载还接收一个 seed 参数，表示初始累加器值。它可以用来在不多次迭代集合的情况下计算多个条件。

```
List<int> items = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

对于集合items，我们想要计算

- 总数.Count
- 偶数的数量
- 收集每第四个元素

使用Aggregate可以这样完成：

```
var result = items.Aggregate(new { Total = 0, Even = 0, FourthItems = new List<int>() },  
    (accumulative,item) =>  
    new {  
        Total = accumulative.Total + 1,  
        Even = accumulative.Even + (item % 2 == 0 ? 1 : 0),  
        FourthItems = (accumulative.Total + 1)%4 == 0 ?  
            new List<int>(accumulative.FourthItems) { item } :  
            accumulative.FourthItems  
    });  
// 结果：
```

All: Returns true if all elements of collection satisfies the lambda expression and false otherwise:

```
var numbers = new List<int>(){ 1, 2, 3, 4, 5};  
bool result = numbers.All(i => i < 10); // true  
bool result = numbers.All(i => i >= 3); // false
```

3. Empty collection

All: Returns true if the collection is empty and a lambda expression is supplied:

```
var numbers = new List<int>();  
bool result = numbers.All(i => i >= 0); // true
```

Note: All will stop iteration of the collection as soon as it finds an element not matching the condition. This means that the collection will not necessarily be fully enumerated; it will only be enumerated far enough to find the first item not matching the condition.

Section 66.13: Aggregate

Aggregate Applies an accumulator function over a sequence.

```
int[] intList = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
int sum = intList.Aggregate((prevSum, current) => prevSum + current);  
// sum = 55
```

- At the first step prevSum = 1
- At the second prevSum = prevSum(at the first step) + 2
- At the i-th step prevSum = prevSum(at the (i-1) step) + i-th element of the array

```
string[] stringList = { "Hello", "World", "!" };  
string joinedString = stringList.Aggregate((prev, current) => prev + " " + current);  
// joinedString = "Hello World !"
```

A second overload of Aggregate also receives an seed parameter which is the initial accumulator value. This can be used to calculate multiple conditions on a collection without iterating it more than once.

```
List<int> items = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

For the collection of items we want to calculate

- The total .Count
- The amount of even numbers
- Collect each forth item

Using Aggregate it can be done like this:

```
var result = items.Aggregate(new { Total = 0, Even = 0, FourthItems = new List<int>() },  
    (accumulative,item) =>  
    new {  
        Total = accumulative.Total + 1,  
        Even = accumulative.Even + (item % 2 == 0 ? 1 : 0),  
        FourthItems = (accumulative.Total + 1)%4 == 0 ?  
            new List<int>(accumulative.FourthItems) { item } :  
            accumulative.FourthItems  
    });  
// Result:
```

```
// 总计 = 12  
// 偶数 = 6  
// 第四项 = [4, 8, 12]
```

注意，使用匿名类型作为种子时，必须为每个项实例化一个新对象，因为属性是只读的。使用自定义类时，可以直接赋值信息，无需new（仅在提供初始seed参数时需要）。

第66.14节：Distinct（去重）

从IEnumerable中返回唯一值。唯一性由默认的相等比较器确定。

```
int[] 数组 = { 1, 2, 3, 4, 2, 5, 3, 1, 2 };  
  
var distinct = 数组.Distinct();  
// distinct = { 1, 2, 3, 4, 5 }
```

要比较自定义数据类型，需要实现IEquatable<T>接口，并为该类型提供GetHashCode和Equals方法。或者可以重写相等比较器：

```
class SSNEqualityComparer : IEqualityComparer<Person> {  
    public bool Equals(Person a, Person b) => return a.SSN == b.SSN;  
    public int GetHashCode(Person p) => p.SSN;  
}  
  
List<Person> people;  
  
distinct = people.Distinct(SSNEqualityComparer);
```

第66.15节：SelectMany：扁平化序列的序列

```
var sequenceOfSequences = new [] { new [] { 1, 2, 3 }, new [] { 4, 5 }, new [] { 6 } };  
var sequence = sequenceOfSequences.SelectMany(x => x);  
// 返回 { 1, 2, 3, 4, 5, 6 }
```

如果你有一个序列的序列，或者你正在创建一个序列的序列，但你想要的结果是一个长序列，则使用SelectMany()。

在LINQ查询语法中：

```
var sequence = from subSequence in sequenceOfSequences  
               from item in subSequence  
               select item;
```

如果你有一个集合的集合，并且希望能够同时处理父集合和子集合中的数据，也可以使用SelectMany来实现。

让我们定义简单的类

```
public class BlogPost  
{  
    public int Id { get; set; }  
    public string Content { get; set; }  
    public List<Comment> Comments { get; set; }  
}
```

```
// Total = 12  
// Even = 6  
// FourthItems = [4, 8, 12]
```

Note that using an anonymous type as the seed one has to instantiate a new object each item because the properties are read only. Using a custom class one can simply assign the information and no new is needed (only when giving the initial seed parameter

Section 66.14: Distinct

Returns unique values from an IEnumerable. Uniqueness is determined using the default equality comparer.

```
int[] array = { 1, 2, 3, 4, 2, 5, 3, 1, 2 };  
  
var distinct = array.Distinct();  
// distinct = { 1, 2, 3, 4, 5 }
```

To compare a custom data type, we need to implement the IEquatable<T> interface and provide GetHashCode and Equals methods for the type. Or the equality comparer may be overridden:

```
class SSNEqualityComparer : IEqualityComparer<Person> {  
    public bool Equals(Person a, Person b) => return a.SSN == b.SSN;  
    public int GetHashCode(Person p) => p.SSN;  
}  
  
List<Person> people;  
  
distinct = people.Distinct(SSNEqualityComparer);
```

Section 66.15: SelectMany: Flattening a sequence of sequences

```
var sequenceOfSequences = new [] { new [] { 1, 2, 3 }, new [] { 4, 5 }, new [] { 6 } };  
var sequence = sequenceOfSequences.SelectMany(x => x);  
// returns { 1, 2, 3, 4, 5, 6 }
```

Use SelectMany() if you have, or you are creating a sequence of sequences, but you want the result as one long sequence.

In LINQ Query Syntax:

```
var sequence = from subSequence in sequenceOfSequences  
               from item in subSequence  
               select item;
```

If you have a collection of collections and would like to be able to work on data from parent and child collection at the same time, it is also possible with SelectMany.

Let's define simple classes

```
public class BlogPost  
{  
    public int Id { get; set; }  
    public string Content { get; set; }  
    public List<Comment> Comments { get; set; }  
}
```

```
public class Comment
{
    public int Id { get; set; }
    public string Content { get; set; }
}
```

假设我们有以下集合。

```
List<BlogPost> posts = new List<BlogPost>()
{
    new BlogPost()
    {
        Id = 1,
        Comments = new List<Comment>()
        {
            new Comment()
            {
                Id = 1,
                Content = "真的很棒！",
            },
            new Comment()
            {
                Id = 2,
                Content = "很酷的帖子！"
            }
        },
        new BlogPost()
        {
            Id = 2,
            Comments = new List<Comment>()
            {
                new Comment()
                {
                    Id = 3,
                    Content = "我不认为你是对的",
                },
                new Comment()
                {
                    Id = 4,
                    Content = "这篇帖子完全是胡说八道"
                }
            }
        }
    };
};
```

现在我们想选择与该评论相关的BlogPost的评论内容Content和Id。为此，我们可以使用合适的SelectMany重载。

```
var commentsWithIds = posts.SelectMany(p => p.Comments, (post, comment) => new { PostId = post.Id, CommentContent = comment.Content });
```

我们的 commentsWithIds 看起来是这样的

```
{
PostId = 1,
CommentContent = "真的很棒！"
},
{
PostId = 1,
```

```
public class Comment
{
    public int Id { get; set; }
    public string Content { get; set; }
}
```

Let's assume we have following collection.

```
List<BlogPost> posts = new List<BlogPost>()
{
    new BlogPost()
    {
        Id = 1,
        Comments = new List<Comment>()
        {
            new Comment()
            {
                Id = 1,
                Content = "It's really great!",
            },
            new Comment()
            {
                Id = 2,
                Content = "Cool post!"
            }
        },
        new BlogPost()
        {
            Id = 2,
            Comments = new List<Comment>()
            {
                new Comment()
                {
                    Id = 3,
                    Content = "I don't think you're right",
                },
                new Comment()
                {
                    Id = 4,
                    Content = "This post is a complete nonsense"
                }
            }
        }
    };
};
```

Now we want to select comments Content along with Id of BlogPost associated with this comment. In order to do so, we can use appropriate SelectMany overload.

```
var commentsWithIds = posts.SelectMany(p => p.Comments, (post, comment) => new { PostId = post.Id, CommentContent = comment.Content });
```

Our commentsWithIds looks like this

```
{
PostId = 1,
CommentContent = "It's really great!"
},
{
PostId = 1,
```

```
CommentContent = "很酷的帖子！"
},
{
PostId = 2,
CommentContent = "我觉得你不对"
},
{
PostId = 2,
CommentContent = "这篇帖子完全是胡说八道"
}
```

```
CommentContent = "Cool post!"
},
{
PostId = 2,
CommentContent = "I don't think you're right"
},
{
PostId = 2,
CommentContent = "This post is a complete nonsense"
}
```

第66.16节：GroupBy

GroupBy 是一种将 `IEnumerable<T>` 项目集合分组为不同组的简便方法。

简单示例

在这个第一个示例中，我们最终得到两个组，奇数项和偶数项。

```
List<int> iList = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var grouped = iList.GroupBy(x => x % 2 == 0);

//将 iList 分组为奇数 [13579] 和偶数 [2468] 项

foreach(var group in grouped)
{
    foreach (int item in group)
    {
        Console.WriteLine(item); // 135792468 (先奇数后偶数)
    }
}
```

更复杂的示例

我们以按年龄对一组人进行分组为例。首先，我们将创建一个 `Person` 对象，它有两个属性，`Name` 和 `Age`。

```
public class Person
{
    public int Age {get; set;}
    public string Name {get; set;}
}
```

然后我们创建了一个包含各种姓名和年龄的样本人名单。

```
List<Person> people = new List<Person>();
people.Add(new Person{Age = 20, Name = "Mouse"});
people.Add(new Person{Age = 30, Name = "Neo"});
people.Add(new Person{Age = 40, Name = "Morpheus"});
people.Add(new Person{Age = 30, Name = "Trinity"});
people.Add(new Person{Age = 40, Name = "Dozer"});
people.Add(new Person{Age = 40, Name = "Smith"});
```

然后我们创建一个 LINQ 查询，将人员列表按年龄分组。

```
var query = people.GroupBy(x => x.Age);
```

Section 66.16: GroupBy

GroupBy is an easy way to sort a `IEnumerable<T>` collection of items into distinct groups.

Simple Example

In this first example, we end up with two groups, odd and even items.

```
List<int> iList = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var grouped = iList.GroupBy(x => x % 2 == 0);

//Groups iList into odd [13579] and even[2468] items

foreach(var group in grouped)
{
    foreach (int item in group)
    {
        Console.WriteLine(item); // 135792468 (first odd then even)
    }
}
```

More Complex Example

Let's take grouping a list of people by age as an example. First, we'll create a `Person` object which has two properties, `Name` and `Age`.

```
public class Person
{
    public int Age {get; set;}
    public string Name {get; set;}
}
```

Then we create our sample list of people with various names and ages.

```
List<Person> people = new List<Person>();
people.Add(new Person{Age = 20, Name = "Mouse"});
people.Add(new Person{Age = 30, Name = "Neo"});
people.Add(new Person{Age = 40, Name = "Morpheus"});
people.Add(new Person{Age = 30, Name = "Trinity"});
people.Add(new Person{Age = 40, Name = "Dozer"});
people.Add(new Person{Age = 40, Name = "Smith"});
```

Then we create a LINQ query to group our list of people by age.

```
var query = people.GroupBy(x => x.Age);
```

这样，我们可以看到每个分组的年龄，并且拥有该组中每个人的列表。

```
foreach(var result in query)
{
    Console.WriteLine(result.Key);

        foreach(var person in result)
            Console.WriteLine(person.Name);
}
```

这将产生以下输出：

```
20
Mouse
30
Neo
三位一体
40
墨菲斯
推土机
史密斯
```

您可以在 [.NET Fiddle 上试玩实时演示](#)

第66.17节：按类型查询集合 / 将元素转换为类型

```
接口 IFoo { }
类 Foo : IFoo { }
类 Bar : IFoo { }
```

```
var item0 = new Foo();
var item1 = new Foo();
var item2 = new Bar();
var item3 = new Bar();
var collection = new IFoo[] { item0, item1, item2, item3 };
```

使用OfType

```
var foos = collection.OfType<Foo>(); // 结果：包含 item0 和 item1 的 IEnumarable<Foo>
var bars = collection.OfType<Bar>(); // 结果：包含 item2 和 item3 的 IEnumarable<Bar>
var foosAndBars = collection.OfType<IFoo>(); // 结果：包含所有四个元素的 IEnumarable<IFoo>
```

使用 Where

```
var foos = collection.Where(item => item is Foo); // 结果：包含 item0 和 item1 的 IEnumarable<IFoo>
var bars = collection.Where(item => item is Bar); // 结果：包含 item2 和 item3 的 IEnumarable<IFoo>
```

使用 Cast

```
var bars = collection.Cast<Bar>(); // 在第一个元素处抛出 InvalidCastException
var foos = collection.Cast<Foo>(); // 在第三个元素处抛出 InvalidCastException
var foosAndBars = collection.Cast<IFoo>(); // 正常
```

Doing so, we can see the Age for each group, and have a list of each person in the group.

```
foreach(var result in query)
{
    Console.WriteLine(result.Key);

        foreach(var person in result)
            Console.WriteLine(person.Name);
}
```

This results in the following output:

```
20
Mouse
30
Neo
三位一体
40
墨菲斯
推土机
史密斯
```

You can play with the [live demo on .NET Fiddle](#)

Section 66.17: Query collection by type / cast elements to type

```
interface IFoo { }
class Foo : IFoo { }
class Bar : IFoo { }
```

```
var item0 = new Foo();
var item1 = new Foo();
var item2 = new Bar();
var item3 = new Bar();
var collection = new IFoo[] { item0, item1, item2, item3 };
```

Using OfType

```
var foos = collection.OfType<Foo>(); // result: IEnumarable<Foo> with item0 and item1
var bars = collection.OfType<Bar>(); // result: IEnumarable<Bar> item item2 and item3
var foosAndBars = collection.OfType<IFoo>(); // result: IEnumarable<IFoo> with all four items
```

Using Where

```
var foos = collection.Where(item => item is Foo); // result: IEnumarable<IFoo> with item0 and item1
var bars = collection.Where(item => item is Bar); // result: IEnumarable<IFoo> with item2 and item3
```

Using Cast

```
var bars = collection.Cast<Bar>(); // throws InvalidCastException on the 1st item
var foos = collection.Cast<Foo>(); // throws InvalidCastException on the 3rd item
var foosAndBars = collection.Cast<IFoo>(); // OK
```

第 66.18 节：枚举 Enumerable

IEnumerable<T> 接口是所有泛型枚举器的基础接口，是理解 LINQ 的核心部分。它的核心是表示序列。

所有泛型集合，如 [Collection<T>](#)、[Array](#)、[List<T>](#)、[Dictionary< TKey, TValue > Class](#) 和 [HashSet<T>](#)，均继承自该底层接口。

除了表示序列外，任何继承自 IEnumerable<T> 的类都必须提供一个 IEnumerator<T>。枚举器暴露了可枚举对象的迭代器，这两个相互关联的接口和概念构成了“枚举可枚举对象”这一说法的来源。

“枚举可枚举对象”是一个重要的短语。可枚举对象仅仅是一个迭代结构，它不包含任何具体的对象。例如，在排序时，可枚举对象可能包含排序字段的条件，但调用 .OrderBy() 本身会返回一个只知道 如何 排序的 IEnumerable<T>。调用会将对象具体化的操作，如遍历集合，称为枚举（例如 .ToList()）。枚举过程会使用可枚举对象中定义的 如何 遍历序列，并返回相关对象（按顺序、过滤、投影等）。

只有当可枚举对象被枚举时，才会导致对象的具体化，这时才能测量诸如时间复杂度（与序列大小相关的执行时间）和空间复杂度（与序列大小相关的空间使用量）等指标。

根据需要枚举的底层序列，创建继承自 IEnumerable<T> 的自定义类可能会有些复杂。一般来说，最好使用现有的泛型集合。不过，也可以继承 IEnumerable<T> 接口，而不必有一个定义好的数组作为底层结构。

例如，使用斐波那契数列作为底层序列。注意，调用 Where 只是构建了一个 IEnumerable，只有在调用枚举该可枚举对象时，才会具体化任何值。

```
void Main()
{
    Fibonacci Fibo = new Fibonacci();
    IEnumerable<long> quadrillionplus = Fibo.Where(i => i > 0000000000000000);
    Console.WriteLine("Enumerable built");
    Console.WriteLine(quadrillionplus.Take(2).Sum());
    Console.WriteLine(quadrillionplus.Skip(2).First());

    IEnumerable<long> fibMod612 = Fibo.OrderBy(i => i % 612);
    Console.WriteLine("Enumerable built");
    Console.WriteLine(fibMod612.First());//最小能被612整除的数
}

public class Fibonacci : IEnumerable<long>
{
    private int max = 90;

    //枚举器通常由foreach调用
    public IEnumerator GetEnumerator() {
        long n0 = 1;
        long n1 = 1;
        Console.WriteLine("正在枚举可枚举对象");
        for(int i = 0; i < max; i++){
            yield return n0+n1;
            n1 += n0;
            n0 = n1-n0;
        }
    }
}
```

Section 66.18: Enumerating the Enumerable

The IEnumerable<T> interface is the base interface for all generic enumerators and is a quintessential part of understanding LINQ. At its core, it represents the sequence.

This underlying interface is inherited by all of the generic collections, such as [Collection<T>](#), [Array](#), [List<T>](#), [Dictionary< TKey, TValue > Class](#), and [HashSet<T>](#).

In addition to representing the sequence, any class that inherits from IEnumerable<T> must provide an IEnumerator<T>. The enumerator exposes the iterator for the enumerable, and these two interconnected interfaces and ideas are the source of the saying "enumerate the enumerable".

"Enumerating the enumerable" is an important phrase. The enumerable is simply a structure for how to iterate, it does not hold any materialized objects. For example, when sorting, an enumerable may hold the criteria of the field to sort, but using .OrderBy() in itself will return an IEnumerable<T> which only knows *how* to sort. Using a call which will materialize the objects, as in iterate the set, is known as enumerating (for example .ToList()). The enumeration process will use the the enumerable definition of *how* in order to move through the series and return the relevant objects (in order, filtered, projected, etc.).

Only once the enumerable has been enumerated does it cause the materialization of the objects, which is when metrics like [time complexity](#) (how long it should take related to series size) and spacial complexity (how much space it should use related to series size) can be measured.

Creating your own class that inherits from IEnumerable<T> can be a little complicated depending on the underlying series that needs to be enumerable. In general it is best to use one of the existing generic collections. That said, it is also possible to inherit from the IEnumerable<T> interface without having a defined array as the underlying structure.

For example, using the Fibonacci series as the underlying sequence. Note that the call to [Where](#) simply builds an IEnumerable, and it is not until a call to enumerate that enumerable is made that any of the values are materialized.

```
void Main()
{
    Fibonacci Fibo = new Fibonacci();
    IEnumerable<long> quadrillionplus = Fibo.Where(i => i > 10000000000000000);
    Console.WriteLine("Enumerable built");
    Console.WriteLine(quadrillionplus.Take(2).Sum());
    Console.WriteLine(quadrillionplus.Skip(2).First());

    IEnumerable<long> fibMod612 = Fibo.OrderBy(i => i % 612);
    Console.WriteLine("Enumerable built");
    Console.WriteLine(fibMod612.First());//smallest divisible by 612
}

public class Fibonacci : IEnumerable<long>
{
    private int max = 90;

    //Enumerator called typically from foreach
    public IEnumerator GetEnumerator() {
        long n0 = 1;
        long n1 = 1;
        Console.WriteLine("Enumerating the Enumerable");
        for(int i = 0; i < max; i++){
            yield return n0+n1;
            n1 += n0;
            n0 = n1-n0;
        }
    }
}
```

```

        }

    //通常从linq调用的Enumerable
    IEnumerator<long> IEnumerable<long>.GetEnumerator() {
        long n0 = 1;
        long n1 = 1;
        Console.WriteLine("正在枚举可枚举对象");
        for(int i = 0; i < max; i++){
            yield return n0+n1;
            n1 += n0;
            n0 = n1-n0;
        }
    }
}

```

输出

```

已构建Enumerable
正在枚举可枚举对象
4052739537881
正在枚举可枚举对象
4052739537881
已构建Enumerable
正在枚举可枚举对象
14930352

```

第二组 (fibMod612) 的优势在于，尽管我们调用了对整个斐波那契数列进行排序，由于只使用了.First()获取了一个值，时间复杂度为O(n)，因为排序算法执行过程中只需比较1个值。这是因为我们的枚举器只请求了1个值，因此不需要将整个可枚举集合具体化。如果我们使用.Take(5)而不是.First()，枚举器将请求5个值，最多需要具体化5个值。相比于需要对整个集合进行排序然后取前5个值，这一原则节省了大量的执行时间和空间。

```

        }

    //Enumerable called typically from linq
    IEnumerator<long> IEnumerable<long>.GetEnumerator() {
        long n0 = 1;
        long n1 = 1;
        Console.WriteLine("Enumerating the Enumerable");
        for(int i = 0; i < max; i++){
            yield return n0+n1;
            n1 += n0;
            n0 = n1-n0;
        }
    }
}

```

Output

```

Enumerable built
Enumerating the Enumerable
4052739537881
Enumerating the Enumerable
4052739537881
Enumerable built
Enumerating the Enumerable
14930352

```

The strength in the second set (the fibMod612) is that even though we made the call to order our entire set of Fibonacci numbers, since only one value was taken using .First() the time complexity was O(n) as only 1 value needed to be compared during the ordering algorithm's execution. This is because our enumerator only asked for 1 value, and so the entire enumerable did not have to be materialized. Had we used .Take(5) instead of .First() the enumerator would have asked for 5 values, and at most 5 values would need to be materialized. Compared to needing to order an entire set *and then* take the first 5 values, the principle of saves a lot of execution time and space.

第66.19节：使用Range与各种Linq方法

您可以将Enumerable类与Linq查询结合使用，将for循环转换为Linq单行代码。

Select示例

与以下做法相反：

```

var asciiCharacters = new List<char>();
for (var x = 0; x < 256; x++)
{
    asciiCharacters.Add((char)x);
}

```

您可以这样做：

```

var asciiCharacters = Enumerable.Range(0, 256).Select(a => (char)a);

```

示例位置

在此示例中，将生成100个数字并提取其中的偶数

Section 66.19: Using Range with various Linq methods

You can use the Enumerable class alongside Linq queries to convert for loops into Linq one liners.

Select Example

Opposed to doing this:

```

var asciiCharacters = new List<char>();
for (var x = 0; x < 256; x++)
{
    asciiCharacters.Add((char)x);
}

```

You can do this:

```

var asciiCharacters = Enumerable.Range(0, 256).Select(a => (char)a);

```

Where Example

In this example, 100 numbers will be generated and even ones will be extracted

```
var evenNumbers = Enumerable.Range(1, 100).Where(a => a % 2 == 0);
```

第66.20节：Where

返回满足指定谓词的项目子集。

```
List<string> trees = new List<string>{ "橡树", "桦树", "山毛榉", "榆树", "榛树", "枫树" };
```

方法语法

```
// 选择所有名称长度为3的树  
var shortTrees = trees.Where(tree => tree.Length == 3); // 橡树, 榆树
```

查询语法

```
var shortTrees = from tree in trees  
                  where tree.Length == 3  
                  select tree; // 橡树, 榆树
```

第66.21节：使用SelectMany代替嵌套循环

给定两个列表

```
var list1 = new List<string> { "a", "b", "c" };  
var list2 = new List<string> { "1", "2", "3", "4" };
```

如果你想输出所有排列组合，可以使用嵌套循环，如

```
var result = new List<string>();  
foreach (var s1 in list1)  
    foreach (var s2 in list2)  
        result.Add($"{s1}{s2}");
```

使用SelectMany你可以执行相同的操作，如

```
var result = list1.SelectMany(x => list2.Select(y => $"{x}{y}", x, y)).ToList();
```

第66.22节：Contains

MSDN：

通过使用指定的IEqualityComparer<T>确定序列是否包含指定元素

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };  
var result1 = numbers.Contains(4); // true  
var result2 = numbers.Contains(8); // false  
  
List<int> secondNumberCollection = new List<int> { 4, 5, 6, 7 };  
// Note that can use the Intersect method in this case  
var result3 = secondNumberCollection.Where(item => numbers.Contains(item)); // will be true only  
for 4,5
```

使用用户自定义对象：

```
var evenNumbers = Enumerable.Range(1, 100).Where(a => a % 2 == 0);
```

Section 66.20: Where

Returns a subset of items which the specified predicate is true for them.

```
List<string> trees = new List<string>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

Method syntax

```
// Select all trees with name of length 3  
var shortTrees = trees.Where(tree => tree.Length == 3); // Oak, Elm
```

Query syntax

```
var shortTrees = from tree in trees  
                  where tree.Length == 3  
                  select tree; // Oak, Elm
```

Section 66.21: Using SelectMany instead of nested loops

Given 2 lists

```
var list1 = new List<string> { "a", "b", "c" };  
var list2 = new List<string> { "1", "2", "3", "4" };
```

if you want to output all permutations you could use nested loops like

```
var result = new List<string>();  
foreach (var s1 in list1)  
    foreach (var s2 in list2)  
        result.Add($"{s1}{s2}");
```

Using SelectMany you can do the same operation as

```
var result = list1.SelectMany(x => list2.Select(y => $"{x}{y}", x, y)).ToList();
```

Section 66.22: Contains

MSDN:

Determines whether a sequence contains a specified element by using a specified IEqualityComparer<T>

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };  
var result1 = numbers.Contains(4); // true  
var result2 = numbers.Contains(8); // false  
  
List<int> secondNumberCollection = new List<int> { 4, 5, 6, 7 };  
// Note that can use the Intersect method in this case  
var result3 = secondNumberCollection.Where(item => numbers.Contains(item)); // will be true only  
for 4,5
```

Using a user defined object:

```

public class Person
{
    public string Name { get; set; }

List<Person> objects = new List<Person>
{
    new Person { Name = "Nikki" },
    new Person { Name = "Gilad" },
    new Person { Name = "Phil" },
    new Person { Name = "John" }
};

// 使用 Person 的 Equals 方法 - 重写 Equals() 和 GetHashCode() - 否则它
// 将按引用比较，结果为 false
var result4 = objects.Contains(new Person { Name = "Phil" }); // true

```

使用Enumerable.Contains(value, comparer)重载：

```

public class Compare : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name;
    }
    public int GetHashCode(Person codeh)
    {
        return codeh.Name.GetHashCode();
    }
}

var result5 = objects.Contains(new Person { Name = "Phil" }, new Compare()); // true

```

一个聪明的Contains用法是用它来替代多个if语句。

所以，不要这样写：

```

if(status == 1 || status == 3 || status == 4)
{
    //执行一些业务操作
}
否则
{
    //执行其他操作
}

```

执行此操作：

```

if(new int[] {1, 3, 4 }.Contains(status)
{
    //执行一些业务操作
}
否则
{
    //执行其他操作
}

```

```

public class Person
{
    public string Name { get; set; }

List<Person> objects = new List<Person>
{
    new Person { Name = "Nikki" },
    new Person { Name = "Gilad" },
    new Person { Name = "Phil" },
    new Person { Name = "John" }
};

//Using the Person's Equals method - override Equals() and GetHashCode() - otherwise it
//will compare by reference and result will be false
var result4 = objects.Contains(new Person { Name = "Phil" }); // true

```

Using the Enumerable.Contains(value, comparer) overload:

```

public class Compare : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        return x.Name == y.Name;
    }
    public int GetHashCode(Person codeh)
    {
        return codeh.Name.GetHashCode();
    }
}

var result5 = objects.Contains(new Person { Name = "Phil" }, new Compare()); // true

```

A smart usage of Contains would be to replace multiple if clauses to a Contains call.

So instead of doing this:

```

if(status == 1 || status == 3 || status == 4)
{
    //Do some business operation
}
else
{
    //Do something else
}

```

Do this:

```

if(new int[] {1, 3, 4 }.Contains(status)
{
    //Do some business operation
}
else
{
    //Do something else
}

```

第66.23节：连接多个序列

考虑如下实体客户(Customer)、购买(Purchase)和购买项(PurchaseItem)

```
public class 客户
{
    public string Id { get; set } // 唯一标识客户的Id
    public string Name { get; set; }
}

public class 购买
{
    public string Id { get; set }
    public string CustomerId { get; set; }
    public string Description { get; set; }
}

public class PurchaseItem
{
    public string Id { get; set }
    public string PurchaseId { get; set; }
    public string Detail { get; set; }
}
```

考虑以下实体的示例数据：

```
var customers = new List<Customer>()
{
    new Customer() {
        Id = Guid.NewGuid().ToString(),
        Name = "Customer1"
    },
    new Customer() {
        Id = Guid.NewGuid().ToString(),
        Name = "Customer2"
    }
};

var purchases = new List<Purchase>()
{
    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[0].Id,
        Description = "Customer1-Purchase1"
    },
    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[0].Id,
        Description = "Customer1-Purchase2"
    },
    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[1].Id,
        Description = "Customer2-Purchase1"
    },
    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[1].Id,
        Description = "Customer2-Purchase2"
    }
};
```

Section 66.23: Joining multiple sequences

Consider entities Customer, Purchase and PurchaseItem as follows:

```
public class Customer
{
    public string Id { get; set } // A unique Id that identifies customer
    public string Name { get; set; }
}

public class Purchase
{
    public string Id { get; set }
    public string CustomerId { get; set; }
    public string Description { get; set; }
}

public class PurchaseItem
{
    public string Id { get; set }
    public string PurchaseId { get; set; }
    public string Detail { get; set; }
}
```

Consider following sample data for above entities:

```
var customers = new List<Customer>()
{
    new Customer() {
        Id = Guid.NewGuid().ToString(),
        Name = "Customer1"
    },
    new Customer() {
        Id = Guid.NewGuid().ToString(),
        Name = "Customer2"
    }
};

var purchases = new List<Purchase>()
{
    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[0].Id,
        Description = "Customer1-Purchase1"
    },
    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[0].Id,
        Description = "Customer1-Purchase2"
    },
    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[1].Id,
        Description = "Customer2-Purchase1"
    },
    new Purchase() {
        Id = Guid.NewGuid().ToString(),
        CustomerId = customers[1].Id,
        Description = "Customer2-Purchase2"
    }
};
```

```

CustomerId = customers[1].Id,
    Description = "Customer2-Purchase2"
}

};

var purchaseItems = new List<PurchaseItem>()
{
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[0].Id,
        Detail = "Purchase1-PurchaseItem1"
    },
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[1].Id,
        Detail = "Purchase2-PurchaseItem1"
    },
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[1].Id,
        Detail = "Purchase2-PurchaseItem2"
    },
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[3].Id,
        Detail = "Purchase3-PurchaseItem1"
    }
};

```

现在，考虑以下 LINQ 查询：

```

var result = from c in customers
            join p in purchases on c.Id equals p.CustomerId           // 第一次连接
            join pi in purchaseItems on p.Id equals pi.PurchaseId   // 第二次连接
            select new
            {
                c.Name, p.Description, pi.Detail
            };

```

输出上述查询的结果：

```

foreach(var resultItem in result)
{
    Console.WriteLine($"{resultItem.Name}, {resultItem.Description}, {resultItem.Detail}");
}

```

查询的输出结果将是：

```

Customer1, Customer1-Purchase1, Purchase1-PurchaseItem1
Customer1, Customer1-Purchase2, Purchase2-PurchaseItem1
Customer1, Customer1-Purchase2, Purchase2-PurchaseItem2
Customer2, Customer2-Purchase2, Purchase3-PurchaseItem1

```

```

CustomerId = customers[1].Id,
    Description = "Customer2-Purchase2"
}

};

var purchaseItems = new List<PurchaseItem>()
{
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[0].Id,
        Detail = "Purchase1-PurchaseItem1"
    },
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[1].Id,
        Detail = "Purchase2-PurchaseItem1"
    },
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[1].Id,
        Detail = "Purchase2-PurchaseItem2"
    },
    new PurchaseItem() {
        Id = Guid.NewGuid().ToString(),
        PurchaseId= purchases[3].Id,
        Detail = "Purchase3-PurchaseItem1"
    }
};

```

Now, consider below linq query:

```

var result = from c in customers
            join p in purchases on c.Id equals p.CustomerId           // first join
            join pi in purchaseItems on p.Id equals pi.PurchaseId   // second join
            select new
            {
                c.Name, p.Description, pi.Detail
            };

```

To output the result of above query:

```

foreach(var resultItem in result)
{
    Console.WriteLine($"{resultItem.Name}, {resultItem.Description}, {resultItem.Detail}");
}

```

The output of the query would be:

```

Customer1, Customer1-Purchase1, Purchase1-PurchaseItem1
Customer1, Customer1-Purchase2, Purchase2-PurchaseItem1
Customer1, Customer1-Purchase2, Purchase2-PurchaseItem2
Customer2, Customer2-Purchase2, Purchase3-PurchaseItem1

```

第66.24节：基于多个键的连接

```
 PropertyInfo[] stringProps = typeof(string).GetProperties(); //string 属性
 PropertyInfo[] builderProps = typeof(StringBuilder).GetProperties(); //StringBuilder 属性

 var query =
    from s in stringProps
    join b in builderProps
    on new { s.Name, s.PropertyType } equals new { b.Name, b.PropertyType }
    select new
    {
        s.Name,
        s.PropertyType,
        StringTokenizer = s.MetadataToken,
        StringBuilderToken = b.MetadataToken
    };
}
```

注意，上述join中的匿名类型必须包含相同的属性，因为只有当所有属性都相等时，对象才被视为相等。否则查询将无法编译。

第66.25节：ToLookup

ToLookup返回一个允许索引的数据结构。它是一个扩展方法。它生成一个ILookup实例，可以通过索引或使用foreach循环枚举。条目在每个键处被组合成分组。 - dotnetperls

```
string[] array = { "one", "two", "three" };
//使用字符串长度作为键创建查找表
var lookup = array.ToLookup(item => item.Length);

//连接长度为3的值
Console.WriteLine(string.Join(", ", lookup[3]));
//输出：one,two
```

另一个示例：

```
int[] array = { 1,2,3,4,5,6,7,8 };
//生成奇偶数的查找表（键将是0和1）
var lookup = array.ToLookup(item => item % 2);

//连接后打印偶数
Console.WriteLine(string.Join(", ", lookup[0]));
//输出：2,4,6,8

//连接后打印奇数
Console.WriteLine(string.Join(", ", lookup[1]));
//输出：1,3,5,7
```

第66.26节：SkipWhile

SkipWhile() 用于排除元素直到第一个不匹配项（这对大多数人来说可能有些反直觉）

```
int[] list = { 42, 42, 6, 6, 6, 42 };
```

Section 66.24: Joining on multiple keys

```
 PropertyInfo[] stringProps = typeof(string).GetProperties(); //string properties
 PropertyInfo[] builderProps = typeof(StringBuilder).GetProperties(); //StringBuilder properties

 var query =
    from s in stringProps
    join b in builderProps
    on new { s.Name, s.PropertyType } equals new { b.Name, b.PropertyType }
    select new
    {
        s.Name,
        s.PropertyType,
        StringTokenizer = s.MetadataToken,
        StringBuilderToken = b.MetadataToken
    };
}
```

Note that anonymous types in above `join` must contain same properties since objects are considered equal only if all their properties are equal. Otherwise query won't compile.

Section 66.25: ToLookup

ToLookup returns a data structure that allows indexing. It is an extension method. It produces an ILookup instance that can be indexed or enumerated using a foreach-loop. The entries are combined into groupings at each key. - dotnetperls

```
string[] array = { "one", "two", "three" };
//create lookup using string length as key
var lookup = array.ToLookup(item => item.Length);

//join the values whose lengths are 3
Console.WriteLine(string.Join(", ", lookup[3]));
//output: one,two
```

Another Example:

```
int[] array = { 1,2,3,4,5,6,7,8 };
//generate lookup for odd even numbers (keys will be 0 and 1)
var lookup = array.ToLookup(item => item % 2);

//print even numbers after joining
Console.WriteLine(string.Join(", ", lookup[0]));
//output: 2,4,6,8

//print odd numbers after joining
Console.WriteLine(string.Join(", ", lookup[1]));
//output: 1,3,5,7
```

Section 66.26: SkipWhile

`SkipWhile()` is used to exclude elements until first non-match (this might be counter intuitive to most)

```
int[] list = { 42, 42, 6, 6, 6, 42 };
```

```
var 结果 = 列表.SkipWhile(i => i == 42);
// 结果: 6, 6, 6, 42
```

第66.27节：查询排序 - OrderBy() ThenBy() OrderByDescending() ThenByDescending()

```
string[] 名字= { "mark", "steve", "adam" };
```

升序：

查询语法

```
var sortedNames =
    from name in names
    orderby name
    select name;
```

方法语法

```
var sortedNames = names.OrderBy(name => name);
```

sortedNames 包含以下顺序的名字："adam","mark","steve"

降序：

查询语法

```
var sortedNames =
    from name in names
    orderby name descending
    select name;
```

方法语法

```
var sortedNames = names.OrderByDescending(name => name);
```

sortedNames 包含以下顺序的名字："steve","mark","adam"

按多个字段排序

```
人员[] 人员 =
{
    new 人员 { 名字 = "史蒂夫", 姓氏 = "柯林斯", 年龄 = 30},
    new 人员 { 名字 = "菲尔", 姓氏 = "柯林斯", 年龄 = 28},
    new 人员 { 名字 = "亚当", 姓氏 = "阿克曼", 年龄 = 29},
    new 人员 { 名字 = "亚当", 姓氏 = "阿克曼", 年龄 = 15}
};
```

查询语法

```
var 排序人员 = 从 人员 中 人员
按照人员.姓氏, 人员.名字, 人员.年龄 降序
选择 人员;
```

方法语法

```
var result = list.SkipWhile(i => i == 42);
// Result: 6, 6, 6, 42
```

Section 66.27: Query Ordering - OrderBy() ThenBy() OrderByDescending() ThenByDescending()

```
string[] names= { "mark", "steve", "adam" };
```

Ascending:

Query Syntax

```
var sortedNames =
    from name in names
    orderby name
    select name;
```

Method Syntax

```
var sortedNames = names.OrderBy(name => name);
```

sortedNames contains the names in following order: "adam","mark","steve"

Descending:

Query Syntax

```
var sortedNames =
    from name in names
    orderby name descending
    select name;
```

Method Syntax

```
var sortedNames = names.OrderByDescending(name => name);
```

sortedNames contains the names in following order: "steve","mark","adam"

Order by several fields

```
Person[] people =
{
    new Person { FirstName = "Steve", LastName = "Collins", Age = 30},
    new Person { FirstName = "Phil", LastName = "Collins", Age = 28},
    new Person { FirstName = "Adam", LastName = "Ackerman", Age = 29},
    new Person { FirstName = "Adam", LastName = "Ackerman", Age = 15}
};
```

Query Syntax

```
var sortedPeople = from person in people
                    orderby person.LastName, person.FirstName, person.Age descending
                    select person;
```

Method Syntax

```
排序人员 = 人员.按顺序排序(人员 => 人员.姓氏)
    .然后按顺序排序(人员 => 人员.名字)
    .然后按降序排序(人员 => 人员.年龄);
```

结果

1. 亚当 阿克曼 29
2. 亚当 阿克曼 15
3. 菲尔 柯林斯 28
4. 史蒂夫 柯林斯 30

第66.28节：总和

Enumerable.Sum扩展方法计算数值的总和。

如果集合的元素本身是数字，则可以直接计算总和。

```
int[] numbers = new int[] { 1, 4, 6 };
Console.WriteLine( numbers.Sum() ); //输出 11
```

如果元素的类型是复杂类型，可以使用 lambda 表达式指定要计算的值：

```
var totalMonthlySalary = employees.Sum( employee => employee.MonthlySalary );
```

Sum 扩展方法可以计算以下类型：

- Int32
- Int64
- Single
- Double
- Decimal

如果集合包含可空类型，可以使用空合并运算符为 null 元素设置默认值：

```
int?[] numbers = new int?[] { 1, null, 6 };
Console.WriteLine( numbers.Sum( number => number ?? 0 ) ); //输出 7
```

第 66.29 节：按一个或多个字段分组

假设我们有一个电影模型：

```
public class Film {
    public string Title { get; set; }
    public string Category { get; set; }
    public int Year { get; set; }
}
```

按类别属性分组：

```
foreach (var grp in films.GroupBy(f => f.Category)) {
    var groupCategory = grp.Key;
    var numberOfFilmsInCategory = grp.Count();
```

```
sortedPeople = people.OrderBy(person => person.LastName)
    .ThenBy(person => person.FirstName)
    .ThenByDescending(person => person.Age);
```

Result

1. Adam Ackerman 29
2. Adam Ackerman 15
3. Phil Collins 28
4. Steve Collins 30

Section 66.28: Sum

The Enumerable.Sum extension method calculates the sum of numeric values.

In case the collection's elements are themselves numbers, you can calculate the sum directly.

```
int[] numbers = new int[] { 1, 4, 6 };
Console.WriteLine( numbers.Sum() ); //outputs 11
```

In case the type of the elements is a complex type, you can use a lambda expression to specify the value that should be calculated:

```
var totalMonthlySalary = employees.Sum( employee => employee.MonthlySalary );
```

Sum extension method can calculate with the following types:

- Int32
- Int64
- Single
- Double
- Decimal

In case your collection contains nullable types, you can use the null-coalescing operator to set a default value for null elements:

```
int?[] numbers = new int?[] { 1, null, 6 };
Console.WriteLine( numbers.Sum( number => number ?? 0 ) ); //outputs 7
```

Section 66.29: GroupBy one or multiple fields

Lets assume we have some Film model:

```
public class Film {
    public string Title { get; set; }
    public string Category { get; set; }
    public int Year { get; set; }
}
```

Group by Category property:

```
foreach (var grp in films.GroupBy(f => f.Category)) {
    var groupCategory = grp.Key;
    var numberOfFilmsInCategory = grp.Count();
```

```
}
```

按类别和年份分组：

```
foreach (var grp in films.GroupBy(f => new { Category = f.Category, Year = f.Year })) {
    var groupCategory = grp.Key.Category;
    var groupYear = grp.Key.Year;
    var numberofFilmsInCategory = grp.Count();
}
```

第66.30节：OrderBy

按指定值对集合进行排序。

当值是整数、双精度浮点数或浮点数时，它从最小值开始，这意味着你首先得到负值，然后是零，最后是正值（参见示例1）。

当你按字符排序时，该方法比较字符的ASCII值来对集合进行排序（参见示例2）。

当你排序字符串时，OrderBy方法通过查看它们的CultureInfo来比较它们，但通常是从字母表中的第一个字母开始（a,b,c...）。

这种排序方式称为升序，如果你想要相反的顺序，则需要使用降序（参见OrderByDescending）。

示例1：

```
int[] 数组 = {2, 1, 0, -1, -2};
IEnumerable<int> 升序 = 数组.OrderBy(x => x);
// 返回 {-2, -1, 0, 1, 2}
```

示例2：

```
char[] 字符集 = {' ', '!', '?', '[', '{', '+', '1', '9', 'a', 'A', 'b', 'B', 'y', 'Y', 'z', 'Z'};
IEnumerable<char> 升序 = 字符集.OrderBy(x => x);
// 返回 {' ', '!', '+', '1', '9', '?', 'A', 'B', 'Y', 'Z', '[', 'a', 'b', 'y', 'z', ']'}
```

示例：

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

var people = new[]
{
    new 人员 {姓名 = "Alice", 年龄 = 25},
    new 人员 {姓名 = "Bob", 年龄 = 21},
    new 人员 {姓名 = "Carol", 年龄 = 43}
};
var 最年轻的人 = people.OrderBy(x => x.Age).First();
var 名字 = 最年轻的人.姓名; // Bob
```

```
}
```

Group by Category and Year:

```
foreach (var grp in films.GroupBy(f => new { Category = f.Category, Year = f.Year })) {
    var groupCategory = grp.Key.Category;
    var groupYear = grp.Key.Year;
    var numberofFilmsInCategory = grp.Count();
}
```

Section 66.30: OrderBy

Orders a collection by a specified value.

When the value is an **integer**, **double** or **float** it starts with the *minimum value*, which means that you get first the negative values, than zero and afterwards the positive values (see Example 1).

When you order by a **char** the method compares the *ascii values* of the chars to sort the collection (see Example 2).

When you sort **strings** the OrderBy method compares them by taking a look at their [CultureInfo](#) but normally starting with the *first letter* in the alphabet (a,b,c...).

This kind of order is called ascending, if you want it the other way round you need descending (see OrderByDescending).

Example 1:

```
int[] numbers = {2, 1, 0, -1, -2};
IEnumerable<int> ascending = numbers.OrderBy(x => x);
// returns {-2, -1, 0, 1, 2}
```

Example 2:

```
char[] letters = {' ', '!', '?', '[', '{', '+', '1', '9', 'a', 'A', 'b', 'B', 'y', 'Y', 'z', 'Z'};
IEnumerable<char> ascending = letters.OrderBy(x => x);
// returns {' ', '!', '+', '1', '9', '?', 'A', 'B', 'Y', 'Z', '[', 'a', 'b', 'y', 'z', ']'}
```

Example:

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

var people = new[]
{
    new Person {Name = "Alice", Age = 25},
    new Person {Name = "Bob", Age = 21},
    new Person {Name = "Carol", Age = 43}
};
var youngestPerson = people.OrderBy(x => x.Age).First();
var name = youngestPerson.Name; // Bob
```

第66.31节：Any和FirstOrDefault - 最佳实践

我不会解释Any和FirstOrDefault的作用，因为已经有两个很好的示例。详见
Any和First、FirstOrDefault、Last、LastOrDefault、Single和SingleOrDefault了解更多信息。

我经常在代码中看到的一个应该避免的模式是

```
if (myEnumerable.Any(t=>t.Foo == "Bob"))
{
    var myFoo = myEnumerable.First(t=>t.Foo == "Bob");
    //执行操作
}
```

可以更高效地写成这样

```
var myFoo = myEnumerable.FirstOrDefault(t=>t.Foo == "Bob");
if (myFoo != null)
{
    //执行操作
}
```

通过使用第二个示例，集合只被搜索一次，并且给出与第一个示例相同的结果。相同的思路也可以应用于Single。

第66.32节：GroupBy求和与计数

我们来看一个示例类：

```
public class Transaction
{
    public string Category { get; set; }
    public DateTime Date { get; set; }
    public decimal Amount { get; set; }
}
```

现在，让我们考虑一个交易列表：

```
var transactions = new List<Transaction>
{
    new Transaction { Category = "储蓄账户", Amount = 56, Date = DateTime.Today.AddDays(1) },
    new Transaction { Category = "储蓄账户", Amount = 10, Date = DateTime.Today.AddDays(-10) },
    new Transaction { Category = "信用卡", Amount = 15, Date = DateTime.Today.AddDays(1) },
    new Transaction { Category = "信用卡", Amount = 56, Date = DateTime.Today },
    new Transaction { Category = "活期账户", Amount = 100, Date = DateTime.Today.AddDays(5) },
};
```

如果您想计算按类别分类的金额总和和计数，可以使用 GroupBy，如下所示：

```
var summaryApproach1 = transactions.GroupBy(t => t.Category)
    .Select(t => new
    {
        Category = t.Key,
        Count = t.Count(),
        Amount = t.Sum(ta => ta.Amount),
    }).ToList();
```

Section 66.31: Any and FirstOrDefault - best practice

I won't explain what Any and FirstOrDefault does because there are already two good example about them. See Any and First, FirstOrDefault, Last, LastOrDefault, Single, and SingleOrDefault for more information.

A pattern I often see in code which **should be avoided** is

```
if (myEnumerable.Any(t=>t.Foo == "Bob"))
{
    var myFoo = myEnumerable.First(t=>t.Foo == "Bob");
    //Do stuff
}
```

It could be written more efficiently like this

```
var myFoo = myEnumerable.FirstOrDefault(t=>t.Foo == "Bob");
if (myFoo != null)
{
    //Do stuff
}
```

By using the second example, the collection is searched only once and give the same result as the first one. The same idea can be applied to Single.

Section 66.32: GroupBy Sum and Count

Let's take a sample class:

```
public class Transaction
{
    public string Category { get; set; }
    public DateTime Date { get; set; }
    public decimal Amount { get; set; }
}
```

Now, let us consider a list of transactions:

```
var transactions = new List<Transaction>
{
    new Transaction { Category = "Saving Account", Amount = 56, Date = DateTime.Today.AddDays(1) },
    new Transaction { Category = "Saving Account", Amount = 10, Date = DateTime.Today.AddDays(-10) },
    new Transaction { Category = "Credit Card", Amount = 15, Date = DateTime.Today.AddDays(1) },
    new Transaction { Category = "Credit Card", Amount = 56, Date = DateTime.Today },
    new Transaction { Category = "Current Account", Amount = 100, Date = DateTime.Today.AddDays(5) },
};
```

If you want to calculate category wise sum of amount and count, you can use GroupBy as follows:

```
var summaryApproach1 = transactions.GroupBy(t => t.Category)
    .Select(t => new
    {
        Category = t.Key,
        Count = t.Count(),
        Amount = t.Sum(ta => ta.Amount),
    }).ToList();
```

```
Console.WriteLine("-- 汇总：方法1 --");
summaryApproach1.ForEach(
    row => Console.WriteLine($"类别: {row.Category}, 金额: {row.Amount}, 数量:
    {row.Count}"));

```

或者，你也可以一步完成：

```
var summaryApproach2 = transactions.GroupBy(t => t.Category, (key, t) =>
{
    var transactionArray = t as Transaction[] ?? t.ToArray();
    return new
    {
        Category = key,
        Count = transactionArray.Length,
        Amount = transactionArray.Sum(ta => ta.Amount),
    };
}).ToList();

Console.WriteLine("-- 摘要：方法2 --");
summaryApproach2.ForEach(
    row => Console.WriteLine($"类别: {row.Category}, 金额: {row.Amount}, 数量: {row.Count}"));

```

以上两个查询的输出结果相同：

```
类别：储蓄账户，金额：66，数量：2
类别：信用卡，金额：71，数量：2
类别：活期账户，金额：100，数量：1
```

[.NET Fiddle 在线演示](#)

第66.33节：SequenceEqual

SequenceEqual 用于比较两个 I`Enumerable<T>` 序列是否相等。

```
int[] a = new int[] {1, 2, 3};
int[] b = new int[] {1, 2, 3};
int[] c = new int[] {1, 3, 2};

bool returnsTrue = a.SequenceEqual(b);
bool returnsFalse = a.SequenceEqual(c);
```

第66.34节：ElementAt 和 ElementAtOrDefault

ElementAt 将返回索引为 `n` 的项。如果 `n` 不在可枚举范围内，则抛出 `ArgumentOutOfRangeException`。

```
int[] numbers = { 1, 2, 3, 4, 5 };
numbers.ElementAt(2); // 3
numbers.ElementAt(10); // 抛出 ArgumentOutOfRangeException
```

ElementAtOrDefault 将返回索引 `n` 处的元素。如果 `n` 不在可枚举范围内，则返回类型 `T` 的默认值。

```
Console.WriteLine("-- Summary: Approach 1 --");
summaryApproach1.ForEach(
    row => Console.WriteLine($"Category: {row.Category}, Amount: {row.Amount}, Count:
    {row.Count}"));

```

Alternatively, you can do this in one step:

```
var summaryApproach2 = transactions.GroupBy(t => t.Category, (key, t) =>
{
    var transactionArray = t as Transaction[] ?? t.ToArray();
    return new
    {
        Category = key,
        Count = transactionArray.Length,
        Amount = transactionArray.Sum(ta => ta.Amount),
    };
}).ToList();

Console.WriteLine("-- Summary: Approach 2 --");
summaryApproach2.ForEach(
    row => Console.WriteLine($"Category: {row.Category}, Amount: {row.Amount}, Count: {row.Count}"));

```

Output for both the above queries would be same:

```
Category: Saving Account, Amount: 66, Count: 2
Category: Credit Card, Amount: 71, Count: 2
Category: Current Account, Amount: 100, Count: 1
```

[Live Demo in .NET Fiddle](#)

Section 66.33: SequenceEqual

SequenceEqual is used to compare two `IEnumerable<T>` sequences with each other.

```
int[] a = new int[] {1, 2, 3};
int[] b = new int[] {1, 2, 3};
int[] c = new int[] {1, 3, 2};

bool returnsTrue = a.SequenceEqual(b);
bool returnsFalse = a.SequenceEqual(c);
```

Section 66.34: ElementAt and ElementAtOrDefault

ElementAt will return the item at index `n`. If `n` is not within the range of the enumerable, throws an `ArgumentOutOfRangeException`.

```
int[] numbers = { 1, 2, 3, 4, 5 };
numbers.ElementAt(2); // 3
numbers.ElementAt(10); // throws ArgumentOutOfRangeException
```

ElementAtOrDefault will return the item at index `n`. If `n` is not within the range of the enumerable, returns a `default(T)`.

```

int[] numbers = { 1, 2, 3, 4, 5 };
numbers.ElementAtOrDefault(2); // 3
numbers.ElementAtOrDefault(10); // 0 = default(int)

```

ElementAt 和 ElementAtOrDefault 都针对源为 `IList<T>` 的情况进行了优化，这种情况下会使用普通索引访问。

注意，对于 `ElementAt`，如果提供的索引大于 `IList<T>` 的大小，列表应该（但技术上不保证）抛出 `ArgumentOutOfRangeException`。

第66.35节：DefaultIfEmpty

`DefaultIfEmpty` 用于在序列不包含任何元素时返回一个默认元素。该元素可以是类型的默认值，也可以是用户定义的该类型实例。示例：

```

var chars = new List<string>() { "a", "b", "c", "d" };

chars.DefaultIfEmpty("N/A").FirstOrDefault(); // 返回 "a";

chars.Where(str => str.Length > 1)
    .DefaultIfEmpty("N/A").FirstOrDefault(); // 返回 "N/A"

chars.Where(str => str.Length > 1)
    .DefaultIfEmpty().First(); // 返回 null;

```

左连接中的用法：

使用`DefaultIfEmpty`，传统的 Linq Join 如果未找到匹配项，可以返回一个默认对象。因此它的作用类似于 SQL 的左连接。示例：

```

var leftSequence = new List<int>() { 99, 100, 5, 20, 102, 105 };
var rightSequence = new List<char>() { 'a', 'b', 'c', 'i', 'd' };

var numbersAsChars = from l in leftSequence
                     join r in rightSequence
                     on l equals (int)r into leftJoin
                     from result in leftJoin.DefaultIfEmpty('?')
                     select new
                     {
                         Number = l,
                         Character = result
                     };

foreach(var item in numbersAsChars)
{
    Console.WriteLine("Num = {0} ** Char = {1}", item.Number, item.Character);
}

```

输出：

```

数字 = 99      字符 = c
数字 = 100     字符 = d
数字 = 5       字符 = ?
Num = 20       Char = ?
Num = 102      Char = ?
Num = 105      Char = i

```

在使用 `DefaultIfEmpty`（未指定默认值）的情况下，如果右侧序列中没有匹配项，

```

int[] numbers = { 1, 2, 3, 4, 5 };
numbers.ElementAtOrDefault(2); // 3
numbers.ElementAtOrDefault(10); // 0 = default(int)

```

Both `ElementAt` and `ElementAtOrDefault` are optimized for when the source is an `IList<T>` and normal indexing will be used in those cases.

Note that for `ElementAt`, if the provided index is greater than the size of the `IList<T>`, the list should (but is technically not guaranteed to) throw an `ArgumentOutOfRangeException`.

Section 66.35: DefaultIfEmpty

`DefaultIfEmpty` 用于在序列不包含任何元素时返回一个默认元素。该元素可以是类型的默认值，也可以是用户定义的该类型实例。Example:

```

var chars = new List<string>() { "a", "b", "c", "d" };

chars.DefaultIfEmpty("N/A").FirstOrDefault(); // returns "a";

chars.Where(str => str.Length > 1)
    .DefaultIfEmpty("N/A").FirstOrDefault(); // return "N/A"

chars.Where(str => str.Length > 1)
    .DefaultIfEmpty().First(); // returns null;

```

Usage in Left Joins:

With `DefaultIfEmpty` the traditional Linq Join can return a default object if no match was found. Thus acting as a SQL's Left Join. Example:

```

var leftSequence = new List<int>() { 99, 100, 5, 20, 102, 105 };
var rightSequence = new List<char>() { 'a', 'b', 'c', 'i', 'd' };

var numbersAsChars = from l in leftSequence
                     join r in rightSequence
                     on l equals (int)r into leftJoin
                     from result in leftJoin.DefaultIfEmpty('?')
                     select new
                     {
                         Number = l,
                         Character = result
                     };

foreach(var item in numbersAsChars)
{
    Console.WriteLine("Num = {0} ** Char = {1}", item.Number, item.Character);
}

```

output:

```

Num = 99      Char = c
Num = 100     Char = d
Num = 5       Char = ?
Num = 20       Char = ?
Num = 102      Char = ?
Num = 105      Char = i

```

In the case where a `DefaultIfEmpty` is used (without specifying a default value) and that will result will no matching

必须确保对象在访问其属性之前不是 null。
否则将导致 NullReferenceException。示例：

```
var leftSequence = new List<int> { 1, 2, 5 };
var rightSequence = new List<dynamic>()
{
    new { Value = 1 },
    new { Value = 2 },
    new { Value = 3 },
    new { Value = 4 },
};

var numbersAsChars = (from l in leftSequence
                      join r in rightSequence
                      on l equals r.Value into leftJoin
                      from result in leftJoin.DefaultIfEmpty()
                      select new
{
    左侧 = l,
    // 5 在右侧不会有匹配的对象，因此结果
    // 将等于 null。
    // 为避免错误，请使用：
    // - C# 6.0 或更高版本 - ?
    // - 低于该版本 - result == null ? 0 : result.Value
    右侧 = result?.Value
}).ToList();
```

第 66.36 节：ToDictionary

LINQ 方法 ToDictionary() 可用于基于给定的 `IEnumerable<T>` 源生成一个 `Dictionary< TKey, TElement >` 集合。

```
IEnumerable<User> users = GetUsers();
Dictionary<int, User> usersById = users.ToDictionary(x => x.Id);
```

在此示例中，传递给 `ToDictionary` 的单个参数类型为 `Func<TSource, TKey>`，返回每个元素的键。

这是执行以下操作的一种简洁方式：

```
Dictionary<int, User> usersById = new Dictionary<int, User>();
foreach (User u in users)
{
    usersById.Add(u.Id, u);
}
```

你也可以向 `ToDictionary` 方法传递第二个参数，该参数类型为 `Func<TSource, TElement>`，返回要为每个条目添加的 `Value`。

```
IEnumerable<User> users = GetUsers();
Dictionary<int, string> userNamesById = users.ToDictionary(x => x.Id, x => x.Name);
```

还可以指定用于比较键值的 `IComparer`。这在键是字符串且你希望忽略大小写匹配时非常有用。

```
IEnumerable<User> users = GetUsers();
Dictionary<string, User> usersByCaseInsensitiveName = users.ToDictionary(x => x.Name,
```

items on the right sequence one must make sure that the object is not `null` before accessing its properties.
Otherwise it will result in a `NullReferenceException`. Example:

```
var leftSequence = new List<int> { 1, 2, 5 };
var rightSequence = new List<dynamic>()
{
    new { Value = 1 },
    new { Value = 2 },
    new { Value = 3 },
    new { Value = 4 },
};

var numbersAsChars = (from l in leftSequence
                      join r in rightSequence
                      on l equals r.Value into leftJoin
                      from result in leftJoin.DefaultIfEmpty()
                      select new
{
    Left = l,
    // 5 will not have a matching object in the right so result
    // will be equal to null.
    // To avoid an error use:
    // - C# 6.0 or above - ?
    // - Under           - result == null ? 0 : result.Value
    Right = result?.Value
}).ToList();
```

Section 66.36: ToDictionary

The `ToDictionary()` LINQ method can be used to generate a `Dictionary< TKey, TElement >` collection based on a given `IEnumerable<T>` source.

```
IEnumerable<User> users = GetUsers();
Dictionary<int, User> usersById = users.ToDictionary(x => x.Id);
```

In this example, the single argument passed to `ToDictionary` is of type `Func<TSource, TKey>`, which returns the key for each element.

This is a concise way to perform the following operation:

```
Dictionary<int, User> usersById = new Dictionary<int, User>();
foreach (User u in users)
{
    usersById.Add(u.Id, u);
}
```

You can also pass a second parameter to the `ToDictionary` method, which is of type `Func<TSource, TElement>` and returns the `Value` to be added for each entry.

```
IEnumerable<User> users = GetUsers();
Dictionary<int, string> userNamesById = users.ToDictionary(x => x.Id, x => x.Name);
```

It is also possible to specify the `IComparer` that is used to compare key values. This can be useful when the key is a string and you want it to match case-insensitive.

```
IEnumerable<User> users = GetUsers();
Dictionary<string, User> usersByCaseInsensitiveName = users.ToDictionary(x => x.Name,
```

```
StringComparer.InvariantCultureIgnoreCase);
```

```
var user1 = usersByCaseInsensitiveName["john"];
var user2 = usersByCaseInsensitiveName["JOHN"];
user1 == user2; // 返回 true
```

注意：ToDictionary方法要求所有键必须唯一，不能有重复键。如果存在重复键，则会抛出异常：ArgumentException：已添加具有相同键的项。如果你知道会有多个元素具有相同键的情况，最好使用ToLookup代替。

第66.37节：Concat

合并两个集合（不去重）

```
List<int> foo = new List<int> { 1, 2, 3 };
List<int> bar = new List<int> { 3, 4, 5 };

// 通过 Enumerable 静态类
var result = Enumerable.Concat(foo, bar).ToList(); // 1,2,3,3,4,5

// 通过扩展方法
var result = foo.Concat(bar).ToList(); // 1,2,3,3,4,5
```

第66.38节：为IEnumerable<T>构建你自己的Linq操作符

Linq的一个优点是它非常容易扩展。你只需要创建一个扩展方法，其参数是IEnumerable<T>。

```
public namespace MyNamespace
{
    public static class LinqExtensions
    {
        public static IEnumerable<List<T>> Batch<T>(this IEnumerable<T> source, int batchSize)
        {
            var batch = new List<T>();
            foreach (T item in source)
            {
                batch.Add(item);
                if (batch.Count == batchSize)
                {
                    yield return batch;
                }
            }
            if (batch.Count > 0)
                yield return batch;
        }
    }
}
```

此示例将 IEnumerable<T> 中的项目拆分为固定大小的列表，最后一个列表包含剩余的项目。注意扩展方法所应用的对象作为初始参数通过 this 关键字传入（参数名为 source）。然后使用 yield 关键字输出下一个结果中的项目 IEnumerable<T>，然后从该点继续执行（参见 yield 关键字）。

```
StringComparer.InvariantCultureIgnoreCase);
```

```
var user1 = usersByCaseInsensitiveName["john"];
var user2 = usersByCaseInsensitiveName["JOHN"];
user1 == user2; // Returns true
```

Note: the ToDictionary method requires all keys to be unique, there must be no duplicate keys. If there are, then an exception is thrown: ArgumentException: An item with the same key has already been added. If you have a scenario where you know that you will have multiple elements with the same key, then you are better off using ToLookup instead.

Section 66.37: Concat

Merges two collections (without removing duplicates)

```
List<int> foo = new List<int> { 1, 2, 3 };
List<int> bar = new List<int> { 3, 4, 5 };

// Through Enumerable static class
var result = Enumerable.Concat(foo, bar).ToList(); // 1,2,3,3,4,5

// Through extension method
var result = foo.Concat(bar).ToList(); // 1,2,3,3,4,5
```

Section 66.38: Build your own Linq operators for IEnumerable<T>

One of the great things about Linq is that it is so easy to extend. You just need to create an extension method whose argument is IEnumerable<T>.

```
public namespace MyNamespace
{
    public static class LinqExtensions
    {
        public static IEnumerable<List<T>> Batch<T>(this IEnumerable<T> source, int batchSize)
        {
            var batch = new List<T>();
            foreach (T item in source)
            {
                batch.Add(item);
                if (batch.Count == batchSize)
                {
                    yield return batch;
                    batch = new List<T>();
                }
            }
            if (batch.Count > 0)
                yield return batch;
        }
    }
}
```

This example splits the items in an IEnumerable<T> into lists of a fixed size, the last list containing the remainder of the items. Notice how the object to which the extension method is applied is passed in (argument source) as the initial argument using the this keyword. Then the yield keyword is used to output the next item in the output IEnumerable<T> before continuing with execution from that point (see yield keyword).

此示例在代码中的使用方式如下：

```
//using MyNamespace;
var items = new List<int> { 2, 3, 4, 5, 6 };
foreach (List<int> sublist in items.Batch(3))
{
    // 执行某些操作
}
```

第一次循环时，sublist 为 {2, 3, 4}，第二次为 {5, 6}。

自定义 LINQ 方法也可以与标准 LINQ 方法结合使用。例如：

```
//using MyNamespace;
var result = Enumerable.Range(0, 13)          // 生成一个列表
    .Where(x => x%2 == 0) // 过滤列表或执行其他操作
    .Batch(3)             // 调用我们的扩展方法
    .ToList()              // 调用其他标准方法
```

此查询将返回按大小为3分组的偶数批次：{0, 2, 4}, {6, 8, 10}, {12}

请记住，您需要一行 `using MyNamespace;` 才能访问该扩展方法。

第66.39节：Select - 转换元素

Select 允许您对实现 `IEnumerable` 的任何数据结构中的每个元素应用转换。

获取以下列表中每个字符串的第一个字符：

```
List<String> trees = new List<String>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

使用常规 (lambda) 语法

```
//下面的 select 语句将 trees 中的每个元素转换为其第一个字符。
IEnumerable<String> initials = trees.Select(tree => tree.Substring(0, 1));
foreach (String initial in initials) {
    System.Console.WriteLine(initial);
}
```

输出：

O
B
B
E
H
M
Fiddle 上的实时演示

使用LINQ查询语法

```
initials = from tree in trees
           select tree.Substring(0, 1);
```

This example would be used in your code like this:

```
//using MyNamespace;
var items = new List<int> { 2, 3, 4, 5, 6 };
foreach (List<int> sublist in items.Batch(3))
{
    // do something
}
```

On the first loop, sublist would be {2, 3, 4} and on the second {5, 6}.

Custom LINQ methods can be combined with standard LINQ methods too. e.g.:

```
//using MyNamespace;
var result = Enumerable.Range(0, 13)          // generate a list
    .Where(x => x%2 == 0) // filter the list or do something other
    .Batch(3)             // call our extension method
    .ToList()              // call other standard methods
```

This query will return even numbers grouped in batches with a size of 3: {0, 2, 4}, {6, 8, 10}, {12}

Remember you need a `using MyNamespace;` line in order to be able to access the extension method.

Section 66.39: Select - Transforming elements

Select allows you to apply a transformation to every element in any data structure implementing `IEnumerable`.

Getting the first character of each string in the following list:

```
List<String> trees = new List<String>{ "Oak", "Birch", "Beech", "Elm", "Hazel", "Maple" };
```

Using regular (lambda) syntax

```
//The below select statement transforms each element in tree into its first character.
IEnumerable<String> initials = trees.Select(tree => tree.Substring(0, 1));
foreach (String initial in initials) {
    System.Console.WriteLine(initial);
}
```

Output:

O
B
B
E
H
M

[Live Demo on .NET Fiddle](#)

Using LINQ Query Syntax

```
initials = from tree in trees
           select tree.Substring(0, 1);
```

第66.40节：OrderByDescending

按指定值对集合进行排序。

当值是整数、双精度浮点数或单精度浮点数时，它从最大值开始，这意味着你首先得到正值，然后是零，最后是负值（见示例1）。

当你按字符排序时，该方法比较字符的ASCII值来对集合进行排序（参见示例2）。

当你排序字符串时，OrderBy方法通过查看它们的CultureInfo来比较它们，但通常是从字母表中的最后一个字母开始（z,y,x,...）。

这种排序方式称为降序，如果你想要相反的顺序，则需要升序（见OrderBy）。

示例1：

```
int[] numbers = {-2, -1, 0, 1, 2};  
IEnumerable<int> descending = numbers.OrderByDescending(x => x);  
// 返回 {2, 1, 0, -1, -2}
```

示例2：

```
char[] letters = {' ', '!', '?', '[', '{', '+', '1', '9', 'a', 'A', 'b', 'B', 'y', 'Y', 'z', 'Z'};  
IEnumerable<char> descending = letters.OrderByDescending(x => x);  
// 返回 {'z', 'y', 'b', 'a', '[', '9', '1', '+', '!', ' '}
```

示例 3：

```
class Person  
{  
    public string Name { get; set; }  
    public int Age { get; set; }  
}  
  
var people = new[]  
{  
    new 人员 {姓名 = "Alice", 年龄 = 25},  
    new 人员 {姓名 = "Bob", 年龄 = 21},  
    new 人员 {姓名 = "Carol", 年龄 = 43}  
};  
var oldestPerson = people.OrderByDescending(x => x.Age).First();  
var name = oldestPerson.Name; // Carol
```

第 66.41 节：并集

合并两个集合，使用默认的相等比较器创建一个不重复的集合

```
int[] numbers1 = { 1, 2, 3 };  
int[] numbers2 = { 2, 3, 4, 5 };  
  
var allElement = numbers1.Union(numbers2); // allElement 现在包含 1,2,3,4,5
```

[.NET Fiddle 在线演示](#)

Section 66.40: OrderByDescending

Orders a collection by a specified value.

When the value is an **integer**, **double** or **float** it starts with the *maximal value*, which means that you get first the positive values, than zero and afterwards the negative values (see Example 1).

When you order by a **char** the method compares the *ascii values* of the chars to sort the collection (see Example 2).

When you sort **strings** the OrderBy method compares them by taking a look at their *CultureInfo* but normally starting with the *last letter* in the alphabet (z,y,x,...).

This kind of order is called descending, if you want it the other way round you need ascending (see OrderBy).

Example 1:

```
int[] numbers = {-2, -1, 0, 1, 2};  
IEnumerable<int> descending = numbers.OrderByDescending(x => x);  
// returns {2, 1, 0, -1, -2}
```

Example 2:

```
char[] letters = {' ', '!', '?', '[', '{', '+', '1', '9', 'a', 'A', 'b', 'B', 'y', 'Y', 'z', 'Z'};  
IEnumerable<char> descending = letters.OrderByDescending(x => x);  
// returns { '{', 'z', 'y', 'b', 'a', '[', '9', '1', '+', '!', ' ' }
```

Example 3:

```
class Person  
{  
    public string Name { get; set; }  
    public int Age { get; set; }  
}  
  
var people = new[]  
{  
    new Person {Name = "Alice", Age = 25},  
    new Person {Name = "Bob", Age = 21},  
    new Person {Name = "Carol", Age = 43}  
};  
var oldestPerson = people.OrderByDescending(x => x.Age).First();  
var name = oldestPerson.Name; // Carol
```

Section 66.41: Union

Merges two collections to create a distinct collection using the default equality comparer

```
int[] numbers1 = { 1, 2, 3 };  
int[] numbers2 = { 2, 3, 4, 5 };  
  
var allElement = numbers1.Union(numbers2); // AllElement now contains 1,2,3,4,5
```

[Live Demo on .NET Fiddle](#)

第66.42节：带有外部范围变量的GroupJoin

```
Customer[] customers = Customers.ToArray();
Purchase[] purchases = Purchases.ToArray();

var groupJoinQuery =
    from c in customers
    join p in purchases on c.ID equals p.CustomerID
    into custPurchases
    select new
    {
        CustName = c.Name,
        custPurchases
    };

```

第66.43节：Linq量词

量词操作会返回一个布尔值，用于判断序列中的某些或所有元素是否满足某个条件。在本文中，我们将看到一些常见的LINQ to Objects场景，可以使用这些操作符。有3种可以在LINQ中使用的量词操作：

All - 用于确定序列中的所有元素是否都满足某个条件。例如：

```
int[] array = { 10, 20, 30 };

// 所有元素都 >= 10 吗？是
array.All(element => element >= 10);

// 所有元素都 >= 20 吗？否
array.All(element => element >= 20);

// 所有元素都 < 40 吗？是
array.All(element => element < 40);
```

Any - 用于判断序列中是否有任意元素满足条件。例如：

```
int[] query=new int[] { 2, 3, 4 }
query.Any (n => n == 3);
```

Contains - 用于判断序列中是否包含指定元素。例如：

```
// 对于 int 数组
int[] query =new int[] { 1,2,3 };
query.Contains(1);

// 用于字符串数组
string[] query={"Tom", "grey"};
query.Contains("Tom");

// for a string
var stringValue="hello";
stringValue.Contains("h");
```

第66.44节：TakeWhile

TakeWhile 返回序列中满足条件为真的元素

Section 66.42: GroupJoin with outer range variable

```
Customer[] customers = Customers.ToArray();
Purchase[] purchases = Purchases.ToArray();

var groupJoinQuery =
    from c in customers
    join p in purchases on c.ID equals p.CustomerID
    into custPurchases
    select new
    {
        CustName = c.Name,
        custPurchases
    };

```

Section 66.43: Linq Quantifiers

Quantifier operations return a Boolean value if some or all of the elements in a sequence satisfy a condition. In this article, we will see some common LINQ to Objects scenarios where we can use these operators. There are 3 Quantifiers operations that can be used in LINQ:

All – used to determine whether all the elements in a sequence satisfy a condition. Eg:

```
int[] array = { 10, 20, 30 };

// Are all elements >= 10? YES
array.All(element => element >= 10);

// Are all elements >= 20? NO
array.All(element => element >= 20);

// Are all elements < 40? YES
array.All(element => element < 40);
```

Any - used to determine whether any elements in a sequence satisfy a condition. Eg:

```
int[] query=new int[] { 2, 3, 4 }
query.Any (n => n == 3);
```

Contains - used to determine whether a sequence contains a specified element. Eg:

```
// for int array
int[] query =new int[] { 1,2,3 };
query.Contains(1);

// for string array
string[] query={"Tom", "grey"};
query.Contains("Tom");

// for a string
var stringValue="hello";
stringValue.Contains("h");
```

Section 66.44: TakeWhile

TakeWhile returns elements from a sequence as long as the condition is true

```
int[] list = { 1, 10, 40, 50, 44, 70, 4 };
var result = list.TakeWhile(item => item < 50).ToList();
// 结果 = { 1, 10, 40 }
```

第66.45节：Reverse

- 反转序列中元素的顺序。
- 如果没有元素，则抛出ArgumentNullException：source为null。

示例：

```
// 创建一个数组。
int[] array = { 1, 2, 3, 4 };
// 调用数组的反转扩展方法。 //4
var reverse = array.Reverse(); //3
// 将数组内容写到屏幕上。 //2
foreach (int value in reverse) //1
    Console.WriteLine(value);
```

[实时代码示例](#)

请记住，Reverse()的行为可能会根据LINQ语句的链式顺序而有所不同。

```
// 创建字符列表
List<int> integerlist = new List<int>() { 1, 2, 3, 4, 5, 6 };

// 反转列表然后取前两个元素
IEnumerable<int> reverseFirst = integerlist.Reverse<int>().Take(2);

// 取两个元素然后只反转这两个元素
IEnumerable<int> reverseLast = integerlist.Take(2).Reverse();

// reverseFirst 输出: 6, 5
// reverseLast 输出: 2, 1
```

[实时代码示例](#)

Reverse() 的工作原理是先缓存所有元素，然后再倒序遍历，这种方式效率不高，但从这个角度看，OrderBy 也不高效。

在 LINQ-to-Objects 中，有缓存操作（Reverse、OrderBy、GroupBy 等）和非缓存操作（Where、Take、Skip 等）。

示例：非缓存的 Reverse 扩展方法

```
public static IEnumerable<T> Reverse<T>(this IList<T> list) {
    for (int i = list.Count - 1; i >= 0; i--)
        yield return list[i];
}
```

[实时代码示例](#)

如果在迭代过程中修改列表，该方法可能会遇到问题。

```
int[] list = { 1, 10, 40, 50, 44, 70, 4 };
var result = list.TakeWhile(item => item < 50).ToList();
// result = { 1, 10, 40 }
```

Section 66.45: Reverse

- Inverts the order of the elements in a sequence.
- If there is no items throws a ArgumentNullException: source is null.

Example:

```
// Create an array.
int[] array = { 1, 2, 3, 4 }; //Output:
// Call reverse extension method on the array. //4
var reverse = array.Reverse(); //3
// Write contents of array to screen. //2
foreach (int value in reverse) //1
    Console.WriteLine(value);
```

[Live code example](#)

Remeber that Reverse() may work diffrent depending on the chain order of your LINQ statements.

```
//Create List of chars
List<int> integerlist = new List<int>() { 1, 2, 3, 4, 5, 6 };

//Reversing the list then taking the two first elements
IEnumerable<int> reverseFirst = integerlist.Reverse<int>().Take(2);

//Taking 2 elements and then reversing only thos two
IEnumerable<int> reverseLast = integerlist.Take(2).Reverse();

//reverseFirst output: 6, 5
//reverseLast output: 2, 1
```

[Live code example](#)

Reverse() works by buffering everything then walk through it backwards, whitch is not very efficient, but neither is OrderBy from that perspective.

In LINQ-to-Objects, there are buffering operations (Reverse, OrderBy, GroupBy, etc) and non-buffering operations (Where, Take, Skip, etc).

Example: Non-buffering Reverse extention

```
public static IEnumerable<T> Reverse<T>(this IList<T> list) {
    for (int i = list.Count - 1; i >= 0; i--)
        yield return list[i];
}
```

[Live code example](#)

This method can encounter problems if u mutate the list while iterating.

第66.46节：Count 和 LongCount

Count 返回 `IEnumerable<T>` 中元素的数量。 Count 还提供了一个可选的谓词参数，允许你筛选想要计数的元素。

```
int[] 数组 = { 1, 2, 3, 4, 2, 5, 3, 1, 2 };

int n = 数组.Count(); // 返回数组中元素的数量
int x = 数组.Count(i => i > 2); // 返回数组中大于2的元素数量
```

LongCount 的工作方式与 Count 相同，但返回类型为 long，适用于计数长度超过 `int.MaxValue` 的 `IEnumerable<T>` 序列

```
int[] 数组 = GetLargeArray();

long n = 数组.LongCount(); // 返回数组中元素的数量
long x = 数组.LongCount(i => i > 100); // 返回数组中大于100的元素数量
```

第66.47节：逐步构建查询

由于LINQ使用**延迟执行**，我们可以拥有一个查询对象，该对象实际上不包含值，但在求值时会返回值。因此，我们可以根据控制流程动态构建查询，并在完成后进行求值：

```
IEnumerable<VehicleModel> BuildQuery(int 车辆类型, SearchModel 搜索模型, int 起始 = 1, int 数量 = -1) {
    IEnumerable<VehicleModel> 查询 = _entities.Vehicles
        .Where(x => x.Active && x.Type == 车辆类型)
        .Select(x => new VehicleModel {
            Id = v.Id,
            Year = v.Year,
            Class = v.Class,
            制造商 = v.Make,
            型号 = v.Model,
            气缸数 = v.Cylinders ?? 0
        });
}
```

我们可以有条件地应用过滤器：

```
if (!search.Years.Contains("all", StringComparer.OrdinalIgnoreCase))
    query = query.Where(v => search.Years.Contains(v.Year));

if (!search.Makes.Contains("all", StringComparer.OrdinalIgnoreCase)) {
    query = query.Where(v => search.Makes.Contains(v.Make));
}

if (!search.Models.Contains("all", StringComparer.OrdinalIgnoreCase)) {
    query = query.Where(v => search.Models.Contains(v.Model));
}

if (!search.Cylinders.Equals("all", StringComparer.OrdinalIgnoreCase)) {
    decimal minCylinders = 0;
    decimal maxCylinders = 0;
    switch (search.Cylinders) {
        case "2-4":
            maxCylinders = 4;
            break;
}
```

Section 66.46: Count and LongCount

Count returns the number of elements in an `IEnumerable<T>`. Count also exposes an optional predicate parameter that allows you to filter the elements you want to count.

```
int[] array = { 1, 2, 3, 4, 2, 5, 3, 1, 2 };

int n = array.Count(); // returns the number of elements in the array
int x = array.Count(i => i > 2); // returns the number of elements in the array greater than 2
```

LongCount works the same way as Count but has a return type of `long` and is used for counting `IEnumerable<T>` sequences that are longer than `int.MaxValue`

```
int[] array = GetLargeArray();

long n = array.LongCount(); // returns the number of elements in the array
long x = array.LongCount(i => i > 100); // returns the number of elements in the array greater than 100
```

Section 66.47: Incrementally building a query

Because LINQ uses **deferred execution**, we can have a query object that doesn't actually contain the values, but will return the values when evaluated. We can thus dynamically build the query based on our control flow, and evaluate it once we are finished:

```
IEnumerable<VehicleModel> BuildQuery(int 车辆类型, SearchModel search, int start = 1, int count = -1) {
    IEnumerable<VehicleModel> query = _entities.Vehicles
        .Where(x => x.Active && x.Type == 车辆类型)
        .Select(x => new VehicleModel {
            Id = v.Id,
            Year = v.Year,
            Class = v.Class,
            Make = v.Make,
            Model = v.Model,
            Cylinders = v.Cylinders ?? 0
        });
}
```

We can conditionally apply filters:

```
if (!search.Years.Contains("all", StringComparer.OrdinalIgnoreCase))
    query = query.Where(v => search.Years.Contains(v.Year));

if (!search.Makes.Contains("all", StringComparer.OrdinalIgnoreCase)) {
    query = query.Where(v => search.Makes.Contains(v.Make));
}

if (!search.Models.Contains("all", StringComparer.OrdinalIgnoreCase)) {
    query = query.Where(v => search.Models.Contains(v.Model));
}

if (!search.Cylinders.Equals("all", StringComparer.OrdinalIgnoreCase)) {
    decimal minCylinders = 0;
    decimal maxCylinders = 0;
    switch (search.Cylinders) {
        case "2-4":
            maxCylinders = 4;
            break;
}
```

```

        case "5-6":
minCylinders = 5;
    maxCylinders = 6;
    break;
    case "8":
minCylinders = 8;
    maxCylinders = 8;
    break;
    case "10+":
minCylinders = 10;
    break;
}
if (minCylinders > 0) {
query = query.Where(v => v.Cylinders >= minCylinders);
}
if (maxCylinders > 0) {
query = query.Where(v => v.Cylinders <= maxCylinders);
}
}

```

我们可以根据条件为查询添加排序顺序：

```

switch (search.SortingColumn.ToLower()) {
    case "make_model":
query = query.OrderBy(v => v.Make).ThenBy(v => v.Model);
    break;
    case "year":
query = query.OrderBy(v => v.Year);
    break;
    case "engine_size":
query = query.OrderBy(v => v.EngineSize).ThenBy(v => v.Cylinders);
    break;
    default:
query = query.OrderBy(v => v.Year); //默认排序。
}

```

我们的查询可以定义为从指定点开始：

```
query = query.Skip(start - 1);
```

并定义返回特定数量的记录：

```

if (count > -1) {
    query = query.Take(count);
}
return query;
}

```

一旦我们有了查询对象，就可以使用foreach循环，或者返回一组值的LINQ方法，如ToList或ToArray，来评估结果：

```

searchModel sm;

// 在这里填充搜索模型
// ...

List<VehicleModel> list = BuildQuery(5, sm).ToList();

```

```

        case "5-6":
minCylinders = 5;
    maxCylinders = 6;
    break;
    case "8":
minCylinders = 8;
    maxCylinders = 8;
    break;
    case "10+":
minCylinders = 10;
    break;
}
if (minCylinders > 0) {
query = query.Where(v => v.Cylinders >= minCylinders);
}
if (maxCylinders > 0) {
query = query.Where(v => v.Cylinders <= maxCylinders);
}
}

```

We can add a sort order to the query based on a condition:

```

switch (search.SortingColumn.ToLower()) {
    case "make_model":
query = query.OrderBy(v => v.Make).ThenBy(v => v.Model);
    break;
    case "year":
query = query.OrderBy(v => v.Year);
    break;
    case "engine_size":
query = query.OrderBy(v => v.EngineSize).ThenBy(v => v.Cylinders);
    break;
    default:
query = query.OrderBy(v => v.Year); //The default sorting.
}

```

Our query can be defined to start from a given point:

```
query = query.Skip(start - 1);
```

and defined to return a specific number of records:

```

if (count > -1) {
    query = query.Take(count);
}
return query;
}

```

Once we have the query object, we can evaluate the results with a foreach loop, or one of the LINQ methods that returns a set of values, such as ToList or ToArray:

```

searchModel sm;

// populate the search model here
// ...

List<VehicleModel> list = BuildQuery(5, sm).ToList();

```

第66.48节：使用Func<TSource, int, TResult> selector进行选择 - 用于获取元素的排名

SELECT扩展方法的一个重载还会传递集合中当前项的index。这是它的一些用法。

获取项目的“行号”

```
var rowNumbers = collection.OrderBy(item => item.Property1)
    .ThenBy(item => item.Property2)
    .ThenByDescending(item => item.Property3)
    .Select((item, index) => new { 项目 = item, 行号 = index })
    .ToList();
```

获取项目在其组内的排名

```
var 组内排名 = collection.GroupBy(item => item.Property1)
    .OrderBy(group => group.Key)
    .SelectMany(group => group.OrderBy(item => item.Property2)
        .ThenByDescending(item => item.Property3)
        .Select((item, index) => new
    {
        项目 = item,
        组内排名 = index
    })).ToList();
```

获取组的排名 (Oracle中也称为dense_rank)

```
var 所属组排名 = collection.GroupBy(item => item.Property1)
    .OrderBy(group => group.Key)
    .Select((group, index) => new
    {
        项目集合 = group,
        排名 = index
    })
    .SelectMany(v => v.项目集合, (s, i) => new
    {
        项目 = i,
        DenseRank = s.Rank
    }).ToList();
```

测试此代码时可以使用：

```
public class SomeObject
{
    public int Property1 { get; set; }
    public int Property2 { get; set; }
    public int Property3 { get; set; }

    public override string ToString()
    {
        return string.Join(", ", Property1, Property2, Property3);
    }
}
```

以及数据：

```
List<SomeObject> collection = new List<SomeObject>
```

Section 66.48: Select with Func<TSource, int, TResult> selector - Use to get ranking of elements

On of the overloads of the **SELECT** extension methods also passes the **index** of the current item in the collection being **SELECTed**. These are a few uses of it.

Get the "row number" of the items

```
var rowNumbers = collection.OrderBy(item => item.Property1)
    .ThenBy(item => item.Property2)
    .ThenByDescending(item => item.Property3)
    .Select((item, index) => new { Item = item, RowNumber = index })
    .ToList();
```

Get the rank of an item *within its group*

```
var rankInGroup = collection.GroupBy(item => item.Property1)
    .OrderBy(group => group.Key)
    .SelectMany(group => group.OrderBy(item => item.Property2)
        .ThenByDescending(item => item.Property3)
        .Select((item, index) => new
    {
        Item = item,
        RankInGroup = index
    })).ToList();
```

Get the ranking of groups (also known in Oracle as dense_rank)

```
var rankOfBelongingGroup = collection.GroupBy(item => item.Property1)
    .OrderBy(group => group.Key)
    .Select((group, index) => new
    {
        Items = group,
        Rank = index
    })
    .SelectMany(v => v.Items, (s, i) => new
    {
        Item = i,
        DenseRank = s.Rank
    }).ToList();
```

For testing this you can use:

```
public class SomeObject
{
    public int Property1 { get; set; }
    public int Property2 { get; set; }
    public int Property3 { get; set; }

    public override string ToString()
    {
        return string.Join(", ", Property1, Property2, Property3);
    }
}
```

And data:

```
List<SomeObject> collection = new List<SomeObject>
```

```
{  
    new SomeObject { Property1 = 1, Property2 = 1, Property3 = 1},  
    new SomeObject { Property1 = 1, Property2 = 2, Property3 = 1},  
    new SomeObject { Property1 = 1, Property2 = 2, Property3 = 2},  
    new SomeObject { Property1 = 2, Property2 = 1, Property3 = 1},  
    new SomeObject { Property1 = 2, Property2 = 2, Property3 = 1},  
    new SomeObject { Property1 = 2, Property2 = 2, Property3 = 1},  
    new SomeObject { Property1 = 2, Property2 = 3, Property3 = 1}  
};
```

```
{  
    new SomeObject { Property1 = 1, Property2 = 1, Property3 = 1},  
    new SomeObject { Property1 = 1, Property2 = 2, Property3 = 1},  
    new SomeObject { Property1 = 1, Property2 = 2, Property3 = 2},  
    new SomeObject { Property1 = 2, Property2 = 1, Property3 = 1},  
    new SomeObject { Property1 = 2, Property2 = 2, Property3 = 1},  
    new SomeObject { Property1 = 2, Property2 = 2, Property3 = 1},  
    new SomeObject { Property1 = 2, Property2 = 3, Property3 = 1}  
};
```

第67章：LINQ 到 XML

第67.1节：使用 LINQ 到 XML 读取 XML

```
<?xml version="1.0" encoding="utf-8" ?>
<Employees>
<Employee>
<EmpId>1</EmpId>
<Name>Sam</Name>
<Sex>男</Sex>
<Phone Type="Home">423-555-0124</Phone>
<Phone Type="Work">424-555-0545</Phone>
<Address>
<Street>考克斯街7A号</Street>
<City>阿坎波</City>
<State>加利福尼亚州</State>
<Zip>95220</Zip>
<Country>美国</Country>
</Address>
</Employee>
<Employee>
<EmpId>2</EmpId>
<Name>露西</Name>
<Sex>女</Sex>
<Phone Type="Home">143-555-0763</Phone>
<Phone Type="Work">434-555-0567</Phone>
<Address>
<Street>杰斯湾</Street>
<City>阿尔塔</City>
<State>加利福尼亚州</State>
<Zip>95701</Zip>
<Country>美国</Country>
</Address>
</Employee>
</Employees>
```

使用 LINQ 读取该 XML 文件

```
XDocument xdocument = XDocument.Load("Employees.xml");
IEnumerable< XElement > employees = xdocument.Root.Elements();
foreach (var employee in employees)
{
    Console.WriteLine(employee);
}
```

访问单个元素

```
XElement xelement = XElement.Load("Employees.xml");
IEnumerable< XElement > employees = xelement.Root.Elements();
Console.WriteLine("所有员工姓名列表：" );
foreach (var employee in employees)
{
    Console.WriteLine(employee.Element("Name").Value);
}
```

访问多个元素

```
XElement xelement = XElement.Load("Employees.xml");
```

Chapter 67: LINQ to XML

Section 67.1: Read XML using LINQ to XML

```
<?xml version="1.0" encoding="utf-8" ?>
<Employees>
<Employee>
<EmpId>1</EmpId>
<Name>Sam</Name>
<Sex>Male</Sex>
<Phone Type="Home">423-555-0124</Phone>
<Phone Type="Work">424-555-0545</Phone>
<Address>
<Street>7A Cox Street</Street>
<City>Acampo</City>
<State>CA</State>
<Zip>95220</Zip>
<Country>USA</Country>
</Address>
</Employee>
<Employee>
<EmpId>2</EmpId>
<Name>Lucy</Name>
<Sex>Female</Sex>
<Phone Type="Home">143-555-0763</Phone>
<Phone Type="Work">434-555-0567</Phone>
<Address>
<Street>Jess Bay</Street>
<City>Alta</City>
<State>CA</State>
<Zip>95701</Zip>
<Country>USA</Country>
</Address>
</Employee>
</Employees>
```

To read that XML file using LINQ

```
XDocument xdocument = XDocument.Load("Employees.xml");
IEnumerable< XElement > employees = xdocument.Root.Elements();
foreach (var employee in employees)
{
    Console.WriteLine(employee);
}
```

To access single element

```
XElement xelement = XElement.Load("Employees.xml");
IEnumerable< XElement > employees = xelement.Root.Elements();
Console.WriteLine("List of all Employee Names :");
foreach (var employee in employees)
{
    Console.WriteLine(employee.Element("Name").Value);
}
```

To access multiple elements

```
XElement xelement = XElement.Load("Employees.xml");
```

```

IEnumarable<XElement> employees = xelement.Root.Elements();
Console.WriteLine("所有员工姓名及其ID列表:");
foreach (var employee in employees)
{
    Console.WriteLine("{0} 的员工ID是 {1}",
        employee.Element("Name").Value,
        employee.Element("EmpId").Value);
}

```

访问具有特定属性的所有元素

```

 XElement xelement = XElement.Load("Employees.xml");
 var name = from nm in xelement.Root.Elements("Employee")
             where (string)nm.Element("Sex") == "Female"
             select nm;
 Console.WriteLine("女性员工详情:");
 foreach (XElement xEle in name)
 Console.WriteLine(xEle);

```

访问具有特定属性的特定元素

```

 XElement xelement = XElement.Load("../..\..\Employees.xml");
 var homePhone = from phoneno in xelement.Root.Elements("Employee")
                  where (string)phoneno.Element("Phone").Attribute("Type") == "Home"
                  select phoneno;
 Console.WriteLine("家庭电话号码列表:");
 foreach (XElement xEle in homePhone)
 {
    Console.WriteLine(xEle.Element("Phone").Value);
}

```

```

IEnumarable<XElement> employees = xelement.Root.Elements();
Console.WriteLine("List of all Employee Names along with their ID:");
foreach (var employee in employees)
{
    Console.WriteLine("{0} has Employee ID {1}",
        employee.Element("Name").Value,
        employee.Element("EmpId").Value);
}

```

To access all Elements having a specific attribute

```

 XElement xelement = XElement.Load("Employees.xml");
 var name = from nm in xelement.Root.Elements("Employee")
             where (string)nm.Element("Sex") == "Female"
             select nm;
 Console.WriteLine("Details of Female Employees:");
 foreach (XElement xEle in name)
 Console.WriteLine(xEle);

```

To access specific element having a specific attribute

```

 XElement xelement = XElement.Load("../..\..\Employees.xml");
 var homePhone = from phoneno in xelement.Root.Elements("Employee")
                  where (string)phoneno.Element("Phone").Attribute("Type") == "Home"
                  select phoneno;
 Console.WriteLine("List HomePhone Nos.");
 foreach (XElement xEle in homePhone)
 {
    Console.WriteLine(xEle.Element("Phone").Value);
}

```

第68章：并行LINQ (PLINQ)

第68.1节：简单示例

此示例展示了如何使用PLINQ通过多个线程计算1到10000之间的偶数。注意，结果列表不会是有序的！

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
    .Where(x => x % 2 == 0)
    .ToList();

// evenNumbers = { 4, 26, 28, 30, ... }
// 不同运行结果的顺序会有所不同
```

第68.2节：WithDegreeOfParallelism

并行度是用于处理查询的最大并发执行任务数。

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
    .WithDegreeOfParallelism(4)
    .Where(x => x % 2 == 0);
```

第68.3节：AsOrdered

此示例展示了如何使用PLINQ通过多个线程计算1到10000之间的偶数。结果列表中将保持顺序，但请注意，AsOrdered可能会影响大量元素的性能，因此在可能的情况下，优先使用无序处理。

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
    .AsOrdered()
    .Where(x => x % 2 == 0)
    .ToList();

// evenNumbers = { 2, 4, 6, 8, ..., 10000 }
```

第68.4节：AsUnordered

处理大量元素时，有序序列可能会影响性能。为缓解这一问题，当序列顺序不再必要时，可以调用AsUnordered。

```
var sequence = Enumerable.Range(1, 10000).Select(x => -1 * x); // -1, -2, ...
var evenNumbers = sequence.AsParallel()
    .OrderBy(x => x)
    .Take(5000)
    .AsUnordered()
    .Where(x => x % 2 == 0) // 这一行不会受到排序的影响
    .ToList();
```

Chapter 68: Parallel LINQ (PLINQ)

Section 68.1: Simple example

This example shows how PLINQ can be used to calculate the even numbers between 1 and 10,000 using multiple threads. Note that the resulting list will won't be ordered!

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
    .Where(x => x % 2 == 0)
    .ToList();

// evenNumbers = { 4, 26, 28, 30, ... }
// Order will vary with different runs
```

Section 68.2: WithDegreeOfParallelism

The degree of parallelism is the maximum number of concurrently executing tasks that will be used to process the query.

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
    .WithDegreeOfParallelism(4)
    .Where(x => x % 2 == 0);
```

Section 68.3: AsOrdered

This example shows how PLINQ can be used to calculate the even numbers between 1 and 10,000 using multiple threads. Order will be maintained in the resulting list, however keep in mind that AsOrdered may hurt performance for a large numbers of elements, so un-ordered processing is preferred when possible.

```
var sequence = Enumerable.Range(1, 10000);
var evenNumbers = sequence.AsParallel()
    .AsOrdered()
    .Where(x => x % 2 == 0)
    .ToList();

// evenNumbers = { 2, 4, 6, 8, ..., 10000 }
```

Section 68.4: AsUnordered

Ordered sequences may hurt performance when dealing with a large number of elements. To mitigate this, it's possible to call AsUnordered when the sequence order is no longer necessary.

```
var sequence = Enumerable.Range(1, 10000).Select(x => -1 * x); // -1, -2, ...
var evenNumbers = sequence.AsParallel()
    .OrderBy(x => x)
    .Take(5000)
    .AsUnordered()
    .Where(x => x % 2 == 0) // This line won't be affected by ordering
    .ToList();
```

第69章： XmlDocument 和 System.Xml 命名空间

第69.1节： XmlDocument 与 XDocument (示例和比较)

有多种方式与 XML 文件交互。

1. XmlDocument
2. XDocument
3. XmlReader/XmlWriter

在 LINQ to XML 出现之前，我们习惯使用 XmlDocument 来操作 XML，比如添加属性、元素等。现在 LINQ to XML 使用 XDocument 来完成同样的操作。语法比 XmlDocument 简单得多，且代码量最少。

此外，XDocument 比 XmlDocument 快得多。 XmlDocument 是一个用于查询 XML 文档的老旧且不够优雅的解决方案。

我将展示一些 XmlDocument 类 和 XDocument 类 的示例：

加载 XML 文件

```
string filename = @"C:\empest.xml";
```

X XmlDocument

```
 XmlDocument _doc = new XmlDocument();
_doc.Load(filename);
```

X XDocument

```
XDocument _doc = XDocument.Load(fileName);
```

创建 XmlDocument

X XmlDocument

```
 XmlDocument doc = new XmlDocument();
XmlElement root = doc.CreateElement("root");
root.SetAttribute("name", "value");
XmlElement child = doc.CreateElement("child");
child.InnerText = "文本节点";
root.AppendChild(child);
doc.AppendChild(root);
```

X XDocument

```
XDocument doc = new XDocument(
```

Chapter 69: XmlDocument and the System.Xml namespace

Section 69.1: XmlDocument vs XDocument (Example and comparison)

There are several ways interact with an Xml file.

1. Xml Document
2. XDocument
3. XmlReader/XmlWriter

Before LINQ to XML we were used XmlDocument for manipulations in XML like adding attributes, elements and so on. Now LINQ to XML uses XDocument for the same kind of thing. Syntaxes are much easier than XmlDocument and it requires a minimal amount of code.

Also XDocument is much faster as XmlDocument. XmlDocument is an old and dirty solution for querying an XML document.

I am going to show some examples of [XmlDocument class](#) and [XDocument class](#) class:

Load XML file

```
string filename = @"C:\temp\test.xml";
```

X XmlDocument

```
 XmlDocument _doc = new XmlDocument();
_doc.Load(filename);
```

X XDocument

```
XDocument _doc = XDocument.Load(fileName);
```

Create XmlDocument

X XmlDocument

```
 XmlDocument doc = new XmlDocument();
XmlElement root = doc.CreateElement("root");
root.SetAttribute("name", "value");
XmlElement child = doc.CreateElement("child");
child.InnerText = "text node";
root.AppendChild(child);
doc.AppendChild(root);
```

X XDocument

```
XDocument doc = new XDocument(
```

```

new XElement("Root", new XAttribute("name", "value"),
new XElement("Child", "text node"))
);

/*result*/
<root name="value">
    <child>"TextNode"</child>
</root>

```

更改XML中节点的InnerText

XmlDocument

```

XmlNode node = _doc.SelectSingleNode("xmlRootNode");
node.InnerText = value;

```

XDocument

```

 XElement rootNote = _doc.XPathSelectElement("xmlRootNode");
rootNode.Value = "New Value";

```

编辑后保存文件

确保在任何更改后保存xml。

```

// 保存 XmlDocument 和 XDocument
_doc.保存(filename);

```

从XML中检索值

XmlDocument

```

XmlNode node = _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
string text = node.InnerText;

```

XDocument

```

 XElement node = _doc.XPathSelectElement("xmlRootNode/levelOneChildNode");
string text = node.Value;

```

从所有子元素中检索属性等于某值的值。

XmlDocument

```

List<string> valueList = new List<string>();
foreach (XmlNode n in nodelist)
{
    if(n.Attributes["type"].InnerText == "City")
    {
        valueList.Add(n.Attributes["type"].InnerText);
    }
}

```

```

new XElement("Root", new XAttribute("name", "value"),
new XElement("Child", "text node"))
);

/*result*/
<root name="value">
    <child>"TextNode"</child>
</root>

```

Change InnerText of node in XML

XmlDocument

```

XmlNode node = _doc.SelectSingleNode("xmlRootNode");
node.InnerText = value;

```

XDocument

```

 XElement rootNote = _doc.XPathSelectElement("xmlRootNode");
rootNode.Value = "New Value";

```

Save File after edit

Make sure to save the xml after any change.

```

// Save XmlDocument and XDocument
_doc.Save(filename);

```

Retrieve Values from XML

XmlDocument

```

XmlNode node = _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
string text = node.InnerText;

```

XDocument

```

 XElement node = _doc.XPathSelectElement("xmlRootNode/levelOneChildNode");
string text = node.Value;

```

Retrieve value from all from all child elements where attribute = something.

XmlDocument

```

List<string> valueList = new List<string>();
foreach (XmlNode n in nodelist)
{
    if(n.Attributes["type"].InnerText == "City")
    {
        valueList.Add(n.Attributes["type"].InnerText);
    }
}

```

```
}
```

XDocument

```
var accounts = _doc.XPathSelectElements("/data/summary/account").Where(c => c.Attribute("type").Value == "setting").Select(c => c.Value);
```

添加节点

XmlDocument

```
XmlNode nodeToAppend = doc.CreateElement("SecondLevelNode");
nodeToAppend.InnerText = "此标题由代码创建";

/* 将节点附加到父节点 */
XmlNode firstNode= _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
firstNode.AppendChild(nodeToAppend);

/* 修改后确保保存文档 */
_doc.Save(fileName);
```

XDocument

```
_doc.XPathSelectElement("ServerManagerSettings/TcpSocket").Add(new XElement("SecondLevelNode"));

/* 修改后确保保存文档 */
_doc.Save(fileName);
```

第69.2节：从XML文档读取

一个示例XML文件

```
<Sample>
<Account>
    <One number="12"/>
    <Two number="14"/>
</Account>
<Account>
    <One number="14"/>
    <Two number="16"/>
</Account>
</Sample>
```

从此XML文件读取：

```
using System.Xml;
using System.Collections.Generic;

public static void Main(string fullpath)
{
    var xmldoc = new XmlDocument();
    xmldoc.Load(fullpath);

    var oneValues = new List<string>();
```

```
}
```

XDocument

```
var accounts = _doc.XPathSelectElements("/data/summary/account").Where(c => c.Attribute("type").Value == "setting").Select(c => c.Value);
```

Append a node

XmlDocument

```
XmlNode nodeToAppend = doc.CreateElement("SecondLevelNode");
nodeToAppend.InnerText = "This title is created by code";

/* Append node to parent */
XmlNode firstNode= _doc.SelectSingleNode("xmlRootNode/levelOneChildNode");
firstNode.AppendChild(nodeToAppend);

/* After a change make sure to save the document */
_doc.Save(fileName);
```

XDocument

```
_doc.XPathSelectElement("ServerManagerSettings/TcpSocket").Add(new XElement("SecondLevelNode"));

/* After a change make sure to save the document */
_doc.Save(fileName);
```

Section 69.2: Reading from XML document

An example XML file

```
<Sample>
<Account>
    <One number="12"/>
    <Two number="14"/>
</Account>
<Account>
    <One number="14"/>
    <Two number="16"/>
</Account>
</Sample>
```

Reading from this XML file:

```
using System.Xml;
using System.Collections.Generic;

public static void Main(string fullpath)
{
    var xmldoc = new XmlDocument();
    xmldoc.Load(fullpath);

    var oneValues = new List<string>();
```

```

// 获取所有具有该标签名称的XML节点
var accountNodes = xmlDoc.GetElementsByTagName("Account");
for (var i = 0; i < accountNodes.Count; i++)
{
    // 使用 Xpath 查找节点
    var account = accountNodes[i].SelectSingleNode("./One");
    if (account != null && account.Attributes != null)
    {
        // 读取节点属性
        oneValues.Add(account.Attributes["number"].Value);
    }
}

```

第69.3节：基本的XML文档交互

```

public static void Main()
{
    var xml = new XmlDocument();
    var root = xml.CreateElement("element");
    // 创建一个属性，因此元素现在将是 "<element attribute='value' />"
    root.SetAttribute("attribute", "value");

    // 所有 XML 文档必须有且仅有一个根元素
    xml.AppendChild(root);

    // 向 XML 文档添加数据
    foreach (var dayOfWeek in Enum.GetNames((typeof(DayOfWeek))))
    {
        var day = xml.CreateElement("dayOfWeek");
        day.SetAttribute("name", dayOfWeek);

        // 别忘了将新值添加到当前文档中！
        root.AppendChild(day);
    }

    // 使用 XPath 查找数据；注意，大小写敏感
    var monday = xml.SelectSingleNode("//dayOfWeek[@name='Monday']");
    if (monday != null)
    {
        // 一旦获得特定节点的引用，可以通过其父节点导航并请求删除来删除它
        monday.ParentNode.RemoveChild(monday);
    }

    // 在屏幕上显示 XML 文档；可选择保存到文件
    xml.Save(Console.Out);
}

```

```

// Getting all XML nodes with the tag name
var accountNodes = xmlDoc.GetElementsByTagName("Account");
for (var i = 0; i < accountNodes.Count; i++)
{
    // Use Xpath to find a node
    var account = accountNodes[i].SelectSingleNode("./One");
    if (account != null && account.Attributes != null)
    {
        // Read node attribute
        oneValues.Add(account.Attributes["number"].Value);
    }
}

```

Section 69.3: Basic XML document interaction

```

public static void Main()
{
    var xml = new XmlDocument();
    var root = xml.CreateElement("element");
    // Creates an attribute, so the element will now be "<element attribute='value' />"
    root.SetAttribute("attribute", "value");

    // All XML documents must have one, and only one, root element
    xml.AppendChild(root);

    // Adding data to an XML document
    foreach (var dayOfWeek in Enum.GetNames((typeof(DayOfWeek))))
    {
        var day = xml.CreateElement("dayOfWeek");
        day.SetAttribute("name", dayOfWeek);

        // Don't forget to add the new value to the current document!
        root.AppendChild(day);
    }

    // Looking for data using XPath; BEWARE, this is case-sensitive
    var monday = xml.SelectSingleNode("//dayOfWeek[@name='Monday']");
    if (monday != null)
    {
        // Once you got a reference to a particular node, you can delete it
        // by navigating through its parent node and asking for removal
        monday.ParentNode.RemoveChild(monday);
    }

    // Displays the XML document in the screen; optionally can be saved to a file
    xml.Save(Console.Out);
}

```

第70章：XDocument 和 System.Xml.Linq 命名空间

第70.1节：生成 XML 文档

目标是生成以下 XML 文档：

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit ID="F0001">
    <FruitName>香蕉</FruitName>
    <FruitColor>黄色</FruitColor>
  </Fruit>
  <Fruit ID="F0002">
    <FruitName>苹果</FruitName>
    <FruitColor>红色</FruitColor>
  </Fruit>
</FruitBasket>
```

代码：

```
XNamespace xns = "http://www.fruitauthority.fake";
XDeclaration xDeclaration = new XDeclaration("1.0", "utf-8", "yes");
XDocument xDoc = new XDocument(xDeclaration);
 XElement xRoot = new XElement(xns + "FruitBasket");
xDoc.Add(xRoot);

 XElement xelFruit1 = new XElement(xns + "Fruit");
 XAttribute idAttribute1 = new XAttribute("ID", "F0001");
 xelFruit1.Add(idAttribute1);
 XElement xelFruitName1 = new XElement(xns + "FruitName", "Banana");
 XElement xelFruitColor1 = new XElement(xns + "FruitColor", "Yellow");
 xelFruit1.Add(xelFruitName1);
 xelFruit1.Add(xelFruitColor1);
 xRoot.Add(xelFruit1);

 XElement xelFruit2 = new XElement(xns + "Fruit");
 XAttribute idAttribute2 = new XAttribute("ID", "F0002");
 xelFruit2.Add(idAttribute2);
 XElement xelFruitName2 = new XElement(xns + "FruitName", "Apple");
 XElement xelFruitColor2 = new XElement(xns + "FruitColor", "Red");
 xelFruit2.Add(xelFruitName2);
 xelFruit2.Add(xelFruitColor2);
 xRoot.Add(xelFruit2);
```

第70.2节：使用流畅语法生成XML文档

目标：

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>香蕉</FruitName>
    <FruitColor>黄色</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>苹果</FruitName>
    <FruitColor>红色</FruitColor>
  </Fruit>
```

Chapter 70: XDocument and the System.Xml.Linq namespace

Section 70.1: Generate an XML document

The goal is to generate the following XML document:

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit ID="F0001">
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit ID="F0002">
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
</FruitBasket>
```

Code:

```
XNamespace xns = "http://www.fruitauthority.fake";
XDeclaration xDeclaration = new XDeclaration("1.0", "utf-8", "yes");
XDocument xDoc = new XDocument(xDeclaration);
 XElement xRoot = new XElement(xns + "FruitBasket");
xDoc.Add(xRoot);

 XElement xelFruit1 = new XElement(xns + "Fruit");
 XAttribute idAttribute1 = new XAttribute("ID", "F0001");
 xelFruit1.Add(idAttribute1);
 XElement xelFruitName1 = new XElement(xns + "FruitName", "Banana");
 XElement xelFruitColor1 = new XElement(xns + "FruitColor", "Yellow");
 xelFruit1.Add(xelFruitName1);
 xelFruit1.Add(xelFruitColor1);
 xRoot.Add(xelFruit1);

 XElement xelFruit2 = new XElement(xns + "Fruit");
 XAttribute idAttribute2 = new XAttribute("ID", "F0002");
 xelFruit2.Add(idAttribute2);
 XElement xelFruitName2 = new XElement(xns + "FruitName", "Apple");
 XElement xelFruitColor2 = new XElement(xns + "FruitColor", "Red");
 xelFruit2.Add(xelFruitName2);
 xelFruit2.Add(xelFruitColor2);
 xRoot.Add(xelFruit2);
```

Section 70.2: Generate an XML document using fluent syntax

Goal:

```
<FruitBasket xmlns="http://www.fruitauthority.fake">
  <Fruit>
    <FruitName>Banana</FruitName>
    <FruitColor>Yellow</FruitColor>
  </Fruit>
  <Fruit>
    <FruitName>Apple</FruitName>
    <FruitColor>Red</FruitColor>
  </Fruit>
```

```
</FruitBasket>
```

代码：

```
XNamespace xns = "http://www.fruitauthority.fake";
XDocument xDoc =
    new XDocument(new XDeclaration("1.0", "utf-8", "yes"),
        new XElement(xns + "FruitBasket",
            new XElement(xns + "Fruit",
                new XElement(xns + "FruitName", "香蕉"),
                new XElement(xns + "FruitColor", "黄色")),
            new XElement(xns + "Fruit",
                new XElement(xns + "FruitName", "苹果"),
                new XElement(xns + "FruitColor", "红色")))
    ));
```

第70.3节：修改XML文件

要使用XDocument修改XML文件，您需要将文件加载到XDocument类型的变量中，在内存中进行修改，然后保存，覆盖原始文件。一个常见错误是仅在内存中修改XML，却期望磁盘上的文件发生变化。

给定一个XML文件：

```
<?xml version="1.0" encoding="utf-8"?>
<FruitBasket xmlns="http://www.fruitauthority.fake">
    <Fruit>
        <FruitName>香蕉</FruitName>
        <FruitColor>黄色</FruitColor>
    </Fruit>
    <Fruit>
        <FruitName>苹果</FruitName>
        <FruitColor>红色</FruitColor>
    </Fruit>
</FruitBasket>
```

你想将香蕉的颜色修改为棕色：

1. 我们需要知道磁盘上文件的路径。
2. XDocument.Load 的一个重载接收一个 URI（文件路径）。
3. 由于 xml 文件使用了命名空间，我们必须使用命名空间和元素名称进行查询。
4. 一个使用 C# 6 语法的 Linq 查询，以适应可能出现的空值。查询中使用的每个 . 如果条件未找到元素，都有可能返回空集合。在 C# 6 之前，你需要分多步进行，沿途检查空值。结果是包含香蕉的 `<Fruit>` 元素。实际上是一个 `IEnumerable< XElement >`，这就是下一步使用 `FirstOrDefault()` 的原因。
5. 现在我们从刚找到的 Fruit 元素中提取 FruitColor 元素。这里我们假设只有一个，或者我们只关心第一个。
- 6.如果不为空，我们将 FruitColor 设置为“Brown”。
7. 最后，我们保存XDocument，覆盖磁盘上的原始文件。

```
// 1.
string xmlFilePath = "c:\\\\users\\\\public\\\\fruit.xml";

// 2.
XDocument xdoc = XDocument.Load(xmlFilePath);

// 3.
```

```
</FruitBasket>
```

Code:

```
XNamespace xns = "http://www.fruitauthority.fake";
XDocument xDoc =
    new XDocument(new XDeclaration("1.0", "utf-8", "yes"),
        new XElement(xns + "FruitBasket",
            new XElement(xns + "Fruit",
                new XElement(xns + "FruitName", "Banana"),
                new XElement(xns + "FruitColor", "Yellow")),
            new XElement(xns + "Fruit",
                new XElement(xns + "FruitName", "Apple"),
                new XElement(xns + "FruitColor", "Red")))
    ));
```

Section 70.3: Modify XML File

To modify an XML file with XDocument, you load the file into a variable of type XDocument, modify it in memory, then save it, overwriting the original file. A common mistake is to modify the XML in memory and expect the file on disk to change.

Given an XML file:

```
<?xml version="1.0" encoding="utf-8"?>
<FruitBasket xmlns="http://www.fruitauthority.fake">
    <Fruit>
        <FruitName>Banana</FruitName>
        <FruitColor>Yellow</FruitColor>
    </Fruit>
    <Fruit>
        <FruitName>Apple</FruitName>
        <FruitColor>Red</FruitColor>
    </Fruit>
</FruitBasket>
```

You want to modify the Banana's color to brown:

1. We need to know the path to the file on disk.
2. One overload of XDocument.Load receives a URI (file path).
3. Since the xml file uses a namespace, we must query with the namespace AND element name.
4. A Linq query utilizing C# 6 syntax to accommodate for the possibility of null values. Every . used in this query has the potential to return a null set if the condition finds no elements. Before C# 6 you would do this in multiple steps, checking for null along the way. The result is the `<Fruit>` element that contains the Banana. Actually an `IEnumerable< XElement >`, which is why the next step uses `FirstOrDefault()`.
5. Now we extract the FruitColor element out of the Fruit element we just found. Here we assume there is just one, or we only care about the first one.
6. If it is not null, we set the FruitColor to "Brown".
7. Finally, we save the XDocument, overwriting the original file on disk.

```
// 1.
string xmlFilePath = "c:\\\\users\\\\public\\\\fruit.xml";

// 2.
XDocument xdoc = XDocument.Load(xmlFilePath);

// 3.
```

```

XNamespace ns = "http://www.fruitauthority.fake";

//4.
var elBanana = xdoc.Descendants()?
    Elements(ns + "FruitName")?.
        Where(x => x.Value == "Banana")?.
    Ancestors(ns + "Fruit");

// 5.
var elColor = elBanana.Elements(ns + "FruitColor").FirstOrDefault();

// 6.
if (elColor != null)
{
    elColor.Value = "Brown";
}

// 7.
xdoc.保存(xmlFilePath);

```

文件现在看起来是这样的：

```

<?xml version="1.0" encoding="utf-8"?>
<FruitBasket xmlns="http://www.fruitauthority.fake">
    <Fruit>
        <FruitName>香蕉</FruitName>
        <FruitColor>棕色</FruitColor>
    </Fruit>
    <Fruit>
        <FruitName>苹果</FruitName>
        <FruitColor>红色</FruitColor>
    </Fruit>
</FruitBasket>

```

```

XNamespace ns = "http://www.fruitauthority.fake";

//4.
var elBanana = xdoc.Descendants()?
    Elements(ns + "FruitName")?.
        Where(x => x.Value == "Banana")?.
    Ancestors(ns + "Fruit");

// 5.
var elColor = elBanana.Elements(ns + "FruitColor").FirstOrDefault();

// 6.
if (elColor != null)
{
    elColor.Value = "Brown";
}

// 7.
xdoc.Save(xmlFilePath);

```

The file now looks like this:

```

<?xml version="1.0" encoding="utf-8"?>
<FruitBasket xmlns="http://www.fruitauthority.fake">
    <Fruit>
        <FruitName>Banana</FruitName>
        <FruitColor>Brown</FruitColor>
    </Fruit>
    <Fruit>
        <FruitName>Apple</FruitName>
        <FruitColor>Red</FruitColor>
    </Fruit>
</FruitBasket>

```

第71章：C# 7.0 特性

C# 7.0 是 C# 的第七个版本。该版本包含一些新特性：对元组 (Tuples)、局部函数、`out var` 声明、数字分隔符、二进制字面量、模式匹配、`throw` 表达式、`ref` 返回和 `ref` 局部变量 以及扩展的表达式主体成员列表的语言支持。

官方参考：[C# 7 新特性](#)

第71.1节：元组的语言支持

基础知识

元组 (tuple) 是一个有序的有限元素列表。元组在编程中常用作将多个元素作为一个整体来处理，而不是单独处理元组的每个元素，也用于表示关系数据库中的单独行（即“记录”）。

在 C# 7.0 中，方法可以有多个返回值。在幕后，编译器将使用新的 `ValueTuple` 结构体。

```
public (int sum, int count) GetTallies()
{
    return (1, 2);
}
```

附注：要在 Visual Studio 2017 中使用此功能，您需要安装 `System.ValueTuple` 包。

如果将返回元组的方法结果赋值给单个变量，可以通过方法签名中定义的成员名称访问这些成员：

```
var result = GetTallies();
// > result.sum
// 1
// > result.count
// 2
```

元组解构

元组解构是将元组拆分成其各个部分。

例如，调用 `GetTallies` 并将返回值赋给两个独立变量，即将元组解构到这两个变量中：

```
(int tallyOne, int tallyTwo) = GetTallies();
```

`var` 也适用：

```
(var s, var c) = GetTallies();
```

你也可以使用更简短的语法，在 () 外使用 `var`：

```
var (s, c) = GetTallies();
```

你也可以解构到已有变量中：

```
int s, c;
```

Chapter 71: C# 7.0 Features

C# 7.0 is the seventh version of C#. This version contains some new features: language support for Tuples, local functions, `out var` declarations, digit separators, binary literals, pattern matching, `throw` expressions, `ref return` and `ref` local and extended expression bodied members list.

Official reference: [What's new in C# 7](#)

Section 71.1: Language support for Tuples

Basics

A **tuple** is an ordered, finite list of elements. Tuples are commonly used in programming as a means to work with one single entity collectively instead of individually working with each of the tuple's elements, and to represent individual rows (ie. "records") in a relational database.

In C# 7.0, methods can have multiple return values. Behind the scenes, the compiler will use the new `ValueTuple` struct.

```
public (int sum, int count) GetTallies()
{
    return (1, 2);
}
```

Side note: for this to work in Visual Studio 2017, you need to get the `System.ValueTuple` package.

If a tuple-returning method result is assigned to a single variable you can access the members by their defined names on the method signature:

```
var result = GetTallies();
// > result.sum
// 1
// > result.count
// 2
```

Tuple Deconstruction

Tuple deconstruction separates a tuple into its parts.

For example, invoking `GetTallies` and assigning the return value to two separate variables deconstructs the tuple into those two variables:

```
(int tallyOne, int tallyTwo) = GetTallies();
```

`var` also works:

```
(var s, var c) = GetTallies();
```

You can also use shorter syntax, with `var` outside of ():

```
var (s, c) = GetTallies();
```

You can also deconstruct into existing variables:

```
int s, c;
```

```
(s, c) = GetTallies();
```

交换现在更简单了（不需要临时变量）：

```
(b, a) = (a, b);
```

有趣的是，任何对象都可以通过在类中定义一个Deconstruct方法来解构：

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public void Deconstruct(out string firstName, out string lastName)
    {
        firstName = FirstName;
        lastName = LastName;
    }
}

var person = new Person { FirstName = "John", LastName = "Smith" };
var (localFirstName, localLastName) = person;
```

在这种情况下，(localFirstName, localLastName) = person 语法是在调用person的Deconstruct方法。

解构甚至可以定义在扩展方法中。这与上述等效：

```
public static class PersonExtensions
{
    public static void Deconstruct(this Person person, out string firstName, out string lastName)
    {
        firstName = person.FirstName;
        lastName = person.LastName;
    }
}

var (localFirstName, localLastName) = person;
```

另一种为Person类定义Name的方法是将Name本身定义为一个Tuple。请考虑以下内容：

```
class Person
{
    public (string First, string Last) Name { get; }

    public Person((string FirstName, string LastName) name)
    {
        Name = name;
    }
}
```

然后你可以这样实例化一个person（我们可以将元组作为参数传入）：

```
var person = new Person(("Jane", "Smith"));

var firstName = person.Name.First; // "Jane"
var lastName = person.Name.Last; // "Smith"
```

元组初始化

```
(s, c) = GetTallies();
```

Swapping is now much simpler (no temp variable needed):

```
(b, a) = (a, b);
```

Interestingly, any object can be deconstructed by defining a Deconstruct method in the class:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public void Deconstruct(out string firstName, out string lastName)
    {
        firstName = FirstName;
        lastName = LastName;
    }
}

var person = new Person { FirstName = "John", LastName = "Smith" };
var (localFirstName, localLastName) = person;
```

In this case, the (localFirstName, localLastName) = person syntax is invoking Deconstruct on the person.

Deconstruction can even be defined in an extension method. This is equivalent to the above:

```
public static class PersonExtensions
{
    public static void Deconstruct(this Person person, out string firstName, out string lastName)
    {
        firstName = person.FirstName;
        lastName = person.LastName;
    }
}

var (localFirstName, localLastName) = person;
```

An alternative approach for the Person class is to define the Name itself as a Tuple. Consider the following:

```
class Person
{
    public (string First, string Last) Name { get; }

    public Person((string FirstName, string LastName) name)
    {
        Name = name;
    }
}
```

Then you can instantiate a person like so (where we can take a tuple as an argument):

```
var person = new Person(("Jane", "Smith"));

var firstName = person.Name.First; // "Jane"
var lastName = person.Name.Last; // "Smith"
```

Tuple Initialization

你也可以在代码中任意创建元组：

```
var name = ("John", "Smith");
Console.WriteLine(name.Item1);
// 输出 John

Console.WriteLine(name.Item2);
// 输出 Smith
```

创建元组时，可以为元组成员分配临时的项名称：

```
var name = (first: "John", middle: "Q", last: "Smith");
Console.WriteLine(name.first);
// 输出 John
```

类型推断

定义多个具有相同签名（匹配类型和数量）的元组时，将推断为匹配类型。例如：

```
public (int sum, double average) Measure(List<int> items)
{
    var stats = (sum: 0, average: 0d);
    stats.sum = items.Sum();
    stats.average = items.Average();
    return stats;
}
```

由于stats变量的声明与方法的返回签名匹配，因此可以返回stats。

反射和元组字段名称

成员名称在运行时不存在。即使成员名称不匹配，反射也会将具有相同数量和类型的元组视为相同。将元组转换为object类型，然后再转换为具有相同成员类型但不同名称的元组，也不会引发异常。

虽然ValueTuple类本身不保留成员名称的信息，但通过反射可以在TupleElementNamesAttribute中获取该信息。该属性不是应用于元组本身，而是应用于方法参数、返回值、属性和字段。这允许元组项名称在程序集之间得以保留，例如，如果一个方法返回(string name, int count)，那么名称name和count将对另一个程序集中的调用者可用，因为返回值会被标记为包含“name”和“count”值的TupleElementNameAttribute。

与泛型和async一起使用

新的元组特性（使用底层的ValueTuple类型）完全支持泛型，并且可以用作泛型类型参数。这使得它们可以与async/a wait模式一起使用：

```
public async Task<(string value, int count)> GetValueAsync()
{
    string fooBar = await _stackOverflow.GetStringAsync();
    int num = await _stackOverflow.GetIntAsync();

    return (fooBar, num);
}
```

与集合一起使用

You can also arbitrarily create tuples in code:

```
var name = ("John", "Smith");
Console.WriteLine(name.Item1);
// Outputs John

Console.WriteLine(name.Item2);
// Outputs Smith
```

When creating a tuple, you can assign ad-hoc item names to the members of the tuple:

```
var name = (first: "John", middle: "Q", last: "Smith");
Console.WriteLine(name.first);
// Outputs John
```

Type inference

Multiple tuples defined with the same signature (matching types and count) will be inferred as matching types. For example:

```
public (int sum, double average) Measure(List<int> items)
{
    var stats = (sum: 0, average: 0d);
    stats.sum = items.Sum();
    stats.average = items.Average();
    return stats;
}
```

stats can be returned since the declaration of the stats variable and the method's return signature are a match.

Reflection and Tuple Field Names

Member names do not exist at runtime. Reflection will consider tuples with the same number and types of members the same even if member names do not match. Converting a tuple to an object and then to a tuple with the same member types, but different names, will not cause an exception either.

While the ValueTuple class itself does not preserve information for member names the information is available through reflection in a TupleElementNamesAttribute. This attribute is not applied to the tuple itself but to method parameters, return values, properties and fields. This allows tuple item names to be preserved across assemblies i.e. if a method returns (string name, int count) the names name and count will be available to callers of the method in another assembly because the return value will be marked with TupleElementNameAttribute containing the values "name" and "count".

Use with generics and async

The new tuple features (using the underlying ValueTuple type) fully support generics and can be used as generic type parameter. That makes it possible to use them with the `async/await` pattern:

```
public async Task<(string value, int count)> GetValueAsync()
{
    string fooBar = await _stackOverflow.GetStringAsync();
    int num = await _stackOverflow.GetIntAsync();

    return (fooBar, num);
}
```

Use with collections

在某些场景中，拥有一个元组集合可能会带来好处，例如当你试图根据条件查找匹配的元组以避免代码分支时。

示例：

```
private readonly List<Tuple<string, string, string>> labels = new List<Tuple<string, string, string>>()
{
    new Tuple<string, string, string>("test1", "test2", "Value"),
    new Tuple<string, string, string>("test1", "test1", "Value2"),
    new Tuple<string, string, string>("test2", "test2", "Value3"),
};

public string FindMatchingValue(string firstElement, string secondElement)
{
    var result = labels
        .Where(w => w.Item1 == firstElement && w.Item2 == secondElement)
        .FirstOrDefault();

    if (result == null)
        throw new ArgumentException("combo not found");

    return result.Item3;
}
```

使用新的元组可以变成：

```
private readonly List<(string firstThingy, string secondThingyLabel, string foundValue)> labels =
new List<(string firstThingy, string secondThingyLabel, string foundValue)>()
{
    ("test1", "test2", "Value"),
    ("test1", "test1", "Value2"),
    ("test2", "test2", "Value3"),
}

public string FindMatchingValue(string firstElement, string secondElement)
{
    var result = labels
        .Where(w => w.firstThingy == firstElement && w.secondThingyLabel == secondElement)
        .FirstOrDefault();

    if (result == null)
        throw new ArgumentException("combo not found");

    return result.foundValue;
}
```

虽然上面示例元组的命名相当通用，但相关标签的概念使得对代码中尝试实现的内容有了更深入的理解，而不是仅仅引用“item1”、“item2”和“item3”。

ValueType 和 Tuple 之间的区别

引入ValueType的主要原因是性能。

类型名称	ValueType	Tuple
类或结构体	结构体	类
可变性 (创建后更改值)	可变的	不可变的
命名成员及其他语言支持	是	否 (待定)

It may become beneficial to have a collection of tuples in (as an example) a scenario where you're attempting to find a matching tuple based on conditions to avoid code branching.

Example:

```
private readonly List<Tuple<string, string, string>> labels = new List<Tuple<string, string, string>>()
{
    new Tuple<string, string, string>("test1", "test2", "Value"),
    new Tuple<string, string, string>("test1", "test1", "Value2"),
    new Tuple<string, string, string>("test2", "test2", "Value3"),
};

public string FindMatchingValue(string firstElement, string secondElement)
{
    var result = labels
        .Where(w => w.Item1 == firstElement && w.Item2 == secondElement)
        .FirstOrDefault();

    if (result == null)
        throw new ArgumentException("combo not found");

    return result.Item3;
}
```

With the new tuples can become:

```
private readonly List<(string firstThingy, string secondThingyLabel, string foundValue)> labels =
new List<(string firstThingy, string secondThingyLabel, string foundValue)>()
{
    ("test1", "test2", "Value"),
    ("test1", "test1", "Value2"),
    ("test2", "test2", "Value3"),
}

public string FindMatchingValue(string firstElement, string secondElement)
{
    var result = labels
        .Where(w => w.firstThingy == firstElement && w.secondThingyLabel == secondElement)
        .FirstOrDefault();

    if (result == null)
        throw new ArgumentException("combo not found");

    return result.foundValue;
}
```

Though the naming on the example tuple above is pretty generic, the idea of relevant labels allows for a deeper understanding of what is being attempted in the code over referencing "item1", "item2", and "item3".

Differences between ValueType and Tuple

The primary reason for introduction of ValueType is performance.

Type name	ValueType	Tuple
Class or structure	struct	class
Mutability (changing values after creation)	mutable	immutable
Naming members and other language support	yes	no (TBD)

- [GitHub 上的原始元组语言特性提案](#)
- [适用于 C# 7.0 特性的可运行 VS 15 解决方案](#)
- [NuGet 元组包](#)

第71.2节：局部函数

局部函数定义在方法内部，且在方法外不可用。它们可以访问所有局部变量，并支持迭代器、`async/await` 和 `lambda` 语法。通过这种方式，特定于函数的重复代码可以被函数化，而不会使类变得臃肿。副作用是，这提升了智能感知（intellisense）建议的性能。

示例

```
double 获取圆柱体积(double 半径, double 高度)
{
    return 获取体积();

    double 获取体积()
    {
        // 你可以在局部函数中声明内部局部函数
        double 获取圆面积(double r) => Math.PI * r * r;

        // 即使父函数没有任何输入，所有父函数的变量仍然可访问。
        return 获取圆面积(半径) * 高度;
    }
}
```

局部函数大大简化了 LINQ 操作符的代码，通常你需要将参数检查与实际逻辑分开，以便参数检查即时进行，而不是在迭代开始后才延迟执行。

示例

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    if (source == null) throw new ArgumentNullException(nameof(source));
    if (predicate == null) throw new ArgumentNullException(nameof(predicate));

    return iterator();

    IEnumerable<TSource> iterator()
    {
        foreach (TSource element in source)
            if (predicate(element))
                yield return element;
    }
}
```

局部函数也支持`async`和`await`关键字。

示例

```
async Task WriteEmailsAsync()
{
    var emailRegex = new Regex(@"(?i)[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+");
    IEnumerable<string> emails1 = await get EmailsFromFileSync("input1.txt");
    IEnumerable<string> emails2 = await get EmailsFromFileSync("input2.txt");
    await writeLinesToFileAsync(emails1.Concat(emails2), "output.txt");
```

References

- [Original Tuples language feature proposal on GitHub](#)
- [A runnable VS 15 solution for C# 7.0 features](#)
- [NuGet Tuple Package](#)

Section 71.2: Local functions

Local functions are defined within a method and aren't available outside of it. They have access to all local variables and support iterators, `async/await` and lambda syntax. This way, repetitions specific to a function can be functionalized without crowding the class. As a side effect, this improves intellisense suggestion performance.

Example

```
double GetCylinderVolume(double radius, double height)
{
    return getVolume();

    double getVolume()
    {
        // You can declare inner-local functions in a local function
        double GetCircleArea(double r) => Math.PI * r * r;

        // ALL parents' variables are accessible even though parent doesn't have any input.
        return GetCircleArea(radius) * height;
    }
}
```

Local functions considerably simplify code for LINQ operators, where you usually have to separate argument checks from actual logic to make argument checks instant, not delayed until after iteration started.

Example

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    if (source == null) throw new ArgumentNullException(nameof(source));
    if (predicate == null) throw new ArgumentNullException(nameof(predicate));

    return iterator();

    IEnumerable<TSource> iterator()
    {
        foreach (TSource element in source)
            if (predicate(element))
                yield return element;
    }
}
```

Local functions also support the `async` and `await` keywords.

Example

```
async Task WriteEmailsAsync()
{
    var emailRegex = new Regex(@"(?i)[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+");
    IEnumerable<string> emails1 = await get EmailsFromFileSync("input1.txt");
    IEnumerable<string> emails2 = await get EmailsFromFileSync("input2.txt");
    await writeLinesToFileAsync(emails1.Concat(emails2), "output.txt");
```

```

async Task<IEnumerable<string>> getEmailsFromFileAsync(string fileName)
{
    string text;

    using (StreamReader reader = File.OpenText(fileName))
    {
text = await reader.ReadToEndAsync();
    }

    return from Match emailMatch in emailRegex.Matches(text) select emailMatch.Value;
}

async Task writeLinesToFileAsync(IEnumerable<string> lines, string fileName)
{
    using (StreamWriter writer = File.CreateText(fileName))
    {
        foreach (string line in lines)
        {
            await writer.WriteLineAsync(line);
        }
    }
}

```

你可能注意到的一件重要事情是，本地函数可以定义在 `return` 语句之下，它们不需要定义在其上方。此外，本地函数通常遵循“lowerCamelCase”命名规范，以便更容易区分它们与类作用域函数。

第71.3节：out var 声明

C# 中一个常见的模式是使用 `bool TryParse(object input, out object value)` 来安全地解析对象。

`out var` 声明是一个简单的特性，用于提高可读性。它允许在作为 `out` 参数传递的同时声明变量。

以这种方式声明的变量的作用域是从声明点开始到其所在代码块的剩余部分。

示例

在 C# 7.0 之前使用 `TryParse`，必须先声明一个变量来接收值，然后再调用函数：

```

版本 < 7.0
int value;
if (int.TryParse(input, out value))
{
    Foo(value); // 可以
}
else
{
    Foo(value); // value 是零
}

Foo(value); // ok

```

在 C# 7.0 中，你可以内联声明传递给 `out` 参数的变量，省去了单独声明变量的需要：

```

版本 ≥ 7.0
if (int.TryParse(input, out var value))

```

```

async Task<IEnumerable<string>> getEmailsFromFileAsync(string fileName)
{
    string text;

    using (StreamReader reader = File.OpenText(fileName))
    {
        text = await reader.ReadToEndAsync();
    }

    return from Match emailMatch in emailRegex.Matches(text) select emailMatch.Value;
}

async Task writeLinesToFileAsync(IEnumerable<string> lines, string fileName)
{
    using (StreamWriter writer = File.CreateText(fileName))
    {
        foreach (string line in lines)
        {
            await writer.WriteLineAsync(line);
        }
    }
}

```

One important thing that you may have noticed is that local functions can be defined under the `return` statement, they do **not** need to be defined above it. Additionally, local functions typically follow the “lowerCamelCase” naming convention as to more easily differentiate themselves from class scope functions.

Section 71.3: out var declaration

A common pattern in C# is using `bool TryParse(object input, out object value)` to safely parse objects.

The `out var` declaration is a simple feature to improve readability. It allows a variable to be declared at the same time that it is passed as an `out` parameter.

A variable declared this way is scoped to the remainder of the body at the point in which it is declared.

Example

Using `TryParse` prior to C# 7.0, you must declare a variable to receive the value before calling the function:

```

Version < 7.0
int value;
if (int.TryParse(input, out value))
{
    Foo(value); // ok
}
else
{
    Foo(value); // value is zero
}

Foo(value); // ok

```

In C# 7.0, you can inline the declaration of the variable passed to the `out` parameter, eliminating the need for a separate variable declaration:

```

Version ≥ 7.0
if (int.TryParse(input, out var value))

```

```

{
    Foo(value); // 可以
}
else
{
    Foo(value); // value 是零
}

Foo(value); // 仍然可以, value 在后续代码块中有效

```

如果函数通过 `out` 返回的某些参数不需要，可以使用 `discard` 操作符 `_`。

```
p.GetCoordinates(out var x, out _); // 我只关心 x
```

`out var` 声明可以用于任何已有的带有 `out` 参数的函数。函数声明语法保持不变，也不需要额外条件来使函数兼容 `out var` 声明。这个特性只是语法糖。

`out var` 声明的另一个特性是它可以与匿名类型一起使用。

版本 ≥ 7.0

```

var a = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var groupedByMod2 = a.Select(x => new
{
    来源 = x,
    Mod2 = x % 2
})
    .按组分组(x => x.Mod2)
    .转为字典(g => g.Key, g => g.转为数组());
if (groupedByMod2.尝试获取值(1, out var 奇数元素))
{
    控制台.写行(奇数元素.长度);
}

```

在这段代码中，我们创建了一个键为int，值为匿名类型数组的Dictionary。在之前的C#版本中，无法在这里使用`TryGetValue`方法，因为它要求你声明`out`变量（该变量是匿名类型！）。然而，使用`out var`后，我们无需显式指定`out`变量的类型。

限制

请注意，`out var` 声明在LINQ查询中用途有限，因为表达式被解释为表达式lambda体，因此引入变量的作用域仅限于这些lambda。例如，以下代码将无法工作：

```

var nums =
    从 item 在 seq 中
    让 success = int.尝试解析(item, out var tmp)
        选择 success ? tmp : 0; // 错误：名称 'tmp' 在当前上下文中不存在

```

引用

- [GitHub 上的原始输出变量声明提案](#)

第71.4节：模式匹配

C#的模式匹配扩展使得许多函数式语言中的模式匹配优势得以实现，但以一种与底层语言风格无缝融合的方式呈现

```

{
    Foo(value); // ok
}
else
{
    Foo(value); // value is zero
}

Foo(value); // still ok, the value in scope within the remainder of the body

```

If some of the parameters that a function returns in `out` is not needed you can use the `discard` operator `_`.

```
p.GetCoordinates(out var x, out _); // I only care about x
```

An `out var` declaration can be used with any existing function which already has `out` parameters. The function declaration syntax remains the same, and no additional requirements are needed to make the function compatible with an `out var` declaration. This feature is simply syntactic sugar.

Another feature of `out var` declaration is that it can be used with anonymous types.

Version ≥ 7.0

```

var a = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var groupedByMod2 = a.Select(x => new
{
    Source = x,
    Mod2 = x % 2
})
    .GroupBy(x => x.Mod2)
    .ToDictionary(g => g.Key, g => g.ToArray());
if (groupedByMod2.TryGetValue(1, out var oddElements))
{
    Console.WriteLine(oddElements.Length);
}

```

In this code we create a Dictionary with `int` key and array of anonymous type value. In the previous version of C# it was impossible to use `TryGetValue` method here since it required you to declare the `out` variable (which is of anonymous type!). However, with `out var` we do not need to explicitly specify the type of the `out` variable.

Limitations

Note that `out var` declarations are of limited use in LINQ queries as expressions are interpreted as expression lambda bodies, so the scope of the introduced variables is limited to these lambdas. For example, the following code will not work:

```

var nums =
    from item in seq
    let success = int.TryParse(item, out var tmp)
        select success ? tmp : 0; // Error: The name 'tmp' does not exist in the current context

```

References

- [Original out var declaration proposal on GitHub](#)

Section 71.4: Pattern Matching

Pattern matching extensions for C# enable many of the benefits of pattern matching from functional languages, but in a way that smoothly integrates with the feel of the underlying language

switch 表达式

模式匹配扩展了 switch 语句，使其可以基于类型进行切换：

```
class Geometry {}

class Triangle : Geometry
{
    public int Width { get; set; }
    public int Height { get; set; }
    public int Base { get; set; }
}

class Rectangle : Geometry
{
    public int Width { get; set; }
    public int Height { get; set; }
}

class Square : Geometry
{
    public int Width { get; set; }
}

public static void PatternMatching()
{
    Geometry g = new Square { Width = 5 };

    switch (g)
    {
        case Triangle t:
            Console.WriteLine($"{t.Width} {t.Height} {t.Base}");
            break;
        case Rectangle sq when sq.Width == sq.Height:
            Console.WriteLine($"Square rectangle: {sq.Width} {sq.Height}");
            break;
        case Rectangle r:
            Console.WriteLine($"{r.Width} {r.Height}");
            break;
        case Square s:
            Console.WriteLine($"{s.Width}");
            break;
        default:
            Console.WriteLine("<other>");
            break;
    }
}
```

switch expression

Pattern matching extends the `switch` statement to switch on types:

```
class Geometry {}

class Triangle : Geometry
{
    public int Width { get; set; }
    public int Height { get; set; }
    public int Base { get; set; }
}

class Rectangle : Geometry
{
    public int Width { get; set; }
    public int Height { get; set; }
}

class Square : Geometry
{
    public int Width { get; set; }
}

public static void PatternMatching()
{
    Geometry g = new Square { Width = 5 };

    switch (g)
    {
        case Triangle t:
            Console.WriteLine($"{t.Width} {t.Height} {t.Base}");
            break;
        case Rectangle sq when sq.Width == sq.Height:
            Console.WriteLine($"Square rectangle: {sq.Width} {sq.Height}");
            break;
        case Rectangle r:
            Console.WriteLine($"{r.Width} {r.Height}");
            break;
        case Square s:
            Console.WriteLine($"{s.Width}");
            break;
        default:
            Console.WriteLine("<other>");
            break;
    }
}
```

is 表达式

模式匹配扩展了is运算符，用于同时检查类型并声明一个新变量。

示例

版本 < 7.0

```
string s = o as string;
if(s != null)
{
    // 对 s 进行某些操作
}
```

is expression

Pattern matching extends the `is` operator to check for a type and declare a new variable at the same time.

Example

Version < 7.0

```
string s = o as string;
if(s != null)
{
    // do something with s
}
```

可以重写为：

版本 ≥ 7.0

```
if(o is string s)
{
    //对 s 进行操作
};
```

还要注意，模式变量 s 的作用域扩展到了 if 块外，直到封闭作用域的末尾，例如：

```
if(someCondition)
{
    if(o is string s)
    {
        //对 s 进行操作
    }
    else
    {
        // 此处 s 未赋值，但可访问
    }
    // 此处 s 未赋值，但可访问
}
// 此处 s 不可访问
```

第 71.5 节：数字分隔符

下划线 _ 可用作数字分隔符。在大型数字字面量中能够对数字进行分组，对可读性有显著影响。

下划线可以出现在数字字面量的任何位置，但以下情况除外。不同的分组方式在不同场景或不同进制下可能更合理。

任意数字序列可以由一个或多个下划线分隔。下划线 (_) 允许出现在小数和指数中。分隔符没有语义影响——它们仅被忽略。

```
int bin = 0b1001_1010_0001_0100;
int hex = 0x1b_a0_44_fe;
int dec = 33_554_432;
int weird = 1_2__3___4____5_____6_____7_____8_____9;
double real = 1_000.111_1e-1_000;
```

以下情况不能使用数字分隔符：

- 在数值开头 (_121)
- 在数值结尾 (121_ 或 121.05_)
- 紧邻小数点 (10_.0)
- 紧邻指数符号 (1.1e_1)
- 紧邻类型说明符 (10_f)
- 紧跟在二进制和十六进制字面量中的0x或0b之后（可能会修改以允许例如 [0b 1001_1000](#)）

第71.6节：二进制字面量

可以使用0b前缀来表示二进制字面量。

can be rewritten as:

Version ≥ 7.0

```
if(o is string s)
{
    //Do something with s
};
```

Also note that the scope of the pattern variable s is extended to outside the if block reaching the end of the enclosing scope, example:

```
if(someCondition)
{
    if(o is string s)
    {
        //Do something with s
    }
    else
    {
        // s is unassigned here, but accessible
    }
    // s is unassigned here, but accessible
}
// s is not accessible here
```

Section 71.5: Digit separators

The underscore _ may be used as a digit separator. Being able to group digits in large numeric literals has a significant impact on readability.

The underscore may occur anywhere in a numeric literal except as noted below. Different groupings may make sense in different scenarios or with different numeric bases.

Any sequence of digits may be separated by one or more underscores. The _ is allowed in decimals as well as exponents. The separators have no semantic impact - they are simply ignored.

```
int bin = 0b1001_1010_0001_0100;
int hex = 0x1b_a0_44_fe;
int dec = 33_554_432;
int weird = 1_2__3___4____5_____6_____7_____8_____9;
double real = 1_000.111_1e-1_000;
```

Where the _ digit separator may not be used:

- at the beginning of the value (_121)
- at the end of the value (121_ or 121.05_)
- next to the decimal (10_.0)
- next to the exponent character (1.1e_1)
- next to the type specifier (10_f)
- immediately following the 0x or 0b in binary and hexadecimal literals ([might be changed to allow e.g. 0b 1001_1000](#))

Section 71.6: Binary literals

The **0b** prefix can be used to represent Binary literals.

二进制字面量允许用零和一构造数字，这使得查看数字的二进制表示中哪些位被设置变得更加容易。这对于处理二进制标志非常有用。

以下是指定值为34 (=25 + 21) 的int的等效方式：

```
// 使用二进制字面量：  
// 位：76543210  
int a1 = 0b00100010; // 二进制：显式指定位  
  
// 现有方法：  
int a2 = 0x22; // 十六进制：每个数字对应4位  
int a3 = 34; // 十进制：难以直观显示哪些位被设置  
int a4 = (1 << 5) | (1 << 1); // 位运算：组合非零位
```

标志枚举

之前，指定enum的标志值只能使用本例中的三种方法之一：

```
[Flags]  
public enum DaysOfWeek  
{  
    // 之前可用的方法：  
    //      十进制          十六进制          位移操作  
星期一    = 1,    //      = 0x01      = 1 << 0  
星期二    = 2,    //      = 0x02      = 1 << 1  
星期三    = 4,    //      = 0x04      = 1 << 2  
星期四    = 8,    //      = 0x08      = 1 << 3  
星期五    = 16,   //      = 0x10      = 1 << 4  
星期六    = 32,   //      = 0x20      = 1 << 5  
星期日    = 64,   //      = 0x40      = 1 << 6  
  
    工作日 = 星期一 | 星期二 | 星期三 | 星期四 | 星期五,  
    周末 = 星期六 | 星期日  
}
```

使用二进制字面量可以更清楚地看到哪些位被设置，且使用它们不需要理解十六进制数和按位运算：

```
[Flags]  
public enum DaysOfWeek  
{  
    星期一    = 0b00000001,  
    星期二    = 0b00000010,  
    星期三    = 0b00000100,  
    星期四    = 0b00001000,  
    星期五    = 0b00010000,  
    星期六    = 0b00100000,  
    Sunday    = 0b01000000,  
  
    Weekdays = Monday | Tuesday | Wednesday | Thursday | Friday,  
    Weekends = Saturday | Sunday  
}
```

第71.7节：throw表达式

C# 7.0允许在某些位置将throw作为表达式使用：

```
class Person  
{
```

Binary literals allow constructing numbers from zeroes and ones, which makes seeing which bits are set in the binary representation of a number much easier. This can be useful for working with binary flags.

The following are equivalent ways of specifying an int with value 34 (=25 + 21):

```
// Using a binary literal:  
// bits: 76543210  
int a1 = 0b00100010; // binary: explicitly specify bits  
  
// Existing methods:  
int a2 = 0x22; // hexadecimal: every digit corresponds to 4 bits  
int a3 = 34; // decimal: hard to visualise which bits are set  
int a4 = (1 << 5) | (1 << 1); // bitwise arithmetic: combining non-zero bits
```

Flags enumerations

Before, specifying flag values for an enum could only be done using one of the three methods in this example:

```
[Flags]  
public enum DaysOfWeek  
{  
    // Previously available methods:  
    //      decimal          hex            bit shifting  
Monday    = 1,    //      = 0x01      = 1 << 0  
Tuesday   = 2,    //      = 0x02      = 1 << 1  
Wednesday = 4,    //      = 0x04      = 1 << 2  
Thursday  = 8,    //      = 0x08      = 1 << 3  
Friday    = 16,   //      = 0x10      = 1 << 4  
Saturday  = 32,   //      = 0x20      = 1 << 5  
Sunday    = 64,   //      = 0x40      = 1 << 6  
  
    Weekdays = Monday | Tuesday | Wednesday | Thursday | Friday,  
    Weekends = Saturday | Sunday  
}
```

With binary literals it is more obvious which bits are set, and using them does not require understanding hexadecimal numbers and bitwise arithmetic:

```
[Flags]  
public enum DaysOfWeek  
{  
    Monday    = 0b00000001,  
    Tuesday   = 0b00000010,  
    Wednesday = 0b00000100,  
    Thursday  = 0b00001000,  
    Friday    = 0b00010000,  
    Saturday  = 0b00100000,  
    Sunday    = 0b01000000,  
  
    Weekdays = Monday | Tuesday | Wednesday | Thursday | Friday,  
    Weekends = Saturday | Sunday  
}
```

Section 71.7: throw expressions

C# 7.0 allows throwing as an expression in certain places:

```
class Person  
{
```

```

public string Name { get; }

public Person(string name) => Name = name ?? throw new ArgumentNullException(nameof(name));

public string GetFirstName()
{
    var parts = Name.Split(' ');
    return (parts.Length > 0) ? parts[0] : throw new InvalidOperationException("No name!");
}

public string GetLastName() => throw new NotImplementedException();
}

```

在 C# 7.0 之前，如果你想从表达式主体中抛出异常，你必须：

```

var spoons = "dinner,desert,soup".Split(',');
var spoonsArray = spoons.Length > 0 ? spoons : null;

if (spoonsArray == null)
{
    throw new Exception("没有勺子");
}

```

或者

```

var spoonsArray = spoons.Length > 0
    ? spoons
    : new Func<string[]>(() =>
    {
        throw new Exception("没有勺子");
    })();

```

在 C# 7.0 中，上述代码简化为：

```
var spoonsArray = spoons.Length > 0 ? spoons : throw new Exception("没有勺子");
```

第 71.8 节：扩展的表达式主体成员列表

C# 7.0 将访问器、构造函数和终结器添加到可以使用表达式主体的成员列表中：

```

class Person
{
    private static ConcurrentDictionary<int, string> names = new ConcurrentDictionary<int, string>();

    private int id = GetId();

    public Person(string name) => names.TryAdd(id, name); // 构造函数

    ~Person() => names.TryRemove(id, out _); // 析构函数

    public string Name
    {
        get => names[id]; // 取值器
        set => names[id] = value; // 赋值器
    }
}

```

```

public string Name { get; }

public Person(string name) => Name = name ?? throw new ArgumentNullException(nameof(name));

public string GetFirstName()
{
    var parts = Name.Split(' ');
    return (parts.Length > 0) ? parts[0] : throw new InvalidOperationException("No name!");
}

public string GetLastName() => throw new NotImplementedException();
}

```

Prior to C# 7.0, if you wanted to throw an exception from an expression body you would have to:

```

var spoons = "dinner,desert,soup".Split(',');
var spoonsArray = spoons.Length > 0 ? spoons : null;

if (spoonsArray == null)
{
    throw new Exception("There are no spoons");
}

```

Or

```

var spoonsArray = spoons.Length > 0
    ? spoons
    : new Func<string[]>(() =>
    {
        throw new Exception("There are no spoons");
    })();

```

In C# 7.0 the above is now simplified to:

```
var spoonsArray = spoons.Length > 0 ? spoons : throw new Exception("There are no spoons");
```

Section 71.8: Extended expression bodied members list

C# 7.0 adds accessors, constructors and finalizers to the list of things that can have expression bodies:

```

class Person
{
    private static ConcurrentDictionary<int, string> names = new ConcurrentDictionary<int, string>();

    private int id = GetId();

    public Person(string name) => names.TryAdd(id, name); // constructors

    ~Person() => names.TryRemove(id, out _); // finalizers

    public string Name
    {
        get => names[id]; // getters
        set => names[id] = value; // setters
    }
}

```

另请参见关于丢弃操作符的 `out var` 声明部分。

第71.9节：ref 返回和 ref 局部变量

`ref` 返回和 `ref` 局部变量对于操作和返回内存块的引用非常有用，而无需使用不安全的指针来复制内存。

参考返回

```
public static ref TValue 选择<TValue>(
    Func<bool> 条件, ref TValue 左值, ref TValue 右值)
{
    return 条件() ? ref 左值 : ref 右值;
}
```

通过这个方法，你可以传递两个引用值，并根据某个条件返回其中一个：

```
Matrix3D 左值 = ..., 右值 = ...;
选择(选择器, ref 左值, ref 右值).M20 = 1.0;
```

引用局部变量

```
public static ref int 最大值(ref int 第一个, ref int 第二个, ref int 第三个)
{
    ref int 最大 = 第一个 > 第二个 ? ref 第一个 : ref 第二个;
    return 最大 > 第三个 ? ref 最大 : ref 第三个;
}

int a = 1, b = 2, c = 3;
最大值(ref a, ref b, ref c) = 4;
Debug.Assert(a == 1); // true
Debug.Assert(b == 2); // true
Debug.Assert(c == 4); // true
```

不安全的引用操作

在 `System.Runtime.CompilerServices.Unsafe` 中，定义了一组不安全操作，允许你基本上像操作指针一样操作 `ref` 值。

例如，将内存地址 (`ref`) 重新解释为不同类型：

```
byte[] b = new byte[4] { 0x42, 0x42, 0x42, 0x42 };

ref int r = ref Unsafe.As<byte, int>(ref b[0]);
Assert.Equal(0x42424242, r);

0x0EF00EF0;
Assert.Equal(0xFE, b[0] | b[1] | b[2] | b[3]);
```

不过在执行此操作时要注意 [endianness](#) (字节序)，例如，如有需要请检查 `BitConverter.IsLittleEndian` 并相应处理。

或者以不安全的方式遍历数组：

```
int[] a = new int[] { 0x123, 0x234, 0x345, 0x456 };

ref int r1 = ref Unsafe.Add(ref a[0], 1);
Assert.Equal(0x234, r1);

ref int r2 = ref Unsafe.Add(ref r1, 2);
```

Also see the `out var` declaration section for the `discard` operator.

Section 71.9: ref return and ref local

`Ref` returns and `ref locals` are useful for manipulating and returning references to blocks of memory instead of copying memory without resorting to unsafe pointers.

Ref Return

```
public static ref TValue Choose<TValue>(
    Func<bool> condition, ref TValue left, ref TValue right)
{
    return condition() ? ref left : ref right;
}
```

With this you can pass two values by reference with one of them being returned based on some condition:

```
Matrix3D left = ..., right = ...;
Choose(chooser, ref left, ref right).M20 = 1.0;
```

Ref Local

```
public static ref int Max(ref int first, ref int second, ref int third)
{
    ref int max = first > second ? ref first : ref second;
    return max > third ? ref max : ref third;
}

int a = 1, b = 2, c = 3;
Max(ref a, ref b, ref c) = 4;
Debug.Assert(a == 1); // true
Debug.Assert(b == 2); // true
Debug.Assert(c == 4); // true
```

Unsafe Ref Operations

In `System.Runtime.CompilerServices.Unsafe` a set of unsafe operations have been defined that allow you to manipulate `ref` values as if they were pointers, basically.

For example, reinterpreting a memory address (`ref`) as a different type:

```
byte[] b = new byte[4] { 0x42, 0x42, 0x42, 0x42 };

ref int r = ref Unsafe.As<byte, int>(ref b[0]);
Assert.Equal(0x42424242, r);

0x0EF00EF0;
Assert.Equal(0xFE, b[0] | b[1] | b[2] | b[3]);
```

Beware of [endianness](#) when doing this, though, e.g. check `BitConverter.IsLittleEndian` if needed and handle accordingly.

Or iterate over an array in an unsafe manner:

```
int[] a = new int[] { 0x123, 0x234, 0x345, 0x456 };

ref int r1 = ref Unsafe.Add(ref a[0], 1);
Assert.Equal(0x234, r1);

ref int r2 = ref Unsafe.Add(ref r1, 2);
```

```
Assert.Equal(0x456, r2);

ref int r3 = ref Unsafe.Add(ref r2, -3);
Assert.Equal(0x123, r3);
```

或者类似的Subtract：

```
string[] a = new string[] { "abc", "def", "ghi", "jkl" };

ref string r1 = ref Unsafe.Subtract(ref a[0], -2);
Assert.Equal("ghi", r1);

ref string r2 = ref Unsafe.Subtract(ref r1, -1);
Assert.Equal("jkl", r2);

ref string r3 = ref Unsafe.Subtract(ref r2, 3);
Assert.Equal("abc", r3);
```

此外，可以检查两个ref值是否相同，即是否为相同地址：

```
long[] a = new long[2];

Assert.True(Unsafe.AreSame(ref a[0], ref a[0]));
Assert.False(Unsafe.AreSame(ref a[0], ref a[1]));
```

链接

[Roslyn Github 问题](#)

[System.Runtime.CompilerServices.Unsafe 在 GitHub 上](#)

第 71.10 节：ValueTask<T>

Task<T> 是一个类，当结果立即可用时，会导致不必要的分配开销。

ValueTask<T> 是一个结构体，旨在防止在等待时异步操作的结果已可用时分配Task对象。

因此，ValueTask<T> 提供了两个好处：

1. 性能提升

这是一个Task<T> 的示例：

- 需要堆分配
- 使用 JIT 需要 120 纳秒

```
async Task<int> TestTask(int d)
{
    await Task.Delay(d);
    return 10;
}
```

这是模拟的ValueTask<T>示例：

- 如果结果是同步已知的，则不会进行堆分配（但在本例中由于 Task.Delay 并非如此，然而在许多实际的 async/await 场景中通常是同步已知的）使用 JIT 时耗时 65 纳秒 Task.Delay，但在许多真实世界的async/await场景中通常是同步已知的）
 - 使用 JIT 时耗时 65 纳秒

```
Assert.Equal(0x456, r2);
```

```
ref int r3 = ref Unsafe.Add(ref r2, -3);
Assert.Equal(0x123, r3);
```

Or the similar Subtract:

```
string[] a = new string[] { "abc", "def", "ghi", "jkl" };

ref string r1 = ref Unsafe.Subtract(ref a[0], -2);
Assert.Equal("ghi", r1);

ref string r2 = ref Unsafe.Subtract(ref r1, -1);
Assert.Equal("jkl", r2);

ref string r3 = ref Unsafe.Subtract(ref r2, 3);
Assert.Equal("abc", r3);
```

Additionally, one can check if two `ref` values are the same i.e. same address:

```
long[] a = new long[2];

Assert.True(Unsafe.AreSame(ref a[0], ref a[0]));
Assert.False(Unsafe.AreSame(ref a[0], ref a[1]));
```

Links

[Roslyn Github Issue](#)

[System.Runtime.CompilerServices.Unsafe on github](#)

Section 71.10: ValueTask<T>

Task<T> is a **class** and causes the unnecessary overhead of its allocation when the result is immediately available.

ValueTask<T> is a **structure** and has been introduced to prevent the allocation of a Task object in case the result of the **async** operation is already available at the time of awaiting.

So ValueTask<T> provides two benefits:

1. Performance increase

Here's a Task<T> example:

- Requires heap allocation
- Takes 120ns with JIT

```
async Task<int> TestTask(int d)
{
    await Task.Delay(d);
    return 10;
}
```

Here's the analog ValueTask<T> example:

- No heap allocation if the result is known synchronously (which it is not in this case because of the Task.Delay, but often is in many real-world **async/await** scenarios)
- Takes 65ns with JIT

```
async ValueTask<int> TestValueTask(int d)
{
    await Task.Delay(d);
    return 10;
}
```

2. 增加实现的灵活性

希望同步执行的异步接口实现，否则将被迫使用 Task.Run 或 Task.FromResult（导致上述性能损失）。因此对同步实现存在一定压力。Task.Run 或 Task.FromResult（导致上述性能损失）。因此对同步实现存在一定压力。

但使用 ValueTask<T>，实现可以更自由地选择同步或异步，而不会影响调用者。

例如，下面是一个带有异步方法的接口：

```
interface IFoo<T>
{
    ValueTask<T> BarAsync();
}
```

...下面是该方法可能的调用方式：

```
IFoo<T> 事物 = getThing();
var x = await 事物.BarAsync();
```

使用ValueTask，上述代码可以适用于同步或异步实现：

同步实现：

```
class 同步Foo<T> : IFoo<T>
{
    public ValueTask<T> BarAsync()
    {
        var 值 = default(T);
        return new ValueTask<T>(值);
    }
}
```

异步实现

```
class 异步Foo<T> : IFoo<T>
{
    public async ValueTask<T> BarAsync()
    {
        var value = default(T);
        await Task.Delay(1);
        return value;
    }
}
```

注意事项

虽然ValueTask结构体原计划添加到C# 7.0中，但目前仍作为另一个库保留。ValueTask<T> System.Threading.Tasks.Extensions包可以从Nuget Gallery下载

```
async ValueTask<int> TestValueTask(int d)
{
    await Task.Delay(d);
    return 10;
}
```

2. Increased implementation flexibility

Implementations of an async interface wishing to be synchronous would otherwise be forced to use either Task.Run or Task.FromResult (resulting in the performance penalty discussed above). Thus there's some pressure against synchronous implementations.

But with ValueTask<T>, implementations are more free to choose between being synchronous or asynchronous without impacting callers.

For example, here's an interface with an asynchronous method:

```
interface IFoo<T>
{
    ValueTask<T> BarAsync();
}
```

...and here's how that method might be called:

```
IFoo<T> 事物 = getThing();
var x = await 事物.BarAsync();
```

With ValueTask, the above code will work with **either synchronous or asynchronous implementations**:

Synchronous implementation:

```
class SynchronousFoo<T> : IFoo<T>
{
    public ValueTask<T> BarAsync()
    {
        var value = default(T);
        return new ValueTask<T>(value);
    }
}
```

Asynchronous implementation

```
class AsynchronousFoo<T> : IFoo<T>
{
    public async ValueTask<T> BarAsync()
    {
        var value = default(T);
        await Task.Delay(1);
        return value;
    }
}
```

Notes

Although ValueTask struct was being planned to be added to C# 7.0, it has been kept as another library for the time being. ValueTask<T> System.Threading.Tasks.Extensions package can be downloaded from [Nuget Gallery](#)

第72章：C# 6.0 特性

C#语言的第六次迭代由Roslyn编译器提供。该编译器随.NET Framework 4.6版本发布，但它可以以向后兼容的方式生成代码，以支持更早的框架版本。C# 6代码可以完全向后兼容地编译到.NET 4.0。它也可以用于更早的框架，但某些需要额外框架支持的功能可能无法正常工作。

第72.1节：异常过滤器

异常过滤器允许开发者向catch块添加一个条件（以boolean表达式的形式），使得catch只有在条件计算为true时才执行。

异常过滤器允许原始异常中的调试信息继续传播，而在catch块内使用if语句并重新抛出异常会阻止原始异常中调试信息的传播。使用异常过滤器时，异常会继续向调用堆栈向上传播，除非满足条件。因此，异常过滤器使调试体验更加轻松。调试器不会停在throw语句上，而是停在抛出异常的语句上，当前状态和所有局部变量都会被保留。崩溃转储也会以类似方式受到影响。

异常过滤器自CLR诞生以来就被支持，并且通过暴露CLR异常处理模型的一部分，VB.NET和F#已经可以使用该功能超过十年。直到C# 6.0发布后，该功能才对C#开发者开放。

使用异常过滤器

异常过滤器通过在catch表达式后附加when子句来使用。可以在when子句中使用任何返回bool的表达式（await除外）。声明的异常变量ex可以在when子句内访问：

```
var SqlErrorToIgnore = 123;
try
{
    执行SQL操作();
}
catch (SqlException ex) when (ex.Number != SqlErrorToIgnore)
{
    throw new Exception("访问数据库时发生错误", ex);
}
```

可以组合多个带有when子句的catch块。第一个返回true的when子句会导致异常被捕获。将进入其catch块，而其他catch子句将被忽略（它们的when子句不会被评估）。例如：

```
try
{ ... }
catch (Exception ex) when (someCondition) //如果someCondition计算结果为true,
                                //其余的catch将被忽略。
{ ... }
catch (NotImplementedException ex) when (someMethod()) //someMethod() 仅在
                                //someCondition 计算结果为 false 时运行
{ ... }
```

Chapter 72: C# 6.0 Features

This sixth iteration of the C# language is provided by the Roslyn compiler. This compiler came out with version 4.6 of the .NET Framework, however it can generate code in a backward compatible manner to allow targeting earlier framework versions. C# version 6 code can be compiled in a fully backwards compatible manner to .NET 4.0. It can also be used for earlier frameworks, however some features that require additional framework support may not function correctly.

Section 72.1: Exception filters

[Exception filters](#) give developers the ability to add a condition (in the form of a boolean expression) to a catch block, allowing the `catch` to execute only if the condition evaluates to `true`.

Exception filters allow the propagation of debug information in the original exception, where as using an `if` statement inside a `catch` block and re-throwing the exception stops the propagation of debug information in the original exception. With exception filters, the exception continues to propagate upwards in the call stack *unless* the condition is met. As a result, exception filters make the debugging experience much easier. Instead of stopping on the `throw` statement, the debugger will stop on the statement throwing the exception, with the current state and all local variables preserved. Crash dumps are affected in a similar way.

Exception filters have been supported by the [CLR](#) since the beginning and they've been accessible from VB.NET and F# for over a decade by exposing a part of the CLR's exception handling model. Only after the release of C# 6.0 has the functionality also been available for C# developers.

Using exception filters

Exception filters are utilized by appending a `when` clause to the `catch` expression. It is possible to use any expression returning a `bool` in a `when` clause (except `await`). The declared Exception variable `ex` is accessible from within the `when` clause:

```
var SqlErrorToIgnore = 123;
try
{
    DoSQLOperations();
}
catch (SqlException ex) when (ex.Number != SqlErrorToIgnore)
{
    throw new Exception("An error occurred accessing the database", ex);
}
```

Multiple `catch` blocks with `when` clauses may be combined. The first `when` clause returning `true` will cause the exception to be caught. Its `catch` block will be entered, while the other `catch` clauses will be ignored (their `when` clauses won't be evaluated). For example:

```
try
{ ... }
catch (Exception ex) when (someCondition) //If someCondition evaluates to true,
                                         //the rest of the catches are ignored.
{ ... }
catch (NotImplementedException ex) when (someMethod()) //someMethod() will only run if
                                         //someCondition evaluates to false
{ ... }
```

```
catch(Exception ex) // 如果两个 when 子句都计算为 false
{ ... }
```

有风险的 when 子句

注意

使用异常过滤器可能存在风险：当 Exception 在 when 子句中抛出时，来自 when 子句的 Exception 会被忽略并视为 false。此方法允许开发者编写 when 子句时无需处理无效情况。

下面的示例说明了这种情况：

```
public static void Main()
{
    int a = 7;
    int b = 0;
    try
    {
        DoSomethingThatMightFail();
    }
    catch (Exception ex) when (a / b == 0)
    {
        // 该代码块永远不会被执行，因为 a / b 抛出一个被忽略的
        // DivideByZeroException，该异常被视为 false。
    }
    catch (Exception ex)
    {
        // 由于忽略了前面 when 子句中的 DivideByZeroException，程序会进入此代码
        // 块。
    }
}

public static void DoSomethingThatMightFail()
{
    // 这将始终抛出 ArgumentNullException。
    Type.GetType(null);
}
```

查看演示

请注意，异常过滤器避免了在失败代码位于同一函数内使用 throw 时出现的令人困惑的行号问题。例如，在此情况下，报告的行号是 6 而不是 3：

```
1. int a = 0, b = 0;
2. try {
3.     int c = a / b;
4. }
5. 捕获 (DivideByZeroException) {
6.     抛出;
7. }
```

异常行号报告为6，因为错误在第6行通过throw语句被捕获并重新抛出。

异常过滤器不会发生同样的情况：

```
catch(Exception ex) // If both when clauses evaluate to false
{ ... }
```

Risky when clause

Caution

It can be risky to use exception filters: when an Exception is thrown from within the when clause, the Exception from the when clause is ignored and is treated as `false`. This approach allows developers to write when clause without taking care of invalid cases.

The following example illustrates such a scenario:

```
public static void Main()
{
    int a = 7;
    int b = 0;
    try
    {
        DoSomethingThatMightFail();
    }
    catch (Exception ex) when (a / b == 0)
    {
        // This block is never reached because a / b throws an ignored
        // DivideByZeroException which is treated as false.
    }
    catch (Exception ex)
    {
        // This block is reached since the DivideByZeroException in the
        // previous when clause is ignored.
    }
}

public static void DoSomethingThatMightFail()
{
    // This will always throw an ArgumentNullException.
    Type.GetType(null);
}
```

View Demo

Note that exception filters avoid the confusing line number problems associated with using `throw` when failing code is within the same function. For example in this case the line number is reported as 6 instead of 3:

```
1. int a = 0, b = 0;
2. try {
3.     int c = a / b;
4. }
5. catch (DivideByZeroException) {
6.     throw;
7. }
```

The exception line number is reported as 6 because the error was caught and re-thrown with the `throw` statement on line 6.

The same does not happen with exception filters:

```

1. int a = 0, b = 0;
2. try {
3.     int c = a / b;
4. }
5. 捕获 (DivideByZeroException) 当 (a != 0) {
6.     抛出;
7. }

```

在此示例中，`a`为0，因此`catch`子句被忽略，但报告的行号是3。这是因为它们不会展开堆栈。更具体地说，异常并未在第5行被捕获，因为`a`实际上等于0，因此第6行没有执行，异常也没有机会在第6行被重新抛出。

作为副作用的日志记录

条件中的方法调用可能会导致副作用，因此异常过滤器可用于在异常发生时运行代码而不捕获异常。一个常见的利用此特性的例子是一个总是返回`false`的`Log`方法。这允许在调试时跟踪日志信息，而无需重新抛出异常。

请注意，虽然这看起来是一种方便的日志记录方式，但可能存在风险，尤其是在使用第三方日志程序集时。这些程序集可能会在日志记录过程中抛出异常，且这些异常在非明显情况下可能不易被发现（参见上文的风险`when(...)`条款）。

```

try
{
DoSomethingThatMightFail(s);
}
catch (Exception ex)
(Log(ex, "发生错误")) { // 该catch块永远不会被执行 } // ... static bool Log(Exception ex, string
message, params object[] args) { Debug.Print(message, args); return false; }

```

查看演示

在之前的C#版本中，常见的做法是记录日志并重新抛出异常。

版本 < 6.0

```

try
{
DoSomethingThatMightFail(s);
}
catch (Exception ex)
{
Log(ex, "发生错误");
    throw;
}

// ...

static void Log(Exception ex, string message, params object[] args)
{
Debug.Print(message, args);
}

```

查看演示

```

1. int a = 0, b = 0;
2. try {
3.     int c = a / b;
4. }
5. catch (DivideByZeroException) when (a != 0) {
6.     throw;
7. }

```

In this example `a` is 0 then `catch` clause is ignored but 3 is reported as line number. This is because they **do not unwind the stack**. More specifically, the exception *is not caught* on line 5 because `a` in fact does equal 0 and thus there is no opportunity for the exception to be re-thrown on line 6 because line 6 does not execute.

Logging as a side effect

Method calls in the condition can cause side effects, so exception filters can be used to run code on exceptions without catching them. A common example that takes advantage of this is a `Log` method that always returns `false`. This allows tracing log information while debugging without the need to re-throw the exception.

Be aware that while this seems to be a comfortable way of logging, it can be risky, especially if 3rd party logging assemblies are used. These might throw exceptions while logging in non-obvious situations that may not be detected easily (see **Risky `when(...)` clause** above).

```

try
{
DoSomethingThatMightFail(s);
}
catch (Exception ex)
(Log(ex, "An error occurred")) { // This catch block will never be reached } // ... static bool Log(Exception ex, string
message, params object[] args) { Debug.Print(message, args); return false; }

```

View Demo

The common approach in previous versions of C# was to log and re-throw the exception.

Version < 6.0

```

try
{
DoSomethingThatMightFail(s);
}
catch (Exception ex)
{
    Log(ex, "An error occurred");
    throw;
}

// ...

static void Log(Exception ex, string message, params object[] args)
{
Debug.Print(message, args);
}

```

View Demo

finally块

finally块无论是否抛出异常都会执行。关于when中的表达式，有一个细节是异常过滤器会在进入内部finally块之前在调用栈更高层执行。这可能导致当代码试图修改全局状态（例如当前线程的用户或文化）并在finally块中将其设置回去时，出现意外的结果和行为。

示例：finally块

```
private static bool Flag = false;

static void Main(string[] args)
{
    Console.WriteLine("Start");
    try
    {
        SomeOperation();
    }
    catch (Exception) when (EvaluatesTo())
    {
        Console.WriteLine("Catch");
    }
    finally
    {
        Console.WriteLine("Outer Finally");
    }
}

private static bool EvaluatesTo()
{
    Console.WriteLine($"EvaluatesTo: {Flag}");
    return true;
}

private static void SomeOperation()
{
    try
    {
        Flag = true;
        throw new Exception("Boom");
    }
    finally
    {
        Flag = false;
    }
    Console.WriteLine("Inner Finally");
}
}
```

生成的输出：

```
开始
求值为：真
内部最终
捕获
外层最终块
```

[查看演示](#)

The finally block

The `finally` block executes every time whether the exception is thrown or not. One subtlety with expressions in `when` is exception filters are executed further up the stack *before* entering the inner `finally` blocks. This can cause unexpected results and behaviors when code attempts to modify global state (like the current thread's user or culture) and set it back in a `finally` block.

Example: finally block

```
private static bool Flag = false;

static void Main(string[] args)
{
    Console.WriteLine("Start");
    try
    {
        SomeOperation();
    }
    catch (Exception) when (EvaluatesTo())
    {
        Console.WriteLine("Catch");
    }
    finally
    {
        Console.WriteLine("Outer Finally");
    }
}

private static bool EvaluatesTo()
{
    Console.WriteLine($"EvaluatesTo: {Flag}");
    return true;
}

private static void SomeOperation()
{
    try
    {
        Flag = true;
        throw new Exception("Boom");
    }
    finally
    {
        Flag = false;
        Console.WriteLine("Inner Finally");
    }
}
}
```

Produced Output:

```
Start
EvaluatesTo: True
Inner Finally
Catch
Outer Finally
```

[View Demo](#)

在上面的例子中，如果方法SomeOperation不希望将全局状态的更改“泄漏”到调用者的when子句中，它也应该包含一个catch块来修改状态。例如：

```
private static void SomeOperation()
{
    try
    {
        Flag = true;
        throw new Exception("Boom");
    }
    catch
    {
        Flag = false;
        throw;
    }
    finally
    {
        Flag = false;
        Console.WriteLine("Inner Finally");
    }
}
```

通常也会看到IDisposable辅助类利用using块的语义来实现相同的目标，因为IDisposable的Dispose方法总会在using块内抛出的异常开始向上传播之前被调用。

第72.2节：字符串插值

字符串插值允许开发者将变量和文本组合成一个字符串。

基本示例

创建了两个int变量：foo和bar。

```
int foo = 34;
int bar = 42;

string resultString = $"foo 是 {foo}, bar 是 {bar}。";

Console.WriteLine(resultString);
```

输出:

```
foo 是 34, bar 是 42。
```

查看演示

字符串中的大括号仍然可以使用，比如这样：

```
var foo = 34;
var bar = 42;

// 字符串插值表示法（新样式）
Console.WriteLine($"foo 是
o}}, bar 是 {{bar}}。");
```

In the example above, if the method SomeOperation does not wish to "leak" the global state changes to caller's when clauses, it should also contain a catch block to modify the state. For example:

```
private static void SomeOperation()
{
    try
    {
        Flag = true;
        throw new Exception("Boom");
    }
    catch
    {
        Flag = false;
        throw;
    }
    finally
    {
        Flag = false;
        Console.WriteLine("Inner Finally");
    }
}
```

It is also common to see IDisposable helper classes leveraging the semantics of using blocks to achieve the same goal, as IDisposable.Dispose will always be called before an exception called within a using block starts bubbling up the stack.

Section 72.2: String interpolation

String interpolation allows the developer to combine variables and text to form a string.

Basic Example

Two int variables are created: foo and bar.

```
int foo = 34;
int bar = 42;

string resultString = $"The foo is {foo}, and the bar is {bar}.";
Console.WriteLine(resultString);
```

Output:

```
The foo is 34, and the bar is 42.
```

View Demo

Braces within strings can still be used, like this:

```
var foo = 34;
var bar = 42;

// String interpolation notation (new style)
Console.WriteLine($"The foo is
o}}, and the bar is {{bar}}.");
```

这将产生以下输出：

```
foo 是 {foo}, bar 是 {bar}。
```

使用带插值的逐字字符串字面量

在字符串前使用@会使字符串被逐字解释。因此，例如，Unicode 字符或换行符将保持它们被输入时的样子。然而，这不会影响插值字符串中的表达式，如下面的示例所示：

```
Console.WriteLine($"如果还不清楚的话：
```

```
\u00B9
```

```
foo
```

```
是
```

```
}, bar 是 {bar}。"); 输出：
```

```
如果还不清楚的话：
```

```
\u00B9
```

```
foo
```

```
是 34,
```

```
bar
```

```
是 42。
```

[查看演示](#)

表达式

使用字符串插值时，花括号{}内的表达式也可以被计算。结果将被插入到字符串中的相应位置。例如，要计算foo和bar的最大值并插入，可以使用

```
花括号内的Math.Max：Console.WriteLine($"较大的是：{Math.Max(foo, bar)}");
```

输出：

```
较大的是：42
```

注意：花括号与表达式之间的任何前导或尾随空白（包括空格、制表符和回车换行/换行符）都会被完全忽略，不包含在输出中

[查看演示](#)

另一个例子，变量可以格式化为货币形式：Console.WriteLine(\$"Foo格式化为四位小数的货币：{c4}");

输出：

```
Foo格式化为四位小数的货币：$34.0000
```

This produces the following output:

```
The foo is {foo}, and the bar is {bar}.
```

Using interpolation with verbatim string literals

Using @ before the string will cause the string to be interpreted verbatim. So, e.g. Unicode characters or line breaks will stay exactly as they've been typed. However, this will not effect the expressions in an interpolated string as shown in the following example:

```
Console.WriteLine($"In case it wasn't clear:
```

```
\u00B9
```

```
The foo
```

```
is
```

```
}, and the bar is {bar}。"); Output:
```

```
In case it wasn't clear:
```

```
\u00B9
```

```
The foo
```

```
is 34,
```

```
and the bar
```

```
is 42.
```

[View Demo](#)

Expressions

With string interpolation, *expressions* within curly braces {} can also be evaluated. The result will be inserted at the corresponding location within the string. For example, to calculate the maximum of foo and bar and insert it, use Math.Max within the curly braces:Console.WriteLine(\$"And the greater one is: {Math.Max(foo, bar)}");

Output:

```
And the greater one is: 42
```

Note: Any leading or trailing whitespace (including space, tab and CRLF/newline) between the curly brace and the expression is completely ignored and not included in the output

[View Demo](#)

As another example, variables can be formatted as a currency:Console.WriteLine(\$"Foo formatted as a currency to 4 decimal places: {c4}");

Output:

```
Foo formatted as a currency to 4 decimal places: $34.0000
```

[查看演示](#)

或者它们可以格式化为日期：Console.WriteLine(\$"今天是：{eTime.Today:dddd, MMMM dd - yyyy}");

输出：

今天是：2015年7月20日，星期一

[查看演示](#)

带有**条件（三元）运算符**的语句也可以在插值中进行计算。但是，这些必须用括号括起来，因为冒号否则会被用来表示如上所示的格式化：

```
Console.WriteLine($"{(foo > bar ? "Foo 大于 bar!" : "Bar 大于 foo!")}");
```

输出：

Bar 大于 foo!

[查看演示](#)

条件表达式和格式说明符可以混合使用：

```
Console.WriteLine($"环境：{((Environment.Is64BitProcess ? 64 : 32):00'位') 进程}");
```

输出：

环境：32-位 进程

转义序列

转义反斜杠 (\) 和引号 ("") 字符在插值字符串中与非插值字符串中完全相同，无论是逐字字符串还是非逐字字符串字面量：

```
Console.WriteLine($"Foo 是：{}`). 在非逐字字符串中，我们需要用反斜杠转义 \" 和 \\。"; Console.WriteLine($@"Foo 是：{foo}）。在逐字字符串中，我们需要用双引号转义 "", 但不需要转义 \"");
```

输出：

Foo 是 34。在非逐字字符串中，我们需要用反斜杠转义 " 和 \。

Foo 是 34。在逐字字符串中，我们需要用额外的引号来转义 "，但不需要转义 \

要在插值字符串中包含大括号 { 或 }，使用两个大括号 {{ 或 }}：\$"{{foo}} 是："

输出：

{foo} 是：34

[View Demo](#)

Or they can be formatted as dates:Console.WriteLine(\$"Today is: {eTime.Today:dddd, MMMM dd - yyyy}");

Output:

Today is: Monday, July, 20 - 2015

[View Demo](#)

Statements with a **Conditional (Ternary) Operator** can also be evaluated within the interpolation. However, these must be wrapped in parentheses, since the colon is otherwise used to indicate formatting as shown above:

```
Console.WriteLine($"{(foo > bar ? "Foo is larger than bar!" : "Bar is larger than foo!")}");
```

Output:

Bar is larger than foo!

[View Demo](#)

Conditional expressions and format specifiers can be mixed:

```
Console.WriteLine($"Environment: {((Environment.Is64BitProcess ? 64 : 32):00'bit') process}");
```

Output:

Environment: 32-bit process

Escape sequences

Escaping backslash (\) and quote (") characters works exactly the same in interpolated strings as in non-interpolated strings, for both verbatim and non-verbatim string literals:

```
Console.WriteLine($"Foo is: `). In a non-verbatim string, we need to escape \" and \\ with backslashes.");
Console.WriteLine($@"Foo is: {foo}). In a verbatim string, we need to escape "" with an extra quote, but we don't need to escape \");
```

Output:

Foo is 34. In a non-verbatim string, we need to escape " and \ with backslashes.

Foo is 34. In a verbatim string, we need to escape " with an extra quote, but we don't need to escape \

To include a curly brace { or } in an interpolated string, use two curly braces {{ or }}:\$"{{foo}} is: {"

Output:

{foo} is: 34

FormattableString 类型

```
$"..." 字符串插值表达式的类型不总是简单的字符串。编译器根据上下文决定分配哪种类型：string s = $"hello, {name}"; System.FormattableString s = $"Hello, {name}"; System.IFormattable s = $"Hello, {name}";
```

当编译器需要选择调用哪个重载方法时，这也是类型优先顺序。

一个新类型，System.FormattableString，表示一个复合格式字符串，以及要格式化的参数。使用它来编写专门处理插值参数的应用程序：

```
public void AddLogItem(FormattableString formattableString)
{
    foreach (var arg in formattableString.GetArguments())
    {
        // 对插值参数 'arg' 进行某些操作
    }

    // 使用标准插值和当前文化信息
    // 获取普通字符串：
    var formatted = formattableString.ToString();

    // ...
}
```

调用上述方法：AddLogItem(\$"foo 是 {bar}"); 例如，如果日志级别已经会过滤该日志项，则可以选择不产生格式化字符串的性能开销。

隐式转换

插值字符串存在隐式类型转换：var s = \$"Foo: {foo}"; System.IFormattable s = \$"Foo: {foo}"; 你也可以生成一个 IFormattable 变量，允许你在不变上下文中转换字符串：var s = \$"Bar: {bar}"; System.FormattableString s = \$"Bar: {bar}";

当前文化和不变文化方法

如果启用了代码分析，所有插值字符串都会产生警告 CA1305（指定 IFormatProvider）。可以使用静态方法应用当前文化。

```
public static class Culture
{
    public static string Current(FormattableString formattableString)
    {
        return formattableString?.ToString(CultureInfo.CurrentCulture);
    }

    public static string Invariant(FormattableString formattableString)
    {
        return formattableString?.ToString(CultureInfo.InvariantCulture);
    }
}
```

然后，要生成适合当前文化的正确字符串，只需使用表达式：Culture.Current(\$"**interpolated**
eof(string).Name} string.") Culture.Invariant(\$"**interpolated {typeof(string).Name}** string.") 注意：Current 和

FormattableString type

The type of a \$"..." string interpolation expression [is not always](#) a simple string. The compiler decides which type to assign depending on the context:`string s = $"hello, {name}"; System.FormattableString s = $"Hello, {name}"; System.IFormattable s = $"Hello, {name}";`

This is also the order of type preference when the compiler needs to choose which overloaded method is going to be called.

A [new type](#), System.FormattableString, represents a composite format string, along with the arguments to be formatted. Use this to write applications that handle the interpolation arguments specifically:

```
public void AddLogItem(FormattableString formattableString)
{
    foreach (var arg in formattableString.GetArguments())
    {
        // do something to interpolation argument 'arg'
    }

    // use the standard interpolation and the current culture info
    // to get an ordinary String:
    var formatted = formattableString.ToString();

    // ...
}
```

Call the above method with: `AddLogItem($"The foo is {bar}");` and the bar is `{bar}.` For example, one could choose not to incur the performance cost of formatting the string if the logging level was already going to filter out the log item.

Implicit conversions

There are implicit type conversions from an interpolated string: `var s = $"Foo: {foo}"; System.IFormattable s = $"Foo: {foo}";` You can also produce an IFormattable variable that allows you to convert the string with invariant context: `var s = $"Bar: {bar}"; System.FormattableString s = $"Bar: {bar}";`

Current and Invariant Culture Methods

If code analysis is turned on, interpolated strings will all produce warning [CA1305](#) (Specify IFormatProvider). A static method may be used to apply current culture.

```
public static class Culture
{
    public static string Current(FormattableString formattableString)
    {
        return formattableString?.ToString(CultureInfo.CurrentCulture);
    }

    public static string Invariant(FormattableString formattableString)
    {
        return formattableString?.ToString(CultureInfo.InvariantCulture);
    }
}
```

Then, to produce a correct string for the current culture, just use the expression: `Culture.Current($"interpolated
eof(string).Name} string.") Culture.Invariant($"interpolated {typeof(string).Name} string.")` **Note:** Current and

Invariant 不能作为扩展方法创建，因为默认情况下，编译器将类型 `String` 分配给插值字符串表达式，这导致以下代码无法编译：

```
$"interpolated {typeof(string).Name} string.".Current();
```

FormattableString 类已经包含 Invariant() 方法，因此切换到不变文化的最简单方法是依赖 `using static : using static System.FormattableString`; string invariant = Invariant(\$"Now = {DateTime.Now}"); string current = \$"Now = {DateTime.Now}";

幕后原理

插值字符串只是 `String.Format()` 的语法糖。编译器（Roslyn）会在幕后将其转换为 `String.Format`：

```
var text = $"Hello {name + lastName};
```

上述内容将被转换成如下形式：

```
string text = string.Format("Hello {0}", new object[] {
    name + lastName
});
```

字符串插值和Linq

可以在Linq语句中使用插值字符串，以进一步提高可读性。

```
var fooBar = (from DataRow x in fooBarTable.Rows
    select string.Format("{0}{1}", x["foo"], x["bar"])).ToList();
```

可以重写为：

```
var fooBar = (from DataRow x in fooBarTable.Rows
    select $"{x["foo"]}{x["bar"]}").ToList();
```

可重用的插值字符串

使用`string.Format`，可以创建可重用的格式字符串：

```
public const string ErrorFormat = "捕获异常：\r{0}";
// ...
Logger.Log(string.Format(ErrorFormat, ex));
```

然而，插值字符串如果引用了不存在的变量，将无法编译。以下代码将无法编译：

```
public const string ErrorFormat = $"捕获异常：\r{error}";
// CS0103: 当前上下文中不存在名称 'error'
```

相反，应创建一个 `Func<>`，接收变量并返回一个 `String`：

```
public static Func<Exception, string> FormatError = error
=> $"捕获异常：\r{error}";
// ...
```

Invariant cannot be created as extension methods because, by default, the compiler assigns type `String` to *interpolated string expression* which causes the following code to fail to compile:

```
$"interpolated {typeof(string).Name} string.".Current();
```

FormattableString class already contains Invariant() method, so the simplest way of switching to invariant culture is by relying on `using static : using static System.FormattableString`; string invariant = Invariant(\$"Now = {DateTime.Now}"); string current = \$"Now = {DateTime.Now}";

Behind the scenes

Interpolated strings are just a syntactic sugar for `String.Format()`. The compiler (Roslyn) will turn it into a `String.Format` behind the scenes:

```
var text = $"Hello {name + lastName};
```

The above will be converted to something like this:

```
string text = string.Format("Hello {0}", new object[] {
    name + lastName
});
```

String Interpolation and Linq

It's possible to use interpolated strings in Linq statements to increase readability further.

```
var fooBar = (from DataRow x in fooBarTable.Rows
    select string.Format("{0}{1}", x["foo"], x["bar"])).ToList();
```

Can be re-written as:

```
var fooBar = (from DataRow x in fooBarTable.Rows
    select $"{x["foo"]}{x["bar"]}").ToList();
```

Reusable Interpolated Strings

With `string.Format`, you can create reusable format strings:

```
public const string ErrorFormat = "Exception caught:\r\n{0}";
// ...
Logger.Log(string.Format(ErrorFormat, ex));
```

Interpolated strings, however, will not compile with placeholders referring to non-existent variables. The following will not compile:

```
public const string ErrorFormat = $"Exception caught:\r\n{error}";
// CS0103: The name 'error' does not exist in the current context
```

Instead, create a `Func<>` which consumes variables and returns a `String`:

```
public static Func<Exception, string> FormatError =
    error => $"Exception caught:\r\n{error}";
// ...
```

```
Logger.Log(FormatError(ex));
```

字符串插值与本地化

如果您正在本地化您的应用程序，您可能会想是否可以将字符串插值与本地化结合使用。确实，能够在资源文件中存储类似于“我的名字是 e”{中间名} {姓氏}的字符串会非常好，而不是使用可读性差得多的：

“我的名字是 {0} {1} {2}”

字符串插值过程发生在编译时，不同于使用`string.Format`格式化字符串，该过程发生在运行时。插值字符串中的表达式必须引用当前上下文中的名称，并且需要存储在资源文件中。这意味着如果你想使用本地化，必须这样做：

```
var FirstName = "John";

// 使用不同资源文件"strings"的方法
// 用于法语 ("strings.fr.resx")、德语 ("strings.de.resx")、
// 和英语 ("strings.en.resx")
void ShowMyNameLocalized(string name, string middlename = "", string surname = "")
{
    // 获取本地化字符串
    var localizedMyNameIs = Properties.strings.Hello;
    // 在必要处插入空格
    name = (string.IsNullOrWhiteSpace(name) ? "" : name + " ");
    middlename = (string.IsNullOrWhiteSpace(middlename) ? "" : middlename + " ");
    surname = (string.IsNullOrWhiteSpace(surname) ? "" : surname + " ");
    // 显示它
    Console.WriteLine($"{localizedMyNameIs} {name}{middlename}{surname}".Trim());
}

// 切换到法语并向约翰问候
Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo("fr-FR");
ShowMyNameLocalized(FirstName);

// 切换到德语并向约翰问候
Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo("de-DE");
ShowMyNameLocalized(FirstName);

// 切换到美国英语并向约翰问候
Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo("en-US");
ShowMyNameLocalized(FirstName);
```

如果上述语言的资源字符串正确存储在各自的资源文件中，您应该得到以下输出：

```
Bonjour, mon nom est John
Hallo, mein Name ist John
Hello, my name is John
```

注意这意味着名字在每种语言中都跟随本地化字符串。如果不是这种情况，您需要在资源字符串中添加占位符并修改上述函数，或者需要在函数中查询文化信息并提供包含不同情况的switch语句。有关资源文件的更多详细信息，请参见如何在C#中使用本地化。

```
Logger.Log(FormatError(ex));
```

String interpolation and localization

If you're localizing your application you may wonder if it is possible to use string interpolation along with localization. Indeed, it would be nice to have the possibility to store in resource files `Strings` like: “My name is e”{middlename} {surname} instead of the much less readable:

“My name is {0} {1} {2}”

`String` interpolation process occurs at *compile time*, unlike formatting string with `string.Format` which occurs at *runtime*. Expressions in an interpolated string must reference names in the current context and need to be stored in resource files. That means that if you want to use localization you have to do it like:

```
var FirstName = "John";

// method using different resource file "strings"
// for French ("strings.fr.resx"), German ("strings.de.resx"),
// and English ("strings.en.resx")
void ShowMyNameLocalized(string name, string middlename = "", string surname = "")
{
    // get localized string
    var localizedMyNameIs = Properties.strings.Hello;
    // insert spaces where necessary
    name = (string.IsNullOrWhiteSpace(name) ? "" : name + " ");
    middlename = (string.IsNullOrWhiteSpace(middlename) ? "" : middlename + " ");
    surname = (string.IsNullOrWhiteSpace(surname) ? "" : surname + " ");
    // display it
    Console.WriteLine($"{localizedMyNameIs} {name}{middlename}{surname}".Trim());
}

// switch to French and greet John
Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo("fr-FR");
ShowMyNameLocalized(FirstName);

// switch to German and greet John
Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo("de-DE");
ShowMyNameLocalized(FirstName);

// switch to US English and greet John
Thread.CurrentThread.CurrentCulture = CultureInfo.GetCultureInfo("en-US");
ShowMyNameLocalized(FirstName);
```

If the resource strings for the languages used above are correctly stored in the individual resource files, you should get the following output:

```
Bonjour, mon nom est John
Hallo, mein Name ist John
Hello, my name is John
```

Note that this implies that the name follows the localized string in every language. If that is not the case, you need to add placeholders to the resource strings and modify the function above or you need to query the culture info in the function and provide a switch case statement containing the different cases. For more details about resource files, see [How to use localization in C#](#).

在没有可用翻译的情况下，使用大多数人都能理解的默认回退语言是一种良好做法。我建议使用英语作为默认回退语言。

递归插值

虽然不是很实用，但允许在另一个大括号内递归使用插值字符串：

```
Console.WriteLine($"字符串长度为 {"$"我的类名是{nameof(MyClass)}".Length} 个字符:");
Console.WriteLine($"我的类名是 {nameof(MyClass)}");
```

输出：

字符串长度为 27 个字符：

我的类名是 MyClass。

第72.3节：自动属性初始化器

介绍

属性可以在闭合的}后使用=运算符进行初始化。下面的Coordinate类展示了初始化属性的可用选项：

```
版本 ≥ 6.0
public class 坐标
{
    public int X { get; set; } = 34; // 带初始化器的自动属性的获取或设置
    public int Y { get; } = 89;      // 带初始化器的只读自动属性
}
```

访问器具有不同的可见性

您可以初始化访问器具有不同可见性的自动属性。这里有一个带有受保护设置器的示例：

```
public string 名称 { get; protected set; } = "Cheeze";
```

访问器也可以是internal、internal protected或private。

只读属性

除了可见性的灵活性之外，您还可以初始化只读自动属性。这里有一个示例：

```
public List<string> 配料 { get; } =
    new List<string> { "面团", "酱料", "奶酪" };
```

此示例还展示了如何初始化具有复杂类型的属性。此外，自动属性不能是只写的，因此也排除了只写初始化。

It is a good practice to use a default fallback language most people will understand, in case a translation is not available. I suggest to use English as default fallback language.

Recursive interpolation

Although not very useful, it is allowed to use an interpolated `string` recursively inside another's curly brackets:

```
Console.WriteLine($"String has ${"My class is called {nameof(MyClass)}".Length} chars");
Console.WriteLine($"My class is called {nameof(MyClass)}");
```

Output:

String has 27 chars:

My class is called MyClass.

Section 72.3: Auto-property initializers

Introduction

Properties can be initialized with the = operator after the closing }. The Coordinate class below shows the available options for initializing a property:

```
Version ≥ 6.0
public class Coordinate
{
    public int X { get; set; } = 34; // get or set auto-property with initializer
    public int Y { get; } = 89;      // read-only auto-property with initializer
}
```

Accessors With Different Visibility

You can initialize auto-properties that have different visibility on their accessors. Here's an example with a protected setter:

```
public string Name { get; protected set; } = "Cheeze";
```

The accessor can also be `internal`, `internal protected`, or `private`.

Read-Only Properties

In addition to flexibility with visibility, you can also initialize read-only auto-properties. Here's an example:

```
public List<string> Ingredients { get; } =
    new List<string> { "dough", "sauce", "cheese" };
```

This example also shows how to initialize a property with a complex type. Also, auto-properties can't be write-only, so that also precludes write-only initialization.

旧风格 (C# 6.0 之前)

在 C# 6 之前，这需要更多冗长的代码。我们使用了一个额外的变量，称为属性的后备字段，用来为属性赋默认值或初始化公共属性，如下所示，

```
版本 < 6.0
public class 坐标
{
    private int _x = 34;
    public int X { get { return _x; } set { _x = value; } }

    private readonly int _y = 89;
    public int Y { get { return _y; } }

    private readonly int _z;
    public int Z { get { return _z; } }

    public Coordinate()
    {
        _z = 42;
    }
}
```

注意：在 C# 6.0 之前，你仍然可以在构造函数中初始化可读写的自动实现属性（带有 *getter* 和 *setter* 的属性），但不能在声明时内联初始化属性

[查看演示](#)

用法

初始化器必须计算为静态表达式，就像字段初始化器一样。如果你需要引用非静态成员，可以像以前一样在构造函数中初始化属性，或者使用表达式体属性。非静态表达式，如下面（已注释）的示例，将会产生编译错误：

```
// public decimal X { get; set; } = InitMe(); // 生成编译器错误
decimal InitMe() { return 4m; }
```

但是静态方法可以用于初始化自动属性：

```
public class Rectangle
{
    public double Length { get; set; } = 1;
    public double Width { get; set; } = 1;
    public double Area { get; set; } = CalculateArea(1, 1);

    public static double CalculateArea(double length, double width)
    {
        return length * width;
    }
}
```

此方法也可应用于具有不同访问级别的属性：

```
public short Type { get; private set; } = 15;
```

自动属性初始化器允许在声明中直接赋值属性。对于只读属性

Old style (pre C# 6.0)

Before C# 6, this required much more verbose code. We were using one extra variable called backing property for the property to give default value or to initialize the public property like below,

```
Version < 6.0
public class Coordinate
{
    private int _x = 34;
    public int X { get { return _x; } set { _x = value; } }

    private readonly int _y = 89;
    public int Y { get { return _y; } }

    private readonly int _z;
    public int Z { get { return _z; } }

    public Coordinate()
    {
        _z = 42;
    }
}
```

Note: Before C# 6.0, you could still initialize read and write **auto implemented properties** (properties with a getter and a setter) from within the constructor, but you could not initialize the property inline with its declaration

[View Demo](#)

Usage

Initializers must evaluate to static expressions, just like field initializers. If you need to reference non-static members, you can either initialize properties in constructors like before, or use expression-bodied properties. Non-static expressions, like the one below (commented out), will generate a compiler error:

```
// public decimal X { get; set; } = InitMe(); // generates compiler error
decimal InitMe() { return 4m; }
```

But static methods **can** be used to initialize auto-properties:

```
public class Rectangle
{
    public double Length { get; set; } = 1;
    public double Width { get; set; } = 1;
    public double Area { get; set; } = CalculateArea(1, 1);

    public static double CalculateArea(double length, double width)
    {
        return length * width;
    }
}
```

This method can also be applied to properties with different level of accessors:

```
public short Type { get; private set; } = 15;
```

The auto-property initializer allows assignment of properties directly within their declaration. For read-only

属性，它满足确保属性不可变所需的所有要求。例如，考虑以下示例中的FingerPrint类：

```
public class FingerPrint
{
    public DateTime 时间戳 { get; } = DateTime.UtcNow;

    public string 用户 { get; } =
        System.Security.Principal.WindowsPrincipal.Current.Identity.Name;

    public string 进程 { get; } =
        System.Diagnostics.Process.GetCurrentProcess().ProcessName;
}
```

[查看演示](#)

注意事项

注意不要将自动属性或字段初始化器与使用`=>`而非`=`的类似表达式体方法混淆，以及不包含`{ get; }`的字段。

例如，以下每个声明都是不同的。

```
public class UserGroupDto
{
    // 只读自动属性带初始化器：
    public ICollection<UserDto> Users1 { get; } = new HashSet<UserDto>();

    // 带有初始化器的读写字段：
    public ICollection<UserDto> Users2 = new HashSet<UserDto>();

    // 带表达式主体的只读自动属性：
    public ICollection<UserDto> Users3 => new HashSet<UserDto>();
}
```

属性声明中缺少`{ get; }`会导致成为公共字段。只读自动属性`Users1`和读写字段`Users2`都只初始化一次，但公共字段允许从类外部更改集合实例，这通常是不希望的。将带表达式主体的只读自动属性改为带初始化器的只读属性，不仅需要将`=>`改为`=`，还需要添加`{ get; }`。

在`Users3`中使用不同的符号`(=> 而非 =)`导致每次访问该属性时都会返回一个新的`HashSet<UserDto>`实例，虽然从编译器角度看这是有效的 C#，但在用作集合成员时这通常不是期望的行为。

上述代码等价于：

```
public class UserGroupDto
{
    // 这是一个返回同一实例的属性
    // 该实例是在 UserGroupDto 实例化时创建的。
    private ICollection<UserDto> _users1 = new HashSet<UserDto>();
    public ICollection<UserDto> Users1 { get { return _users1; } }

    // 这是一个返回同一实例的字段
    // 该实例是在 UserGroupDto 实例化时创建的。
    public virtual ICollection<UserDto> Users2 = new HashSet<UserDto>();
}
```

properties, it takes care of all the requirements required to ensure the property is immutable. Consider, for example, the FingerPrint class in the following example:

```
public class FingerPrint
{
    public DateTime TimeStamp { get; } = DateTime.UtcNow;

    public string User { get; } =
        System.Security.Principal.WindowsPrincipal.Current.Identity.Name;

    public string Process { get; } =
        System.Diagnostics.Process.GetCurrentProcess().ProcessName;
}
```

[View Demo](#)

Cautionary notes

Take care to not confuse auto-property or field initializers with similar-looking expression-body methods which make use of`=>` as opposed to`=`, and fields which do not include`{ get; }`.

For example, each of the following declarations are different.

```
public class UserGroupDto
{
    // Read-only auto-property with initializer:
    public ICollection<UserDto> Users1 { get; } = new HashSet<UserDto>();

    // Read-write field with initializer:
    public ICollection<UserDto> Users2 = new HashSet<UserDto>();

    // Read-only auto-property with expression body:
    public ICollection<UserDto> Users3 => new HashSet<UserDto>();
}
```

Missing`{ get; }`在属性声明中会导致成为公共字段。Both read-only auto-property`Users1`和read-write field`Users2`都只初始化一次，但公共字段允许从类外部更改集合实例，这通常是不希望的。Changing a read-only auto-property with expression body to read-only property with initializer requires not only removing`>` from`=>`, but adding`{ get; }`。

The different symbol`(=> instead of =)` in`Users3`导致每次访问该属性时都会返回一个新的`HashSet<UserDto>`实例，虽然从编译器角度看这是有效的 C#，但在用作集合成员时这通常不是期望的行为。

The above code is equivalent to:

```
public class UserGroupDto
{
    // This is a property returning the same instance
    // which was created when the UserGroupDto was instantiated.
    private ICollection<UserDto> _users1 = new HashSet<UserDto>();
    public ICollection<UserDto> Users1 { get { return _users1; } }

    // This is a field returning the same instance
    // which was created when the UserGroupDto was instantiated.
    public virtual ICollection<UserDto> Users2 = new HashSet<UserDto>();
}
```

```
// 这是一个属性，每次调用时都会返回一个新的 HashSet<UserDto>，作为 ICollection<UserDto>
public ICollection<UserDto> Users3 { get { return new HashSet<UserDto>(); } }
```

第72.4节：空传播

? . 操作符和 ?[...] 操作符被称为 空条件操作符。它有时也被称为 安全导航操作符 等其他名称。

这很有用，因为如果对一个求值为 null 的表达式使用 . (成员访问) 操作符，程序将抛出 NullReferenceException。如果开发者改用 ?. (空条件) 操作符，表达式将返回 null，而不是抛出异常。

注意，如果使用 ?. 操作符且表达式非空，则 ?. 和 . 是等价的。

基础知识

```
var teacherName = classroom.GetTeacher().Name;
// 如果 GetTeacher() 返回 null，则抛出 NullReferenceException
```

查看演示

如果 classroom 没有老师，GetTeacher() 可能返回 null。当它为 null 并访问 Name 属性时，将抛出 NullReferenceException。

如果我们将此语句修改为使用?.语法，整个表达式的结果将是null：

```
var teacherName = classroom.GetTeacher()?.Name;
// 如果GetTeacher()返回null，则teacherName为null
```

查看演示

随后，如果classroom也可能是null，我们也可以这样写这条语句：

```
var teacherName = classroom?.GetTeacher()?.Name;
// 如果GetTeacher()返回null或classroom为null，则teacherName为null
```

查看演示

这是短路求值的一个例子：当使用空条件访问操作符的任何条件访问操作计算结果为null时，整个表达式会立即计算为 null，而不会处理链中的其余部分。

当包含空条件访问操作符的表达式的终端成员是值类型时，表达式会计算为该类型的Nullable<T>，因此不能直接用作不带?.的表达式的替代。

```
bool hasCertification = classroom.GetTeacher().HasCertification;
// 编译时无错误，但运行时可能抛出NullReferenceException异常

bool hasCertification = classroom?.GetTeacher()?.HasCertification;
// 编译时错误：不允许从 bool? 隐式转换为 bool

bool? hasCertification = classroom?.GetTeacher()?.HasCertification;
// 正常工作，如果链中的任何部分为 null，hasCertification 将为 null
```

```
// This is a property which returns a new HashSet<UserDto> as
// an ICollection<UserDto> on each call to it.
public ICollection<UserDto> Users3 { get { return new HashSet<UserDto>(); } }
```

Section 72.4: Null propagation

The ?. operator and ?[...] operator are called the [null-conditional operator](#). It is also sometimes referred to by other names such as the [safe navigation operator](#).

This is useful, because if the . (member accessor) operator is applied to an expression that evaluates to `null`, the program will throw a `NullReferenceException`. If the developer instead uses the ?. (null-conditional) operator, the expression will evaluate to `null` instead of throwing an exception.

Note that if the ?. operator is used and the expression is non-null, ?. and . are equivalent.

Basics

```
var teacherName = classroom.GetTeacher()?.Name;
// throws NullReferenceException if GetTeacher() returns null
```

View Demo

If the `classroom` does not have a teacher, `GetTeacher()` may return `null`. When it is `null` and the `Name` property is accessed, a `NullReferenceException` will be thrown.

If we modify this statement to use the ?. syntax, the result of the entire expression will be `null`:

```
var teacherName = classroom.GetTeacher()?.Name;
// teacherName is null if GetTeacher() returns null
```

View Demo

Subsequently, if `classroom` could also be `null`, we could also write this statement as:

```
var teacherName = classroom?.GetTeacher()?.Name;
// teacherName is null if GetTeacher() returns null OR classroom is null
```

View Demo

This is an example of short-circuiting: When any conditional access operation using the null-conditional operator evaluates to null, the entire expression evaluates to null immediately, without processing the rest of the chain.

When the terminal member of an expression containing the null-conditional operator is of a value type, the expression evaluates to a `Nullable<T>` of that type and so cannot be used as a direct replacement for the expression without ?..

```
bool hasCertification = classroom.GetTeacher().HasCertification;
// compiles without error but may throw a NullReferenceException at runtime
```

```
bool hasCertification = classroom?.GetTeacher()?.HasCertification;
// compile time error: implicit conversion from bool? to bool not allowed
```

```
bool? hasCertification = classroom?.GetTeacher()?.HasCertification;
// works just fine, hasCertification will be null if any part of the chain is null
```

```
bool hasCertification = classroom?.GetTeacher()?.HasCertification.GetValueOrDefault();
// 必须从可空类型中提取值以赋值给值类型变量
```

使用空合并运算符 (??)

你可以将空条件运算符与空合并运算符(??)结合使用，以在表达式结果为null时返回默认值。使用上面的示例：

```
var teacherName = classroom?.GetTeacher()?.Name ?? "无名氏";
// 当 GetTeacher()
// 返回 null 或 classroom 为 null 或 Name 为 null 时, teacherName 将为 "无名氏"
```

与索引器一起使用

空条件运算符可以与索引器一起使用：

```
var firstStudentName = classroom?.Students?[0]?.Name;
```

在上述示例中：

- 第一个?.确保classroom不是null。
- 第二个?确保整个Students集合不是null。
- 索引器后的第三个?.确保[0]索引器没有返回null对象。需要注意的是此操作仍然可能抛出IndexOutOfRangeException异常。

与void函数一起使用

空条件运算符也可以用于void函数。但在这种情况下，语句不会求值为null。它只是防止NullReferenceException异常。

```
List<string> list = null;
list?.Add("hi"); // 不会计算为null
```

事件调用的使用

假设以下事件定义：

```
private event EventArgs OnCompleted;
```

在调用事件时，传统的最佳实践是检查事件是否为null，以防没有订阅者存在：

```
var handler = OnCompleted;
if (handler != null)
{
    handler(EventArgs.Empty);
}
```

自从引入了空条件运算符后，调用可以简化为一行代码：

```
OnCompleted?.Invoke(EventArgs.Empty);
```

限制

```
bool hasCertification = classroom?.GetTeacher()?.HasCertification.GetValueOrDefault();
// must extract value from nullable to assign to a value type variable
```

Use with the Null-Coalescing Operator (??)

You can combine the null-conditional operator with the [Null-coalescing Operator](#) (??) to return a default value if the expression resolves to null. Using our example above:

```
var teacherName = classroom?.GetTeacher()?.Name ?? "No Name";
// teacherName will be "No Name" when GetTeacher()
// returns null OR classroom is null OR Name is null
```

Use with Indexers

The null-conditional operator can be used with [indexers](#):

```
var firstStudentName = classroom?.Students?[0]?.Name;
```

In the above example:

- The first ?. ensures that classroom is not null.
- The second ? ensures that the entire Students collection is not null.
- The third ?. after the indexer ensures that the [0] indexer did not return a null object. It should be noted that this operation can still throw an IndexOutOfRangeException.

Use with void Functions

Null-conditional operator can also be used with [void](#) functions. However in this case, the statement will not evaluate to null. It will just prevent a NullReferenceException.

```
List<string> list = null;
list?.Add("hi"); // Does not evaluate to null
```

Use with Event Invocation

Assuming the following event definition:

```
private event EventArgs OnCompleted;
```

When invoking an event, traditionally, it is best practice to check if the event is null in case no subscribers are present:

```
var handler = OnCompleted;
if (handler != null)
{
    handler(EventArgs.Empty);
}
```

Since the null-conditional operator has been introduced, the invocation can be reduced to a single line:

```
OnCompleted?.Invoke(EventArgs.Empty);
```

Limitations

空条件运算符产生的是右值，而非左值，也就是说，它不能用于属性赋值、事件订阅等。例如，以下代码将无法工作：

```
// 错误：赋值语句的左侧必须是变量、属性或索引器  
Process.GetProcessById(1337)?.EnableRaisingEvents = true;  
// 错误：事件只能出现在 += 或 -= 的左侧  
Process.GetProcessById(1337)?Exited += OnProcessExited;
```

注意事项

注意：

```
int? nameLength = person?.Name.Length; // 如果 'person' 为 null，则安全
```

不同于：

```
int? nameLength = (person?.Name).Length; // 避免这样写
```

因为前者对应于：

```
int? nameLength = person != null ? (int?)person.Name.Length : null;
```

后者对应于：

```
int? nameLength = (person != null ? person.Name : null).Length;
```

尽管这里使用了三元运算符?:来解释两种情况的区别，但这些运算符并不等价。以下示例可以轻松证明这一点：

```
void Main()  
{  
    var foo = new Foo();  
    Console.WriteLine("Null propagation");  
    Console.WriteLine(foo.Bar?.Length);  
  
    Console.WriteLine("Ternary");  
    Console.WriteLine(foo.Bar != null ? foo.Bar.Length : (int?)null);  
}  
  
class Foo  
{  
    public string Bar  
    {  
        get  
        {  
            Console.WriteLine("我被读取了");  
            return string.Empty;  
        }  
    }  
}
```

输出结果为：

```
空值传播  
我被读取了  
0
```

Null-conditional operator produces rvalue, not lvalue, that is, it cannot be used for property assignment, event subscription etc. For example, the following code will not work:

```
// Error: The left-hand side of an assignment must be a variable, property or indexer  
Process.GetProcessById(1337)?.EnableRaisingEvents = true;  
// Error: The event can only appear on the left hand side of += or -=  
Process.GetProcessById(1337)?.Exited += OnProcessExited;
```

Gotchas

Note that:

```
int? nameLength = person?.Name.Length; // safe if 'person' is null
```

is **not** the same as:

```
int? nameLength = (person?.Name).Length; // avoid this
```

because the former corresponds to:

```
int? nameLength = person != null ? (int?)person.Name.Length : null;
```

and the latter corresponds to:

```
int? nameLength = (person != null ? person.Name : null).Length;
```

Despite ternary operator ?: is used here for explaining the difference between two cases, these operators are not equivalent. This can be easily demonstrated with the following example:

```
void Main()  
{  
    var foo = new Foo();  
    Console.WriteLine("Null propagation");  
    Console.WriteLine(foo.Bar?.Length);  
  
    Console.WriteLine("Ternary");  
    Console.WriteLine(foo.Bar != null ? foo.Bar.Length : (int?)null);  
}  
  
class Foo  
{  
    public string Bar  
    {  
        get  
        {  
            Console.WriteLine("I was read");  
            return string.Empty;  
        }  
    }  
}
```

Which outputs:

```
Null propagation  
I was read  
0
```

三元运算符
我被读取了
我被读取了
0

[查看演示](#)

为了避免多次调用，等价写法为：

```
var interimResult = foo.Bar;
Console.WriteLine(interimResult != null ? interimResult.Length : (int?)null);
```

这个差异在某种程度上解释了为什么空值传播操作符尚未支持表达式树。

第72.5节：表达式体函数成员

表达式主体函数成员允许将 lambda 表达式用作成员体。对于简单的成员，它可以使代码更简洁、更易读。

表达式主体函数可用于属性、索引器、方法和运算符。

属性

```
public decimal TotalPrice => BasePrice + Taxes;
```

等同于：

```
public decimal TotalPrice
{
    get
    {
        return BasePrice + Taxes;
    }
}
```

当表达式主体函数用于属性时，该属性被实现为只读属性（仅有 getter）。

[查看演示](#)

索引器

```
public object this[string key] => dictionary[key];
```

等同于：

```
public object this[string key]
{
    get
    {
        return dictionary[key];
    }
}
```

Ternary
I was read
I was read
0

[View Demo](#)

To avoid multiple invocations equivalent would be:

```
var interimResult = foo.Bar;
Console.WriteLine(interimResult != null ? interimResult.Length : (int?)null);
```

And this difference somewhat explains why null propagation operator is [not yet supported](#) in expression trees.

Section 72.5: Expression-bodied function members

Expression-bodied function members allow the use of lambda expressions as member bodies. For simple members, it can result in cleaner and more readable code.

Expression-bodied functions can be used for properties, indexers, methods, and operators.

Properties

```
public decimal TotalPrice => BasePrice + Taxes;
```

Is equivalent to:

```
public decimal TotalPrice
{
    get
    {
        return BasePrice + Taxes;
    }
}
```

When an expression-bodied function is used with a property, the property is implemented as a getter-only property.

[View Demo](#)

Indexers

```
public object this[string key] => dictionary[key];
```

Is equivalent to:

```
public object this[string key]
{
    get
    {
        return dictionary[key];
    }
}
```

方法

```
static int Multiply(int a, int b) => a * b;
```

等同于：

```
static int Multiply(int a, int b)
{
    return a * b;
}
```

也可以用于void方法：

```
public void Dispose() => resource?.Dispose();
```

可以为Pair<T>类添加一个ToString的重写：

```
public override string ToString() => $"{First}, {Second}";
```

此外，这种简化的方法也适用于override关键字：

```
public class Foo
{
    public int Bar { get; }

    public string override ToString() => $"Bar: {Bar}";
}
```

运算符

这也可用于运算符：

```
public class Land
{
    public double Area { get; set; }

    public static Land operator +(Land first, Land second) =>
        new Land { Area = first.Area + second.Area };
}
```

限制

表达式主体函数成员有一些限制。它们不能包含块语句以及任何包含块的其他语句：if、switch、for、foreach、while、do、try等。

一些if语句可以用三元运算符替代。一些for和foreach语句可以转换为LINQ查询，例如：

```
IEnumerable<string> 数字
{
    get
    {
        for (int i = 0; i < 10; i++)
            yield return i.ToString();
    }
}
```

Methods

```
static int Multiply(int a, int b) => a * b;
```

Is equivalent to:

```
static int Multiply(int a, int b)
{
    return a * b;
}
```

Which can also be used with `void` methods:

```
public void Dispose() => resource?.Dispose();
```

An override of `ToString` could be added to the `Pair<T>` class:

```
public override string ToString() => $"{First}, {Second}";
```

Additionally, this simplistic approach works with the `override` keyword:

```
public class Foo
{
    public int Bar { get; }

    public string override ToString() => $"Bar: {Bar}";
}
```

Operators

This also can be used by operators:

```
public class Land
{
    public double Area { get; set; }

    public static Land operator +(Land first, Land second) =>
        new Land { Area = first.Area + second.Area };
}
```

Limitations

Expression-bodied function members have some limitations. They can't contain block statements and any other statements that contain blocks: if, switch, for, foreach, while, do, try, etc.

Some if statements can be replaced with ternary operators. Some for and foreach statements can be converted to LINQ queries, for example:

```
IEnumerable<string> Digits
{
    get
    {
        for (int i = 0; i < 10; i++)
            yield return i.ToString();
    }
}
```

```
IEnumerable<string> 数字 => Enumerable.Range(0, 10).Select(i => i.ToString());
```

在所有其他情况下，可以使用旧的函数成员语法。

表达式主体函数成员可以包含 `async/await`，但通常是多余的：

```
async Task<int> Foo() => await Bar();
```

可以替换为：

```
Task<int> Foo() => Bar();
```

第72.6节：nameof运算符

`nameof`操作符返回代码元素的名称，类型为`string`。这在抛出与方法参数相关的异常时非常有用，也适用于实现`INotifyPropertyChanged`接口时。

```
public string SayHello(string greeted)
{
    if (greeted == null)
        throw new ArgumentNullException(nameof(greeted));

    Console.WriteLine("Hello, " + greeted);
}
```

`nameof`操作符在编译时求值，并将表达式转换为字符串字面量。这对于以其成员名称命名的字符串也很有用。请考虑以下示例：

```
public static class Strings
{
    public const string Foo = nameof(Foo); // 而不是 Foo = "Foo"
    public const string Bar = nameof(Bar); // 而不是 Bar = "Bar"
}
```

由于 `nameof` 表达式是编译时常量，因此它们可以用于特性、`case` 标签、`switch` 语句等。

使用 `nameof` 与 `Enum` 一起非常方便。与其写成：

```
Console.WriteLine(Enum.One.ToString());
```

可以使用：

```
Console.WriteLine(nameof(Enum.One))
```

两种情况下的输出都是One。

`nameof`操作符可以使用类似静态的语法访问非静态成员。与其这样做：

```
string foo = "Foo";
string lengthName = nameof(foo.Length);
```

```
IEnumerable<string> Digits => Enumerable.Range(0, 10).Select(i => i.ToString());
```

In all other cases, the old syntax for function members can be used.

Expression-bodied function members can contain `async/await`，but it's often redundant:

```
async Task<int> Foo() => await Bar();
```

Can be replaced with:

```
Task<int> Foo() => Bar();
```

Section 72.6: Operator nameof

The `nameof` operator returns the name of a code element as a `string`. This is useful when throwing exceptions related to method arguments and also when implementing `INotifyPropertyChanged`.

```
public string SayHello(string greeted)
{
    if (greeted == null)
        throw new ArgumentNullException(nameof(greeted));

    Console.WriteLine("Hello, " + greeted);
}
```

The `nameof` operator is evaluated at compile time and changes the expression into a string literal. This is also useful for strings that are named after their member that exposes them. Consider the following:

```
public static class Strings
{
    public const string Foo = nameof(Foo); // Rather than Foo = "Foo"
    public const string Bar = nameof(Bar); // Rather than Bar = "Bar"
}
```

Since `nameof` expressions are compile-time constants, they can be used in attributes, `case` labels, `switch` statements, and so on.

It is convenient to use `nameof` with `Enums`. Instead of:

```
Console.WriteLine(Enum.One.ToString());
```

it is possible to use:

```
Console.WriteLine(nameof(Enum.One))
```

The output will be One in both cases.

The `nameof` operator can access non-static members using static-like syntax. Instead of doing:

```
string foo = "Foo";
string lengthName = nameof(foo.Length);
```

可以替换为：

```
string lengthName = nameof(string.Length);
```

两种示例的输出都是Length。然而，后者避免了不必要的实例创建。

虽然nameof操作符适用于大多数语言结构，但存在一些限制。例如，不能对开放泛型类型或方法返回值使用nameof操作符：

```
public static int Main()
{
    Console.WriteLine(nameof(List<>)); // 编译时错误
    Console.WriteLine(nameof(Main())); // 编译时错误
}
```

此外，如果将其应用于泛型类型，泛型类型参数将被忽略：

```
Console.WriteLine(nameof(List<int>)); // "List"
Console.WriteLine(nameof(List<bool>)); // "List"
```

更多示例，请参见专门介绍nameof的主题。

旧版本的解决方法（详细说明）

虽然在 C# 6.0 之前的版本中不存在nameof运算符，但可以通过使用 MemberExpression 实现类似功能，如下所示：

版本 < 6.0

表达式：

```
public static string NameOf<T>(Expression<Func<T>> propExp)
{
    var memberExpression = propExp.Body as MemberExpression;
    return memberExpression != null ? memberExpression.Member.Name : null;
}

public static string NameOf<TObj, T>(Expression<Func<TObj, T>> propExp)
{
    var memberExpression = propExp.Body as MemberExpression;
    return memberExpression != null ? memberExpression.Member.Name : null;
}
```

用法：

```
string 变量名 = NameOf(() => 变量);
string 属性名 = NameOf((Foo o) => o.Bar);
```

请注意，这种方法会在每次调用时创建表达式树，因此性能远不如nameof操作符，后者在编译时求值，运行时没有任何开销。

Can be replaced with:

```
string lengthName = nameof(string.Length);
```

The output will be Length in both examples. However, the latter prevents the creation of unnecessary instances.

Although the nameof operator works with most language constructs, there are some limitations. For example, you cannot use the nameof operator on open generic types or method return values:

```
public static int Main()
{
    Console.WriteLine(nameof(List<>)); // Compile-time error
    Console.WriteLine(nameof(Main())); // Compile-time error
}
```

Furthermore, if you apply it to a generic type, the generic type parameter will be ignored:

```
Console.WriteLine(nameof(List<int>)); // "List"
Console.WriteLine(nameof(List<bool>)); // "List"
```

For more examples, see this topic dedicated to [nameof](#).

Workaround for previous versions (more detail)

Although the nameof operator does not exist in C# for versions prior to 6.0, similar functionality can be had by using MemberExpression as in the following:

Version < 6.0

Expression:

```
public static string NameOf<T>(Expression<Func<T>> propExp)
{
    var memberExpression = propExp.Body as MemberExpression;
    return memberExpression != null ? memberExpression.Member.Name : null;
}

public static string NameOf<TObj, T>(Expression<Func<TObj, T>> propExp)
{
    var memberExpression = propExp.Body as MemberExpression;
    return memberExpression != null ? memberExpression.Member.Name : null;
}
```

Usage:

```
string 变量名 = NameOf(() => 变量);
string 属性名 = NameOf((Foo o) => o.Bar);
```

Note that this approach causes an expression tree to be created on every call, so the performance is much worse compared to nameof operator which is evaluated at compile time and has zero overhead at runtime.

第72.7节：使用静态类型

using static [命名空间.类型]指令允许导入类型的静态成员和枚举值。扩展方法作为扩展方法导入（仅来自一个类型），而不是导入到顶层作用域。

版本 ≥ 6.0

```
using static System.Console;
using static System.ConsoleColor;
using static System.Math;
```

```
class 程序
{
    static void Main()
    {
        BackgroundColor = DarkBlue;
        WriteLine(Sqrt(2));
    }
}
```

实时演示示例

版本 < 6.0

```
using System;

class Program
{
    static void Main()
    {
        Console.BackgroundColor = ConsoleColor.DarkBlue;
        Console.WriteLine(Math.Sqrt(2));
    }
}
```

第72.8节：索引初始化器

索引初始化器使得可以同时创建和初始化带索引的对象。

这使得初始化字典非常简单：

```
var dict = new Dictionary<string, int>()
{
    ["foo"] = 34,
    ["bar"] = 42
};
```

任何具有索引访问器（getter或setter）的对象都可以使用此语法：

```
class 程序
{
    public class MyClassWithIndexer
    {
        public int this[string index]
        {
            设置
            {
                Console.WriteLine($"索引: {index}, 值: {value}");
            }
        }
    }
}
```

Section 72.7: Using static type

The `using static [Namespace.Type]` directive allows the importing of static members of types and enumeration values. Extension methods are imported as extension methods (from just one type), not into top-level scope.

Version ≥ 6.0

```
using static System.Console;
using static System.ConsoleColor;
using static System.Math;
```

```
class Program
{
    static void Main()
    {
        BackgroundColor = DarkBlue;
        WriteLine(Sqrt(2));
    }
}
```

Live Demo Fiddle

Version < 6.0

```
using System;

class Program
{
    static void Main()
    {
        Console.BackgroundColor = ConsoleColor.DarkBlue;
        Console.WriteLine(Math.Sqrt(2));
    }
}
```

Section 72.8: Index initializers

Index initializers make it possible to create and initialize objects with indexes at the same time.

This makes initializing Dictionaries very easy:

```
var dict = new Dictionary<string, int>()
{
    ["foo"] = 34,
    ["bar"] = 42
};
```

Any object that has an indexed getter or setter can be used with this syntax:

```
class 程序
{
    public class MyClassWithIndexer
    {
        public int this[string index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
    }
}
```

```

public static void Main()
{
    var x = new 带索引器的类()
    {
        ["foo"] = 34,
        ["bar"] = 42
    };

    Console.ReadKey();
}

```

输出：

```

索引: foo, 值: 34
索引: bar, 值: 42

```

[查看演示](#)

如果类有多个索引器，可以在一组语句中为它们全部赋值：

```

class 程序
{
    public class MyClassWithIndexer
    {
        public int this[string index]
        {
            设置
            {
                Console.WriteLine($"索引: {index}, 值: {value}");
            }
        }

        public string this[int 索引]
        {
            设置
            {
                Console.WriteLine($"索引: {index}, 值: {value}");
            }
        }
    }

    public static void Main()
    {
        var x = new 带索引器的类()
        {
            ["foo"] = 34,
            ["bar"] = 42,
            [10] = "十",
            [42] = "生命的意义"
        };
    }
}

```

输出：

```

索引: foo, 值: 34
索引: bar, 值: 42

```

```

public static void Main()
{
    var x = new MyClassWithIndexer()
    {
        ["foo"] = 34,
        ["bar"] = 42
    };

    Console.ReadKey();
}

```

Output:

```

Index: foo, value: 34
Index: bar, value: 42

```

[View Demo](#)

If the class has multiple indexers it is possible to assign them all in a single group of statements:

```

class Program
{
    public class MyClassWithIndexer
    {
        public int this[string index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }

        public string this[int index]
        {
            set
            {
                Console.WriteLine($"Index: {index}, value: {value}");
            }
        }
    }

    public static void Main()
    {
        var x = new MyClassWithIndexer()
        {
            ["foo"] = 34,
            ["bar"] = 42,
            [10] = "Ten",
            [42] = "Meaning of life"
        };
    }
}

```

Output:

```

Index: foo, value: 34
Index: bar, value: 42

```

索引：10，值：十
索引：42，值：生命的意义

需要注意的是，索引器的set访问器行为可能与Add方法（用于集合初始化器）不同。

例如：

```
var d = new Dictionary<string, int>
{
    ["foo"] = 34,
    ["foo"] = 42,
}; // 不会抛出异常，第二个值会覆盖第一个值
```

相对比：

```
var d = new Dictionary<string, int>
{
    { "foo", 34 },
    { "foo", 42 },
}; // 运行时 ArgumentException : 已添加了具有相同键的项。
```

第72.9节：改进的重载解析

以下代码片段展示了在期望委托时传递方法组（而非lambda表达式）的示例。重载解析现在能够解析此情况，而不会因C# 6能够检查传入方法的返回类型而引发重载歧义错误。

```
using System;
public class Program
{
    public static void Main()
    {
        Overloaded(DoSomething);
    }

    static void Overloaded(Action action)
    {
        Console.WriteLine("调用了带Action的重载");
    }

    static void Overloaded(Func<int> function)
    {
        Console.WriteLine("调用了带Func<int>的重载");
    }

    static int DoSomething()
    {
        Console.WriteLine(0);
        return 0;
    }
}
```

结果：

版本 = 6.0

Index: 10, value: Ten
Index: 42, value: Meaning of life

It should be noted that the indexer set accessor might behave differently compared to an Add method (used in collection initializers).

For example:

```
var d = new Dictionary<string, int>
{
    [ "foo" ] = 34,
    [ "foo" ] = 42,
}; // does not throw, second value overwrites the first one
```

versus:

```
var d = new Dictionary<string, int>
{
    { "foo", 34 },
    { "foo", 42 },
}; // run-time ArgumentException: An item with the same key has already been added.
```

Section 72.9: Improved overload resolution

Following snippet shows an example of passing a method group (as opposed to a lambda) when a delegate is expected. Overload resolution will now resolve this instead of raising an ambiguous overload error due to the ability of C# 6 to check the return type of the method that was passed.

```
using System;
public class Program
{
    public static void Main()
    {
        Overloaded(DoSomething);
    }

    static void Overloaded(Action action)
    {
        Console.WriteLine("overload with action called");
    }

    static void Overloaded(Func<int> function)
    {
        Console.WriteLine("overload with Func<int> called");
    }

    static int DoSomething()
    {
        Console.WriteLine(0);
        return 0;
    }
}
```

Results:

Version = 6.0

输出

调用了带 Func<int> 的重载

查看演示

版本 = 5.0

错误

错误 CS0121：调用在以下方法或属性之间存在歧义：
'Program.Overloaded(System.Action)' 和 'Program.Overloaded(System.Func)'

C# 6 也能很好地处理以下针对 lambda 表达式的精确匹配情况，
这在 C# 5 中会导致错误。

```
using System;

class Program
{
    static void Foo(Func<Func<long>> func) {}
    static void Foo(Func<Func<int>> func) {}

    static void Main()
    {
        Foo(() => () => 7);
    }
}
```

第72.10节：catch和finally中的await

在 C# 6 中，可以在 catch 和 finally 块中使用 await 表达式。

由于编译器的限制，早期版本无法在 catch 和 finally 块中使用 await 表达式。C# 6 通过允许使用 await 表达式，使等待异步任务变得更加容易。

```
try
{
    //自C#5起
    await service.InitializeAsync();
}
catch (Exception e)
{
    //自C#6起
    await logger.LogAsync(e);
}
finally
{
    //自C#6起
    await service.CloseAsync();
}
```

在 C# 5 中，执行异步操作时需要使用 bool 类型或在 try catch 外声明一个 Exception。

Output

overload with Func<int> called

View Demo

Version = 5.0

Error

error CS0121: The call is ambiguous between the following methods or properties:
'Program.Overloaded(System.Action)' and 'Program.Overloaded(System.Func)'

C# 6 can also handle well the following case of exact matching for lambda expressions which would have resulted in an error in C# 5.

```
using System;

class Program
{
    static void Foo(Func<Func<long>> func) {}
    static void Foo(Func<Func<int>> func) {}

    static void Main()
    {
        Foo(() => () => 7);
    }
}
```

Section 72.10: Await in catch and finally

It is possible to use `await` expression to apply `await operator` to `Tasks` or `Task<TResult>` in the `catch` and `finally` blocks in C# 6.

It was not possible to use the `await` expression in the `catch` and `finally` blocks in earlier versions due to compiler limitations. C# 6 makes awaiting async tasks a lot easier by allowing the `await` expression.

```
try
{
    //since C#5
    await service.InitializeAsync();
}
catch (Exception e)
{
    //since C#6
    await logger.LogAsync(e);
}
finally
{
    //since C#6
    await service.CloseAsync();
}
```

It was required in C# 5 to use a `bool` or declare an `Exception` outside the try catch to perform async operations.

该方法示例如下：

```
bool error = false;
Exception ex = null;

try
{
    // 自 C#5 起
    await service.InitializeAsync();
}
catch (Exception e)
{
    // 声明 bool 或将异常放入变量中
    error = true;
    ex = e;
}

// 如果不使用异常
if (error)
{
    // 处理异步任务
}

// 如果想使用异常中的信息
if (ex != null)
{
    await logger.LogAsync(ex);
}

// 关闭服务，因为这在 finally 中无法实现
await service.CloseAsync();
```

第72.11节：小改动和错误修复

命名参数周围现在禁止使用括号。以下代码在 C#5 中可编译，但在 C#6 中不可编译

版本 ≤ 5.0
Console.WriteLine((value: 23));

操作数is和as不再允许是方法组。以下代码在C#5中可以编译，但在C#6中不行

版本 ≤ 5.0
var result = "".Any is byte;

本地编译器允许这样做（尽管它确实显示了警告），实际上它甚至没有检查扩展方法的兼容性，允许诸如1.Any is string或IDisposable.Dispose is object之类的疯狂操作。

有关更改的更新，请参见此参考文献。

第72.12节：使用扩展方法进行集合初始化

当实例化任何实现了IEnumerable且具有一个名为Add且接受单个参数的方法的类时，可以使用集合初始化语法。

This method is shown in the following example:

```
bool error = false;
Exception ex = null;

try
{
    // Since C#5
    await service.InitializeAsync();
}
catch (Exception e)
{
    // Declare bool or place exception inside variable
    error = true;
    ex = e;
}

// If you don't use the exception
if (error)
{
    // Handle async task
}

// If want to use information from the exception
if (ex != null)
{
    await logger.LogAsync(ex);
}

// Close the service, since this isn't possible in the finally
await service.CloseAsync();
```

Section 72.11: Minor changes and bugfixes

Parentheses are now forbidden around named parameters. The following compiles in C#5, but not C#6

Version ≤ 5.0
Console.WriteLine((value: 23));

Operands of is and as are no longer allowed to be method groups. The following compiles in C#5, but not C#6

Version ≤ 5.0
var result = "".Any is byte;

The native compiler allowed this (although it did show a warning), and in fact didn't even check extension method compatibility, allowing crazy things like 1.Any is string or IDisposable.Dispose is object.

See [this reference](#) for updates on changes.

Section 72.12: Using an extension method for collection initialization

Collection initialization syntax can be used when instantiating any class which implements IEnumerable and has a method named Add which takes a single parameter.

在以前的版本中，这个Add方法必须是被初始化类的实例方法。在C#6中，它也可以是扩展方法。

```
public class CollectionWithAdd : IEnumerable
{
    public void Add<T>(T item)
    {
        Console.WriteLine("使用实例添加方法添加的项: " + item);
    }

    public IEnumerator GetEnumerator()
    {
        // 一些实现代码
    }
}

public class CollectionWithoutAdd : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        // 一些实现代码
    }
}

public static class Extensions
{
    public static void Add<T>(this CollectionWithoutAdd collection, T item)
    {
        Console.WriteLine("通过扩展添加方法添加的项: " + item);
    }
}

public class Program
{
    public static void Main()
    {
        var collection1 = new CollectionWithAdd{1, 2, 3}; // 在所有 C# 版本中有效
        var collection2 = new CollectionWithoutAdd{4, 5, 6}; // 仅自 C# 6 起有效
    }
}
```

这将输出：

```
通过实例添加方法添加的项: 1
通过实例添加方法添加的项: 2
通过实例添加方法添加的项: 3
通过扩展添加方法添加的项: 4
通过扩展添加方法添加的项: 5
通过扩展添加方法添加的项: 6
```

第72.13节：禁用警告增强功能

在 C# 5.0 及更早版本中，开发者只能通过编号来抑制警告。随着 Roslyn 分析器的引入，C# 需要一种方法来禁用来自特定库的警告。在 C# 6.0 中，pragma 指令可以通过名称来抑制警告。

之前：

In previous versions, this `Add` method had to be an **instance** method on the class being initialized. In C#6, it can also be an extension method.

```
public class CollectionWithAdd : IEnumerable
{
    public void Add<T>(T item)
    {
        Console.WriteLine("Item added with instance add method: " + item);
    }

    public IEnumerator GetEnumerator()
    {
        // Some implementation here
    }
}

public class CollectionWithoutAdd : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        // Some implementation here
    }
}

public static class Extensions
{
    public static void Add<T>(this CollectionWithoutAdd collection, T item)
    {
        Console.WriteLine("Item added with extension add method: " + item);
    }
}

public class Program
{
    public static void Main()
    {
        var collection1 = new CollectionWithAdd{1, 2, 3}; // Valid in all C# versions
        var collection2 = new CollectionWithoutAdd{4, 5, 6}; // Valid only since C# 6
    }
}
```

This will output:

```
Item added with instance add method: 1
Item added with instance add method: 2
Item added with instance add method: 3
Item added with extension add method: 4
Item added with extension add method: 5
Item added with extension add method: 6
```

Section 72.13: Disable Warnings Enhancements

In C# 5.0 and earlier the developer could only suppress warnings by number. With the introduction of Roslyn Analyzers, C# needs a way to disable warnings issued from specific libraries. With C# 6.0 the pragma directive can suppress warnings by name.

Before:

```
#pragma warning disable 0501
```

C# 6.0 :

```
#pragma warning disable CS0501
```

```
#pragma warning disable 0501
```

C# 6.0:

```
#pragma warning disable CS0501
```

belindoc.com

第73章：C# 5.0 特性

带参数的方法/修饰符

详情
T 是返回类型

第73.1节：异步与等待

`async` 和 `await` 是两个操作符，旨在通过释放线程并等待操作完成后再继续，来提升性能。

下面是一个获取字符串并返回其长度的示例：

```
//该方法是异步的，因为：  
//1. 它有 async 和 Task 或 Task<T> 作为修饰符  
//2. 它以 "Async" 结尾  
async Task<int> GetStringLengthAsync(string URL){  
    HttpClient client = new HttpClient();  
    //发送GET请求并返回响应体作为字符串  
    Task<string> getString = client.GetStringAsync(URL);  
    //等待getString完成后再返回其长度  
    string contents = await getString;  
    return contents.Length;  
}  
  
private async void doProcess(){  
    int length = await GetStringLengthAsync("http://example.com/");  
    //等待上述所有操作完成后再打印数字  
    Console.WriteLine(length);  
}
```

这是另一个下载文件的示例，以及处理下载进度变化和下载完成时发生的情况（有两种方法）：

方法1：

```
//此方法使用异步事件处理器，但不是与await结合的异步  
private void DownloadAndUpdateAsync(string uri, string DownloadLocation){  
    WebClient web = new WebClient();  
    //分配事件处理器  
    web.DownloadProgressChanged += new DownloadProgressChangedEventArgs(ProgressChanged);  
    web.DownloadFileCompleted += new AsyncCompletedEventHandler(FileCompleted);  
    //异步下载文件  
    web.DownloadFileAsync(new Uri(uri), DownloadLocation);  
}  
  
//下载进度变化时调用的事件  
private void ProgressChanged(object sender, DownloadProgressChangedEventArgs e){  
    //示例代码  
    int i = 0;  
    i++;  
    doSomething();  
}  
  
//下载完成时调用的事件  
private void FileCompleted(object sender, AsyncCompletedEventArgs e){  
    Console.WriteLine("Completed!")  
}
```

Chapter 73: C# 5.0 Features

Method/Modifier with Parameter Details
Type<T> T is the return type

Section 73.1: Async & Await

`async` and `await` are two operators that are intended to improve performance by freeing up Threads and waiting for operations to complete before moving forward.

Here's an example of getting a string before returning its length:

```
//This method is async because:  
//1. It has async and Task or Task<T> as modifiers  
//2. It ends in "Async"  
async Task<int> GetStringLengthAsync(string URL){  
    HttpClient client = new HttpClient();  
    //Sends a GET request and returns the response body as a string  
    Task<string> getString = client.GetStringAsync(URL);  
    //Waits for getString to complete before returning its length  
    string contents = await getString;  
    return contents.Length;  
}  
  
private async void doProcess(){  
    int length = await GetStringLengthAsync("http://example.com/");  
    //Waits for all the above to finish before printing the number  
    Console.WriteLine(length);  
}
```

Here's another example of downloading a file and handling what happens when its progress has changed and when the download completes (there are two ways to do this):

Method 1:

```
//This one using async event handlers, but not async coupled with await  
private void DownloadAndUpdateAsync(string uri, string DownloadLocation){  
    WebClient web = new WebClient();  
    //Assign the event handler  
    web.DownloadProgressChanged += new DownloadProgressChangedEventArgs(ProgressChanged);  
    web.DownloadFileCompleted += new AsyncCompletedEventHandler(FileCompleted);  
    //Download the file asynchronously  
    web.DownloadFileAsync(new Uri(uri), DownloadLocation);  
}  
  
//event called for when download progress has changed  
private void ProgressChanged(object sender, DownloadProgressChangedEventArgs e){  
    //example code  
    int i = 0;  
    i++;  
    doSomething();  
}  
  
//event called for when download has finished  
private void FileCompleted(object sender, AsyncCompletedEventArgs e){  
    Console.WriteLine("Completed!")  
}
```

方法 2：

```
//不过，这个方法确实如此  
//参考第一个示例，了解为什么此方法是异步的  
private void DownloadAndUpdateAsync(string uri, string DownloadLocation){  
    WebClient web = new WebClient();  
    //分配事件处理程序  
    web.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);  
    //异步下载文件  
    web.DownloadFileAsync(new Uri(uri), DownloadLocation);  
    //注意这里没有完成事件，而是使用了第一个示例中的技术  
}  
private void ProgressChanged(object sender, DownloadProgressChangedEventArgs e){  
    int i = 0;  
    i++;  
    doSomething();  
}  
private void doProcess(){  
    //等待下载完成  
    await DownloadAndUpdateAsync(new Uri("http://example.com/file"))  
    doSomething();  
}
```

第73.2节：调用者信息属性

调用者信息属性 (C.I.A.s) 旨在作为一种简单方式，从调用目标方法的任何对象获取属性。实际上只有一种使用方式，且只有3个属性。

示例：

```
//这是“调用方法”：调用目标方法的方法  
public void doProcess()  
{  
    GetMessageCallerAttributes("显示我的属性。");  
}  
//这是目标方法  
//只有3个调用者属性  
public void GetMessageCallerAttributes(string message,  
    //获取调用此方法的名称  
    [System.Runtime.CompilerServices.CallerMemberName] string memberName = "",  
    //获取“调用方法”所在文件的路径  
    [System.Runtime.CompilerServices.CallerFilePath] string sourceFilePath = "",  
    //获取“调用方法”的行号  
    [System.Runtime.CompilerServices.CallerLineNumber] int sourceLineNumber = 0)  
{  
    //写出所有属性的行  
    System.Diagnostics.Trace.WriteLine("Message: " + message);  
    System.Diagnostics.Trace.WriteLine("Member: " + memberName);  
    System.Diagnostics.Trace.WriteLine("Source File Path: " + sourceFilePath);  
    System.Diagnostics.Trace.WriteLine("Line Number: " + sourceLineNumber);  
}
```

示例输出：

```
//Message: Show my attributes.  
//Member: doProcess  
//Source File Path: c:\Path\To\The\File  
//Line Number: 13
```

Method 2:

```
//however, this one does  
//Refer to first example on why this method is async  
private void DownloadAndUpdateAsync(string uri, string DownloadLocation){  
    WebClient web = new WebClient();  
    //Assign the event handler  
    web.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);  
    //Download the file async  
    web.DownloadFileAsync(new Uri(uri), DownloadLocation);  
    //Notice how there is no complete event, instead we're using techniques from the first example  
}  
private void ProgressChanged(object sender, DownloadProgressChangedEventArgs e){  
    int i = 0;  
    i++;  
    doSomething();  
}  
private void doProcess(){  
    //Wait for the download to finish  
    await DownloadAndUpdateAsync(new Uri("http://example.com/file"))  
    doSomething();  
}
```

Section 73.2: Caller Information Attributes

C.I.A.s are intended as a simple way of getting attributes from whatever is calling the targeted method. There is really only 1 way to use them and there are only 3 attributes.

Example:

```
//This is the "calling method": the method that is calling the target method  
public void doProcess()  
{  
    GetMessageCallerAttributes("Show my attributes.");  
}  
//This is the target method  
//There are only 3 caller attributes  
public void GetMessageCallerAttributes(string message,  
    //gets the name of what is calling this method  
    [System.Runtime.CompilerServices.CallerMemberName] string memberName = "",  
    //gets the path of the file in which the "calling method" is in  
    [System.Runtime.CompilerServices.CallerFilePath] string sourceFilePath = "",  
    //gets the line number of the "calling method"  
    [System.Runtime.CompilerServices.CallerLineNumber] int sourceLineNumber = 0)  
{  
    //Writes lines of all the attributes  
    System.Diagnostics.Trace.WriteLine("Message: " + message);  
    System.Diagnostics.Trace.WriteLine("Member: " + memberName);  
    System.Diagnostics.Trace.WriteLine("Source File Path: " + sourceFilePath);  
    System.Diagnostics.Trace.WriteLine("Line Number: " + sourceLineNumber);  
}
```

Example Output:

```
//Message: Show my attributes.  
//Member: doProcess  
//Source File Path: c:\Path\To\The\File  
//Line Number: 13
```

第74章：C# 4.0 特性

第74.1节：可选参数和命名参数

如果该参数是可选参数，我们可以在调用时省略该参数。每个可选参数都有其自己的默认值，如果我们不提供值，则会采用默认值。可选参数的默认值必须是

1. 常量表达式。
2. 必须是枚举或结构体等值类型。
3. 必须是形如 default(valueType) 的表达式。

它必须设置在参数列表的末尾

带有默认值的方法参数：

```
public void ExampleMethod(int required, string optValue = "test", int optNum = 42)  
{  
    //...  
}
```

正如MSDN所说，命名参数，

通过将参数名称与函数参数关联，使您能够按名称传递参数。无需记住参数的位置，因为我们并不总是清楚参数的位置。也无需查看被调用函数参数列表中的参数顺序。我们可以通过参数名称为每个实参指定对应的参数。

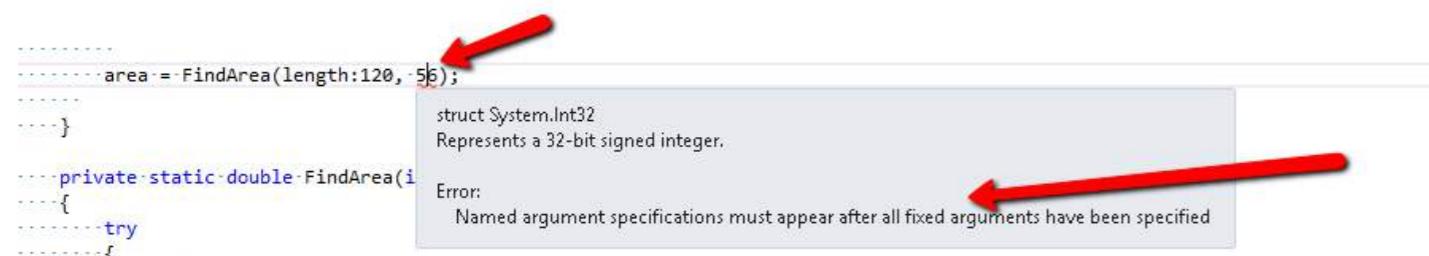
命名参数：

```
// required = 3, optValue = "test", optNum = 4  
ExampleMethod(3, optNum: 4);  
// required = 2, optValue = "foo", optNum = 42  
ExampleMethod(2, optValue: "foo");  
// required = 6, optValue = "bar", optNum = 1  
ExampleMethod(optNum: 1, optValue: "bar", required: 6);
```

使用命名参数的限制

命名参数的指定必须出现在所有固定参数指定之后。

如果在固定参数之前使用命名参数，将会出现如下编译时错误。



命名参数的指定必须出现在所有固定参数指定之后

Chapter 74: C# 4.0 Features

Section 74.1: Optional parameters and named arguments

We can omit the argument in the call if that argument is an Optional Argument Every Optional Argument has its own default value It will take default value if we do not supply the value A default value of a Optional Argument must be a

1. Constant expression.
2. Must be a value type such as enum or struct.
3. Must be an expression of the form default(valueType)

It must be set at the end of parameter list

Method parameters with default values:

```
public void ExampleMethod(int required, string optValue = "test", int optNum = 42)  
{  
    //...  
}
```

As said by MSDN, A named argument ,

Enables you to pass the argument to the function by associating the parameter's name No needs for remembering the parameters position that we are not aware of always. No need to look the order of the parameters in the parameters list of called function. We can specify parameter for each arguments by its name.

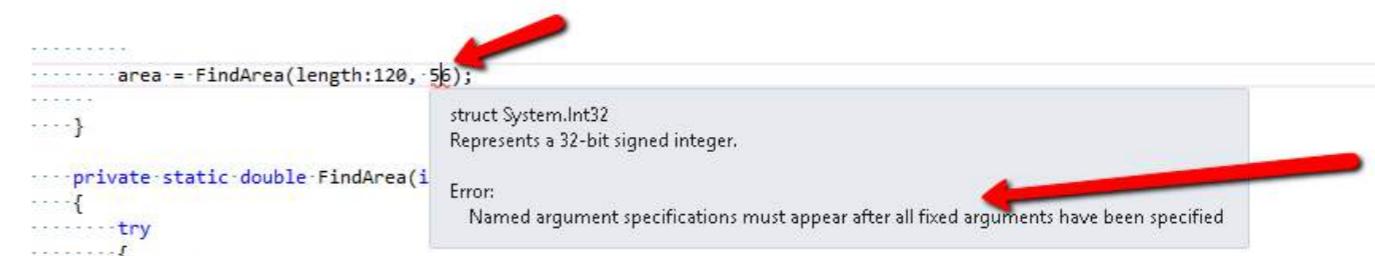
Named arguments:

```
// required = 3, optValue = "test", optNum = 4  
ExampleMethod(3, optNum: 4);  
// required = 2, optValue = "foo", optNum = 42  
ExampleMethod(2, optValue: "foo");  
// required = 6, optValue = "bar", optNum = 1  
ExampleMethod(optNum: 1, optValue: "bar", required: 6);
```

Limitation of using a Named Argument

Named argument specification must appear after all fixed arguments have been specified.

If you use a named argument before a fixed argument you will get a compile time error as follows.



Named argument specification must appear after all fixed arguments have been specified

第74.2节：协变与逆变

泛型接口和委托可以使用关键字out和in分别将其类型参数标记为协变（covariant）或逆变（contravariant）。这些声明会被隐式和显式的类型转换所遵守，且适用于编译时和运行时。

例如，现有接口IEnumerable<T>已被重新定义为协变接口：

```
interface IEnumerable<out T>
{
    IEnumerator<T> GetEnumerator();
}
```

现有接口IComparer已被重新定义为逆变接口：

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

第74.3节：动态成员查找

在C#类型系统中引入了一种新的伪类型dynamic。它被视为System.Object，但除此之外，对该类型的值进行的任何成员访问（方法调用、字段、属性或索引器访问，或委托调用）或运算符的应用都允许不进行任何类型检查，其解析被推迟到运行时。这被称为鸭子类型或延迟绑定。例如：

```
// 返回任何对象的Length属性或字段的值
int GetLength(dynamic obj)
{
    return obj.Length;

GetLength("Hello, world");           // 字符串有Length属性,
GetLength(new int[] { 1, 2, 3 });     // 数组也有,
GetLength(42);                      // 但整数没有——在GetLength方法中运行时会抛出异常
```

在这种情况下，使用dynamic类型可以避免更冗长的反射。它底层仍然使用反射，但通常由于缓存而更快。

该特性主要针对与动态语言的互操作性。

```
// 初始化引擎并执行文件
var runtime = ScriptRuntime.CreateFromConfiguration();
dynamic globals = runtime.Globals;
runtime.ExecuteFile("Calc.rb");

// 使用来自 Ruby 的 Calc 类型
dynamic calc = globals.Calc.@new();
calc.valueA = 1337;
calc.valueB = 666;
dynamic answer = calc.Calculate();
```

动态类型即使在主要是静态类型的代码中也有应用，例如它使[双重分派](#)成为可能而无需实现访问者模式（Visitor pattern）。

Section 74.2: Variance

Generic interfaces and delegates can have their type parameters marked as *covariant* or *contravariant* using the `out` and `in` keywords respectively. These declarations are then respected for type conversions, both implicit and explicit, and both compile time and run time.

For example, the existing interface `IEnumerable<T>` has been redefined as being covariant:

```
interface IEnumerable<out T>
{
    IEnumerator<T> GetEnumerator();
}
```

The existing interface `IComparer` has been redefined as being contravariant:

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

Section 74.3: Dynamic member lookup

A new pseudo-type `dynamic` is introduced into the C# type system. It is treated as `System.Object`, but in addition, any member access (method call, field, property, or indexer access, or a delegate invocation) or application of an operator on a value of such type is permitted without any type checking, and its resolution is postponed until run-time. This is known as duck typing or late binding. For example:

```
// Returns the value of Length property or field of any object
int GetLength(dynamic obj)
{
    return obj.Length;

GetLength("Hello, world");           // a string has a Length property,
GetLength(new int[] { 1, 2, 3 });     // and so does an array,
GetLength(42);                      // but not an integer - an exception will be thrown
                                    // in GetLength method at run-time
```

In this case, dynamic type is used to avoid more verbose Reflection. It still uses Reflection under the hood, but it's usually faster thanks to caching.

This feature is primarily targeted at interoperability with dynamic languages.

```
// Initialize the engine and execute a file
var runtime = ScriptRuntime.CreateFromConfiguration();
dynamic globals = runtime.Globals;
runtime.ExecuteFile("Calc.rb");

// Use Calc type from Ruby
dynamic calc = globals.Calc.@new();
calc.valueA = 1337;
calc.valueB = 666;
dynamic answer = calc.Calculate();
```

Dynamic type has applications even in mostly statically typed code, for example it makes [double dispatch](#) possible without implementing Visitor pattern.

第74.4节：使用COM时的可选ref关键字

调用COM接口提供的方法时，调用者的ref关键字现在是可选的。

给定一个COM方法，其签名为

```
void Increment(ref int x);
```

调用现在可以写成以下任意一种形式

```
Increment(0); // 不再需要“ref”或占位变量
```

Section 74.4: Optional ref keyword when using COM

The ref keyword for callers of methods is now optional when calling into methods supplied by COM interfaces.

Given a COM method with the signature

```
void Increment(ref int x);
```

the invocation can now be written as either

```
Increment(0); // no need for "ref" or a place holder variable any more
```

belindoc.com

第75章：C# 3.0 特性

第75.1节：隐式类型变量 (var)

关键字var允许程序员在编译时隐式地为变量指定类型。var声明的变量类型与显式声明的变量类型相同。

```
var squaredNumber = 10 * 10;
var squaredNumberDouble = 10.0 * 10.0;
var builder = new StringBuilder();
var anonymousObject = new
{
    One = SquaredNumber,
    Two = SquaredNumberDouble,
    Three = Builder
}
```

上述变量的类型分别是int、double、StringBuilder和一个匿名类型。

需要注意的是，var变量并非动态类型。SquaredNumber = Builder是不合法的，因为你试图将一个int类型赋值为StringBuider的实例

第75.2节：语言集成查询 (LINQ)

```
//示例 1
int[] 数组 = { 1, 5, 2, 10, 7 };

// 选择数组中所有奇数的平方，并按降序排序
IQueryable<int> 查询 = 从 x 在 数组
    其中 x % 2 == 1
    按 x 降序排列
        选择 x * x;
// 结果：49, 25, 1
```

[来自维基百科关于 C# 3.0 的文章示例，LINQ 子部分](#)

示例 1 使用了查询语法，设计上类似于 SQL 查询。

```
//示例 2
IQueryable<int> 查询 = 数组.Where(x => x % 2 == 1)
    .OrderByDescending(x => x)
    .Select(x => x * x);
// 结果：49, 25, 1, 使用前面示例中定义的“array”
```

[来自维基百科关于C# 3.0的文章示例，LINQ小节](#)

示例2使用方法语法实现与示例1相同的结果。

需要注意的是，在C#中，LINQ查询语法是[语法糖](#)，用于LINQ方法语法。编译器在编译时将查询转换为方法调用。有些查询必须用方法语法表达。
摘自MSDN - “例如，必须使用方法调用来表达检索匹配指定条件的元素数量的查询。”

Chapter 75: C# 3.0 Features

Section 75.1: Implicitly typed variables (var)

The var keyword allows a programmer to implicitly type a variable at compile time. var declarations have the same type as explicitly declared variables.

```
var squaredNumber = 10 * 10;
var squaredNumberDouble = 10.0 * 10.0;
var builder = new StringBuilder();
var anonymousObject = new
{
    One = SquaredNumber,
    Two = SquaredNumberDouble,
    Three = Builder
}
```

The types of the above variables are int, double, StringBuilder, and an anonymous type respectively.

It is important to note that a var variable is not dynamically typed. SquaredNumber = Builder is not valid since you are trying to set an int to an instance of StringBuilder

Section 75.2: Language Integrated Queries (LINQ)

```
//Example 1
int[] array = { 1, 5, 2, 10, 7 };

// Select squares of all odd numbers in the array sorted in descending order
IQueryable<int> query = from x in array
    where x % 2 == 1
    orderby x descending
    select x * x;
// Result: 49, 25, 1
```

[Example from wikipedia article on C# 3.0, LINQ sub-section](#)

Example 1 uses query syntax which was designed to look similar to SQL queries.

```
//Example 2
IQueryable<int> query = array.Where(x => x % 2 == 1)
    .OrderByDescending(x => x)
    .Select(x => x * x);
// Result: 49, 25, 1 using 'array' as defined in previous example
```

[Example from wikipedia article on C# 3.0, LINQ sub-section](#)

Example 2 uses method syntax to achieve the same outcome as example 1.

It is important to note that, in C#, LINQ query syntax is [syntactic sugar](#) for LINQ method syntax. The compiler translates the queries into method calls at compile time. Some queries have to be expressed in method syntax.
[From MSDN](#) - "For example, you must use a method call to express a query that retrieves the number of elements that match a specified condition."

第75.3节：Lambda表达式

Lambda表达式是匿名方法的扩展，允许隐式类型参数和返回值。它们的语法比匿名方法更简洁，遵循函数式编程风格。

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var numberList = new List<int> {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        var sumOfSquares = numberList.Select( number => number * number )
            .Aggregate( (int first, int second) => { return first + second; } );
        Console.WriteLine( sumOfSquares );
    }
}
```

上述代码将输出数字1到10的平方和到控制台。

第一个lambda表达式对列表中的数字进行平方。由于只有一个参数，括号可以省略。如果愿意，也可以加上括号：

```
.Select( 数字 => 数字 * 数字);
```

或者显式地指定参数类型，但这时必须加括号：

```
.Select( (int 数字) => 数字 * 数字);
```

lambda体是一个表达式，具有隐式返回值。如果需要，也可以使用语句体。
这对于更复杂的lambda表达式很有用。

```
.Select( 数字 => { return 数字 * 数字; } );
```

Select方法返回一个包含计算值的新IEnumerable。

第二个lambda表达式对从Select方法返回的列表中的数字求和。由于有多个参数，必须加括号。参数类型被显式指定，但这不是必须的。下面的方法是等价的。

```
.Aggregate( (first, second) => { return first + second; } );
```

就像这个：

```
.Aggregate( (int first, int second) => first + second );
```

第75.4节：匿名类型

匿名类型提供了一种方便的方式，将一组只读属性封装到单个对象中，而无需先显式定义类型。类型名称由编译器生成，源代码级别不可见。每个属性的类型由编译器推断。

你可以使用new关键字，后跟大括号{}来创建匿名类型。在大括号内，

Section 75.3: Lambda expresions

Lambda Expressions are an extension of anonymous methods that allow for implicitly typed parameters and return values. Their syntax is less verbose than anonymous methods and follows a functional programming style.

```
using System;
using System.Collections.Generic;
using System.Linq;

public class Program
{
    public static void Main()
    {
        var numberList = new List<int> {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        var sumOfSquares = numberList.Select( number => number * number )
            .Aggregate( (int first, int second) => { return first + second; } );
        Console.WriteLine( sumOfSquares );
    }
}
```

The above code will output the sum of the squares of the numbers 1 through 10 to the console.

The first lambda expression squares the numbers in the list. Since there is only 1 parameter parenthesis may be omitted. You can include parenthesis if you wish:

```
.Select( (number) => number * number);
```

or explicitly type the parameter but then parenthesis are required:

```
.Select( (int number) => number * number);
```

The lambda body is an expression and has an implicit return. You can use a statement body if you want as well.
This is useful for more complex lambdas.

```
.Select( number => { return number * number; } );
```

The select method returns a new IEnumerable with the computed values.

The second lambda expression sums the numbers in list returned from the select method. Parentheses are required as there are multiple parameters. The types of the parameters are explicitly typed but this is not necessary. The below method is equivalent.

```
.Aggregate( (first, second) => { return first + second; } );
```

As is this one:

```
.Aggregate( (int first, int second) => first + second );
```

Section 75.4: Anonymous types

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first. The type name is generated by the compiler and is not available at the source code level. The type of each property is inferred by the compiler.

You can make anonymous types by using the new keyword followed by a curly brace {}. Inside the curly braces, you

你可以像下面代码中那样定义属性。

```
var v = new { Amount = 108, Message = "Hello" };
```

也可以创建匿名类型数组。见下面代码：

```
var a = new[] {
    new {
        水果 = "苹果",
        颜色 = "红色"
    },
    新 {
        水果 = "香蕉",
        颜色 = "黄色"
    }
};
```

或者将其与 LINQ 查询一起使用：

```
var 产品查询 = from 产品 in 产品集合
               select new { 产品.颜色, 产品.价格 };
```

could define properties like on code below.

```
var v = new { Amount = 108, Message = "Hello" };
```

It's also possible to create an array of anonymous types. See code below:

```
var a = new[] {
    new {
        Fruit = "Apple",
        Color = "Red"
    },
    new {
        Fruit = "Banana",
        Color = "Yellow"
    }
};
```

Or use it with LINQ queries:

```
var productQuery = from prod in products
                   select new { prod.Color, prod.Price };
```

第76章：异常处理

第76.1节：创建自定义异常

允许你实现可以像其他异常一样抛出的自定义异常。当你希望在运行时使你的异常与其他错误区分开时，这样做是有意义的。

在本例中，我们将创建一个自定义异常，以便清晰地处理应用程序在解析复杂输入时可能遇到的问题。

创建自定义异常类

要创建自定义异常，请创建一个Exception的子类：

```
public class ParserException : Exception
{
    public ParserException() :
        base("解析出错，且没有更多的附加信息。") { }
}
```

当你想向捕获者提供更多信息时，自定义异常非常有用：

```
public class ParserException : Exception
{
    public ParserException(string fileName, int lineNumber) :
        base($"解析错误发生在 {fileName}:{lineNumber}")
    {
        FileName = fileName;
        LineNumber = lineNumber;
    }
    public string FileName {get; private set;}
    public int LineNumber {get; private set;}
}
```

现在，当你catch(ParserException x)时，你将拥有额外的语义信息来细化异常处理。

自定义类可以实现以下功能以支持更多场景。

重新抛出

在解析过程中，原始异常仍然很重要。在此示例中，它是一个FormatException，因为代码尝试解析一段字符串，该字符串预期是一个数字。在这种情况下，自定义异常应支持包含“InnerException”：

```
//新构造函数：
ParserException(string msg, Exception inner) : base(msg, inner) { }
```

序列化

在某些情况下，您的异常可能需要跨越AppDomain边界。如果您的解析器在其自己的AppDomain中运行以支持新的解析器配置的热重载，就会出现这种情况。在Visual Studio中，您可以使用Exception 模板生成如下代码。

```
[Serializable]
public class ParserException : Exception
{ }
```

Chapter 76: Exception Handling

Section 76.1: Creating Custom Exceptions

You are allowed to implement custom exceptions that can be thrown just like any other exception. This makes sense when you want to make your exceptions distinguishable from other errors during runtime.

In this example we will create a custom exception for clear handling of problems the application may have while parsing a complex input.

Creating Custom Exception Class

To create a custom exception create a sub-class of Exception:

```
public class ParserException : Exception
{
    public ParserException() :
        base("The parsing went wrong and we have no additional information.") { }
}
```

Custom exception become very useful when you want to provide additional information to the catcher:

```
public class ParserException : Exception
{
    public ParserException(string fileName, int lineNumber) :
        base($"Parser error in {fileName}:{lineNumber}")
    {
        FileName = fileName;
        LineNumber = lineNumber;
    }
    public string FileName {get; private set;}
    public int LineNumber {get; private set;}
}
```

Now, when you catch(ParserException x) you will have additional semantics to fine-tune exception handling.

Custom classes can implement the following features to support additional scenarios.

re-throwing

During the parsing process, the original exception is still of interest. In this example it is a FormatException because the code attempts to parse a piece of string, which is expected to be a number. In this case the custom exception should support the inclusion of the 'InnerException':

```
//new constructor:
ParserException(string msg, Exception inner) : base(msg, inner) { }
```

serialization

In some cases your exceptions may have to cross AppDomain boundaries. This is the case if your parser is running in its own AppDomain to support hot reloading of new parser configurations. In Visual Studio, you can use Exception template to generate code like this.

```
[Serializable]
public class ParserException : Exception
{ }
```

```

// 无参构造函数允许抛出异常时不提供任何信息，包括错误消息。如果你的异常在没有任何附加细节的情况下仍有意义，则应包含此构造函数。应通过调用基类构造函数来设置消息（默认消息没有帮助）。
public ParserException()
    : base("解析器失败。")
{}

// 带消息参数的构造函数允许覆盖默认错误消息。
// 如果用户能够提供比通用自动生成消息更有用的消息，则应包含此构造函数。
public ParserException(string message)
    : base(message)
{}

// 支持序列化的构造函数。如果您的异常包含自定义属性，请在此处读取它们的值。
protected ParserException(SerializationInfo info, StreamingContext context)
    : base(info, context)
{}

```

使用 ParserException

```

try
{
    Process.StartRun(fileName)
}
catch (ParserException ex)
{
    Console.WriteLine($"{ex.Message} 在 ${ex.FileName}:${ex.LineNumber}");
}
catch (PostProcessException x)
{
    ...
}

```

您也可以使用自定义异常来捕获和封装异常。这样，许多不同的错误可以转换为对应用程序更有用的单一错误类型：

```

try
{
    int foo = int.Parse(token);
}
catch (FormatException ex)
{
    //假设您添加了这个构造函数
    throw new ParserException(
        $"无法将 {token} 读取为数字。",
        FileName,
        LineNumber,
        ex);
}

```

在通过抛出自定义异常来处理异常时，通常应在 `InnerException` 属性中包含对原始异常的引用，如上所示。

安全问题

如果暴露异常的原因可能会通过让用户看到应用程序的内部工作原理而危及安全，那么包装内部异常可能不是一个好主意。如果你正在创建一个将被他人使用的类库，这种情况可能适用。

```

// Constructor without arguments allows throwing your exception without
// providing any information, including error message. Should be included
// if your exception is meaningful without any additional details. Should
// set message by calling base constructor (default message is not helpful).
public ParserException()
    : base("Parser failure.")
{}

// Constructor with message argument allows overriding default error message.
// Should be included if users can provide more helpful messages than
// generic automatically generated messages.
public ParserException(string message)
    : base(message)
{}

// Constructor for serialization support. If your exception contains custom
// properties, read their values here.
protected ParserException(SerializationInfo info, StreamingContext context)
    : base(info, context)
{}

```

Using the ParserException

```

try
{
    Process.StartRun(fileName)
}
catch (ParserException ex)
{
    Console.WriteLine($"{ex.Message} in ${ex.FileName}:${ex.LineNumber}");
}
catch (PostProcessException x)
{
    ...
}

```

You may also use custom exceptions for catching and wrapping exceptions. This way many different errors can be converted into a single error type that is more useful to the application:

```

try
{
    int foo = int.Parse(token);
}
catch (FormatException ex)
{
    //Assuming you added this constructor
    throw new ParserException(
        $"Failed to read {token} as number.",
        FileName,
        LineNumber,
        ex);
}

```

When handling exceptions by raising your own custom exceptions, you should generally include a reference to the original exception in the `InnerException` property, as shown above.

Security Concerns

If exposing the reason for the exception might compromise security by allowing users to see the inner workings of your application it can be a bad idea to wrap the inner exception. This might apply if you are creating a class library

这适用于将被他人使用的类库。

以下是如何在不包装内部异常的情况下引发自定义异常的方法：

```
try
{
    // ...
}
catch (SomeStandardException ex)
{
    // ...
    throw new MyCustomException(someMessage);
}
```

结论

在引发自定义异常时（无论是包装内部异常还是引发未包装的新异常），都应引发对调用者有意义的异常。例如，类库的用户可能对该库的内部工作原理了解不多。类库依赖项抛出的异常对用户来说没有意义。相反，用户希望得到一个与类库如何错误使用这些依赖项相关的异常。

```
try
{
    // ...
}
catch (IOException ex)
{
    // ...
    throw new StorageServiceException(@"存储服务在保存您的数据时遇到问题。请查看内部异常以获取技术细节
。
如果您无法解决问题，请致电555-555-1234寻求技术支持。
}
```

第76.2节：finally块

```
try
{
    /* 可能抛出异常的代码 */
}
catch (Exception)
{
    /* 处理异常 */
}
finally
{
    /* 无论是否抛出或捕获异常，都会执行的代码 */
}
```

try / catch / finally 块在读取文件时非常有用。

例如：

```
FileStream f = null;

try
{
    f = File.OpenRead("file.txt");
    /* 在这里处理文件 */
}
```

that will be used by others.

Here is how you could raise a custom exception without wrapping the inner exception:

```
try
{
    // ...
}
catch (SomeStandardException ex)
{
    // ...
    throw new MyCustomException(someMessage);
}
```

Conclusion

When raising a custom exception (either with wrapping or with an unwrapped new exception), you should raise an exception that is meaningful to the caller. For instance, a user of a class library may not know much about how that library does its internal work. The exceptions that are thrown by the dependencies of the class library are not meaningful. Rather, the user wants an exception that is relevant to how the class library is using those dependencies in an erroneous way.

```
try
{
    // ...
}
catch (IOException ex)
{
    // ...
    throw new StorageServiceException(@"The Storage Service encountered a problem saving
your data. Please consult the inner exception for technical details.
If you are not able to resolve the problem, please call 555-555-1234 for technical
assistance.", ex);
}
```

Section 76.2: Finally block

```
try
{
    /* code that could throw an exception */
}
catch (Exception)
{
    /* handle the exception */
}
finally
{
    /* Code that will be executed, regardless if an exception was thrown / caught or not */
}
```

The try / catch / finally block can be very handy when reading from files.

For example:

```
FileStream f = null;

try
{
    f = File.OpenRead("file.txt");
    /* process the file here */
}
```

```

}
finally
{
    f?.Close(); // f 可能为 null, 因此使用了空条件操作符。
}

```

try 块后必须跟一个 `catch` 或 `finally` 块。然而，由于没有 `catch` 块，执行将导致终止。在终止之前，`finally` 块中的语句将被执行。

在文件读取中，我们本可以使用 `using` 块，因为 `FileStream`（即 `OpenRead` 返回的对象）实现了 `IDisposable` 接口。

即使在 `try` 块中有 `return` 语句，`finally` 块通常也会执行；只有少数情况下不会执行：

- 当发生 [StackOverflow](#) 时。
- [Environment.FailFast](#)
- 应用程序进程被终止，通常是由外部原因导致。

第 76.3 节：最佳实践

备忘单

应做	不要
使用控制语句控制流程	使用异常控制流程
通过日志记录跟踪被忽略（吸收）的异常	忽略异常
使用 <code>throw</code> 重复抛出异常	重新抛出异常 - <code>throw new ArgumentNullException()</code> 或 <code>throw ex</code>
抛出预定义的系统异常	抛出类似于预定义系统异常的自定义异常
如果对应用逻辑至关重要，则抛出自定义/预定义异常	抛出自定义/预定义异常以在流程中发出警告
捕获你想要处理的异常	捕获所有异常
不要用异常来管理业务逻辑。	

流程控制不应通过异常来完成。应使用条件语句。如果控制可以通过 `if-else` 语句清晰地完成，不要使用异常，因为这会降低可读性和性能。

请考虑以下由坏习惯先生提供的代码片段：

```

// 这是一个“不应该做”的代码片段示例
object myObject;
void DoingSomethingWithMyObject()
{
    Console.WriteLine(myObject.ToString());
}

```

当执行到 `Console.WriteLine(myObject.ToString());` 时，应用程序将抛出一个 `NullReferenceException`。糟糕的实践先生意识到 `myObject` 是 `null`，并修改了他的代码片段以捕获并处理 `NullReferenceException`：

```

// 这是一个“不应该做”的代码片段示例
object myObject;
void DoingSomethingWithMyObject()
{

```

```

}
finally
{
    f?.Close(); // f may be null, so use the null conditional operator.
}

```

A try block must be followed by either a `catch` or a `finally` block. However, since there is no catch block, the execution will cause termination. Before termination, the statements inside the finally block will be executed.

In the file-reading we could have used a `using` block as `FileStream` (what `OpenRead` returns) implements `IDisposable`.

Even if there is a `return` statement in `try` block, the `finally` block will usually execute; there are a few cases where it will not:

- When a [StackOverflow](#) occurs.
- [Environment.FailFast](#)
- The application process is killed, usually by an external source.

Section 76.3: Best Practices

Cheatsheet

DO	DON'T
Control flow with control statements	Control flow with exceptions
Keep track of ignored (absorbed) exception by logging	Ignore exception
Repeat exception by using <code>throw</code>	Re-throw exception - <code>throw new ArgumentNullException()</code> or <code>throw ex</code>
Throw predefined system exceptions	Throw custom exceptions similar to predefined system exceptions
Throw custom/predefined exception if it is crucial to application logic	Throw custom/predefined exceptions to state a warning in flow
Catch exceptions that you want to handle	Catch every exception
DO NOT manage business logic with exceptions.	

Flow control should NOT be done by exceptions. Use conditional statements instead. If a control can be done with `if-else` statement clearly, don't use exceptions because it reduces readability and performance.

Consider the following snippet by Mr. Bad Practices:

```

// This is a snippet example for DO NOT
object myObject;
void DoingSomethingWithMyObject()
{
    Console.WriteLine(myObject.ToString());
}

```

When execution reaches `Console.WriteLine(myObject.ToString());` application will throw an `NullReferenceException`. Mr. Bad Practices realized that `myObject` is `null` and edited his snippet to catch & handle `NullReferenceException`:

```

// This is a snippet example for DO NOT
object myObject;
void DoingSomethingWithMyObject()
{

```

```

try
{
Console.WriteLine(myObject.ToString());
}
catch(NullReferenceException ex)
{
    // 嗯, 如果我创建一个新的对象实例并赋值给myObject :
myObject = new object();
    // 很好, 现在我可以继续使用myObject了
DoSomethingElseWithMyObject();
}
}

```

由于之前的代码片段只涵盖了异常的逻辑, 如果此时myObject不是null, 我该怎么办? 我应该在哪里处理这部分逻辑? 就在Console.WriteLine(myObject.ToString());之后吗? 还是在try...catch块之后?

那么最佳实践先生会怎么做呢? 他会如何处理这个问题?

```

// 这是一个用于DO的代码片段示例
object myObject;
void DoingSomethingWithMyObject()
{
    if(myObject == null)
        myObject = new object();

    // 当执行到这里时, 我们确定 myObject 不是 null
    DoSomethingElseWithMyObject();
}

```

最佳实践先生用更少的代码实现了相同的逻辑, 且逻辑清晰易懂。

不要重新抛出异常

重新抛出异常代价高昂, 会对性能产生负面影响。对于经常失败的代码, 可以使用设计模式来最小化性能问题。本主题介绍了两种在异常可能显著影响性能时有用的设计模式。

不要吞掉异常且不记录日志

```

try
{
    // 可能抛出异常的代码
}
catch(Exception ex)
{
    // 空的 catch 块, 糟糕的做法
}

```

永远不要吞掉异常。忽略异常可以暂时避免问题, 但会给后续的维护带来混乱。在记录异常时, 应始终记录异常实例, 以便完整的堆栈跟踪被记录, 而不仅仅是异常消息。

```

try
{
    // 可能抛出异常的代码
}
catch(NullException ex)
{

```

```

try
{
    Console.WriteLine(myObject.ToString());
}
catch(NullReferenceException ex)
{
    // Hmmm, if I create a new instance of object and assign it to myObject:
myObject = new object();
    // Nice, now I can continue to work with myObject
    DoSomethingElseWithMyObject();
}
}

```

Since previous snippet only covers logic of exception, what should I do if myObject is not null at this point? Where should I cover this part of logic? Right after Console.WriteLine(myObject.ToString())? How about after the try...catch block?

How about Mr. Best Practices? How would he handle this?

```

// This is a snippet example for DO
object myObject;
void DoingSomethingWithMyObject()
{
    if(myObject == null)
        myObject = new object();

    // When execution reaches this point, we are sure that myObject is not null
    DoSomethingElseWithMyObject();
}

```

Mr. Best Practices achieved same logic with fewer code and a clear & understandable logic.

DO NOT re-throw Exceptions

Re-throwing exceptions is expensive. It negatively impact performance. For code that routinely fails, you can use design patterns to minimize performance issues. [This topic](#) describes two design patterns that are useful when exceptions might significantly impact performance.

DO NOT absorb exceptions with no logging

```

try
{
    //Some code that might throw an exception
}
catch(Exception ex)
{
    //empty catch block, bad practice
}

```

Never swallow exceptions. Ignoring exceptions will save that moment but will create a chaos for maintainability later. When logging exceptions, you should always log the exception instance so that the complete stack trace is logged and not the exception message only.

```

try
{
    //Some code that might throw an exception
}
catch(NullException ex)
{

```

```
LogManager.Log(ex.ToString());  
}
```

不要捕获你无法处理的异常

许多资源，例如this one，强烈建议你考虑为什么要在捕获异常的地方捕获它。你应该只在能够处理异常的位置捕获异常。如果你能在那采取措施缓解问题，比如尝试替代算法、连接备用数据库、尝试另一个文件名、等待30秒后重试，或者通知管理员，那么你可以捕获错误并执行这些操作。如果你无法合理且切实地做任何事情，就“放手”让异常在更高层级被处理。如果异常足够严重，且除了让整个程序崩溃外没有合理的选项，那么就让它崩溃。

```
try  
{  
    //尝试将数据保存到主数据库。  
}  
catch(SqlException ex)  
{  
    //尝试将数据保存到备用数据库。  
}  
//如果抛出的是SqlException以外的异常，我们无能为力。让异常向上传递到可以处理它的层级。
```

```
LogManager.Log(ex.ToString());  
}
```

Do not catch exceptions that you cannot handle

Many resources, such as [this one](#), strongly urge you to consider why you are catching an exception in the place that you are catching it. You should only catch an exception if you can handle it at that location. If you can do something there to help mitigate the problem, such as trying an alternative algorithm, connecting to a backup database, trying another filename, waiting 30 seconds and trying again, or notifying an administrator, you can catch the error and do that. If there is nothing that you can plausibly and reasonably do, just "let it go" and let the exception be handled at a higher level. If the exception is sufficiently catastrophic and there is no reasonable option other than for the entire program to crash because of the severity of the problem, then let it crash.

```
try  
{  
    //Try to save the data to the main database.  
}  
catch(SqlException ex)  
{  
    //Try to save the data to the alternative database.  
}  
//If anything other than a SqlException is thrown, there is nothing we can do here. Let the exception  
bubble up to a level where it can be handled.
```

第76.4节：异常反模式

吞噬异常

应该始终以以下方式重新抛出异常：

```
try  
{  
    ...  
}  
catch (Exception ex)  
{  
    ...  
    throw;  
}
```

像下面这样重新抛出异常会混淆原始异常并丢失原始堆栈跟踪。绝对不要这样做！在捕获和重新抛出之前的堆栈跟踪将会丢失。

```
try  
{  
    ...  
}  
catch (Exception ex)  
{  
    ...  
    throw ex;  
}
```

棒球异常处理

不应将异常用作正常流程控制结构（如 if-then 语句和 while 循环）的替代。这种反模式有时被称为“棒球异常处理”。

Section 76.4: Exception Anti-patterns

Swallowing Exceptions

One should always re-throw exception in the following way:

```
try  
{  
    ...  
}  
catch (Exception ex)  
{  
    ...  
    throw;  
}
```

Re-throwing an exception like below will obfuscate the original exception and will lose the original stack trace. One should never do this! The stack trace prior to the catch and rethrow will be lost.

```
try  
{  
    ...  
}  
catch (Exception ex)  
{  
    ...  
    throw ex;  
}
```

Baseball Exception Handling

One should not use exceptions as a [substitute for normal flow control constructs](#) like if-then statements and while loops. This anti-pattern is sometimes called [Baseball Exception Handling](#).

以下是该反模式的一个示例：

```
try
{
    while (AccountManager.HasMoreAccounts())
    {
        account = AccountManager.GetNextAccount();
        if (account.Name == userName)
        {
            //我们找到了它
            throw new AccountFoundException(account);
        }
    }
}
catch (AccountFoundException found)
{
    Console.WriteLine("这是您的账户详情: " + found.Account.Details.ToString());
}
```

这里有一个更好的方法：

```
Account found = null;
while (AccountManager.HasMoreAccounts() && (found==null))
{
    account = AccountManager.GetNextAccount();
    if (account.Name == userName)
    {
        //我们找到了它
        found = account;
    }
}
Console.WriteLine("这是您的账户详情: " + found.Details.ToString());
catch (Exception)
```

几乎没有（有人说根本没有！）理由在代码中捕获通用异常类型。你应该只捕获你预期会发生的异常类型，否则你会隐藏代码中的错误。

```
try
{
    var f = File.Open(myfile);
    // 执行某些操作
}
catch (Exception x)
{
    // 假设文件未找到
    Console.WriteLine("无法打开文件");
    // 但错误可能是由于文件处理代码中的错误导致的 NullReferenceException ?
}
```

最好这样做：

```
try
{
    var f = File.Open(myfile);
    // 执行通常不会抛出异常的操作
}
catch (IOException)
{
    Console.WriteLine("文件未找到");
}
```

Here is an example of the anti-pattern:

```
try
{
    while (AccountManager.HasMoreAccounts())
    {
        account = AccountManager.GetNextAccount();
        if (account.Name == userName)
        {
            //We found it
            throw new AccountFoundException(account);
        }
    }
}
catch (AccountFoundException found)
{
    Console.WriteLine("Here are your account details: " + found.Account.Details.ToString());
}
```

Here is a better way to do it:

```
Account found = null;
while (AccountManager.HasMoreAccounts() && (found==null))
{
    account = AccountManager.GetNextAccount();
    if (account.Name == userName)
    {
        //We found it
        found = account;
    }
}
Console.WriteLine("Here are your account details: " + found.Details.ToString());
catch (Exception)
```

There are almost no (some say none!) reasons to catch the generic exception type in your code. You should catch only the exception types you expect to happen, because you hide bugs in your code otherwise.

```
try
{
    var f = File.Open(myfile);
    // do something
}
catch (Exception x)
{
    // Assume file not found
    Console.WriteLine("Could not open file");
    // but maybe the error was a NullReferenceException because of a bug in the file handling code?
}
```

Better do:

```
try
{
    var f = File.Open(myfile);
    // do something which should normally not throw exceptions
}
catch (IOException)
{
    Console.WriteLine("File not found");
}
```

```

}
// 不幸的是，这个异常类并不继承自上述异常类，因此需要单独声明
catch (UnauthorizedAccessException)
{
    Console.WriteLine("权限不足");
}

```

如果发生任何其他异常，我们故意让应用程序崩溃，这样它会直接进入调试器，我们可以修复问题。无论如何，我们都应该发布会发生除这些异常以外其他异常的程序，所以程序崩溃并不是问题。

下面的例子也很糟糕，因为它使用异常来绕过编程错误。这并不是异常设计的初衷。

```

public void DoSomething(String s)
{
    if (s == null)
        throw new ArgumentNullException(nameof(s));
    // 实现代码写在这里
}

try
{
    DoSomething(myString);
}
catch(ArgumentNullException x)
{
    // 如果发生这种情况，说明我们有一个编程错误，应该检查
    // myString 最初为什么是 null。
}

```

第76.5节：基本异常处理

```

try
{
    /* 可能抛出异常的代码 */
}
catch (Exception ex)
{
    /* 处理异常 */
}

```

注意，用相同代码处理所有异常通常不是最佳方法。
当任何内部异常处理程序失败时，这通常作为最后的手段使用。

第76.6节：处理特定异常类型

```

try
{
    /* 打开文件的代码 */
}
catch (System.IO.FileNotFoundException)
{
    /* 处理文件未找到的代码 */
}
catch (System.IO.UnauthorizedAccessException)
{
    /* 处理无权访问文件的代码 */
}

```

```

}
// Unfortunately, this one does not derive from the above, so declare separately
catch (UnauthorizedAccessException)
{
    Console.WriteLine("Insufficient rights");
}

```

If any other exception happens, we purposely let the application crash, so it directly steps in the debugger and we can fix the problem. We mustn't ship a program where any other exceptions than these happen anyway, so it's not a problem to have a crash.

The following is a bad example, too, because it uses exceptions to work around a programming error. That's not what they're designed for.

```

public void DoSomething(String s)
{
    if (s == null)
        throw new ArgumentNullException(nameof(s));
    // Implementation goes here
}

try
{
    DoSomething(myString);
}
catch(ArgumentNullException x)
{
    // if this happens, we have a programming error and we should check
    // why myString was null in the first place.
}

```

Section 76.5: Basic Exception Handling

```

try
{
    /* code that could throw an exception */
}
catch (Exception ex)
{
    /* handle the exception */
}

```

Note that handling all exceptions with the same code is often not the best approach. This is commonly used when any inner exception handling routines fail, as a last resort.

Section 76.6: Handling specific exception types

```

try
{
    /* code to open a file */
}
catch (System.IO.FileNotFoundException)
{
    /* code to handle the file being not found */
}
catch (System.IO.UnauthorizedAccessException)
{
    /* code to handle not being allowed access to the file */
}

```

```

}
catch (System.IO.IOException)
{
    /* 处理 IOException 或其除前两者外的子类的代码 */
}
catch (System.Exception)
{
    /* 处理其他错误的代码 */
}

```

注意异常是按顺序评估且应用继承关系的。因此需要从最具体的异常开始，最后处理它们的祖先。在任何时刻，只有一个 catch 块会被执行。

第76.7节：来自一种方法的汇总例外/多重例外

谁说你不能在一个方法中抛出多个异常。如果你不习惯使用AggregateException，你可能会想创建自己的数据结构来表示许多错误情况。当然，也有其他不是异常的数据结构更为理想，比如验证结果。即使你使用AggregateException，你可能总是在接收端处理它们，却没有意识到它们对你是有用的。

一个方法执行时，即使整体上是失败的，你也可能希望在抛出的异常中突出显示多个错误。例如，这种行为可以在并行方法中看到，任务被拆分成多个线程，任何一个线程都可能抛出异常，这些异常需要被报告。下面是一个简单的例子，展示你如何从中受益：

```

public void Run()
{
    try
    {
        this.SillyMethod(1, 2);
    }
    catch (AggregateException ex)
    {
        Console.WriteLine(ex.Message);
        foreach (Exception innerException in ex.InnerExceptions)
        {
            Console.WriteLine(innerException.Message);
        }
    }
}

private void SillyMethod(int input1, int input2)
{
    var exceptions = new List<Exception>();

    if (input1 == 1)
    {
        exceptions.Add(new ArgumentException("我不喜欢数字一"));
    }
    if (input2 == 2)
    {
        exceptions.Add(new ArgumentException("我不喜欢数字二"));
    }
    if (exceptions.Any())
    {
        throw new AggregateException("执行过程中发生了有趣的事情", exceptions);
    }
}

```

```

}
catch (System.IO.IOException)
{
    /* code to handle IOException or it's descendant other than the previous two */
}
catch (System.Exception)
{
    /* code to handle other errors */
}

```

Be careful that exceptions are evaluated in order and inheritance is applied. So you need to start with the most specific ones and end with their ancestor. At any given point, only one catch block will get executed.

Section 76.7: Aggregate exceptions / multiple exceptions from one method

Who says you cannot throw multiple exceptions in one method. If you are not used to playing around with AggregateExceptions you may be tempted to create your own data-structure to represent many things going wrong. There are of course were another data-structure that is not an exception would be more ideal such as the results of a validation. Even if you do play with AggregateExceptions you may be on the receiving side and always handling them not realizing they can be of use to you.

It is quite plausible to have a method execute and even though it will be a failure as a whole you will want to highlight multiple things that went wrong in the exceptions that are thrown. As an example this behavior can be seen with how Parallel methods work were a task broken into multiple threads and any number of them could throw exceptions and this needs to be reported. Here is a silly example of how you could benefit from this:

```

public void Run()
{
    try
    {
        this.SillyMethod(1, 2);
    }
    catch (AggregateException ex)
    {
        Console.WriteLine(ex.Message);
        foreach (Exception innerException in ex.InnerExceptions)
        {
            Console.WriteLine(innerException.Message);
        }
    }
}

private void SillyMethod(int input1, int input2)
{
    var exceptions = new List<Exception>();

    if (input1 == 1)
    {
        exceptions.Add(new ArgumentException("I do not like ones"));
    }
    if (input2 == 2)
    {
        exceptions.Add(new ArgumentException("I do not like twos"));
    }
    if (exceptions.Any())
    {
        throw new AggregateException("Funny stuff happened during execution", exceptions);
    }
}

```

第76.8节：抛出异常

当发生异常情况时，您的代码可以且通常应该抛出异常。

```
public void WalkInto(Destination destination)
{
    if (destination.Name == "魔多")
    {
        throw new InvalidOperationException("不能简单地走进魔多。");
    }
    // ... 在此处实现您的正常行走代码。
}
```

Section 76.8: Throwing an exception

Your code can, and often should, throw an exception when something unusual has happened.

```
public void WalkInto(Destination destination)
{
    if (destination.Name == "Mordor")
    {
        throw new InvalidOperationException("One does not simply walk into Mordor.");
    }
    // ... Implement your normal walking code here.
}
```

第76.9节：未处理异常和线程异常

AppDomain.UnhandledException 此事件提供未捕获异常的通知。它允许应用程序在系统默认处理器向用户报告异常并终止应用程序之前，记录有关异常的信息。如果有足够的应用程序状态信息，可采取其他操作—例如保存程序数据以便后续恢复。需谨慎，因为当异常未被处理时，程序数据可能会损坏。

```
/// <summary>
/// 应用程序的主入口点。
/// </summary>
[STAThread]
private static void Main(string[] args)
{
    AppDomain.CurrentDomain.UnhandledException += new
    UnhandledExceptionEventHandler(UnhandledException);
}
```

Application.ThreadException 此事件允许您的Windows窗体应用程序处理在Windows窗体线程中发生的其他未处理异常。将事件处理器附加到ThreadException事件，以处理这些异常，否则会使您的应用程序处于未知状态。尽可能应通过结构化异常处理块来处理异常。

```
/// <summary>
/// 应用程序的主入口点。
/// </summary>
[STAThread]
private static void Main(string[] args)
{
    AppDomain.CurrentDomain.UnhandledException += new
    UnhandledExceptionEventHandler(UnhandledException);
    Application.ThreadException += new ThreadExceptionEventHandler(ThreadException);
}
```

最后是异常处理

```
static void UnhandledException(object sender, UnhandledExceptionEventArgs e)
{
    Exception ex = (Exception)e.ExceptionObject;
    // 你的代码
}
```

Section 76.9: Unhandled and Thread Exception

AppDomain.UnhandledException This event provides notification of uncaught exceptions. It allows the application to log information about the exception before the system default handler reports the exception to the user and terminates the application. If sufficient information about the state of the application is available, other actions may be undertaken — such as saving program data for later recovery. Caution is advised, because program data can become corrupted when exceptions are not handled.

```
/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
private static void Main(string[] args)
{
    AppDomain.CurrentDomain.UnhandledException += new
    UnhandledExceptionEventHandler(UnhandledException);
}
```

Application.ThreadException This event allows your Windows Forms application to handle otherwise unhandled exceptions that occur in Windows Forms threads. Attach your event handlers to the ThreadException event to deal with these exceptions, which will leave your application in an unknown state. Where possible, exceptions should be handled by a structured exception handling block.

```
/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
private static void Main(string[] args)
{
    AppDomain.CurrentDomain.UnhandledException += new
    UnhandledExceptionEventHandler(UnhandledException);
    Application.ThreadException += new ThreadExceptionEventHandler(ThreadException);
}
```

And finally exception handling

```
static void UnhandledException(object sender, UnhandledExceptionEventArgs e)
{
    Exception ex = (Exception)e.ExceptionObject;
    // your code
}
```

```

static void ThreadException(object sender, ThreadExceptionEventArgs e)
{
    Exception ex = e.Exception;
    // 你的代码
}

```

第76.10节：为WCF服务实现IErrorHandler

为WCF服务实现IErrorHandler是集中处理错误和日志记录的好方法。这里展示的实现应能捕获因调用你的某个WCF服务而抛出的任何未处理异常。示例中还展示了如何返回自定义对象，以及如何返回JSON而非默认的XML。

实现 IErrorHandler：

```

using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;
using System.Runtime.Serialization.Json;
using System.ServiceModel;
using System.ServiceModel.Web;

namespace BehaviorsAndInspectors
{
    public class ErrorHandler : IErrorHandler
    {

        public bool HandleError(Exception ex)
        {
            // 在此记录异常
            return true;
        } // 结束

        public void ProvideFault(Exception ex, MessageVersion version, ref Message fault)
        {
            // 获取当前上下文的传出响应部分
            var response = WebOperationContext.Current.OutgoingResponse;

            // 设置默认的HTTP状态码
            response.StatusCode = HttpStatusCode.InternalServerError;

            // 添加指定我们使用JSON的ContentType头
            response.ContentType = new MediaTypeHeaderValue("application/json").ToString();

            // 创建返回的错误消息 (注意ref参数) 与
            BaseDataResponseContract
            fault = Message.CreateMessage(
                version,
                string.Empty,
                new CustomReturnType { ErrorMessage = "发生未处理的异常！" },
                new DataContractJsonSerializer(typeof(BaseDataResponseContract)), new List<Type> {
                    typeof(BaseDataResponseContract) });

            if (ex.GetType() == typeof(VariousExceptionTypes))
            {
                // 你可能想在这里捕获不同类型的异常并进行不同的处理
            }
        }
    }
}

```

```

static void ThreadException(object sender, ThreadExceptionEventArgs e)
{
    Exception ex = e.Exception;
    // your code
}

```

Section 76.10: Implementing IErrorHandler for WCF Services

Implementing IErrorHandler for WCF services is a great way to centralize error handling and logging. The implementation shown here should catch any unhandled exception that is thrown as a result of a call to one of your WCF services. Also shown in this example is how to return a custom object, and how to return JSON rather than the default XML.

Implement IErrorHandler:

```

using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;
using System.Runtime.Serialization.Json;
using System.ServiceModel;
using System.ServiceModel.Web;

namespace BehaviorsAndInspectors
{
    public class ErrorHandler : IErrorHandler
    {

        public bool HandleError(Exception ex)
        {
            // Log exceptions here
            return true;
        } // end

        public void ProvideFault(Exception ex, MessageVersion version, ref Message fault)
        {
            // Get the outgoing response portion of the current context
            var response = WebOperationContext.Current.OutgoingResponse;

            // Set the default http status code
            response.StatusCode = HttpStatusCode.InternalServerError;

            // Add ContentType header that specifies we are using JSON
            response.ContentType = new MediaTypeHeaderValue("application/json").ToString();

            // Create the fault message that is returned (note the ref parameter) with
            BaseDataResponseContract
            fault = Message.CreateMessage(
                version,
                string.Empty,
                new CustomReturnType { ErrorMessage = "An unhandled exception occurred!" },
                new DataContractJsonSerializer(typeof(BaseDataResponseContract)), new List<Type> {
                    typeof(BaseDataResponseContract) });

            if (ex.GetType() == typeof(VariousExceptionTypes))
            {
                // You might want to catch different types of exceptions here and process them
                differently
            }
        }
    }
}

```

```

    // 告诉 WCF 使用 JSON 编码而非默认的 XML
    var webBodyFormatMessageProperty = new
WebBodyFormatMessageProperty(WebContentFormat.Json);
    fault.Properties.Add(WebBodyFormatMessageProperty.Name, webBodyFormatMessageProperty);

} // 结束

} // 类结束

} // 命名空间结束

```

在此示例中，我们将处理程序附加到服务行为。你也可以以类似方式将其附加到 IEndpointBehavior、IContractBehavior 或 IOperationBehavior。

附加到服务行为：

```

using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Configuration;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;

namespace BehaviorsAndInspectors
{
    public class ErrorHandlerExtension : BehaviorExtensionElement, IServiceBehavior
    {
        public override Type BehaviorType
        {
            get { return GetType(); }
        }

        protected override object CreateBehavior()
        {
            return this;
        }

        private IErrorHandler GetInstance()
        {
            return new ErrorHandler();
        }

        void IServiceBehavior.AddBindingParameters(ServiceDescription serviceDescription,
ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints, BindingParameterCollection
bindingParameters) { } // end

        void IServiceBehavior.ApplyDispatchBehavior(ServiceDescription serviceDescription,
ServiceHostBase serviceHostBase)
        {
            var errorHandlerInstance = GetInstance();

            foreach (ChannelDispatcher dispatcher in serviceHostBase.ChannelDispatchers)
            {
dispatcher.ErrorHandlers.Add(errorHandlerInstance);
            }
        }

        void IServiceBehavior.Validate(ServiceDescription serviceDescription, ServiceHostBase
serviceHostBase) { } // end
    }
}

```

```

    // Tell WCF to use JSON encoding rather than default XML
    var webBodyFormatMessageProperty = new
WebBodyFormatMessageProperty(WebContentFormat.Json);
    fault.Properties.Add(WebBodyFormatMessageProperty.Name, webBodyFormatMessageProperty);

} // end

} // end class

} // end namespace

```

In this example we attach the handler to the service behavior. You could also attach this to IEndpointBehavior, IContractBehavior, or IOperationBehavior in a similar way.

Attach to Service Behaviors:

```

using System;
using System.Collections.ObjectModel;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Configuration;
using System.ServiceModel.Description;
using System.ServiceModel.Dispatcher;

namespace BehaviorsAndInspectors
{
    public class ErrorHandlerExtension : BehaviorExtensionElement, IServiceBehavior
    {
        public override Type BehaviorType
        {
            get { return GetType(); }
        }

        protected override object CreateBehavior()
        {
            return this;
        }

        private IErrorHandler GetInstance()
        {
            return new ErrorHandler();
        }

        void IServiceBehavior.AddBindingParameters(ServiceDescription serviceDescription,
ServiceHostBase serviceHostBase, Collection<ServiceEndpoint> endpoints, BindingParameterCollection
bindingParameters) { } // end

        void IServiceBehavior.ApplyDispatchBehavior(ServiceDescription serviceDescription,
ServiceHostBase serviceHostBase)
        {
            var errorHandlerInstance = GetInstance();

            foreach (ChannelDispatcher dispatcher in serviceHostBase.ChannelDispatchers)
            {
                dispatcher.ErrorHandlers.Add(errorHandlerInstance);
            }
        }

        void IServiceBehavior.Validate(ServiceDescription serviceDescription, ServiceHostBase
serviceHostBase) { } // end
    }
}

```

```
} // 类结束
```

```
} // 命名空间结束
```

Web.config 中的配置：

```
...
<system.serviceModel>

    <services>
        <service name="WebServices.MyService">
            <endpoint binding="webHttpBinding" contract="WebServices.IMyService" />
        </service>
    </services>

    <extensions>
        <behaviorExtensions>
            <!-- 此扩展用于 WCF 错误处理-->
            <add name="ErrorHandlerBehavior"
                type="WebServices.BehaviorsAndInspectors.ErrorHandlerExtensionBehavior, WebServices,
                Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
        </behaviorExtensions>
    </extensions>

    <behaviors>
        <serviceBehaviors>
            <behavior>
                <serviceMetadata httpGetEnabled="true" />
                <serviceDebug includeExceptionDetailInFaults="true" />
                <ErrorHandlerBehavior />
            </behavior>
        </serviceBehaviors>
    </behaviors>

    ...
</system.serviceModel>
```

以下是一些可能对该主题有帮助的链接：

[https://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.ierrorhandler\(v=vs.100\).aspx](https://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.ierrorhandler(v=vs.100).aspx)

<http://www.brainhud.com/cards/5218/25441/which-four-behavior-interfaces-exist-for-interacting-with-a-service-or-client-description-what-methods-do-they-implement-and>

其他示例：

[IErrorHandler 返回错误的消息体，当 HTTP 状态码为 401 未授权时](#)

[IErrorHandler 似乎没有处理我在 WCF 中的错误.....有什么想法吗？](#)

[如何让自定义的 WCF 错误处理程序返回带有非 OK HTTP 状态码的 JSON 响应？](#)

[如何为 HttpClient 请求设置 Content-Type 头？](#)

第 76.11 节：使用异常对象

你可以在自己的代码中创建并抛出异常。实例化异常的方式与其他 C# 对象相同。

```
} // end class
```

```
} // end namespace
```

Configs in Web.config:

```
...
<system.serviceModel>

    <services>
        <service name="WebServices.MyService">
            <endpoint binding="webHttpBinding" contract="WebServices.IMyService" />
        </service>
    </services>

    <extensions>
        <behaviorExtensions>
            <!-- This extension if for the WCF Error Handling-->
            <add name="ErrorHandlerBehavior"
                type="WebServices.BehaviorsAndInspectors.ErrorHandlerExtensionBehavior, WebServices,
                Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
        </behaviorExtensions>
    </extensions>

    <behaviors>
        <serviceBehaviors>
            <behavior>
                <serviceMetadata httpGetEnabled="true" />
                <serviceDebug includeExceptionDetailInFaults="true" />
                <ErrorHandlerBehavior />
            </behavior>
        </serviceBehaviors>
    </behaviors>

    ...
</system.serviceModel>
```

Here are a few links that may be helpful on this topic:

[https://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.ierrorhandler\(v=vs.100\).aspx](https://msdn.microsoft.com/en-us/library/system.servicemodel.dispatcher.ierrorhandler(v=vs.100).aspx)

<http://www.brainhud.com/cards/5218/25441/which-four-behavior-interfaces-exist-for-interacting-with-a-service-or-client-description-what-methods-do-they-implement-and>

Other Examples:

[IErrorHandler returning wrong message body when HTTP status code is 401 Unauthorized](#)

[IErrorHandler doesn't seem to be handling my errors in WCF .. any ideas?](#)

[How to make custom WCF error handler return JSON response with non-OK http code?](#)

[How do you set the Content-Type header for an HttpClient request?](#)

Section 76.11: Using the exception object

You are allowed to create and throw exceptions in your own code. Instantiating an exception is done the same way that any other C# object.

```

Exception ex = new Exception();

// 带有接受消息字符串的重载构造函数
Exception ex = new Exception("错误消息");

```

然后你可以使用throw关键字来抛出异常：

```

try
{
    throw new Exception("Error");
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message); // 将 'Error' 记录到输出窗口
}

```

注意：如果你在 catch 块中抛出一个新的异常，确保将原始异常作为“inner exception”传递，例如

```

void DoSomething()
{
    int b=1; int c=5;
    try
    {
        var a = 1;
        b = a - 1;
        c = a / b;
        a = a / c;
    }
    catch (DivideByZeroException dEx) when (b==0)
    {
        // 我们抛出的是同一种异常
        throw new DivideByZeroException("不能除以 b, 因为它是零", dEx);
    }
    catch (DivideByZeroException dEx) when (c==0)
    {
        // 我们抛出的是同一种异常
        throw new DivideByZeroException("不能除以 c, 因为它是零", dEx);
    }
}

void Main()
{
    try
    {
        DoSomething();
    }
    catch (Exception ex)
    {
        // 记录完整的错误信息 (包括内部异常)
        Console.WriteLine(ex.ToString());
    }
}

```

在这种情况下，假设异常无法被处理，但会向消息中添加一些有用的信息（并且外层异常块仍然可以通过 ex.InnerException 访问原始异常）。

它将显示类似如下内容：

```

Exception ex = new Exception();

// constructor with an overload that takes a message string
Exception ex = new Exception("Error message");

```

You can then use the `throw` keyword to raise the exception:

```

try
{
    throw new Exception("Error");
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message); // Logs 'Error' to the output window
}

```

Note: If you're throwing a new exception inside a catch block, ensure that the original exception is passed as "inner exception", e.g.

```

void DoSomething()
{
    int b=1; int c=5;
    try
    {
        var a = 1;
        b = a - 1;
        c = a / b;
        a = a / c;
    }
    catch (DivideByZeroException dEx) when (b==0)
    {
        // we're throwing the same kind of exception
        throw new DivideByZeroException("Cannot divide by b because it is zero", dEx);
    }
    catch (DivideByZeroException dEx) when (c==0)
    {
        // we're throwing the same kind of exception
        throw new DivideByZeroException("Cannot divide by c because it is zero", dEx);
    }
}

void Main()
{
    try
    {
        DoSomething();
    }
    catch (Exception ex)
    {
        // Logs full error information (incl. inner exception)
        Console.WriteLine(ex.ToString());
    }
}

```

In this case it is assumed that the exception cannot be handled but some useful information is added to the message (and the original exception can still be accessed via `ex.InnerException` by an outer exception block).

It will show something like:

```

System.DivideByZeroException : 无法除以 b，因为它是零 --> System.DivideByZeroException :
尝试除以零。
位于 UserQuery.g__DoSomething0_0(), 路径 C:[...]\LINQPadQuery.cs, 第 36 行
--- 内部异常堆栈跟踪的结尾 ---
位于 UserQuery.g__DoSomething0_0(), 路径 C:[...]\LINQPadQuery.cs, 第 42 行
位于 UserQuery.Main(), 路径 C:[...]\LINQPadQuery.cs, 第 55 行

```

如果你在 LinqPad 中尝试这个示例，你会注意到行号不是很有意义（它们并不总是能帮到你）。但如上所述传递有用错误文本通常能显著减少定位错误位置的时间，在这个示例中错误位置显然是

```
c = a / b;
```

在函数 DoSomething() 中。

[在 .NET Fiddle 中试试](#)

第76.12节：异常和try catch块的嵌套

可以将一个异常 / try catch 块嵌套在另一个里面。

这样可以管理能够正常工作的较小代码块，而不会破坏整个机制。

```

try
{
    //这里是一些代码
    try
    {
        //一些会抛出异常的东西。例如：除以0
    }
    catch (DivideByZeroException dzEx)
    {
        //仅在这里处理此异常
        //从这里抛出的异常将传递给父catch块
    }
    finally
    {
        //完成后要做的任何事情。
    }
    //从这里恢复并正常继续；
}
catch(Exception e)
{
    //在这里处理
}

```

注意：避免在抛出到父级捕获块时吞掉异常

```

System.DivideByZeroException: Cannot divide by b because it is zero --> System.DivideByZeroException:
Attempted to divide by zero.
at UserQuery.g__DoSomething0_0() in C:[...]\LINQPadQuery.cs:line 36
--- End of inner exception stack trace ---
at UserQuery.g__DoSomething0_0() in C:[...]\LINQPadQuery.cs:line 42
at UserQuery.Main() in C:[...]\LINQPadQuery.cs:line 55

```

If you're trying this example in LinqPad, you'll notice that the line numbers aren't very meaningful (they don't always help you). But passing a helpful error text as suggested above oftentimes significantly reduces the time to track down the location of the error, which is in this example clearly the line

```
c = a / b;
```

in function DoSomething().

[Try it in .NET Fiddle](#)

Section 76.12: Nesting of Exceptions & try catch blocks

One is able to nest one exception / try catch block inside the other.

This way one can manage small blocks of code which are capable of working without disrupting your whole mechanism.

```

try
{
    //some code here
    try
    {
        //some thing which throws an exception. For Eg : divide by 0
    }
    catch (DivideByZeroException dzEx)
    {
        //handle here only this exception
        //throw from here will be passed on to the parent catch block
    }
    finally
    {
        //any thing to do after it is done.
    }
    //resume from here & proceed as normal;
}
catch(Exception e)
{
    //handle here
}

```

Note: Avoid Swallowing Exceptions when throwing to the parent catch block

第77章：NullReferenceException（空引用异常）

第77.1节：NullReferenceException（空引用异常）解释

当你尝试访问一个引用对象的非静态成员（属性、方法、字段或事件），但该对象为null时，会抛出NullReferenceException异常。

```
Car myFirstCar = new Car();
Car mySecondCar = null;
Color myFirstColor = myFirstCar.Color; // 没问题，因为myFirstCar存在/不为null
Color mySecondColor = mySecondCar.Color; // 抛出NullReferenceException异常
// 因为mySecondCar为null，但我们仍尝试访问它的Color属性。
```

调试此类异常非常简单：在抛出异常的那一行，你只需查看每个'.'或'['之前的内容，偶尔也要看'('之前的内容。

```
myGarage.CarCollection[currentIndex.Value].Color = theCarInTheStreet.Color;
```

我的异常是从哪里来的？可能是：

- myGarage 是 null
- myGarage.CarCollection 是 null
- currentIndex 是 null
- myGarage.CarCollection[currentIndex.Value] 是 null
- theCarInTheStreet 是 null

在调试模式下，你只需将鼠标光标放在这些元素上，就能找到你的空引用。然后，你需要做的是理解为什么它没有值。修正方法完全取决于你的方法目标。

你是否忘记实例化/初始化它了？

```
myGarage.CarCollection = new Car[10];
```

如果对象是null，你是否应该做不同的处理？

```
if (myGarage == null)
{
    Console.WriteLine("也许你应该先买个车库！");
}
```

或者可能有人给了你一个不该是null的参数：

```
if (街上的车 == null)
{
    throw new ArgumentNullException("街上的车");
}
```

无论如何，请记住方法绝不应抛出 NullReferenceException。如果抛出了，说明你忘记检查某些内容。

Chapter 77: NullReferenceException

Section 77.1: NullReferenceException explained

A NullReferenceException is thrown when you try to access a non-static member (property, method, field or event) of a reference object but it is null.

```
Car myFirstCar = new Car();
Car mySecondCar = null;
Color myFirstColor = myFirstCar.Color; // No problem as myFirstCar exists / is not null
Color mySecondColor = mySecondCar.Color; // Throws a NullReferenceException
// as mySecondCar is null and yet we try to access its color.
```

To debug such an exception, it's quite easy: on the line where the exception is thrown, you just have to look before every '.' or '[', or on rare occasions '('.

```
myGarage.CarCollection[currentIndex.Value].Color = theCarInTheStreet.Color;
```

Where does my exception come from? Either:

- myGarage is null
- myGarage.CarCollection is null
- currentIndex is null
- myGarage.CarCollection[currentIndex.Value] is null
- theCarInTheStreet is null

In debug mode, you only have to put your mouse cursor on every of these elements and you will find your null reference. Then, what you have to do is understand why it doesn't have a value. The correction totally depends on the goal of your method.

Have you forgotten to instantiate/initialize it?

```
myGarage.CarCollection = new Car[10];
```

Are you supposed to do something different if the object is null?

```
if (myGarage == null)
{
    Console.WriteLine("Maybe you should buy a garage first!");
}
```

Or maybe someone gave you a null argument, and was not supposed to:

```
if (theCarInTheStreet == null)
{
    throw new ArgumentNullException("theCarInTheStreet");
}
```

In any case, remember that a method should never throw a NullReferenceException. If it does, that means you have forgotten to check something.

第78章：处理将字符串转换为其他类型时的 FormatException

第78.1节：将字符串转换为整数

有多种方法可以显式地将字符串转换为整数，例如：

1. Convert.ToInt16();
2. Convert.ToInt32();
3. Convert.ToInt64();
4. int.Parse();

但是如果输入字符串包含非数字字符，所有这些方法都会抛出FormatException异常。为此，我们需要编写额外的异常处理（try..catch）来处理这种情况。

示例说明：

所以，假设我们的输入是：

```
string inputString = "10.2";
```

示例 1：Convert.ToInt32()

```
int convertedInt = Convert.ToInt32(inputString); // 转换失败  
// 抛出异常“输入字符串的格式不正确。”
```

注意：其他提到的方法同样如此，即 - Convert.ToInt16()；和 Convert.ToInt64()；

示例 2：int.Parse()

```
int convertedInt = int.Parse(inputString); // 同样结果“输入字符串的格式不正确。”
```

我们如何规避这个问题？

如前所述，处理异常时我们通常需要使用try..catch，如下所示：

```
try  
{  
    string inputString = "10.2";  
    int convertedInt = int.Parse(inputString);  
}  
catch (Exception Ex)  
{  
    //显示一些消息，表示转换失败。  
}
```

但是，在任何地方都使用try..catch并不是一个好习惯，并且可能存在一些场景，我们希望在输入错误时返回0，（如果按照上述方法，我们需要在catch块中将0赋值给convertedInt）。为处理这种情况，我们可以使用一个特殊的方法，称为.TryParse()。

Chapter 78: Handling FormatException when converting string to other types

Section 78.1: Converting string to integer

There are various methods available for explicitly converting a `string` to an `integer`, such as:

1. Convert.ToInt16();
2. Convert.ToInt32();
3. Convert.ToInt64();
4. `int.Parse()`;

But all these methods will throw a `FormatException`, if the input string contains non-numeric characters. For this, we need to write an additional exception handling(`try..catch`) to deal them in such cases.

Explanation with Examples:

So, let our input be:

```
string inputString = "10.2";
```

Example 1: Convert.ToInt32()

```
int convertedInt = Convert.ToInt32(inputString); // Failed to Convert  
// Throws an Exception "Input string was not in a correct format."
```

Note: Same goes for the other mentioned methods namely - `Convert.ToInt16()`; and `Convert.ToInt64()`;

Example 2: int.Parse()

```
int convertedInt = int.Parse(inputString); // Same result "Input string was not in a correct format."
```

How do we circumvent this?

As told earlier, for handling the exceptions we usually need a `try..catch` as shown below:

```
try  
{  
    string inputString = "10.2";  
    int convertedInt = int.Parse(inputString);  
}  
catch (Exception Ex)  
{  
    //Display some message, that the conversion has failed.  
}
```

But, using the `try..catch` everywhere will not be a good practice, and there may be some scenarios where we wanted to give 0 if the input is wrong, (*If we follow the above method we need to assign 0 to convertedInt from the catch block*). To handle such scenarios we can make use of a special method called `.TryParse()`.

.TryParse()方法内部包含异常处理，它会将输出结果赋给out参数，并返回一个布尔值，指示转换状态（如果转换成功返回true；失败返回false）。根据返回值，我们可以判断转换是否成功。来看一个示例：

用法1：将返回值存储在布尔变量中

```
int convertedInt; // 需要的整数  
bool isSuccessConversion = int.TryParse(inputString, out convertedInt);
```

我们可以在执行后检查变量isSuccessConversion以确认转换状态。如果它为假那么convertedInt的值将是0（如果转换失败时你想要0，则无需检查返回值）。

用法 2：使用 if 检查返回值

```
if (int.TryParse(inputString, out convertedInt))  
{  
    // convertedInt 将包含转换后的值  
    // 继续使用该值  
}  
else  
{  
    // 显示错误信息  
}
```

用法 3：如果不关心返回值，可以使用以下方法（无论是否转换成功，0 都是可以的）

```
int.TryParse(inputString, out convertedInt);  
// 使用 convertedInt 的值  
// 但如果未转换成功，值将为 0
```

The .TryParse() method having an internal Exception handling, which will give you the output to the `out` parameter, and returns a Boolean value indicating the conversion status (`true` if the conversion was successful; `false` if it failed). Based on the return value we can determine the conversion status. Lets see one Example:

Usage 1: Store the return value in a Boolean variable

```
int convertedInt; // Be the required integer  
bool isSuccessConversion = int.TryParse(inputString, out convertedInt);
```

We can check The variable isSuccessConversion after the Execution to check the conversion status. If it is false then the value of convertedInt will be 0 (no need to check the return value if you want 0 for conversion failure).

Usage 2: Check the return value with if

```
if (int.TryParse(inputString, out convertedInt))  
{  
    // convertedInt will have the converted value  
    // Proceed with that  
}  
else  
{  
    // Display an error message  
}
```

Usage 3: Without checking the return value you can use the following, if you don't care about the return value (converted or not, 0 will be ok)

```
int.TryParse(inputString, out convertedInt);  
// use the value of convertedInt  
// But it will be 0 if not converted
```

第 79 章：读取并理解堆栈跟踪

堆栈跟踪在调试程序时非常有帮助。当程序抛出异常时，您会获得堆栈跟踪，有时程序异常终止时也会获得。

第79.1节：Windows窗体中简单NullReferenceException的堆栈跟踪

让我们创建一小段抛出异常的代码：

```
private void button1_Click(object sender, EventArgs e)
{
    string msg = null;
    msg.ToCharArray();
}
```

如果执行这段代码，我们会得到以下异常和堆栈跟踪：

```
System.NullReferenceException: “对象引用未设置为对象的实例。”
在 WindowsFormsApplication1.Form1.button1_Click(Object sender, EventArgs e) in
F:\WindowsFormsApplication1\WindowsFormsApplication1\Form1.cs:第 29 行
在 System.Windows.Forms.Control.OnClick(EventArgs e)
在 System.Windows.Forms.Button.OnClick(EventArgs e)
在 System.Windows.Forms.Button.OnMouseUp(MouseEventArgs mevent)
```

堆栈跟踪还会继续，但这部分内容足以满足我们的需求。

在堆栈跟踪的顶部，我们看到这一行：

在 WindowsFormsApplication1.Form1.button1_Click(Object sender, EventArgs e) 中
F:\WindowsFormsApplication1\WindowsFormsApplication1\Form1.cs:第29行

这是最重要的部分。它告诉我们异常发生的确切行数：Form1.cs 的第29行。
所以，这就是你开始查找的地方。

第二行是

在 System.Windows.Forms.Control.OnClick(EventArgs e)

这是调用button1_Click的方法。所以现在我们知道发生错误的button1_Click是从System.Windows.Forms.Control.OnClick调用的。

我们可以这样继续；第三行是

在 System.Windows.Forms.Button.OnClick(EventArgs e)

这反过来是调用System.windows.Forms.Control.OnClick的代码。

堆栈跟踪是直到你的代码遇到异常时被调用的函数列表。通过跟踪

Chapter 79: Read & Understand Stacktraces

A stack trace is a great aid when debugging a program. You will get a stack trace when your program throws an Exception, and sometimes when the program terminates abnormally.

Section 79.1: Stack trace for a simple NullReferenceException in Windows Forms

Let's create a small piece of code that throws an exception:

```
private void button1_Click(object sender, EventArgs e)
{
    string msg = null;
    msg.ToCharArray();
}
```

If we execute this, we get the following Exception and stack trace:

```
System.NullReferenceException: “Object reference not set to an instance of an object.”
at WindowsFormsApplication1.Form1.button1_Click(Object sender, EventArgs e) in
F:\WindowsFormsApplication1\WindowsFormsApplication1\Form1.cs:line 29
at System.Windows.Forms.Control.OnClick(EventArgs e)
at System.Windows.Forms.Button.OnClick(EventArgs e)
at System.Windows.Forms.Button.OnMouseUp(MouseEventArgs mevent)
```

The stack trace goes on like that, but this part will suffice for our purposes.

At the top of the stack trace we see the line:

at WindowsFormsApplication1.Form1.button1_Click(Object sender, EventArgs e) in
F:\WindowsFormsApplication1\WindowsFormsApplication1\Form1.cs:line 29

This is the most important part. It tells us the *exact* line where the Exception occurred: line 29 in Form1.cs .
So, this is where you begin your search.

The second line is

at System.Windows.Forms.Control.OnClick(EventArgs e)

This is the method that called button1_Click. So now we know that button1_Click, where the error occurred, was called from System.Windows.Forms.Control.OnClick.

We can continue like this; the third line is

at System.Windows.Forms.Button.OnClick(EventArgs e)

This is, in turn, the code that called System.windows.Forms.Control.OnClick.

The stack trace is the list of functions that was called until your code encountered the Exception. And by following

通过这个，你可以弄清楚你的代码执行了哪个路径，直到遇到问题为止！

请注意，堆栈跟踪包含来自 .Net 系统的调用；你通常不需要跟踪所有微软的 System.Windows.Forms 代码来找出错误原因，只需关注属于你自己应用程序的代码。

那么，为什么这被称为“堆栈跟踪”呢？

因为，每当程序调用一个方法时，它都会记录自己之前的位置。它有一个名为“堆栈”的数据结构，用来存放它的上一个位置。

如果方法执行完毕，它会查看堆栈，找到调用该方法之前的位置——然后从那里继续执行。

所以堆栈让计算机知道在调用新方法之前它停在哪里了。

但它也作为调试的辅助工具。就像侦探追踪罪犯犯罪时的步骤一样，程序员可以利用堆栈追踪程序崩溃前的执行步骤。

this, you can figure out which execution path your code followed until it ran into trouble!

Note that the stack trace includes calls from the .Net system; you don't normally need to follow all Microsofts System.Windows.Forms code to find out what went wrong, only the code that belongs to your own application.

So, why is this called a "stack trace"?

Because, every time a program calls a method, it keeps track of where it was. It has a data structure called the "stack", where it dumps its last location.

If it is done executing the method, it looks on the stack to see where it was before it called the method - and continues from there.

So the stack lets the computer know where it left off, before calling a new method.

But it also serves as a debugging help. Like a detective tracing the steps that a criminal took when committing their crime, a programmer can use the stack to trace the steps a program took before it crashed.

belindoc.com

第80章：诊断

第80.1节：使用TraceListeners重定向日志输出

您可以通过向Debug.Listeners集合添加TextWriterTraceListener，将调试输出重定向到文本文件。

```
public static void Main(string[] args)
{
    TextWriterTraceListener myWriter = new TextWriterTraceListener(@"debug.txt");
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");

    myWriter.Flush();
}
```

您可以使用ConsoleTraceListener将调试输出重定向到控制台应用程序的输出流。

```
public static void Main(string[] args)
{
    ConsoleTraceListener myWriter = new ConsoleTraceListener();
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");
}
```

第80.2节：Debug.WriteLine

当应用程序以调试配置编译时，向Listeners集合中的跟踪监听器写入内容。

```
public static void Main(string[] args)
{
    Debug.WriteLine("Hello");
}
```

在 Visual Studio 或 Xamarin Studio 中，这将显示在应用程序输出窗口中。这是由于 TraceListenerCollection 中存在默认跟踪监听器（[default trace listener](#)）。

Chapter 80: Diagnostics

Section 80.1: Redirecting log output with TraceListeners

You can redirect the debug output to a text file by adding a TextWriterTraceListener to the Debug.Listeners collection.

```
public static void Main(string[] args)
{
    TextWriterTraceListener myWriter = new TextWriterTraceListener(@"debug.txt");
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");

    myWriter.Flush();
}
```

You can redirect the debug output to a console application's out stream using a ConsoleTraceListener.

```
public static void Main(string[] args)
{
    ConsoleTraceListener myWriter = new ConsoleTraceListener();
    Debug.Listeners.Add(myWriter);
    Debug.WriteLine("Hello");
}
```

Section 80.2: Debug.WriteLine

Writes to the trace listeners in the Listeners collection when the application is compiled in debug configuration.

```
public static void Main(string[] args)
{
    Debug.WriteLine("Hello");
}
```

In Visual Studio or Xamarin Studio this will appear in the Application Output window. This is due to the presence of the [default trace listener](#) in the TraceListenerCollection.

第81章：溢出

第81.1节：整数溢出

整数有最大存储容量。当超过该限制时，它会回绕到负数一侧。对于 int 类型，最大值是 2147483647

```
int x = int.MaxValue;           //.MaxValue 是 2147483647
checked(x + 1);                //显式使操作不进行检查，以便当项目设置中启用算术溢出/下溢检查时，示例仍能正常工作

Console.WriteLine(x);          //将打印 -2147483648
Console.WriteLine(int.MinValue); //与最小值相同
```

对于超出此范围的任何整数，请使用 System.Numerics 命名空间中的 BigInteger 数据类型。更多信息请查看以下链接 [https://msdn.microsoft.com/en-us/library/system.numerics.biginteger\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.numerics.biginteger(v=vs.110).aspx)

第81.2节：操作过程中的溢出

溢出也会在操作过程中发生。在以下示例中，x 是 int 类型，数字 1 默认也是 int 类型。因此，加法是 int 类型的加法，结果也是 int 类型，并且会发生溢出。

```
int x = int.MaxValue;           //.MaxValue 是 2147483647
long y = x + 1;                 //将会溢出
Console.WriteLine(y);           //将打印 -2147483648
Console.WriteLine(int.MinValue); //与最小值相同
```

你可以通过使用1L来防止这种情况。现在1将是一个long类型，加法也将是一个long加法

```
int x = int.MaxValue;           //.MaxValue是2147483647
long y = x + 1L;                //这样是可以的
Console.WriteLine(y);           //将打印2147483648
```

第81.3节：顺序很重要

以下代码中存在溢出

```
int x = int.MaxValue;
Console.WriteLine(x + x + 1L); //打印-1
```

而在以下代码中没有溢出

```
int x = int.MaxValue;
Console.WriteLine(x + 1L + x); //打印4294967295
```

这是由于操作的从左到右顺序。在第一个代码片段中， $x + x$ 发生溢出，之后才变成long类型。另一方面， $x + 1L$ 先变成long类型，然后再将x加到该值上。

Chapter 81: Overflow

Section 81.1: Integer overflow

There is a maximum capacity an integer can store. And when you go over that limit, it will loop back to the negative side. For `int`, it is `2147483647`

```
int x = int.MaxValue;           //.MaxValue is 2147483647
x = unchecked(x + 1);           //make operation explicitly unchecked so that the example also
                                //works when the check for arithmetic overflow/underflow is enabled in the project settings
Console.WriteLine(x);           //Will print -2147483648
Console.WriteLine(int.MinValue); //Same as Min value
```

For any integers out of this range use namespace `System.Numerics` which has datatype `BigInteger`. Check below link for more information [https://msdn.microsoft.com/en-us/library/system.numerics.biginteger\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.numerics.biginteger(v=vs.110).aspx)

Section 81.2: Overflow during operation

Overflow also happens during the operation. In the following example, x is an `int`, 1 is an `int` by default. Therefore addition is an `int` addition. And the result will be an `int`. And it will overflow.

```
int x = int.MaxValue;           //.MaxValue is 2147483647
long y = x + 1;                 //It will be overflow
Console.WriteLine(y);           //Will print -2147483648
Console.WriteLine(int.MinValue); //Same as Min value
```

You can prevent that by using 1L. Now 1 will be a `long` and addition will be a `long` addition

```
int x = int.MaxValue;           //.MaxValue is 2147483647
long y = x + 1L;                //It will be OK
Console.WriteLine(y);           //Will print 2147483648
```

Section 81.3: Ordering matters

There is overflow in the following code

```
int x = int.MaxValue;
Console.WriteLine(x + x + 1L); //prints -1
```

Whereas in the following code there is no overflow

```
int x = int.MaxValue;
Console.WriteLine(x + 1L + x); //prints 4294967295
```

This is due to the left-to-right ordering of the operations. In the first code fragment $x + x$ overflows and after that it becomes a `long`. On the other hand $x + 1L$ becomes `long` and after that x is added to this value.

第82章：入门：使用C#处理Json

以下内容将介绍使用C#语言处理Json的方法，以及序列化和反序列化的概念。

第82.1节：简单的Json示例

```
{  
    "id": 89,  
    "name": "奥尔德斯·赫胥黎",  
    "type": "作者",  
    "books": [  
        {  
            "name": "美丽新世界",  
            "date": 1932  
        },  
        {  
            "name": "加沙无眼",  
            "date": 1936  
        },  
        {  
            "name": "天才与女神",  
            "date": 1955  
        }  
    ]  
}
```

如果你是Json新手，这里有一个示例教程。

第82.2节：首先要做的事：用于处理Json的库

要使用C#处理Json，需要使用Newtonsoft (.net库)。该库提供了允许程序员序列化和反序列化对象等方法。如果你想了解其方法和用法，有一个教程。

如果你使用Visual Studio，进入工具/NuGet包管理器/管理解决方案的包/，在搜索栏输入“Newtonsoft”并安装该包。如果你没有NuGet，这个详细教程可能会帮助你。

第82.3节：C#实现

在阅读代码之前，理解一些主要概念很重要，这些概念将帮助你使用json编写应用程序。

序列化：将对象转换为可以通过应用程序传输的字节流的过程。

以下代码可以被序列化并转换成之前的json。

反序列化：将json/字节流转换为对象的过程。它正好是序列化的相反过程。之前的json可以反序列化为C#对象，如下面的示例所示。

为了解决这个问题，将json结构转换为类非常重要，以便使用之前描述的流程。

如果你使用Visual Studio，只需选择“编辑/特殊粘贴/粘贴JSON为类”并粘贴json结构，就可以自动将json转换为类。

Chapter 82: Getting Started: Json with C#

The following topic will introduce a way to work with Json using C# language and concepts of Serialization and Deserialization.

Section 82.1: Simple Json Example

```
{  
    "id": 89,  
    "name": "Aldous Huxley",  
    "type": "Author",  
    "books": [  
        {  
            "name": "Brave New World",  
            "date": 1932  
        },  
        {  
            "name": "Eyeless in Gaza",  
            "date": 1936  
        },  
        {  
            "name": "The Genius and the Goddess",  
            "date": 1955  
        }  
    ]  
}
```

If you are new into Json, here is an [exemplified tutorial](#).

Section 82.2: First things First: Library to work with Json

To work with Json using C#, it is need to use Newtonsoft (.net library). This library provides methods that allows the programmer serialize and deserialize objects and more. [There is a tutorial](#) if you want to know details about its methods and usages.

If you use Visual Studio, go to Tools/Nuget Package Manager/Manage Package to Solution/ and type "Newtonsoft" into the search bar and install the package. If you don't have NuGet, this [detailed tutorial](#) might help you.

Section 82.3: C# Implementation

Before reading some code, it is important to undersand the main concepts that will help to program applications using json.

Serialization: Process of converting a object into a stream of bytes that can be sent through applications.
The following code can be serialized and converted into the previous json.

Deserialization: Process of converting a json/stream of bytes into an object. Its exactly the opposite process of serialization. The previous json can be deserialized into an C# object as demonstrated in examples below.

To work this out, it is important to turn the json structure into classes in order to use processes already described. If you use Visual Studio, you can turn a json into a class automatically just by selecting "Edit/Paste Special/Paste JSON as Classes" and pasting the json structure.

```

using Newtonsoft.Json;

class Author
{
    [JsonProperty("id")] // 将下面的变量设置为表示json属性
    public int id;      // "id"
    [JsonProperty("name")]
    public string name;
    [JsonProperty("type")]
    public string type;
    [JsonProperty("books")]
    public Book[] books;

    public Author(int id, string name, string type, Book[] books) {
        this.id = id;
        this.name = name;
        this.type = type;
        this.books = books;
    }
}

class Book
{
    [JsonProperty("name")]
    public string name;
    [JsonProperty("date")]
    public DateTime date;
}

```

第82.4节：序列化

```

static void Main(string[] args)
{
    Book[] books = new Book[3];
    Author author = new Author(89, "Aldous Huxley", "Author", books);
    string objectDeserialized = JsonConvert.SerializeObject(author);
    //将作者转换为json
}

```

方法".SerializeObject"接收一个type object类型的参数，因此你可以传入任何内容。

第82.5节：反序列化

你可以从任何地方接收json，比如文件甚至服务器，因此以下代码中未包含接收部分。

```

static void Main(string[] args)
{
    string jsonExample; // 包含之前的json
    Author author = JsonConvert.DeserializeObject<Author>(jsonExample);
}

```

方法".DeserializeObject"将'jsonExample'反序列化为一个"Author"对象。这就是为什么在类定义中设置json变量很重要，以便该方法访问并填充它们。

第82.6节：序列化与反序列化通用工具函数

此示例用于所有类型对象的序列化和反序列化的通用函数。

```

using Newtonsoft.Json;

class Author
{
    [JsonProperty("id")] // Set the variable below to represent the json attribute
    public int id;      // "id"
    [JsonProperty("name")]
    public string name;
    [JsonProperty("type")]
    public string type;
    [JsonProperty("books")]
    public Book[] books;

    public Author(int id, string name, string type, Book[] books) {
        this.id = id;
        this.name = name;
        this.type = type;
        this.books = books;
    }
}

class Book
{
    [JsonProperty("name")]
    public string name;
    [JsonProperty("date")]
    public DateTime date;
}

```

Section 82.4: Serialization

```

static void Main(string[] args)
{
    Book[] books = new Book[3];
    Author author = new Author(89, "Aldous Huxley", "Author", books);
    string objectDeserialized = JsonConvert.SerializeObject(author);
    //Converting author into json
}

```

The method ".SerializeObject" receives as parameter a type object, so you can put anything into it.

Section 82.5: Deserialization

You can receive a json from anywhere, a file or even a server so it is not included in the following code.

```

static void Main(string[] args)
{
    string jsonExample; // Has the previous json
    Author author = JsonConvert.DeserializeObject<Author>(jsonExample);
}

```

The method ".DeserializeObject" deserializes 'jsonExample' into an "Author" object. This is why it is important to set the json variables in the classes definition, so the method access it in order to fill it.

Section 82.6: Serialization & De-Serialization Common Utilities function

This sample used to common function for all type object serialization and deserialization.

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Xml.Serialization;

namespace Framework
{
    public static class IGUtilities
    {
        public static string Serialization(this T obj)
        {
            string data = JsonConvert.SerializeObject(obj);
            return data;
        }

        public static T 反序列化(this string jsonData)
        {
            T copy = JsonConvert.DeserializeObject(jsonData);
            return copy;
        }

        public static T 克隆(this T obj)
        {
            string data = JsonConvert.SerializeObject(obj);
            T copy = JsonConvert.DeserializeObject(data);
            return copy;
        }
    }
}
```

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Xml.Serialization;

namespace Framework
{
    public static class IGUtilities
    {
        public static string Serialization(this T obj)
        {
            string data = JsonConvert.SerializeObject(obj);
            return data;
        }

        public static T Deserialization(this string jsonData)
        {
            T copy = JsonConvert.DeserializeObject(jsonData);
            return copy;
        }

        public static T Clone(this T obj)
        {
            string data = JsonConvert.SerializeObject(obj);
            T copy = JsonConvert.DeserializeObject(data);
            return copy;
        }
    }
}
```

第83章：使用 json.net

使用JSON.net JsonConverter类。

第83.1节：在简单值上使用 JsonConverter

示例：使用 JsonConverter 将 API 响应中的 runtime 属性反序列化为 Movies 模型中的Timespan对象

JSON (<http://www.omdbapi.com/?i=tt1663662>)

```
{  
    标题: "环太平洋",  
    年份: "2013",  
    评级: "PG-13",  
    上映日期: "2013年7月12日",  
    片长: "131分钟",  
    类型: "动作, 冒险, 科幻",  
    导演: "吉尔莫·德尔·托罗",  
    编剧: "特拉维斯·比查姆 (剧本), 吉尔莫·德尔·托罗 (剧本), 特拉维斯·比查姆 (故事)",  
    演员: "查理·汉纳姆, 迭戈·克拉滕霍夫, 伊德里斯·艾尔巴, 菊地凛子",  
    剧情: "在人类与怪兽海洋生物的战争持续进行之际, 一名前飞行员和一名实习生被配对驾驶一件看似过时的特殊武器, 拼命努力拯救世界免于末日灾难。",  
  
    语言: "英语, 日语, 粤语, 普通话",  
    国家: "美国",  
    奖项: "获得1项英国电影和电视艺术学院奖 (BAFTA) 提名。另有6项获奖和46项提名。",  
    海报:  
        "https://images-na.ssl-images-amazon.com/images/M/MV5BMTY3MTI5NjQ4N15BMl5BanBnXkFtZTcwOTU10TU00Q@._V1_SX300.jpg",  
    评分: [  
        来源: "互联网电影数据库",  
            评分: "7.0/10"  
        },  
        {  
            来源: "烂番茄",  
                评分: "71%"  
        },  
        {  
            来源: "Metacritic",  
                评分: "64/100"  
        }  
    ],  
    综合评分: "64",  
    imdb评分: "7.0",  
    imdb投票数: "398,198",  
    imdb编号: "tt1663662",  
    类型: "电影",  
    DVD发行日期: "2013年10月15日",  
    票房: "$101,785,482.00",  
    制作公司: "华纳兄弟影业",  
    官方网站: "http://pacificrimmovie.com",  
    响应: "True"  
}
```

电影模型

```
using Project.Serializers;  
using Newtonsoft.Json;  
using System;  
using System.Collections.Generic;
```

Chapter 83: Using json.net

Using [JSON.net JsonConverter class](#).

Section 83.1: Using JsonConverter on simple values

Example using JsonConverter to deserialize the runtime property from the api response into a [Timespan](#) Object in the Movies model

JSON (<http://www.omdbapi.com/?i=tt1663662>)

```
{  
    Title: "Pacific Rim",  
    Year: "2013",  
    Rated: "PG-13",  
    Released: "12 Jul 2013",  
    Runtime: "131 min",  
    Genre: "Action, Adventure, Sci-Fi",  
    Director: "Guillermo del Toro",  
    Writer: "Travis Beacham (screenplay), Guillermo del Toro (screenplay), Travis Beacham (story)",  
    Actors: "Charlie Hunnam, Diego Klattenhoff, Idris Elba, Rinko Kikuchi",  
    Plot: "As a war between humankind and monstrous sea creatures wages on, a former pilot and a trainee are paired up to drive a seemingly obsolete special weapon in a desperate effort to save the world from the apocalypse.",  
    Language: "English, Japanese, Cantonese, Mandarin",  
    Country: "USA",  
    Awards: "Nominated for 1 BAFTA Film Award. Another 6 wins & 46 nominations.",  
    Poster:  
        "https://images-na.ssl-images-amazon.com/images/M/MV5BMTY3MTI5NjQ4N15BMl5BanBnXkFtZTcwOTU10TU00Q@._V1_SX300.jpg",  
    Ratings: [  
        {  
            Source: "Internet Movie Database",  
            Value: "7.0/10"  
        },  
        {  
            Source: "Rotten Tomatoes",  
            Value: "71%"  
        },  
        {  
            Source: "Metacritic",  
            Value: "64/100"  
        }  
    ],  
    Metascore: "64",  
    imdbRating: "7.0",  
    imdbVotes: "398,198",  
    imdbID: "tt1663662",  
    Type: "movie",  
    DVD: "15 Oct 2013",  
    BoxOffice: "$101,785,482.00",  
    Production: "Warner Bros. Pictures",  
    Website: "http://pacificrimmovie.com",  
    Response: "True"  
}
```

Movie Model

```
using Project.Serializers;  
using Newtonsoft.Json;  
using System;  
using System.Collections.Generic;
```

```

using System.Linq;
using System.Runtime.Serialization;
using System.Threading.Tasks;

namespace Project.Models
{
    [DataContract]
    public class Movie
    {
        public Movie() { }

        [DataMember]
        public int Id { get; set; }

        [DataMember]
        public string ImdbId { get; set; }

        [DataMember]
        public string Title { get; set; }

        [DataMember]
        public DateTime Released { get; set; }

        [DataMember]
        [JsonConverter(typeof(RuntimeSerializer))]
        public TimeSpan Runtime { get; set; }
    }
}

```

RuntimeSerializer

```

using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.RegularExpressions;
using System.Threading.Tasks;

namespace Project.Serializers
{
    public class RuntimeSerializer : JsonConverter
    {
        public override bool CanConvert(Type objectType)
        {
            return objectType == typeof(TimeSpan);
        }

        public override object ReadJson(JsonReader reader, Type objectType, object existingValue,
JsonSerializer serializer)
        {
            if (reader.TokenType == JsonToken.Null)
                return null;

            JToken jt = JToken.Load(reader);
            String value = jt.Value<String>();

            Regex rx = new Regex("(\\s*)min$");
            value = rx.Replace(value, (m) => "");

            int timespanMin;
            if(!Int32.TryParse(value, out timespanMin))

```

```

using System.Linq;
using System.Runtime.Serialization;
using System.Threading.Tasks;

namespace Project.Models
{
    [DataContract]
    public class Movie
    {
        public Movie() { }

        [DataMember]
        public int Id { get; set; }

        [DataMember]
        public string ImdbId { get; set; }

        [DataMember]
        public string Title { get; set; }

        [DataMember]
        public DateTime Released { get; set; }

        [DataMember]
        [JsonConverter(typeof(RuntimeSerializer))]
        public TimeSpan Runtime { get; set; }
    }
}

```

RuntimeSerializer

```

using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text.RegularExpressions;
using System.Threading.Tasks;

namespace Project.Serializers
{
    public class RuntimeSerializer : JsonConverter
    {
        public override bool CanConvert(Type objectType)
        {
            return objectType == typeof(TimeSpan);
        }

        public override object ReadJson(JsonReader reader, Type objectType, object existingValue,
JsonSerializer serializer)
        {
            if (reader.TokenType == JsonToken.Null)
                return null;

            JToken jt = JToken.Load(reader);
            String value = jt.Value<String>();

            Regex rx = new Regex("(\\s*)min$");
            value = rx.Replace(value, (m) => "");

            int timespanMin;
            if(!Int32.TryParse(value, out timespanMin))

```

```

    {
        throw new NotSupportedException();
    }

    return new TimeSpan(0, timespanMin, 0);
}

public override void WriteJson(JsonWriter writer, object value, JsonSerializer serializer)
{
    serializer.Serialize(writer, value);
}
}

```

调用示例

```
Movie m = JsonConvert.DeserializeObject<Movie>(apiResponse);
```

第83.2节：收集JSON对象的所有字段

```

using Newtonsoft.Json.Linq;
using System.Collections.Generic;

public class JsonFieldsCollector
{
    private readonly Dictionary<string, JValue> fields;

    public JsonFieldsCollector(JToken token)
    {
        fields = new Dictionary<string, JValue>();
        CollectFields(token);
    }

    private void CollectFields(JToken jToken)
    {
        switch (jToken.Type)
        {
            case JTokenType.Object:
                foreach (var child in jToken.Children<JProperty>())
                    CollectFields(child);
                break;
            case JTokenType.Array:
                foreach (var child in jToken.Children())
                    CollectFields(child);
                break;
            case JTokenType.Property:
                CollectFields(((JProperty) jToken).Value);
                break;
            default:
                fields.Add(jToken.Path, (JValue)jToken);
                break;
        }
    }

    public IEnumerable<KeyValuePair<string, JValue>> GetAllFields() => fields;
}

```

用法：

```
var json = JToken.Parse(/* JSON string */);
var fieldsCollector = new JsonFieldsCollector(json);
```

```

    {
        throw new NotSupportedException();
    }

    return new TimeSpan(0, timespanMin, 0);
}

public override void WriteJson(JsonWriter writer, object value, JsonSerializer serializer)
{
    serializer.Serialize(writer, value);
}
}

```

Calling It

```
Movie m = JsonConvert.DeserializeObject<Movie>(apiResponse);
```

Section 83.2: Collect all fields of JSON object

```

using Newtonsoft.Json.Linq;
using System.Collections.Generic;

public class JsonFieldsCollector
{
    private readonly Dictionary<string, JValue> fields;

    public JsonFieldsCollector(JToken token)
    {
        fields = new Dictionary<string, JValue>();
        CollectFields(token);
    }

    private void CollectFields(JToken jToken)
    {
        switch (jToken.Type)
        {
            case JTokenType.Object:
                foreach (var child in jToken.Children<JProperty>())
                    CollectFields(child);
                break;
            case JTokenType.Array:
                foreach (var child in jToken.Children())
                    CollectFields(child);
                break;
            case JTokenType.Property:
                CollectFields(((JProperty) jToken).Value);
                break;
            default:
                fields.Add(jToken.Path, (JValue)jToken);
                break;
        }
    }

    public IEnumerable<KeyValuePair<string, JValue>> GetAllFields() => fields;
}

```

Usage:

```
var json = JToken.Parse(/* JSON string */);
var fieldsCollector = new JsonFieldsCollector(json);
```

```
var fields = fieldsCollector.GetAllFields();

foreach (var field in fields)
Console.WriteLine($"{field.Key}: '{field.Value}'");
```

演示

对于此 JSON 对象

```
{
    "用户": "约翰",
    "工作日": {
        "星期一": 是,
        "星期二": 是,
        "星期五": 否
    },
    "年龄": 42
}
```

预期输出将是：

```
用户: '约翰'
工作日.星期一: '真'
工作日.星期二: '真'
工作日.星期五: '假'
年龄: '42'
```

```
var fields = fieldsCollector.GetAllFields();
```

```
foreach (var field in fields)
Console.WriteLine($"{field.Key}: '{field.Value}'");
```

Demo

For this JSON object

```
{
    "User": "John",
    "Workdays": {
        "Monday": true,
        "Tuesday": true,
        "Friday": false
    },
    "Age": 42
}
```

expected output will be:

```
User: 'John'
Workdays.Monday: 'True'
Workdays.Tuesday: 'True'
Workdays.Friday: 'False'
Age: '42'
```

第84章：Lambda表达式

第84.1节：Lambda表达式作为委托初始化的简写

```
public delegate int ModifyInt(int input);
ModifyInt multiplyByTwo = x => x * 2;
```

上述Lambda表达式语法等同于以下冗长代码：

```
public delegate int ModifyInt(int input);

ModifyInt multiplyByTwo = delegate(int x){
    return x * 2;
};
```

第84.2节：作为事件处理程序的Lambda表达式

Lambda表达式可以用来处理事件，这在以下情况下非常有用：

- 处理程序很短。
- 处理程序永远不需要取消订阅。

下面给出了一个适合使用Lambda事件处理程序的示例：

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
```

如果在代码的某个后续点需要取消订阅已注册的事件处理程序，事件处理程序表达式应保存到变量中，并通过该变量进行注册/注销操作：

```
EventHandler handler = (sender, args) => Console.WriteLine("Email sent");

smtpClient.SendCompleted += handler;
smtpClient.SendCompleted -= handler;
```

这样做的原因，而不是简单地逐字重写Lambda表达式来取消订阅`(-=)`，是因为C#编译器不一定会认为这两个表达式是相等的：

```
EventHandler handlerA = (sender, args) => Console.WriteLine("Email sent");
EventHandler handlerB = (sender, args) => Console.WriteLine("Email sent");
Console.WriteLine(handlerA.Equals(handlerB)); // 可能返回"False"
```

请注意，如果在lambda表达式中添加了额外的语句，可能会不小心遗漏所需的花括号，而不会导致编译时错误。例如：

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
emailSendButton.Enabled = true;
```

这将能够编译，但结果是将lambda表达式`(sender, args) => Console.WriteLine("Email sent")`作为事件处理程序添加，并立即执行语句`emailSendButton.Enabled = true;`。要修复此问题，lambda的内容必须用花括号括起来。可以通过从一开始就使用花括号、在向lambda事件处理程序添加额外语句时保持谨慎，或者从一开始用圆括号包围lambda来避免此问题：

Chapter 84: Lambda expressions

Section 84.1: Lambda Expressions as Shorthand for Delegate Initialization

```
public delegate int ModifyInt(int input);
ModifyInt multiplyByTwo = x => x * 2;
```

The above Lambda expression syntax is equivalent to the following verbose code:

```
public delegate int ModifyInt(int input);

ModifyInt multiplyByTwo = delegate(int x){
    return x * 2;
};
```

Section 84.2: Lambda Expression as an Event Handler

Lambda expressions can be used to handle events, which is useful when:

- The handler is short.
- The handler never needs to be unsubscribed.

A good situation in which a lambda event handler might be used is given below:

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
```

If unsubscribing a registered event handler at some future point in the code is necessary, the event handler expression should be saved to a variable, and the registration/unregistration done through that variable:

```
EventHandler handler = (sender, args) => Console.WriteLine("Email sent");

smtpClient.SendCompleted += handler;
smtpClient.SendCompleted -= handler;
```

The reason that this is done rather than simply retyping the lambda expression verbatim to unsubscribe it`(-=)` is that the C# compiler won't necessarily consider the two expressions equal:

```
EventHandler handlerA = (sender, args) => Console.WriteLine("Email sent");
EventHandler handlerB = (sender, args) => Console.WriteLine("Email sent");
Console.WriteLine(handlerA.Equals(handlerB)); // May return "False"
```

Note that if additional statements are added to the lambda expression, then the required surrounding curly braces may be accidentally omitted, without causing compile-time error. For example:

```
smtpClient.SendCompleted += (sender, args) => Console.WriteLine("Email sent");
emailSendButton.Enabled = true;
```

This will compile, but will result in adding the lambda expression`(sender, args) => Console.WriteLine("Email sent")` as an event handler, and executing the statement`emailSendButton.Enabled = true;` immediately. To fix this, the contents of the lambda must be surrounded in curly braces. This can be avoided by using curly braces from the start, being cautious when adding additional statements to a lambda-event-handler, or surrounding the lambda in round brackets from the start:

```
smtpClient.SendCompleted += ((sender, args) => Console.WriteLine("Email sent"));
//添加额外语句将导致编译时错误
```

第 84.3 节：带多个参数或无参数的 Lambda 表达式

使用圆括号包围 => 运算符左侧的表达式以表示多个参数。

```
delegate int ModifyInt(int input1, int input2);
ModifyInt multiplyTwoInts = (x,y) => x * y;
```

同样，一对空的圆括号表示该函数不接受参数。

```
delegate string ReturnString();
ReturnString getGreeting = () => "Hello world. ";
```

第84.4节：Lambda表达式既可以作为`Func`也可以作为`Expression`来生成

假设有以下Person类：

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

以下Lambda表达式：

```
p => p.Age > 18
```

可以作为参数传递给以下两个方法：

```
public void AsFunc(Func<Person, bool> func)
public void AsExpression(Expression<Func<Person, bool>> expr)
```

因为编译器能够将Lambda表达式转换为委托和Expression两种形式。

显然，LINQ 提供程序在很大程度上依赖于Expression（主要通过IQueryable<T>接口暴露）以便能够解析查询并将其转换为存储查询。

第84.5节：在语句Lambda中放置多个语句

与表达式Lambda不同，语句Lambda可以包含由分号分隔的多个语句。

```
delegate void ModifyInt(int input);

ModifyInt addOneAndTellMe = x =>
{
    int result = x + 1;
    Console.WriteLine(result);
};
```

注意这些语句被大括号{}包围。

```
smtpClient.SendCompleted += ((sender, args) => Console.WriteLine("Email sent"));
//Adding an extra statement will result in a compile-time error
```

Section 84.3: Lambda Expressions with Multiple Parameters or No Parameters

Use parentheses around the expression to the left of the => operator to indicate multiple parameters.

```
delegate int ModifyInt(int input1, int input2);
ModifyInt multiplyTwoInts = (x,y) => x * y;
```

Similarly, an empty set of parentheses indicates that the function does not accept parameters.

```
delegate string ReturnString();
ReturnString getGreeting = () => "Hello world. ";
```

Section 84.4: Lambdas can be emitted both as `Func` and `Expression`

Assuming the following Person class:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

The following lambda:

```
p => p.Age > 18
```

Can be passed as an argument to both methods:

```
public void AsFunc(Func<Person, bool> func)
public void AsExpression(Expression<Func<Person, bool>> expr)
```

Because the compiler is capable of transforming lambdas both to delegates and Expressions.

Obviously, LINQ providers rely heavily on Expressions (exposed mainly through the IQueryable<T> interface) in order to be able to parse queries and translate them to store queries.

Section 84.5: Put Multiple Statements in a Statement Lambda

Unlike an expression lambda, a statement lambda can contain multiple statements separated by semicolons.

```
delegate void ModifyInt(int input);

ModifyInt addOneAndTellMe = x =>
{
    int result = x + 1;
    Console.WriteLine(result);
};
```

Note that the statements are enclosed in braces {}.

请记住，语句Lambda不能用于创建表达式树。

第84.6节：用于`Func`和`Action`的Lambda

通常Lambda用于定义简单的函数（通常在linq表达式的上下文中）：

```
var incremented = myEnumerable.Select(x => x + 1);
```

这里return是隐式的。

但是，也可以将actions作为lambda传递：

```
myObservable.Do(x => Console.WriteLine(x));
```

第84.7节：使用lambda语法创建闭包

有关闭包的讨论请参见备注。假设我们有一个接口：

```
public interface IMachine<TState, TInput>
{
    TState State { get; }
    public void Input(TInput input);
}
```

然后执行以下代码：

```
IMachine<int, int> machine = ...;
Func<int, int> machineClosure = i => {
    machine.Input(i);
    return machine.State;
};
```

现在，`machineClosure` 指的是一个从 `int` 到 `int` 的函数，该函数在幕后使用 `IMachine` 实例，该实例由 `machine` 引用以执行计算。即使引用 `machine` 超出作用域，只要保持 `machineClosure` 对象，原始的 `IMachine` 实例将作为“闭包”的一部分被保留，该闭包由编译器自动定义。

警告：这可能意味着同一个函数调用在不同时间返回不同的值（例如，在此示例中，如果机器保持其输入的总和）。在许多情况下，这可能是意料之外的，并且应避免在任何函数式风格的代码中出现——意外且出乎意料的闭包可能成为错误的来源。

第84.8节：将Lambda表达式作为参数传递给方法

```
List<int> l2 = l1.FindAll(x => x > 6);
```

这里 `x => x > 6` 是一个作为谓词的lambda表达式，确保只返回大于6的元素。

第84.9节：基本的lambda表达式

```
Func<int, int> add1 = i => i + 1;
```

```
Func<int, int, int> add = (i, j) => i + j;
```

Remember that statement lambdas cannot be used to create expression trees.

Section 84.6: Lambdas for both `Func` and `Action`

Typically lambdas are used for defining simple *functions* (generally in the context of a linq expression):

```
var incremented = myEnumerable.Select(x => x + 1);
```

Here the `return` is implicit.

However, it is also possible to pass *actions* as lambdas:

```
myObservable.Do(x => Console.WriteLine(x));
```

Section 84.7: Using lambda syntax to create a closure

See remarks for discussion of closures. Suppose we have an interface:

```
public interface IMachine<TState, TInput>
{
    TState State { get; }
    public void Input(TInput input);
}
```

and then the following is executed:

```
IMachine<int, int> machine = ...;
Func<int, int> machineClosure = i => {
    machine.Input(i);
    return machine.State;
};
```

Now `machineClosure` refers to a function from `int` to `int`, which behind the scenes uses the `IMachine` instance which `machine` refers to in order to carry out the computation. Even if the reference `machine` goes out of scope, as long as the `machineClosure` object is maintained, the original `IMachine` instance will be retained as part of a 'closure', automatically defined by the compiler.

Warning: this can mean that the same function call returns different values at different times (e.g. In this example if the machine keeps a sum of its inputs). In lots of cases, this may be unexpected and is to be avoided for any code in a functional style - accidental and unexpected closures can be a source of bugs.

Section 84.8: Passing a Lambda Expression as a Parameter to a Method

```
List<int> l2 = l1.FindAll(x => x > 6);
```

Here `x => x > 6` is a lambda expression acting as a predicate that makes sure that only elements above 6 are returned.

Section 84.9: Basic lambda expressions

```
Func<int, int> add1 = i => i + 1;
```

```
Func<int, int, int> add = (i, j) => i + j;
```

// 行为上等同于：

```
int Add1(int i)
{
    return i + 1;
}

int Add(int i, int j)
{
    return i + j;
}

...
Console.WriteLine(add1(42)); //43
Console.WriteLine(Add1(42)); //43
Console.WriteLine(add(100, 250)); //350
Console.WriteLine(Add(100, 250)); //350
```

第84.10节：使用LINQ的基本lambda表达式

```
// 假设source是{0, 1, 2, ..., 10}

var evens = source.Where(n => n%2 == 0);
// evens = {0, 2, 4, ... 10}

var strings = source.Select(n => n.ToString());
// strings = {"0", "1", ..., "10"}
```

第84.11节：带语句块主体的lambda语法

```
Func<int, string> doubleThenAddElevenThenQuote = i => {
    var doubled = 2 * i;
    var addedEleven = 11 + doubled;
    return $"'{addedEleven}'";
};
```

第84.12节：使用 System.Linq.Expressions的Lambda表达式

```
Expression<Func<int, bool>> checkEvenExpression = i => i%2 == 0;
// lambda表达式会自动转换为Expression<Func<int, bool>>
```

// Behaviourally equivalent to:

```
int Add1(int i)
{
    return i + 1;
}

int Add(int i, int j)
{
    return i + j;
}

...
Console.WriteLine(add1(42)); //43
Console.WriteLine(Add1(42)); //43
Console.WriteLine(add(100, 250)); //350
Console.WriteLine(Add(100, 250)); //350
```

Section 84.10: Basic lambda expressions with LINQ

```
// assume source is {0, 1, 2, ..., 10}

var evens = source.Where(n => n%2 == 0);
// evens = {0, 2, 4, ... 10}

var strings = source.Select(n => n.ToString());
// strings = {"0", "1", ..., "10"}
```

Section 84.11: Lambda syntax with statement block body

```
Func<int, string> doubleThenAddElevenThenQuote = i => {
    var doubled = 2 * i;
    var addedEleven = 11 + doubled;
    return $"'{addedEleven}'";
};
```

Section 84.12: Lambda expressions with System.Linq.Expressions

```
Expression<Func<int, bool>> checkEvenExpression = i => i%2 == 0;
// lambda expression is automatically converted to an Expression<Func<int, bool>>
```

第85章：通用Lambda查询构建器

第85.1节：QueryFilter类

该类保存谓词过滤器的值。

```
public class QueryFilter
{
    public string PropertyName { get; set; }
    public string Value { get; set; }
    public Operator Operator { get; set; }

    // 在查询中 {a => a.Name.Equals("Pedro")}
    // 用于过滤的属性名 - propertyName = "Name"
    // 过滤值 - value = "Pedro"
    // 执行的操作 - operation = enum Operator.Equals
    public QueryFilter(string propertyName, string value, Operator operatorValue)
    {
        PropertyName = propertyName;
        Value = value;
        Operator = operatorValue;
    }
}
```

用于保存操作值的枚举：

```
public enum Operator
{
    Contains,
    GreaterThan,
    GreaterThanOrEqual,
    LessThan,
    LessThanOrEqualTo,
    StartsWith,
    以...结尾,
    等于,
    不等于
}
```

第85.2节：GetExpression方法

```
public static Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)
{
    Expression exp = null;

    // 表示一个命名参数表达式。{parm => parm.Name.Equals()}, 其中parm是参数部分// 创建ParameterExpression需要
    // 查询所针对实体的类型和一个名称

    // 类型可以通过泛型T获得，名称固定为parm
    ParameterExpression param = Expression.Parameter(typeof(T), "parm");

    // 最好不要完全信任客户端，因此验证是明智的做法。
    if (filters.Count == 0)
        return null;

    // 表达式的创建会根据过滤器数量是一个、两个还是多个而有所不同。
    if (filters.Count != 1)
    {
```

Chapter 85: Generic Lambda Query Builder

Section 85.1: QueryFilter class

This class holds predicate filters values.

```
public class QueryFilter
{
    public string PropertyName { get; set; }
    public string Value { get; set; }
    public Operator Operator { get; set; }

    // In the query {a => a.Name.Equals("Pedro")}
    // Property name to filter - propertyName = "Name"
    // Filter value - value = "Pedro"
    // Operation to perform - operation = enum Operator.Equals
    public QueryFilter(string propertyName, string value, Operator operatorValue)
    {
        PropertyName = propertyName;
        Value = value;
        Operator = operatorValue;
    }
}
```

Enum to hold the operations values:

```
public enum Operator
{
    Contains,
    GreaterThan,
    GreaterThanOrEqual,
    LessThan,
    LessThanOrEqualTo,
    StartsWith,
    EndsWith,
    Equals,
    NotEqual
}
```

Section 85.2: GetExpression Method

```
public static Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)
{
    Expression exp = null;

    // Represents a named parameter expression. {parm => parm.Name.Equals()}, it is the param part
    // To create a ParameterExpression need the type of the entity that the query is against an a
    // name
    // The type is possible to find with the generic T and the name is fixed param
    ParameterExpression param = Expression.Parameter(typeof(T), "parm");

    // It is good practice never trust in the client, so it is wise to validate.
    if (filters.Count == 0)
        return null;

    // The expression creation differ if there is one, two or more filters.
    if (filters.Count != 1)
    {
```

```

if (filters.Count == 2)
    // 这是直接调用的结果。
    // 为了简化，私有重载将在另一个示例中说明。
exp = GetExpression<T>(param, filters[0], filters[1]);
else
{
    // 由于没有超过两个过滤器的方法,
    // 我遍历所有过滤器，每次将两个放入查询中
    while (filters.Count > 0)
    {
        // 获取前两个过滤器
        var f1 = filters[0];
        var f2 = filters[1];

        // 构建一个带有条件AND操作的表达式，只有当第一个操作数为真时才计算第二个操作数。
        // 需要使用继承自 Expression 的 BinaryExpression 类
        // 该类具有将两个表达式连接在一起的 AndAlso 方法
exp = exp == null ? GetExpression<T>(param, filters[0], filters[1]) :
Expression.AndAlso(exp, GetExpression<T>(param, filters[0], filters[1]));

        // 移除刚使用的两个过滤器，以便下一次迭代的方法找到
        // 下一个过滤器
filters.Remove(f1);
filters.Remove(f2);

        // 如果是最后一个过滤器，添加最后一个并移除它
        if (filters.Count == 1)
        {
exp = Expression.AndAlso(exp, GetExpression<T>(param, filters[0]));
            filters.RemoveAt(0);
        }
    }
}
else
    // 这是直接调用的结果。
exp = GetExpression<T>(param, filters[0]);

    // 将表达式转换为Lambda并返回查询
return Expression.Lambda<Func<T, bool>>(exp, param);
}

```

```

if (filters.Count == 2)
    // It is result from direct call.
    // For simplicity sake the private overloads will be explained in another example.
exp = GetExpression<T>(param, filters[0], filters[1]);
else
{
    // As there is no method for more than two filters,
    // I iterate through all the filters and put 2 in the query two at a time
    while (filters.Count > 0)
    {
        // Retrieve the first two filters
        var f1 = filters[0];
        var f2 = filters[1];

        // To build a expression with a conditional AND operation that evaluates
        // the second operand only if the first operand evaluates to true.
        // It needed to use the BinaryExpression a Expression derived class
        // That has the AndAlso method that join two expression together
exp = exp == null ? GetExpression<T>(param, filters[0], filters[1]) :
Expression.AndAlso(exp, GetExpression<T>(param, filters[0], filters[1]));

        // Remove the two just used filters, for the method in the next iteration finds the
        // next filters
filters.Remove(f1);
filters.Remove(f2);

        // If it is that last filter, add the last one and remove it
        if (filters.Count == 1)
        {
            exp = Expression.AndAlso(exp, GetExpression<T>(param, filters[0]));
            filters.RemoveAt(0);
        }
    }
}
else
    // It is result from direct call.
exp = GetExpression<T>(param, filters[0]);

    // converts the Expression into Lambda and retuns the query
return Expression.Lambda<Func<T, bool>>(exp, param);
}

```

第85.3节：GetExpression 私有重载

对于一个过滤器：

这里是创建查询的地方，它接收一个表达式参数和一个过滤器。

```

private static Expression GetExpression<T>(ParameterExpression param, QueryFilter queryFilter)
{
    // 表示访问字段或属性，所以这里我们访问例如：
    // 实体的"Name"属性
MemberExpression member = Expression.Property(param, queryFilter.PropertyName);

    // 表示具有常量值的表达式，所以这里我们访问例如：
    // "Name"属性的值。
    // 为了清晰起见，GetConstant将在另一个示例中解释。
ConstantExpression constant = GetConstant(member.Type, queryFilter.Value);

    // 有了这两个，现在我可以构建表达式了
}

```

Section 85.3: GetExpression Private overload

For one filter:

Here is where the query is created, it receives a expression parameter and a filter.

```

private static Expression GetExpression<T>(ParameterExpression param, QueryFilter queryFilter)
{
    // Represents accessing a field or property, so here we are accessing for example:
    // the property "Name" of the entity
MemberExpression member = Expression.Property(param, queryFilter.PropertyName);

    //Represents an expression that has a constant value, so here we are accessing for example:
    // the values of the Property "Name".
    // Also for clarity sake the GetConstant will be explained in another example.
ConstantExpression constant = GetConstant(member.Type, queryFilter.Value);

    // With these two, now I can build the expression
}

```

```
// 每个操作符都有一种调用方式，所以用switch就行了。
switch (queryFilter.Operator)
{
    case Operator.Equals:
        return Expression.Equal(member, constant);

    case Operator.Contains:
        return Expression.Call(member, ContainsMethod, constant);

    case Operator.GreaterThan:
        return Expression.GreaterThan(member, constant);

    case Operator.GreaterThanOrEqual:
        return Expression.GreaterThanOrEqual(member, constant);

    case Operator.LessThan:
        return Expression.LessThan(member, constant);

    case Operator.LessThanOrEqual:
        return Expression.LessThanOrEqual(member, constant);

    case Operator.StartsWith:
        return Expression.Call(member, StartsWithMethod, constant);

    case Operator.EndsWith:
        return Expression.Call(member, EndsWithMethod, constant);
}

return null;
}
```

对于两个过滤器：

它返回的是 BinaryExpression 实例，而不是简单的 Expression。

```
private static BinaryExpression GetExpression<T>(ParameterExpression param, QueryFilter filter1,
QueryFilter filter2)
{
    // 构建两个独立的表达式，然后将它们连接起来。
    Expression result1 = GetExpression<T>(param, filter1);
    Expression result2 = GetExpression<T>(param, filter2);
    return Expression.AndAlso(result1, result2);
}
```

第85.4节：ConstantExpression 方法

ConstantExpression 必须与 MemberExpression 类型相同。此示例中的值是字符串，在创建 ConstantExpression 实例之前进行了转换。

```
private static ConstantExpression GetConstant(Type type, string value)
{
    // 识别类型，进行转换，并创建 ConstantExpression
    ConstantExpression constant = null;
    if (type == typeof(int))
    {
        int num;
        int.TryParse(value, out num);
        constant = Expression.Constant(num);
    }
    else if (type == typeof(string))
    {
```

```
// every operator has it one way to call, so the switch will do.
switch (queryFilter.Operator)
{
    case Operator.Equals:
        return Expression.Equal(member, constant);

    case Operator.Contains:
        return Expression.Call(member, ContainsMethod, constant);

    case Operator.GreaterThan:
        return Expression.GreaterThan(member, constant);

    case Operator.GreaterThanOrEqual:
        return Expression.GreaterThanOrEqual(member, constant);

    case Operator.LessThan:
        return Expression.LessThan(member, constant);

    case Operator.LessThanOrEqual:
        return Expression.LessThanOrEqual(member, constant);

    case Operator.StartsWith:
        return Expression.Call(member, StartsWithMethod, constant);

    case Operator.EndsWith:
        return Expression.Call(member, EndsWithMethod, constant);
}

return null;
}
```

For two filters:

It returns the BinaryExpression instance instead of the simple Expression.

```
private static BinaryExpression GetExpression<T>(ParameterExpression param, QueryFilter filter1,
QueryFilter filter2)
{
    // Built two separated expression and join them after.
    Expression result1 = GetExpression<T>(param, filter1);
    Expression result2 = GetExpression<T>(param, filter2);
    return Expression.AndAlso(result1, result2);
}
```

Section 85.4: ConstantExpression Method

ConstantExpression must be the same type of the MemberExpression. The value in this example is a string, which is converted before creating the ConstantExpression instance.

```
private static ConstantExpression GetConstant(Type type, string value)
{
    // Discover the type, convert it, and create ConstantExpression
    ConstantExpression constant = null;
    if (type == typeof(int))
    {
        int num;
        int.TryParse(value, out num);
        constant = Expression.Constant(num);
    }
    else if (type == typeof(string))
```

```

constant = Expression.Constant(value);
}
else if (type == typeof(DateTime))
{
DateTime date;
DateTime.TryParse(value, out date);
constant = Expression.Constant(date);
}
else if (type == typeof(bool))
{
    bool flag;
    if (bool.TryParse(value, out flag))
    {
flag = true;
    }
constant = Expression.Constant(flag);
}
else if (type == typeof(decimal))
{
    decimal number;
    decimal.TryParse(value, out number);
    constant = Expression.Constant(number);
}
return constant;
}

```

第85.5节：用法

```

Collection filters = new List(); QueryFilter filter = new QueryFilter("Name", "Burger", Operator.StartsWith);
filters.Add(filter);

```

Expression<Func<Food, bool>> query = ExpressionBuilder.GetExpression<Food>(filters);

在这种情况下，这是针对Food实体的查询，目的是查找名称以“Burger”开头的所有食品。

输出：

```

query = {parm => a.parm.StartsWith("Burger")}

```

Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)

```

constant = Expression.Constant(value);
}
else if (type == typeof(DateTime))
{
    DateTime date;
    DateTime.TryParse(value, out date);
    constant = Expression.Constant(date);
}
else if (type == typeof(bool))
{
    bool flag;
    if (bool.TryParse(value, out flag))
    {
        flag = true;
    }
    constant = Expression.Constant(flag);
}
else if (type == typeof(decimal))
{
    decimal number;
    decimal.TryParse(value, out number);
    constant = Expression.Constant(number);
}
return constant;
}

```

Section 85.5: Usage

```

Collection filters = new List(); QueryFilter filter = new QueryFilter("Name", "Burger", Operator.StartsWith);
filters.Add(filter);

```

Expression<Func<Food, bool>> query = ExpressionBuilder.GetExpression<Food>(filters);

In this case, it is a query against the Food entity, that want to find all foods that start with "Burger" in the name.

Output:

```

query = {parm => a.parm.StartsWith("Burger")}

```

Expression<Func<T, bool>> GetExpression<T>(IList<QueryFilter> filters)

第86章：属性

第86.1节：自动实现属性

自动实现属性是在C# 3中引入的。

自动实现属性是用空的getter和setter（访问器）声明的：

```
public bool IsValid { get; set; }
```

当代码中写入自动实现属性时，编译器会创建一个私有的匿名字段，该字段只能通过属性的访问器访问。

上述自动实现属性语句等同于编写以下冗长代码：

```
private bool _isValid;
public bool IsValid
{
    get { return _isValid; }
    set { _isValid = value; }
}
```

自动实现的属性在其访问器中不能包含任何逻辑，例如：

```
public bool IsValid { get; set { PropertyChanged("IsValid"); } } // 无效代码
```

但是，自动实现的属性可以为其访问器设置不同的访问修饰符：

```
public bool IsValid { get; private set; }
```

C# 6 允许自动实现的属性完全没有 setter（使其不可变，因为其值只能在构造函数内部设置或硬编码）：

```
public bool IsValid { get; }
public bool IsValid { get; } = true;
```

有关初始化自动实现属性的更多信息，请参阅自动属性初始化器文档。

第86.2节：属性的默认值

可以通过使用初始化器（C#6）来设置默认值

```
public class Name
{
    public string First { get; set; } = "James";
    public string Last { get; set; } = "Smith";
}
```

如果是只读的，你可以这样返回值：

```
public class Name
{
    public string First => "James";
    public string Last => "Smith";
```

Chapter 86: Properties

Section 86.1: Auto-implemented properties

Auto-implemented properties were introduced in C# 3.

An auto-implemented property is declared with an empty getter and setter (accessors):

```
public bool IsValid { get; set; }
```

When an auto-implemented property is written in your code, the compiler creates a private anonymous field that can only be accessed through the property's accessors.

The above auto-implemented property statement is equivalent to writing this lengthy code:

```
private bool _isValid;
public bool IsValid
{
    get { return _isValid; }
    set { _isValid = value; }
}
```

Auto-implemented properties cannot have any logic in their accessors, for example:

```
public bool IsValid { get; set { PropertyChanged("IsValid"); } } // Invalid code
```

An auto-implemented property *can* however have different access modifiers for its accessors:

```
public bool IsValid { get; private set; }
```

C# 6 allows auto-implemented properties to have no setter at all (making it immutable, since its value can be set only inside the constructor or hard coded):

```
public bool IsValid { get; }
public bool IsValid { get; } = true;
```

For more information on initializing auto-implemented properties, read the Auto-property initializers documentation.

Section 86.2: Default Values for Properties

Setting a default value can be done by using Initializers (C#6)

```
public class Name
{
    public string First { get; set; } = "James";
    public string Last { get; set; } = "Smith";
}
```

If it is read only you can return values like this:

```
public class Name
{
    public string First => "James";
    public string Last => "Smith";
```

}

第86.3节：公共获取

Getter用于从类中公开值。

```
string name;
public string Name
{
    get { return this.name; }
}
```

第86.4节：公共设置

Setter 用于为属性赋值。

```
string name;
public string Name
{
    set { this.name = value; }
}
```

第86.5节：访问属性

```
class Program
{
    public static void Main(string[] args)
    {
        Person aPerson = new Person("Ann Xena Sample", new DateTime(1984, 10, 22));
        //访问属性 (Id, Name 和 DOB) 的示例
        Console.WriteLine("Id 是: {0}Name 是: '{1}'。DOB 是: {2:yyyy-MM-dd}。Age 是:
{3}", aPerson.Id, aPerson.Name, aPerson.DOB, aPerson.GetAgeInYears());
        //设置属性的示例

        aPerson.Name = " Hans Trimmer ";
        aPerson.DOB = new DateTime(1961, 11, 11);
        //aPerson.Id = 5; //这不会编译，因为 Id 的 SET 方法是私有的；因此只能在 Person 类内部访问。
        //aPerson.DOB = DateTime.UtcNow.AddYears(1); //这会抛出运行时错误，因为有验证确保 DOB 必须是过去的日期。

        //查看我们上述更改的效果；注意名称已被修剪
        Console.WriteLine("Id 是: {0}Name 是: '{1}'。DOB 是: {2:yyyy-MM-dd}。Age 是:
{3}", aPerson.Id, aPerson.Name, aPerson.DOB, aPerson.GetAgeInYears());

        Console.WriteLine("按任意键继续");
        Console.Read();
    }

    public class Person
    {
        private static int nextId = 0;
        private string name;
        private DateTime dob; //日期以UTC保存；即我们忽略时区
        public Person(string name, DateTime dob)
        {
            this.Id = ++Person.nextId;
            this.Name = name;
        }
    }
}
```

}

Section 86.3: Public Get

Getters are used to expose values from classes.

```
string name;
public string Name
{
    get { return this.name; }
}
```

Section 86.4: Public Set

Setters are used to assign values to properties.

```
string name;
public string Name
{
    set { this.name = value; }
}
```

Section 86.5: Accessing Properties

```
class Program
{
    public static void Main(string[] args)
    {
        Person aPerson = new Person("Ann Xena Sample", new DateTime(1984, 10, 22));
        //example of accessing properties (Id, Name & DOB)
        Console.WriteLine("Id is: \t{0}\nName is:\t'{1}'.\nDOB is: \t{2:yyyy-MM-dd}.\nAge is:
\t{3}", aPerson.Id, aPerson.Name, aPerson.DOB, aPerson.GetAgeInYears());
        //example of setting properties

        aPerson.Name = " Hans Trimmer ";
        aPerson.DOB = new DateTime(1961, 11, 11);
        //aPerson.Id = 5; //this won't compile as Id's SET method is private; so only accessible
        //within the Person class.
        //aPerson.DOB = DateTime.UtcNow.AddYears(1); //this would throw a runtime error as there's
        //validation to ensure the DOB is in past.

        //see how our changes above take effect; note that the Name has been trimmed
        Console.WriteLine("Id is: \t{0}\nName is:\t'{1}'.\nDOB is: \t{2:yyyy-MM-dd}.\nAge is:
\t{3}", aPerson.Id, aPerson.Name, aPerson.DOB, aPerson.GetAgeInYears());

        Console.WriteLine("Press any key to continue");
        Console.Read();
    }

    public class Person
    {
        private static int nextId = 0;
        private string name;
        private DateTime dob; //dates are held in UTC; i.e. we disregard timezones
        public Person(string name, DateTime dob)
        {
            this.Id = ++Person.nextId;
            this.Name = name;
        }
    }
}
```

```

        this.DOB = dob;
    }
    public int Id
    {
        get;
        private set;
    }
    public string Name
    {
        get { return this.name; }
        set
        {
            if (string.IsNullOrWhiteSpace(value)) throw new InvalidNameException(value);
            this.name = value.Trim();
        }
    }
    public DateTime 出生日期
    {
        get { return this.dob; }
        set
        {
            if (value < DateTime.UtcNow.AddYears(-200) || value > DateTime.UtcNow) throw new
InvalidDobException(value);
            this.dob = value;
        }
    }
    public int 获取年龄(年)()
    {
        DateTime 今天 = DateTime.UtcNow;
        int 偏移量 = 是否已过今年生日() ? 0 : -1;
        return 今天.Year - this.dob.Year + 偏移量;
    }
    private bool 是否已过今年生日()
    {
        bool 已过今年生日 = true;
        DateTime 今天 = DateTime.UtcNow;
        if (今天.Month > this.dob.Month)
        {
            hasHadBirthdayThisYear = true;
        }
        else
        {
            if (today.Month == this.dob.Month)
            {
                hasHadBirthdayThisYear = today.Day > this.dob.Day;
            }
            else
            {
                hasHadBirthdayThisYear = false;
            }
        }
        return hasHadBirthdayThisYear;
    }
}

public class InvalidNameException : ApplicationException
{
    const string InvalidNameExceptionMessage = "{0}' 是一个无效的名字。";
    public InvalidNameException(string value):
base(string.Format(InvalidNameExceptionMessage, value)){}
}
public class InvalidDobException : ApplicationException

```

```

        this.DOB = dob;
    }
    public int Id
    {
        get;
        private set;
    }
    public string Name
    {
        get { return this.name; }
        set
        {
            if (string.IsNullOrWhiteSpace(value)) throw new InvalidNameException(value);
            this.name = value.Trim();
        }
    }
    public DateTime DOB
    {
        get { return this.dob; }
        set
        {
            if (value < DateTime.UtcNow.AddYears(-200) || value > DateTime.UtcNow) throw new
InvalidDobException(value);
            this.dob = value;
        }
    }
    public int GetAgeInYears()
    {
        DateTime today = DateTime.UtcNow;
        int offset = HasHadBirthdayThisYear() ? 0 : -1;
        return today.Year - this.dob.Year + offset;
    }
    private bool HasHadBirthdayThisYear()
    {
        bool hasHadBirthdayThisYear = true;
        DateTime today = DateTime.UtcNow;
        if (today.Month > this.dob.Month)
        {
            hasHadBirthdayThisYear = true;
        }
        else
        {
            if (today.Month == this.dob.Month)
            {
                hasHadBirthdayThisYear = today.Day > this.dob.Day;
            }
            else
            {
                hasHadBirthdayThisYear = false;
            }
        }
        return hasHadBirthdayThisYear;
    }
}

public class InvalidNameException : ApplicationException
{
    const string InvalidNameExceptionMessage = "'{0}' is an invalid name.";
    public InvalidNameException(string value):
base(string.Format(InvalidNameExceptionMessage, value)){}
}
public class InvalidDobException : ApplicationException

```

```

{
    const string InvalidDobExceptionMessage = "{0:yyyy-MM-dd}' 是无效的出生日期。日期不能
是未来的时间，也不能超过200年前。";
    public InvalidDobException(DateTime value):
base(string.Format(InvalidDobExceptionMessage,value)){}
}

```

第86.6节：只读属性

声明

一个常见的误解，尤其是初学者，认为只读属性是用 `readonly` 关键字标记的属性。那是不正确的，实际上以下是编译时错误：

```
public readonly string SomeProp { get; set; }
```

当属性只有getter时，该属性是只读的。

```
public string SomeProp { get; }
```

使用只读属性创建不可变类

```

public Address
{
    public string ZipCode { get; }
    public string City { get; }
    public string StreetAddress { get; }

    public Address(
        string 邮政编码,
        string 城市,
        string 街道地址)
    {
        if (邮政编码 == null)
            throw new ArgumentNullException(nameof(邮政编码));
        if (城市 == null)
            throw new ArgumentNullException(nameof(城市));
        if (街道地址 == null)
            throw new ArgumentNullException(nameof(街道地址));

        邮政编码 = 邮政编码;
        城市 = 城市;
        街道地址 = 街道地址;
    }
}

```

第86.7节：上下文中的各种属性

```

public class Person
{
    //Id属性可以被其他类读取，但只能由Person类设置
    public int Id {get; private set;}
    //名称属性可以被获取或赋值
    public string Name {get; set;}

    private DateTime dob;
    //出生日期属性存储在私有变量中，但通过公共属性进行获取或赋值。
    public DateTime 出生日期
    {

```

```

{
    const string InvalidDobExceptionMessage = "'{0:yyyy-MM-dd}' is an invalid DOB. The date must
not be in the future, or over 200 years in the past.";
    public InvalidDobException(DateTime value):
base(string.Format(InvalidDobExceptionMessage,value)){}
}

```

Section 86.6: Read-only properties

Declaration

A common misunderstanding, especially beginners, have is read-only property is the one marked with `readonly` keyword. That's not correct and in fact *following is a compile time error:*

```
public readonly string SomeProp { get; set; }
```

A property is read-only when it only has a getter.

```
public string SomeProp { get; }
```

Using read-only properties to create immutable classes

```

public Address
{
    public string ZipCode { get; }
    public string City { get; }
    public string StreetAddress { get; }

    public Address(
        string zipCode,
        string city,
        string streetAddress)
    {
        if (zipCode == null)
            throw new ArgumentNullException(nameof(zipCode));
        if (city == null)
            throw new ArgumentNullException(nameof(city));
        if (streetAddress == null)
            throw new ArgumentNullException(nameof(streetAddress));

        ZipCode = zipCode;
        City = city;
        StreetAddress = streetAddress;
    }
}

```

Section 86.7: Various Properties in Context

```

public class Person
{
    //Id property can be read by other classes, but only set by the Person class
    public int Id {get; private set;}
    //Name property can be retrieved or assigned
    public string Name {get; set;}

    private DateTime dob;
    //Date of Birth property is stored in a private variable, but retrieved or assigned through the
    public DateTime DOB
    {

```

```

        get { return this.dob; }
        set { this.dob = value; }
    }
    //年龄属性只能被获取；其值是根据出生日期计算得出
    public int Age
    {
        get
        {
            int offset = HasHadBirthdayThisYear() ? 0 : -1;
            return DateTime.UtcNow.Year - this.dob.Year + offset;
        }
    }

    //这不是一个属性，而是一个方法；不过如果需要也可以重写成属性。
    private bool 是否已过今年生日()
    {
        bool 已过今年生日 = true;
        DateTime 今天 = DateTime.UtcNow;
        if (今天.Month > this.dob.Month)
        {
            hasHadBirthdayThisYear = true;
        }
        else
        {
            if (today.Month == this.dob.Month)
            {
                hasHadBirthdayThisYear = today.Day > this.dob.Day;
            }
            else
            {
                hasHadBirthdayThisYear = false;
            }
        }
        return hasHadBirthdayThisYear;
    }
}

```

```

        get { return this.dob; }
        set { this.dob = value; }
    }
    //Age property can only be retrieved; it's value is derived from the date of birth
    public int Age
    {
        get
        {
            int offset = HasHadBirthdayThisYear() ? 0 : -1;
            return DateTime.UtcNow.Year - this.dob.Year + offset;
        }
    }

    //this is not a property but a method; though it could be rewritten as a property if desired.
    private bool HasHadBirthdayThisYear()
    {
        bool hasHadBirthdayThisYear = true;
        DateTime today = DateTime.UtcNow;
        if (today.Month > this.dob.Month)
        {
            hasHadBirthdayThisYear = true;
        }
        else
        {
            if (today.Month == this.dob.Month)
            {
                hasHadBirthdayThisYear = today.Day > this.dob.Day;
            }
            else
            {
                hasHadBirthdayThisYear = false;
            }
        }
        return hasHadBirthdayThisYear;
    }
}

```

第87章：初始化属性

第87.1节：C# 6.0：初始化自动实现属性

创建一个带有getter和/or setter的属性，并在一行中完成初始化：

```
public string Foobar { get; set; } = "xyz";
```

第87.2节：使用后备字段初始化属性

```
public string Foobar {
    get { return _foobar; }
    set { _foobar = value; }
}
private string _foobar = "xyz";
```

第87.3节：对象实例化时的属性初始化

对象实例化时可以设置属性。

```
var redCar = new Car
{
    Wheels = 2,
    Year = 2016,
    Color = Color.Red
};
```

第87.4节：在构造函数中初始化属性

```
class Example
{
    public string Foobar { get; set; }
    public List<string> Names { get; set; }
    public Example()
    {
        Foobar = "xyz";
        Names = new List<string>(){ "carrot", "fox", "ball" };
    }
}
```

Chapter 87: Initializing Properties

Section 87.1: C# 6.0: Initialize an Auto-Implemented Property

Create a property with getter and/or setter and initialize all in one line:

```
public string Foobar { get; set; } = "xyz";
```

Section 87.2: Initializing Property with a Backing Field

```
public string Foobar {
    get { return _foobar; }
    set { _foobar = value; }
}
private string _foobar = "xyz";
```

Section 87.3: Property Initialization during object instantiation

Properties can be set when an object is instantiated.

```
var redCar = new Car
{
    Wheels = 2,
    Year = 2016,
    Color = Color.Red
};
```

Section 87.4: Initializing Property in Constructor

```
class Example
{
    public string Foobar { get; set; }
    public List<string> Names { get; set; }
    public Example()
    {
        Foobar = "xyz";
        Names = new List<string>(){ "carrot", "fox", "ball" };
    }
}
```

第88章：INotifyPropertyChanged 接口

第88.1节：在C# 6中实现INotifyPropertyChanged

INotifyPropertyChanged的实现容易出错，因为该接口要求以字符串形式指定属性名称。为了使实现更健壮，可以使用CallerMemberName属性。

```
class C : INotifyPropertyChanged
{
    // backing field
    int offset;
    // 属性
    public int Offset
    {
        get
        {
            return offset;
        }
        set
        {
            if (offset == value)
                return;
            offset = value;
            RaisePropertyChanged();
        }
    }

    // 触发 PropertyChanged 事件的辅助方法
    void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    // 接口实现
    public event PropertyChangedEventHandler PropertyChanged;
}
```

如果你有多个类实现了INotifyPropertyChanged接口，你可能会发现将接口实现和辅助方法重构到公共基类中会很有用：

```
class NotifyPropertyChangedImpl : INotifyPropertyChanged
{
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    // 接口实现
    public event PropertyChangedEventHandler PropertyChanged;
}

class C : NotifyPropertyChangedImpl
{
    int offset;
    public int Offset
    {
        get { return offset; }
        set { if (offset != value) { offset = value; RaisePropertyChanged(); } }
    }
}
```

Chapter 88: INotifyPropertyChanged interface

Section 88.1: Implementing INotifyPropertyChanged in C# 6

The implementation of INotifyPropertyChanged can be error-prone, as the interface requires specifying property name as a string. In order to make the implementation more robust, an attribute CallerMemberName can be used.

```
class C : INotifyPropertyChanged
{
    // backing field
    int offset;
    // property
    public int Offset
    {
        get
        {
            return offset;
        }
        set
        {
            if (offset == value)
                return;
            offset = value;
            RaisePropertyChanged();
        }
    }

    // helper method for raising PropertyChanged event
    void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    // interface implementation
    public event PropertyChangedEventHandler PropertyChanged;
}
```

If you have several classes implementing INotifyPropertyChanged, you may find it useful to refactor out the interface implementation and the helper method to the common base class:

```
class NotifyPropertyChangedImpl : INotifyPropertyChanged
{
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    // interface implementation
    public event PropertyChangedEventHandler PropertyChanged;
}

class C : NotifyPropertyChangedImpl
{
    int offset;
    public int Offset
    {
        get { return offset; }
        set { if (offset != value) { offset = value; RaisePropertyChanged(); } }
    }
}
```

第88.2节：带有泛型Set方法的INotifyPropertyChanged

下面的NotifyPropertyChangedBase类定义了一个泛型Set方法，可以从任何派生类型调用。

```
public class NotifyPropertyChangedBase : INotifyPropertyChanged
{
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    public event PropertyChangedEventHandler PropertyChanged;

    public virtual bool Set<T>(ref T field, T value, [CallerMemberName] string propertyName = null)
    {
        if (Equals(field, value))
            return false;
        storage = value;
        RaisePropertyChanged(propertyName);
        return true;
    }
}
```

要使用这个通用的 Set 方法，你只需创建一个继承自 NotifyPropertyChangedBase 的类。

```
public class SomeViewModel : NotifyPropertyChangedBase
{
    private string _foo;
    private int _bar;

    public string Foo
    {
        get { return _foo; }
        set { Set(ref _foo, value); }
    }

    public int Bar
    {
        get { return _bar; }
        set { Set(ref _bar, value); }
    }
}
```

如上所示，您可以在属性的 setter 中调用 Set(ref _fieldName, value);，如果需要，它会自动触发 PropertyChanged 事件。

然后，您可以从另一个需要处理属性更改的类中注册 PropertyChanged 事件。

```
public class SomeListener
{
    public SomeListener()
    {
        _vm = new SomeViewModel();
        _vm.PropertyChanged += OnViewModelPropertyChanged;
    }

    private void OnViewModelPropertyChanged(object sender, PropertyChangedEventArgs e)
    {
        Console.WriteLine($"属性 {e.PropertyName} 已更改。");
    }
}
```

Section 88.2: INotifyPropertyChanged With Generic Set Method

The NotifyPropertyChangedBase class below defines a generic Set method that can be called from any derived type.

```
public class NotifyPropertyChangedBase : INotifyPropertyChanged
{
    protected void RaisePropertyChanged([CallerMemberName] string propertyName = null) =>
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

    public event PropertyChangedEventHandler PropertyChanged;

    public virtual bool Set<T>(ref T field, T value, [CallerMemberName] string propertyName = null)
    {
        if (Equals(field, value))
            return false;
        storage = value;
        RaisePropertyChanged(propertyName);
        return true;
    }
}
```

To use this generic Set method, you simply need to create a class that derives from NotifyPropertyChangedBase.

```
public class SomeViewModel : NotifyPropertyChangedBase
{
    private string _foo;
    private int _bar;

    public string Foo
    {
        get { return _foo; }
        set { Set(ref _foo, value); }
    }

    public int Bar
    {
        get { return _bar; }
        set { Set(ref _bar, value); }
    }
}
```

As shown above, you can call `Set(ref _fieldName, value)` in a property's setter and it will automatically raise a PropertyChanged event if it is needed.

You can then register to the PropertyChanged event from another class that needs to handle property changes.

```
public class SomeListener
{
    public SomeListener()
    {
        _vm = new SomeViewModel();
        _vm.PropertyChanged += OnViewModelPropertyChanged;
    }

    private void OnViewModelPropertyChanged(object sender, PropertyChangedEventArgs e)
    {
        Console.WriteLine($"Property {e.PropertyName} was changed.");
    }
}
```

```
}

private readonly SomeViewModel _vm;

}
```

```
}

private readonly SomeViewModel _vm;

}
```

belindoc.com

第89章：事件

参数	详情
EventArgsT	从 EventArgs 派生并包含事件参数的类型。
事件名称	事件的名称。
处理程序名称	事件处理程序的名称。
发送者对象	调用事件的对象。
事件参数	包含事件参数的 EventArgsT 类型的实例。

事件是某种事情已经发生（例如鼠标点击）或在某些情况下即将发生（例如价格变动）的通知。

类可以定义事件，其实例（对象）可以触发这些事件。例如，按钮可能包含一个点击事件，当用户点击它时该事件被触发。

事件处理程序是当对应事件被触发时调用的方法。例如，一个窗体可能包含它所包含的每个按钮的点击事件处理程序。

第89.1节：声明和触发事件

声明事件

你可以在任何类或结构体上使用以下语法声明事件：

```
public class MyClass
{
    // 为 MyClass 声明事件
    public event EventHandler MyEvent;

    // 触发 MyEvent 事件
    public void RaiseEvent()
    {
        OnMyEvent();
    }
}
```

声明事件有一种扩展语法，你可以持有事件的私有实例，并使用add和set访问器定义一个公共实例。该语法与C#属性非常相似。在所有情况下，建议优先使用上面演示的语法，因为编译器会生成代码，帮助确保多个线程能够安全地向你的类的事件添加和移除事件处理程序。

触发事件

版本 ≥ 6.0

```
private void OnMyEvent()
{
    EventName?.Invoke(this, EventArgs.Empty);
}
```

版本 < 6.0

```
private void OnMyEvent()
{
    // 使用局部变量eventName，因为另一个线程可能会在我们检查它是否为null和触发事件之间修改// 公共的EventName。
    var eventName = EventName;
}
```

Chapter 89: Events

Parameter	Details
EventArgsT	The type that derives from EventArgs and contains the event parameters.
EventName	The name of the event.
HandlerName	The name of the event handler.
SenderObject	The object that's invoking the event.
EventArgs	An instance of the EventArgsT type that contains the event parameters.

An event is a notification that something has occurred (such as a mouse click) or, in some cases, is about to occur (such as a price change).

Classes can define events and their instances (objects) may raise these events. For instance, a Button may contain a Click event that gets raised when a user has clicked it.

Event handlers are then methods that get called when their corresponding event is raised. A form may contain a Clicked event handler for every Button it contains, for instance.

Section 89.1: Declaring and Raising Events

Declaring an Event

You can declare an event on any `class` or `struct` using the following syntax:

```
public class MyClass
{
    // Declares the event for MyClass
    public event EventHandler MyEvent;

    // Raises the MyEvent event
    public void RaiseEvent()
    {
        OnMyEvent();
    }
}
```

There is an expanded syntax for declaring events, where you hold a private instance of the event, and define a public instance using `add` and `set` accessors. The syntax is very similar to C# properties. In all cases, the syntax demonstrated above should be preferred, because the compiler emits code to help ensure that multiple threads can safely add and remove event handlers to the event on your class.

Raising the Event

Version ≥ 6.0

```
private void OnMyEvent()
{
    EventName?.Invoke(this, EventArgs.Empty);
}
```

Version < 6.0

```
private void OnMyEvent()
{
    // Use a local for EventName, because another thread can modify the
    // public EventName between when we check it for null, and when we
    // raise the event.
    var eventName = EventName;
}
```

```

// 如果eventName == null, 表示没有事件订阅者,
// 因此, 我们无法触发事件。
if(eventName != null)
    eventName(this, EventArgs.Empty);
}

```

请注意, 事件只能由声明该事件的类型触发。客户端只能订阅或取消订阅。

对于 C# 6.0 之前的版本, 由于不支持EventName?.Invoke, 通常的做法是在调用事件之前将事件赋值给一个临时变量, 如示例所示, 这样可以确保在多个线程执行相同代码时的线程安全。如果不这样做, 在多个线程使用同一对象实例的某些情况下, 可能会抛出NullReferenceException异常。在 C# 6.0 中, 编译器会生成类似于示例代码中所示的代码。

第89.2节：事件属性

如果一个类触发大量事件, 每个委托使用一个字段的存储成本可能无法接受。针对这种情况, .NET 框架提供了事件属性。这样你可以使用其他数据结构, 比如 [EventHandlerList](#) 来存储事件委托:

```

public class SampleClass
{
    // 定义委托集合。
    protected EventHandlerList eventDelegates = new EventHandlerList();

    // 为每个事件定义唯一的键。
    static readonly object someEventKey = new object();

    // 定义 SomeEvent 事件属性。
    public event EventHandler SomeEvent
    {
        add
        {
            // 将输入的委托添加到集合中。
            eventDelegates.AddHandler(someEventKey, value);
        }
        remove
        {
            // 从集合中移除输入的委托。
            eventDelegates.RemoveHandler(someEventKey, value);
        }
    }

    // 使用 someEventKey 指定的委托触发事件
    protected void OnSomeEvent(EventArgs e)
    {
        var handler = (EventHandler)eventDelegates[someEventKey];
        if (handler != null)
            handler(this, e);
    }
}

```

这种方法广泛应用于像 WinForms 这样的图形用户界面框架中, 其中控件可以拥有数十甚至数百个事件。

请注意, EventHandlerList 不是线程安全的, 因此如果您预计您的类会被多个线程使用, 您需要添加锁语句或其他同步机制 (或者使用提供线程安全的存储)

```

// If eventName == null, then it means there are no event-subscribers,
// and therefore, we cannot raise the event.
if(eventName != null)
    eventName(this, EventArgs.Empty);
}

```

Note that events can only be raised by the declaring type. Clients can only subscribe/unsubscribe.

For C# versions before 6.0, where EventName?.Invoke is not supported, it is a good practice to assign the event to a temporary variable before invocation, as shown in the example, which ensures thread-safety in cases where multiple threads execute the same code. Failing to do so may cause a NullReferenceException to be thrown in certain cases where multiple threads are using the same object instance. In C# 6.0, the compiler emits code similar to that shown in the code example for C# 6.

Section 89.2: Event Properties

If a class raises a large the number of events, the storage cost of one field per delegate may not be acceptable. The .NET Framework provides [event properties](#) for these cases. This way you can use another data structure like [EventHandlerList](#) to store event delegates:

```

public class SampleClass
{
    // Define the delegate collection.
    protected EventHandlerList eventDelegates = new EventHandlerList();

    // Define a unique key for each event.
    static readonly object someEventKey = new object();

    // Define the SomeEvent event property.
    public event EventHandler SomeEvent
    {
        add
        {
            // Add the input delegate to the collection.
            eventDelegates.AddHandler(someEventKey, value);
        }
        remove
        {
            // Remove the input delegate from the collection.
            eventDelegates.RemoveHandler(someEventKey, value);
        }
    }

    // Raise the event with the delegate specified by someEventKey
    protected void OnSomeEvent(EventArgs e)
    {
        var handler = (EventHandler)eventDelegates[someEventKey];
        if (handler != null)
            handler(this, e);
    }
}

```

This approach is widely used in GUI frameworks like WinForms where controls can have dozens and even hundreds of events.

Note that EventHandlerList is not thread-safe, so if you expect your class to be used from multiple threads, you will need to add lock statements or other synchronization mechanism (or use a storage that provides thread

第89.3节：创建可取消事件

当类即将执行可以取消的操作时，例如
窗体的 [FormClosing](#) 事件。

要创建此类事件：

- 创建一个继承自 [CancelEventArgs](#) 的新事件参数，并添加用于事件数据的附加属性。
- 使用 [EventHandler<T>](#) 创建一个事件，并使用你创建的新取消事件参数类。

示例

在下面的示例中，我们为一个类的 Price 属性创建了一个 PriceChangingEventArgs 事件。事件数据类包含一个 Value，通知使用者新的值。事件在你为 Price 属性赋新值时触发，通知使用者值正在改变，并允许他们取消事件。如果使用者取消事件，将使用 Price 的先前值：

PriceChangingEventArgs

```
public class PriceChangingEventArgs : CancelEventArgs
{
    int value;
    public int Value
    {
        get { return value; }
    }
    public PriceChangingEventArgs(int value)
    {
        this.value = value;
    }
}
```

Product

```
public class Product
{
    int price;
    public int Price
    {
        get { return price; }
        set
        {
            var e = new PriceChangingEventArgs(value);
            OnPriceChanging(e);
            if (!e.Cancel)
                price = value;
        }
    }

    public event EventHandler<PriceChangingEventArgs> PropertyChanging;
    protected void OnPriceChanging(PriceChangingEventArgs e)
    {
        var handler = PropertyChanging;
        if (handler != null)
            PropertyChanging(this, e);
    }
}
```

safety)。

Section 89.3: Creating cancelable event

A cancelable event can be raised by a class when it is about to perform an action that can be canceled, such as the [FormClosing](#) event of a [Form](#).

To create such event:

- Create a new event arg deriving from [CancelEventArgs](#) and add additional properties for event data.
- Create an event using [EventHandler<T>](#) and use the new cancel event arg class which you created.

Example

In the below example, we create a PriceChangingEventArgs event for Price property of a class. The event data class contains a Value which let the consumer know about the new . The event raises when you assign a new value to Price property and lets the consumer know the value is changing and let them to cancel the event. If the consumer cancels the event, the previous value for Price will be used:

PriceChangingEventArgs

```
public class PriceChangingEventArgs : CancelEventArgs
{
    int value;
    public int Value
    {
        get { return value; }
    }
    public PriceChangingEventArgs(int value)
    {
        this.value = value;
    }
}
```

Product

```
public class Product
{
    int price;
    public int Price
    {
        get { return price; }
        set
        {
            var e = new PriceChangingEventArgs(value);
            OnPriceChanging(e);
            if (!e.Cancel)
                price = value;
        }
    }

    public event EventHandler<PriceChangingEventArgs> PropertyChanging;
    protected void OnPriceChanging(PriceChangingEventArgs e)
    {
        var handler = PropertyChanging;
        if (handler != null)
            PropertyChanging(this, e);
    }
}
```

}

第89.4节：标准事件声明

事件声明：

```
public event EventHandler<EventArgsT> EventName;
```

事件处理程序声明：

```
public void HandlerName(object sender, EventArgsT args) { /* 处理程序逻辑 */ }
```

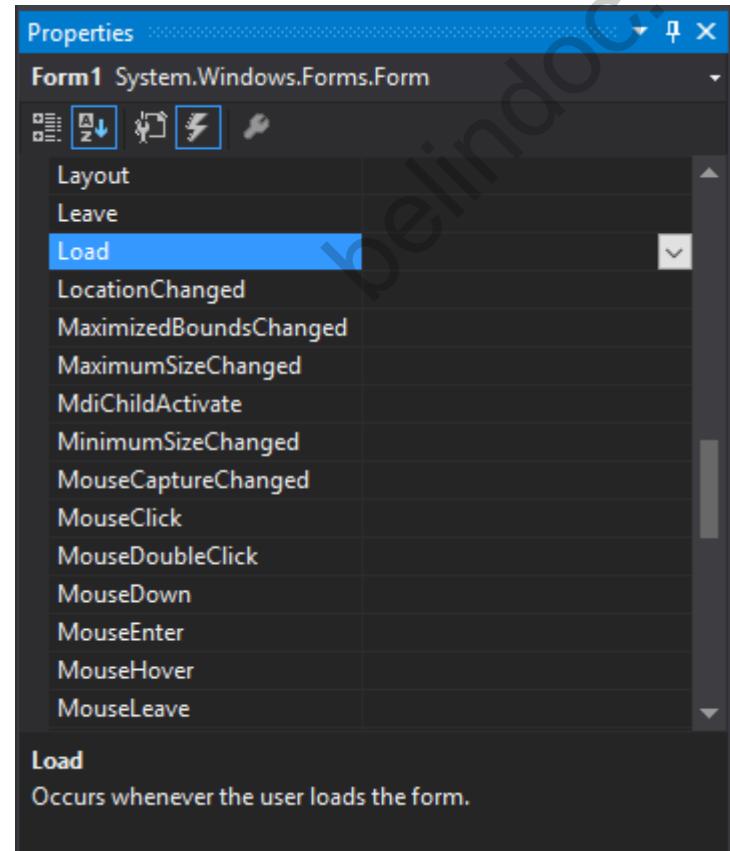
订阅事件：

动态方式：

```
EventName += HandlerName;
```

通过设计器：

1. 点击控件属性窗口中的事件按钮（闪电图标）
2. 双击事件名称：



3. Visual Studio 将生成事件代码：

```
private void Form1_Load(object sender, EventArgs e)
{}
```

调用方法：

}

Section 89.4: Standard Event Declaration

Event declaration:

```
public event EventHandler<EventArgsT> EventName;
```

Event handler declaration:

```
public void HandlerName(object sender, EventArgsT args) { /* Handler logic */ }
```

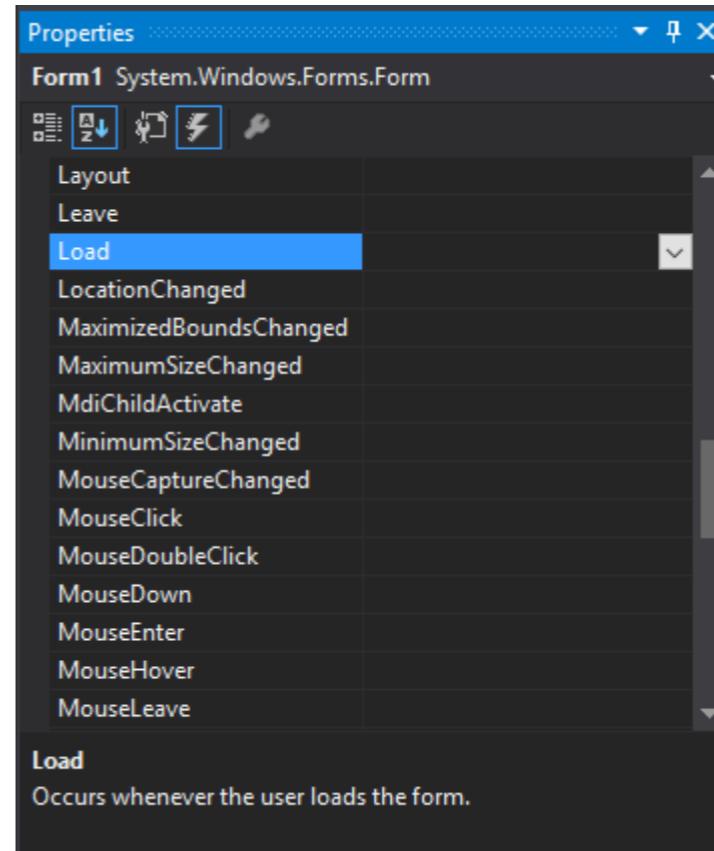
Subscribing to the event:

Dynamically:

```
EventName += HandlerName;
```

Through the Designer:

1. Click the Events button on the control's properties window (Lightening bolt)
2. Double-click the Event name:



3. Visual Studio will generate the event code:

```
private void Form1_Load(object sender, EventArgs e)
{}
```

Invoking the method:

```
EventName(SenderObject, EventArgs);
```

第89.5节：匿名事件处理器声明

事件声明：

```
public event EventHandler<EventArgsType> EventName;
```

使用lambda运算符=>声明事件处理器并订阅事件：

```
EventName += (obj, eventArgs) => { /* 处理程序逻辑 */ };
```

使用delegate匿名方法语法声明事件处理器：

```
EventName += delegate(object obj, EventArgsType eventArgs) { /* 处理程序逻辑 */ };
```

声明并订阅一个不使用事件参数的事件处理器，因此可以使用上述语法而无需指定参数：

```
EventName += delegate { /* 处理程序逻辑 */ }
```

触发事件：

```
EventName?.Invoke(SenderObject, EventArgs);
```

第89.6节：非标准事件声明

事件可以是任何委托类型，不仅限于EventHandler和EventHandler<T>。例如：

```
//声明一个事件
public event Action<Param1Type, Param2Type, ...> EventName;
```

这与标准的EventHandler事件的用法类似：

```
//添加一个具名事件处理器
public void HandlerName(Param1Type parameter1, Param2Type parameter2, ...) {
    /* 处理程序逻辑 */
}
EventName += HandlerName;

//添加一个匿名事件处理器
EventName += (parameter1, parameter2, ...) => { /* 处理程序逻辑 */ };

//调用事件
EventName(parameter1, parameter2, ...);
```

可以在一条语句中声明多个相同类型的事件，类似于字段和局部变量（尽管这通常不是一个好主意）：

```
public event EventHandler Event1, Event2, Event3;
```

这声明了三个独立的事件（Event1、Event2 和 Event3），它们的类型均为 EventHandler。

注意：虽然某些编译器可能接受接口和类中这种语法，但 C# 规范 (v5.0 §13.2.3) 为接口提供的语法不允许这样做，因此在接口中使用此语法可能因不同

```
EventName(SenderObject, EventArgs);
```

Section 89.5: Anonymous Event Handler Declaration

Event declaration:

```
public event EventHandler<EventArgsType> EventName;
```

Event handler declaration using lambda operator => and subscribing to the event:

```
EventName += (obj, eventArgs) => { /* Handler logic */ };
```

Event handler declaration using delegate anonymous method syntax:

```
EventName += delegate(object obj, EventArgsType eventArgs) { /* Handler Logic */ };
```

Declaration & subscription of an event handler that does not use the event's parameter, and so can use the above syntax without needing to specify parameters:

```
EventName += delegate { /* Handler Logic */ }
```

Invoking the event:

```
EventName?.Invoke(SenderObject, EventArgs);
```

Section 89.6: Non-Standard Event Declaration

Events can be of any delegate type, not just EventHandler and EventHandler<T>. For example:

```
//Declaring an event
public event Action<Param1Type, Param2Type, ...> EventName;
```

This is used similarly to standard EventHandler events:

```
//Adding a named event handler
public void HandlerName(Param1Type parameter1, Param2Type parameter2, ...) {
    /* Handler logic */
}
EventName += HandlerName;

//Adding an anonymous event handler
EventName += (parameter1, parameter2, ...) => { /* Handler Logic */ };

//Invoking the event
EventName(parameter1, parameter2, ...);
```

It is possible to declare multiple events of the same type in a single statement, similar to with fields and local variables (though this may often be a bad idea):

```
public event EventHandler Event1, Event2, Event3;
```

This declares three separate events (Event1, Event2, and Event3) all of type EventHandler.

Note: Although some compilers may accept this syntax in interfaces as well as classes, the C# specification (v5.0 §13.2.3) provides grammar for interfaces that does not allow it, so using this in interfaces may be unreliable with different

第89.7节：创建包含

附加数据的自定义 EventArgs

自定义事件通常需要包含事件信息的自定义事件参数。例如

[MouseEventArgs](#) 用于鼠标事件，如 `MouseDown` 或 `MouseUp` 事件，包含有关位置或按钮的信息，这些信息用于生成事件。

创建新事件时，若要创建自定义事件参数：

- 创建一个继承自 `EventArgs` 的类，并定义所需数据的属性。
- 按照惯例，类名应以 `EventArgs` 结尾。

示例

在下面的示例中，我们为某个类的 `Price` 属性创建了一个 `PriceChangingEventArgs` 事件。事件数据类包含 `CurrentPrice` 和 `NewPrice`。当你为 `Price` 属性赋新值时，事件会触发，通知使用者价格正在变化，并让他们了解当前价格和新价格：

PriceChangingEventArgs

```
public class PriceChangingEventArgs : EventArgs
{
    public PriceChangingEventArgs(int currentPrice, int newPrice)
    {
        this.CurrentPrice = currentPrice;
        this.NewPrice = newPrice;
    }

    public int CurrentPrice { get; private set; }
    public int NewPrice { get; private set; }
}
```

Product

```
public class Product
{
    public event EventHandler<PriceChangingEventArgs> PriceChanging;

    int price;
    public int Price
    {
        get { return price; }
        set
        {
            var e = new PriceChangingEventArgs(price, value);
            OnPriceChanging(e);
            price = value;
        }
    }

    protected void OnPriceChanging(PriceChangingEventArgs e)
    {
        var handler = PriceChanging;
        if (handler != null)
            handler(this, e);
    }
}
```

compilers.

Section 89.7: Creating custom EventArgs containing additional data

Custom events usually need custom event arguments containing information about the event. For example [MouseEventArgs](#) which is used by mouse events like `MouseDown` or `MouseUp` events, contains information about Location or Buttons which used to generate the event.

When creating new events, to create a custom event arg:

- Create a class deriving from `EventArgs` and define properties for necessary data.
- As a convention, the name of the class should ends with `EventArgs`.

Example

In the below example, we create a `PriceChangingEventArgs` event for `Price` property of a class. The event data class contains a `CurrentPrice` and a `NewPrice`. The event raises when you assign a new value to `Price` property and lets the consumer know the value is changing and let them to know about current price and new price:

PriceChangingEventArgs

```
public class PriceChangingEventArgs : EventArgs
{
    public PriceChangingEventArgs(int currentPrice, int newPrice)
    {
        this.CurrentPrice = currentPrice;
        this.NewPrice = newPrice;
    }

    public int CurrentPrice { get; private set; }
    public int NewPrice { get; private set; }
}
```

Product

```
public class Product
{
    public event EventHandler<PriceChangingEventArgs> PriceChanging;

    int price;
    public int Price
    {
        get { return price; }
        set
        {
            var e = new PriceChangingEventArgs(price, value);
            OnPriceChanging(e);
            price = value;
        }
    }

    protected void OnPriceChanging(PriceChangingEventArgs e)
    {
        var handler = PriceChanging;
        if (handler != null)
            handler(this, e);
    }
}
```

```
}
```

您可以通过允许使用者更改新值来增强示例，然后该值将用于属性。为此，只需在类中应用这些更改即可。

将NewPrice的定义更改为可设置：

```
public int NewPrice { get; set; }
```

更改Price的定义，在调用OnPriceChanging后使用 e.NewPrice作为属性的值：

```
int price;
public int Price
{
    get { return price; }
    set
    {
        var e = new PriceChangingEventArgs(price, value);
        OnPriceChanging(e);
        price = e.NewPrice;
    }
}
```

```
}
```

You can enhance the example by allowing the consumer to change the new value and then the value will be used for property. To do so it's enough to apply these changes in classes.

Change the definition of NewPrice to be settable:

```
public int NewPrice { get; set; }
```

Change the definition of Price to use e.NewPrice as value of property, after calling OnPriceChanging :

```
int price;
public int Price
{
    get { return price; }
    set
    {
        var e = new PriceChangingEventArgs(price, value);
        OnPriceChanging(e);
        price = e.NewPrice;
    }
}
```

第90章：表达式树

参数	详情
TDelegate	用于表达式的委托类型
lambdaExpression	该lambda表达式（例如 num => num < 5）表达式树是以树状数据结构排列的表达式。树中的每个节点都表示一个表达式，表达式即代码。Lambda表达式的内存表示就是表达式树，它包含查询的实际元素（即代码），但不包含其结果。表达式树使lambda表达式的结构变得透明且明确。

第90.1节：使用lambda表达式创建表达式树

下面是由lambda创建的最基本的表达式树。

```
Expression<Func<int, bool>> lambda = num => num == 42;
```

要“手动”创建表达式树，应使用Expression类。

上述表达式等价于：

```
ParameterExpression parameter = Expression.Parameter(typeof(int), "num"); // num 参数
ConstantExpression constant = Expression.Constant(42, typeof(int)); // 42 常量
BinaryExpression equality = Expression.Equals(parameter, constant); // 两个表达式的相等性
(Expression<Func<int, bool>> lambda = Expression.Lambda<Func<int, bool>>(equality, parameter));
```

第90.2节：使用API创建表达式树

```
using System.Linq.Expressions;

// 手动构建表达式树
// 对应lambda表达式 num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

第90.3节：编译表达式树

```
// 定义一个表达式树，输入为整数，返回bool类型。
Expression<Func<int, bool>> expr = num => num < 5;

// 调用表达式树的 Compile 方法以返回一个可调用的委托。
Func<int, bool> result = expr.Compile();

// 调用委托并将结果写入控制台。
Console.WriteLine(result(4)); // 输出 true

// 输出 True.

// 你也可以将编译步骤与调用/执行步骤合并，如下所示：
```

Chapter 90: Expression Trees

Parameter	Details
TDelegate	The delegate type to be used for the expression
lambdaExpression	The lambda expression (ex. num => num < 5)

Expression Trees are Expressions arranged in a treelike data structure. Each node in the tree is a representation of an expression, an expression being code. An In-Memory representation of a Lambda expression would be an Expression tree, which holds the actual elements (i.e. code) of the query, but not its result. Expression trees make the structure of a lambda expression transparent and explicit.

Section 90.1: Create Expression Trees with a lambda expression

Following is most basic expression tree that is created by lambda.

```
Expression<Func<int, bool>> lambda = num => num == 42;
```

To create expression trees 'by hand', one should use Expression class.

Expression above would be equivalent to:

```
ParameterExpression parameter = Expression.Parameter(typeof(int), "num"); // num argument
ConstantExpression constant = Expression.Constant(42, typeof(int)); // 42 constant
BinaryExpression equality = Expression.Equals(parameter, constant); // equality of two expressions
(Expression<Func<int, bool>> lambda = Expression.Lambda<Func<int, bool>>(equality, parameter));
```

Section 90.2: Creating Expression Trees by Using the API

```
using System.Linq.Expressions;

// Manually build the expression tree for
// the lambda expression num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });
```

Section 90.3: Compiling Expression Trees

```
// Define an expression tree, taking an integer, returning a bool.
Expression<Func<int, bool>> expr = num => num < 5;

// Call the Compile method on the expression tree to return a delegate that can be called.
Func<int, bool> result = expr.Compile();

// Invoke the delegate and write the result to the console.
Console.WriteLine(result(4)); // Prints true

// Prints True.

// You can also combine the compile step with the call/invoke step as below:
```

```
Console.WriteLine(expr.Compile()(4));
```

第90.4节：解析表达式树

```
using System.Linq.Expressions;

// 创建一个表达式树。
Expression<Func<int, bool>> exprTree = num => num < 5;

// 分解表达式树。
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("分解后的表达式: {0} => {1} {2} {3}",
    param.Name, left.Name, operation.NodeType, right.Value);

// 分解后的表达式: num => num LessThan 5
```

第90.5节：表达式树基础

表达式树以树状数据结构表示代码，其中每个节点都是一个表达式。表达式树支持动态修改可执行代码，在各种数据库中执行 LINQ 查询，以及创建动态查询。你可以编译并运行由表达式树表示的代码。

表达式树也用于动态语言运行时 (DLR)，以实现动态语言与 .NET 框架之间的互操作，并使编译器编写者能够生成表达式树，而不是微软中间语言 (MSIL)。

表达式树可以通过以下方式创建

1. 匿名 lambda 表达式，
2. 使用 System.Linq.Expressions 命名空间手动创建。

来自 Lambda 表达式的表达式树

当一个 lambda 表达式被赋值给 Expression 类型变量时，编译器会生成代码来构建一个表示该 lambda 表达式的表达式树。

下面的代码示例展示了如何让 C# 编译器创建一个表示 lambda 表达式 num => num < 5 的表达式树。

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

使用 API 创建表达式树

表达式树也可以使用 Expression 类创建。该类包含创建特定类型表达式树节点的静态工厂方法。

以下是几种类型的树节点。

1. ParameterExpression
2. MethodCallExpression

```
Console.WriteLine(expr.Compile()(4));
```

Section 90.4: Parsing Expression Trees

```
using System.Linq.Expressions;

// Create an expression tree.
Expression<Func<int, bool>> exprTree = num => num < 5;

// Decompose the expression tree.
ParameterExpression param = (ParameterExpression)exprTree.Parameters[0];
BinaryExpression operation = (BinaryExpression)exprTree.Body;
ParameterExpression left = (ParameterExpression)operation.Left;
ConstantExpression right = (ConstantExpression)operation.Right;

Console.WriteLine("Decomposed expression: {0} => {1} {2} {3}",
    param.Name, left.Name, operation.NodeType, right.Value);

// Decomposed expression: num => num LessThan 5
```

Section 90.5: Expression Tree Basic

Expression trees represent code in a tree-like data structure, where each node is an expression

Expression Trees enables dynamic modification of executable code, the execution of LINQ queries in various databases, and the creation of dynamic queries. You can compile and run code represented by expression trees.

These are also used in the dynamic language run-time (DLR) to provide interoperability between dynamic languages and the .NET Framework and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (MSIL).

Expression Trees can be created via

1. Anonymous lambda expression,
2. Manually by using the System.Linq.Expressions namespace.

Expression Trees from Lambda Expressions

When a lambda expression is assigned to Expression type variable , the compiler emits code to build an expression tree that represents the lambda expression.

The following code examples shows how to have the C# compiler create an expression tree that represents the lambda expression num => num < 5.

```
Expression<Func<int, bool>> lambda = num => num < 5;
```

Expression Trees by Using the API

Expression Trees also created using the **Expression** Class. This class contains static factory methods that create expression tree nodes of specific types.

Below are few type of Tree nodes.

1. ParameterExpression
2. MethodCallExpression

下面的代码示例展示了如何使用 API 创建一个表示 `lambda` 表达式 `num => num < 5` 的表达式树。

```
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 = Expression.Lambda<Func<int, bool>>(numLessThanFive, new ParameterExpression[] { numParam });
```

第90.6节：使用访问者检查表达式的结构

通过重写 `ExpressionVisitor` 的一些方法来定义一个新的访问者类：

```
class PrintingVisitor : ExpressionVisitor {
    protected override Expression VisitConstant(ConstantExpression node) {
        Console.WriteLine("常量: {0}", node);
        return base.VisitConstant(node);
    }
    protected override Expression VisitParameter(ParameterExpression node) {
        Console.WriteLine("参数: {0}", node);
        return base.VisitParameter(node);
    }
    protected override Expression VisitBinary(BinaryExpression node) {
        Console.WriteLine("二元操作符 {0}", node.NodeType);
        return base.VisitBinary(node);
    }
}
```

调用 `Visit` 以在现有表达式上使用此访问者：

```
Expression<Func<int, bool>> isBig = a => a > 1000000;
var visitor = new PrintingVisitor();
visitor.Visit(isBig);
```

第90.7节：理解表达式API

我们将使用表达式树 API 来创建一个 `CalculateSalesTax` 树。用通俗的话来说，以下是创建该树的步骤总结。

1. 检查产品是否应纳税
2. 如果是，则将该行总额乘以适用的税率并返回该金额
3. 否则返回0

```
//作为参考，我们使用API来构建这个lambda表达式
orderLine => orderLine.IsTaxable ? orderLine.Total * orderLine.Order.TaxRate : 0;

//传入方法的orderLine参数。我们指定它的类型 (OrderLine) 和参数名称。
ParameterExpression orderLine = Expression.Parameter(typeof(OrderLine), "orderLine");

//检查参数是否应纳税；首先需要访问is taxable属性，然后检查其是否为真
 PropertyInfo isTaxableAccessor = typeof(OrderLine).GetProperty("IsTaxable");
 MemberExpression getIsTaxable = Expression.MakeMemberAccess(orderLine, isTaxableAccessor);
 UnaryExpression isLineTaxable = Expression.IsTrue(getIsTaxable);
```

The following code example shows how to create an expression tree that represents the lambda expression `num => num < 5` by using the API.

```
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 = Expression.Lambda<Func<int, bool>>(numLessThanFive, new ParameterExpression[] { numParam });
```

Section 90.6: Examining the Structure of an Expression using Visitor

Define a new visitor class by overriding some of the methods of [ExpressionVisitor](#):

```
class PrintingVisitor : ExpressionVisitor {
    protected override Expression VisitConstant(ConstantExpression node) {
        Console.WriteLine("Constant: {0}", node);
        return base.VisitConstant(node);
    }
    protected override Expression VisitParameter(ParameterExpression node) {
        Console.WriteLine("Parameter: {0}", node);
        return base.VisitParameter(node);
    }
    protected override Expression VisitBinary(BinaryExpression node) {
        Console.WriteLine("Binary with operator {0}", node.NodeType);
        return base.VisitBinary(node);
    }
}
```

Call `Visit` to use this visitor on an existing expression:

```
Expression<Func<int, bool>> isBig = a => a > 1000000;
var visitor = new PrintingVisitor();
visitor.Visit(isBig);
```

Section 90.7: Understanding the expressions API

We're going to use the expression tree API to create a `CalculateSalesTax` tree. In plain English, here's a summary of the steps it takes to create the tree.

1. Check if the product is taxable
2. If it is, multiply the line total by the applicable tax rate and return that amount
3. Otherwise return 0

```
//For reference, we're using the API to build this lambda expression
orderLine => orderLine.IsTaxable ? orderLine.Total * orderLine.Order.TaxRate : 0;

//The orderLine parameter we pass in to the method. We specify it's type (OrderLine) and the name of the parameter.
ParameterExpression orderLine = Expression.Parameter(typeof(OrderLine), "orderLine");

//Check if the parameter is taxable; First we need to access the is taxable property, then check if it's true
PropertyInfo isTaxableAccessor = typeof(OrderLine).GetProperty("IsTaxable");
MemberExpression getIsTaxable = Expression.MakeMemberAccess(orderLine, isTaxableAccessor);
UnaryExpression isLineTaxable = Expression.IsTrue(getIsTaxable);
```

```

//在创建if之前，我们需要创建分支//如果该行应纳税，我们将返回总额
    乘以税率；获取总额和税率，然后相乘

    //获取总额
    PropertyInfo totalAccessor = typeof(OrderLine).GetProperty("Total");
    MemberExpression getTotal = Expression.MakeMemberAccess(orderLine, totalAccessor);

    //获取订单
    PropertyInfo orderAccessor = typeof(OrderLine).GetProperty("Order");
    MemberExpression getOrder = Expression.MakeMemberAccess(orderLine, orderAccessor);

    //获取税率 - 注意我们直接将 getOrder 表达式传递给成员访问
    PropertyInfo taxRateAccessor = typeof(Order).GetProperty("TaxRate");
    MemberExpression getTaxRate = Expression.MakeMemberAccess(getOrder, taxRateAccessor);

    //将两者相乘 - 注意我们直接将两个操作数表达式传递给乘法
    BinaryExpression multiplyTotalByRate = Expression.Multiply(getTotal, getTaxRate);

    //如果该行不需缴税，则返回常量值 - 0.0 (decimal 类型)
    ConstantExpression zero = Expression.Constant(0M);

    //创建实际的 if 判断和分支
    ConditionalExpression ifTaxableTernary = Expression.Condition(isLineTaxable,
        multiplyTotalByRate, zero);

    //将整个内容封装在一个"方法"中——一个Lambda表达式
    Expression<Func<OrderLine, decimal>> method = Expression.Lambda<Func<OrderLine,
        decimal>>(ifTaxableTernary, orderLine);

```

```

//Before creating the if, we need to create the branches
    //If the line is taxable, we'll return the total times the tax rate; get the total and tax rate,
    then multiply
    //Get the total
    PropertyInfo totalAccessor = typeof(OrderLine).GetProperty("Total");
    MemberExpression getTotal = Expression.MakeMemberAccess(orderLine, totalAccessor);

    //Get the order
    PropertyInfo orderAccessor = typeof(OrderLine).GetProperty("Order");
    MemberExpression getOrder = Expression.MakeMemberAccess(orderLine, orderAccessor);

    //Get the tax rate - notice that we pass the getOrder expression directly to the member access
    PropertyInfo taxRateAccessor = typeof(Order).GetProperty("TaxRate");
    MemberExpression getTaxRate = Expression.MakeMemberAccess(getOrder, taxRateAccessor);

    //Multiply the two - notice we pass the two operand expressions directly to multiply
    BinaryExpression multiplyTotalByRate = Expression.Multiply(getTotal, getTaxRate);

    //If the line is not taxable, we'll return a constant value - 0.0 (decimal)
    ConstantExpression zero = Expression.Constant(0M);

    //Create the actual if check and branches
    ConditionalExpression ifTaxableTernary = Expression.Condition(isLineTaxable,
        multiplyTotalByRate, zero);

    //Wrap the whole thing up in a "method" - a LambdaExpression
    Expression<Func<OrderLine, decimal>> method = Expression.Lambda<Func<OrderLine,
        decimal>>(ifTaxableTernary, orderLine);

```

第91章：重载解析

第91.1节：基本重载示例

此代码包含一个名为Hello的重载方法：

```
class Example
{
    public static void Hello(int arg)
    {
        Console.WriteLine("int");
    }

    public static void Hello(double arg)
    {
        Console.WriteLine("double");
    }

    public static void Main(string[] args)
    {
        你好(0);
        你好(0.0);
    }
}
```

当调用Main方法时，它将打印

```
int
double
```

在编译时，当编译器发现方法调用Hello(0)时，它会找到所有名为Hello的方法。在这种情况下，它找到了两个。然后它尝试确定哪个方法更好。确定哪个方法更好的算法很复杂，但通常归结为“尽可能少进行隐式转换”。

因此，在Hello(0)的情况下，方法Hello(int)不需要转换，但方法Hello(double)需要隐式数值转换。因此，编译器选择第一个方法。

在Hello(0.0)的情况下，无法将0.0隐式转换为int，所以方法Hello(int)甚至不会被考虑进行重载解析。只剩下一个方法，因此被编译器选择。

第91.2节：“params”不会展开，除非必要

以下程序：

```
class 程序
{
    static void Method(params Object[] objects)
    {
        System.Console.WriteLine(objects.Length);
    }

    static void Method(Object a, Object b)
    {
        System.Console.WriteLine("two");
    }

    static void Main(string[] args)
```

Chapter 91: Overload Resolution

Section 91.1: Basic Overloading Example

This code contains an overloaded method named **Hello**:

```
class Example
{
    public static void Hello(int arg)
    {
        Console.WriteLine("int");
    }

    public static void Hello(double arg)
    {
        Console.WriteLine("double");
    }

    public static void Main(string[] args)
    {
        Hello(0);
        Hello(0.0);
    }
}
```

When the **Main** method is called, it will print

```
int
double
```

At compile-time, when the compiler finds the method call `Hello(0)`, it finds all methods with the name `Hello`. In this case, it finds two of them. It then tries to determine which of the methods is *better*. The algorithm for determining which method is better is complex, but it usually boils down to "make as few implicit conversions as possible".

Thus, in the case of `Hello(0)`, no conversion is needed for the method `Hello(int)` but an implicit numeric conversion is needed for the method `Hello(double)`. Thus, the first method is chosen by the compiler.

In the case of `Hello(0.0)`, there is no way to convert `0.0` to an `int` implicitly, so the method `Hello(int)` is not even considered for overload resolution. Only method remains and so it is chosen by the compiler.

Section 91.2: "params" is not expanded, unless necessary

The following program:

```
class Program
{
    static void Method(params Object[] objects)
    {
        System.Console.WriteLine(objects.Length);
    }

    static void Method(Object a, Object b)
    {
        System.Console.WriteLine("two");
    }

    static void Main(string[] args)
```

```

{
    object[] objectArray = new object[5];

    Method(objectArray);
    Method(objectArray, objectArray);
    Method(objectArray, objectArray, objectArray);
}

```

将打印：

```

5
two
3

```

调用表达式Method(objectArray)可以有两种解释：一种是单个Object参数，恰好是一个数组（因此程序会输出1，因为那是参数的数量）；另一种是作为参数数组，按正常形式给出，就好像方法Method没有params关键字。在这些情况下，正常的非展开形式总是优先。因此，程序输出5。

在第二个表达式中，Method(objectArray, objectArray)，既适用第一种方法的展开形式，也适用传统的第二种方法。在这种情况下，非展开形式仍然优先，因此程序输出two。

在第三个表达式中，Method(objectArray, objectArray, objectArray)，唯一的选择是使用第一个方法的展开形式，因此程序输出3。

第91.3节：将null作为参数传递

如果你有

```

void F1(MyType1 x) {
    // 执行某些操作
}

void F1(MyType2 x) {
    // 执行其他操作
}

```

并且由于某种原因你需要调用F1的第一个重载，但传入的 `x = null`，那么简单地写

`F1(null);`

将无法编译，因为调用存在歧义。为了解决这个问题，你可以写成

`F1(null as MyType1);`

```

{
    object[] objectArray = new object[5];

    Method(objectArray);
    Method(objectArray, objectArray);
    Method(objectArray, objectArray, objectArray);
}

```

will print:

```

5
two
3

```

The call expression `Method(objectArray)` could be interpreted in two ways: a single `Object` argument that happens to be an array (so the program would output 1 because that would be the number of arguments, or as an array of arguments, given in the normal form, as though the method `Method` did not have the keyword `params`. In these situations, the normal, non-expanded form always takes precedence. So, the program outputs 5.

In the second expression, `Method(objectArray, objectArray)`, both the expanded form of the first method and the traditional second method are applicable. In this case also, non-expanded forms take precedence, so the program prints two.

In the third expression, `Method(objectArray, objectArray, objectArray)`, the only option is to use the expanded form of the first method, and so the program prints 3.

Section 91.3: Passing null as one of the arguments

If you have

```

void F1(MyType1 x) {
    // do something
}

void F1(MyType2 x) {
    // do something else
}

```

and for some reason you need to call the first overload of F1 but with `x = null`, then doing simply

`F1(null);`

will not compile as the call is ambiguous. To counter this you can do

`F1(null as MyType1);`

第92章：BindingList<T>

第92.1节：向列表添加项目

```
BindingList<string> listOfUIItems = new BindingList<string>();  
listOfUIItems.Add("Alice");  
listOfUIItems.Add("Bob");
```

第92.2节：避免N*2次迭代

这段代码放在Windows窗体事件处理程序中

```
var nameList = new BindingList<string>();  
ComboBox1.DataSource = nameList;  
for(long i = 0; i < 10000; i++ ) {  
    nameList.AddRange(new [] {"Alice", "Bob", "Carol" });  
}
```

这段代码执行时间很长，解决方法如下：

```
var nameList = new BindingList<string>();  
ComboBox1.DataSource = nameList;  
nameList.RaiseListChangedEvents = false;  
for(long i = 0; i < 10000; i++ ) {  
    nameList.AddRange(new [] {"爱丽丝", "鲍勃", "卡罗尔" });  
}  
nameList.RaiseListChangedEvents = true;  
nameList.ResetBindings();
```

Chapter 92: BindingList<T>

Section 92.1: Add item to list

```
BindingList<string> listOfUIItems = new BindingList<string>();  
listOfUIItems.Add("Alice");  
listOfUIItems.Add("Bob");
```

Section 92.2: Avoiding N*2 iteration

This is placed in a Windows Forms event handler

```
var nameList = new BindingList<string>();  
ComboBox1.DataSource = nameList;  
for(long i = 0; i < 10000; i++ ) {  
    nameList.AddRange(new [] {"Alice", "Bob", "Carol" });  
}
```

This takes a long time to execute, to fix, do the below:

```
var nameList = new BindingList<string>();  
ComboBox1.DataSource = nameList;  
nameList.RaiseListChangedEvents = false;  
for(long i = 0; i < 10000; i++ ) {  
    nameList.AddRange(new [] {"Alice", "Bob", "Carol" });  
}  
nameList.RaiseListChangedEvents = true;  
nameList.ResetBindings();
```

第93章：预处理器指令

第93.1节：条件表达式

当编译以下内容时，返回的值将根据定义了哪些指令而不同。

```
// 使用 /d:A 或 /d:B 编译以查看差异
string SomeFunction()
{
    #if A
        return "A";
    #elif B
        return "B";
    #else
        return "C";
    #endif
}
```

条件表达式通常用于记录调试版本的附加信息。

```
void SomeFunc()
{
    try
    {
        SomeRiskyMethod();
    }
    catch (ArgumentException ex)
    {
        #if DEBUG
        log.Error("SomeFunc", ex);
        #endif

        HandleException(ex);
    }
}
```

第93.2节：其他编译器指令

线

#line 控制编译器在输出警告和错误时报告的行号和文件名。

```
void Test()
{
    #line 42 "Answer"
    #line filename "SomeFile.cs"
    int life; // 编译器警告 CS0168 在 "SomeFile.cs" 的第 42 行
    #line default
    // 编译器警告重置为默认
}
```

Pragma Checksum

#pragma checksum 允许为生成的程序数据库（PDB）指定特定的校验和以便调试。

```
#pragma checksum "MyCode.cs" "{00000000-0000-0000-0000-000000000000}" "{0123456789A}"
```

Chapter 93: Preprocessor directives

Section 93.1: Conditional Expressions

When the following is compiled, it will return a different value depending on which directives are defined.

```
// Compile with /d:A or /d:B to see the difference
string SomeFunction()
{
    #if A
        return "A";
    #elif B
        return "B";
    #else
        return "C";
    #endif
}
```

Conditional expressions are typically used to log additional information for debug builds.

```
void SomeFunc()
{
    try
    {
        SomeRiskyMethod();
    }
    catch (ArgumentException ex)
    {
        #if DEBUG
        log.Error("SomeFunc", ex);
        #endif

        HandleException(ex);
    }
}
```

Section 93.2: Other Compiler Instructions

Line

#line controls the line number and filename reported by the compiler when outputting warnings and errors.

```
void Test()
{
    #line 42 "Answer"
    #line filename "SomeFile.cs"
    int life; // compiler warning CS0168 in "SomeFile.cs" at Line 42
    #line default
    // compiler warnings reset to default
}
```

Pragma Checksum

#pragma checksum allows the specification of a specific checksum for a generated program database (PDB) for debugging.

```
#pragma checksum "MyCode.cs" "{00000000-0000-0000-0000-000000000000}" "{0123456789A}"
```

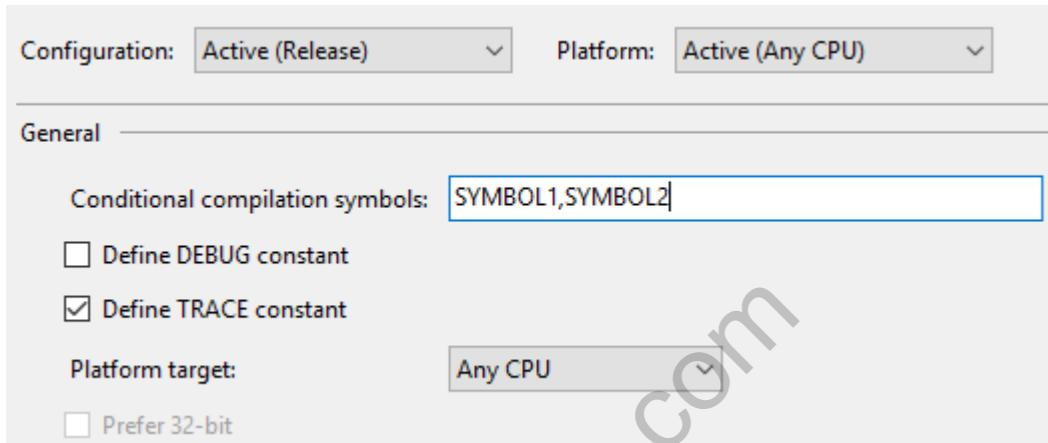
第93.3节：定义和取消定义符号

编译器符号是在编译时定义的关键字，可以用来检查以有条件地执行特定代码段。

定义编译器符号有三种方式。它们可以通过代码定义：

```
#define MYSYMBOL
```

它们可以在Visual Studio中定义，路径为项目属性 > 生成 > 条件编译符号：



(注意DEBUG和TRACE有各自的复选框，无需显式指定。)

或者可以在编译时使用C#编译器csc.exe的/define:[name]开关来定义。

你也可以使用#define指令来取消定义符号。

最常见的例子是DEBUG符号，当应用程序以调试模式（相对于发布模式）在Visual Studio中编译时，该符号会被定义。

```
public void DoBusinessLogic()
{
    try
    {
        AuthenticateUser();
        LoadAccount();
        ProcessAccount();
        FinalizeTransaction();
    }
    catch (Exception ex)
    {
        #if DEBUG
            System.Diagnostics.Trace.WriteLine("未处理的异常！");
            System.Diagnostics.Trace.WriteLine(ex);
            throw;
        #else
            LoggingFramework.LogError(ex);
            DisplayFriendlyErrorMessage();
        #endif
    }
}
```

在上面的示例中，当应用程序的业务逻辑中发生错误时，如果应用程序以调试模式（Debug mode）编译（并且设置了DEBUG符号），错误将被写入跟踪日志，并且异常将被重新抛出以便调试。

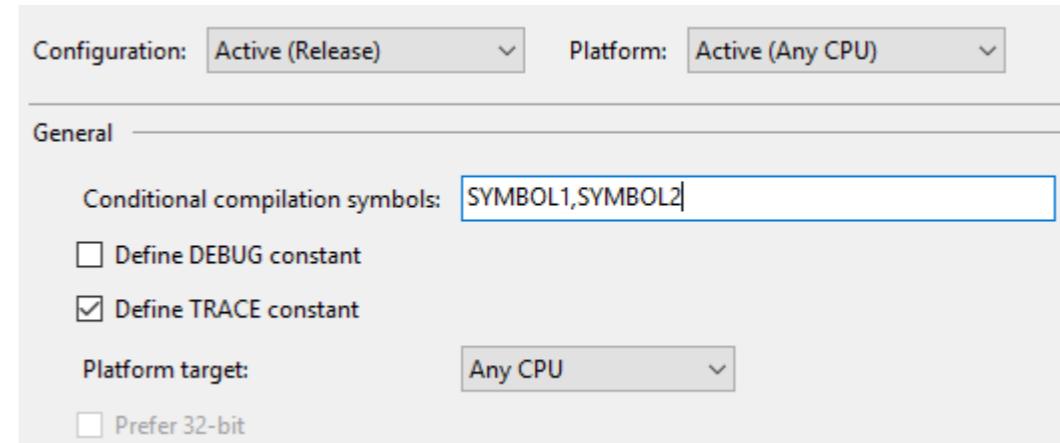
Section 93.3: Defining and Undefining Symbols

A compiler symbol is a keyword that is defined at compile-time that can be checked for to conditionally execute specific sections of code.

There are three ways to define a compiler symbol. They can be defined via code:

```
#define MYSYMBOL
```

They can be defined in Visual Studio, under Project Properties > Build > Conditional Compilation Symbols:



(Note that DEBUG and TRACE have their own checkboxes and do not need to be specified explicitly.)

Or they can be defined at compile-time using the /define:[name] switch on the C# compiler, csc .exe.

You can also undefined symbols using the #undefine directive.

The most prevalent example of this is the DEBUG symbol, which gets defined by Visual Studio when an application is compiled in Debug mode (versus Release mode).

```
public void DoBusinessLogic()
{
    try
    {
        AuthenticateUser();
        LoadAccount();
        ProcessAccount();
        FinalizeTransaction();
    }
    catch (Exception ex)
    {
        #if DEBUG
            System.Diagnostics.Trace.WriteLine("Unhandled exception!");
            System.Diagnostics.Trace.WriteLine(ex);
            throw;
        #else
            LoggingFramework.LogError(ex);
            DisplayFriendlyErrorMessage();
        #endif
    }
}
```

In the example above, when an error occurs in the business logic of the application, if the application is compiled in Debug mode (and the DEBUG symbol is set), the error will be written to the trace log, and the exception will be re-

但是，如果应用程序以发布模式（Release mode）编译（且未设置DEBUG符号），则使用日志框架静默记录错误，并向最终用户显示友好的错误信息。

第93.4节：区域块

使用#region和#endregion来定义可折叠的代码区域。

```
#region 事件处理程序

public void Button_Click(object s, EventArgs e)
{
    // ...
}

public void DropDown_SelectedIndexChanged(object s, EventArgs e)
{
    // ...
}

#endregion
```

这些指令仅在使用支持可折叠区域的集成开发环境（如Visual Studio）编辑代码时有用。

第93.5节：禁用和恢复编译器警告

您可以使用#pragma warning disable来禁用编译器警告，使用#pragma warning restore来恢复它们：

```
#pragma warning disable CS0168

// 由于警告被禁用，将不会生成“未使用变量”的编译器警告
var x = 5;

#pragma warning restore CS0168

// 由于警告刚刚被恢复，将会生成编译器警告
var y = 8;
```

允许使用逗号分隔的警告编号：

```
#pragma warning disable CS0168, CS0219
```

“CS”前缀是可选的，甚至可以混合使用（尽管这不是最佳实践）：

```
#pragma warning disable 0168, 0219, CS0414
```

第93.6节：生成编译器警告和错误

可以使用#warning指令生成编译器警告，同样也可以使用#error指令生成错误。

```
#if SOME_SYMBOL
#error 这是一个编译器错误。
#elif SOME_OTHER_SYMBOL
#warning 这是一个编译器警告。
```

thrown for debugging. However, if the application is compiled in Release mode (and no DEBUG symbol is set), a logging framework is used to quietly log the error, and a friendly error message is displayed to the end user.

Section 93.4: Region Blocks

Use #region and #endregion to define a collapsible code region.

```
#region Event Handlers

public void Button_Click(object s, EventArgs e)
{
    // ...
}

public void DropDown_SelectedIndexChanged(object s, EventArgs e)
{
    // ...
}

#endregion
```

These directives are only beneficial when an IDE that supports collapsible regions (such as [Visual Studio](#)) is used to edit the code.

Section 93.5: Disabling and Restoring Compiler Warnings

You can disable compiler warnings using #pragma warning disable and restore them using #pragma warning restore:

```
#pragma warning disable CS0168

// Will not generate the "unused variable" compiler warning since it was disabled
var x = 5;

#pragma warning restore CS0168

// Will generate a compiler warning since the warning was just restored
var y = 8;
```

Comma-separated warning numbers are allowed:

```
#pragma warning disable CS0168, CS0219
```

The CS prefix is optional, and can even be intermixed (though this is not a best practice):

```
#pragma warning disable 0168, 0219, CS0414
```

Section 93.6: Generating Compiler Warnings and Errors

Compiler warnings can be generated using the #warning directive, and errors can likewise be generated using the #error directive.

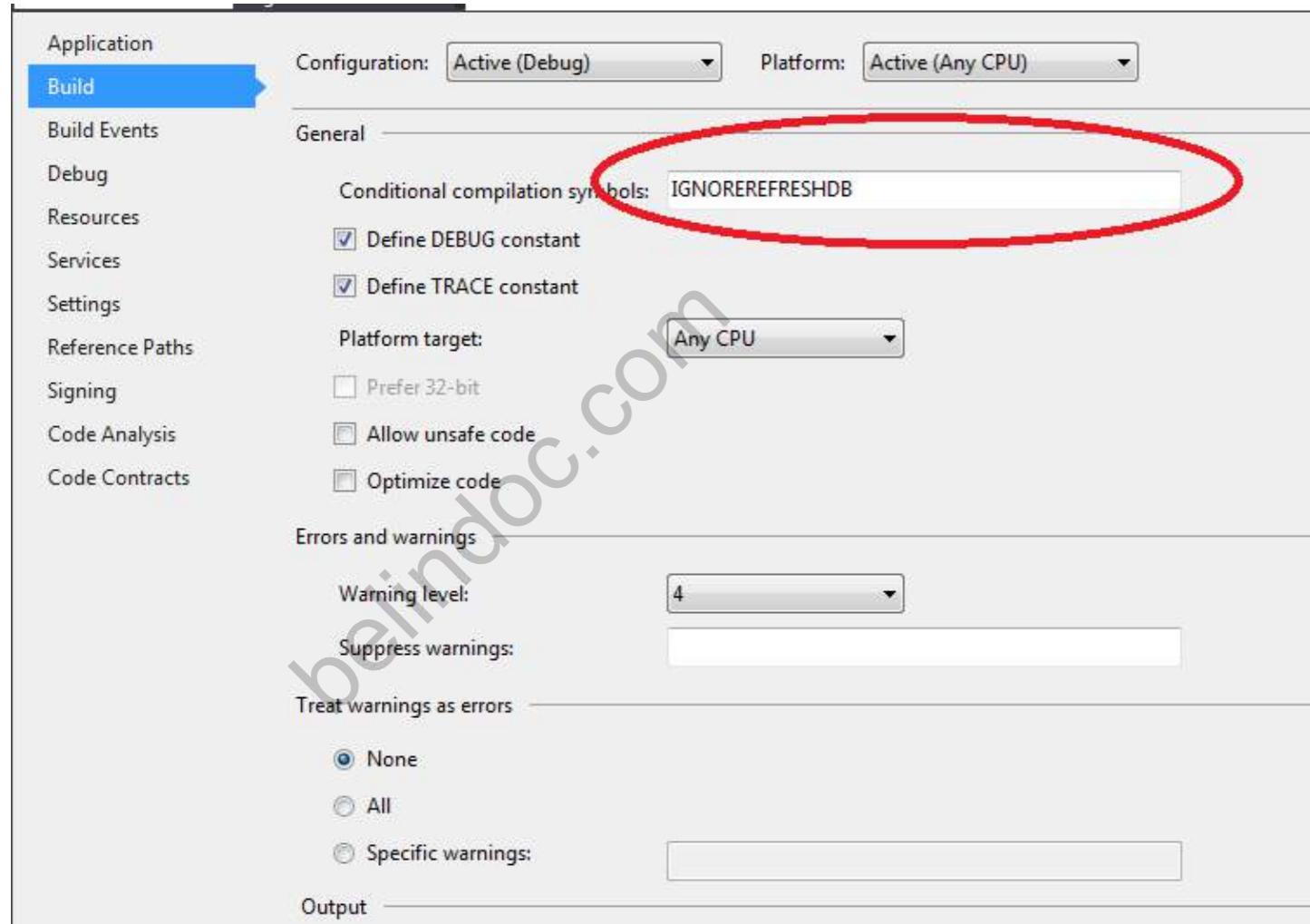
```
#if SOME_SYMBOL
#error This is a compiler Error.
#elif SOME_OTHER_SYMBOL
#warning This is a compiler Warning.
```

```
#endif
```

第93.7节：项目级别的自定义预处理器

当需要跳过某些操作，比如测试时，在项目级别设置自定义条件预处理是很方便的。

进入解决方案资源管理器 -> 点击 **项目上点击鼠标右键** -> 属性 -> 生成 -> 在“常规”中找到字段条件编译符号并在此输入你的条件变量



跳过部分代码的示例：

```
public void Init()
{
    #if !IGNOREREFRESHDB
    // will skip code here
    db.Initialize();
    #endif
}
```

第93.8节：使用Conditional属性

从System.Diagnostics命名空间添加Conditional属性到方法，是一种干净的方式来控制哪些方法会在你的构建中被调用，哪些不会。

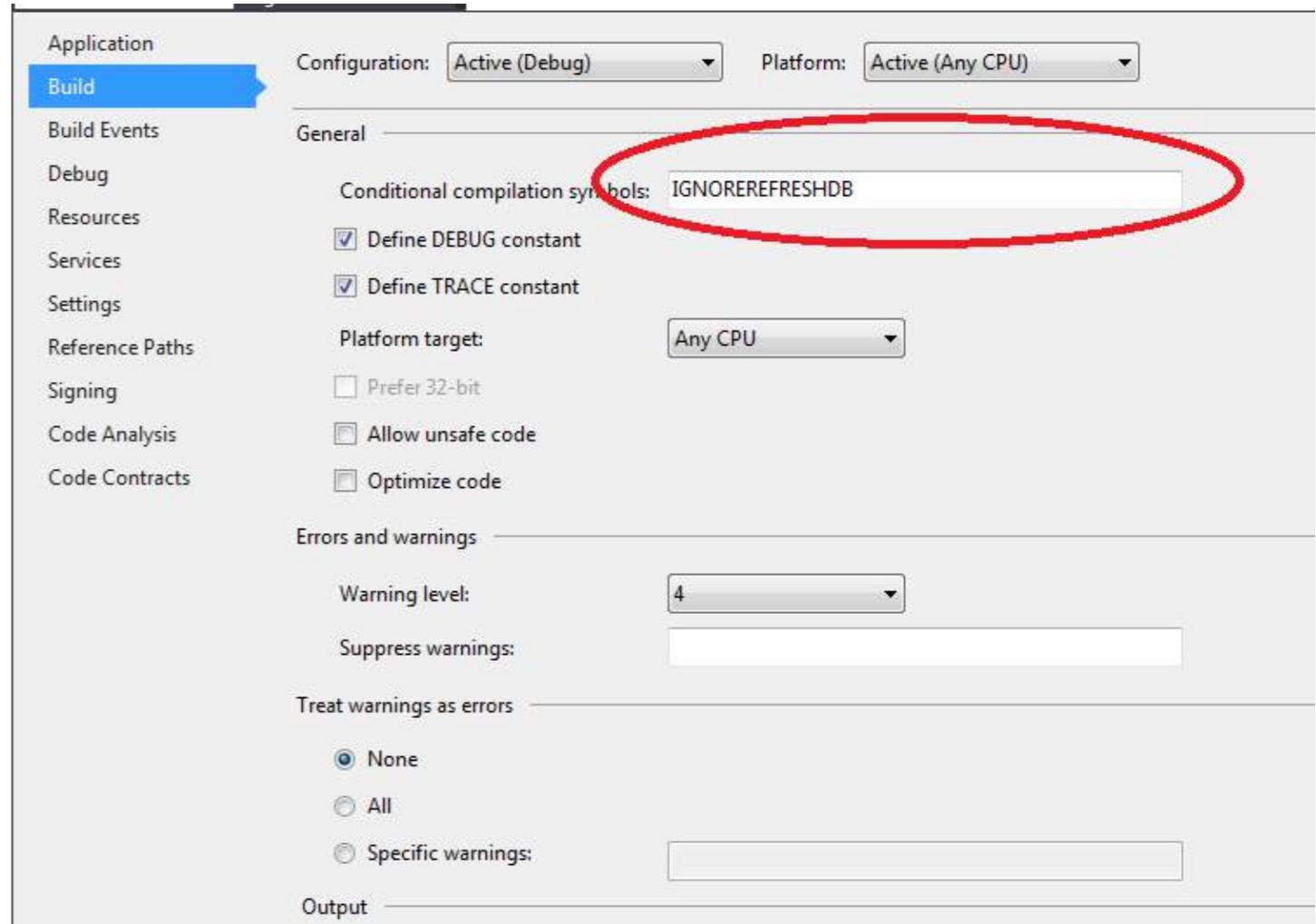
```
#define EXAMPLE_A
```

```
#endif
```

Section 93.7: Custom Preprocessors at project level

It is convenient to set custom conditional preprocessing at project level when some actions need to be skipped lets say for tests.

Go to Solution Explorer -> Click **Right Mouse** on project you want to set variable to -> Properties -> Build -> In General find field Conditional compilation symbols and enter your conditional variable here



Code example that will skip some code:

```
public void Init()
{
    #if !IGNOREREFRESHDB
    // will skip code here
    db.Initialize();
    #endif
}
```

Section 93.8: Using the Conditional attribute

Adding a Conditional attribute from System.Diagnostics namespace to a method is a clean way to control which methods are called in your builds and which are not.

```
#define EXAMPLE_A
```

```
using System.Diagnostics;
class Program
{
    static void Main()
    {
ExampleA(); // 该方法将被调用
        ExampleB(); // 该方法将不会被调用
    }

[Conditional("EXAMPLE_A")]
static void ExampleA() {...}

[Conditional("EXAMPLE_B")]
static void ExampleB() {...}
}
```

```
using System.Diagnostics;
class Program
{
    static void Main()
    {
        ExampleA(); // This method will be called
        ExampleB(); // This method will not be called
    }

[Conditional("EXAMPLE_A")]
static void ExampleA() {...}

[Conditional("EXAMPLE_B")]
static void ExampleB() {...}
}
```

第94章：结构体

第94.1节：声明结构体

```
public struct 向量
{
    public int X;
    public int Y;
    public int Z;
}

public struct 点
{
    public decimal x, y;

    public 点(decimal pointX, decimal pointY)
    {
        x = pointX;
        y = pointY;
    }
}
```

- 结构体的实例字段可以通过带参数的构造函数设置，也可以在结构体构造后单独设置。
- 私有成员只能由构造函数初始化。
- 结构体定义了一个隐式继承自 `System.ValueType` 的密封类型。
- 结构体不能继承自任何其他类型，但它们可以实现接口。
- 结构体在赋值时会被复制，这意味着所有数据都会被复制到新的实例中，对其中一个的更改不会反映到另一个上。
- 结构体不能为`null`，尽管它可以用作可空类型：

```
Vector v1 = null; //非法
Vector? v2 = null; //合法
Nullable<Vector> v3 = null // 合法
```

- 结构体可以使用或不使用`new`操作符来实例化。

```
//这两种方式都是可接受的
Vector v1 = new Vector();
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Vector v2;
v2.X = 1;
v2.Y = 2;
v2.Z = 3;
```

但是，必须使用`new`操作符才能使用初始化器：

```
Vector v1 = new MyStruct { X=1, Y=2, Z=3 }; // 合法
Vector v2 { X=1, Y=2, Z=3 }; // 非法
```

Chapter 94: Structs

Section 94.1: Declaring a struct

```
public struct Vector
{
    public int X;
    public int Y;
    public int Z;
}

public struct Point
{
    public decimal x, y;

    public Point(decimal pointX, decimal pointY)
    {
        x = pointX;
        y = pointY;
    }
}
```

- `struct` instance fields can be set via a parametrized constructor or individually after `struct` construction.
- Private members can only be initialized by the constructor.
- `struct` defines a sealed type that implicitly inherits from `System.ValueType`.
- Structs cannot inherit from any other type, but they can implement interfaces.
- Structs are copied on assignment, meaning all data is copied to the new instance and changes to one of them are not reflected by the other.
- A struct cannot be `null`, although it *can* be used as a nullable type:

```
Vector v1 = null; //illegal
Vector? v2 = null; //OK
Nullable<Vector> v3 = null // OK
```

- Structs can be instantiated with or without using the `new` operator.

```
//Both of these are acceptable
Vector v1 = new Vector();
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Vector v2;
v2.X = 1;
v2.Y = 2;
v2.Z = 3;
```

However, the `new` operator must be used in order to use an initializer:

```
Vector v1 = new MyStruct { X=1, Y=2, Z=3 }; // OK
Vector v2 { X=1, Y=2, Z=3 }; // illegal
```

结构体可以声明类能声明的所有内容，但有一些例外：

- 结构体不能声明无参数构造函数。结构体实例字段可以通过带参数的构造函数设置，或者在结构体构造后单独设置。私有成员只能由构造函数初始化。
- 结构体不能声明受保护的成员，因为它隐式为密封的。
- 结构体字段只有在是常量 (const) 或静态 (static) 时才能初始化。

第94.2节：结构体的使用

带构造函数：

```
Vector v1 = new Vector();
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}", v1.X, v1.Y, v1.Z);
// 输出 X=1,Y=2,Z=3

Vector v1 = new Vector();
//v1.X 未被赋值
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}", v1.X, v1.Y, v1.Z);
// 输出 X=0,Y=2,Z=3

Point point1 = new Point();
point1.x = 0.5;
point1.y = 0.6;

Point point2 = new Point(0.5, 0.6);
```

无构造函数：

```
Vector v1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}", v1.X, v1.Y, v1.Z);
//输出错误 "可能未赋值字段 'X' 的使用"

Vector v1;
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}", v1.X, v1.Y, v1.Z);
// 输出 X=1,Y=2,Z=3

Point point3;
point3.x = 0.5;
point3.y = 0.6;
```

如果我们使用带有构造函数的结构体，就不会遇到未赋值字段的问题（每个未赋值字段的值为 null）。

A struct can declare everything a class can declare, with a few exceptions:

- A struct cannot declare a parameterless constructor. `struct` instance fields can be set via a parameterized constructor or individually after `struct` construction. Private members can only be initialized by the constructor.
- A struct cannot declare members as protected, since it is implicitly sealed.
- Struct fields can only be initialized if they are const or static.

Section 94.2: Struct usage

With constructor:

```
Vector v1 = new Vector();
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}", v1.X, v1.Y, v1.Z);
// Output X=1,Y=2,Z=3

Vector v1 = new Vector();
//v1.X is not assigned
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}", v1.X, v1.Y, v1.Z);
// Output X=0,Y=2,Z=3

Point point1 = new Point();
point1.x = 0.5;
point1.y = 0.6;

Point point2 = new Point(0.5, 0.6);
```

Without constructor:

```
Vector v1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}", v1.X, v1.Y, v1.Z);
//Output ERROR "Use of possibly unassigned field 'X'

Vector v1;
v1.X = 1;
v1.Y = 2;
v1.Z = 3;

Console.WriteLine("X = {0}, Y = {1}, Z = {2}", v1.X, v1.Y, v1.Z);
// Output X=1,Y=2,Z=3

Point point3;
point3.x = 0.5;
point3.y = 0.6;
```

If we use a struct with its constructor, we aren't going to have problems with unassigned field (each unassigned field has null value).

与类不同，结构体不必被构造，即除非需要调用某个构造函数，否则不需要使用 new 关键字。结构体不需要 new 关键字，因为它是值类型，因此不能为 null。

第94.3节：结构体在赋值时会被复制

由于结构体是值类型，所有数据在赋值时都会被复制，对新副本的任何修改都不会改变原始副本的数据。下面的代码片段显示了 p1 被复制到 p2，对 p1 的修改不会影响 p2 实例。

```
var p1 = new Point {  
    x = 1,  
    y = 2  
};  
  
Console.WriteLine($"{p1.x} {p1.y}"); // 1 2  
  
var p2 = p1;  
Console.WriteLine($"{p2.x} {p2.y}"); // 相同输出：1 2  
  
p1.x = 3;  
Console.WriteLine($"{p1.x} {p1.y}"); // 3 2  
Console.WriteLine($"{p2.x} {p2.y}"); // p2 保持不变：1 2
```

Unlike classes, a struct doesn't have to be constructed, i.e. there is no need to use the new keyword, unless you need to call one of the constructors. A struct does not require the new keyword because it is a value-type and thus cannot be null.

Section 94.3: Structs are copied on assignment

Since structs are value types all the data is *copied* on assignment, and any modification to the new copy does not change the data for the original copy. The code snippet below shows that p1 is *copied* to p2 and changes made on p1 does not affect p2 instance.

```
var p1 = new Point {  
    x = 1,  
    y = 2  
};  
  
Console.WriteLine($"{p1.x} {p1.y}"); // 1 2  
  
var p2 = p1;  
Console.WriteLine($"{p2.x} {p2.y}"); // Same output: 1 2  
  
p1.x = 3;  
Console.WriteLine($"{p1.x} {p1.y}"); // 3 2  
Console.WriteLine($"{p2.x} {p2.y}"); // p2 remain the same: 1 2
```

第94.4节：实现接口的结构体

```
public interface IShape  
{  
    decimal Area();  
}  
  
public struct Rectangle : IShape  
{  
    public decimal Length { get; set; }  
    public decimal Width { get; set; }  
  
    public decimal Area()  
    {  
        return Length * Width;  
    }  
}
```

Section 94.4: Struct implementing interface

```
public interface IShape  
{  
    decimal Area();  
}  
  
public struct Rectangle : IShape  
{  
    public decimal Length { get; set; }  
    public decimal Width { get; set; }  
  
    public decimal Area()  
    {  
        return Length * Width;  
    }  
}
```

第95章：特性

第95.1节：创建自定义特性

```
//1) 所有特性都应继承自 System.Attribute  
//2) 你可以通过使用 System.AttributeUsage 来自定义特性的使用（例如设置限制）  
Attribute  
//3) 你只能通过反射以预期的方式使用此特性  
//4) MethodMetadataAttribute 只是一个名称。你可以在不加“Attribute”后缀的情况下使用它——例如  
[MethodMetadata("此文本可通过反射获取")].  
//5) 你可以重载属性构造函数  
[System.AttributeUsage(System.AttributeTargets.Method | System.AttributeTargets.Class)]  
public class MethodMetadataAttribute : System.Attribute  
{  
    //这是一个仅作为示例的自定义字段  
    //你可以创建没有任何字段的属性  
    //甚至可以使用空属性——作为标记  
    public string Text { get; set; }  
  
    //此构造函数可用作 [MethodMetadata]  
    public MethodMetadataAttribute ()  
    {  
    }  
  
    //此构造函数可用作 [MethodMetadata("字符串")]  
    public MethodMetadataAttribute (string text)  
    {  
        Text = text;  
    }  
}
```

第95.2节：读取属性

方法GetCustomAttributes返回应用于成员的自定义属性数组。检索到该数组后，您可以搜索一个或多个特定的属性。

```
var attribute = typeof(MyClass).GetCustomAttributes().OfType<MyCustomAttribute>().Single();
```

或者遍历它们

```
foreach(var attribute in typeof(MyClass).GetCustomAttributes()) {  
    Console.WriteLine(attribute.GetType());  
}
```

来自System.Reflection.CustomAttributeExtensions的GetCustomAttribute扩展方法检索指定类型的自定义属性，它可以应用于任何MemberInfo。

```
var attribute = (MyCustomAttribute) typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute));
```

GetCustomAttribute也有泛型签名以指定要搜索的属性类型。

```
var attribute = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>();
```

布尔参数inherit可以传递给这两个方法。如果该值设置为true，则元素的祖先也会被检查。

Chapter 95: Attributes

Section 95.1: Creating a custom attribute

```
//1) All attributes should be inherited from System.Attribute  
//2) You can customize your attribute usage (e.g. place restrictions) by using System.AttributeUsage  
Attribute  
//3) You can use this attribute only via reflection in the way it is supposed to be used  
//4) MethodMetadataAttribute is just a name. You can use it without "Attribute" postfix - e.g.  
[MethodMetadata("This text could be retrieved via reflection")].  
//5) You can overload an attribute constructors  
[System.AttributeUsage(System.AttributeTargets.Method | System.AttributeTargets.Class)]  
public class MethodMetadataAttribute : System.Attribute  
{  
    //this is custom field given just for an example  
    //you can create attribute without any fields  
    //even an empty attribute can be used - as marker  
    public string Text { get; set; }  
  
    //this constructor could be used as [MethodMetadata]  
    public MethodMetadataAttribute ()  
    {  
    }  
  
    //This constructor could be used as [MethodMetadata("String")]  
    public MethodMetadataAttribute (string text)  
    {  
        Text = text;  
    }  
}
```

Section 95.2: Reading an attribute

Method GetCustomAttributes returns an array of custom attributes applied to the member. After retrieving this array you can search for one or more specific attributes.

```
var attribute = typeof(MyClass).GetCustomAttributes().OfType<MyCustomAttribute>().Single();
```

Or iterate through them

```
foreach(var attribute in typeof(MyClass).GetCustomAttributes()) {  
    Console.WriteLine(attribute.GetType());  
}
```

GetCustomAttribute extension method from System.Reflection.CustomAttributeExtensions retrieves a custom attribute of a specified type, it can be applied to any MemberInfo.

```
var attribute = (MyCustomAttribute) typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute));
```

GetCustomAttribute also has generic signature to specify type of attribute to search for.

```
var attribute = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>();
```

Boolean argument inherit can be passed to both of those methods. If this value set to true the ancestors of element would be also to inspected.

第95.3节：使用属性

```
[StackDemo(Text = "Hello, World!")]
public class MyClass
{
    [StackDemo("Hello, World!")]
    static void MyMethod()
    {
    }
}
```

第95.4节：DebuggerDisplay 属性

添加DebuggerDisplay属性将改变调试器在悬停时显示类的方式。

用{}包裹的表达式将由调试器计算。这可以是像下面示例中的简单属性，也可以是更复杂的逻辑。

```
[DebuggerDisplay("{StringProperty} - {IntProperty}")]
public class AnObject
{
    public int ObjectId { get; set; }
    public string StringProperty { get; set; }
    public int IntProperty { get; set; }
}
```



在闭括号前添加,nq可以在输出字符串时去除引号。

```
[DebuggerDisplay("{StringProperty,nq} - {IntProperty}")]

```

尽管在{}中允许使用通用表达式，但不推荐这样做。DebuggerDisplay属性将作为字符串写入程序集元数据。{}中的表达式不会被检查其有效性。因此，包含比简单算术更复杂逻辑的DebuggerDisplay属性在C#中可能运行良好，但在VB.NET中评估相同表达式时，语法可能无效并在调试时产生错误。

使DebuggerDisplay更具语言无关性的一种方法是将表达式写入方法或属性中，然后调用它。

```
[DebuggerDisplay("{DebuggerDisplay(),nq}")]
public class 一个对象
{
    public int ObjectId { get; set; }
    public string StringProperty { get; set; }
    public int IntProperty { get; set; }

    private string DebuggerDisplay()
    {
    }
}
```

Section 95.3: Using an attribute

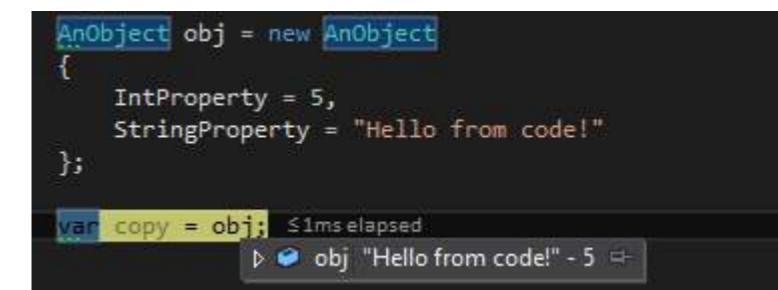
```
[StackDemo(Text = "Hello, World!")]
public class MyClass
{
    [StackDemo("Hello, World!")]
    static void MyMethod()
    {
    }
}
```

Section 95.4: DebuggerDisplay Attribute

Adding the DebuggerDisplay Attribute will change the way the debugger displays the class when it is hovered over.

Expressions that are wrapped in {} will be evaluated by the debugger. This can be a simple property like in the following sample or more complex logic.

```
[DebuggerDisplay("{StringProperty} - {IntProperty}")]
public class AnObject
{
    public int ObjectId { get; set; }
    public string StringProperty { get; set; }
    public int IntProperty { get; set; }
}
```



Adding ,nq before the closing bracket removes the quotes when outputting a string.

```
[DebuggerDisplay("{StringProperty,nq} - {IntProperty}")]

```

Even though general expressions are allowed in the {} they are not recommended. The DebuggerDisplay attribute will be written into the assembly metadata as a string. Expressions in {} are not checked for validity. So a DebuggerDisplay attribute containing more complex logic than i.e. some simple arithmetic might work fine in C#, but the same expression evaluated in VB.NET will probably not be syntactically valid and produce an error while debugging.

A way to make DebuggerDisplay more language agnostic is to write the expression in a method or property and call it instead.

```
[DebuggerDisplay("{DebuggerDisplay(),nq}")]
public class AnObject
{
    public int ObjectId { get; set; }
    public string StringProperty { get; set; }
    public int IntProperty { get; set; }

    private string DebuggerDisplay()
    {
    }
}
```

```

        return $"{StringProperty} - {IntProperty}"";
    }
}

```

可能希望DebuggerDisplay输出所有或部分属性，并在调试和检查时也显示对象的类型。

下面的示例还用#if DEBUG包围了辅助方法，因为DebuggerDisplay用于调试环境。

```

[DebuggerDisplay("{DebuggerDisplay(),nq}")]
public class 一个对象
{
    public int ObjectId { get; set; }
    public string StringProperty { get; set; }
    public int IntProperty { get; set; }

#if DEBUG
    private string DebuggerDisplay()
    {
        return
            $"ObjectId:{this.ObjectId}, StringProperty:{this.StringProperty},
            Type:{this.GetType()}";
    }
#endif
}

```

第95.5节：调用者信息属性

调用者信息属性可用于将调用者的信息传递给被调用的方法。

声明如下：

```

using System.Runtime.CompilerServices;

public void LogException(Exception ex,
    [CallerMemberName]string callerMemberName = "",
    [CallerLineNumber]int callerLineNumber = 0,
    [CallerFilePath]string callerFilePath = "")
{
    //执行日志记录
}

```

调用方式如下：

```

public void Save(DBContext context)
{
    try
    {
        context.SaveChanges();
    }
    catch (Exception ex)
    {
        LogException(ex);
    }
}

```

请注意，只有第一个参数被显式传递给LogException方法，而其余参数将在编译时提供相应的值。

```

        return $"{StringProperty} - {IntProperty}"";
    }
}

```

One might want DebuggerDisplay to output all or just some of the properties and when debugging and inspecting also the type of the object.

The example below also surrounds the helper method with #if DEBUG as DebuggerDisplay is used in debugging environments.

```

[DebuggerDisplay("'{DebuggerDisplay(),nq}'")]
public class AnObject
{
    public int ObjectId { get; set; }
    public string StringProperty { get; set; }
    public int IntProperty { get; set; }

#if DEBUG
    private string DebuggerDisplay()
    {
        return
            $"ObjectId:{this.ObjectId}, StringProperty:{this.StringProperty},
            Type:{this.GetType()}";
    }
#endif
}

```

Section 95.5: Caller info attributes

Caller info attributes can be used to pass down information about the invoker to the invoked method. The declaration looks like this:

```

using System.Runtime.CompilerServices;

public void LogException(Exception ex,
    [CallerMemberName]string callerMemberName = "",
    [CallerLineNumber]int callerLineNumber = 0,
    [CallerFilePath]string callerFilePath = "")
{
    //perform logging
}

```

And the invocation looks like this:

```

public void Save(DBContext context)
{
    try
    {
        context.SaveChanges();
    }
    catch (Exception ex)
    {
        LogException(ex);
    }
}

```

Notice that only the first parameter is passed explicitly to the LogException method whereas the rest of them will be provided at compile time with the relevant values.

调用者成员名 (callerMemberName) 参数将接收值“Save”——调用方法的名称。

调用者行号 (callerLineNumber) 参数将接收LogException方法调用所在的行号。

而调用者文件路径 (callerFilePath) 参数将接收声明Save方法的文件的完整路径。

第95.6节：过时属性 (Obsolete Attribute)

System.Obsolete 是一个用于标记某个类型或成员已有更好版本，因此不应使用的属性。

```
[Obsolete("此类已过时。请改用 SomeOtherClass。")]
class SomeClass
{
    //
}
```

如果使用上述类，编译器将发出警告“此类已过时。请改用 SomeOtherClass。”

第95.7节：从接口读取属性

由于类不会从接口继承属性，因此没有简单的方法从接口获取属性。每当实现接口或在派生类中重写成员时，都需要重新声明属性。所以在下面的示例中，输出在三种情况下都会是True。

```
using System;
using System.Linq;
using System.Reflection;

namespace InterfaceAttributesDemo {

    [AttributeUsage(AttributeTargets.Interface, Inherited = true)]
    class MyCustomAttribute : Attribute {
        public string Text { get; set; }
    }

    [MyCustomAttribute(Text = "Hello from interface attribute")]
    interface IMyClass {
        void MyMethod();
    }

    class MyClass : IMyClass {
        public void MyMethod() { }
    }

    public class Program {
        public static void Main(string[] args) {
            GetInterfaceAttributeDemo();
        }

        private static void GetInterfaceAttributeDemo() {
            var attribute1 = (MyCustomAttribute)
                typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute), true);
            Console.WriteLine(attribute1 == null); // True
        }
    }
}
```

The callerMemberName parameter will receive the value "Save" - the name of the calling method.

The callerLineNumber parameter will receive the number of whichever line the LogException method call is written on.

And the 'callerFilePath' parameter will receive the full path of the file Save method is declared in.

Section 95.6: Obsolete Attribute

System.Obsolete is an attribute that is used to mark a type or a member that has a better version, and thus should not be used.

```
[Obsolete("This class is obsolete. Use SomeOtherClass instead.")]
class SomeClass
{
    //
}
```

In case the class above is used, the compiler will give the warning "This class is obsolete. Use SomeOtherClass instead."

Section 95.7: Reading an attribute from interface

There is no simple way to obtain attributes from an interface, since classes does not inherit attributes from an interface. Whenever implementing an interface or overriding members in a derived class, you need to re-declare the attributes. So in the example below output would be True in all three cases.

```
using System;
using System.Linq;
using System.Reflection;

namespace InterfaceAttributesDemo {

    [AttributeUsage(AttributeTargets.Interface, Inherited = true)]
    class MyCustomAttribute : Attribute {
        public string Text { get; set; }
    }

    [MyCustomAttribute(Text = "Hello from interface attribute")]
    interface IMyClass {
        void MyMethod();
    }

    class MyClass : IMyClass {
        public void MyMethod() { }
    }

    public class Program {
        public static void Main(string[] args) {
            GetInterfaceAttributeDemo();
        }

        private static void GetInterfaceAttributeDemo() {
            var attribute1 = (MyCustomAttribute)
                typeof(MyClass).GetCustomAttribute(typeof(MyCustomAttribute), true);
            Console.WriteLine(attribute1 == null); // True
        }
    }
}
```

```
typeof(MyClass).GetCustomAttributes(true).OfType<MyCustomAttribute>().SingleOrDefault();
Console.WriteLine(attribute2 == null); // True

var attribute3 = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>(true);
Console.WriteLine(attribute3 == null); // True
}
}
```

获取接口属性的一种方法是通过类实现的所有接口进行搜索。

```
var attribute = typeof(MyClass).GetInterfaces().SelectMany(x =>
x.GetCustomAttributes().OfType<MyCustomAttribute>()).SingleOrDefault();
Console.WriteLine(attribute == null); // False
Console.WriteLine(attribute.Text); // Hello from interface attribute
```

```
typeof(MyClass).GetCustomAttributes(true).OfType<MyCustomAttribute>().SingleOrDefault();
Console.WriteLine(attribute2 == null); // True

var attribute3 = typeof(MyClass).GetCustomAttribute<MyCustomAttribute>(true);
Console.WriteLine(attribute3 == null); // True
}
}
```

One way to retrieve interface attributes is to search for them through all the interfaces implemented by a class.

```
var attribute = typeof(MyClass).GetInterfaces().SelectMany(x =>
x.GetCustomAttributes().OfType<MyCustomAttribute>()).SingleOrDefault();
Console.WriteLine(attribute == null); // False
Console.WriteLine(attribute.Text); // Hello from interface attribute
```

第96章：委托

第96.1节：声明委托类型

以下语法创建了一个名为NumberInOutDelegate的delegate类型，表示一个接受一个int并返回一个int的方法。

```
public delegate int NumberInOutDelegate(int input);
```

可以按如下方式使用：

```
public static class Program
{
    static void Main()
    {
        NumberInOutDelegate square = MathDelegates.Square;
        int answer1 = square(4);
        Console.WriteLine(answer1); // 将输出16

        NumberInOutDelegate cube = MathDelegates.Cube;
        int answer2 = cube(4);
        Console.WriteLine(answer2); // 将输出64
    }
}

public static class MathDelegates
{
    static int Square (int x)
    {
        return x*x;
    }

    static int Cube (int x)
    {
        return x*x*x;
    }
}
```

示例委托实例的执行方式与Square方法相同。委托实例实际上充当调用者的代理：调用者调用委托，然后委托调用目标方法。这种间接调用将调用者与目标方法解耦。

您可以声明一个泛型委托类型，在这种情况下，您可以指定某些类型参数是协变的（out）或逆变的（in）。例如：

```
public delegate TTo Converter<in TFrom, out TTo>(TFrom input);
```

像其他泛型类型一样，泛型委托类型可以有约束，例如where TFrom : struct, IConvertible where TTo : new()。

避免对用于多播委托（如事件处理程序类型）的委托类型使用协变和逆变。这是因为如果运行时类型与编译时类型因变而不同，连接（+）可能会失败。例如，避免：

```
public delegate void EventHandler<in TEventArgs>(object sender, TEventArgs e);
```

Chapter 96: Delegates

Section 96.1: Declaring a delegate type

The following syntax creates a delegate type with name NumberInOutDelegate, representing a method which takes an int and returns an int.

```
public delegate int NumberInOutDelegate(int input);
```

This can be used as follows:

```
public static class Program
{
    static void Main()
    {
        NumberInOutDelegate square = MathDelegates.Square;
        int answer1 = square(4);
        Console.WriteLine(answer1); // Will output 16

        NumberInOutDelegate cube = MathDelegates.Cube;
        int answer2 = cube(4);
        Console.WriteLine(answer2); // Will output 64
    }
}

public static class MathDelegates
{
    static int Square (int x)
    {
        return x*x;
    }

    static int Cube (int x)
    {
        return x*x*x;
    }
}
```

The example delegate instance is executed in the same way as the Square method. A delegate instance literally acts as a delegate for the caller: the caller invokes the delegate, and then the delegate calls the target method. This indirection decouples the caller from the target method.

You can declare a generic delegate type, and in that case you may specify that the type is covariant (out) or contravariant (in) in some of the type arguments. For example:

```
public delegate TTo Converter<in TFrom, out TTo>(TFrom input);
```

Like other generic types, generic delegate types can have constraints, such as where TFrom : struct, IConvertible where TTo : new().

Avoid co- and contravariance for delegate types that are meant to be used for multicast delegates, such as event handler types. This is because concatenation (+) can fail if the run-time type is different from the compile-time type because of the variance. For example, avoid:

```
public delegate void EventHandler<in TEventArgs>(object sender, TEventArgs e);
```

相反，使用不变的泛型类型：

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

还支持一些参数通过ref或out修饰的委托，例如：

```
public delegate bool TryParser<T>(string input, out T result);
```

(示例用法TryParser<decimal> example = decimal.TryParse;)，或者最后一个参数带有params修饰符的委托。委托类型可以有可选参数（提供默认值）。委托类型可以在其签名或返回类型中使用指针类型，如int*或char*（需使用unsafe关键字）。委托类型及其参数可以携带自定义属性。

第96.2节：Func<T, TResult>、Action<T>和Predicate<T>委托类型

System命名空间包含带有0到15个泛型参数的Func<..., TResult>委托类型，返回类型为TResult。

```
private void UseFunc(Func<string> func)
{
    string output = func(); // 带有单个泛型类型参数的Func返回该类型
    Console.WriteLine(output);
}

private void UseFunc(Func<int, int, string> func)
{
    string output = func(4, 2); // 带有多个泛型类型参数的Func，除第一个外，其他作为该类型的参数
    Console.WriteLine(output);
}
```

System 命名空间还包含 Action<...> 委托类型，具有不同数量的泛型参数（从 0 到 16）。它类似于 Func<T1, ..., Tn>，但它始终返回 void。

```
private void 使用Action(Action action)
{
    action(); // 非泛型的Action没有参数
}

private void 使用Action(Action<int, string> action)
{
    action(4, "two"); // 泛型Action以与其类型参数匹配的参数调用
}
```

Predicate<T> 也是一种 Func，但它总是返回 bool。Predicate是一种指定自定义条件的方式。根据输入的值和Predicate中定义的逻辑，它将返回 true 或false。 Predicate<T> 因此行为与 Func<T, bool> 相同，两者都可以以相同的方式初始化和使用。

```
Predicate<string> predicate = s => s.StartsWith("a");
Func<string, bool> func = s => s.StartsWith("a");

// 这两者都返回true
var predicateReturnsTrue = predicate("abc");
var funcReturnsTrue = func("abc");
```

Instead, use an invariant generic type:

```
public delegate void EventHandler<TEventArgs>(object sender, TEventArgs e);
```

Also supported are delegates where some parameters are modified by ref or out, as in:

```
public delegate bool TryParser<T>(string input, out T result);
```

(sample use TryParser<decimal> example = decimal.TryParse;)，或者最后一个参数带有params修饰符的委托。委托类型可以有可选参数（提供默认值）。委托类型可以在其签名或返回类型中使用指针类型，如int*或char*（需使用unsafe关键字）。委托类型及其参数可以携带自定义属性。

Section 96.2: The Func<T, TResult>, Action<T> and Predicate<T> delegate types

The System namespace contains Func<..., TResult> delegate types with between 0 and 15 generic parameters, returning type TResult.

```
private void UseFunc(Func<string> func)
{
    string output = func(); // Func with a single generic type parameter returns that type
    Console.WriteLine(output);
}

private void UseFunc(Func<int, int, string> func)
{
    string output = func(4, 2); // Func with multiple generic type parameters takes all but the
    first as parameters of that type
    Console.WriteLine(output);
}
```

The System namespace also contains Action<...> delegate types with different number of generic parameters (from 0 to 16). It is similar to Func<T1, ..., Tn>, but it always returns void.

```
private void UseAction(Action action)
{
    action(); // The non-generic Action has no parameters
}

private void UseAction(Action<int, string> action)
{
    action(4, "two"); // The generic action is invoked with parameters matching its type arguments
}
```

Predicate<T> is also a form of Func but it will always return bool. A predicate is a way of specifying a custom criteria. Depending on the value of the input and the logic defined within the predicate, it will return either true or false. Predicate<T> therefore behaves in the same way as Func<T, bool> and both can be initialized and used in the same way.

```
Predicate<string> predicate = s => s.StartsWith("a");
Func<string, bool> func = s => s.StartsWith("a");

// Both of these return true
var predicateReturnsTrue = predicate("abc");
var funcReturnsTrue = func("abc");
```

```
// 这两者都返回false
var predicateReturnsFalse = predicate("xyz");
var funcReturnsFalse = func("xyz");
```

是否使用Predicate<T>还是Func<T, bool>实际上是个个人观点的问题。 Predicate<T>可以说更能表达作者的意图，而Func<T, bool>则可能为更多C#开发者所熟悉。

除此之外，在某些情况下只有其中一个选项可用，尤其是在与其他API交互时。例如，List<T>和Array<T>通常在其方法中使用Predicate<T>，而大多数LINQ扩展只接受Func<T, bool>。

第96.3节：组合委托（多播委托）

加法+和减法-操作可用于组合委托实例。委托包含已分配委托的列表。

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace DelegatesExample {
    class MainClass {
        private delegate void MyDelegate(int a);

        private static void PrintInt(int a) {
            Console.WriteLine(a);
        }

        private static void PrintType<T>(T a) {
            Console.WriteLine(a.GetType());
        }

        public static void Main (string[] args)
        {
            MyDelegate d1 = PrintInt;
            MyDelegate d2 = PrintType;

            // 输出：
            // 1
            d1(1);

            // 输出：
            // System.Int32
            d2(1);

            MyDelegate d3 = d1 + d2;
            // 输出：
            // 1
            // System.Int32
            d3(1);

            MyDelegate d4 = d3 - d2;
            // 输出：
            // 1
            d4(1);

            // 输出：
            // True
            Console.WriteLine(d1 == d4);
        }
    }
}
```

```
// Both of these return false
var predicateReturnsFalse = predicate("xyz");
var funcReturnsFalse = func("xyz");
```

The choice of whether to use Predicate<T> or Func<T, bool> is really a matter of opinion. Predicate<T> is arguably more expressive of the author's intent, while Func<T, bool> is likely to be familiar to a greater proportion of C# developers.

In addition to that, there are some cases where only one of the options is available, especially when interacting with another API. For example List<T> and Array<T> generally take Predicate<T> for their methods, while most LINQ extensions only accept Func<T, bool>.

Section 96.3: Combine Delegates (Multicast Delegates)

Addition + and subtraction - operations can be used to combine delegate instances. The delegate contains a list of the assigned delegates.

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace DelegatesExample {
    class MainClass {
        private delegate void MyDelegate(int a);

        private static void PrintInt(int a) {
            Console.WriteLine(a);
        }

        private static void PrintType<T>(T a) {
            Console.WriteLine(a.GetType());
        }

        public static void Main (string[] args)
        {
            MyDelegate d1 = PrintInt;
            MyDelegate d2 = PrintType;

            // Output:
            // 1
            d1(1);

            // Output:
            // System.Int32
            d2(1);

            MyDelegate d3 = d1 + d2;
            // Output:
            // 1
            // System.Int32
            d3(1);

            MyDelegate d4 = d3 - d2;
            // Output:
            // 1
            d4(1);

            // Output:
            // True
            Console.WriteLine(d1 == d4);
        }
    }
}
```

```
    }
}
```

在此示例中，d3 是 d1 和 d2 委托的组合，因此调用时程序会输出 1 和 System.Int32 字符串。

结合具有 非 void 返回类型的委托：

如果多播委托具有 非 void 返回类型，调用者将接收最后一个被调用方法的返回值。前面的所有方法仍会被调用，但它们的返回值会被丢弃。

```
class 程序
{
    public delegate int Transformer(int x);

    static void Main(string[] args)
    {
Transformer t = Square;
        t += Cube;
Console.WriteLine(t(2)); // 输出 8
    }

    static int Square(int x) { return x * x; }

    static int Cube(int x) { return x*x*x; }
}
```

t(2) 会先调用 Square，然后调用 Cube。Square 的返回值会被丢弃，最后一个方法即 Cube 的返回值会被保留。

第96.4节：安全调用多播委托

是否曾想调用多播委托，但希望即使链中的某个调用发生异常，整个调用列表仍能全部执行完毕？那么你很幸运，我创建了一个扩展方法，正是实现了这一点，且仅在整个列表执行完成后抛出一个 AggregateException（聚合异常）：

```
public static class DelegateExtensions
{
    public static void SafeInvoke(this Delegate del, params object[] args)
    {
        var exceptions = new List<Exception>();

        foreach (var handler in del.GetInvocationList())
        {
            try
            {
handler.Method.Invoke(handler.Target, args);
            }
            catch (Exception ex)
            {
                exceptions.Add(ex);
            }
        }

        if(exceptions.Any())
        {
            throw new AggregateException(exceptions);
        }
    }
}
```

```
    }
}
```

In this example d3 is a combination of d1 and d2 delegates, so when called the program outputs both 1 and System.Int32 strings.

Combining delegates with **non void** return types:

If a multicast delegate has a nonvoid return type, the caller receives the return value from the last method to be invoked. The preceding methods are still called, but their return values are discarded.

```
class Program
{
    public delegate int Transformer(int x);

    static void Main(string[] args)
    {
        Transformer t = Square;
        t += Cube;
        Console.WriteLine(t(2)); // O/P 8
    }

    static int Square(int x) { return x * x; }

    static int Cube(int x) { return x*x*x; }
}
```

t(2) will call first Square and then Cube. The return value of Square is discarded and return value of the last method i.e. Cube is retained.

Section 96.4: Safe invoke multicast delegate

Ever wanted to call a multicast delegate but you want the entire invocation list to be called even if an exception occurs in any in the chain. Then you are in luck, I have created an extension method that does just that, throwing an AggregateException only after execution of the entire list completes:

```
public static class DelegateExtensions
{
    public static void SafeInvoke(this Delegate del, params object[] args)
    {
        var exceptions = new List<Exception>();

        foreach (var handler in del.GetInvocationList())
        {
            try
            {
                handler.Method.Invoke(handler.Target, args);
            }
            catch (Exception ex)
            {
                exceptions.Add(ex);
            }
        }

        if(exceptions.Any())
        {
            throw new AggregateException(exceptions);
        }
    }
}
```

```

    }

public class Test
{
    public delegate void SampleDelegate();

    public void Run()
    {
        SampleDelegate delegateInstance = this.Target2;
        delegateInstance += this.Target1;

        try
        {
            delegateInstance.SafeInvoke();
        }
        catch(AggregateException ex)
        {
            // 在这里进行任何异常处理
        }
    }

    private void Target1()
    {
        Console.WriteLine("Target 1 executed");
    }

    private void Target2()
    {
        Console.WriteLine("Target 2 executed");
        throw new Exception();
    }
}

```

这将输出：

```

目标 2 已执行
目标 1 已执行

```

直接调用，不使用 SaveInvoke，只会执行目标 2。

第 96.5 节：委托的相等性

调用 .Equals() 方法比较委托时，是按引用相等比较的：

```

Action action1 = () => Console.WriteLine("Hello delegates");
Action action2 = () => Console.WriteLine("Hello delegates");
Action action1Again = action1;

Console.WriteLine(action1.Equals(action1)) // True
Console.WriteLine(action1.Equals(action2)) // False
Console.WriteLine(action1Again.Equals(action1)) // True

```

这些规则同样适用于对多播委托使用 += 或 -=，例如订阅和取消订阅事件时。

第 96.6 节：命名方法的底层引用

```

    }

public class Test
{
    public delegate void SampleDelegate();

    public void Run()
    {
        SampleDelegate delegateInstance = this.Target2;
        delegateInstance += this.Target1;

        try
        {
            delegateInstance.SafeInvoke();
        }
        catch(AggregateException ex)
        {
            // Do any exception handling here
        }
    }

    private void Target1()
    {
        Console.WriteLine("Target 1 executed");
    }

    private void Target2()
    {
        Console.WriteLine("Target 2 executed");
        throw new Exception();
    }
}

```

This outputs:

```

Target 2 executed
Target 1 executed

```

Invoking directly, without SaveInvoke, would only execute Target 2.

Section 96.5: Delegate Equality

Calling .Equals() on a delegate compares by reference equality:

```

Action action1 = () => Console.WriteLine("Hello delegates");
Action action2 = () => Console.WriteLine("Hello delegates");
Action action1Again = action1;

Console.WriteLine(action1.Equals(action1)) // True
Console.WriteLine(action1.Equals(action2)) // False
Console.WriteLine(action1Again.Equals(action1)) // True

```

These rules also apply when doing += or -= on a multicast delegate, for example when subscribing and unsubscribing from events.

Section 96.6: Underlying references of named method

委托

当将具名方法分配给委托时，如果满足以下条件，它们将引用相同的底层对象：

- 它们是在同一个类的同一个实例上的相同实例方法
- 它们是在同一个类上的相同静态方法

```
public class Greeter
{
    public void WriteInstance()
    {
        Console.WriteLine("Instance");
    }

    public static void WriteStatic()
    {
        Console.WriteLine("Static");
    }
}

// ...

Greeter greeter1 = new Greeter();
Greeter greeter2 = new Greeter();

Action instance1 = greeter1.WriteInstance;
Action instance2 = greeter2.WriteInstance;
Action instance1Again = greeter1.WriteInstance;

Console.WriteLine(instance1.Equals(instance2)); // False
Console.WriteLine(instance1.Equals(instance1Again)); // True

Action @static = Greeter.WriteStatic;
Action staticAgain = Greeter.WriteStatic;

Console.WriteLine(@static.Equals(staticAgain)); // True
```

delegates

When assigning named methods to delegates, they will refer to the same underlying object if:

- They are the same instance method, on the same instance of a class
- They are the same static method on a class

```
public class Greeter
{
    public void WriteInstance()
    {
        Console.WriteLine("Instance");
    }

    public static void WriteStatic()
    {
        Console.WriteLine("Static");
    }
}

// ...

Greeter greeter1 = new Greeter();
Greeter greeter2 = new Greeter();

Action instance1 = greeter1.WriteInstance;
Action instance2 = greeter2.WriteInstance;
Action instance1Again = greeter1.WriteInstance;

Console.WriteLine(instance1.Equals(instance2)); // False
Console.WriteLine(instance1.Equals(instance1Again)); // True

Action @static = Greeter.WriteStatic;
Action staticAgain = Greeter.WriteStatic;

Console.WriteLine(@static.Equals(staticAgain)); // True
```

第96.7节：将具名方法分配给委托

具有匹配签名的方法可以分配给委托：

```
public static class Example
{
    public static int AddOne(int input)
    {
        return input + 1;
    }
}
```

```
Func<int,int> addOne = Example.AddOne
```

Example.AddOne 接受一个 int 并返回一个 int，其签名与委托 Func<int,int> 匹配。
Example.AddOne 可以直接赋值给 addOne，因为它们的签名匹配。

Section 96.7: Assigning a named method to a delegate

Named methods can be assigned to delegates with matching signatures:

```
public static class Example
{
    public static int AddOne(int input)
    {
        return input + 1;
    }
}
```

```
Func<int,int> addOne = Example.AddOne
```

Example.AddOne takes an int and returns an int, its signature matches the delegate Func<int,int>. Example.AddOne can be directly assigned to addOne because they have matching signatures.

第96.8节：通过lambda赋值给委托

可以使用lambda创建匿名方法并赋值给委托：

```
Func<int,int> addOne = x => x+1;
```

注意，当以这种方式创建变量时，必须显式声明类型：

```
var addOne = x => x+1; // 无效
```

第96.9节：在函数中封装转换

```
public class MyObject{
    public DateTime? TestDate { get; set; }

    public Func<MyObject, bool> DateIsValid = myObject => myObject.TestDate.HasValue &&
myObject.TestDate > DateTime.Now;

    public void DoSomething(){
        //我们可以这样做：
        if(this.TestDate.HasValue && this.TestDate > DateTime.Now){
            CallAnotherMethod();
        }

        //或者这样：
        if(DateIsValid(this)){
            CallAnotherMethod();
        }
    }
}
```

秉持简洁编码的精神，将上述类似的检查和转换封装为一个 Func，可以使你的代码更易于阅读和理解。虽然上述示例非常简单，但如果多个 DateTime 属性，每个属性都有不同的验证规则，并且我们想要检查不同的组合怎么办？简单来说，每个具有既定返回逻辑的一行 Func 既可读性强，又能减少代码的表面复杂度。请考虑下面的 Func 调用，想象一下如果没有它们，方法中会有多少代码杂乱无章：

```
public void CheckForIntegrity(){
    if(ShipDateIsValid(this) && TestResultsHaveBeenCalled(this) && !TestResultsFail(this)){
        SendPassingTestNotification();
    }
}
```

第96.10节：作为参数传递委托

委托可以用作类型化的函数指针：

```
class FuncAsParameters
{
    public void Run()
    {
        DoSomething(ErrorHandler1);
        DoSomething(ErrorHandler2);
    }

    public bool ErrorHandler1(string message)
    {
    }
}
```

Section 96.8: Assigning to a delegate by lambda

Lambdas can be used to create anonymous methods to assign to a delegate:

```
Func<int,int> addOne = x => x+1;
```

Note that the explicit declaration of type is required when creating a variable this way:

```
var addOne = x => x+1; // Does not work
```

Section 96.9: Encapsulating transformations in funcs

```
public class MyObject{
    public DateTime? TestDate { get; set; }

    public Func<MyObject, bool> DateIsValid = myObject => myObject.TestDate.HasValue &&
myObject.TestDate > DateTime.Now;

    public void DoSomething(){
        //We can do this:
        if(this.TestDate.HasValue && this.TestDate > DateTime.Now){
            CallAnotherMethod();
        }

        //or this:
        if(DateIsValid(this)){
            CallAnotherMethod();
        }
    }
}
```

In the spirit of clean coding, encapsulating checks and transformations like the one above as a Func can make your code easier to read and understand. While the above example is very simple, what if there were multiple DateTime properties each with their own differing validation rules and we wanted to check different combinations? Simple, one-line Funcs that each have established return logic can be both readable and reduce the apparent complexity of your code. Consider the below Func calls and imagine how much more code would be cluttering up the method:

```
public void CheckForIntegrity(){
    if(ShipDateIsValid(this) && TestResultsHaveBeenCalled(this) && !TestResultsFail(this)){
        SendPassingTestNotification();
    }
}
```

Section 96.10: Passing delegates as parameters

Delegates can be used as typed function pointers:

```
class FuncAsParameters
{
    public void Run()
    {
        DoSomething(ErrorHandler1);
        DoSomething(ErrorHandler2);
    }

    public bool ErrorHandler1(string message)
    {
    }
}
```

```

Console.WriteLine(message);
var shouldWeContinue = ...;
return shouldWeContinue;
}

public bool ErrorHandler2(string message)
{
    // ...将消息写入文件...
    var shouldWeContinue = ...;
    return shouldWeContinue;
}

public void DoSomething(Func<string, bool> errorHandler)
{
    // 在这里，我们不关心传入的处理程序是什么！
    ...
    if (...error...)
    {
        if (!errorHandler("发生了一些错误！"))
        {
            // 处理程序决定我们不能继续
            return;
        }
    }
}

```

第96.11节：委托内部的闭包

闭包是内联匿名方法，能够使用父方法的变量以及定义在父作用域中的其他匿名方法。

本质上，闭包是一段代码块，可以在以后执行，但它会保持最初创建时的环境——即使创建它的方法已经执行完毕，它仍然可以使用该方法的局部变量等。-- Jon Skeet

```

delegate int testDel();
static void Main(string[] args)
{
    int foo = 4;
    testDel myClosure = delegate()
    {
        return foo;
    };
    int bar = myClosure();
}

```

示例取自[Closures in .NET](#)。

```

Console.WriteLine(message);
var shouldWeContinue = ...;
return shouldWeContinue;
}

public bool ErrorHandler2(string message)
{
    // ...Write message to file...
    var shouldWeContinue = ...;
    return shouldWeContinue;
}

public void DoSomething(Func<string, bool> errorHandler)
{
    // In here, we don't care what handler we got passed!
    ...
    if (...error...)
    {
        if (!errorHandler("Some error occurred!"))
        {
            // The handler decided we can't continue
            return;
        }
    }
}

```

Section 96.11: Closure inside a delegate

Closures are inline anonymous methods that have the ability to use Parent method variables and other anonymous methods which are defined in the parent's scope.

In essence, a closure is a block of code which can be executed at a later time, but which maintains the environment in which it was first created - i.e. it can still use the local variables etc of the method which created it, even after that method has finished executing. -- **Jon Skeet**

```

delegate int testDel();
static void Main(string[] args)
{
    int foo = 4;
    testDel myClosure = delegate()
    {
        return foo;
    };
    int bar = myClosure();
}

```

Example taken from [Closures in .NET](#).

第97章：文件和流输入输出

参数

	详情
路径	文件的位置。
追加	如果文件存在, true 将在文件末尾添加数据(追加), false 将覆盖文件。
文本	要写入或存储的文本。
内容	要写入的字符串集合。
源	您想使用的文件的位置。
目标	您希望文件保存的位置。

管理文件。

第97.1节：使用 System.IO.File 类从文件读取

您可以使用System.IO.File.ReadAllText函数将整个文件内容读取到字符串中。

```
string text = System.IO.File.ReadAllText(@"C:\MyFolder\MyTextFile.txt");
```

您也可以使用System.IO.File.ReadAllLines函数将文件读取为行数组：

```
string[] lines = System.IO.File.ReadAllLines(@"C:\MyFolder\MyTextFile.txt");
```

第97.2节：通过

IEnumerable<string>懒惰地逐行读取文件

处理大文件时，您可以使用System.IO.File.ReadLines方法将文件的所有行读取到一个IEnumerable<string>中。这类似于System.IO.File.ReadAllText，但它不会一次性将整个文件加载到内存中，因此在处理大文件时更高效。

```
IEnumerable<string> AllLines = File.ReadLines("file_name.txt", Encoding.Default);
```

File.ReadLines 的第二个参数是可选的。当需要指定编码时，可以使用该参数。

需要注意的是，调用ToArray、ToList或其他类似函数会强制一次性加载所有行，这意味着使用ReadLines的优势将被抵消。使用此方法时，最好通过foreach循环或LINQ枚举IEnumerable。

第97.3节：使用StreamWriter异步写入文本到文件

```
// filename是包含完整路径的字符串  
// true表示追加写入  
using (System.IO.StreamWriter file = new System.IO.StreamWriter(filename, true))  
{  
    // 可以写入字符串或字符数组  
    await file.WriteLineAsync(text);  
}
```

第97.4节：复制文件

File静态类

File静态类可以很方便地用于此目的。

Chapter 97: File and Stream I/O

Parameter

path	The location of the file.
append	If the file exist, true will add data to the end of the file (append), false will overwrite the file.
text	Text to be written or stored.
contents	A collection of strings to be written.
source	The location of the file you want to use.
dest	The location you want a file to go to.

Manages files.

Section 97.1: Reading from a file using the System.IO.File class

You can use the [System.IO.File.ReadAllText](#) function to read the entire contents of a file into a string.

```
string text = System.IO.File.ReadAllText(@"C:\MyFolder\MyTextFile.txt");
```

You can also read a file as an array of lines using the [System.IO.File.ReadAllLines](#) function:

```
string[] lines = System.IO.File.ReadAllLines(@"C:\MyFolder\MyTextFile.txt");
```

Section 97.2: Lazily reading a file line-by-line via an IEnumerable

When working with large files, you can use the [System.IO.File.ReadLines](#) method to read all lines from a file into an [IEnumerable<string>](#). This is similar to [System.IO.File.ReadAllText](#), except that it doesn't load the whole file into memory at once, making it more efficient when working with large files.

```
IEnumerable<string> AllLines = File.ReadLines("file_name.txt", Encoding.Default);
```

The second parameter of [File.ReadLines](#) is optional. You may use it when it is required to specify encoding.

It is important to note that calling [ToArray](#), [ToList](#) or another similar function will force all of the lines to be loaded at once, meaning that the benefit of using [ReadLines](#) is nullified. It is best to enumerate over the [IEnumerable](#) using a [foreach](#) loop or LINQ if using this method.

Section 97.3: Async write text to a file using StreamWriter

```
// filename is a string with the full path  
// true is to append  
using (System.IO.StreamWriter file = new System.IO.StreamWriter(filename, true))  
{  
    // Can write either a string or char array  
    await file.WriteLineAsync(text);  
}
```

Section 97.4: Copy File

File static class

File static class can be easily used for this purpose.

```
File.Copy(@"sourcePath\abc.txt", @"destinationPath\abc.txt");
File.Copy(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

备注：通过此方法，文件是被复制的，意味着它会先从源读取，然后写入目标路径。这是一个资源消耗较大的过程，所需时间与文件大小成正比，如果不使用线程，可能会导致程序冻结。

第97.5节：使用 System.IO.StreamWriter类向文件写入行

System.IO.StreamWriter 类：

实现了一个TextWriter，用于以特定编码向流写入字符。

使用WriteLine方法，可以逐行向文件写入内容。

注意使用了using关键字，它确保StreamWriter对象在超出作用域时被释放，从而关闭文件。

```
string[] lines = { "我的第一个字符串", "我的第二个字符串", "甚至第三个字符串" };
using (System.IO.StreamWriter sw = new System.IO.StreamWriter(@"C:\MyFolder\OutputText.txt"))
{
    foreach (string line in lines)
    {
        sw.WriteLine(line);
    }
}
```

注意StreamWriter的构造函数可以接收第二个bool参数，允许Append到文件而不是覆盖文件：

```
bool appendExistingFile = true;
using (System.IO.StreamWriter sw = new System.IO.StreamWriter(@"C:\MyFolder\OutputText.txt",
appendExistingFile ))
{
    sw.WriteLine("这行将被追加到现有文件中");
}
```

第97.6节：使用 System.IO.File 类写入文件

你可以使用 [System.IO.File.WriteAllText](#) 函数将字符串写入文件。

```
string text = "将存储在文件中的字符串";
System.IO.File.WriteAllText(@"C:\MyFolder\OutputFile.txt", text);
```

你也可以使用 [System.IO.File.WriteAllLines](#) 函数，该函数接收一个 `IEnumerable<String>` 作为第二个参数（与前一个示例中的单个字符串相反）。这允许你从字符串数组写入内容。

```
string[] lines = { "我的第一个字符串", "我的第二个字符串", "甚至第三个字符串" };
System.IO.File.WriteAllLines(@"C:\MyFolder\OutputFile.txt", lines);
```

```
File.Copy(@"sourcePath\abc.txt", @"destinationPath\abc.txt");
File.Copy(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

Remark: By this method, file is copied, meaning that it will be read from the source and then written to the destination path. This is a resource consuming process, it would take relative time to the file size, and can cause your program to freeze if you don't utilize threads.

Section 97.5: Writing lines to a file using the System.IO.StreamWriter class

The [System.IO.StreamWriter](#) class:

Implements a TextWriter for writing characters to a stream in a particular encoding.

Using the `WriteLine` method, you can write content line-by-line to a file.

Notice the use of the `using` keyword which makes sure the StreamWriter object is disposed as soon as it goes out of scope and thus the file is closed.

```
string[] lines = { "My first string", "My second string", "and even a third string" };
using (System.IO.StreamWriter sw = new System.IO.StreamWriter(@"C:\MyFolder\OutputText.txt"))
{
    foreach (string line in lines)
    {
        sw.WriteLine(line);
    }
}
```

Note that the StreamWriter can receive a second `bool` parameter in its constructor, allowing to Append to a file instead of overwriting the file:

```
bool appendExistingFile = true;
using (System.IO.StreamWriter sw = new System.IO.StreamWriter(@"C:\MyFolder\OutputText.txt",
appendExistingFile ))
{
    sw.WriteLine("This line will be appended to the existing file");
}
```

Section 97.6: Writing to a file using the System.IO.File class

You can use the [System.IO.File.WriteAllText](#) function to write a string to a file.

```
string text = "String that will be stored in the file";
System.IO.File.WriteAllText(@"C:\MyFolder\OutputFile.txt", text);
```

You can also use the [System.IO.File.WriteAllLines](#) function which receives an `IEnumerable<String>` as the second parameter (as opposed to a single string in the previous example). This lets you write content from an array of lines.

```
string[] lines = { "My first string", "My second string", "and even a third string" };
System.IO.File.WriteAllLines(@"C:\MyFolder\OutputFile.txt", lines);
```

第97.7节：创建文件

File静态类

通过使用 File 静态类的 Create 方法，我们可以创建文件。该方法在指定路径创建文件，同时打开文件并返回文件的 FileStream。确保在完成操作后关闭文件。

示例1：

```
var fileStream1 = File.Create("samplePath");
/// 你可以向 fileStream1 写入数据
fileStream1.Close();
```

示例2：

```
using(var fileStream1 = File.Create("samplePath"))
{
    /// 你可以向 fileStream1 写入数据
}
```

示例3：

```
File.Create("samplePath").Close();
```

FileStream 类

这个类的构造函数有很多重载，文档中有详细说明[here](#)。下面的示例是针对该类最常用功能的构造函数。

```
var fileStream2 = new FileStream("samplePath", FileMode.OpenOrCreate, FileAccess.ReadWrite,
FileShare.None);
```

你可以通过这些链接查看 [FileMode](#)、 [FileAccess](#)和 [FileShare](#)的枚举。它们基本上的含义如下：

[FileMode](#): 回答“文件应该被创建？打开？如果不存在则创建然后打开？”这类问题。

[FileAccess](#): 回答“我是否应该能够读取文件、写入文件或两者都能？”这类问题。

[FileShare](#): 回答“当我同时使用文件时，其他用户是否能够读取、写入等？”这类问题。

第97.8节：移动文件

File静态类

可以轻松使用File静态类来实现此目的。

```
File.Move(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

备注1：仅更改文件的索引（如果文件在同一卷内移动）。此操作所需时间与文件大小无关。

备注2：不能覆盖目标路径上已存在的文件。

Section 97.7: Create File

File static class

By using Create method of the File static class we can create files. Method creates the file at the given path, at the same time it opens the file and gives us the FileStream of the file. Make sure you close the file after you are done with it.

ex1:

```
var fileStream1 = File.Create("samplePath");
/// you can write to the fileStream1
fileStream1.Close();
```

ex2:

```
using(var fileStream1 = File.Create("samplePath"))
{
    /// you can write to the fileStream1
}
```

ex3:

```
File.Create("samplePath").Close();
```

FileStream class

There are many overloads of this classes constructor which is actually well documented [here](#). Below example is for the one that covers most used functionalities of this class.

```
var fileStream2 = new FileStream("samplePath", FileMode.OpenOrCreate, FileAccess.ReadWrite,
FileShare.None);
```

You can check the enums for [FileMode](#), [FileAccess](#), and [FileShare](#) from those links. What they basically means are as follows:

[FileMode](#): Answers "Should file be created? opened? create if not exist then open?" kinda questions.

[FileAccess](#): Answers "Should I be able to read the file, write to the file or both?" kinda questions.

[FileShare](#): Answers "Should other users be able to read, write etc. to the file while I am using it simultaneously?" kinda questions.

Section 97.8: Move File

File static class

File static class can easily be used for this purpose.

```
File.Move(@"sourcePath\abc.txt", @"destinationPath\xyz.txt");
```

Remark1: Only changes the index of the file (if the file is moved in the same volume). This operation does not take relative time to the file size.

Remark2: Cannot override an existing file on destination path.

第97.9节：删除文件

```
string path = @"c:\path\file.txt";
File.Delete(path);
```

虽然Delete在文件不存在时不会抛出异常，但如果指定的路径无效或调用者没有所需权限，则会抛出异常。你应始终将对Delete的调用包裹在try-catch块中，并处理所有预期的异常。在可能发生竞态条件的情况下，应将逻辑包裹在lock语句中。

第97.10节：文件和目录

获取目录中的所有文件

```
var FileSearchRes = Directory.GetFiles(@Path, "*.*", SearchOption.AllDirectories);
```

返回一个FileInfo数组，表示指定目录中的所有文件。

获取特定扩展名的文件

```
var FileSearchRes = Directory.GetFiles(@Path, "*.pdf", SearchOption.AllDirectories);
```

返回一个FileInfo数组，表示指定目录中具有指定扩展名的所有文件。

Section 97.9: Delete File

```
string path = @"c:\path\to\file.txt";
File.Delete(path);
```

While Delete does not throw exception if file doesn't exist, it will throw exception e.g. if specified path is invalid or caller does not have the required permissions. You should always wrap calls to Delete inside try-catch block and handle all expected exceptions. In case of possible race conditions, wrap logic inside lock statement.

Section 97.10: Files and Directories

Get all files in Directory

```
var FileSearchRes = Directory.GetFiles(@Path, "*.*", SearchOption.AllDirectories);
```

Returns an array of FileInfo, representing all the files in the specified directory.

Get Files with specific extension

```
var FileSearchRes = Directory.GetFiles(@Path, "*.pdf", SearchOption.AllDirectories);
```

Returns an array of FileInfo, representing all the files in the specified directory with the specified extension.

第98章：网络

第98.1节：基础TCP通信客户端

此代码示例创建了一个TCP客户端，通过套接字连接发送“Hello World”，然后将服务器响应写入控制台，最后关闭连接。

```
// 声明变量
string host = "stackoverflow.com";
int port = 9999;
int timeout = 5000;

// 创建TCP客户端并连接
using (var _client = new TcpClient(host, port))
using (var _netStream = _client.GetStream())
{
    _netStream.ReadTimeout = timeout;

    // 通过套接字发送消息
    string message = "Hello World!";
    byte[] dataToSend = System.Text.Encoding.ASCII.GetBytes(message);
    _netStream.Write(dataToSend, 0, dataToSend.Length);

    // 读取服务器响应
    byte[] recvData = new byte[256];
    int bytes = _netStream.Read(recvData, 0, recvData.Length);
    message = System.Text.Encoding.ASCII.GetString(recvData, 0, bytes);
    Console.WriteLine(string.Format("Server: {0}", message));
};// 当控制流退出using块时，客户端和流将关闭（等同但比调用Close()更安全）
```

第98.2节：从网络服务器下载文件

从互联网下载文件是几乎每个你可能构建的应用程序都需要完成的非常常见的任务。

为此，你可以使用“System.Net.WebClient”类。

下面展示了使用“using”模式的最简单用法：

```
using (var webClient = new WebClient())
{
    webClient.DownloadFile("http://www.server.com/file.txt", "C:\\file.txt");
}
```

这个示例的作用是使用“using”确保在完成后正确清理你的网络客户端，并且简单地将第一个参数中的URL指定的资源传输到第二个参数中本地硬盘上的指定文件。

第一个参数的类型是“[System.Uri](#)”，第二个参数的类型是“[System.String](#)”

你也可以使用这个函数的异步形式，这样它会在后台执行下载，而你的应用程序可以继续处理其他事情，以这种方式调用在现代应用程序中非常重要，因为它有助于保持用户界面的响应性。

当你使用异步方法时，可以连接事件处理器来监控进度，这样

Chapter 98: Networking

Section 98.1: Basic TCP Communication Client

This code example creates a TCP client, sends "Hello World" over the socket connection, and then writes the server response to the console before closing the connection.

```
// Declare Variables
string host = "stackoverflow.com";
int port = 9999;
int timeout = 5000;

// Create TCP client and connect
using (var _client = new TcpClient(host, port))
using (var _netStream = _client.GetStream())
{
    _netStream.ReadTimeout = timeout;

    // Write a message over the socket
    string message = "Hello World!";
    byte[] dataToSend = System.Text.Encoding.ASCII.GetBytes(message);
    _netStream.Write(dataToSend, 0, dataToSend.Length);

    // Read server response
    byte[] recvData = new byte[256];
    int bytes = _netStream.Read(recvData, 0, recvData.Length);
    message = System.Text.Encoding.ASCII.GetString(recvData, 0, bytes);
    Console.WriteLine(string.Format("Server: {0}", message));
}// The client and stream will close as control exits the using block (Equivalent but safer than calling Close());
```

Section 98.2: Download a file from a web server

Downloading a file from the internet is a very common task required by almost every application you're likely to build.

To accomplish this, you can use the "[System.Net.WebClient](#)" class.

The simplest use of this, using the "using" pattern, is shown below:

```
using (var webClient = new WebClient())
{
    webClient.DownloadFile("http://www.server.com/file.txt", "C:\\file.txt");
}
```

What this example does is it uses "using" to make sure that your web client is cleaned up correctly when finished, and simply transfers the named resource from the URL in the first parameter, to the named file on your local hard drive in the second parameter.

The first parameter is of type "[System.Uri](#)", the second parameter is of type "[System.String](#)"

You can also use this function in an async form, so that it goes off and performs the download in the background, while your application gets on with something else, using the call in this way is of major importance in modern applications, as it helps to keep your user interface responsive.

When you use the Async methods, you can hook up event handlers that allow you to monitor the progress, so that

例如，你可以更新一个进度条，类似如下：

```
var webClient = new WebClient();
webClient.DownloadFileCompleted += new AsyncCompletedEventHandler(Completed);
webClient.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
webClient.DownloadFileAsync("http://www.server.com/file.txt", "C:\\file.txt");
```

然而，如果你使用异步版本，有一点非常重要需要记住，那就是“在‘using’语法中使用它们时要非常小心”。

原因很简单。一旦你调用下载文件方法，它会立即返回。如果你把它放在using块中，你会返回然后退出该块，立即释放类对象，从而取消正在进行的下载。

如果你使用‘using’方式执行异步传输，那么一定要确保在传输完成之前保持在该块内。

第98.3节：异步TCP客户端

在 C# 应用程序中使用async/await简化了多线程。这是如何将async/await与 TcpClient 结合使用的方法。

```
// 声明变量
string host = "stackoverflow.com";
int port = 9999;
int timeout = 5000;

// 创建 TCP 客户端并连接
// 然后获取网络流并传递
// 给我们的 StreamWriter 和 StreamReader
using (var client = new TcpClient())
using (var netstream = client.GetStream())
using (var writer = new StreamWriter(netstream))
using (var reader = new StreamReader(netstream))
{
    // 异步尝试连接服务器
    await client.ConnectAsync(host, port);

    // 自动刷新 StreamWriter
    // 以防止缓冲区溢出
    writer.AutoFlush = true;

    // 可选设置超时
    netstream.ReadTimeout = timeout;

    // 通过 TCP 连接发送消息
    string message = "Hello World!";
    await writer.WriteLineAsync(message);

    // 读取服务器响应
    string response = await reader.ReadLineAsync();
    Console.WriteLine(string.Format($"Server: {response}"));
}

// 客户端和流将在控制退出using块时关闭
// (等同但比调用Close()更安全)
```

you could for example, update a progress bar, something like the following:

```
var webClient = new WebClient();
webClient.DownloadFileCompleted += new AsyncCompletedEventHandler(Completed);
webClient.DownloadProgressChanged += new DownloadProgressChangedEventHandler(ProgressChanged);
webClient.DownloadFileAsync("http://www.server.com/file.txt", "C:\\file.txt");
```

One important point to remember if you use the Async versions however, and that's "Be very carefull about using them in a 'using' syntax".

The reason for this is quite simple. Once you call the download file method, it will return immediately. If you have this in a using block, you will return then exit that block, and immediately dispose the class object, and thus cancel your download in progress.

If you use the 'using' way of performing an Async transfer, then be sure to stay inside the enclosing block until the transfer completes.

Section 98.3: Async TCP Client

Using `async/await` in C# applications simplifies multi-threading. This is how you can use `async/await` in conjunction with a `TcpClient`.

```
// Declare Variables
string host = "stackoverflow.com";
int port = 9999;
int timeout = 5000;

// Create TCP client and connect
// Then get the netstream and pass it
// To our StreamWriter and StreamReader
using (var client = new TcpClient())
using (var netstream = client.GetStream())
using (var writer = new StreamWriter(netstream))
using (var reader = new StreamReader(netstream))
{
    // Asynchronously attempt to connect to server
    await client.ConnectAsync(host, port);

    // AutoFlush the StreamWriter
    // so we don't go over the buffer
    writer.AutoFlush = true;

    // Optionally set a timeout
    netstream.ReadTimeout = timeout;

    // Write a message over the TCP Connection
    string message = "Hello World!";
    await writer.WriteLineAsync(message);

    // Read server response
    string response = await reader.ReadLineAsync();
    Console.WriteLine(string.Format($"Server: {response}"));
}

// The client and stream will close as control exits
// the using block (Equivalent but safer than calling Close());
```

第98.4节：基础UDP客户端

此代码示例创建一个UDP客户端，然后将“Hello World”发送到网络上的目标接收方。监听器不必处于活动状态，因为UDP是无连接的，无论如何都会广播该消息。消息发送后，客户端的工作就完成了。

```
byte[] data = Encoding.ASCII.GetBytes("Hello World");
string ipAddress = "192.168.1.141";
string sendPort = 55600;
try
{
    using (var client = new UdpClient())
    {
        IPEndPoint ep = new IPEndPoint(IPAddress.Parse(ipAddress), sendPort);
        client.Connect(ep);
        client.Send(data, data.Length);
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}
```

下面是一个UDP监听器的示例，用于配合上述客户端。它将持续监听指定端口的流量，并将数据写入控制台。该示例包含一个控制标志'done'，该标志不会在内部设置，依赖外部某物来设置它，以便结束监听器并退出。

```
bool done = false;
int listenPort = 55600;
using(UdpClinet listener = new UdpClient(listenPort))
{
    IPEndPoint listenEndPoint = new IPEndPoint(IPAddress.Any, listenPort);
    while(!done)
    {
        byte[] receivedData = listener.Receive(ref listenPort);

        Console.WriteLine("Received broadcast message from client {0}", listenEndPoint.ToString());

        Console.WriteLine("Decoded data is:");
        Console.WriteLine(Encoding.ASCII.GetString(receivedData)); //应该是上面客户端发送的"Hello World"
    }
}
```

Section 98.4: Basic UDP Client

This code example creates a UDP client then sends "Hello World" across the network to the intended recipient. A listener does not have to be active, as UDP is connectionless and will broadcast the message regardless. Once the message is sent, the clients work is done.

```
byte[] data = Encoding.ASCII.GetBytes("Hello World");
string ipAddress = "192.168.1.141";
string sendPort = 55600;
try
{
    using (var client = new UdpClient())
    {
        IPEndPoint ep = new IPEndPoint(IPAddress.Parse(ipAddress), sendPort);
        client.Connect(ep);
        client.Send(data, data.Length);
    }
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}
```

Below is an example of a UDP listener to complement the above client. It will constantly sit and listen for traffic on a given port and simply write that data to the console. This example contains a control flag 'done' that is not set internally and relies on something to set this to allow for ending the listener and exiting.

```
bool done = false;
int listenPort = 55600;
using(UdpClinet listener = new UdpClient(listenPort))
{
    IPEndPoint listenEndPoint = new IPEndPoint(IPAddress.Any, listenPort);
    while(!done)
    {
        byte[] receivedData = listener.Receive(ref listenPort);

        Console.WriteLine("Received broadcast message from client {0}", listenEndPoint.ToString());

        Console.WriteLine("Decoded data is:");
        Console.WriteLine(Encoding.ASCII.GetString(receivedData)); //should be "Hello World" sent
from above client
    }
}
```

第99章：执行HTTP请求

第99.1节：创建并发送HTTP POST请求

```
using System.Net;
using System.IO;

...
string requestUrl = "https://www.example.com/submit.html";
HttpWebRequest request = HttpWebRequest.CreateHttp(requestUrl);
request.Method = "POST";

// 可选地，设置HttpWebRequest的属性，例如：
request.AutomaticDecompression = DecompressionMethods.Deflate | DecompressionMethods.GZip;
request.ContentType = "application/x-www-form-urlencoded";
// 也可以设置其他HTTP头，如Request.UserAgent、Request.Referer、
// Request.Accept，或通过Request.Headers集合设置其他头。

// 设置POST请求的正文数据。在此示例中，POST数据采用
// application/x-www-form-urlencoded格式。
string postData = "myparam1=myvalue1&myparam2=myvalue2";
using (var writer = new StreamWriter(request.GetRequestStream()))
{
    writer.Write(postData);
}

// 提交请求，并从远程服务器获取响应正文。
string responseFromRemoteServer;
using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
{
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        responseFromRemoteServer = reader.ReadToEnd();
    }
}
```

第99.2节：创建并发送HTTP GET请求

```
using System.Net;
using System.IO;

...
string requestUrl = "https://www.example.com/page.html";
HttpWebRequest request = HttpWebRequest.CreateHttp(requestUrl);

// 可选地，设置HttpWebRequest的属性，例如：
request.AutomaticDecompression = DecompressionMethods.GZip | DecompressionMethods.Deflate;
request.Timeout = 2 * 60 * 1000; // 2分钟，单位为毫秒

// 提交请求，并获取响应正文。
string responseBodyFromRemoteServer;
using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
{
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        responseBodyFromRemoteServer = reader.ReadToEnd();
    }
}
```

Chapter 99: Performing HTTP requests

Section 99.1: Creating and sending an HTTP POST request

```
using System.Net;
using System.IO;

...
string requestUrl = "https://www.example.com/submit.html";
HttpWebRequest request = HttpWebRequest.CreateHttp(requestUrl);
request.Method = "POST";

// Optionally, set properties of the HttpWebRequest, such as:
request.AutomaticDecompression = DecompressionMethods.Deflate | DecompressionMethods.GZip;
request.ContentType = "application/x-www-form-urlencoded";
// Could also set other HTTP headers such as Request.UserAgent, Request.Referer,
// Request.Accept, or other headers via the Request.Headers collection.

// Set the POST request body data. In this example, the POST data is in
// application/x-www-form-urlencoded format.
string postData = "myparam1=myvalue1&myparam2=myvalue2";
using (var writer = new StreamWriter(request.GetRequestStream()))
{
    writer.Write(postData);
}

// Submit the request, and get the response body from the remote server.
string responseFromRemoteServer;
using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
{
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        responseFromRemoteServer = reader.ReadToEnd();
    }
}
```

Section 99.2: Creating and sending an HTTP GET request

```
using System.Net;
using System.IO;

...
string requestUrl = "https://www.example.com/page.html";
HttpWebRequest request = HttpWebRequest.CreateHttp(requestUrl);

// Optionally, set properties of the HttpWebRequest, such as:
request.AutomaticDecompression = DecompressionMethods.GZip | DecompressionMethods.Deflate;
request.Timeout = 2 * 60 * 1000; // 2 minutes, in milliseconds

// Submit the request, and get the response body.
string responseBodyFromRemoteServer;
using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
{
    using (StreamReader reader = new StreamReader(response.GetResponseStream()))
    {
        responseBodyFromRemoteServer = reader.ReadToEnd();
    }
}
```

}

第99.3节：特定HTTP响应代码的错误处理 (例如404未找到)

使用 System.Net;

...

```

字符串 serverResponse;
尝试
{
    // 调用执行HTTP请求的方法 (参见上述示例)。
serverResponse = PerformHttpRequest();
}
捕获 (WebException ex)
{
    如果 (ex.Status == WebExceptionStatus.ProtocolError)
    {
        HttpWebResponse response = ex.Response as HttpWebResponse;
        如果 (response != null)
        {
            如果 ((int)response.StatusCode == 404) // 未找到
            {
                // 处理 404 未找到错误
                // ...
            }
            else
            {
                // 可以在这里处理其他 response.StatusCode 值。
                // ...
            }
        }
    }
    else
    {
        // 可以在这里处理其他错误情况，例如 WebExceptionStatus.ConnectFailure。
        // ...
    }
}

```

第 99.4 节：获取网页的 HTML (简单)

```

string contents = "";
string url = "http://msdn.microsoft.com";

using (System.Net.WebClient client = new System.Net.WebClient())
{
    contents = client.DownloadString(url);
}

Console.WriteLine(contents);

```

第 99.5 节：发送带有 JSON 正文的异步 HTTP POST 请求

```

public static async Task PostAsync(this Uri uri, object value)
{

```

}

Section 99.3: Error handling of specific HTTP response codes (such as 404 Not Found)

```

using System.Net;

...
string serverResponse;
try
{
    // Call a method that performs an HTTP request (per the above examples).
    serverResponse = PerformHttpRequest();
}
catch (WebException ex)
{
    if (ex.Status == WebExceptionStatus.ProtocolError)
    {
        HttpWebResponse response = ex.Response as HttpWebResponse;
        if (response != null)
        {
            if ((int)response.StatusCode == 404) // Not Found
            {
                // Handle the 404 Not Found error
                // ...
            }
            else
            {
                // Could handle other response.StatusCode values here.
                // ...
            }
        }
    }
    else
    {
        // Could handle other error conditions here, such as WebExceptionStatus.ConnectFailure.
        // ...
    }
}

```

Section 99.4: Retrieve HTML for Web Page (Simple)

```

string contents = "";
string url = "http://msdn.microsoft.com";

using (System.Net.WebClient client = new System.Net.WebClient())
{
    contents = client.DownloadString(url);
}

Console.WriteLine(contents);

```

Section 99.5: Sending asynchronous HTTP POST request with JSON body

```

public static async Task PostAsync(this Uri uri, object value)
{

```

```
var content = new ObjectContext(value.GetType(), value, new JsonMediaTypeFormatter());  
  
using (var client = new HttpClient())  
{  
    return await client.PostAsync(uri, content);  
}  
  
.  
  
var uri = new Uri("http://stackoverflow.com/documentation/c%23/1971/performing-http-requests");  
await uri.PostAsync(new { foo = 123.45, bar = "Richard Feynman" });
```

belindoc.com

```
var content = new ObjectContext(value.GetType(), value, new JsonMediaTypeFormatter());  
  
using (var client = new HttpClient())  
{  
    return await client.PostAsync(uri, content);  
}  
  
.  
  
var uri = new Uri("http://stackoverflow.com/documentation/c%23/1971/performing-http-requests");  
await uri.PostAsync(new { foo = 123.45, bar = "Richard Feynman" });
```

第100章：读取和写入.zip文件

参数

archiveFileName 要打开的归档文件路径。指定为相对路径或绝对路径。相对路径是解释为相对于当前工作目录。

详情

第100.1节：写入zip文件

写入新的.zip文件：

```
System.IO.Compression  
System.IO.Compression.FileSystem  
  
using ((FileStream zipToOpen = new FileStream(@"C:\emp", FileMode.Open)))  
{  
    using ((ZipArchive archive = new ZipArchive(zipToOpen, ZipArchiveMode.Update)))  
    {  
        ZipArchiveEntry readmeEntry = archive.CreateEntry("Readme.txt");  
        using ((StreamWriter writer = new StreamWriter(readmeEntry.Open())))  
        {  
            writer.WriteLine("Information about this package.");  
            writer.WriteLine("=====");  
        }  
    }  
}
```

第100.2节：内存中写入Zip文件

以下示例将返回包含所提供文件的压缩文件的byte[]数据，无需访问文件系统。

```
public static byte[] ZipFiles(Dictionary<string, byte[]> files)  
{  
    using ((MemoryStream ms = new MemoryStream())  
    {  
        using ((ZipArchive archive = new ZipArchive(ms, ZipArchiveMode.Update))  
        {  
            foreach (var file in files)  
            {  
                ZipArchiveEntry orderEntry = archive.CreateEntry(file.Key); //使用此名称创建文件  
  
                using (BinaryWriter writer = new BinaryWriter(orderEntry.Open()))  
                {  
                    writer.Write(file.Value); //写入二进制数据  
                }  
            }  
            //在 MemoryStream 有数据之前必须释放 ZipArchive  
            return ms.ToArray();  
        }  
    }  
}
```

第100.3节：从Zip文件获取文件

此示例从提供的zip归档二进制数据中获取文件列表：

```
public static Dictionary<string, byte[]> GetFiles(byte[] zippedFile)
```

Chapter 100: Reading and writing .zip files

Parameter

archiveFileName The path to the archive to open, specified as a relative or absolute path. A relative path is interpreted as relative to the current working directory.

Details

Section 100.1: Writing to a zip file

To write a new .zip file:

```
System.IO.Compression  
System.IO.Compression.FileSystem  
  
using (FileStream zipToOpen = new FileStream(@"C:\temp", FileMode.Open))  
{  
    using (ZipArchive archive = new ZipArchive(zipToOpen, ZipArchiveMode.Update))  
    {  
        ZipArchiveEntry readmeEntry = archive.CreateEntry("Readme.txt");  
        using (StreamWriter writer = new StreamWriter(readmeEntry.Open()))  
        {  
            writer.WriteLine("Information about this package.");  
            writer.WriteLine("=====");  
        }  
    }  
}
```

Section 100.2: Writing Zip Files in-memory

The following example will return the byte[] data of a zipped file containing the files provided to it, without needing access to the file system.

```
public static byte[] ZipFiles(Dictionary<string, byte[]> files)  
{  
    using (MemoryStream ms = new MemoryStream())  
    {  
        using (ZipArchive archive = new ZipArchive(ms, ZipArchiveMode.Update))  
        {  
            foreach (var file in files)  
            {  
                ZipArchiveEntry orderEntry = archive.CreateEntry(file.Key); //create a file with  
                this name  
                using (BinaryWriter writer = new BinaryWriter(orderEntry.Open()))  
                {  
                    writer.Write(file.Value); //write the binary data  
                }  
            }  
            //ZipArchive must be disposed before the MemoryStream has data  
            return ms.ToArray();  
        }  
    }  
}
```

Section 100.3: Get files from a Zip file

This example gets a listing of files from the provided zip archive binary data:

```
public static Dictionary<string, byte[]> GetFiles(byte[] zippedFile)
```

```

{
    using (MemoryStream ms = new MemoryStream(zippedFile))
    using (ZipArchive archive = new ZipArchive(ms, ZipArchiveMode.Read))
    {
        return archive.Entries.ToDictionary(x => x.FullName, x => ReadStream(x.Open()));
    }
}

private static byte[] ReadStream(Stream stream)
{
    using (var ms = new MemoryStream())
    {
        stream.CopyTo(ms);
        return ms.ToArray();
    }
}

```

第100.4节：以下示例展示了如何打开一个zip压缩包并将所有.txt文件提取到一个文件夹中

```

using System;
using System.IO;
using System.IO.Compression;

namespace ConsoleApplication1
{
    class 程序
    {
        static void Main(string[] args)
        {
            string zipPath = @"c:\example\start.zip";
            string extractPath = @"c:\example\extract";

            using (ZipArchive archive = ZipFile.OpenRead(zipPath))
            {
                foreach (ZipArchiveEntry entry in archive.Entries)
                {
                    if (entry.FullName.EndsWith(".txt", StringComparison.OrdinalIgnoreCase))
                    {
                        entry.ExtractToFile(Path.Combine(extractPath, entry.FullName));
                    }
                }
            }
        }
    }
}

```

```

{
    using (MemoryStream ms = new MemoryStream(zippedFile))
    using (ZipArchive archive = new ZipArchive(ms, ZipArchiveMode.Read))
    {
        return archive.Entries.ToDictionary(x => x.FullName, x => ReadStream(x.Open()));
    }
}

private static byte[] ReadStream(Stream stream)
{
    using (var ms = new MemoryStream())
    {
        stream.CopyTo(ms);
        return ms.ToArray();
    }
}

```

Section 100.4: The following example shows how to open a zip archive and extract all .txt files to a folder

```

using System;
using System.IO;
using System.IO.Compression;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string zipPath = @"c:\example\start.zip";
            string extractPath = @"c:\example\extract";

            using (ZipArchive archive = ZipFile.OpenRead(zipPath))
            {
                foreach (ZipArchiveEntry entry in archive.Entries)
                {
                    if (entry.FullName.EndsWith(".txt", StringComparison.OrdinalIgnoreCase))
                    {
                        entry.ExtractToFile(Path.Combine(extractPath, entry.FullName));
                    }
                }
            }
        }
    }
}

```

第101章：FileSystemWatcher

路径
要监视的目录，使用标准或通用命名规范（UNC）表示法。

过滤器
要监视的文件类型。例如，“*.txt”监视所有文本文件的更改。

第101.1节：IsFileReady

许多刚开始使用FileSystemWatcher的人常犯的一个错误是没有考虑到FileWatcher事件是在文件创建后立即触发的。然而，文件可能需要一些时间才能完成。

示例：

以1 GB的文件大小为例。该文件是由另一个程序（Explorer.exe从某处复制）创建的，但完成该过程需要几分钟。事件在创建时触发，您需要等待文件准备好才能复制。

这是检查文件是否准备好的方法。

```
public static bool IsFileReady(String sFilename)
{
    // 如果文件可以被独占访问打开，说明该文件不再被其他进程锁定。
    try
    {
        using (FileStream inputStream = File.Open(sFilename, FileMode.Open, FileAccess.Read,
FileShare.None))
        {
            if (inputStream.Length > 0)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
    catch (Exception)
    {
        return false;
    }
}
```

第101.2节：基础FileWatcher

下面的示例创建了一个FileSystemWatcher来监视运行时指定的目录。该组件设置为监视LastWrite和LastAccess时间的更改，以及目录中文本文件的创建、删除或重命名。如果文件被更改、创建或删除，文件路径将打印到控制台。当文件被重命名时，旧路径和新路径将打印到控制台。

此示例使用System.Diagnostics和System.IO命名空间。

```
FileSystemWatcher watcher;
```

Chapter 101: FileSystemWatcher

path
The directory to monitor, in standard or Universal Naming Convention (UNC) notation.

filter
The type of files to watch. For example, “*.txt” watches for changes to all text files.

Section 101.1: IsFileReady

A common mistake a lot of people starting out with FileSystemWatcher does is not taking into account That the FileWatcher event is raised as soon as the file is created. However, it may take some time for the file to be finished .

Example:

Take a file size of 1 GB for example . The file apr ask created by another program (Explorer.exe copying it from somewhere) but it will take minutes to finish that process. The event is raised that creation time and you need to wait for the file to be ready to be copied.

This is a method for checking if the file is ready.

```
public static bool IsFileReady(String sFilename)
{
    // If the file can be opened for exclusive access it means that the file
    // is no longer locked by another process.
    try
    {
        using (FileStream inputStream = File.Open(sFilename, FileMode.Open, FileAccess.Read,
FileShare.None))
        {
            if (inputStream.Length > 0)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
    catch (Exception)
    {
        return false;
    }
}
```

Section 101.2: Basic FileWatcher

The following example creates a FileSystemWatcher to watch the directory specified at run time. The component is set to watch for changes in **LastWrite** and **LastAccess** time, the creation, deletion, or renaming of text files in the directory. If a file is changed, created, or deleted, the path to the file prints to the console. When a file is renamed, the old and new paths print to the console.

Use the System.Diagnostics and System.IO namespaces for this example.

```
FileSystemWatcher watcher;
```

```

private void watch()
{
    // 创建一个新的 FileSystemWatcher 并设置其属性。
    watcher = new FileSystemWatcher();
    watcher.Path = path;

    /* 监视 LastAccess 和 LastWrite 时间的更改,
       以及文件或目录的重命名。 */
    watcher.NotifyFilter = NotifyFilters.LastAccess | NotifyFilters.LastWrite
        | NotifyFilters.FileName | NotifyFilters.DirectoryName;

    // 仅监视文本文件。
    watcher.Filter = "*.*";

    // 添加事件处理程序。
    watcher.Changed += new FileSystemEventHandler(OnChanged);
    // 开始监视。
    watcher.EnableRaisingEvents = true;
}

// 定义事件处理程序。
private void OnChanged(object source, FileSystemEventArgs e)
{
    // 将文件复制到另一个目录或执行其他操作。
    Console.WriteLine("文件: " + e.FullPath + " " + e.ChangeType);
}

```

```

private void watch()
{
    // Create a new FileSystemWatcher and set its properties.
    watcher = new FileSystemWatcher();
    watcher.Path = path;

    /* Watch for changes in LastAccess and LastWrite times, and
       the renaming of files or directories. */
    watcher.NotifyFilter = NotifyFilters.LastAccess | NotifyFilters.LastWrite
        | NotifyFilters.FileName | NotifyFilters.DirectoryName;

    // Only watch text files.
    watcher.Filter = "*.*";

    // Add event handler.
    watcher.Changed += new FileSystemEventHandler(OnChanged);
    // Begin watching.
    watcher.EnableRaisingEvents = true;
}

// Define the event handler.
private void OnChanged(object source, FileSystemEventArgs e)
{
    //Copies file to another directory or another action.
    Console.WriteLine("File: " + e.FullPath + " " + e.ChangeType);
}

```

第102章：使用用户名和密码访问网络共享文件夹

使用PInvoke访问网络共享文件。

第102.1节：访问网络共享文件的代码

```
public class NetworkConnection : IDisposable
{
    string _networkName;

    public NetworkConnection(string networkName,
        NetworkCredential credentials)
    {
        _networkName = networkName;

        var netResource = new NetResource()
        {
Scope = ResourceScope.GlobalNetwork,
            ResourceType = ResourceType.Disk,
            DisplayType = ResourceDisplaytype.Share,
            RemoteName = networkName
        };

        var userName = string.IsNullOrEmpty(credentials.Domain)
            ? credentials.UserName
            : string.Format("{0}\{1}", credentials.Domain, credentials.UserName);

        var result = WNetAddConnection2(
            netResource,
            credentials.Password,
            userName,
            0);

        if (result != 0)
        {
            throw new Win32Exception(result);
        }
    }

    ~NetworkConnection()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        WNetCancelConnection2(_networkName, 0, true);
    }

    [DllImport("mpr.dll")]
    private static extern int WNetAddConnection2(NetResource netResource,
        string password, string username, int flags);
}
```

Chapter 102: Access network shared folder with username and password

Accessing network share file using PInvoke.

Section 102.1: Code to access network shared file

```
public class NetworkConnection : IDisposable
{
    string _networkName;

    public NetworkConnection(string networkName,
        NetworkCredential credentials)
    {
        _networkName = networkName;

        var netResource = new NetResource()
        {
Scope = ResourceScope.GlobalNetwork,
            ResourceType = ResourceType.Disk,
            DisplayType = ResourceDisplaytype.Share,
            RemoteName = networkName
        };

        var userName = string.IsNullOrEmpty(credentials.Domain)
            ? credentials.UserName
            : string.Format("{0}\{1}", credentials.Domain, credentials.UserName);

        var result = WNetAddConnection2(
            netResource,
            credentials.Password,
            userName,
            0);

        if (result != 0)
        {
            throw new Win32Exception(result);
        }
    }

    ~NetworkConnection()
    {
        Dispose(false);
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        WNetCancelConnection2(_networkName, 0, true);
    }

    [DllImport("mpr.dll")]
    private static extern int WNetAddConnection2(NetResource netResource,
        string password, string username, int flags);
}
```

```

[DllImport("mpr.dll")]
private static extern int WNetCancelConnection2(string name, int flags,
    bool force);
}

[StructLayout(LayoutKind.Sequential)]
public class NetResource
{
    public ResourceScope 范围;
    public ResourceType 资源类型;
    public ResourceDisplaytype 显示类型;
    public int 使用;
    public string 本地名称;
    public string 远程名称;
    public string 注释;
    public string 提供者;
}

public enum ResourceScope : int
{
    已连接 = 1,
    全局网络,
    记忆,
    最近,
    上下文
};

public enum ResourceType : int
{
    任意 = 0,
    磁盘 = 1,
    打印 = 2,
    保留 = 8,
}

public enum ResourceDisplaytype : int
{
    通用 = 0x0,
    域 = 0x01,
    服务器 = 0x02,
    共享 = 0x03,
    文件 = 0x04,
    组 = 0x05,
    网络 = 0x06,
    根 = 0x07,
    共享管理员 = 0x08,
    目录 = 0x09,
    树 = 0x0a,
    Nds容器 = 0x0b
}

```

```

[DllImport("mpr.dll")]
private static extern int WNetCancelConnection2(string name, int flags,
    bool force);
}

[StructLayout(LayoutKind.Sequential)]
public class NetResource
{
    public ResourceScope Scope;
    public ResourceType ResourceType;
    public ResourceDisplaytype DisplayType;
    public int Usage;
    public string LocalName;
    public string RemoteName;
    public string Comment;
    public string Provider;
}

public enum ResourceScope : int
{
    Connected = 1,
    GlobalNetwork,
    Remembered,
    Recent,
    Context
};

public enum ResourceType : int
{
    Any = 0,
    Disk = 1,
    Print = 2,
    Reserved = 8,
}

public enum ResourceDisplaytype : int
{
    Generic = 0x0,
    Domain = 0x01,
    Server = 0x02,
    Share = 0x03,
    File = 0x04,
    Group = 0x05,
    Network = 0x06,
    Root = 0x07,
    Shareadmin = 0x08,
    Directory = 0x09,
    Tree = 0x0a,
    Ndscontainer = 0x0b
}

```

第103章：异步套接字

通过使用异步套接字，服务器可以监听传入连接的同时执行其他逻辑，这与同步套接字形成对比，后者在监听时会阻塞主线程，应用程序会变得无响应并冻结，直到有客户端连接。

第103.1节：异步套接字（客户端/服务器）示例

服务器端示例

创建服务器监听器

首先创建一个服务器，用于处理连接的客户端和将要发送的请求。因此，创建一个监听器类来处理这些。

```
class Listener
{
    public Socket ListenerSocket; //这是用于监听任何传入连接的套接字
    public short Port = 1234; //我们将在此端口监听

    public Listener()
    {
        ListenerSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
        ProtocolType.Tcp);
    }
}
```

首先需要初始化监听器套接字，以便监听任何连接。我们将使用Tcp套接字，这就是为什么使用SocketType.Stream。同时，我们指定服务器应监听的端口

然后开始监听任何传入的连接。

我们这里使用的三个方法是：

1. [ListenerSocket.Bind\(\);](#)

此方法将套接字绑定到一个IPPEndPoint。该类包含主机以及本地或远程端口信息，应用程序需要这些信息来连接主机上的服务。

2. [ListenerSocket.Listen\(10\);](#)

backlog参数指定可以排队等待接受的传入连接数。

3. [ListenerSocket.BeginAccept\(\);](#)

服务器将开始监听传入连接，并继续执行其他逻辑。当有连接时，服务器会切换回此方法并运行AcceptCallBack方法

```
public void StartListening()
{
```

Chapter 103: Asynchronous Socket

By using asynchronous sockets a server can listening for incoming connections and do some other logic in the mean time in contrast to synchronous socket when they are listening they block the main thread and the application is becoming unresponsive and will freeze until a client connects.

Section 103.1: Asynchronous Socket (Client / Server) example

Server Side example

Create Listener for server

Start off with creating a server that will handle clients that connect, and requests that will be send. So create an Listener Class that will handle this.

```
class Listener
{
    public Socket ListenerSocket; //This is the socket that will listen to any incoming connections
    public short Port = 1234; // on this port we will listen

    public Listener()
    {
        ListenerSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
        ProtocolType.Tcp);
    }
}
```

First we need to initialize the Listener socket where we can listen on for any connections. We are going to use an Tcp Socket that is why we use SocketType.Stream. Also we specify to which port the server should listen to

Then we start listening for any incoming connections.

The three methods we use here are:

1. [ListenerSocket.Bind\(\);](#)

This method binds the socket to an IPPEndPoint. This class contains the host and local or remote port information needed by an application to connect to a service on a host.

2. [ListenerSocket.Listen\(10\);](#)

The backlog parameter specifies the number of incoming connections that can be queued for acceptance.

3. [ListenerSocket.BeginAccept\(\);](#)

The server will start listening for incoming connections and will go on with other logic. When there is a connection the server switches back to this method and will run the AcceptCallBack method

```
public void StartListening()
{
```

```

try
{
    MessageBox.Show($"监听已启动，端口:{Port}, 协议类型：
    '{ProtocolType.Tcp}'");
    ListenerSocket.Bind(new IPEndPoint(IPAddress.Any, Port));
    ListenerSocket.Listen(10);
    ListenerSocket.BeginAccept(AcceptCallback, ListenerSocket);
}
catch(Exception ex)
{
    throw new Exception("listening error" + ex);
}
}

```

所以当客户端连接时，我们可以通过此方法接受它们：

这里我们使用的三个方法是：

[1.ListenerSocket.EndAccept\(\)](#)

我们用Listener.BeginAccept()启动了回调，现在必须结束该回调。EndAccept()方法接受一个IAsyncResult参数，该参数存储异步方法的状态，通过该状态我们可以提取出传入连接的套接字。

[2. ClientController.AddClient\(\)](#)

通过EndAccept()获得的套接字，我们用自定义方法创建一个客户端（代码示例见服务器下方的ClientController）。

[3.ListenerSocket.BeginAccept\(\)](#)

当套接字完成处理新连接时，我们需要重新开始监听。传入将接收此回调的方法。同时传入监听套接字，以便我们可以重用此套接字来处理即将到来的连接。

```

public void AcceptCallback(IAsyncResult ar)
{
    try
    {
        Console.WriteLine($"Accept CallBack 端口:{Port} 协议类型: {ProtocolType.Tcp}");
        Socket acceptedSocket = ListenerSocket.EndAccept(ar);
        ClientController.AddClient(acceptedSocket);

        ListenerSocket.BeginAccept(AcceptCallback, ListenerSocket);
    }
    catch (Exception ex)
    {
        throw new Exception("基础接受错误" + ex);
    }
}

```

现在我们有了一个监听套接字，但如何接收客户端发送的数据呢？接下来的代码将展示这一点。

```

try
{
    MessageBox.Show($"Listening started port:{Port} protocol type:
    {ProtocolType.Tcp}");
    ListenerSocket.Bind(new IPEndPoint(IPAddress.Any, Port));
    ListenerSocket.Listen(10);
    ListenerSocket.BeginAccept(AcceptCallback, ListenerSocket);
}
catch(Exception ex)
{
    throw new Exception("listening error" + ex);
}
}

```

So when a client connects we can accept them by this method:

Three methods we use here are:

[1. ListenerSocket.EndAccept\(\)](#)

We started the callback with Listener.BeginAccept() and now we have to end that call back. The EndAccept() method accepts an IAsyncResult parameter, this will store the state of the asynchronous method. From this state we can extract the socket where the incoming connection was coming from.

[2. ClientController.AddClient\(\)](#)

With the socket we got from EndAccept() we create an Client with an own made method (*code ClientController below server example*).

[3. ListenerSocket.BeginAccept\(\)](#)

We need to start listening again when the socket is done with handling the new connection. Pass in the method who will catch this callback. And also pass int the Listener socket so we can reuse this socket for upcoming connections.

```

public void AcceptCallback(IAsyncResult ar)
{
    try
    {
        Console.WriteLine($"Accept CallBack port:{Port} protocol type: {ProtocolType.Tcp}");
        Socket acceptedSocket = ListenerSocket.EndAccept(ar);
        ClientController.AddClient(acceptedSocket);

        ListenerSocket.BeginAccept(AcceptCallback, ListenerSocket);
    }
    catch (Exception ex)
    {
        throw new Exception("Base Accept error" + ex);
    }
}

```

Now we have an Listening Socket but how do we receive data send by the client that is what the next code is showing.

为每个客户端创建服务器接收器

首先创建一个接收类，其构造函数接收一个套接字作为参数：

```
public class ReceivePacket
{
    private byte[] _buffer;
    private Socket _receiveSocket;

    public ReceivePacket(Socket receiveSocket)
    {
        _receiveSocket = receiveSocket;
    }
}
```

在下一个方法中，我们首先给缓冲区分配4字节 (Int32) 的大小，或者说包包含两个部分 {长度，实际数据}。所以前4个字节我们保留给数据的长度，剩下的用于实际数据。

接下来我们使用 [BeginReceive\(\)](#) 方法。该方法用于开始从已连接的客户端接收数据，当接收到数据时，它会运行 [ReceiveCallback](#) 函数。

```
public void StartReceiving()
{
    try
    {
        _buffer = new byte[4];
        _receiveSocket.BeginReceive(_buffer, 0, _buffer.Length, SocketFlags.None,
        ReceiveCallback, null);
    }
    catch {}
}

private void ReceiveCallback(IAsyncResult AR)
{
    try
    {
        // 如果字节数小于1，表示客户端已断开与服务器的连接。
        // 因此我们对当前客户端运行断开连接函数
        if (_receiveSocket.EndReceive(AR) > 1)
        {
            // 将接收到的前4个字节 (int 32) 转换为Int32
            // (这是即将接收数据的大小)。
            _buffer = new byte[BitConverter.ToInt32(_buffer, 0)];
            // 接下来接收之前接收到的数据到缓冲区中
            _receiveSocket.Receive(_buffer, _buffer.Length, SocketFlags.None);
            // 当我们接收完所有数据后，接下来由你将其转换成你发送的数据类型。

            // 例如字符串、整数等.....在此示例中，我只实现了发送和接收字符串的功能。

            // 将字节转换为字符串并在消息框中显示
            string data = Encoding.Default.GetString(_buffer);
            MessageBox.Show(data);
            // 现在我们必须重新开始，等待来自套接字的数据。
        }
    }
    StartReceiving();
}
else
{
    断开连接();
}
```

Create Server Receiver for each client

First of create a receive class with a constructor that takes in a Socket as parameter:

```
public class ReceivePacket
{
    private byte[] _buffer;
    private Socket _receiveSocket;

    public ReceivePacket(Socket receiveSocket)
    {
        _receiveSocket = receiveSocket;
    }
}
```

In the next method we first start off with giving the buffer a size of 4 bytes (Int32) or package contains to parts {length, actual data}. So the first 4 bytes we reserve for the lenght of the data the rest for the actual data.

Next we use [BeginReceive\(\)](#) method. This method is used to start receiving from connected clients and when it will receive data it will run the [ReceiveCallback](#) function.

```
public void StartReceiving()
{
    try
    {
        _buffer = new byte[4];
        _receiveSocket.BeginReceive(_buffer, 0, _buffer.Length, SocketFlags.None,
        ReceiveCallback, null);
    }
    catch {}
}

private void ReceiveCallback(IAsyncResult AR)
{
    try
    {
        // if bytes are less than 1 takes place when a client disconnect from the server.
        // So we run the Disconnect function on the current client
        if (_receiveSocket.EndReceive(AR) > 1)
        {
            // Convert the first 4 bytes (int 32) that we received and convert it to an Int32
            // (this is the size for the coming data).
            _buffer = new byte[BitConverter.ToInt32(_buffer, 0)];
            // Next receive this data into the buffer with size that we did receive before
            _receiveSocket.Receive(_buffer, _buffer.Length, SocketFlags.None);
            // When we received everything its onto you to convert it into the data that you've
            send.
            // For example string, int etc... in this example I only use the implementation for
            sending and receiving a string.

            // Convert the bytes to string and output it in a message box
            string data = Encoding.Default.GetString(_buffer);
            MessageBox.Show(data);
            // Now we have to start all over again with waiting for a data to come from the
            socket.
            StartReceiving();
        }
        else
        {
            Disconnect();
        }
    }
}
```

```

        }
    catch
    {
        // 如果抛出异常，检查套接字是否已连接，因为这样你可以重新开始接收
        // 否则断开连接
        if (!_receiveSocket.Connected)
        {
            断开连接();
        }
        else
        {
            StartReceiving();
        }
    }

    private void 断开连接()
    {
        // 关闭连接
        _receiveSocket.Disconnect(true);
        // 下一行仅适用于服务器端接收
        ClientController.RemoveClient(_clientId);
        // 下一行仅适用于客户端接收端
        这里你想运行方法 TryToConnect()
    }
}

```

所以我们已经设置了一个服务器，可以接收并监听传入连接。当客户端连接时，它将被添加到客户端列表中，每个客户端都有自己的接收类。要让服务器监听：

```

Listener listener = new Listener();
listener.StartListening();

```

我在此示例中使用的一些类

```

class Client
{
    public Socket _socket { get; set; }
    public ReceivePacket Receive { get; set; }
    public int Id { get; set; }

    public Client(Socket socket, int id)
    {
        Receive = new ReceivePacket(socket, id);
        Receive.StartReceiving();
        _socket = socket;
        Id = id;
    }
}

```

```

静态类 ClientController
{
    public static List<Client> Clients = new List<Client>();

    public static void AddClient(Socket socket)
    {
        Clients.Add(new Client(socket, Clients.Count));
    }

    public static void RemoveClient(int id)
    {

```

```

        }
    }
    catch
    {
        // if exception is throw check if socket is connected because than you can startreive
        again else Disconnect
        if (!_receiveSocket.Connected)
        {
            Disconnect();
        }
        else
        {
            StartReceiving();
        }
    }
}

private void Disconnect()
{
    // Close connection
    _receiveSocket.Disconnect(true);
    // Next line only apply for the server side receive
    ClientController.RemoveClient(_clientId);
    // Next line only apply on the Client Side receive
    Here you want to run the method TryToConnect()
}

```

So we've setup a server that can receive and listen for incoming connections. When a clients connect it will be added to a list of clients and every client has his own receive class. To make the server listen:

```

Listener listener = new Listener();
listener.StartListening();

```

Some Classes I use in this example

```

class Client
{
    public Socket _socket { get; set; }
    public ReceivePacket Receive { get; set; }
    public int Id { get; set; }

    public Client(Socket socket, int id)
    {
        Receive = new ReceivePacket(socket, id);
        Receive.StartReceiving();
        _socket = socket;
        Id = id;
    }
}

static class ClientController
{
    public static List<Client> Clients = new List<Client>();

    public static void AddClient(Socket socket)
    {
        Clients.Add(new Client(socket, Clients.Count));
    }

    public static void RemoveClient(int id)
    {

```

```
Clients.RemoveAt(Clients.FindIndex(x => x.Id == id));  
}  
}
```

客户端示例

连接到服务器

首先我们要创建一个连接到服务器的类，我们给它的名字是：Connector：

```
类 Connector  
{  
    private Socket _connectingSocket;  
}
```

此类的下一个方法是 TryToConnect()

该方法有几个有趣的地方：

1. 创建套接字；
2. 接着我循环直到套接字连接成功
3. 每次循环只是让线程暂停1秒，我们不想让服务器遭受拒绝服务攻击 XD
4. 使用[Connect\(\)](#)时，它会尝试连接服务器。如果失败会抛出异常，但while循环会保持程序继续连接服务器。你可以使用[Connect CallBack](#)方法来实现，但我会选择在套接字连接成功时调用一个方法。
5. 注意客户端现在尝试连接本地电脑的1234端口。

```
public void TryToConnect()  
{  
    _connectingSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,  
    ProtocolType.Tcp);  
  
    while (!_connectingSocket.Connected)  
    {  
        Thread.Sleep(1000);  
  
        try  
        {  
            _connectingSocket.Connect(new IPEndPoint(IPAddress.Parse("127.0.0.1"), 1234));  
        }  
        catch {}  
    }  
    SetupForReceiveing();  
}  
  
private void SetupForReceiveing()  
{  
    // 查看客户端类，位于客户端示例底部  
    Client.SetClient(_connectingSocket);  
    Client.StartReceiving();  
}
```

```
Clients.RemoveAt(Clients.FindIndex(x => x.Id == id));  
}  
}
```

Client Side example

Connecting to server

First of all we want to create a class what connects to the server te name we give it is: Connector:

```
class Connector  
{  
    private Socket _connectingSocket;  
}
```

Next Method for this class is TryToConnect()

This method goth a few interestin things:

1. Create the socket;
2. Next I loop until the socket is connected
3. Every loop it is just holding the Thread for 1 second we don't want to DOS the server XD
4. With [Connect\(\)](#) it will try to connect to the server. If it fails it will throw an exception but the wile will keep the program connecting to the server. You can use a [Connect CallBack](#) method for this, but I'll just go for calling a method when the Socket is connected.
5. Notice the Client is now trying to connect to your local pc on port 1234.

```
public void TryToConnect()  
{  
    _connectingSocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,  
    ProtocolType.Tcp);  
  
    while (!_connectingSocket.Connected)  
    {  
        Thread.Sleep(1000);  
  
        try  
        {  
            _connectingSocket.Connect(new IPEndPoint(IPAddress.Parse("127.0.0.1"), 1234));  
        }  
        catch {}  
    }  
    SetupForReceiveing();  
}  
  
private void SetupForReceiveing()  
{  
    // View Client Class bottom of Client Example  
    Client.SetClient(_connectingSocket);  
    Client.StartReceiving();  
}
```

向服务器发送消息

现在我们几乎完成了一个Socket应用程序。唯一还没有的是一个用于向服务器发送消息的类。

```
public class SendPacket
{
    private Socket _sendSocked;

    public SendPacket(Socket sendSocket)
    {
        _sendSocked = sendSocket;
    }

    public void Send(string data)
    {
        try
        {
            /* 这里发生的事情：
            1. 创建一个字节列表
            2. 将字符串的长度添加到列表中。
            这样当消息到达服务器时，我们可以轻松读取即将到来的消息的长度。
            3. 添加消息（字符串）的字节
            */
            var fullPacket = new List<byte>();
            fullPacket.AddRange(BitConverter.GetBytes(data.Length));
            fullPacket.AddRange(Encoding.Default.GetBytes(data));

            /* 将消息发送到我们当前连接的服务器。*/
            或者包结构是 {数据长度 4 字节 (int32) , 实际数据}/*
            _sendSocked.Send(fullPacket.ToArray());
        }
        catch (Exception ex)
        {
            throw new Exception();
        }
    }
}
```

最后创建两个按钮，一个用于连接，另一个用于发送消息：

```
private void ConnectClick(object sender, EventArgs e)
{
    Connector tpp = new Connector();
    tpp.TryToConnect();
}

private void SendClick(object sender, EventArgs e)
{
    Client.SendString("来自客户端的测试数据");
}
```

我在此示例中使用的客户端类

```
public static void SetClient(Socket socket)
{
    Id = 1;
    Socket = socket;
    Receive = new ReceivePacket(socket, Id);
}
```

Sending a message to the server

So now we have an almost finish or Socket application. The only thing that we don't have yet is a Class for sending a message to the server.

```
public class SendPacket
{
    private Socket _sendSocked;

    public SendPacket(Socket sendSocket)
    {
        _sendSocked = sendSocket;
    }

    public void Send(string data)
    {
        try
        {
            /* what happens here:
            1. Create a list of bytes
            2. Add the length of the string to the list.
            So if this message arrives at the server we can easily read the length of the
            coming message.
            3. Add the message(string) bytes
            */
            var fullPacket = new List<byte>();
            fullPacket.AddRange(BitConverter.GetBytes(data.Length));
            fullPacket.AddRange(Encoding.Default.GetBytes(data));

            /* Send the message to the server we are currently connected to.
            Or package stucture is {length of data 4 bytes (int32), actual data}/*
            _sendSocked.Send(fullPacket.ToArray());
        }
        catch (Exception ex)
        {
            throw new Exception();
        }
    }
}
```

Finally create two buttons one for connect and the other for sending a message:

```
private void ConnectClick(object sender, EventArgs e)
{
    Connector tpp = new Connector();
    tpp.TryToConnect();
}

private void SendClick(object sender, EventArgs e)
{
    Client.SendString("Test data from client");
}
```

The client class I used in this example

```
public static void SetClient(Socket socket)
{
    Id = 1;
    Socket = socket;
    Receive = new ReceivePacket(socket, Id);
}
```

```
SendPacket = new SendPacket(socket);  
    }  
}
```

注意

服务器端的 Receive 类与客户端的 Receive 类相同。

结论

你现在有了一个服务器和一个客户端。你可以完善这个基本示例。例如，让服务器也能够接收文件或其他东西。或者向客户端发送消息。在服务器中你有一个客户端列表，所以当你接收到某些东西时，你会知道它来自哪个客户端。

最终结果：

A screenshot of a Windows desktop environment. In the foreground, there is a code editor window showing C# code related to network communication. The code includes imports for System, System.Net, System.Net.Sockets, and System.Threading. It defines a class Client with a StartServer method that creates a listening socket and a client socket. The client socket connects to a host at port 1234. A ProtocolType enum is also defined with values Udp and Tcp. In the background, there are four windows titled 'Form1'. Three of these windows contain a 'Send' button and a 'Connect' button. The fourth window contains a single 'Start Server' button. There are also three overlapping message boxes with the title 'Test data from client' and an 'OK' button.

```
SendPacket = new SendPacket(socket);
```

Notice

The Receive Class from the server is the same as the receive class from the client.

Conclusion

You now have a server and a client. You can work this basic example out. For example make it that the server also can receive files or other things. Or send a message to the client. In the server you got a list of clients so when you receive something you will know from which client it came from.

Final result

The screenshot shows a Windows application interface with several windows and code snippets:

- Client**: The main window title bar.
- Client.Connector**: A sub-window or tab showing the following code:

```
new Client.Connector("127.0.0.1", 1234);
```
- ProtocolType**: A sub-window or tab showing the following code:

```
ProtocolType.Tcp;
```
- Start Server**: A button labeled "Start Server" located in the center of the application area.
- Form1**: Three separate windows titled "Form1" arranged horizontally. Each window contains:
 - A "Send" button.
 - A "Connect" button.
 - An "OK" button.
 - The text "Test data from client".

第104章：操作过滤器

第104.1节：自定义操作过滤器

我们编写自定义操作过滤器有多种原因。我们可能有一个用于日志记录的自定义操作过滤器，或者用于在任何操作执行之前将数据保存到数据库。我们也可能有一个用于从数据库获取数据并将其设置为应用程序的全局值的过滤器。

要创建自定义操作过滤器，我们需要执行以下任务：

1. 创建一个类
2. 继承自 ActionFilterAttribute 类

至少重写以下方法之一：

OnActionExecuting – 该方法在控制器操作执行之前调用。

OnActionExecuted – 该方法在控制器操作执行之后调用。

OnResultExecuting – 该方法在控制器操作结果执行之前调用。

OnResultExecuted – 该方法在控制器操作结果执行之后调用。

过滤器可以按下面的示例创建：

```
using System;
using System.Diagnostics;
using System.Web.Mvc;

namespace WebApplication1
{
    public class MyFirstCustomFilter : ActionFilterAttribute
    {
        public override void OnResultExecuting(ResultExecutingContext filterContext)
        {
            //您可以在这里从数据库获取数据
            filterContext.Controller.ViewBag.GreetMessage = "Hello Foo";
            base.OnResultExecuting(filterContext);
        }

        public override void OnActionExecuting(ActionExecutingContext filterContext)
        {
            var controllerName = filterContext.RouteData.Values["controller"];
            var actionName = filterContext.RouteData.Values["action"];
            var message = String.Format("{0} controller:{1} action:{2}", "onactionexecuting",
                controllerName, actionName);
            Debug.WriteLine(message, "Action Filter Log");
            base.OnActionExecuting(filterContext);
        }
    }
}
```

Chapter 104: Action Filters

Section 104.1: Custom Action Filters

We write custom action filters for various reasons. We may have a custom action filter for logging, or for saving data to database before any action execution. We could also have one for fetching data from the database and setting it as the global values of the application.

To create a custom action filter, we need to perform the following tasks:

1. Create a class
2. Inherit it from ActionFilterAttribute class

Override at least one of the following methods:

OnActionExecuting – This method is called before a controller action is executed.

OnActionExecuted – This method is called after a controller action is executed.

OnResultExecuting – This method is called before a controller action result is executed.

OnResultExecuted – This method is called after a controller action result is executed.

The filter can be created as shown in the listing below:

```
using System;
using System.Diagnostics;
using System.Web.Mvc;

namespace WebApplication1
{
    public class MyFirstCustomFilter : ActionFilterAttribute
    {
        public override void OnResultExecuting(ResultExecutingContext filterContext)
        {
            //You may fetch data from database here
            filterContext.Controller.ViewBag.GreetMessage = "Hello Foo";
            base.OnResultExecuting(filterContext);
        }

        public override void OnActionExecuting(ActionExecutingContext filterContext)
        {
            var controllerName = filterContext.RouteData.Values["controller"];
            var actionName = filterContext.RouteData.Values["action"];
            var message = String.Format("{0} controller:{1} action:{2}", "onactionexecuting",
                controllerName, actionName);
            Debug.WriteLine(message, "Action Filter Log");
            base.OnActionExecuting(filterContext);
        }
    }
}
```

第105章：多态性

第105.1节：多态性的类型

多态性意味着一个操作也可以应用于某些其他类型的值。

多态性有多种类型：

- 特设多态 (Ad hoc polymorphism)：
包含函数重载。目标是方法可以用于不同类型，而无需泛型。
- 参数多态 (Parametric polymorphism)：
是泛型类型的使用。参见泛型 (Generics)
- 子类型 (Subtyping)：
目标是通过继承一个类来泛化类似的功能

特设多态

特设多态的目标是创建一个方法，该方法可以被不同数据类型调用，而无需在函数调用中进行类型转换或使用泛型。以下方法 sumInt(par1, par2)可以用不同数据类型调用，并且针对每种类型组合都有自己的实现：

```
public static int sumInt( int a, int b)
{
    return a + b;
}

public static int sumInt( string a, string b)
{
    int _a, _b;

    if(!Int32.TryParse(a, out _a))
        _a = 0;

    if(!Int32.TryParse(b, out _b))
        _b = 0;

    return _a + _b;
}

public static int sumInt(string a, int b)
{
    int _a;

    if(!Int32.TryParse(a, out _a))
        _a = 0;

    return _a + b;
}

public static int sumInt(int a, string b)
{
    return sumInt(b,a);
}
```

这是一个示例调用：

Chapter 105: Polymorphism

Section 105.1: Types of Polymorphism

Polymorphism means that a operation can also be applied to values of some other types.

There are multiple types of Polymorphism:

- **Ad hoc polymorphism:**
contains function overloading. The target is that a Method can be used with different types without the need of being generic.
- **Parametric polymorphism:**
is the use of generic types. See Generics
- **Subtyping:**
has the target inherit of a class to generalize a similar functionality

Ad hoc polymorphism

The target of Ad hoc polymorphism is to create a method, that can be called by different datatypes without a need of type-conversion in the function call or generics. The following method(s) sumInt(par1, par2) can be called with different datatypes and has for each combination of types a own implementation:

```
public static int sumInt( int a, int b)
{
    return a + b;
}

public static int sumInt( string a, string b)
{
    int _a, _b;

    if(!Int32.TryParse(a, out _a))
        _a = 0;

    if(!Int32.TryParse(b, out _b))
        _b = 0;

    return _a + _b;
}

public static int sumInt(string a, int b)
{
    int _a;

    if(!Int32.TryParse(a, out _a))
        _a = 0;

    return _a + b;
}

public static int sumInt(int a, string b)
{
    return sumInt(b,a);
}
```

Here's a example call:

```

public static void Main()
{
    Console.WriteLine(sumInt( 1 , 2 )); // 3
    Console.WriteLine(sumInt("3","4")); // 7
    Console.WriteLine(sumInt("5", 6 )); // 11
    Console.WriteLine(sumInt( 7 , "8")); // 15
}

```

子类型

子类型是通过继承基类来泛化相似行为的使用：

```

public interface Car{
    void refuel();
}

public class NormalCar : Car
{
    public void refuel()
    {
        Console.WriteLine("正在加汽油");
    }
}

public class ElectricCar : Car
{
    public void refuel()
    {
        Console.WriteLine("正在给电池充电");
    }
}

```

NormalCar 和 ElectricCar 两个类现在都有一个加油方法，但各自实现不同。示例如下：

```

public static void Main()
{
    List<Car> cars = new List<Car>(){
        new NormalCar(),
        new ElectricCar()
    };

    cars.ForEach(x => x.refuel());
}

```

输出结果如下：

加油
充电电池

第105.2节：另一个多态示例

多态是面向对象编程的支柱之一。Poly源自希腊语，意为“多种形式”。

下面是一个展示多态的示例。类Vehicle作为基类，具有多种形式。

```

public static void Main()
{
    Console.WriteLine(sumInt( 1 , 2 )); // 3
    Console.WriteLine(sumInt("3","4")); // 7
    Console.WriteLine(sumInt("5", 6 )); // 11
    Console.WriteLine(sumInt( 7 , "8")); // 15
}

```

Subtyping

Subtyping is the use of inherit from a base class to generalize a similar behavior:

```

public interface Car{
    void refuel();
}

public class NormalCar : Car
{
    public void refuel()
    {
        Console.WriteLine("Refueling with petrol");
    }
}

public class ElectricCar : Car
{
    public void refuel()
    {
        Console.WriteLine("Charging battery");
    }
}

```

Both classes NormalCar and ElectricCar now have a method to refuel, but their own implementation. Here's a Example:

```

public static void Main()
{
    List<Car> cars = new List<Car>(){
        new NormalCar(),
        new ElectricCar()
    };

    cars.ForEach(x => x.refuel());
}

```

The output will be was following:

Refueling with petrol
Charging battery

Section 105.2: Another Polymorphism Example

Polymorphism is one of the pillar of OOP. Poly derives from a Greek term which means 'multiple forms'.

Below is an example which exhibits Polymorphism. The class Vehicle takes multiple forms as a base class.

派生类Ducati和Lamborghini继承自Vehicle，并重写基类的Display()方法，以显示各自的NumberOfWheels。

```
public class Vehicle
{
    protected int NumberOfWheels { get; set; } = 0;
    public Vehicle()
    {
    }

    public virtual void Display()
    {
        Console.WriteLine($"类{nameof(Vehicle)}的轮子数量是{NumberOfWheels}");
    }
}

public class Ducati : Vehicle
{
    public Ducati()
    {
        NoOfWheels = 2;
    }

    public override void Display()
    {
        Console.WriteLine($"名为 {nameof(Ducati)} 的车轮数量是 {NumberOfWheels}");
    }
}

public class Lamborghini : Vehicle
{
    public Lamborghini()
    {
        NoOfWheels = 4;
    }

    public override void Display()
    {
        Console.WriteLine($"名为 {nameof(Lamborghini)} 的车轮数量是 {NumberOfWheels}");
    }
}
```

下面是展示多态性的代码片段。在第1行，使用变量vehicle为基类型Vehicle创建对象。第2行调用基类方法Display()并显示如下输出。

```
static void Main(string[] args)
{
    Vehicle vehicle = new Vehicle(); //第1行
    vehicle.Display(); //第2行
    vehicle = new Ducati(); //第3行
    vehicle.Display(); //第4行
    vehicle = new Lamborghini(); //第5行
    vehicle.Display(); //第6行
}
```

在第3行，vehicle 对象指向派生类 Ducati 并调用其 Display() 方法，显示的输出如图所示。这里体现了多态行为，尽管对象 vehicle 的类型是 Vehicle，它调用了派生类的方法 Display()，因为类型 Ducati 重写了基类的 Display() 方法，原因是

The Derived classes Ducati and Lamborghini inherits from Vehicle and overrides the base class's Display() method, to display its own NumberOfWheels.

```
public class Vehicle
{
    protected int NumberOfWheels { get; set; } = 0;
    public Vehicle()
    {
    }

    public virtual void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Vehicle)} is {NumberOfWheels}");
    }
}

public class Ducati : Vehicle
{
    public Ducati()
    {
        NoOfWheels = 2;
    }

    public override void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Ducati)} is {NumberOfWheels}");
    }
}

public class Lamborghini : Vehicle
{
    public Lamborghini()
    {
        NoOfWheels = 4;
    }

    public override void Display()
    {
        Console.WriteLine($"The number of wheels for the {nameof(Lamborghini)} is {NumberOfWheels}");
    }
}
```

Below is the code snippet where Polymorphism is exhibited. The object is created for the base type Vehicle using a variable vehicle at Line 1. It calls the base class method Display() at Line 2 and display the output as shown.

```
static void Main(string[] args)
{
    Vehicle vehicle = new Vehicle(); //Line 1
    vehicle.Display(); //Line 2
    vehicle = new Ducati(); //Line 3
    vehicle.Display(); //Line 4
    vehicle = new Lamborghini(); //Line 5
    vehicle.Display(); //Line 6
}
```

At Line 3, the vehicle object is pointed to the derived class Ducati and calls its Display() method, which displays the output as shown. Here comes the polymorphic behavior, even though the object vehicle is of type Vehicle, it calls the derived class method Display() as the type Ducati overrides the base class Display() method, since the

车辆对象指向杜卡迪。

当调用兰博基尼类型的Display()方法时，同样的解释适用。

输出如下所示

```
车辆的轮子数量是0      // 第2行  
杜卡迪的轮子数量是2    // 第4行  
兰博基尼的轮子数量是4  // 第6行
```

vehicle object is pointed towards Ducati.

The same explanation is applicable when it invokes the Lamborghini type's Display() method.

The Output is shown below

```
The number of wheels for the Vehicle is 0      // Line 2  
The number of wheels for the Ducati is 2        // Line 4  
The number of wheels for the Lamborghini is 4    // Line 6
```

belindoc.com

第106章：不可变性

第106.1节：System.String类

在C# (和.NET) 中，字符串由System.String类表示。关键字string是该类的别名。

System.String类是不可变的，即一旦创建，其状态就不能被更改。

所以你对字符串执行的所有操作，比如 Substring、Remove、Replace、使用+运算符进行的连接等都会创建一个新的字符串并返回它。

请参见以下程序示例 -

```
string str = "mystring";
string newString = str.Substring(3);
Console.WriteLine(newString);
Console.WriteLine(str);
```

这将分别打印string和mystring。

第106.2节：字符串与不可变性

不可变类型是指当其被更改时，会在内存中创建该对象的新版本，而不是更改内存中已有的对象。最简单的例子就是内置的string类型。

以下代码将“ world”附加到单词“Hello”上

```
string myString = "hello";
myString += " world";
```

在这种情况下，内存中发生的情况是，当你在第二行向string追加内容时，会创建一个新对象。如果你在一个大型循环中这样做，可能会导致应用程序的性能问题。

字符串的可变等价物是 StringBuilder

以下代码

```
StringBuilder myStringBuilder = new StringBuilder("hello");
myStringBuilder.append(" world");
```

当你运行此代码时，你实际上是在修改内存中的StringBuilder对象本身。

Chapter 106: Immutability

Section 106.1: System.String class

In C# (and .NET) a string is represented by class System.String. The `string` keyword is an alias for this class.

The System.String class is immutable, i.e once created its state cannot be altered.

So all the operations you perform on a string like Substring, Remove, Replace, concatenation using + operator etc will create a new string and return it.

See the following program for demonstration -

```
string str = "mystring";
string newString = str.Substring(3);
Console.WriteLine(newString);
Console.WriteLine(str);
```

This will print `string` and `mystring` respectively.

Section 106.2: Strings and immutability

Immutable types are types that when changed create a new version of the object in memory, rather than changing the existing object in memory. The simplest example of this is the built-in `string` type.

Taking the following code, that appends " world" onto the word "Hello"

```
string myString = "hello";
myString += " world";
```

What is happening in memory in this case is that a new object is created when you append to the `string` in the second line. If you do this as part of a large loop, there is the potential for this to cause performance issues in your application.

The mutable equivalent for a `string` is a `StringBuilder`

Taking the following code

```
StringBuilder myStringBuilder = new StringBuilder("hello");
myStringBuilder.append(" world");
```

When you run this, you are modifying the `StringBuilder` object itself in memory.

第107章：索引器

第107.1节：一个简单的索引器

```
class Foo
{
    private string[] cities = new[] { "巴黎", "伦敦", "柏林" };

    public string this[int index]
    {
        get {
            return cities[index];
        }
        set {
            cities[index] = value;
        }
    }
}
```

用法：

```
var foo = new Foo();

// 访问一个值
string berlin = foo[2];

// 赋值
foo[0] = "Rome";
```

[查看演示](#)

第107.2节：重载索引器以创建一个稀疏数组

通过重载索引器，你可以创建一个看起来和感觉像数组但实际上不是的类。它将具有O(1)的获取和设置方法，可以访问索引为100的元素，同时其大小仍然是其中元素的大小。该稀疏数组类

```
class 稀疏数组
{
    Dictionary<int, string> array = new Dictionary<int, string>();

    public string this[int i]
    {
        get
        {
            if(!array.ContainsKey(i))
            {
                return null;
            }
            return array[i];
        }
        set
        {
            if(!array.ContainsKey(i))
                array.Add(i, value);
        }
    }
}
```

Chapter 107: Indexer

Section 107.1: A simple indexer

```
class Foo
{
    private string[] cities = new[] { "Paris", "London", "Berlin" };

    public string this[int index]
    {
        get {
            return cities[index];
        }
        set {
            cities[index] = value;
        }
    }
}
```

Usage:

```
var foo = new Foo();

// access a value
string berlin = foo[2];

// assign a value
foo[0] = "Rome";
```

[View Demo](#)

Section 107.2: Overloading the indexer to create a SparseArray

By overloading the indexer you can create a class that looks and feels like an array but isn't. It will have O(1) get and set methods, can access an element at index 100, and yet still have the size of the elements inside of it. The SparseArray class

```
class SparseArray
{
    Dictionary<int, string> array = new Dictionary<int, string>();

    public string this[int i]
    {
        get
        {
            if(!array.ContainsKey(i))
            {
                return null;
            }
            return array[i];
        }
        set
        {
            if(!array.ContainsKey(i))
                array.Add(i, value);
        }
    }
}
```

```
}
```

第107.3节：带有两个参数的索引器和接口

```
interface ITable {  
    // 接口中可以声明索引器  
    object this[int x, int y] { get; set; }  
}  
  
class DataTable : ITable  
{  
    private object[,] cells = new object[10, 10];  
  
    /// <summary>  
    /// 接口中声明的索引器的实现  
    ///</summary>  
    ///<param name="x">X索引</param>  
    ///<param name="y">Y索引</param>  
    ///<returns>该单元格的内容</returns>  
    public object this[int x, int y]  
    {  
        get  
        {  
            return cells[x, y];  
        }  
        设置  
        {  
            cells[x, y] = value;  
        }  
    }  
}
```

```
}
```

Section 107.3: Indexer with 2 arguments and interface

```
interface ITable {  
    // an indexer can be declared in an interface  
    object this[int x, int y] { get; set; }  
}  
  
class DataTable : ITable  
{  
    private object[,] cells = new object[10, 10];  
  
    /// <summary>  
    /// implementation of the indexer declared in the interface  
    /// </summary>  
    ///<param name="x">X-Index</param>  
    ///<param name="y">Y-Index</param>  
    ///<returns>Content of this cell</returns>  
    public object this[int x, int y]  
    {  
        get  
        {  
            return cells[x, y];  
        }  
        set  
        {  
            cells[x, y] = value;  
        }  
    }  
}
```

第108章：Checked 和 Unchecked

第108.1节：Checked 和 Unchecked

C# 语句在 checked 或 unchecked 上下文中执行。在 checked 上下文中，算术溢出会引发异常。在 unchecked 上下文中，算术溢出被忽略，结果被截断。

```
short m = 32767;
short n = 32767;
int result1 = checked((short)(m + n)); //将抛出 OverflowException
int result2 = unchecked((short)(m + n)); // 将返回 -2
```

如果未指定这两者中的任何一个，则默认上下文将依赖其他因素，例如编译器选项。

第108.2节：作为作用域的 Checked 和 Unchecked

这些关键字也可以创建作用域，以便对多个操作进行（不）检查。

```
short m = 32767;
short n = 32767;
checked
{
    int result1 = (short)(m + n); //将抛出 OverflowException
}
unchecked
{
    int result2 = (short)(m + n); // 将返回 -2
}
```

Chapter 108: Checked and Unchecked

Section 108.1: Checked and Unchecked

C# statements executes in either checked or unchecked context. In a checked context, arithmetic overflow raises an exception. In an unchecked context, arithmetic overflow is ignored and the result is truncated.

```
short m = 32767;
short n = 32767;
int result1 = checked((short)(m + n)); //will throw an OverflowException
int result2 = unchecked((short)(m + n)); // will return -2
```

If neither of these are specified then the default context will rely on other factors, such as compiler options.

Section 108.2: Checked and Unchecked as a scope

The keywords can also create scopes in order to (un)check multiple operations.

```
short m = 32767;
short n = 32767;
checked
{
    int result1 = (short)(m + n); //will throw an OverflowException
}
unchecked
{
    int result2 = (short)(m + n); // will return -2
}
```

第109章：流

第109.1节：使用流

流是一个提供低级数据传输手段的对象。它们本身不作为数据容器。

我们处理的数据是以字节数组 (byte []) 的形式存在。读写函数都是面向字节的，例如WriteByte()。

没有处理整数、字符串等的函数。这使得流非常通用，但如果你只是想传输文本，使用起来就不那么简单。流在处理大量数据时尤其有用。

我们需要根据写入/读取的位置（即后备存储）使用不同类型的流。

例如，如果源是文件，我们需要使用FileStream：

```
string filePath = @"c:\Users\exampleuser\Documents\userinputlog.txt";
using (FileStream fs = new FileStream(filePath, FileMode.Open, FileAccess.Read,
FileShare.ReadWrite))
{
    // 在这里执行操作...

fs.Close();
}
```

同样，如果后备存储是内存，则使用MemoryStream：

```
// 从磁盘上的文件中读取所有字节。
byte[] file = File.ReadAllBytes("C:\\file.txt");

// 从这些字节创建一个内存流。
using (MemoryStream memory = new MemoryStream(file))
{
    // 在这里执行操作...
}
```

类似地，System.Net.Sockets.NetworkStream 用于网络访问。

所有流都派生自通用类 System.IO.Stream。数据不能直接从流中读取或写入。.NET 框架提供了辅助类，如 StreamReader、StreamWriter、BinaryReader 和BinaryWriter，它们在本地类型和底层流接口之间进行转换，并为你传输数据到流或从流中传输数据。

可以通过 StreamReader 和 StreamWriter 进行流的读写。关闭这些时应当小心。默认情况下，关闭操作也会关闭所包含的流，使其无法再使用。可以通过使用带有 bool leaveOpen 参数的构造函数来改变此默认行为，并将其值设置为 true。

StreamWriter：

```
FileStream fs = new FileStream("sample.txt", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
string NextLine = "This is the appended line.";
sw.WriteLine(NextLine);
```

Chapter 109: Stream

Section 109.1: Using Streams

A stream is an object that provides a low-level means to transfer data. They themselves do not act as data containers.

The data that we deal with is in form of byte array([byte](#) []). The functions for reading and writing are all byte orientated, e.g. `WriteByte()`.

There are no functions for dealing with integers, strings etc. This makes the stream very general-purpose, but less simple to work with if, say, you just want to transfer text. Streams can be particularly very helpful when you are dealing with large amount of data.

We will need to use different type of Stream based where it needs to be written/read from (i.e. the backing store). For example, if the source is a file, we need to use `FileStream`:

```
string filePath = @"c:\Users\exampleuser\Documents\userinputlog.txt";
using (FileStream fs = new FileStream(filePath, FileMode.Open, FileAccess.Read,
FileShare.ReadWrite))
{
    // do stuff here...

fs.Close();
}
```

Similarly, `MemoryStream` is used if the backing store is memory:

```
// Read all bytes in from a file on the disk.
byte[] file = File.ReadAllBytes("C:\\file.txt");

// Create a memory stream from those bytes.
using (MemoryStream memory = new MemoryStream(file))
{
    // do stuff here...
}
```

Similarly, `System.Net.Sockets.NetworkStream` is used for network access.

All Streams are derived from the generic class `System.IO.Stream`. Data cannot be directly read or written from streams. The .NET Framework provides helper classes such as `StreamReader`, `StreamWriter`, `BinaryReader` and `BinaryWriter` that convert between native types and the low-level stream interface, and transfer the data to or from the stream for you.

Reading and writing to streams can be done via `StreamReader` and `StreamWriter`. One should be careful when closing these. By default, closing will also close contained stream as well and make it unusable for further uses. This default behaviour can be change by using a [constructor](#) which has `bool` `leaveOpen` parameter and setting its value as `true`.

StreamWriter:

```
FileStream fs = new FileStream("sample.txt", FileMode.Create);
StreamWriter sw = new StreamWriter(fs);
string NextLine = "This is the appended line.";
sw.WriteLine(NextLine);
```

```
sw.Close();
//fs.Close(); 不需要关闭 fs。关闭 sw 也会关闭它所包含的流。
```

StreamReader:

```
using (var ms = new MemoryStream())
{
    StreamWriter sw = new StreamWriter(ms);
    sw.Write(123);
    //sw.Close(); 这将关闭 ms, 当我们稍后尝试使用 ms 时会导致异常
    sw.Flush(); //你可以将剩余数据发送到流。关闭时会自动执行此操作
    // 我们需要将位置设置为 0 以便从头开始读取
    // 从头开始读取。
    ms.Position = 0;
    StreamReader sr = new StreamReader(ms);
    var myStr = sr.ReadToEnd();
    sr.Close();
    ms.Close();
}
```

belindoc.com

```
sw.Close();
//fs.Close(); There is no need to close fs. Closing sw will also close the stream it contains.
```

StreamReader:

```
using (var ms = new MemoryStream())
{
    StreamWriter sw = new StreamWriter(ms);
    sw.Write(123);
    //sw.Close(); This will close ms and when we try to use ms later it will cause an exception
    sw.Flush(); //You can send the remaining data to stream. Closing will do this automatically
    // We need to set the position to 0 in order to read
    // from the beginning.
    ms.Position = 0;
    StreamReader sr = new StreamReader(ms);
    var myStr = sr.ReadToEnd();
    sr.Close();
    ms.Close();
}
```

Since Classes Stream, StreamReader, StreamWriter, etc. implement the IDisposable interface, we can call the Dispose() method on objects of these classes.

第110章：计时器

第110.1节：多线程计时器

System.Threading.Timer - 最简单的多线程计时器。包含两个方法和一个构造函数。

示例：一个计时器调用DataWrite方法，该方法在五秒后写入“multithread executed...”，然后每秒写入一次，直到用户按下回车键为止：

```
using System;
using System.Threading;
class Program
{
    static void Main()
    {
        // 第一个间隔 = 5000毫秒；后续间隔 = 1000毫秒
        Timer timer = new Timer (DataWrite, "multithread executed...", 5000, 1000);
        Console.ReadLine();
        timer.Dispose(); // 这既停止计时器又进行清理。
    }

    static void DataWrite (object data)
    {
        // 这在一个线程池线程上运行
        Console.WriteLine (data); // 输出 "multithread executed..."
    }
}
```

注意：将单独发布一个关于释放多线程定时器的部分。

Change - 当你想更改定时器间隔时可以调用此方法。

Timeout.Infinite - 如果你只想触发一次。在构造函数的最后一个参数中指定此值。

System.Timers - .NET 框架提供的另一个定时器类。它封装了 System.Threading.Timer。

功能：

- IComponent - 允许它被放置在 Visual Studio‘设计器’的组件托盘中
- Interval 属性代替 Change 方法
- Elapsed 事件 代替回调 委托
- Enabled 属性用于启动和停止定时器（默认值 = false）
- Start 和 Stop 方法，以防你对上面的 Enabled 属性感到困惑
- AutoReset - 指示是否为重复事件（默认值 = true）
- SynchronizingObject 属性与 Invoke 和 BeginInvoke 方法，用于安全调用 WPF 元素和 Windows 窗体控件上的方法

示例，展示上述所有特性：

```
using System;
// 使用 System.Timers; // 使用 Timers 命名空间而非 Threading
class SystemTimer
{
    static void Main()
    {
        Timer timer = new Timer(); // 不需要任何参数
        timer.Interval = 500;
```

Chapter 110: Timers

Section 110.1: Multithreaded Timers

System.Threading.Timer - Simplest multithreaded timer. Contains two methods and one constructor.

Example: A timer calls the DataWrite method, which writes "multithread executed..." after five seconds have elapsed, and then every second after that until the user presses Enter:

```
using System;
using System.Threading;
class Program
{
    static void Main()
    {
        // First interval = 5000ms; subsequent intervals = 1000ms
        Timer timer = new Timer (DataWrite, "multithread executed...", 5000, 1000);
        Console.ReadLine();
        timer.Dispose(); // This both stops the timer and cleans up.
    }

    static void DataWrite (object data)
    {
        // This runs on a pooled thread
        Console.WriteLine (data); // Writes "multithread executed..."
    }
}
```

Note : Will post a separate section for disposing multithreaded timers.

Change - This method can be called when you would like change the timer interval.

Timeout.Infinite - If you want to fire just once. Specify this in the last argument of the constructor.

System.Timers - Another timer class provided by .NET Framework. It wraps the System.Threading.Timer.

Features:

- IComponent - Allowing it to be sited in the Visual Studio's Designer's component tray
- Interval property instead of a Change method
- Elapsed event instead of a callback delegate
- Enabled property to start and stop the timer (default value = false)
- Start & Stop methods in case if you get confused by Enabled property (above point)
- AutoReset - for indicating a recurring event (default value = true)
- SynchronizingObject property with Invoke and BeginInvoke methods for safely calling methods on WPF elements and Windows Forms controls

Example representing all the above features:

```
using System;
// Timers namespace rather than Threading
class SystemTimer
{
    static void Main()
    {
        Timer timer = new Timer(); // Doesn't require any args
        timer.Interval = 500;
```

```

timer.Elapsed += timer_Elapsed; // 使用事件而非委托
    timer.Start(); // 启动计时器
Console.ReadLine();
    timer.Stop(); // 停止计时器
    Console.ReadLine();
timer.Start(); // 重新启动计时器
    Console.ReadLine();
timer.Dispose(); // 永久停止计时器
}

static void timer_Elapsed(object sender, EventArgs e)
{
Console.WriteLine ("Tick");
}

```

多线程计时器 - 使用线程池允许少数线程服务多个计时器。这意味着回调方法或Elapsed事件每次调用时可能在不同的线程上触发。

Elapsed - 此事件总是准时触发—无论之前的Elapsed事件是否已完成执行。

因此，回调或事件处理程序必须是线程安全的。多线程计时器的准确性取决于操作系统，通常在10-20毫秒之间。

interop - 当需要更高精度时使用此方法，调用Windows多媒体计时器。其精度可达到1毫秒，定义在winmm.dll中。

timeBeginPeriod - 首先调用此函数通知操作系统需要高精度计时

timeSetEvent - 在 timeBeginPeriod之后调用以启动多媒体计时器。

timeKillEvent - 完成后调用此函数以停止计时器

timeEndPeriod - 调用此函数通知操作系统不再需要高精度计时。

您可以在互联网上通过搜索关键词找到使用多媒体计时器的完整示例
dllimport winmm.dll timesetevent.

第110.2节：创建计时器实例

计时器用于在特定时间间隔执行任务（每隔Y秒执行X操作）。以下是创建计时器新实例的示例。

注意：这适用于使用 WinForms 的计时器。如果使用 WPF，您可能需要了解一下DispatcherTimer

```

using System.Windows.Forms; //Timers 使用 Windows.Forms 命名空间

public partial class Form1 : Form
{

    Timer myTimer = new Timer(); //创建一个名为 myTimer 的 Timer 实例

    public Form1()
    {
        InitializeComponent();
    }
}

```

```

timer.Elapsed += timer_Elapsed; // Uses an event instead of a delegate
    timer.Start(); // Start the timer
Console.ReadLine();
    timer.Stop(); // Stop the timer
    Console.ReadLine();
timer.Start(); // Restart the timer
    Console.ReadLine();
timer.Dispose(); // Permanently stop the timer
}

static void timer_Elapsed(object sender, EventArgs e)
{
    Console.WriteLine ("Tick");
}

```

Multithreaded timers - use the thread pool to allow a few threads to serve many timers. It means that callback method or Elapsed event may trigger on a different thread each time it is called.

Elapsed - this event always fires on time—regardless of whether the previous Elapsed event finished executing. Because of this, callbacks or event handlers must be thread-safe. The accuracy of multithreaded timers depends on the OS, and is typically in the 10–20 ms.

interop - when ever you need greater accuracy use this and call the Windows multimedia timer. This has accuracy down to 1 ms and it is defined in winmm.dll.

timeBeginPeriod - Call this first to inform OS that you need high timing accuracy

timeSetEvent - call this after timeBeginPeriod to start a multimedia timer.

timeKillEvent - call this when you are done, this stops the timer

timeEndPeriod - Call this to inform the OS that you no longer need high timing accuracy.

You can find complete examples on the Internet that use the multimedia timer by searching for the keywords
dllimport winmm.dll timesetevent.

Section 110.2: Creating an Instance of a Timer

Timers are used to perform tasks at specific intervals of time (Do X every Y seconds) Below is an example of creating a new instance of a Timer.

NOTE: This applies to Timers using WinForms. If using WPF, you may want to look into DispatcherTimer

```

using System.Windows.Forms; //Timers use the Windows.Forms namespace

public partial class Form1 : Form
{

    Timer myTimer = new Timer(); //create an instance of Timer named myTimer

    public Form1()
    {
        InitializeComponent();
    }
}

```

第110.3节：将“Tick”事件处理程序分配给计时器

计时器中执行的所有操作都在“Tick”事件中处理。

```
public partial class Form1 : Form
{
    Timer myTimer = new Timer();

    public Form1()
    {
        InitializeComponent();

        myTimer.Tick += myTimer_Tick; //分配名为 "myTimer_Tick" 的事件处理程序
    }

    private void myTimer_Tick(object sender, EventArgs e)
    {
        // 在这里执行你的操作。
    }
}
```

第110.4节：示例：使用计时器执行简单倒计时

```
public partial class Form1 : Form
{
    Timer myTimer = new Timer();
    int timeLeft = 10;

    public Form1()
    {
        InitializeComponent();

        // 设置计时器的属性
        myTimer.Interval = 1000;
        myTimer.Enabled = true;

        // 为计时器设置名为 "myTimer_Tick" 的事件处理程序
        myTimer.Tick += myTimer_Tick;

        // 表单加载后立即启动计时器
        myTimer.Start();

        // 显示 "timeLeft" 变量中设置的时间
        lblCountDown.Text = timeLeft.ToString();
    }

    private void myTimer_Tick(object sender, EventArgs e)
    {
        // 在属性中设置的间隔时间执行这些操作。
        lblCountDown.Text = timeLeft.ToString();
        timeLeft -= 1;

        if (timeLeft < 0)
        {
            myTimer.Stop();
        }
    }
}
```

Section 110.3: Assigning the "Tick" event handler to a Timer

All actions performed in a timer are handled in the "Tick" event.

```
public partial class Form1 : Form
{
    Timer myTimer = new Timer();

    public Form1()
    {
        InitializeComponent();

        myTimer.Tick += myTimer_Tick; //assign the event handler named "myTimer_Tick"
    }

    private void myTimer_Tick(object sender, EventArgs e)
    {
        // Perform your actions here.
    }
}
```

Section 110.4: Example: Using a Timer to perform a simple countdown

```
public partial class Form1 : Form
{
    Timer myTimer = new Timer();
    int timeLeft = 10;

    public Form1()
    {
        InitializeComponent();

        //set properties for the Timer
        myTimer.Interval = 1000;
        myTimer.Enabled = true;

        //Set the event handler for the timer, named "myTimer_Tick"
        myTimer.Tick += myTimer_Tick;

        //Start the timer as soon as the form is loaded
        myTimer.Start();

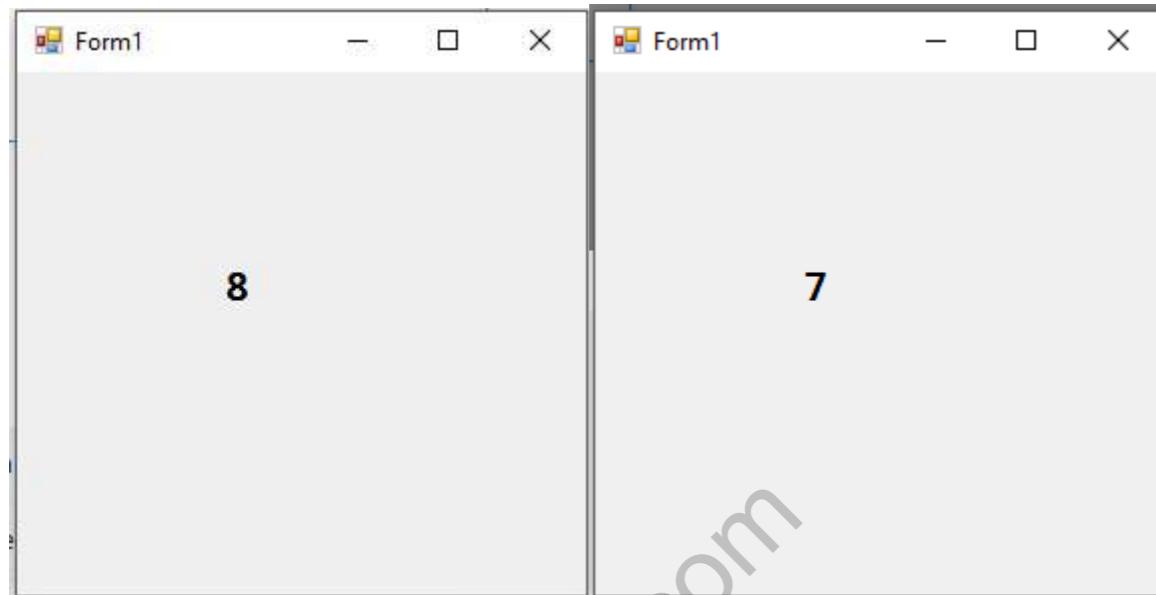
        //Show the time set in the "timeLeft" variable
        lblCountDown.Text = timeLeft.ToString();
    }

    private void myTimer_Tick(object sender, EventArgs e)
    {
        //perform these actions at the interval set in the properties.
        lblCountDown.Text = timeLeft.ToString();
        timeLeft -= 1;

        if (timeLeft < 0)
        {
            myTimer.Stop();
        }
    }
}
```

```
    }  
}
```

结果是...

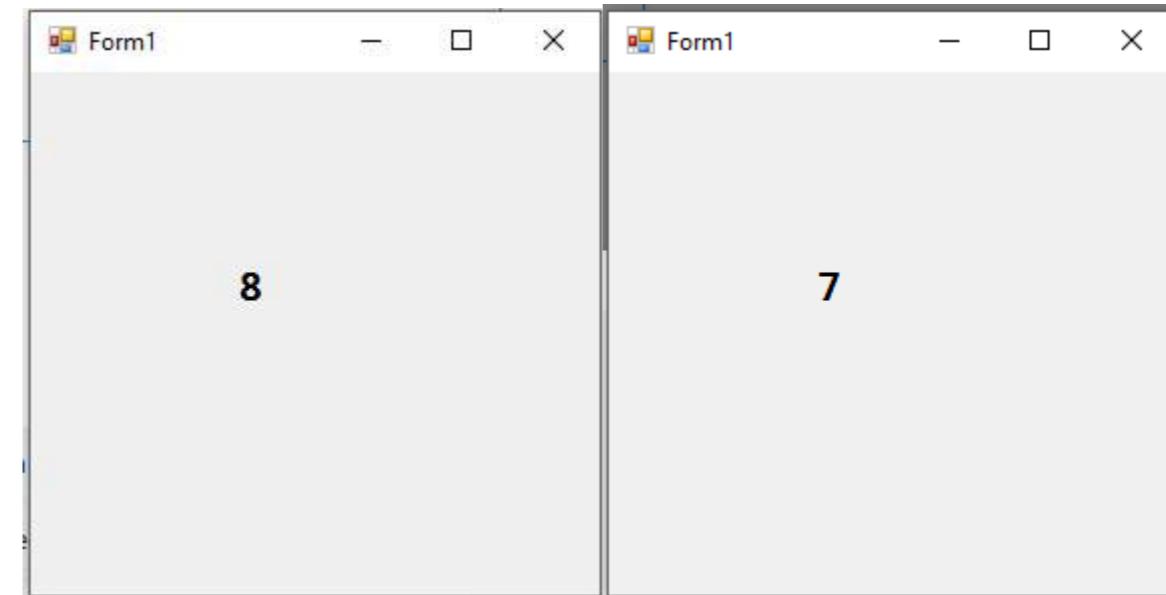


等等...

belindoc.com

```
    }  
}
```

Results in...



And so on...

第111章：秒表

第111.1节：IsHighResolution

- IsHighResolution属性指示计时器是基于高分辨率性能计数器还是基于DateTime类。

- 此字段为只读。

```
// 显示计时器频率和分辨率。
if (Stopwatch.IsHighResolution)
{
    Console.WriteLine("操作使用系统的高分辨率性能计数器计时。");
}
else
{
    Console.WriteLine("操作使用DateTime类计时。");
}

long frequency = Stopwatch.Frequency;
Console.WriteLine(" 计时器频率（每秒计时刻度数）= {0}",
    frequency);
long nanosecPerTick = (1000L * 1000L * 1000L) / frequency;
Console.WriteLine(" Timer is accurate within {0} nanoseconds",
    nanosecPerTick);
}
```

<https://dotnetfiddle.net/cKrWUo>

Stopwatch 类使用的计时器取决于系统硬件和操作系统。如果 Stopwatch 计时器基于高分辨率性能计数器，则 IsHighResolution 为 true。否则，IsHighResolution 为 false，表示 Stopwatch 计时器基于系统计时器。

Stopwatch 中的计时单位 (Ticks) 依赖于机器和操作系统，因此绝不应指望两个系统之间，甚至同一系统重启后，Stopwatch 计时单位与秒的比例相同。因此，Stopwatch 的计时单位与 DateTime/TimeSpan 的计时单位间隔不一定相同。

要获得与系统无关的时间，请确保使用 Stopwatch 的 Elapsed 或 ElapsedMilliseconds 属性，这些属性已经考虑了 Stopwatch.Frequency (每秒计时单位数)。

Stopwatch 应始终优先于 DateTime 用于计时过程，因为它更轻量，并且如果无法使用高分辨率性能计数器，则会使用 DateTime。

[来源](#)

第 111.2 节：创建 Stopwatch 实例

Stopwatch 实例可以测量多个时间间隔的累计经过时间，总经过时间是所有单独间隔的总和。这提供了一种可靠的方法来测量两个或多个事件之间的经过时间。

```
Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();

double d = 0;
for (int i = 0; i < 1000 * 1000 * 1000; i++)
{
```

Chapter 111: Stopwatches

Section 111.1: IsHighResolution

- The IsHighResolution property indicates whether the timer is based on a high-resolution performance counter or based on the DateTime class.
- This field is read-only.

```
// Display the timer frequency and resolution.
if (Stopwatch.IsHighResolution)
{
    Console.WriteLine("Operations timed using the system's high-resolution performance counter.");
}
else
{
    Console.WriteLine("Operations timed using the DateTime class.");
}

long frequency = Stopwatch.Frequency;
Console.WriteLine(" Timer frequency in ticks per second = {0}",
    frequency);
long nanosecPerTick = (1000L * 1000L * 1000L) / frequency;
Console.WriteLine(" Timer is accurate within {0} nanoseconds",
    nanosecPerTick);
}
```

<https://dotnetfiddle.net/cKrWUo>

The timer used by the Stopwatch class depends on the system hardware and operating system. IsHighResolution is true if the Stopwatch timer is based on a high-resolution performance counter. Otherwise, IsHighResolution is false, which indicates that the Stopwatch timer is based on the system timer.

Ticks in Stopwatch are machine/OS dependent, thus you should never count on the ratio of Stopwatch ticks to seconds to be the same between two systems, and possibly even on the same system after a reboot. Thus, you can never count on Stopwatch ticks to be the same interval as DateTime/TimeSpan ticks.

To get system-independent time, make sure to use the Stopwatch's Elapsed or ElapsedMilliseconds properties, which already take the Stopwatch.Frequency (ticks per second) into account.

Stopwatch should always be used over Datetime for timing processes as it is more lightweight and uses Dateime if it cant use a high-resolution performance counter.

[Source](#)

Section 111.2: Creating an Instance of a Stopwatch

A Stopwatch instance can measure elapsed time over several intervals with the total elapsed time being all individual intervals added together. This gives a reliable method of measuring elapsed time between two or more events.

```
Stopwatch stopWatch = new Stopwatch();
stopWatch.Start();

double d = 0;
for (int i = 0; i < 1000 * 1000 * 1000; i++)
{
```

```
d += 1;  
}  
  
stopWatch.Stop();  
Console.WriteLine("Time elapsed: {0:hh\\:mm\\:ss\\.fffffff}", stopWatch.Elapsed);
```

Stopwatch 位于 System.Diagnostics 中，因此你需要在文件中添加 using System.Diagnostics;

belindoc.com

```
d += 1;  
}  
  
stopWatch.Stop();  
Console.WriteLine("Time elapsed: {0:hh\\:mm\\:ss\\.fffffff}", stopWatch.Elapsed);
```

Stopwatch is in System.Diagnostics so you need to add `using System.Diagnostics;` to your file.

第112章：线程

第112.1节：避免同时读取和写入数据

有时，你希望线程能够同时共享数据。发生这种情况时，重要的是要注意代码并锁定任何可能出错的部分。下面是两个线程计数的简单示例。

这里有一些危险的（错误的）代码：

```
using System.Threading;

class MainClass
{
    static int count { get; set; }

    static void Main()
    {
        for (int i = 1; i <= 2; i++)
        {
            var thread = new Thread(ThreadMethod);
            thread.Start(i);
            Thread.Sleep(500);
        }
    }

    static void ThreadMethod(object threadNumber)
    {
        while (true)
        {
            var temp = count;
            System.Console.WriteLine("Thread " + threadNumber + ": Reading the value of count.");
            Thread.Sleep(1000);
            count = temp + 1;
            System.Console.WriteLine("Thread " + threadNumber + ": Incrementing the value of count
to:" + count);
            Thread.Sleep(1000);
        }
    }
}
```

你会注意到，我们不是按1,2,3,4,5...计数，而是按1,1,2,2,3...计数。

为了解决这个问题，我们需要**lock**计数值，这样多个不同的线程就不能同时读取和写入它。通过添加锁和钥匙，我们可以防止线程同时访问数据。

```
using System.Threading;

class MainClass
{

    static int count { get; set; }
    static readonly object key = new object();

    static void Main()
    {
        for (int i = 1; i <= 2; i++)
```

Chapter 112: Threading

Section 112.1: Avoiding Reading and Writing Data Simultaneously

Sometimes, you want your threads to simultaneously share data. When this happens it is important to be aware of the code and lock any parts that could go wrong. A simple example of two threads counting is shown below.

Here is some dangerous (incorrect) code:

```
using System.Threading;

class MainClass
{
    static int count { get; set; }

    static void Main()
    {
        for (int i = 1; i <= 2; i++)
        {
            var thread = new Thread(ThreadMethod);
            thread.Start(i);
            Thread.Sleep(500);
        }
    }

    static void ThreadMethod(object threadNumber)
    {
        while (true)
        {
            var temp = count;
            System.Console.WriteLine("Thread " + threadNumber + ": Reading the value of count.");
            Thread.Sleep(1000);
            count = temp + 1;
            System.Console.WriteLine("Thread " + threadNumber + ": Incrementing the value of count
to:" + count);
            Thread.Sleep(1000);
        }
    }
}
```

You'll notice, instead of counting 1,2,3,4,5... we count 1,1,2,2,3...

To fix this problem, we need to **lock** the value of count, so that multiple different threads cannot read and write to it at the same time. With the addition of a lock and a key, we can prevent the threads from accessing the data simultaneously.

```
using System.Threading;

class MainClass
{

    static int count { get; set; }
    static readonly object key = new object();

    static void Main()
    {
        for (int i = 1; i <= 2; i++)
```

```

    {
        var thread = new Thread(ThreadMethod);
        thread.Start(i);
        Thread.Sleep(500);
    }

    static void ThreadMethod(object threadNumber)
    {
        while (true)
        {
            lock (key)
            {
                var temp = count;
                System.Console.WriteLine("Thread " + threadNumber + ": Reading the value of
count.");
                Thread.Sleep(1000);
                count = temp + 1;
                System.Console.WriteLine("线程 " + threadNumber + ": 将count的值递增到：" + count);

            }
            Thread.Sleep(1000);
        }
    }
}

```

第112.2节：创建和启动第二个线程

如果你有多个长时间计算，可以在电脑的不同线程上同时运行它们。为此，我们创建一个新的Thread并让它指向不同的方法。

```

using System.Threading;

class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
        thread.Start();
    }

    static void Secondary() {
        System.Console.WriteLine("Hello World!");
    }
}

```

第112.3节：Parallel.ForEach 循环

如果你有一个foreach循环想要加速，并且不介意输出的顺序，可以通过以下方式将其转换为并行foreach循环：

```

using System;
using System.Threading;
using System.Threading.Tasks;

public class MainClass {

    public static void Main() {
        int[] Numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        // 单线程
        Console.WriteLine("普通 foreach 循环: ");
    }
}

```

```

    {
        var thread = new Thread(ThreadMethod);
        thread.Start(i);
        Thread.Sleep(500);
    }

    static void ThreadMethod(object threadNumber)
    {
        while (true)
        {
            lock (key)
            {
                var temp = count;
                System.Console.WriteLine("Thread " + threadNumber + ": Reading the value of
count.");
                Thread.Sleep(1000);
                count = temp + 1;
                System.Console.WriteLine("Thread " + threadNumber + ": Incrementing the value of
count to:" + count);
            }
            Thread.Sleep(1000);
        }
    }
}

```

Section 112.2: Creating and Starting a Second Thread

If you're doing multiple long calculations, you can run them at the same time on different threads on your computer. To do this, we make a new **Thread** and have it point to a different method.

```

using System.Threading;

class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
        thread.Start();
    }

    static void Secondary() {
        System.Console.WriteLine("Hello World!");
    }
}

```

Section 112.3: Parallel.ForEach Loop

If you have a foreach loop that you want to speed up and you don't mind what order the output is in, you can convert it to a parallel foreach loop by doing the following:

```

using System;
using System.Threading;
using System.Threading.Tasks;

public class MainClass {

    public static void Main() {
        int[] Numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        // Single-threaded
        Console.WriteLine("Normal foreach loop: ");
    }
}

```

```

foreach (var number in Numbers) {
    Console.WriteLine(longCalculation(number));
}
// 这是并行 (多线程) 解决方案
Console.WriteLine("并行 foreach 循环: ");
Parallel.ForEach(Numbers, number => {
    Console.WriteLine(longCalculation(number));
});
}

private static int longCalculation(int number) {
    Thread.Sleep(1000); // 休眠以模拟长时间计算
    return number * number;
}

```

第112.4节：死锁（持有资源并等待）

死锁是指两个或多个线程相互等待对方完成或释放资源，从而导致它们永远等待的情况。

如果线程1持有资源A的锁并等待资源B被释放，而线程2持有资源B并等待资源A被释放，则它们发生了死锁。

点击以下示例代码中的button1按钮将导致您的应用程序进入上述死锁状态并挂起

```

private void button_Click(object sender, EventArgs e)
{
DeadlockWorkers workers = new DeadlockWorkers();
workers.StartThreads();
textBox.Text = workers.GetResult();
}

private class DeadlockWorkers
{
Thread thread1, thread2;

object resourceA = new object();
object resourceB = new object();

string output;

public void StartThreads()
{
thread1 = new Thread(Thread1DoWork);
thread2 = new Thread(Thread2DoWork);
thread1.Start();
thread2.Start();
}

public string GetResult()
{
thread1.Join();
thread2.Join();
return output;
}

public void Thread1DoWork()
{

```

```

foreach (var number in Numbers) {
    Console.WriteLine(longCalculation(number));
}
// This is the Parallel (Multi-threaded solution)
Console.WriteLine("Parallel foreach loop: ");
Parallel.ForEach(Numbers, number => {
    Console.WriteLine(longCalculation(number));
});
}

private static int longCalculation(int number) {
    Thread.Sleep(1000); // Sleep to simulate a long calculation
    return number * number;
}

```

Section 112.4: Deadlocks (hold resource and wait)

A deadlock is what occurs when two or more threads are waiting for each other to complete or to release a resource in such a way that they wait forever.

If thread1 holds a lock on resource A and is waiting for resource B to be released while thread2 holds resource B and is waiting for resource A to be released, they are deadlocked.

Clicking button1 for the following example code will cause your application to get into aforementioned deadlocked state and hang

```

private void button_Click(object sender, EventArgs e)
{
DeadlockWorkers workers = new DeadlockWorkers();
workers.StartThreads();
textBox.Text = workers.GetResult();
}

private class DeadlockWorkers
{
Thread thread1, thread2;

object resourceA = new object();
object resourceB = new object();

string output;

public void StartThreads()
{
thread1 = new Thread(Thread1DoWork);
thread2 = new Thread(Thread2DoWork);
thread1.Start();
thread2.Start();
}

public string GetResult()
{
thread1.Join();
thread2.Join();
return output;
}

public void Thread1DoWork()
{

```

```

Thread.Sleep(100);
    lock (resourceA)
{
Thread.Sleep(100);
    lock (resourceB)
{
output += "T1#";
}
}

public void Thread2DoWork()
{
Thread.Sleep(100);
    lock (resourceB)
{
Thread.Sleep(100);
    lock (resourceA)
{
output += "T2#";
}
}
}

```

为了避免这种死锁，可以使用 `Monitor.TryEnter(lock_object, timeout_in_milliseconds)` 来检查对象上是否已经持有锁。如果 `Monitor.TryEnter` 未能在 `timeout_in_milliseconds` 时间内成功获取 `lock_object` 的锁，它会返回 `false`，给线程一个释放其他持有资源并让步的机会，从而让其他线程有机会完成，下面是上述代码稍作修改的版本：

```

private void button_Click(object sender, EventArgs e)
{
MonitorWorkers workers = new MonitorWorkers();
workers.StartThreads();
textBox.Text = workers.GetResult();
}

private class MonitorWorkers
{
Thread thread1, thread2;

object resourceA = new object();
object resourceB = new object();

string output;

public void StartThreads()
{
thread1 = new Thread(Thread1DoWork);
    thread2 = new Thread(Thread2DoWork);
    thread1.Start();
thread2.Start();
}

public string GetResult()
{
thread1.Join();
    thread2.Join();
    return output;
}
}

```

```

Thread.Sleep(100);
    lock (resourceA)
{
    Thread.Sleep(100);
    lock (resourceB)
{
        output += "T1#";
}
}

public void Thread2DoWork()
{
Thread.Sleep(100);
    lock (resourceB)
{
    Thread.Sleep(100);
    lock (resourceA)
{
        output += "T2#";
}
}
}

```

To avoid being deadlocked this way, one can use `Monitor.TryEnter(lock_object, timeout_in_milliseconds)` to check if a lock is held on an object already. If `Monitor.TryEnter` does not succeed in acquiring a lock on `lock_object` before `timeout_in_milliseconds`, it returns `false`, giving the thread a chance to release other held resources and yielding, thus giving other threads a chance to complete as in this slightly modified version of the above:

```

private void button_Click(object sender, EventArgs e)
{
MonitorWorkers workers = new MonitorWorkers();
workers.StartThreads();
textBox.Text = workers.GetResult();
}

private class MonitorWorkers
{
Thread thread1, thread2;

object resourceA = new object();
object resourceB = new object();

string output;

public void StartThreads()
{
    thread1 = new Thread(Thread1DoWork);
    thread2 = new Thread(Thread2DoWork);
    thread1.Start();
    thread2.Start();
}

public string GetResult()
{
    thread1.Join();
    thread2.Join();
    return output;
}
}

```

```

public void Thread1DoWork()
{
    bool mustDoWork = true;
    Thread.Sleep(100);
    while (mustDoWork)
    {
        lock (resourceA)
        {
            Thread.Sleep(100);
            if (Monitor.TryEnter(resourceB, 0))
            {
                output += "T1#";
                mustDoWork = false;
                Monitor.Exit(resourceB);
            }
        }
        if (mustDoWork) Thread.Yield();
    }
}

public void Thread2DoWork()
{
    Thread.Sleep(100);
    lock (resourceB)
    {
        Thread.Sleep(100);
        lock (resourceA)
        {
            output += "T2#";
        }
    }
}

```

请注意，这种解决方法依赖于线程2对其锁的固执以及线程1愿意让步，使得线程2始终优先执行。还要注意，线程1在让出时必须重新执行它在锁定资源A后所做的工作。因此，在实现多个让步线程的这种方法时要小心，因为你可能会陷入所谓的活锁——即两个线程不断执行它们工作的第一部分，然后相互让步，反复重新开始的状态。

第112.5节：简单完整的线程演示

```

class 程序
{
    static void Main(string[] args)
    {
        // 创建2个线程对象。我们使用委托是因为需要向线程传递参数。
        var thread1 = new Thread(new ThreadStart(() => PerformAction(1)));
        var thread2 = new Thread(new ThreadStart(() => PerformAction(2)));

        // 启动线程运行
        thread1.Start();
        // 注意：上述代码一启动线程，下一行代码就开始执行；
        // 即使线程1仍在处理。
        thread2.Start();

        // 等待线程1完成后再继续
        thread1.Join();
        // 等待 thread2 完成后继续
    }
}

```

```

public void Thread1DoWork()
{
    bool mustDoWork = true;
    Thread.Sleep(100);
    while (mustDoWork)
    {
        lock (resourceA)
        {
            Thread.Sleep(100);
            if (Monitor.TryEnter(resourceB, 0))
            {
                output += "T1#";
                mustDoWork = false;
                Monitor.Exit(resourceB);
            }
        }
        if (mustDoWork) Thread.Yield();
    }
}

public void Thread2DoWork()
{
    Thread.Sleep(100);
    lock (resourceB)
    {
        Thread.Sleep(100);
        lock (resourceA)
        {
            output += "T2#";
        }
    }
}

```

Note that this workaround relies on thread2 being stubborn about its locks and thread1 being willing to yield, such that thread2 always take precedence. Also note that thread1 has to redo the work it did after locking resource A, when it yields. Therefore be careful when implementing this approach with more than one yielding thread, as you'll then run the risk of entering a so-called livelock - a state which would occur if two threads kept doing the first bit of their work and then yield mutually, starting over repeatedly.

Section 112.5: Simple Complete Threading Demo

```

class Program
{
    static void Main(string[] args)
    {
        // Create 2 thread objects. We're using delegates because we need to pass
        // parameters to the threads.
        var thread1 = new Thread(new ThreadStart(() => PerformAction(1)));
        var thread2 = new Thread(new ThreadStart(() => PerformAction(2)));

        // Start the threads running
        thread1.Start();
        // NB: as soon as the above line kicks off the thread, the next line starts;
        // even if thread1 is still processing.
        thread2.Start();

        // Wait for thread1 to complete before continuing
        thread1.Join();
        // Wait for thread2 to complete before continuing
    }
}

```

```

        thread2.Join();

        Console.WriteLine("Done");
        Console.ReadKey();
    }

    // 简单方法, 用于演示线程并行运行。
    static void PerformAction(int id)
    {
        var rnd = new Random(id);
        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine("线程: {0}: {1}", id, i);
            Thread.Sleep(rnd.Next(0, 1000));
        }
    }
}

```

第112.6节：为每个处理器创建一个线程

Environment.ProcessorCount 获取当前机器上的逻辑处理器数量。

CLR 将会将每个线程调度到一个逻辑处理器，这理论上可能意味着每个线程在不同的逻辑处理器上，所有线程在同一个逻辑处理器上，或者其他组合。

```

using System;
using System.Threading;

class MainClass {
    static void Main() {
        for (int i = 0; i < Environment.ProcessorCount; i++) {
            var thread = new Thread(Secondary);
            thread.Start(i);
        }
    }

    static void Secondary(object threadNumber) {
        System.Console.WriteLine("Hello World from thread: " + threadNumber);
    }
}

```

第112.7节：使用任务的简单完整线程演示

```

class 程序
{
    static void Main(string[] args)
    {
        // 运行2个任务。
        var task1 = Task.Run(() => PerformAction(1));
        var task2 = Task.Run(() => PerformAction(2));

        // 等待 (即阻塞此线程) 直到两个任务都完成。
        Task.WaitAll(new [] { task1, task2 });

        Console.WriteLine("Done");
        Console.ReadKey();
    }
}

```

```

        thread2.Join();

        Console.WriteLine("Done");
        Console.ReadKey();
    }

    // Simple method to help demonstrate the threads running in parallel.
    static void PerformAction(int id)
    {
        var rnd = new Random(id);
        for (int i = 0; i < 100; i++)
        {
            Console.WriteLine("Thread: {0}: {1}", id, i);
            Thread.Sleep(rnd.Next(0, 1000));
        }
    }
}

```

Section 112.6: Creating One Thread Per Processor

Environment.ProcessorCount Gets the number of **logical** processors on the current machine.

The CLR will then schedule each thread to a logical processor, this theoretically could mean each thread on a different logical processor, all threads on a single logical processor or some other combination.

```

using System;
using System.Threading;

class MainClass {
    static void Main() {
        for (int i = 0; i < Environment.ProcessorCount; i++) {
            var thread = new Thread(Secondary);
            thread.Start(i);
        }
    }

    static void Secondary(object threadNumber) {
        System.Console.WriteLine("Hello World from thread: " + threadNumber);
    }
}

```

Section 112.7: Simple Complete Threading Demo using Tasks

```

class Program
{
    static void Main(string[] args)
    {
        // Run 2 Tasks.
        var task1 = Task.Run(() => PerformAction(1));
        var task2 = Task.Run(() => PerformAction(2));

        // Wait (i.e. block this thread) until both Tasks are complete.
        Task.WaitAll(new [] { task1, task2 });

        Console.WriteLine("Done");
        Console.ReadKey();
    }
}

```

```
// 简单方法，用于演示线程并行运行。
static void PerformAction(int id)
{
    var rnd = new Random(id);
    for (int i = 0; i < 100; i++)
    {
        Console.WriteLine("任务: {0}: {1}", id, i);
        Thread.Sleep(rnd.Next(0, 1000));
    }
}
```

```
// Simple method to help demonstrate the threads running in parallel.
static void PerformAction(int id)
{
    var rnd = new Random(id);
    for (int i = 0; i < 100; i++)
    {
        Console.WriteLine("Task: {0}: {1}", id, i);
        Thread.Sleep(rnd.Next(0, 1000));
    }
}
```

第112.8节：死锁（两个线程相互等待）

死锁是指两个或多个线程相互等待对方完成或释放资源，从而导致它们永远等待的情况。

两个线程相互等待完成的典型场景是，当Windows窗体GUI线程等待一个工作线程，而该工作线程试图调用由GUI线程管理的对象。请注意，使用此代码示例，点击button1将导致程序挂起。

```
private void button1_Click(object sender, EventArgs e)
{
    Thread workerthread= new Thread(dowork);
    workerthread.Start();
    workerthread.Join();
    // 之后执行某些操作
}

private void dowork()
{
    // 之前执行某些操作
    textBox1.Invoke(new Action(() => textBox1.Text = "Some Text"));
    // 之后执行某些操作
}
```

workerthread.Join() 是一个阻塞调用，直到workerthread完成才返回。

textBox1.Invoke(invoker_delegate) 是一个阻塞调用，直到GUI线程处理完invoker_delegate才返回，但如果GUI线程已经在等待调用线程完成，则此调用会导致死锁。

为了解决这个问题，可以使用一种非阻塞的方式来调用文本框：

```
private void dowork()
{
    // 执行工作
    textBox1.BeginInvoke(new Action(() => textBox1.Text = "Some Text"));
    // 执行不依赖于先更新 textBox1 的工作
}
```

但是，如果你需要运行依赖于文本框先被更新的代码，这将会引起问题。

在这种情况下，将该代码作为 invoke 的一部分运行，但要注意这会使其在 GUI 线程上运行。

```
private void dowork()
{
    // 执行工作
    textBox1.BeginInvoke(new Action(() => {
        textBox1.Text = "Some Text";
    }));
```

Section 112.8: Deadlocks (two threads waiting on each other)

A deadlock is what occurs when two or more threads are waiting for each other to complete or to release a resource in such a way that they wait forever.

A typical scenario of two threads waiting on each other to complete is when a Windows Forms GUI thread waits for a worker thread and the worker thread attempts to invoke an object managed by the GUI thread. Observe that with this code example, clicking button1 will cause the program to hang.

```
private void button1_Click(object sender, EventArgs e)
{
    Thread workerthread= new Thread(dowork);
    workerthread.Start();
    workerthread.Join();
    // Do something after
}

private void dowork()
{
    // Do something before
    textBox1.Invoke(new Action(() => textBox1.Text = "Some Text"));
    // Do something after
}
```

workerthread.Join() is a call that blocks the calling thread until workerthread completes.

textBox1.Invoke(invoker_delegate) is a call that blocks the calling thread until the GUI thread has processed invoker_delegate, but this call causes deadlocks if the GUI thread is already waiting for the calling thread to complete.

To get around this, one can use a non-blocking way of invoking the textbox instead:

```
private void dowork()
{
    // Do work
    textBox1.BeginInvoke(new Action(() => textBox1.Text = "Some Text"));
    // Do work that is not dependent on textBox1 being updated first
}
```

However, this will cause trouble if you need to run code that is dependent on the textbox being updated first. In that case, run that as part of the invoke, but be aware that this will make it run on the GUI thread.

```
private void dowork()
{
    // Do work
    textBox1.BeginInvoke(new Action(() => {
        textBox1.Text = "Some Text";
    }));
```

```

    // 执行依赖于先更新 textBox1 的工作,
    // 启动另一个工作线程或触发一个事件
});
// 执行不依赖于先更新 textBox1 的工作
}

```

或者，启动一个全新的线程，让它在 GUI 线程上等待，这样工作线程可能会完成。

```

private void dowork()
{
    // 执行工作
线程 workerthread2 = new 线程(() =>
{
    textBox1.Invoke(new Action(() => textBox1.Text = "Some Text"));
    // 执行依赖于先更新 textBox1 的工作,
    // 启动另一个工作线程或触发一个事件
});
workerthread2.Start();
    // 执行不依赖于先更新 textBox1 的工作
}

```

为了尽量减少发生线程相互等待死锁的风险，应尽可能避免线程之间的循环引用。线程层级中，低级线程仅向高级线程发送消息且绝不等待它们，将不会遇到此类问题。然而，这种方式仍然可能因资源锁定而导致死锁。

第112.9节：显式任务并行

```

private static void explicitTaskParallism()
{
    Thread.CurrentThread.Name = "Main";

    // 使用lambda表达式创建任务并提供用户委托。
    Task taskA = new Task(() => Console.WriteLine($"Hello from task {nameof(taskA)}."));
    Task taskB = new Task(() => Console.WriteLine($"Hello from task {nameof(taskB)}."));

    // 启动任务。
    taskA.Start();
    taskB.Start();

    // 从调用线程输出一条消息。
    Console.WriteLine("来自线程 '{0}' 的问候。",
        Thread.CurrentThread.Name);
    taskA.Wait();
    taskB.Wait();
    Console.Read();
}

```

第112.10节：隐式任务并行

```

private static void Main(string[] args)
{
    var a = new A();
    var b = new B();
    // 隐式任务并行
    Parallel.Invoke(
        () => a.DoSomeWork(),
        () => b.DoSomeWork()
    );
}

```

```

    // Do work dependent on textBox1 being updated first,
    // start another worker thread or raise an event
});
// Do work that is not dependent on textBox1 being updated first
}

```

Alternatively start a whole new thread and let that one do the waiting on the GUI thread, so that worker thread might complete.

```

private void dowork()
{
    // Do work
    Thread workerthread2 = new Thread(() =>
{
    textBox1.Invoke(new Action(() => textBox1.Text = "Some Text"));
    // Do work dependent on textBox1 being updated first,
    // start another worker thread or raise an event
});
    workerthread2.Start();
    // Do work that is not dependent on textBox1 being updated first
}

```

To minimize the risk of running into a deadlock of mutual waiting, always avoid circular references between threads when possible. A hierarchy of threads where lower-ranking threads only leave messages for higher-ranking threads and never waiting on them will not run into this kind of issue. However, it would still be vulnerable to deadlocks based on resource locking.

Section 112.9: Explicit Task Parallelism

```

private static void explicitTaskParallism()
{
    Thread.CurrentThread.Name = "Main";

    // Create a task and supply a user delegate by using a lambda expression.
    Task taskA = new Task(() => Console.WriteLine($"Hello from task {nameof(taskA)}."));
    Task taskB = new Task(() => Console.WriteLine($"Hello from task {nameof(taskB)}."));

    // Start the task.
    taskA.Start();
    taskB.Start();

    // Output a message from the calling thread.
    Console.WriteLine("Hello from thread '{0}'.",
        Thread.CurrentThread.Name);
    taskA.Wait();
    taskB.Wait();
    Console.Read();
}

```

Section 112.10: Implicit Task Parallelism

```

private static void Main(string[] args)
{
    var a = new A();
    var b = new B();
    // implicit task parallelism
    Parallel.Invoke(
        () => a.DoSomeWork(),
        () => b.DoSomeWork()
    );
}

```

```
(() => b.DoSomeOtherWork()
);  
}
```

第112.11节：带参数启动线程

```
using System.Threading;
```

```
class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
        thread.Start("SecondThread");
    }

    static void Secondary(object threadName) {
        System.Console.WriteLine("Hello World from thread: " + threadName);
    }
}
```

```
(() => b.DoSomeOtherWork()
);  
}
```

Section 112.11: Starting a thread with parameters

```
using System.Threading;
```

```
class MainClass {
    static void Main() {
        var thread = new Thread(Secondary);
        thread.Start("SecondThread");
    }

    static void Secondary(object threadName) {
        System.Console.WriteLine("Hello World from thread: " + threadName);
    }
}
```

belindoc.com

第113章：异步/等待 (Async/await)、Backgroundworker、任务 (Task) 和线程示例

第113.1节：ASP.NET 配置 Await

当 ASP.NET 处理请求时，会从线程池分配一个线程，并创建一个**请求上下文**。该请求上下文包含有关当前请求的信息，可以通过静态 `HttpContext.Current` 属性访问。然后，将请求的请求上下文分配给处理该请求的线程。

给定的请求上下文一次只能在一个线程上处于活动状态。

当执行到 `await` 时，处理请求的线程会返回线程池，而异步方法继续运行，请求上下文则可供另一个线程使用。

```
public async Task<ActionResult> Index()
{
    // 最初分配的线程上执行
    var products = await dbContext.Products.ToListAsync();

    // 执行在线程池中的“随机”线程上恢复
    // 执行继续使用原始请求上下文。
    return View(products);
}
```

当任务完成时，线程池会分配另一个线程继续执行请求。请求上下文随后被分配给该线程。这个线程可能是也可能不是原始线程。

阻塞

当同步等待一个 `async` 方法调用的结果时，可能会产生死锁。例如，以下代码在调用 `IndexSync()` 时会导致死锁：

```
public async Task<ActionResult> Index()
{
    // 最初分配的线程上执行
    List<Product> products = await dbContext.Products.ToListAsync();

    // 执行在线程池中的“随机”线程上恢复
    return View(products);
}

public ActionResult IndexSync()
{
    Task<ActionResult> task = Index();

    // 同步阻塞等待结果
    ActionResult result = Task.Result;

    return result;
}
```

这是因为，默认情况下，被等待的任务（在本例中为 `db.Products.ToListAsync()`）会捕获上下文（在 ASP.NET 中是请求上下文），并在完成后尝试使用该上下文。

Chapter 113: Async/await, Backgroundworker, Task and Thread Examples

Section 113.1: ASP.NET Configure Await

When ASP.NET handles a request, a thread is assigned from the thread pool and a **request context** is created. The request context contains information about the current request which can be accessed through the static `HttpContext.Current` property. The request context for the request is then assigned to the thread handling the request.

A given request context **may only be active on one thread at a time**.

When execution reaches `await`, the thread handling a request is returned to the thread pool while the asynchronous method runs and the request context is free for another thread to use.

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    var products = await dbContext.Products.ToListAsync();

    // Execution resumes on a "random" thread from the pool
    // Execution continues using the original request context.
    return View(products);
}
```

When the task completes the thread pool assigns another thread to continue execution of the request. The request context is then assigned to this thread. This may or may not be the original thread.

Blocking

When the result of an `async` method call is waited for **synchronously** deadlocks can arise. For example the following code will result in a deadlock when `IndexSync()` is called:

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    List<Product> products = await dbContext.Products.ToListAsync();

    // Execution resumes on a "random" thread from the pool
    return View(products);
}

public ActionResult IndexSync()
{
    Task<ActionResult> task = Index();

    // Block waiting for the result synchronously
    ActionResult result = Task.Result;

    return result;
}
```

This is because, by default the awaited task, in this case `db.Products.ToListAsync()` will capture the context (in the case of ASP.NET the request context) and try to use it once it has completed.

当整个调用栈都是异步时不会有问题，因为一旦达到 await，原始线程就会被释放，从而释放请求上下文。

当我们使用 Task.Result 或 Task.Wait()（或其他阻塞方法）同步阻塞时，原始线程仍然处于活动状态并保留请求上下文。被等待的方法仍然异步运行，一旦回调尝试执行，即等待的任务返回时，它会尝试获取请求上下文。

因此，死锁产生的原因是阻塞线程持有请求上下文等待异步操作完成，而异步操作又试图获取请求上下文以完成操作。

ConfigureAwait

默认情况下，对一个等待的任务的调用会捕获当前上下文，并尝试在任务完成后在该上下文中恢复执行。

通过使用ConfigureAwait(false)，可以阻止这种行为，从而避免死锁。

```
public async Task<ActionResult> Index()
{
    // 最初分配的线程上执行
    List<Product> products = await dbContext.Products.ToListAsync().ConfigureAwait(false);

    // 执行将在没有原始请求上下文的线程池中的“随机”线程上恢复
    return View(products);
}

public ActionResult IndexSync()
{
    Task<ActionResult> task = Index();

    // 同步阻塞等待结果
    ActionResult result = Task.Result;

    return result;
}
```

当必须阻塞异步代码时，这可以避免死锁，然而代价是继续执行（await调用之后的代码）时会丢失上下文。

在ASP.NET中，这意味着如果你在调用await someTask.ConfigureAwait(false);之后的代码尝试访问上下文中的信息，例如HttpContext.Current.User，那么这些信息已经丢失。

在这种情况下，HttpContext.Current为null。例如：

```
public async Task<ActionResult> Index()
{
    // 包含发送请求的用户信息
    var user = System.Web.HttpContext.Current.User;

    using (var client = new HttpClient())
    {
        await client.GetAsync("http://google.com").ConfigureAwait(false);
    }

    // 空引用异常，Current 为 null
    var user2 = System.Web.HttpContext.Current.User;

    return View();
}
```

When the entire call stack is asynchronous there is no problem because, once `await` is reached the original thread is released, freeing the request context.

When we block synchronously using `Task.Result` or `Task.Wait()` (or other blocking methods) the original thread is still active and retains the request context. The awaited method still operates asynchronously and once the callback tries to run, i.e. once the awaited task has returned, it attempts to obtain the request context.

Therefore the deadlock arises because while the blocking thread with the request context is waiting for the asynchronous operation to complete, the asynchronous operation is trying to obtain the request context in order to complete.

ConfigureAwait

By default calls to an awaited task will capture the current context and attempt to resume execution on the context once complete.

By using `ConfigureAwait(false)` this can be prevented and deadlocks can be avoided.

```
public async Task<ActionResult> Index()
{
    // Execution on the initially assigned thread
    List<Product> products = await dbContext.Products.ToListAsync().ConfigureAwait(false);

    // Execution resumes on a "random" thread from the pool without the original request context
    return View(products);
}

public ActionResult IndexSync()
{
    Task<ActionResult> task = Index();

    // Block waiting for the result synchronously
    ActionResult result = Task.Result;

    return result;
}
```

This can avoid deadlocks when it is necessary to block on asynchronous code, however this comes at the cost of losing the context in the continuation (code after the call to await).

In ASP.NET this means that if your code following a call to `await someTask.ConfigureAwait(false)` attempts to access information from the context, for example `HttpContext.Current.User` then the information has been lost. In this case the `HttpContext.Current` is null. For example:

```
public async Task<ActionResult> Index()
{
    // Contains information about the user sending the request
    var user = System.Web.HttpContext.Current.User;

    using (var client = new HttpClient())
    {
        await client.GetAsync("http://google.com").ConfigureAwait(false);
    }

    // Null Reference Exception, Current is null
    var user2 = System.Web.HttpContext.Current.User;

    return View();
}
```

```
}
```

如果使用ConfigureAwait(true)（等同于根本不使用ConfigureAwait），那么user和user2都会被填充相同的数据。

因此，通常建议在不再使用上下文的库代码中使用ConfigureAwait(false)。

第113.2节：任务“运行并忘记”扩展

在某些情况下（例如日志记录），运行任务但不等待结果可能很有用。以下扩展允许运行任务并继续执行后续代码：

```
public static class TaskExtensions
{
    public static async void RunAndForget(
        this Task task, Action<Exception> onException = null)
    {
        try
        {
            await task;
        }
        catch (Exception ex)
        {
            onException?.Invoke(ex);
        }
    }
}
```

结果仅在扩展方法内部等待。由于使用了async/await，可以捕获异常并调用可选的处理方法。

如何使用该扩展方法的示例：

```
var task = Task.FromResult(0); // 或者来自例如外部库的其他任务。
task.RunAndForget(
    e =>
{
    // 出现了问题，进行处理。
});
```

第113.3节：Async/await

下面是一个简单示例，展示如何使用async/await在后台进程中执行一些耗时操作，同时保持执行其他不需要等待耗时操作完成的任务的选项。

但是，如果你需要稍后使用耗时方法的结果，可以通过等待执行来实现。

```
public async Task ProcessDataAsync()
{
    // 启动耗时方法
    任务<int> 任务 = 耗时方法(@"PATH_TO_SOME_FILE");

    // 在 TimeintensiveMethod 返回之前，控制权返回到这里
    Console.WriteLine("当 TimeintensiveMethod 仍在运行时，你可以读取这条信息。");
}
```

```
}
```

If ConfigureAwait(true) is used (equivalent to having no ConfigureAwait at all) then both user and user2 are populated with the same data.

For this reason it is often recommended to use ConfigureAwait(false) in library code where the context is no longer used.

Section 113.2: Task "run and forget" extension

In certain cases (e.g. logging) it might be useful to run task and do not await for the result. The following extension allows to run task and continue execution of the rest code:

```
public static class TaskExtensions
{
    public static async void RunAndForget(
        this Task task, Action<Exception> onException = null)
    {
        try
        {
            await task;
        }
        catch (Exception ex)
        {
            onException?.Invoke(ex);
        }
    }
}
```

The result is awaited only inside the extension method. Since `async/await` is used, it is possible to catch an exception and call an optional method for handling it.

An example how to use the extension:

```
var task = Task.FromResult(0); // Or any other task from e.g. external lib.
task.RunAndForget(
    e =>
{
    // Something went wrong, handle it.
});
```

Section 113.3: Async/await

See below for a simple example of how to use `async/await` to do some time intensive stuff in a background process while maintaining the option of doing some other stuff that do not need to wait on the time intensive stuff to complete.

However, if you need to work with the result of the time intensive method later, you can do this by awaiting the execution.

```
public async Task ProcessDataAsync()
{
    // Start the time intensive method
    Task<int> task = TimeintensiveMethod(@"PATH_TO_SOME_FILE");

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");
}
```

```

// 等待 TimeintensiveMethod 完成并获取其结果
int x = await task;
Console.WriteLine("计数: " + x);
}

private async Task<int> TimeintensiveMethod(object file)
{
Console.WriteLine("开始 TimeintensiveMethod。");

// 进行一些耗时的计算...
using (StreamReader reader = new StreamReader(file.ToString()))
{
    string s = await reader.ReadToEndAsync();

    for (int i = 0; i < 10000; i++)
        s.GetHashCode();
}
Console.WriteLine("End TimeintensiveMethod.");

// 返回某个“结果”
return new Random().Next(100);
}

```

第113.4节：BackgroundWorker

下面是一个简单示例，展示如何使用BackgroundWorker对象在后台线程中执行耗时操作。

您需要：

1. 定义一个执行耗时工作的工作方法，并从BackgroundWorker的DoWork事件处理程序中调用它。BackgroundWorker的DoWork事件。
2. 使用RunWorkerAsync启动执行。任何工作方法所需的参数都附加到DoWork事件处理程序。可以通过DoWorkEventArgs参数传递给RunWorkerAsync。

除了DoWork事件，BackgroundWorker类还定义了两个用于与用户界面交互的事件。这些事件是可选的。

- RunWorkerCompleted事件在DoWork处理程序完成时触发。
- ProgressChanged事件在调用ReportProgress方法时触发。

```

public void ProcessDataAsync()
{
    // 启动耗时方法
    BackgroundWorker bw = new BackgroundWorker();
    bw.DoWork += BwDoWork;
    bw.RunWorkerCompleted += BwRunWorkerCompleted;
    bw.RunWorkerAsync(@"PATH_TO_SOME_FILE");

    // 在 TimeintensiveMethod 返回之前，控制权返回到这里
    Console.WriteLine("当TimeintensiveMethod仍在运行时，你可以读取此内容。");
}

// BwDoWork退出后将调用的方法
private void BwRunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    // 我们可以通过参数e访问方法可能的返回值
    Console.WriteLine("计数: " + e.Result);
}

```

```

// Wait for TimeintensiveMethod to complete and get its result
int x = await task;
Console.WriteLine("Count: " + x);
}

private async Task<int> TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod。");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = await reader.ReadToEndAsync();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod。");

    // return something as a "result"
    return new Random().Next(100);
}

```

Section 113.4: BackgroundWorker

See below for a simple example of how to use a BackgroundWorker object to perform time-intensive operations in a background thread.

You need to:

1. Define a worker method that does the time-intensive work and call it from an event handler for the DoWork event of a BackgroundWorker.
2. Start the execution with RunWorkerAsync. Any argument required by the worker method attached to DoWork can be passed in via the DoWorkEventArgs parameter to RunWorkerAsync.

In addition to the DoWork event the BackgroundWorker class also defines two events that should be used for interacting with the user interface. These are optional.

- The RunWorkerCompleted event is triggered when the DoWork handlers have completed.
- The ProgressChanged event is triggered when the ReportProgress method is called.

```

public void ProcessDataAsync()
{
    // Start the time intensive method
    BackgroundWorker bw = new BackgroundWorker();
    bw.DoWork += BwDoWork;
    bw.RunWorkerCompleted += BwRunWorkerCompleted;
    bw.RunWorkerAsync(@"PATH_TO_SOME_FILE");

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");
}

// Method that will be called after BwDoWork exits
private void BwRunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    // we can access possible return values of our Method via the Parameter e
    Console.WriteLine("Count: " + e.Result);
}

```

```

// 执行我们耗时的方法
private void BwDoWork(object sender, DoWorkEventArgs e)
{
    e.Result = TimeintensiveMethod(e.Argument);
}

private int TimeintensiveMethod(object file)
{
    Console.WriteLine("开始 TimeintensiveMethod。");

    // 进行一些耗时的计算...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // 返回某个“结果”
    return new Random().Next(100);
}

```

第113.5节：任务

下面是一个简单示例，展示如何使用Task在后台进程中执行一些耗时操作。

你所需要做的就是将耗时的方法包装在一个Task.Run()调用中。

```

public void ProcessDataAsync()
{
    // 启动耗时方法
    Task<int> t = Task.Run(() => TimeintensiveMethod(@"PATH_TO_SOME_FILE"));

    // 在 TimeintensiveMethod 返回之前，控制权返回到这里
    Console.WriteLine("当 TimeintensiveMethod 仍在运行时，您可以读取此内容。");

    Console.WriteLine("计数: " + t.Result);
}

private int TimeintensiveMethod(object file)
{
    Console.WriteLine("开始 TimeintensiveMethod。");

    // 进行一些耗时的计算...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // 返回某个“结果”
    return new Random().Next(100);
}

```

```

// execution of our time intensive Method
private void BwDoWork(object sender, DoWorkEventArgs e)
{
    e.Result = TimeintensiveMethod(e.Argument);
}

private int TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod。");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}

```

Section 113.5: Task

See below for a simple example of how to use a Task to do some time intensive stuff in a background process.

All you need to do is wrap your time intensive method in a Task.Run() call.

```

public void ProcessDataAsync()
{
    // Start the time intensive method
    Task<int> t = Task.Run(() => TimeintensiveMethod(@"PATH_TO_SOME_FILE"));

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");

    Console.WriteLine("Count: " + t.Result);
}

private int TimeintensiveMethod(object file)
{
    Console.WriteLine("Start TimeintensiveMethod。");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file.ToString()))
    {
        string s = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something as a "result"
    return new Random().Next(100);
}

```

第113.6节：线程

以下是一个简单示例，展示如何使用Thread在后台进程中执行一些耗时操作。

```
public async void ProcessDataAsync()
{
    // 启动耗时方法
    Thread t = new Thread(TimeintensiveMethod);

    // 在 TimeintensiveMethod 返回之前，控制权返回到这里
    Console.WriteLine("当TimeintensiveMethod仍在运行时，你可以读取此内容。");
}

private void TimeintensiveMethod()
{
    Console.WriteLine("开始 TimeintensiveMethod。");

    // 进行一些耗时的计算...
    using (StreamReader reader = new StreamReader(@"PATH_TO_SOME_FILE"))
    {
        string v = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            v.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");
}
```

正如你所见，我们无法从TimeIntensiveMethod返回值，因为Thread期望一个返回void的方法作为其参数。

要从Thread获取返回值，可以使用事件或以下方法：

```
int ret;
Thread t= new Thread(() =>
{
    Console.WriteLine("开始 TimeintensiveMethod。");

    // 进行一些耗时的计算...
    using (StreamReader reader = new StreamReader(file))
    {
        string s = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // 返回一些东西以展示await-async的酷炫
    ret = new Random().Next(100);
});

t.Start();
t.Join(1000);
Console.WriteLine("Count: " + ret);
```

Section 113.6: Thread

See below for a simple example of how to use a Thread to do some time intensive stuff in a background process.

```
public async void ProcessDataAsync()
{
    // Start the time intensive method
    Thread t = new Thread(TimeintensiveMethod);

    // Control returns here before TimeintensiveMethod returns
    Console.WriteLine("You can read this while TimeintensiveMethod is still running.");
}

private void TimeintensiveMethod()
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(@"PATH_TO_SOME_FILE"))
    {
        string v = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            v.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");
}
```

As you can see we can not return a value from our TimeIntensiveMethod because Thread expects a void Method as its parameter.

To get a return value from a Thread use either an event or the following:

```
int ret;
Thread t= new Thread(() =>
{
    Console.WriteLine("Start TimeintensiveMethod.");

    // Do some time intensive calculations...
    using (StreamReader reader = new StreamReader(file))
    {
        string s = reader.ReadToEnd();

        for (int i = 0; i < 10000; i++)
            s.GetHashCode();
    }
    Console.WriteLine("End TimeintensiveMethod.");

    // return something to demonstrate the coolness of await-async
    ret = new Random().Next(100);
});

t.Start();
t.Join(1000);
Console.WriteLine("Count: " + ret);
```

第114章：异步-等待（Async-Await）

在C#中，声明为`async`的方法在同步进程中不会阻塞，尤其是在使用基于I/O的操作（例如网络访问、文件处理等）时。此类标记为`async`的方法的结果可以通过使用`await`关键字来等待。

第114.1节：Await操作符和async关键字

`await`操作符和`async`关键字是配套使用的：

使用`await`的异步方法必须用`async`关键字修饰。

反之不一定成立：你可以将方法标记为`async`而不在其主体中使用`await`。

`await`的实际作用是暂停代码执行，直到被等待的任务完成；任何任务都可以被等待。

注意：你不能等待返回`void`的异步方法。

实际上，“挂起”这个词有点误导，因为不仅执行停止，线程还可能变得可用于执行其他操作。在底层，`await`是通过编译器的一些魔法实现的：它将一个方法分成两部分——`await`之前和之后。后半部分在等待的任务完成时执行。

如果忽略一些重要细节，编译器大致为你做了以下操作：

```
public async Task<TResult> DoIt()
{
    // 执行一些操作并获取类型为 Task<TSomeResult> 的 someTask
    var awaitedResult = await someTask;
    // ... 继续执行一些操作并生成类型为 TResult 的结果
    return result;
}
```

变成：

```
public Task<TResult> DoIt()
{
    // ...
    return someTask.ContinueWith(task => {
        var result = ((Task<TSomeResult>)task).Result;
        return DoIt_Continuation(result);
    });
}

private TResult DoIt_Continuation(TSomeResult awaitedResult)
{
    // ...
}
```

任何常用方法都可以通过以下方式转换为异步：

```
await Task.Run(() => YourSyncMethod());
```

当你需要在UI线程上执行一个长时间运行的方法而不冻结

Chapter 114: Async-Await

In C#, a method declared `async` won't block within a synchronous process, in case of you're using I/O based operations (e.g. web access, working with files, ...). The result of such `async` marked methods may be awaited via the use of the `await` keyword.

Section 114.1: Await operator and async keyword

`await` operator and `async` keyword come together:

The asynchronous method in which `await` is used must be modified by the `async` keyword.

The opposite is not always true: you can mark a method as `async` without using `await` in its body.

What `await` actually does is to suspend execution of the code until the awaited task completes; any task can be awaited.

Note: you cannot await for `async` method which returns nothing (`void`).

Actually, the word 'suspends' is a bit misleading because not only the execution stops, but the thread may become free for executing other operations. Under the hood, `await` is implemented by a bit of compiler magic: it splits a method into two parts - before and after `await`. The latter part is executed when the awaited task completes.

If we ignore some important details, the compiler roughly does this for you:

```
public async Task<TResult> DoIt()
{
    // do something and acquire someTask of type Task<TSomeResult>
    var awaitedResult = await someTask;
    // ... do something more and produce result of type TResult
    return result;
}
```

becomes:

```
public Task<TResult> DoIt()
{
    // ...
    return someTask.ContinueWith(task => {
        var result = ((Task<TSomeResult>)task).Result;
        return DoIt_Continuation(result);
    });
}

private TResult DoIt_Continuation(TSomeResult awaitedResult)
{
    // ...
}
```

Any usual method can be turned into `async` in the following way:

```
await Task.Run(() => YourSyncMethod());
```

This can be advantageous when you need to execute a long running method on the UI thread without freezing the

但这里有一个非常重要的说明：异步并不总是意味着并发（并行甚至多线程）。即使在单线程上，`async-await`仍然允许异步代码。例如，参见这个自定义任务调度器。这样的“疯狂”任务调度器可以简单地将任务转换为在消息循环处理中调用的函数。

我们需要问自己：哪个线程将执行我们方法`DoIt_Continuation`的后续操作？

默认情况下，`await`操作符会使用当前的同步上下文来调度后续操作的执行。这意味着默认情况下，对于WinForms和WPF，后续操作在UI线程中运行。如果出于某种原因，你需要改变这种行为，可以使用方法`Task.ConfigureAwait()`：

```
await Task.Run(() => YourSyncMethod()).ConfigureAwait(continueOnCapturedContext: false);
```

第114.2节：并发调用

可以通过先调用可等待的任务，然后再等待它们，实现同时等待多个调用。

```
public async Task RunConcurrentTasks()
{
    var firstTask = DoSomethingAsync();
    var secondTask = DoSomethingElseAsync();

    await firstTask;
    await secondTask;
}
```

或者，可以使用`Task.WhenAll`将多个任务组合成一个单一的Task，当所有传入的任务完成时，该任务也完成。

```
public async Task RunConcurrentTasks()
{
    var firstTask = DoSomethingAsync();
    var secondTask = DoSomethingElseAsync();

    await Task.WhenAll(firstTask, secondTask);
}
```

你也可以在循环中这样做，例如：

```
List<Task> tasks = new List<Task>();
while (something) {
    // 执行操作
    Task someAsyncTask = someAsyncMethod();
    tasks.Add(someAsyncTask);
}

await Task.WhenAll(tasks);
```

在使用`Task.WhenAll`等待多个任务后，要获取某个任务的结果，只需再次等待该任务。由于该任务已经完成，它会直接返回结果。

```
var task1 = SomeOpAsync();
var task2 = SomeOtherOpAsync();
```

UI.

But there is a very important remark here: **Asynchronous does not always mean concurrent (parallel or even multi-threaded)**. Even on a single thread, `async-await` still allows for asynchronous code. For example, see this custom [task scheduler](#). Such a 'crazy' task scheduler can simply turn tasks into functions which are called within message loop processing.

We need to ask ourselves: What thread will execute the continuation of our method `DoIt_Continuation`?

By default the `await` operator schedules the execution of continuation with the current [Synchronization context](#). It means that by default for WinForms and WPF continuation runs in the UI thread. If, for some reason, you need to change this behavior, use [method](#) `Task.ConfigureAwait()`:

```
await Task.Run(() => YourSyncMethod()).ConfigureAwait(continueOnCapturedContext: false);
```

Section 114.2: Concurrent calls

It is possible to await multiple calls concurrently by first invoking the awaitable tasks and *then* awaiting them.

```
public async Task RunConcurrentTasks()
{
    var firstTask = DoSomethingAsync();
    var secondTask = DoSomethingElseAsync();

    await firstTask;
    await secondTask;
}
```

Alternatively, `Task.WhenAll` can be used to group multiple tasks into a single Task, which completes when all of its passed tasks are complete.

```
public async Task RunConcurrentTasks()
{
    var firstTask = DoSomethingAsync();
    var secondTask = DoSomethingElseAsync();

    await Task.WhenAll(firstTask, secondTask);
}
```

You can also do this inside a loop, for example:

```
List<Task> tasks = new List<Task>();
while (something) {
    // do stuff
    Task someAsyncTask = someAsyncMethod();
    tasks.Add(someAsyncTask);
}

await Task.WhenAll(tasks);
```

To get results from a task after awaiting multiple tasks with `Task.WhenAll`, simply await the task again. Since the task is already completed it will just return the result back

```
var task1 = SomeOpAsync();
var task2 = SomeOtherOpAsync();
```

```
await Task.WhenAll(task1, task2);

var result = await task2;
```

此外, Task.WhenAny 可以用来并行执行多个任务, 类似于上面的 Task.WhenAll, 不同之处在于该方法会在任意一个提供的任务完成时结束。

```
public async Task RunConcurrentTasksWhenAny()
{
    var firstTask = TaskOperation("#firstTask executed");
    var secondTask = TaskOperation("#secondTask executed");
    var thirdTask = TaskOperation("#thirdTask executed");
    await Task.WhenAny(firstTask, secondTask, thirdTask);
}
```

RunConcurrentTasksWhenAny 返回的 Task 会在 firstTask、secondTask 或 thirdTask 中任意一个完成时结束。

第114.3节 : Try/Catch/Finally

版本 ≥ 6.0

从 C# 6.0 开始, await 关键字现在可以在 catch 和 finally 块中使用。

```
try {
    var client = new AsyncClient();
    await client.DoSomething();
} catch (MyException ex) {
    await client.LogExceptionAsync();
    throw;
} finally {
    await client.CloseAsync();
}
```

版本 ≥ 5.0 版本 < 6.0

在 C# 6.0 之前, 你需要做类似如下的操作。注意, 6.0 还通过空传播操作符简化了空检查。

```
AsyncClient client;
MyException caughtException;
try {
    client = new AsyncClient();
    await client.DoSomething();
} catch (MyException ex) {
    caughtException = ex;
}

if (client != null) {
    if (caughtException != null) {
        await client.LogExceptionAsync();
    }
    await client.CloseAsync();
    if (caughtException != null) throw caughtException;
}
```

请注意, 如果你等待的任务不是由async创建的 (例如由Task.Run创建的任务), 某些调试器可能会在任务抛出异常时中断, 尽管异常似乎已被外围的try/catch处理。这是因为调试器认为该异常对于用户代码来说是未处理的。在Visual Studio中, 有一个

```
await Task.WhenAll(task1, task2);

var result = await task2;
```

Also, the Task.WhenAny can be used to execute multiple tasks in parallel, like the Task.WhenAll above, with the difference that this method will complete when *any* of the supplied tasks will be completed.

```
public async Task RunConcurrentTasksWhenAny()
{
    var firstTask = TaskOperation("#firstTask executed");
    var secondTask = TaskOperation("#secondTask executed");
    var thirdTask = TaskOperation("#thirdTask executed");
    await Task.WhenAny(firstTask, secondTask, thirdTask);
}
```

The Task returned by RunConcurrentTasksWhenAny will complete when any of firstTask, secondTask, or thirdTask completes.

Section 114.3: Try/Catch/Finally

Version ≥ 6.0

As of C# 6.0, the `await` keyword can now be used within a `catch` and `finally` block.

```
try {
    var client = new AsyncClient();
    await client.DoSomething();
} catch (MyException ex) {
    await client.LogExceptionAsync();
    throw;
} finally {
    await client.CloseAsync();
}
```

Version ≥ 5.0 Version < 6.0

Prior to C# 6.0, you would need to do something along the lines of the following. Note that 6.0 also cleaned up the null checks with the Null Propagating operator.

```
AsyncClient client;
MyException caughtException;
try {
    client = new AsyncClient();
    await client.DoSomething();
} catch (MyException ex) {
    caughtException = ex;
}

if (client != null) {
    if (caughtException != null) {
        await client.LogExceptionAsync();
    }
    await client.CloseAsync();
    if (caughtException != null) throw caughtException;
}
```

Please note that if you await a task not created by `async` (e.g. a task created by `Task.Run`), some debuggers may break on exceptions thrown by the task even when it is seemingly handled by the surrounding try/catch. This happens because the debugger considers it to be unhandled with respect to user code. In Visual Studio, there is an

名为"Just My Code"的选项，可以禁用该选项以防止调试器在此类情况下中断。

第114.4节：返回一个未使用await的Task

执行异步操作的方法如果满足以下条件，则不需要使用await：

- 方法内部只有一个异步调用
- 异步调用位于方法末尾
- 不需要捕获/处理可能在Task中发生的异常

考虑以下返回Task的方法：

```
public async Task<User> GetUserAsync(int id)
{
    var lookupKey = "Users" + id;
    return await dataStore.GetByKeyAsync(lookupKey);
}
```

如果GetByKeyAsync与 GetUserAsync具有相同的签名（返回Task<User>），则该方法可以简化为：

```
public Task<User> GetUserAsync(int id)
{
    var lookupKey = "Users" + id;
    return dataStore.GetByKeyAsync(lookupKey);
}
```

在这种情况下，尽管方法执行的是异步操作，但不需要将其标记为async。由GetByKeyAsync返回的Task会直接传递给调用方法，在那里它将被await。

重要: 返回Task而不是等待它，会改变方法的异常行为，因为异常不会在启动任务的方法中抛出，而是在等待该任务的方法中抛出。

```
public Task SaveAsync()
{
    try {
        return dataStore.SaveChangesAsync();
    }
    catch(Exception ex)
    {
        // 这段代码永远不会被调用
        logger.LogException(ex);
    }
}

// 其他调用 SaveAsync() 的代码

// 如果发生异常，将在这里抛出，而不是在 SaveAsync() 内部抛出
await SaveAsync();
```

这将提升性能，因为它可以避免编译器生成额外的async状态机。

第114.5节：只有当async/await允许状态机执行额外工作时，才会提升性能

考虑以下代码：

option called "["Just My Code"](#)"，which can be disabled to prevent the debugger from breaking in such situations.

Section 114.4: Returning a Task without await

Methods that perform asynchronous operations don't need to use `await` if:

- There is only one asynchronous call inside the method
- The asynchronous call is at the end of the method
- Catching/handling exception that may happen within the Task is not necessary

Consider this method that returns a Task:

```
public async Task<User> GetUserAsync(int id)
{
    var lookupKey = "Users" + id;
    return await dataStore.GetByKeyAsync(lookupKey);
}
```

If GetByKeyAsync has the same signature as GetUserAsync (returning a Task<User>)，the method can be simplified:

```
public Task<User> GetUserAsync(int id)
{
    var lookupKey = "Users" + id;
    return dataStore.GetByKeyAsync(lookupKey);
}
```

In this case, the method doesn't need to be marked `async`, even though it's performing an asynchronous operation. The Task returned by GetByKeyAsync is passed directly to the calling method, where it will be `awaited`.

Important: Returning the Task instead of awaiting it, changes the exception behavior of the method, as it won't throw the exception inside the method which starts the task but in the method which awaits it.

```
public Task SaveAsync()
{
    try {
        return dataStore.SaveChangesAsync();
    }
    catch(Exception ex)
    {
        // this will never be called
        logger.LogException(ex);
    }
}

// Some other code calling SaveAsync()

// If exception happens, it will be thrown here, not inside SaveAsync()
await SaveAsync();
```

This will improve performance as it will save the compiler the generation of an extra `async` state machine.

Section 114.5: Async/await will only improve performance if it allows the machine to do additional work

Consider the following code:

```

public async Task MethodA()
{
    await MethodB();
    // 执行其他工作
}

public async Task MethodB()
{
    await MethodC();
    // 执行其他工作
}

public async Task MethodC()
{
    // 或等待其他异步操作
    await Task.Delay(100);
}

```

这不会比

```

public void MethodA()
{
    MethodB();
    // 执行其他工作
}

public void MethodB()
{
    MethodC();
    // 执行其他工作
}

public void MethodC()
{
    Thread.Sleep(100);
}

```

async/await 的主要目的是允许机器执行额外的工作——例如，允许调用线程在等待某个 I/O 操作结果时执行其他工作。在这种情况下，调用线程永远不会比直接同步调用 MethodA()、MethodB() 和 MethodC() 能做更多的工作，因此在性能上并没有提升。

第114.6节：Web.config设置为目标4.5以实现正确的异步行为

web.config中的system.web.httpRuntime必须设置为目标4.5，以确保线程在恢复异步方法之前会重新进入请求上下文。

```
<httpRuntime targetFramework="4.5" />
```

在ASP.NET 4.5之前，Async和await的行为未定义。Async/await将在可能没有请求上下文的任意线程上恢复。负载下的应用程序会随机因await后访问HttpContext时出现空引用异常而失败。由于异步的原因，在WebApi中使用HttpContext.Current是危险的。

第114.7节：简单的连续调用

```
public async Task<JobResult> GetDataFromWebAsync()
```

```

public async Task MethodA()
{
    await MethodB();
    // Do other work
}

public async Task MethodB()
{
    await MethodC();
    // Do other work
}

public async Task MethodC()
{
    // Or await some other async work
    await Task.Delay(100);
}

```

This will not perform any better than

```

public void MethodA()
{
    MethodB();
    // Do other work
}

public void MethodB()
{
    MethodC();
    // Do other work
}

public void MethodC()
{
    Thread.Sleep(100);
}

```

The primary purpose of async/await is to allow the machine to do additional work - for example, to allow the calling thread to do other work while it's waiting for a result from some I/O operation. In this case, the calling thread is never allowed to do more work than it would have been able to do otherwise, so there's no performance gain over simply calling MethodA(), MethodB(), and MethodC() synchronously.

Section 114.6: Web.config setup to target 4.5 for correct async behaviour

The web.config system.web.httpRuntime must target 4.5 to ensure the thread will reenter the request context before resuming your async method.

```
<httpRuntime targetFramework="4.5" />
```

Async and await have undefined behavior on ASP.NET prior to 4.5. Async / await will resume on an arbitrary thread that may not have the request context. Applications under load will randomly fail with null reference exceptions accessing the HttpContext after the await. [Using HttpContext.Current in WebApi is dangerous because of async](#)

Section 114.7: Simple consecutive calls

```
public async Task<JobResult> GetDataFromWebAsync()
```

```

{
    var nextJob = await _database.GetNextJobAsync();
    var response = await _httpClient.GetAsync(nextJob.Uri);
    var pageContents = await response.Content.ReadAsStringAsync();
    return await _database.SaveJobResultAsync(pageContents);
}

```

这里主要需要注意的是，虽然每个await调用的方法都是异步调用的——在该调用期间控制权会交还给系统——但方法内部的流程是线性的，不需要因异步而进行特殊处理。如果调用的任何方法失败，异常将按“预期”处理，也就是说方法执行将被中止，异常会向上传递。

第114.8节：在异步代码上阻塞可能导致死锁

在异步调用中阻塞是一种不好的做法，因为它可能会在具有同步上下文的环境中导致死锁。最佳实践是“全程”使用 async/await。例如，以下 Windows 窗体代码会导致死锁：

```

private async Task<bool> TryThis()
{
    Trace.TraceInformation("开始 TryThis");
    await Task.Run(() =>
    {
        Trace.TraceInformation("在 TryThis 任务中");
        for (int i = 0; i < 100; i++)
        {
            // 这段代码运行成功——循环执行完毕
        }
        Trace.TraceInformation("For 循环 " + i);
        System.Threading.Thread.Sleep(10);
    });
}

// 由于死锁，这段代码永远不会执行
Trace.TraceInformation("即将返回");
return true;
}

// 按钮点击事件处理程序
private void button1_Click(object sender, EventArgs e)
{
    // .Result 导致此处在异步调用上阻塞
    bool result = TryThis().Result;
    // 实际上永远不会执行到这里
    Trace.TraceInformation("完成结果");
}

```

本质上，一旦异步调用完成，它会等待同步上下文变得可用。然而，事件处理程序在等待TryThis()方法完成时“占用”了同步上下文，从而导致循环等待。

为了解决这个问题，代码应修改为

```

private async void button1_Click(object sender, EventArgs e)
{
    bool result = await TryThis();
    Trace.TraceInformation("完成结果");
}

```

```

{
    var nextJob = await _database.GetNextJobAsync();
    var response = await _httpClient.GetAsync(nextJob.Uri);
    var pageContents = await response.Content.ReadAsStringAsync();
    return await _database.SaveJobResultAsync(pageContents);
}

```

The main thing to note here is that while every `await`-ed method is called asynchronously - and for the time of that call the control is yielded back to the system - the flow inside the method is linear and does not require any special treatment due to asynchrony. If any of the methods called fail, the exception will be processed "as expected", which in this case means that the method execution will be aborted and the exception will be going up the stack.

Section 114.8: Blocking on async code can cause deadlocks

It is a bad practice to block on async calls as it can cause deadlocks in environments that have a synchronization context. The best practice is to use `async/await` "all the way down." For example, the following Windows Forms code causes a deadlock:

```

private async Task<bool> TryThis()
{
    Trace.TraceInformation("Starting TryThis");
    await Task.Run(() =>
    {
        Trace.TraceInformation("In TryThis task");
        for (int i = 0; i < 100; i++)
        {
            // This runs successfully - the loop runs to completion
            Trace.TraceInformation("For loop " + i);
            System.Threading.Thread.Sleep(10);
        }
    });

    // This never happens due to the deadlock
    Trace.TraceInformation("About to return");
    return true;
}

// Button click event handler
private void button1_Click(object sender, EventArgs e)
{
    // .Result causes this to block on the asynchronous call
    bool result = TryThis().Result;
    // Never actually gets here
    Trace.TraceInformation("Done with result");
}

```

Essentially, once the async call completes, it waits for the synchronization context to become available. However, the event handler "holds on" to the synchronization context while it's waiting for the TryThis() method to complete, thus causing a circular wait.

To fix this, code should be modified to

```

private async void button1_Click(object sender, EventArgs e)
{
    bool result = await TryThis();
    Trace.TraceInformation("Done with result");
}

```

注意：事件处理器是唯一可以使用`async void`的地方（因为你无法等待一个`async void`方法）。

Note: event handlers are the only place where `async void` should be used (because you can't await an `async void` method).

belindoc.com

第115章：异步等待中的同步上下文

第115.1节：async/await关键字的伪代码

考虑一个简单的异步方法：

```
async Task Foo()
{
    Bar();
    await Baz();
    Qux();
}
```

简化来说，我们可以说这段代码实际上意味着以下内容：

```
Task Foo()
{
    Bar();
    Task t = Baz();
    var context = SynchronizationContext.Current;
    t.ContinueWith(task) =>
    {
        如果(context == null)
            Qux();
        否则
            context.Post((obj) => Qux(), null);
    }, TaskScheduler.Current);

    return t;
}
```

这意味着`async/await`关键字会使用当前的同步上下文（如果存在）。也就是说，你可以编写库代码使其在UI、Web和控制台应用程序中都能正确工作。

[来源文章](#)

第115.2节：禁用同步上下文

要禁用同步上下文，应调用`ConfigureAwait`方法：

```
async Task() Foo()
{
    await Task.Run(() => Console.WriteLine("Test"));
}

...
Foo().ConfigureAwait(false);
```

`ConfigureAwait`提供了一种避免默认`SynchronizationContext`捕获行为的方法；将`flowContext`参数传递为`false`可以防止在`await`之后使用`SynchronizationContext`来恢复执行。

Chapter 115: Synchronization Context in Async-Await

Section 115.1: Pseudocode for async/await keywords

Consider a simple asynchronous method:

```
async Task Foo()
{
    Bar();
    await Baz();
    Qux();
}
```

Simplifying, we can say that this code actually means the following:

```
Task Foo()
{
    Bar();
    Task t = Baz();
    var context = SynchronizationContext.Current;
    t.ContinueWith(task) =>
    {
        if (context == null)
            Qux();
        else
            context.Post((obj) => Qux(), null);
    }, TaskScheduler.Current);

    return t;
}
```

It means that `async/await` keywords use current synchronization context if it exists. I.e. you can write library code that would work correctly in UI, Web, and Console applications.

[Source article](#).

Section 115.2: Disabling synchronization context

To disable synchronization context you should call the `ConfigureAwait` method:

```
async Task() Foo()
{
    await Task.Run(() => Console.WriteLine("Test"));
}

...
Foo().ConfigureAwait(false);
```

`ConfigureAwait` provides a means to avoid the default `SynchronizationContext` capturing behavior; passing `false` for the `flowContext` parameter prevents the `SynchronizationContext` from being used to resume execution after the `await`.

第115.3节：为什么 SynchronizationContext 如此重要？

考虑以下示例：

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = RunTooLong();
}
```

此方法会冻结 UI 应用程序，直到RunTooLong完成。应用程序将无响应。

你可以尝试异步运行内部代码：

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() => label1.Text = RunTooLong());
}
```

但这段代码不会执行，因为内部主体可能在非 UI 线程上运行，且不应直接更改 UI 属性：

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() =>
    {
        var label1Text = RunTooLong();

        if (label1.InvokeRequired)
            label1.BeginInvoke((Action) delegate() { label1.Text = label1Text; });
        else
            label1.Text = label1Text;
    });
}
```

现在别忘了总是使用这个模式。或者，试试SynchronizationContext.Post，它会帮你完成：

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() =>
    {
        var label1Text = RunTooLong();
        SynchronizationContext.Current.Post((obj) =>
        {
            label1.Text = label1.Text;
        }, null);
    });
}
```

Section 115.3: Why SynchronizationContext is so important?

Consider this example:

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = RunTooLong();
}
```

This method will freeze UI application until the RunTooLong will be completed. The application will be unresponsive.

You can try run inner code asynchronously:

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() => label1.Text = RunTooLong());
}
```

But this code won't execute because inner body may be run on non-UI thread and [it shouldn't change UI properties directly](#):

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() =>
    {
        var label1Text = RunTooLong();

        if (label1.InvokeRequired)
            label1.BeginInvoke((Action) delegate() { label1.Text = label1Text; });
        else
            label1.Text = label1Text;
    });
}
```

Now don't forget always to use this pattern. Or, try [SynchronizationContext.Post](#) that will make it for you:

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Run(() =>
    {
        var label1Text = RunTooLong();
        SynchronizationContext.Current.Post((obj) =>
        {
            label1.Text = label1.Text;
        }, null);
    });
}
```

第116章：BackgroundWorker

第116.1节：使用BackgroundWorker完成任务

以下示例演示了如何使用 BackgroundWorker 来更新 WinForms 的进度条。backgroundWorker 会更新进度条的值，而不会阻塞 UI 线程，从而在后台工作时显示响应式的 UI。

```
namespace BgWorkerExample
{
    public partial class Form1 : Form
    {

        // 创建了一个新的 BackgroundWorker 实例。
        BackgroundWorker bgWorker = new BackgroundWorker();

        public Form1()
        {
            InitializeComponent();

            prgProgressBar.Step = 1;

            // 为 backgroundWorker 分配事件处理程序
            bgWorker.DoWork += bgWorker_DoWork;
            bgWorker.RunWorkerCompleted += bgWorker_WorkComplete;

            // 告诉 backgroundWorker 触发 "DoWork" 事件，从而启动它。
            // 检查 backgroundWorker 是否已经在运行。
            if (!bgWorker.IsBusy)
                bgWorker.RunWorkerAsync();
        }

        private void bgWorker_DoWork(object sender, DoWorkEventArgs e)
        {
            // 这是后台线程中 backgroundworker 将执行的方法。
            /* 有一点需要注意！不需要 try catch，因为任何异常都会终止
            backgroundWorker 并报告
            错误到 "RunWorkerCompleted" 事件 */
            CountToY();
        }

        private void bgWorker_WorkComplete(object sender, RunWorkerCompletedEventArgs e)
        {
            // e.Error 将包含 backgroundWorker 捕获的任何异常
            if (e.Error != null)
            {
                MessageBox.Show(e.Error.Message);
            }
            else
            {
                MessageBox.Show("任务完成！");
                prgProgressBar.Value = 0;
            }
        }

        // 执行“长时间”任务的示例方法。
        private void CountToY()
        {
            int x = 0;
```

Chapter 116: BackgroundWorker

Section 116.1: Using a BackgroundWorker to complete a task

The following example demonstrates the use of a BackgroundWorker to update a WinForms ProgressBar. The backgroundWorker will update the value of the progress bar without blocking the UI thread, thus showing a reactive UI while work is done in the background.

```
namespace BgWorkerExample
{
    public partial class Form1 : Form
    {

        // a new instance of a backgroundWorker is created.
        BackgroundWorker bgWorker = new BackgroundWorker();

        public Form1()
        {
            InitializeComponent();

            prgProgressBar.Step = 1;

            // this assigns event handlers for the backgroundWorker
            bgWorker.DoWork += bgWorker_DoWork;
            bgWorker.RunWorkerCompleted += bgWorker_WorkComplete;

            // tell the backgroundWorker to raise the "DoWork" event, thus starting it.
            // Check to make sure the background worker is not already running.
            if (!bgWorker.IsBusy)
                bgWorker.RunWorkerAsync();
        }

        private void bgWorker_DoWork(object sender, DoWorkEventArgs e)
        {
            // this is the method that the backgroundworker will perform on in the background thread.
            /* One thing to note! A try catch is not necessary as any exceptions will terminate the
            backgroundWorker and report
            the error to the "RunWorkerCompleted" event */
            CountToY();
        }

        private void bgWorker_WorkComplete(object sender, RunWorkerCompletedEventArgs e)
        {
            // e.Error will contain any exceptions caught by the backgroundWorker
            if (e.Error != null)
            {
                MessageBox.Show(e.Error.Message);
            }
            else
            {
                MessageBox.Show("Task Complete!");
                prgProgressBar.Value = 0;
            }
        }

        // example method to perform a "long" running task.
        private void CountToY()
        {
            int x = 0;
```

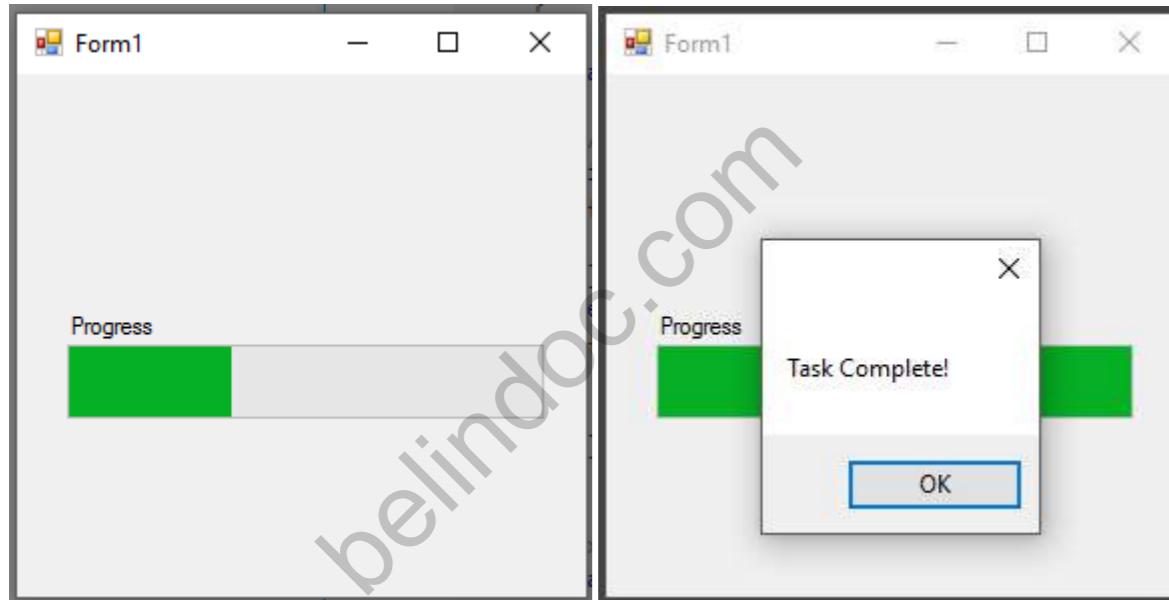
```

int maxProgress = 100;
prgProgressBar.Maximum = maxProgress;

while (x < maxProgress)
{
    System.Threading.Thread.Sleep(50);
Invoke(new Action(() => { prgProgressBar.PerformStep(); }));
    x += 1;
}
}

```

结果如下...



第116.2节：为BackgroundWorker分配事件 处理程序

一旦声明了 BackgroundWorker 的实例，就必须为其执行的任务设置属性和事件处理程序。

```

/* 这是 BackgroundWorker 的 "DoWork" 事件处理程序。
   该方法包含
   了你希望程序执行的所有工作，且不会阻塞用户界面。 */

```

```
bgWorker.DoWork += bgWorker_DoWork;
```

```

/* 这是 DoWork 事件方法的签名示例：*/
private void bgWorker_DoWork(object sender, DoWorkEventArgs e)
{
    // 在这里执行工作
    // ...
    // 获取当前 BackgroundWorker 的引用：
    BackgroundWorker worker = sender as BackgroundWorker;
        // BackgroundWorker 的引用通常用于报告进度
    worker.ReportProgress(...);
}

```

```

int maxProgress = 100;
prgProgressBar.Maximum = maxProgress;

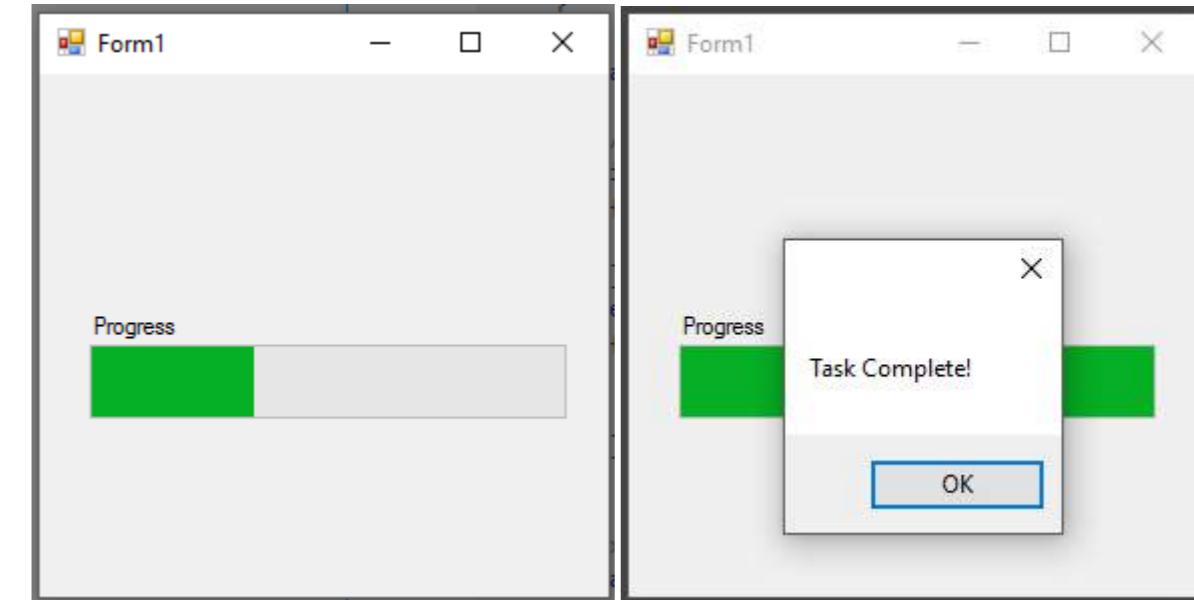
```

```

while (x < maxProgress)
{
    System.Threading.Thread.Sleep(50);
Invoke(new Action(() => { prgProgressBar.PerformStep(); }));
    x += 1;
}
}

```

The result is the following...



Section 116.2: Assigning Event Handlers to a BackgroundWorker

Once the instance of the BackgroundWorker has been declared, it must be given properties and event handlers for the tasks it performs.

```

/* This is the backgroundworker's "DoWork" event handler. This
   method is what will contain all the work you
   wish to have your program perform without blocking the UI. */

```

```
bgWorker.DoWork += bgWorker_DoWork;
```

```

/*This is how the DoWork event method signature looks like:*/
private void bgWorker_DoWork(object sender, DoWorkEventArgs e)
{
    // Work to be done here
    // ...
    // To get a reference to the current Backgroundworker:
    BackgroundWorker worker = sender as BackgroundWorker;
        // The reference to the BackgroundWorker is often used to report progress
    worker.ReportProgress(...);
}

```

```

/* 这是 BackgroundWorker 完成任务后将运行的方法 */
bgWorker.RunWorkerCompleted += bgWorker_CompletedWork;

/*这是 RunWorkerCompletedEvent 事件方法签名的样子：*/
private void bgWorker_CompletedWork(object sender, RunWorkerCompletedEventArgs e)
{
    // 后台工作完成后要执行的操作
}

/* 当你希望在进度发生变化时（例如某个特定任务完成时）触发某些操作，使用“ProgressChanged”事件处理程序。注意，只有当 bgWorker.WorkerReportsProgress 设置为 true 时，调用 bgWorker.ReportProgress(...) 才会触发 ProgressChanged 事件。*/
// 当报告进度变化时要执行的操作
// ProgressChangedEventArgs 提供了一个百分比，方便报告进度的完成程度
int progress = e.ProgressPercentage;
}

```

第116.3节：创建一个新的BackgroundWorker实例

BackgroundWorker 通常用于执行任务，有时这些任务耗时较长，但不会阻塞 UI 线程。

```

// BackgroundWorker 是 ComponentModel 命名空间的一部分。
using System.ComponentModel;

namespace BGWorkerExample
{
    public partial class ExampleForm : Form
    {
        // 以下代码创建了一个名为“bgWorker”的 BackgroundWorker 实例
        BackgroundWorker bgWorker = new BackgroundWorker();

        public ExampleForm() { ... }
    }
}

```

第116.4节：为 BackgroundWorker 分配属性

这允许 BackgroundWorker 在任务之间被取消

```
bgWorker.WorkerSupportsCancellation = true;
```

这允许工作线程在完成任务之间报告进度...

```
bgWorker.WorkerReportsProgress = true;
```

```
//这也必须与 ProgressChanged 事件一起使用
```

```

/*This is the method that will be run once the BackgroundWorker has completed its tasks */

bgWorker.RunWorkerCompleted += bgWorker_CompletedWork;

/*This is how the RunWorkerCompletedEvent event method signature looks like:*/
private void bgWorker_CompletedWork(object sender, RunWorkerCompletedEventArgs e)
{
    // Things to be done after the backgroundworker has finished
}

/* When you wish to have something occur when a change in progress occurs, (like the completion of a specific task) the "ProgressChanged" event handler is used. Note that ProgressChanged events may be invoked by calls to bgWorker.ReportProgress(...) only if bgWorker.WorkerReportsProgress is set to true. */
bgWorker.ProgressChanged += bgWorker_ProgressChanged;

/*This is how the ProgressChanged event method signature looks like:*/
private void bgWorker_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
    // Things to be done when a progress change has been reported

    /* The ProgressChangedEventArgs gives access to a percentage, allowing for easy reporting of how far along a process is*/
    int progress = e.ProgressPercentage;
}

```

Section 116.3: Creating a new BackgroundWorker instance

A BackgroundWorker is commonly used to perform tasks, sometimes time consuming, without blocking the UI thread.

```

// BackgroundWorker is part of the ComponentModel namespace.
using System.ComponentModel;

namespace BGWorkerExample
{
    public partial class ExampleForm : Form
    {
        // the following creates an instance of the BackgroundWorker named "bgWorker"
        BackgroundWorker bgWorker = new BackgroundWorker();

        public ExampleForm() { ... }
    }
}

```

Section 116.4: Assigning Properties to a BackgroundWorker

This allows the BackgroundWorker to be cancelled in between tasks

```
bgWorker.WorkerSupportsCancellation = true;
```

This allows the worker to report progress between completion of tasks...

```
bgWorker.WorkerReportsProgress = true;
```

```
//this must also be used in conjunction with the ProgressChanged event
```

第117章：任务并行库

第117.1节：Parallel.ForEach

一个使用 Parallel.ForEach 循环来 ping 给定网站 URL 数组的示例。

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.ForEach(urls, url =>
    {
        var ping = new System.Net.NetworkInformation.Ping();
        var result = ping.Send(url);

        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
        {
            Console.WriteLine(string.Format("{0} is online", url));
        }
    });
}
```

第117.2节：Parallel.For

使用Parallel.For循环对给定的网站URL数组进行ping操作的示例。

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.For(0, urls.Length, i =>
    {
        var ping = new System.Net.NetworkInformation.Ping();
        var result = ping.Send(urls[i]);

        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
        {
            Console.WriteLine(string.Format("{0} is online", urls[\t[i]]));
        }
    });
}
```

Chapter 117: Task Parallel Library

Section 117.1: Parallel.ForEach

An example that uses Parallel.ForEach loop to ping a given array of website urls.

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.ForEach(urls, url =>
    {
        var ping = new System.Net.NetworkInformation.Ping();
        var result = ping.Send(url);

        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
        {
            Console.WriteLine(string.Format("{0} is online", url));
        }
    });
}
```

Section 117.2: Parallel.For

An example that uses Parallel.For loop to ping a given array of website urls.

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.For(0, urls.Length, i =>
    {
        var ping = new System.Net.NetworkInformation.Ping();
        var result = ping.Send(urls[i]);

        if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
        {
            Console.WriteLine(string.Format("{0} is online", urls[i]));
        }
    });
}
```

第117.3节 : Parallel.Invoke

并行调用方法或操作（并行区域）

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.Invoke(
        () => PingUrl(urls[0]),
        () => PingUrl(urls[1]),
        () => PingUrl(urls[2]),
        () => PingUrl(urls[3])
    );
}

void PingUrl(string url)
{
    var ping = new System.Net.NetworkInformation.Ping();

    var result = ping.Send(url);

    if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
    {
        Console.WriteLine(string.Format("{0} 在线", url));
    }
}
```

Section 117.3: Parallel.Invoke

Invoking methods or actions in parallel (Parallel region)

```
static void Main()
{
    string [] urls =
    {
        "www.stackoverflow.com",
        "www.google.net",
        "www.facebook.com",
        "www.twitter.com"
    };

    System.Threading.Tasks.Parallel.Invoke(
        () => PingUrl(urls[0]),
        () => PingUrl(urls[1]),
        () => PingUrl(urls[2]),
        () => PingUrl(urls[3])
    );
}

void PingUrl(string url)
{
    var ping = new System.Net.NetworkInformation.Ping();

    var result = ping.Send(url);

    if (result.Status == System.Net.NetworkInformation.IPStatus.Success)
    {
        Console.WriteLine(string.Format("{0} is online", url));
    }
}
```

第118章：使变量线程安全

第118.1节：在Parallel.For

循环中控制对变量的访问

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static void Main( string[] args )
    {
        object sync = new object();
        int sum = 0;
        Parallel.For( 1, 1000, ( i ) => {
            lock( sync ) sum = sum + i; // 需要锁定

            // 实际操作中，通过模拟耗时操作确保此“并行循环”在多个线程上执行。
            Thread.Sleep( 1 );
        } );
        Console.WriteLine( "正确答案应为 499500。 sum 是: {0}", sum );
    }
}
```

仅仅执行 `sum = sum + i` 而不加锁是不够的，因为读-改-写操作不是原子的。
线程会覆盖在读取当前 `sum` 值后、但在将修改后的 `sum + i` 值存回 `sum` 之前发生的任何外部对 `sum` 的修改。

Chapter 118: Making a variable thread safe

Section 118.1: Controlling access to a variable in a Parallel.For loop

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static void Main( string[] args )
    {
        object sync = new object();
        int sum = 0;
        Parallel.For( 1, 1000, ( i ) => {
            lock( sync ) sum = sum + i; // lock is necessary

            // As a practical matter, ensure this `parallel for` executes
            // on multiple threads by simulating a lengthy operation.
            Thread.Sleep( 1 );
        } );
        Console.WriteLine( "Correct answer should be 499500. sum is: {0}", sum );
    }
}
```

It is not sufficient to just do `sum = sum + i` without the lock because the read-modify-write operation is not atomic.
A thread will overwrite any external modifications to `sum` that occur after it has read the current value of `sum`, but before it stores the modified value of `sum + i` back into `sum`.

第119章：锁语句

第119.1节：在锁语句中抛出异常

以下代码将释放锁。不会有问题是。在幕后，lock语句的工作方式相当于try finally

```
lock(locker)
{
    throw new Exception();
}
```

更多内容可见于C# 5.0 规范：

形式为lock语句

```
lock (x) ...
```

其中 x 是一个引用类型的表达式，完全等同于

```
bool __lockWasTaken = false;
try {
    System.Threading.Monitor.Enter(x, ref __lockWasTaken);
    ...
}
finally {
    if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

除了 x 只被计算一次。

第119.2节：简单用法

“lock”的常见用法是关键区段。

在下面的示例中，假设“ReserveRoom”会被不同线程调用。使用“lock”进行同步是防止竞争条件的最简单方法。方法体被“lock”包围，确保两个或多个线程不能同时执行它。

```
public class Hotel
{
    private readonly object _roomLock = new object();

    public void ReserveRoom(int roomNumber)
    {
        // lock关键字确保一次只有一个线程执行关键区段
        // 在此情况下，预订指定号码的酒店房间
        // 防止重复预订
        lock (_roomLock)
        {
            // 预订房间的逻辑写在这里
        }
    }
}
```

如果一个线程在另一个线程正在执行“lock”块时到达该块，前者将等待后者退出

Chapter 119: Lock Statement

Section 119.1: Throwing exception in a lock statement

Following code will release the lock. There will be no problem. Behind the scenes lock statement works as try finally

```
lock(locker)
{
    throw new Exception();
}
```

More can be seen in the [C# 5.0 Specification](#):

A lock statement of the form

```
lock (x) ...
```

where x is an expression of a *reference-type*, is precisely equivalent to

```
bool __lockWasTaken = false;
try {
    System.Threading.Monitor.Enter(x, ref __lockWasTaken);
    ...
}
finally {
    if (__lockWasTaken) System.Threading.Monitor.Exit(x);
}
```

except that x is only evaluated once.

Section 119.2: Simple usage

Common usage of lock is a critical section.

In the following example ReserveRoom is supposed to be called from different threads. Synchronization with lock is the simplest way to prevent race condition here. Method body is surrounded with lock which ensures that two or more threads cannot execute it simultaneously.

```
public class Hotel
{
    private readonly object _roomLock = new object();

    public void ReserveRoom(int roomNumber)
    {
        // lock keyword ensures that only one thread executes critical section at once
        // in this case, reserves a hotel room of given number
        // preventing double bookings
        lock (_roomLock)
        {
            // reserve room logic goes here
        }
    }
}
```

If a thread reaches lock-ed block while another thread is running within it, the former will wait another to exit the

块。

最佳做法是定义一个私有对象作为锁，或者定义一个私有静态对象变量来保护所有实例共有的数据。

第119.3节：在lock语句中的return

以下代码将释放锁。

```
lock(locker)
{
    return 5;
}
```

详细说明请参考这个SO答案。[_____](#)

第119.4节：反模式和陷阱

对栈分配的/局部变量加锁

使用lock时的一个误区是在函数中使用局部对象作为锁。由于这些局部对象实例在每次调用函数时都会不同，lock将无法按预期工作。

```
List<string> stringList = new List<string>();

public void AddToListNotThreadSafe(string something)
{
    // 不要这样做，因为每次调用此方法
    // 都会锁定不同的 Object 实例。
    // 这不会提供线程安全，只会降低性能。
    var localLock = new Object();
    lock(localLock)
    {
        stringList.Add(something);
    }
}

// 定义可用于 AddToList 方法中线程安全的对象
readonly object classLock = new object();

public void AddToList(List<string> stringList, string something)
{
    // 使用 classLock 实例字段来实现
    // 在添加到 stringList 之前的线程安全锁定
    lock(classLock)
    {
        stringList.Add(something);
    }
}
```

假设锁定限制对同步对象本身的访问

如果一个线程调用：lock(obj)，另一个线程调用obj.ToString()，第二个线程不会被阻塞。

```
object obj = new Object();

public void SomeMethod()
```

block.

Best practice is to define a private object to lock on, or a private static object variable to protect data common to all instances.

Section 119.3: Return in a lock statement

Following code will release lock.

```
lock(locker)
{
    return 5;
}
```

For a detailed explanation, [this SO answer](#) is recommended.

Section 119.4: Anti-Patterns and gotchas

Locking on an stack-allocated / local variable

One of the fallacies while using `lock` is the usage of local objects as locker in a function. Since these local object instances will differ on each call of the function, `lock` will not perform as expected.

```
List<string> stringList = new List<string>();

public void AddToListNotThreadSafe(string something)
{
    // DO NOT do this, as each call to this method
    // will lock on a different instance of an Object.
    // This provides no thread safety, it only degrades performance.
    var localLock = new Object();
    lock(localLock)
    {
        stringList.Add(something);
    }
}

// Define object that can be used for thread safety in the AddToList method
readonly object classLock = new object();

public void AddToList(List<string> stringList, string something)
{
    // USE THE classLock instance field to achieve a
    // thread-safe lock before adding to stringList
    lock(classLock)
    {
        stringList.Add(something);
    }
}
```

Assuming that locking restricts access to the synchronizing object itself

If one thread calls: `lock(obj)` and another thread calls `obj.ToString()` second thread is not going to be blocked.

```
object obj = new Object();

public void SomeMethod()
```

```

    lock(obj)
    {
        //执行危险操作
    }
}

//与此同时在另一个线程上
public void SomeOtherMethod()
{
    var objInString = obj.ToString(); //这不会阻塞
}

```

期望子类知道何时加锁

有时基类的设计要求其子类在访问某些受保护字段时必须使用锁：

```

public abstract class Base
{
    受保护的只读对象 padlock;
    受保护的只读 List<string> list;

    公共 Base()
    {
        this.padlock = new object();
        this.list = new List<string>();
    }

    public abstract void Do();
}

public class Derived1 : Base
{
    public override void Do()
    {
        lock (this.padlock)
        {
            this.list.Add("Derived1");
        }
    }
}

public class Derived2 : Base
{
    public override void Do()
    {
        this.list.Add("Derived2"); // 哎呀！我忘记加锁了！
    }
}

```

使用模板方法封装锁定会更安全：

```

public abstract class Base
{
    private readonly object padlock; // 现在这是私有的
    protected readonly List<string> list;

    public Base()
    {
        this.padlock = new object();
    }
}

```

```

    lock(obj)
    {
        //do dangerous stuff
    }
}

//Meanwhile on other tread
public void SomeOtherMethod()
{
    var objInString = obj.ToString(); //this does not block
}

```

Expecting subclasses to know when to lock

Sometimes base classes are designed such that their subclasses are required to use a lock when accessing certain protected fields:

```

public abstract class Base
{
    protected readonly object padlock;
    protected readonly List<string> list;

    public Base()
    {
        this.padlock = new object();
        this.list = new List<string>();
    }

    public abstract void Do();
}

public class Derived1 : Base
{
    public override void Do()
    {
        lock (this.padlock)
        {
            this.list.Add("Derived1");
        }
    }
}

public class Derived2 : Base
{
    public override void Do()
    {
        this.list.Add("Derived2"); // OOPS! I forgot to lock!
    }
}

```

It is much safer to *encapsulate locking* by using a [Template Method](#):

```

public abstract class Base
{
    private readonly object padlock; // This is now private
    protected readonly List<string> list;

    public Base()
    {
        this.padlock = new object();
    }
}

```

```

        this.list = new List<string>();
    }

    public void Do()
    {
        lock (this.padlock) {
            this.DoInternal();
        }
    }

    protected abstract void DoInternal();
}

public class Derived1 : Base
{
    protected override void DoInternal()
    {
        this.list.Add("Derived1"); // 太好了！不需要加锁
    }
}

```

对装箱的值类型变量加锁不会实现同步

在以下示例中，一个私有变量被隐式装箱，因为它作为一个object参数传递给一个函数，该函数期望一个监视器资源来进行锁定。装箱发生在调用IncInSync函数之前，因此每次调用该函数时，装箱的实例对应不同的堆对象。

```

public int Count { get; private set; }

private readonly int counterLock = 1;

public void Inc()
{
    IncInSync(counterLock);
}

private void IncInSync(object monitorResource)
{
    lock (monitorResource)
    {
        Count++;
    }
}

```

装箱发生在Inc函数中：

```

BulemicCounter.Inc:
IL_0000:  nop
IL_0001:  ldarg.0
IL_0002:  ldarg.0
IL_0003:  ldfld   UserQuery+BulemicCounter.counterLock
IL_0008:  box     System.Int32**
IL_000D:  call    UserQuery+BulemicCounter.IncInSync
IL_0012:  nop
IL_0013:  ret

```

这并不意味着装箱的值类型完全不能用于监视器锁定：

```
private readonly object counterLock = 1;
```

```

        this.list = new List<string>();
    }

    public void Do()
    {
        lock (this.padlock) {
            this.DoInternal();
        }
    }

    protected abstract void DoInternal();
}

public class Derived1 : Base
{
    protected override void DoInternal()
    {
        this.list.Add("Derived1"); // Yay! No need to lock
    }
}

```

Locking on a boxed ValueType variable does not synchronize

In the following example, a private variable is implicitly boxed as it's supplied as an `object` argument to a function, expecting a monitor resource to lock at. The boxing occurs just prior to calling the `IncInSync` function, so the boxed instance corresponds to a different heap object each time the function is called.

```

public int Count { get; private set; }

private readonly int counterLock = 1;

public void Inc()
{
    IncInSync(counterLock);
}

private void IncInSync(object monitorResource)
{
    lock (monitorResource)
    {
        Count++;
    }
}

```

Boxing occurs in the `Inc` function:

```

BulemicCounter.Inc:
IL_0000:  nop
IL_0001:  ldarg.0
IL_0002:  ldarg.0
IL_0003:  ldfld   UserQuery+BulemicCounter.counterLock
IL_0008:  box     System.Int32**
IL_000D:  call    UserQuery+BulemicCounter.IncInSync
IL_0012:  nop
IL_0013:  ret

```

It does not mean that a boxed ValueType can't be used for monitor locking at all:

```
private readonly object counterLock = 1;
```

现在装箱发生在构造函数中，这对于锁定来说是可以的：

```
IL_0001: ldc.i4.1
IL_0002: box      System.Int32
IL_0007: stfld    UserQuery+BulementicCounter.counterLock
```

在存在更安全的替代方案时不必要地使用锁

一个非常常见的模式是在一个线程安全的类中使用私有的List或Dictionary，并且每次访问时都加锁：

```
public class 缓存
{
    private readonly object padlock;
    private readonly Dictionary<string, object> values;

    public WordStats()
    {
        this.padlock = new object();
        this.values = new Dictionary<string, object>();
    }

    public void Add(string key, object value)
    {
        lock (this.padlock)
        {
            this.values.Add(key, value);
        }
    }

    /* 类的其余部分省略 */
}
```

如果有多个方法访问values字典，代码会变得非常冗长，更重要的是，一直加锁会掩盖其意图。加锁也很容易被遗忘，缺乏适当的加锁可能导致非常难以发现的错误。

通过使用`ConcurrentDictionary`，我们可以完全避免加锁：

```
public class 缓存
{
    private readonly ConcurrentDictionary<string, object> values;

    public WordStats()
    {
        this.values = new ConcurrentDictionary<string, object>();
    }

    public void Add(string key, object value)
    {
        this.values.Add(key, value);
    }

    /* 类的其余部分省略 */
}
```

使用并发集合也能提升性能，因为它们都在某种程度上采用了无锁技术。

Now boxing occurs in constructor, which is fine for locking:

```
IL_0001: ldc.i4.1
IL_0002: box      System.Int32
IL_0007: stfld    UserQuery+BulementicCounter.counterLock
```

Using locks unnecessarily when a safer alternative exists

A very common pattern is to use a private List or Dictionary in a thread safe class and lock every time it is accessed:

```
public class Cache
{
    private readonly object padlock;
    private readonly Dictionary<string, object> values;

    public WordStats()
    {
        this.padlock = new object();
        this.values = new Dictionary<string, object>();
    }

    public void Add(string key, object value)
    {
        lock (this.padlock)
        {
            this.values.Add(key, value);
        }
    }

    /* rest of class omitted */
}
```

If there are multiple methods accessing the values dictionary, the code can get very long and, more importantly, locking all the time obscures its *intent*. Locking is also very easy to forget and lack of proper locking can cause very hard to find bugs.

By using a `ConcurrentDictionary`, we can avoid locking completely:

```
public class Cache
{
    private readonly ConcurrentDictionary<string, object> values;

    public WordStats()
    {
        this.values = new ConcurrentDictionary<string, object>();
    }

    public void Add(string key, object value)
    {
        this.values.Add(key, value);
    }

    /* rest of class omitted */
}
```

Using concurrent collections also improves performance because [all of them employ lock-free techniques](#) to some extent.

第119.5节：使用Object实例作为锁

使用C#内置的lock语句时需要一个某种类型的实例，但其实例的状态无关紧要。一个object实例非常适合用作锁：

```
public class ThreadSafe {  
    private static readonly object locker = new object();  
  
    public void SomeThreadSafeMethod() {  
        lock (locker) {  
            // 同一时间只有一个线程能进入这里。  
        }  
    }  
}
```

belindoc.com

注意：不应使用Type的实例作为锁（如上面代码中的typeof(ThreadSafe）），因为Type的实例在不同的AppDomain之间是共享的，因此锁的范围可能会意外地包含不该包含的代码（例如，如果ThreadSafe被加载到同一进程的两个AppDomain中，那么对其Type实例加锁会导致互相锁定）。

Section 119.5: Using instances of Object for lock

When using C#'s inbuilt `lock` statement an instance of some type is needed, but its state does not matter. An instance of `object` is perfect for this:

```
public class ThreadSafe {  
    private static readonly object locker = new object();  
  
    public void SomeThreadSafeMethod() {  
        lock (locker) {  
            // Only one thread can be here at a time.  
        }  
    }  
}
```

NB. instances of Type should not be used for this (in the code above `typeof(ThreadSafe)`) because instances of Type are shared across AppDomains and thus the extent of the lock can expectedly include code it shouldn't (eg. if ThreadSafe is loaded into two AppDomains in the same process then locking on its Type instance would mutually lock).

第120章：产量关键词

当你在语句中使用 `yield` 关键字时，表示该方法、操作符或 `get` 访问器是一个迭代器。使用 `yield` 来定义迭代器时，在实现自定义集合类型的 `IEnumerable` 和 `IEnumerator` 模式时，无需显式创建额外的类（用于保存枚举状态的类）。

第120.1节：简单用法

关键字 `yield` 用于定义一个返回 `IEnumerable` 或 `IEnumerator`（以及它们派生的泛型变体）的函数，其值在调用者遍历返回的集合时按需生成。有关用途的更多信息，请参阅备注部分。

下面的示例中，`yield return` 语句位于一个 `for` 循环内。

```
public static IEnumerable<int> Count(int start, int count)
{
    for (int i = 0; i <= count; i++)
    {
        yield return start + i;
    }
}
```

然后你可以这样调用它：

```
foreach (int value in Count(start: 4, count: 10))
{
    Console.WriteLine(value);
}
```

控制台输出

.NET Fiddle 上的演示

`foreach` 语句体的每次迭代都会创建对 `Count` 迭代器函数的调用。对迭代器的每次调用函数会继续执行下一次 `yield return` 语句，该语句发生在 `for` 循环的下一次迭代中。

第120.2节：正确检查参数

迭代器方法在返回值被枚举之前不会执行。因此，在迭代器外断言前置条件是有利的。

```
public static IEnumerable<int> Count(int start, int count)
{
    // 异常将在方法被调用时抛出，而不是在结果被迭代时抛出
    if (count < 0)
        throw new ArgumentOutOfRangeException(nameof(count));
```

Chapter 120: Yield Keyword

When you use the `yield` keyword in a statement, you indicate that the method, operator, or `get` accessor in which it appears is an iterator. Using `yield` to define an iterator removes the need for an explicit extra class (the class that holds the state for an enumeration) when you implement the `IEnumerable` and `IEnumerator` pattern for a custom collection type.

Section 120.1: Simple Usage

The `yield` keyword is used to define a function which returns an `IEnumerable` or `IEnumerator` (as well as their derived generic variants) whose values are generated lazily as a caller iterates over the returned collection. Read more about the purpose in the remarks section.

The following example has a `yield return` statement that's inside a `for` loop.

```
public static IEnumerable<int> Count(int start, int count)
{
    for (int i = 0; i <= count; i++)
    {
        yield return start + i;
    }
}
```

Then you can call it:

```
foreach (int value in Count(start: 4, count: 10))
{
    Console.WriteLine(value);
}
```

Console Output

```
4  
5  
6  
...  
14
```

[Live Demo on .NET Fiddle](#)

Each iteration of the `foreach` statement body creates a call to the `Count` iterator function. Each call to the iterator function proceeds to the next execution of the `yield return` statement, which occurs during the next iteration of the `for` loop.

Section 120.2: Correctly checking arguments

An iterator method is not executed until the return value is enumerated. It's therefore advantageous to assert preconditions outside of the iterator.

```
public static IEnumerable<int> Count(int start, int count)
{
    // The exception will throw when the method is called, not when the result is iterated
    if (count < 0)
        throw new ArgumentOutOfRangeException(nameof(count));
```

```

        return CountCore(start, count);
    }

private static IEnumerable<int> CountCore(int start, int count)
{
    // 如果异常在这里抛出，它将在 IEnumerator 的第一次 MoveNext() 调用期间引发，// 可能发生在代码中
    // 远离传入错误值的位置。

    for (int i = 0; i < count; i++)
    {
        yield return start + i;
    }
}

```

调用方代码（用法）：

```

// 获取计数
var count = Count(1,10);
// 遍历结果
foreach(var x in count)
{
    Console.WriteLine(x);
}

```

输出：

1
2
3
4
5
6
7
8
9
10
Live
Demo
on
.NET
Fiddle

当一个方法使用yield来生成一个可枚举对象时，编译器会创建一个状态机，该状态机在迭代时会运行代码直到遇到yield。然后返回yield的项，并保存其状态。

这意味着你不会在首次调用方法时发现无效参数（传入null等）（因为那时创建了状态机），而是在尝试访问第一个元素时才会发现（因为只有那时状态机才会运行方法内的代码）。通过将其包装在一个先检查参数的普通方法中，你可以在调用方法时检查参数。这是快速失败的一个例子。

在使用C# 7+时，CountCore函数可以方便地作为本地函数隐藏在Count函数中。
示例见此处。

第120.3节：提前终止

你可以通过传入一个或多个值或元素来扩展现有yield方法的功能，这些值或元素可以定义函数内的终止条件，通过调用yield break来停止内部循环的执行。

```

        return CountCore(start, count);
    }

private static IEnumerable<int> CountCore(int start, int count)
{
    // If the exception was thrown here it would be raised during the first MoveNext()
    // call on the IEnumerator, potentially at a point in the code far away from where
    // an incorrect value was passed.
    for (int i = 0; i < count; i++)
    {
        yield return start + i;
    }
}

```

Calling Side Code (Usage):

```

// Get the count
var count = Count(1,10);
// Iterate the results
foreach(var x in count)
{
    Console.WriteLine(x);
}

```

Output:

1
2
3
4
5
6
7
8
9
10

[Live Demo on .NET Fiddle](#)

When a method uses `yield` to generate an enumerable the compiler creates a state machine that when iterated over will run code up to a `yield`. It then returns the yielded item, and saves its state.

This means you won't find out about invalid arguments (passing `null` etc.) when you first call the method (because that creates the state machine), only when you try and access the first element (because only then does the code within the method get ran by the state machine). By wrapping it in a normal method that first checks arguments you can check them when the method is called. This is an example of failing fast.

When using C# 7+, the `CountCore` function can be conveniently hidden into the `Count` function as a *local function*. See example here.

Section 120.3: Early Termination

You can extend the functionality of existing `yield` methods by passing in one or more values or elements that could define a terminating condition within the function by calling a `yield break` to stop the inner loop from executing.

```

public static IEnumerable<int> CountUntilAny(int start, HashSet<int> earlyTerminationSet)
{
    int curr = start;

    while (true)
    {
        if (earlyTerminationSet.Contains(curr))
        {
            // 我们已经到达了其中一个结束值
            yield break;
        }

        yield return curr;

        if (curr == Int32.MaxValue)
        {
            // 如果到达末尾则不溢出；直接停止
            yield break;
        }

        curr++;
    }
}

```

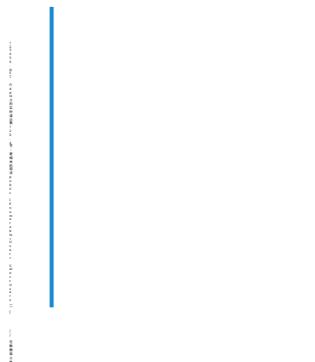
上述方法会从给定的start位置开始迭代，直到遇到earlyTerminationSet中的某个值。

```

// 从起始点迭代，直到遇到定义为
// 终止元素的任意元素
var terminatingElements = new HashSet<int>{ 7, 9, 11 };
// 这将从1开始迭代，直到遇到终止元素之一（7）
foreach(var x in CountUntilAny(1, terminatingElements))
{
    // 这将输出从1到7的结果（触发终止）
    Console.WriteLine(x);
}

```

输出：



```

ExecuteReader(CommandType.Text, "SELECT Id, Name FROM
Users"))
{

```

```

public static IEnumerable<int> CountUntilAny(int start, HashSet<int> earlyTerminationSet)
{
    int curr = start;

    while (true)
    {
        if (earlyTerminationSet.Contains(curr))
        {
            // we've hit one of the ending values
            yield break;
        }

        yield return curr;

        if (curr == Int32.MaxValue)
        {
            // don't overflow if we get all the way to the end; just stop
            yield break;
        }

        curr++;
    }
}

```

The above method would iterate from a given start position until one of the values within the earlyTerminationSet was encountered.

```

// Iterate from a starting point until you encounter any elements defined as
// terminating elements
var terminatingElements = new HashSet<int>{ 7, 9, 11 };
// This will iterate from 1 until one of the terminating elements is encountered (7)
foreach(var x in CountUntilAny(1, terminatingElements))
{
    // This will write out the results from 1 until 7 (which will trigger terminating)
    Console.WriteLine(x);
}

```

Output:

```

1
2
3
4
5
6

```

[Live Demo on .NET Fiddle](#)

Section 120.4: More Pertinent Usage

```

public IEnumerable<User> SelectUsers()
{
    // Execute an SQL query on a database.
    using (IDataReader reader = this.Database.ExecuteReader(CommandType.Text, "SELECT Id, Name FROM
Users"))
    {

```

```

    while (reader.Read())
    {
        int id = reader.GetInt32(0);
        string name = reader.GetString(1);
        yield return new User(id, name);
    }
}

```

当然，还有其他方法可以从SQL数据库获取`IEnumerable<User>`——这只是演示你可以使用`yield`将任何具有“元素序列”语义的内容转换为`IEnumerable<T>`，供他人迭代。

第120.5节：惰性求值

只有当`foreach`语句移动到下一个元素时，迭代器块才会计算到下一个`yield`语句。

考虑以下示例：

```

private IEnumerable<int> Integers()
{
    var i = 0;
    while(true)
    {
Console.WriteLine("迭代器内部: " + i);
        yield return i;
i++;
    }
}

private void PrintNumbers()
{
    var numbers = Integers().Take(3);
Console.WriteLine("开始迭代");

    foreach(var number in numbers)
    {
Console.WriteLine("foreach内部: " + number);
    }
}

```

这将输出：

```

开始迭代
迭代器内部: 0
foreach内部: 0
迭代器内部: 1
foreach内部: 1
迭代器内部: 2
foreach内部: 2

```

[查看演示](#)

因此：

```

while (reader.Read())
{
    int id = reader.GetInt32(0);
    string name = reader.GetString(1);
    yield return new User(id, name);
}
}

```

There are other ways of getting an `IEnumerable<User>` from an SQL database, of course -- this just demonstrates that you can use `yield` to turn anything that has "sequence of elements" semantics into an `IEnumerable<T>` that someone can iterate over.

Section 120.5: Lazy Evaluation

Only when the `foreach` statement moves to the next item does the iterator block evaluate up to the next `yield` statement.

Consider the following example:

```

private IEnumerable<int> Integers()
{
    var i = 0;
    while(true)
    {
        Console.WriteLine("Inside iterator: " + i);
        yield return i;
        i++;
    }
}

private void PrintNumbers()
{
    var numbers = Integers().Take(3);
Console.WriteLine("Starting iteration");

    foreach(var number in numbers)
    {
        Console.WriteLine("Inside foreach: " + number);
    }
}

```

This will output:

```

Starting iteration
Inside iterator: 0
Inside foreach: 0
Inside iterator: 1
Inside foreach: 1
Inside iterator: 2
Inside foreach: 2

```

[View Demo](#)

As a consequence:

- “Starting iteration” 会先被打印，尽管迭代器方法在打印该行之前被调用，因为这一行 Integers().Take(3); 实际上并没有开始迭代（没有调用 IEnumator.MoveNext()）。
- 打印到控制台的行在迭代器方法内部的行和 foreach 内的行之间交替出现，而不是先全部执行迭代器方法内部的行。该程序由于 .Take() 方法而终止，尽管迭代器方法中有一个永远不会跳出的 while true。

第120.6节：Try...finally

如果迭代器方法在 try...finally 中包含 yield，则返回的 IEnumator 在调用 Dispose 时会执行 finally 语句，只要当前的执行点仍在 try 块内。

给定函数：

```
private IEnumerable<int> Numbers()
{
    yield return 1;
    try
    {
        yield return 2;
        yield return 3;
    }
    finally
    {
        Console.WriteLine("Finally executed");
    }
}
```

调用时：

```
private void DisposeOutsideTry()
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.Dispose();
}
```

然后它打印：

1

[查看演示](#)

调用时：

```
private void DisposeInsideTry()
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
```

- “Starting iteration” is printed first even though the iterator method was called before the line printing it because the line Integers().Take(3); does not actually start iteration (no call to IEnumator.MoveNext() was made)
- The lines printing to console alternate between the one inside the iterator method and the one inside the foreach, rather than all the ones inside the iterator method evaluating first
- This program terminates due to the .Take() method, even though the iterator method has a while true which it never breaks out of.

Section 120.6: Try...finally

If an iterator method has a yield inside a try...finally, then the returned IEnumator will execute the finally statement when Dispose is called on it, as long as the current point of evaluation is inside the try block.

Given the function:

```
private IEnumerable<int> Numbers()
{
    yield return 1;
    try
    {
        yield return 2;
        yield return 3;
    }
    finally
    {
        Console.WriteLine("Finally executed");
    }
}
```

When calling:

```
private void DisposeOutsideTry()
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.Dispose();
}
```

Then it prints:

1

[View Demo](#)

When calling:

```
private void DisposeInsideTry()
{
    var enumerator = Numbers().GetEnumerator();

    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
    enumerator.MoveNext();
    Console.WriteLine(enumerator.Current);
```

```
enumerator.Dispose();  
}
```

然后它打印：

```
enumerator.Dispose();  
}
```

Then it prints:

```
1  
2  
Finally executed
```

[View Demo](#)

Section 120.7: Eager evaluation

The `yield` keyword allows lazy-evaluation of the collection. Forcibly loading the whole collection into memory is called **eager evaluation**.

The following code shows this:

```
IEnumerable<int> myMethod()  
{  
    for(int i=0; i <= 8675309; i++)  
    {  
        yield return i;  
    }  
}  
...  
// define the iterator  
var it = myMethod.Take(3);  
// force its immediate evaluation  
// list will contain 0, 1, 2  
var list = it.ToList();
```

Calling `ToList`, `ToDictionary` or `ToArray` will force the immediate evaluation of the enumeration, retrieving all the elements into a collection.

Section 120.8: Using yield to create an `IEnumerable<T>` when implementing `IEnumerable<T>`

The `IEnumerable<T>` interface has a single method, `GetEnumerator()`, which returns an `IEnumerator<T>`.

While the `yield` keyword can be used to directly create an `IEnumerable<T>`, it can *also* be used in exactly the same way to create an `IEnumerator<T>`. The only thing that changes is the return type of the method.

This can be useful if we want to create our own class which implements `IEnumerable<T>`:

```
public class PrintingEnumerable<T> : IEnumerable<T>  
{  
    private IEnumerable<T> _wrapped;  
  
    public PrintingEnumerable(IEnumerable<T> wrapped)  
    {  
        _wrapped = wrapped;  
    }  
  
    // This method returns an Ienumerator<T>, rather than an IEnumerable<T>
```

```
// 但yield语法和用法是相同的。
public IEnumerator<T> GetEnumerator()
{
    foreach(var item in _wrapped)
    {
        Console.WriteLine("Yielding: " + item);
        yield return item;
    }
}

IEnumerator IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}
}
```

(请注意，这个特定示例仅用于说明，实际上可以通过一个返回IEnumerable<T>的迭代器方法更简洁地实现。)

第120.9节：惰性求值示例：斐波那契数列

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Numerics; // 还需添加对System.Numerics的引用

命名空间 ConsoleApplication33
{
    class 程序
    {
        私有静态 IEnumerable<BigInteger> Fibonacci()
        {

BigInteger prev = 0;
            BigInteger current = 1;
            while (true)
            {
                yield return current;
                var next = prev + current;
                prev = current;
                current = next;
            }
        }

        static void Main()
        {
            // 打印第10001到10010个斐波那契数
            var numbers = Fibonacci().Skip(10000).Take(10).ToArray();
            Console.WriteLine(string.Join(Environment.NewLine, numbers));
        }
    }
}
```

其底层工作原理（建议使用IL Disassembler工具反编译生成的.exe文件进行查看）：

1. C#编译器生成了一个实现IEnumerable<BigInteger>和IEnumerator<BigInteger>的类（<Fibonacci>d_0 在 ildasm 中）。
2. 该类实现了一个状态机。状态包括方法中的当前位置和局部变量的值。
3. 最有趣的代码在 bool IEnumerator.MoveNext() 方法中。基本上，MoveNext() 的作用是：

```
// But the yield syntax and usage is identical.
public IEnumerator<T> GetEnumerator()
{
    foreach(var item in _wrapped)
    {
        Console.WriteLine("Yielding: " + item);
        yield return item;
    }
}

IEnumerator IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}
}
```

(Note that this particular example is just illustrative, and could be more cleanly implemented with a single iterator method returning an IEnumerable<T>.)

Section 120.9: Lazy Evaluation Example: Fibonacci Numbers

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Numerics; // also add reference to System.Numerics

namespace ConsoleApplication33
{
    class Program
    {
        private static IEnumerable<BigInteger> Fibonacci()
        {
            BigInteger prev = 0;
            BigInteger current = 1;
            while (true)
            {
                yield return current;
                var next = prev + current;
                prev = current;
                current = next;
            }
        }

        static void Main()
        {
            // print Fibonacci numbers from 10001 to 10010
            var numbers = Fibonacci().Skip(10000).Take(10).ToArray();
            Console.WriteLine(string.Join(Environment.NewLine, numbers));
        }
    }
}
```

How it works under the hood (I recommend to decompile resulting .exe file in IL Disassembler tool):

1. C# compiler generates a class implementing IEnumerable<BigInteger> and IEnumerator<BigInteger> (<Fibonacci>d_0 in ildasm).
2. This class implements a state machine. State consists of current position in method and values of local variables.
3. The most interesting code are in bool IEnumerator.MoveNext() method. Basically, what MoveNext() do:

- 恢复当前状态。像 prev 和 current 这样的变量变成了我们类中的字段（在 ildasm 中为 <current>5_2 和<prev>5_1）。在我们的方法中有两个位置 (<>1_state)：第一个在开括号处，第二个在 yield return 处。
- 执行代码直到下一个 yield return 或 yield break/}。
- 对于 yield return，结果值被保存，因此 Current 属性可以返回它。返回 true。此时当前状态再次保存，以供下一次 MoveNext 调用。
- 对于 yield break/}，方法直接返回 false，表示迭代完成。

另请注意，第10001个数字长度为468字节。状态机仅将 current 和 prev 变量作为字段保存。而如果我们想保存从第一个到第10000个序列中的所有数字，所消耗的内存大小将超过4兆字节。因此，惰性求值如果使用得当，可以在某些情况下减少内存占用。

第120.10节：断裂与屈服断裂的区别

使用yield break而不是break可能没有人们想象的那么明显。网上有很多糟糕的例子，将两者的用法混用，实际上并没有真正展示它们的区别。

令人困惑的是，这两个关键字（或关键短语）只有在循环（foreach、while...）中才有意义。那么，什么时候选择其中一个呢？

重要的是要意识到，一旦你在方法中使用了yield关键字，你实际上就把该方法变成了一个迭代器。这样的方法的唯一目的是遍历一个有限或无限的集合并生成（输出）其元素。一旦目的达成，就没有理由继续执行方法。有时，这自然发生在方法的最后一个闭合括号}处。但有时，你想提前结束方法。在普通（非迭代）方法中，你会使用return关键字。但在迭代器中不能使用return，必须使用yield break。换句话说，对于迭代器来说，yield break相当于普通方法中的return。而break语句只是终止最近的循环。

让我们看一些例子：

```
/// <summary>
/// 生成从0到9的数字
/// </summary>
/// <returns>{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}</returns>
public static IEnumerable<int> YieldBreak()
{
    for (int i = 0; ; i++)
    {
        if (i < 10)
        {
            // 生成一个数字
            yield return i;
        }
        else
        {
            // 表示迭代已结束，从这一行开始的所有内容
            // 都将被忽略
            yield break;
        }
    }
    yield return 10; // 这行代码永远不会被执行
}

/// <summary>
/// 生成从0到10的数字
/// </summary>
/// <returns>{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}</returns>
```

- Restores current state. Variables like prev and current become fields in our class (<current>5_2 and <prev>5_1 in ildasm). In our method we have two positions (<>1_state): first at the opening curly brace, second at yield return.
- Executes code until next yield return or yield break/}.
- For yield return resulting value is saved, so Current property can return it. true is returned. At this point current state is saved again for the next MoveNext invocation.
- For yield break/} method just returns false meaning iteration is done.

Also note, that 10001th number is 468 bytes long. State machine only saves current and prev variables as fields. While if we would like to save all numbers in the sequence from the first to the 10000th, the consumed memory size will be over 4 megabytes. So lazy evaluation, if properly used, can reduce memory footprint in some cases.

Section 120.10: The difference between break and yield break

Using yield break as opposed to break might not be as obvious as one may think. There are lot of bad examples on the Internet where the usage of the two is interchangeable and doesn't really demonstrate the difference.

The confusing part is that both of the keywords (or key phrases) make sense only within loops (foreach, while...) So when to choose one over the other?

It's important to realize that once you use the [yield](#) keyword in a method you effectively turn the method into an [iterator](#). The only purpose of the such method is then to iterate over a finite or infinite collection and yield (output) its elements. Once the purpose is fulfilled, there's no reason to continue method's execution. Sometimes, it happens naturally with the last closing bracket of the method }. But sometimes, you want to end the method prematurely. In a normal (non-iterating) method you would use the [return](#) keyword. But you can't use [return](#) in an iterator, you have to use [yield break](#). In other words, [yield break](#) for an iterator is the same as [return](#) for a standard method. Whereas, the [break](#) statement just terminates the closest loop.

Let's see some examples:

```
/// <summary>
/// Yields numbers from 0 to 9
/// </summary>
/// <returns>{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}</returns>
public static IEnumerable<int> YieldBreak()
{
    for (int i = 0; ; i++)
    {
        if (i < 10)
        {
            // Yields a number
            yield return i;
        }
        else
        {
            // Indicates that the iteration has ended, everything
            // from this line on will be ignored
            yield break;
        }
    }
    yield return 10; // This will never get executed
}

/// <summary>
/// Yields numbers from 0 to 10
/// </summary>
/// <returns>{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}</returns>
```

```

public static IEnumerable<int> Break()
{
    for (int i = 0; ; i++)
    {
        if (i < 10)
        {
            // 生成一个数字
            yield return i;
        }
        else
        {
            // 仅终止循环
            break;
        }
    }
    // 继续执行
    yield return 10;
}

```

第120.11节：在返回Enumerable的方法中返回另一个Enumerable

```

public IEnumerable<int> F1()
{
    for (int i = 0; i < 3; i++)
        yield return i;

    //return F2(); // 编译错误！
    foreach (var element in F2())
        yield return element;
}

public int[] F2()
{
    return new[] { 3, 4, 5 };
}

```

```

public static IEnumerable<int> Break()
{
    for (int i = 0; ; i++)
    {
        if (i < 10)
        {
            // Yields a number
            yield return i;
        }
        else
        {
            // Terminates just the loop
            break;
        }
    }
    // Execution continues
    yield return 10;
}

```

Section 120.11: Return another Enumerable within a method returning Enumerable

```

public IEnumerable<int> F1()
{
    for (int i = 0; i < 3; i++)
        yield return i;

    //return F2(); // Compile Error!!
    foreach (var element in F2())
        yield return element;
}

public int[] F2()
{
    return new[] { 3, 4, 5 };
}

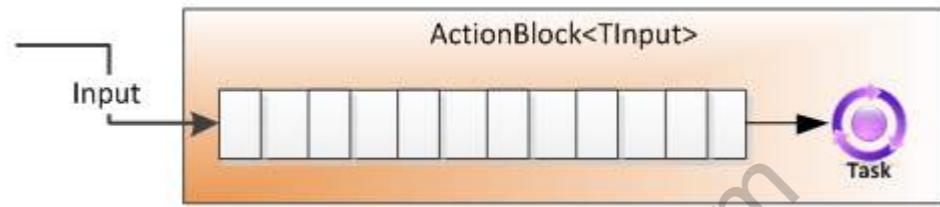
```

第121章：任务并行库 (TPL) 数据流构造

第121.1节：ActionBlock<T>

(foreach)

这个类可以逻辑上被视为一个用于处理数据的缓冲区，结合了处理这些数据的任务，“数据流块”同时管理这两者。在最基本的用法中，我们可以实例化一个ActionBlock并向其“发布”数据；在ActionBlock构造时提供的委托将会异步执行每一条发布的数据。



同步计算

```
var ab = new ActionBlock<TInput>(i =>
{
    Compute(i);
});

...
ab.Post(1);
ab.Post(2);
ab.Post(3);
```

限制异步下载最多同时进行5个

```
var downloader = new ActionBlock<string>(async url =>
{
    byte[] imageData = await DownloadAsync(url);
    Process(imageData);
}, new DataflowBlockOptions { MaxDegreeOfParallelism = 5 });

downloader.Post("http://website.com/path/to/images");
downloader.Post("http://another-website.com/path/to/images");
```

[Stephen Toub 的 TPL 数据流介绍](#)

第121.2节：BroadcastBlock<T>

(复制一个项目并将副本发送到与其'链接的每个块) 与 BufferBlock

不同，BroadcastBlock'的使命是使所有从该块链接的目标都能获得每个发布元素的副本，持续用传播给它的“当前”值覆盖。

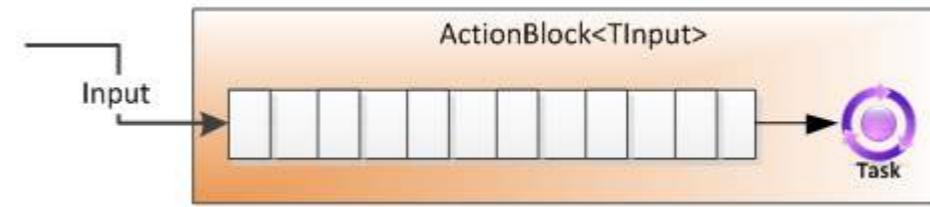
此外，与 BufferBlock 不同，BroadcastBlock 不会不必要地保留数据。在某个数据被提供给所有目标后，该元素将被下一条数据覆盖（与所有数据流块一样，消息按 FIFO 顺序处理）。该元素将被提供给所有目标，依此类推。

Chapter 121: Task Parallel Library (TPL) Dataflow Constructs

Section 121.1: ActionBlock<T>

(foreach)

This class can be thought of logically as a buffer for data to be processed combined with tasks for processing that data, with the “dataflow block” managing both. In its most basic usage, we can instantiate an ActionBlock and “post” data to it; the delegate provided at the ActionBlock’s construction will be executed asynchronously for every piece of data posted.



Synchronous Computation

```
var ab = new ActionBlock<TInput>(i =>
{
    Compute(i);
});

...
ab.Post(1);
ab.Post(2);
ab.Post(3);
```

Throttling Asynchronous Downloads to at most 5 concurrently

```
var downloader = new ActionBlock<string>(async url =>
{
    byte[] imageData = await DownloadAsync(url);
    Process(imageData);
}, new DataflowBlockOptions { MaxDegreeOfParallelism = 5 });

downloader.Post("http://website.com/path/to/images");
downloader.Post("http://another-website.com/path/to/images");
```

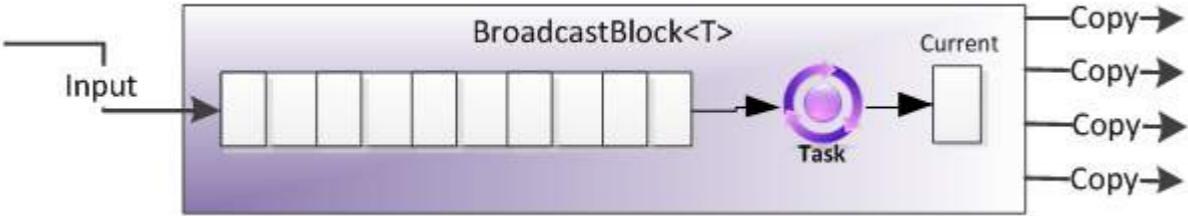
[Introduction to TPL Dataflow by Stephen Toub](#)

Section 121.2: BroadcastBlock<T>

(Copy an item and send the copies to every block that it's linked to)

Unlike BufferBlock, BroadcastBlock's mission in life is to enable all targets linked from the block to get a copy of every element published, continually overwriting the “current” value with those propagated to it.

Additionally, unlike BufferBlock, BroadcastBlock doesn't hold on to data unnecessarily. After a particular datum has been offered to all targets, that element will be overwritten by whatever piece of data is next in line (as with all dataflow blocks, messages are handled in FIFO order). That element will be offered to all targets, and so on.



带节流生产者的异步生产者/消费者

```
var ui = TaskScheduler.FromCurrentSynchronizationContext();
var bb = new BroadcastBlock<ImageData>(i => i);

var saveToDiskBlock = new ActionBlock<ImageData>(item =>
    item.Image.Save(item.Path)
);

var showInUiBlock = new ActionBlock<ImageData>(item =>
    imagePanel.AddImage(item.Image),
    new DataflowBlockOptions { TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext() }
);

bb.LinkTo(saveToDiskBlock);
bb.LinkTo(showInUiBlock);
```

从代理暴露状态

```
public class MyAgent
{
    public ISourceBlock<string> Status { get; private set; }

    public MyAgent()
    {
        Status = new BroadcastBlock<string>();
        Run();
    }

    private void Run()
    {
        Status.Post("Starting");
        Status.Post("Doing cool stuff");
        ...
        Status.Post("Done");
    }
}
```

[Stephen Toub 的 TPL 数据流介绍](#)

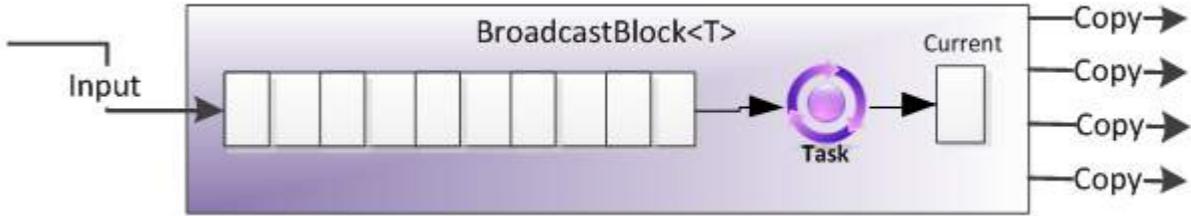
第121.3节：BufferBlock<T>

(先进先出队列：进入的数据就是出去的数据)

简而言之，`BufferBlock` 提供了一个用于存储 `T` 类型实例的无界或有界缓冲区。

您可以“post”类型为 `T` 的实例到该块，这会导致被发布的数据按照先进先出（FIFO）顺序被块存储。

您可以“receive”从该块接收，这允许您同步或异步地获取之前存储的或将来可用的类型 `T` 实例（同样是 FIFO）。



Asynchronous Producer/Consumer with a Throttled Producer

```
var ui = TaskScheduler.FromCurrentSynchronizationContext();
var bb = new BroadcastBlock<ImageData>(i => i);

var saveToDiskBlock = new ActionBlock<ImageData>(item =>
    item.Image.Save(item.Path)
);

var showInUiBlock = new ActionBlock<ImageData>(item =>
    imagePanel.AddImage(item.Image),
    new DataflowBlockOptions { TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext() }
);

bb.LinkTo(saveToDiskBlock);
bb.LinkTo(showInUiBlock);
```

Exposing Status from an Agent

```
public class MyAgent
{
    public ISourceBlock<string> Status { get; private set; }

    public MyAgent()
    {
        Status = new BroadcastBlock<string>();
        Run();
    }

    private void Run()
    {
        Status.Post("Starting");
        Status.Post("Doing cool stuff");
        ...
        Status.Post("Done");
    }
}
```

[Introduction to TPL Dataflow by Stephen Toub](#)

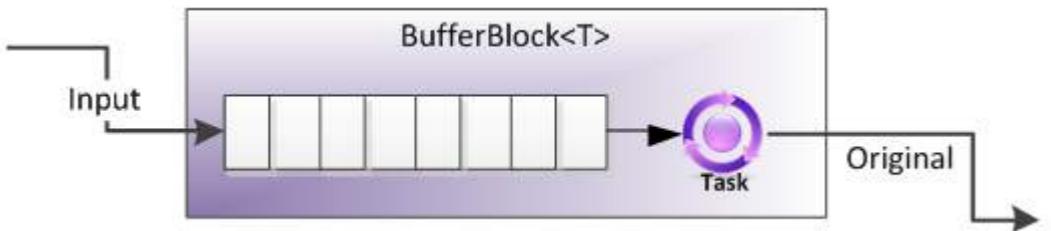
Section 121.3: BufferBlock<T>

(FIFO Queue: The data that comes in is the data that goes out)

In short, `BufferBlock` provides an unbounded or bounded buffer for storing instances of `T`.

You can “post” instances of `T` to the block, which cause the data being posted to be stored in a first-in-first-out (FIFO) order by the block.

You can “receive” from the block, which allows you to synchronously or asynchronously obtain instances of `T` previously stored or available in the future (again, FIFO).



带节流生产者的异步生产者/消费者

```
// 通过有界的 BufferBlock<T> 进行交接
private static BufferBlock<int> _Buffer = new BufferBlock<int>(
    new DataflowBlockOptions { BoundedCapacity = 10 });

// 生产者
private static async void Producer()
{
    while(true)
    {
        await _Buffer.SendAsync(Produce());
    }
}

// 消费者
private static async Task Consumer()
{
    while(true)
    {
        Process(await _Buffer.ReceiveAsync());
    }
}

// 启动生产者和消费者
private static async Task Run()
{
    await Task.WhenAll(Producer(), Consumer());
}
```

[Stephen Toub 的 TPL 数据流介绍](#)

第121.4节 : JoinBlock<T1, T2,...>

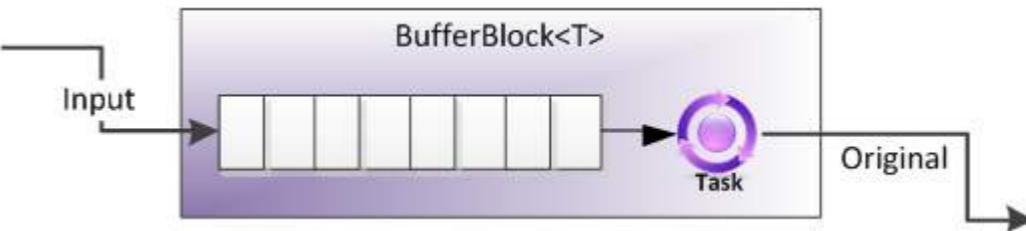
(收集2-3个输入并将它们组合成一个元组)

与BatchBlock类似，JoinBlock<T1, T2, ...>能够从多个数据源分组数据。事实上，这就是JoinBlock<T1, T2, ...>的主要用途。

例如，JoinBlock<string, double, int>是一个ISourceBlock<Tuple<string, double, int>>。

与BatchBlock一样，JoinBlock<T1, T2,...>能够在贪婪和非贪婪模式下运行。

- 在默认的贪婪模式下，所有提供给目标的数据都会被接受，即使其他目标没有形成元组所需的数据。
- 在非贪婪模式下，块's 目标将会推迟数据，直到所有目标都已提供创建元组所需的数据，此时该块将启动两阶段提交协议，以原子方式从源获取所有必要的项目。这种推迟使得其他实体可以在此期间消费数据，从而允许整个系统向前推进。



Asynchronous Producer/Consumer with a Throttled Producer

```
// Hand-off through a bounded BufferBlock<T>
private static BufferBlock<int> _Buffer = new BufferBlock<int>(
    new DataflowBlockOptions { BoundedCapacity = 10 });

// Producer
private static async void Producer()
{
    while(true)
    {
        await _Buffer.SendAsync(Produce());
    }
}

// Consumer
private static async Task Consumer()
{
    while(true)
    {
        Process(await _Buffer.ReceiveAsync());
    }
}

// Start the Producer and Consumer
private static async Task Run()
{
    await Task.WhenAll(Producer(), Consumer());
}
```

[Introduction to TPL Dataflow by Stephen Toub](#)

Section 121.4: JoinBlock<T1, T2,...>

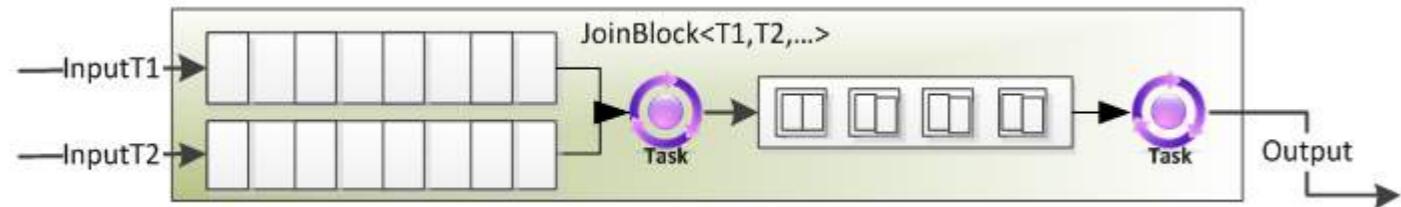
(Collects 2-3 inputs and combines them into a Tuple)

Like BatchBlock, JoinBlock<T1, T2, ...> is able to group data from multiple data sources. In fact, that's JoinBlock<T1, T2, ...>'s primary purpose.

For example, a JoinBlock<string, double, int> is an ISourceBlock<Tuple<string, double, int>>.

As with BatchBlock, JoinBlock<T1, T2,...> is capable of operating in both greedy and non-greedy mode.

- In the default greedy mode, all data offered to targets are accepted, even if the other target doesn't have the necessary data with which to form a tuple.
- In non-greedy mode, the block's targets will postpone data until all targets have been offered the necessary data to create a tuple, at which point the block will engage in a two-phase commit protocol to atomically retrieve all necessary items from the sources. This postponement makes it possible for another entity to consume the data in the meantime so as to allow the overall system to make forward progress.



使用有限数量的对象池处理请求

```
var throttle = new JoinBlock<ExpensiveObject, Request>();
for(int i=0; i<10; i++)
{
    requestProcessor.Target1.Post(new ExpensiveObject());
}

var processor = new Transform<Tuple<ExpensiveObject, Request>, ExpensiveObject>(pair =>
{
    var resource = pair.Item1;
    var request = pair.Item2;

    request.ProcessWith(resource);

    return resource;
});

throttle.LinkTo(processor);
processor.LinkTo(throttle.Target1);
```

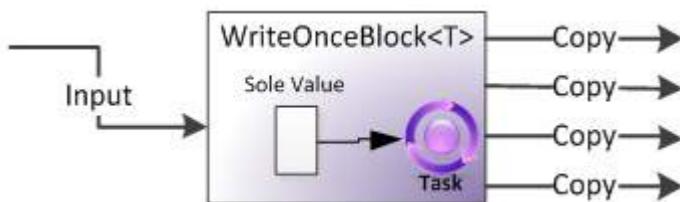
[Stephen Toub 的 TPL 数据流介绍](#)

第121.5节：WriteOnceBlock<T>

(只读变量：记忆其第一个数据项并输出其副本。忽略所有其他数据项)

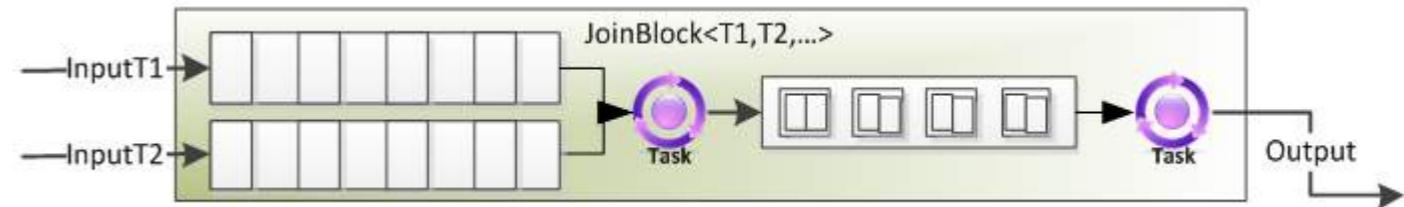
如果说BufferBlock是TPL Dataflow中最基本的块，那么WriteOnceBlock就是最简单的。
它最多存储一个值，一旦该值被设置，就永远不会再被替换或覆盖。

你可以将WriteOnceBlock看作类似于C#中的只读成员变量，不同的是它不是只能在构造函数中设置然后不可变，而是只可设置一次，之后不可变。



拆分任务的潜在输出

```
public static async void SplitIntoBlocks(this Task<T> task,
    out IPropagatorBlock<T> result,
    out IPropagatorBlock<Exception> exception)
{
    result = new WriteOnceBlock<T>(i => i);
    exception = new WriteOnceBlock<Exception>(i => i);
```



Processing Requests with a Limited Number of Pooled Objects

```
var throttle = new JoinBlock<ExpensiveObject, Request>();
for(int i=0; i<10; i++)
{
    requestProcessor.Target1.Post(new ExpensiveObject());
}

var processor = new Transform<Tuple<ExpensiveObject, Request>, ExpensiveObject>(pair =>
{
    var resource = pair.Item1;
    var request = pair.Item2;

    request.ProcessWith(resource);

    return resource;
});

throttle.LinkTo(processor);
processor.LinkTo(throttle.Target1);
```

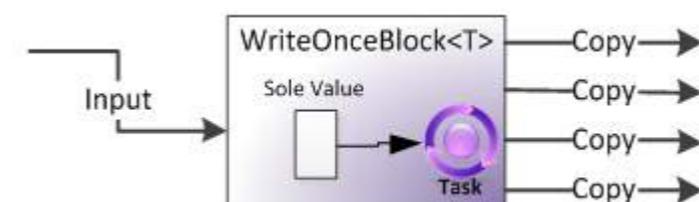
[Introduction to TPL Dataflow by Stephen Toub](#)

Section 121.5: WriteOnceBlock<T>

(Readonly variable: Memorizes its first data item and passes out copies of it as its output. Ignores all other data items)

If BufferBlock is the most fundamental block in TPL Dataflow, WriteOnceBlock is the simplest.
It stores at most one value, and once that value has been set, it will never be replaced or overwritten.

You can think of WriteOnceBlock in as being similar to a readonly member variable in C#, except instead of only being settable in a constructor and then being immutable, it's only settable once and is then immutable.



Splitting a Task's Potential Outputs

```
public static async void SplitIntoBlocks(this Task<T> task,
    out IPropagatorBlock<T> result,
    out IPropagatorBlock<Exception> exception)
{
    result = new WriteOnceBlock<T>(i => i);
    exception = new WriteOnceBlock<Exception>(i => i);
```

```

try
{
result.Post(await task);
}
catch(Exception ex)
{
exception.Post(ex);
}
}

```

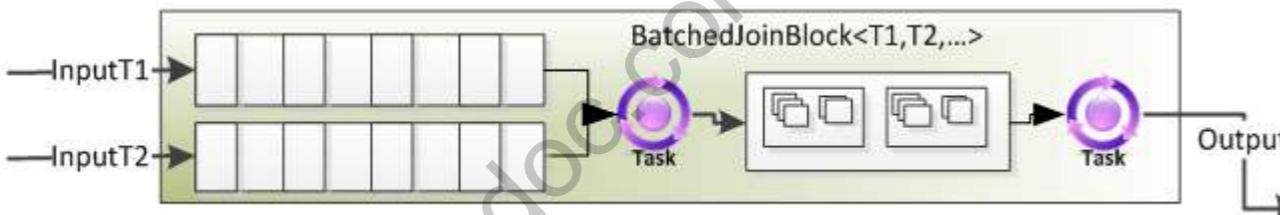
[Stephen Toub 的 TPL 数据流介绍](#)

第121.6节：BatchedJoinBlock<T1, T2,...>

(从2-3个输入中收集一定数量的总项，并将它们分组为数据项集合的元组)

BatchedJoinBlock<T1, T2,...>在某种意义上是BatchBlock和JoinBlock<T1, T2,...>的组合。

JoinBlock<T1, T2,...>用于将每个目标的一个输入聚合成一个元组，BatchBlock用于将N个输入聚合成一个集合，而BatchedJoinBlock<T1, T2,...>用于从所有目标中收集N个输入，形成集合的元组。



分散/收集

考虑一个分散/收集问题，其中启动了N个操作，有些操作可能成功并产生字符串输出，有些操作可能失败并产生异常。

```

var batchedJoin = new BatchedJoinBlock<string, Exception>(10);

for (int i=0; i<10; i++)
{
    Task.Factory.StartNew(() => {
        try { batchedJoin.Target1.Post(DoWork()); }
        catch(Exception ex) { batchedJoin.Target2.Post(ex); }
    });
}

var results = await batchedJoin.ReceiveAsync();

foreach(string s in results.Item1)
{
    Console.WriteLine(s);
}

foreach(Exception e in results.Item2)
{
    Console.WriteLine(e);
}

```

[Stephen Toub 的 TPL 数据流介绍](#)

```

try
{
    result.Post(await task);
}
catch(Exception ex)
{
    exception.Post(ex);
}
}

```

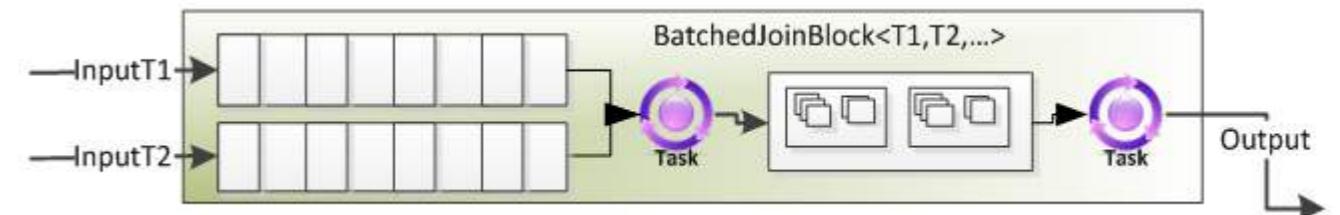
[Introduction to TPL Dataflow by Stephen Toub](#)

Section 121.6: BatchedJoinBlock<T1, T2,...>

(Collects a certain number of total items from 2-3 inputs and groups them into a Tuple of collections of data items)

BatchedJoinBlock<T1, T2,...> is in a sense a combination of BatchBlock and JoinBlock<T1, T2,...>.

Whereas JoinBlock<T1, T2,...> is used to aggregate one input from each target into a tuple, and BatchBlock is used to aggregate N inputs into a collection, BatchedJoinBlock<T1, T2,...> is used to gather N inputs from across all of the targets into tuples of collections.



Scatter/Gather

Consider a scatter/gather problem where N operations are launched, some of which may succeed and produce string outputs, and others of which may fail and produce Exceptions.

```

var batchedJoin = new BatchedJoinBlock<string, Exception>(10);

for (int i=0; i<10; i++)
{
    Task.Factory.StartNew(() => {
        try { batchedJoin.Target1.Post(DoWork()); }
        catch(Exception ex) { batchedJoin.Target2.Post(ex); }
    });
}

var results = await batchedJoin.ReceiveAsync();

foreach(string s in results.Item1)
{
    Console.WriteLine(s);
}

foreach(Exception e in results.Item2)
{
    Console.WriteLine(e);
}

```

[Introduction to TPL Dataflow by Stephen Toub](#)

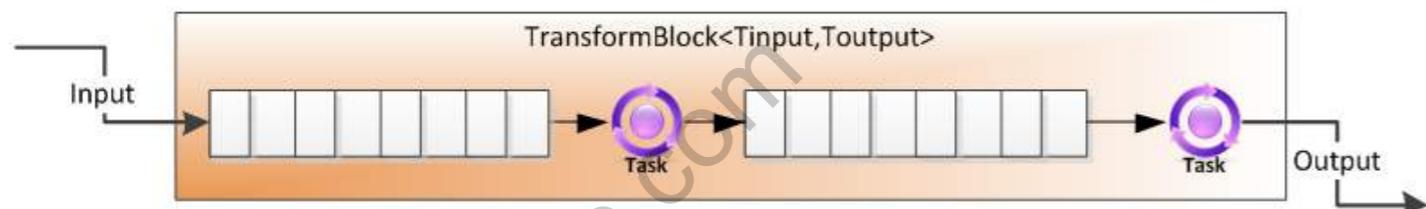
第121.7节 : TransformBlock<TInput, TOutput>

(选择, 一对一)

与ActionBlock类似, TransformBlock<TInput, TOutput>允许执行一个委托来对每个输入数据执行某些操作 ; 不同于ActionBlock, 这个处理有输出。该委托可以是Func<TInput, TOutput>, 在这种情况下, 当委托返回时, 该元素的处理被视为完成 ; 也可以是Func<TInput, Task>, 在这种情况下, 该元素的处理被视为完成不是在委托返回时, 而是在返回的Task完成时。对于熟悉LINQ的人来说, 它'有点类似于Select(), 因为它接受输入, 以某种方式转换该输入, 然后产生输出。

默认情况下, TransformBlock<TInput, TOutput>以最大并行度为1顺序处理其数据。除了接收缓冲的输入并进行处理外, 该块还会获取所有已处理的输出并对其进行缓冲 (未处理的数据和已处理的数据)。

它有两个任务 : 一个用于处理数据, 另一个用于将数据推送到下一个块。



一个并发管道

```
var compressor = new TransformBlock<byte[], byte[]>(input => Compress(input));
var encryptor = new TransformBlock<byte[], byte[]>(input => Encrypt(input));

compressor.LinkTo(Encryptor);
```

[Stephen Toub 的 TPL 数据流介绍](#)

第121.8节 : TransformManyBlock<TInput, TOutput>

(SelectMany, 1对多 : 此映射的结果被“扁平化”, 就像LINQ的SelectMany一样)

TransformManyBlock<TInput, TOutput>与TransformBlock<TInput, TOutput>非常相似。关键区别在于, TransformBlock<TInput, TOutput>对每个输入只产生一个输出, 而TransformManyBlock<TInput, TOutput>对每个输入可以产生任意数量 (零个或多个) 输出。与ActionBlock和TransformBlock<TInput, TOutput>一样, 这种处理可以通过委托指定, 既支持同步也支持异步处理。

同步处理使用Func<TInput, IEnumerable>; 异步处理使用Func<TInput, Task<IEnumerable>>; 与ActionBlock和TransformBlock<TInput, TOutput>一样, TransformManyBlock<TInput, TOutput>默认采用顺序处理, 但也可以配置为其他方式。

映射委托返回一个项目集合, 这些项目会被逐个插入到输出缓冲区中。

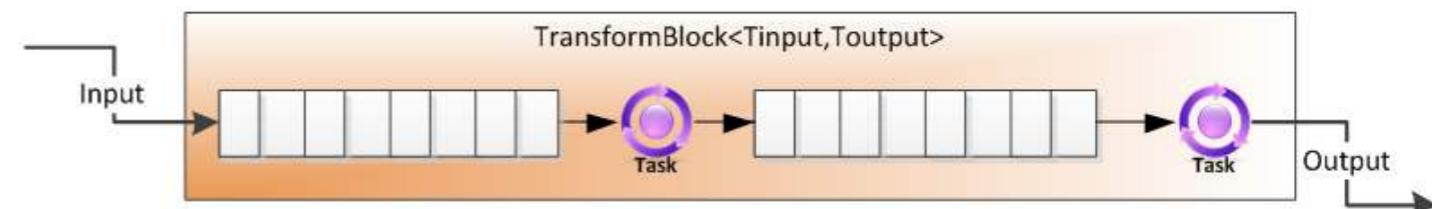
Section 121.7: TransformBlock<TInput, TOutput>

(Select, one-to-one)

As with ActionBlock, TransformBlock<TInput, TOutput> enables the execution of a delegate to perform some action for each input datum; **unlike with ActionBlock, this processing has an output**. This delegate can be a Func<TInput, TOutput>, in which case processing of that element is considered completed when the delegate returns, or it can be a Func<TInput, Task>, in which case processing of that element is considered completed not when the delegate returns but when the returned Task completes. For those familiar with LINQ, it's somewhat similar to Select() in that it takes an input, transforms that input in some manner, and then produces an output.

By default, TransformBlock<TInput, TOutput> processes its data sequentially with a MaxDegreeOfParallelism equal to 1. In addition to receiving buffered input and processing it, this block will take all of its processed output and buffer that as well (data that has not been processed, and data that has been processed).

It has 2 tasks: One to process the data, and one to push data to the next block.



A Concurrent Pipeline

```
var compressor = new TransformBlock<byte[], byte[]>(input => Compress(input));
var encryptor = new TransformBlock<byte[], byte[]>(input => Encrypt(input));

compressor.LinkTo(Encryptor);
```

[Introduction to TPL Dataflow by Stephen Toub](#)

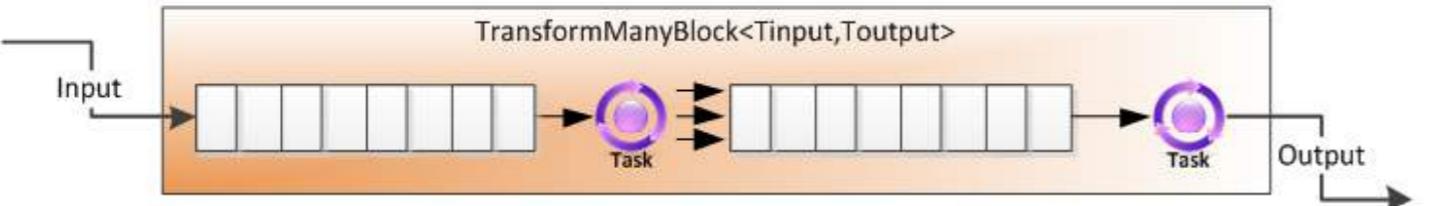
Section 121.8: TransformManyBlock<TInput, TOutput>

(SelectMany, 1-m: The results of this mapping are “flattened”, just like LINQ's SelectMany)

TransformManyBlock<TInput, TOutput> is very similar to TransformBlock<TInput, TOutput>. The key difference is that whereas a TransformBlock<TInput, TOutput> produces one and only one output for each input, TransformManyBlock<TInput, TOutput> produces any number (zero or more) outputs for each input. As with ActionBlock and TransformBlock<TInput, TOutput>, this processing may be specified using delegates, both for synchronous and asynchronous processing.

A Func<TInput, IEnumerable> is used for synchronous, and a Func<TInput, Task<IEnumerable>> is used for asynchronous. As with both ActionBlock and TransformBlock<TInput, TOutput>, TransformManyBlock<TInput, TOutput> defaults to sequential processing, but may be configured otherwise.

The mapping delegate returns a collection of items, which are inserted individually into the output buffer.



异步网页爬虫

```
var downloader = new TransformManyBlock<string, string>(async url =>
{
    Console.WriteLine("正在下载 " + url);
    try
    {
        return ParseLinks(await DownloadContents(url));
    }
    catch {}

    return Enumerable.Empty<string>();
});
downloader.LinkTo(downloader);
```

将一个 `Enumerable` 展开为其组成元素

```
var expanded = new TransformManyBlock<T[], T>(array => array);
```

通过从1到0或1个元素进行过滤

```
public IPropagatorBlock<T> CreateFilteredBuffer<T>(Predicate<T> filter)
{
    return new TransformManyBlock<T, T>(item =>
        filter(item) ? new [] { item } : Enumerable.Empty<T>());
}
```

[Stephen Toub 的 TPL 数据流介绍](#)

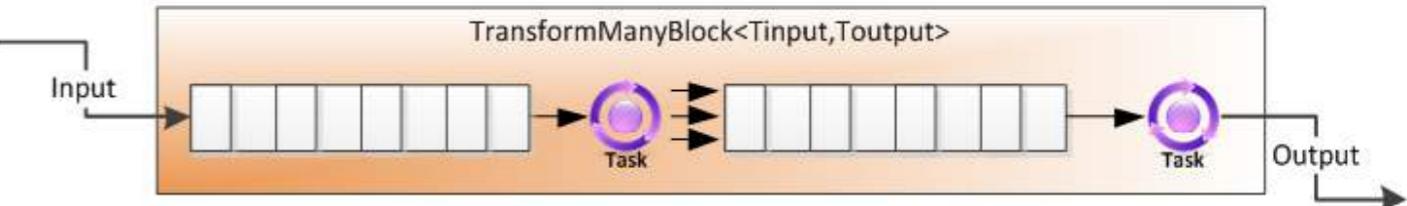
第121.9节：BatchBlock<T>

(将一定数量的连续数据项分组为数据项集合)BatchBlock 将 N 个单个项组合成一个

批处理项，表示为元素数组。实例以特定的批处理大小创建，块在接收到该数量的元素后立即创建一个批处理，异步地将批处理输出到输出缓冲区。

BatchBlock能够在贪婪模式和非贪婪模式下执行。

- 在默认的贪婪模式下，来自任意数量源的所有消息都会被接受并缓冲，以便转换成批次。
 - 在非贪婪模式下，所有消息都会被延迟处理，直到有足够的源向块提供消息以创建一个批次。因此，BatchBlock 可以用来从每个 N 个源接收 1 个元素，从 1 个源接收 N 个元素，以及介于两者之间的各种选项。



Asynchronous Web Crawler

```
var downloader = new TransformManyBlock<string, string>(async url =>
{
    Console.WriteLine("Downloading " + url);
    try
    {
        return ParseLinks(await DownloadContents(url));
    }
    catch {}

    return Enumerable.Empty<string>();
});
downloader.LinkTo(downloader);
```

Expanding an `Enumerable` Into Its Constituent Elements

```
var expanded = new TransformManyBlock<T[], T>(array => array);
```

Filtering by going from 1 to 0 or 1 elements

```
public IPropagatorBlock<T> CreateFilteredBuffer<T>(Predicate<T> filter)
{
    return new TransformManyBlock<T, T>(item =>
        filter(item) ? new [] { item } : Enumerable.Empty<T>());
}
```

[Introduction to TPL Dataflow by Stephen Toub](#)

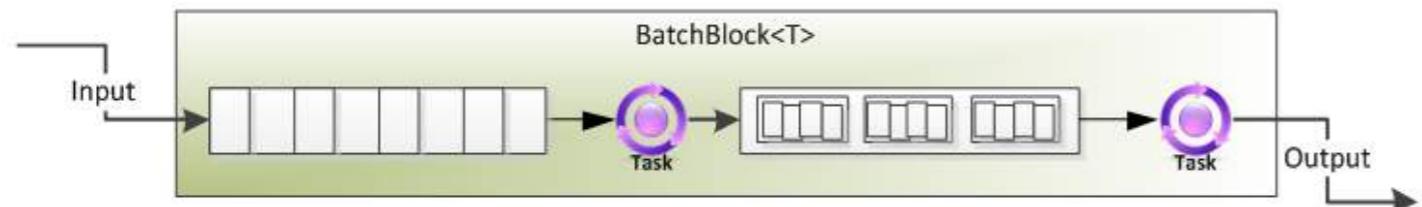
Section 121.9: BatchBlock<T>

(Groups a certain number of sequential data items into collections of data items)

BatchBlock combines N single items into one batch item, represented as an array of elements. An instance is created with a specific batch size, and the block then creates a batch as soon as it's received that number of elements, asynchronously outputting the batch to the output buffer.

BatchBlock is capable of executing in both greedy and non-greedy modes.

- In the default greedy mode, all messages offered to the block from any number of sources are accepted and buffered to be converted into batches.
 - In non-greedy mode, all messages are postponed from sources until enough sources have offered messages to the block to create a batch. Thus, a BatchBlock can be used to receive 1 element from each of N sources, N elements from 1 source, and a myriad of options in between.



将请求批量分组为 100 个以提交到数据库

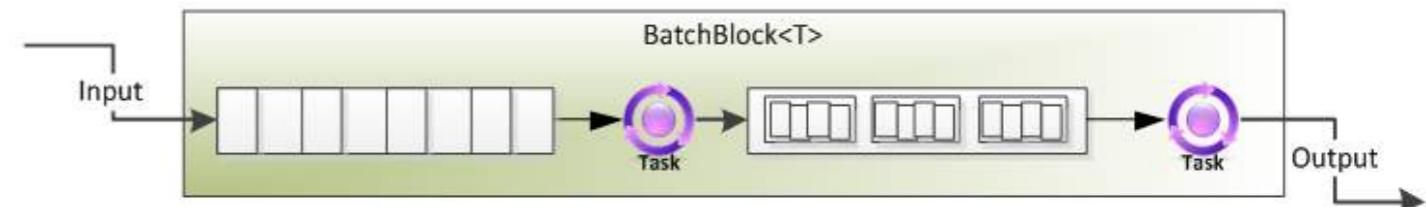
```
var batchRequests = new BatchBlock<Request>(batchSize:100);
var sendToDb = new ActionBlock<Request[]>(reqs => SubmitToDatabase(reqs));

batchRequests.LinkTo(sendToDb);
```

每秒创建一个批次

```
var batch = new BatchBlock<T>(batchSize:Int32.MaxValue);
new Timer(() => { batch.TriggerBatch(); }).Change(1000, 1000);
```

[Stephen Toub 的 TPL 数据流介绍](#)



Batching Requests into groups of 100 to Submit to a Database

```
var batchRequests = new BatchBlock<Request>(batchSize:100);
var sendToDb = new ActionBlock<Request[]>(reqs => SubmitToDatabase(reqs));

batchRequests.LinkTo(sendToDb);
```

Creating a batch once a second

```
var batch = new BatchBlock<T>(batchSize:Int32.MaxValue);
new Timer(() => { batch.TriggerBatch(); }).Change(1000, 1000);
```

[Introduction to TPL Dataflow by Stephen Toub](#)

第122章：函数式编程

第122.1节：Func 和 Action

Func 提供了一个用于参数化匿名函数的持有者。前面的类型是输入类型，最后一个类型始终是返回值。

```
// 计算一个数的平方。  
Func<double, double> square = (x) => { return x * x; };  
  
// 计算平方根。  
// 注意签名如何与内置方法匹配。  
Func<double, double> squareroot = Math.Sqrt;  
  
// 提供你的计算过程。  
Func<double, double, string> workings = (x, y) =>  
    string.Format("The square of {0} is {1}.", x, square(y))
```

Action 对象类似于void方法，因此它们只有输入类型。评估栈上不会放置结果。

```
// 直角三角形。  
class Triangle  
{  
    public double a;  
    public double b;  
    public double h;  
}  
  
// 勾股定理。  
Action<Triangle> pythagoras = (x) =>  
    x.h = squareroot(square(x.a) + square(x.b));  
  
Triangle t = new Triangle { a = 3, b = 4 };  
pythagoras(t);  
Console.WriteLine(t.h); // 5.
```

第122.2节：高阶函数

高阶函数是指接受另一个函数作为参数或返回一个函数（或两者兼有）的函数。

这通常通过lambda表达式实现，例如在将谓词传递给LINQ的Where子句时：

```
var results = data.Where(p => p.Items == 0);
```

Where()子句可以接收许多不同的谓词，这赋予了它相当大的灵活性。

将一个方法传递给另一个方法也常见于策略设计模式的实现。例如，可以根据运行时的需求从多种排序方法中选择并传送给对象的Sort方法。

第122.3节：避免空引用

C#开发者经常需要处理空引用异常。F#开发者则不需要，因为他们有Option类型。Option<>类型（有些人更喜欢称为Maybe<>）提供了Some和None两种返回类型。它明确表示一个方法可能会返回空记录。

Chapter 122: Functional Programming

Section 122.1: Func and Action

Func provides a holder for parameterised anonymous functions. The leading types are the inputs and the last type is always the return value.

```
// square a number.  
Func<double, double> square = (x) => { return x * x; };  
  
// get the square root.  
// note how the signature matches the built in method.  
Func<double, double> squareroot = Math.Sqrt;  
  
// provide your workings.  
Func<double, double, string> workings = (x, y) =>  
    string.Format("The square of {0} is {1}.", x, square(y))
```

Action objects are like void methods so they only have an input type. No result is placed on the evaluation stack.

```
// right-angled triangle.  
class Triangle  
{  
    public double a;  
    public double b;  
    public double h;  
}  
  
// Pythagorean theorem.  
Action<Triangle> pythagoras = (x) =>  
    x.h = squareroot(square(x.a) + square(x.b));  
  
Triangle t = new Triangle { a = 3, b = 4 };  
pythagoras(t);  
Console.WriteLine(t.h); // 5.
```

Section 122.2: Higher-Order Functions

A higher-order function is one that takes another function as an argument or returns a function (or both).

This is commonly done with lambdas, for example when passing a predicate to a LINQ Where clause:

```
var results = data.Where(p => p.Items == 0);
```

The Where() clause could receive many different predicates which gives it considerable flexibility.

Passing a method into another method is also seen when implementing the Strategy design pattern. For example, various sorting methods could be chosen from and passed into a Sort method on an object depending on the requirements at run-time.

Section 122.3: Avoid Null References

C# developers get a lot of null reference exceptions to deal with. F# developers don't because they have the Option type. An Option<> type (some prefer Maybe<> as a name) provides a Some and a None return type. It makes it explicit that a method may be about to return a null record.

例如，你无法通过以下代码判断是否需要处理空值。

```
var user = _repository.GetUser(id);
```

如果你知道可能会有空值，可以引入一些样板代码来处理它。

```
var username = user != null ? user.Name : string.Empty;
```

如果返回的是Option<>类型呢？

```
Option<User> maybeUser = _repository.GetUser(id);
```

代码现在明确表示可能返回None记录，因此需要样板代码来检查Some或None：

```
var username = maybeUser.HasValue ? maybeUser.Value.Name : string.Empty;
```

以下方法展示了如何返回一个 Option<>；

```
public Option<User> GetUser(int id)
{
    var users = new List<User>;
    {
        new User { Id = 1, Name = "Joe Bloggs" },
        new User { Id = 2, Name = "John Smith" }
    };

    var user = users.FirstOrDefault(user => user.Id == id);

    return user != null ? new Option<User>(user) : new Option<User>();
}
```

这是 Option<> 的一个最小实现。

```
public struct Option<T>
{
    private readonly T _value;

    public T Value
    {
        get
        {
            if (!HasValue)
                throw new InvalidOperationException();

            return _value;
        }
    }

    public bool HasValue
    {
        get { return _value != null; }
    }

    public Option(T value)
    {
        _value = value;
    }
}
```

For instance, you can't read the following and know if you will have to deal with a null value.

```
var user = _repository.GetUser(id);
```

If you do know about the possible null you can introduce some boilerplate code to deal with it.

```
var username = user != null ? user.Name : string.Empty;
```

What if we have an Option<> returned instead?

```
Option<User> maybeUser = _repository.GetUser(id);
```

The code now makes it explicit that we may have a None record returned and the boilerplate code to check for Some or None is required:

```
var username = maybeUser.HasValue ? maybeUser.Value.Name : string.Empty;
```

The following method shows how to return an Option<>:

```
public Option<User> GetUser(int id)
{
    var users = new List<User>;
    {
        new User { Id = 1, Name = "Joe Bloggs" },
        new User { Id = 2, Name = "John Smith" }
    };

    var user = users.FirstOrDefault(user => user.Id == id);

    return user != null ? new Option<User>(user) : new Option<User>();
}
```

Here is a minimal implementation of Option<>.

```
public struct Option<T>
{
    private readonly T _value;

    public T Value
    {
        get
        {
            if (!HasValue)
                throw new InvalidOperationException();

            return _value;
        }
    }

    public bool HasValue
    {
        get { return _value != null; }
    }

    public Option(T value)
    {
        _value = value;
    }
}
```

```
public static implicit operator Option<T>(T value)
{
    return new Option<T>(value);
}
```

为了演示上述内容，可以使用 [C# REPL](#) 运行 `avoidNull.csx`。

如前所述，这是一个最简实现。搜索 "Maybe" NuGet packages 会找到许多优秀的库。

第122.4节：不可变性

不可变性在函数式编程中很常见，而在面向对象编程中则很少见。

例如，创建一个具有可变状态的地址类型：

```
public class Address ()
{
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string City { get; set; }
}
```

上述对象中的任何代码都可以更改任何属性。

现在创建不可变的地址类型：

```
public class Address ()
{
    public readonly string Line1;
    public readonly string Line2;
    public readonly string City;

    public Address(string line1, string line2, string city)
    {
        Line1 = line1;
        Line2 = line2;
        City = city;
    }
}
```

请记住，拥有只读集合并不等同于遵守不可变性。例如，

```
public class Classroom
{
    public readonly List<Student> Students;

    public Classroom(List<Student> students)
    {
        Students = students;
    }
}
```

它不是不可变的，因为对象的使用者可以更改集合（添加或移除元素）。为了使其不可变，必须使用像 `IEnumerable` 这样的接口，该接口不暴露添加方法，或者将其设为 `ReadOnlyCollection`。

```
public static implicit operator Option<T>(T value)
{
    return new Option<T>(value);
}
```

To demonstrate the above `avoidNull.csx` can be run with the C# REPL.

As stated, this is a minimal implementation. A search for "["Maybe" NuGet packages](#)" will turn up a number of good libraries.

Section 122.4: Immutability

Immutability is common in functional programming and rare in object oriented programming.

Create, for example, an address type with mutable state:

```
public class Address ()
{
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string City { get; set; }
}
```

Any piece of code could alter any property in the above object.

Now create the immutable address type:

```
public class Address ()
{
    public readonly string Line1;
    public readonly string Line2;
    public readonly string City;

    public Address(string line1, string line2, string city)
    {
        Line1 = line1;
        Line2 = line2;
        City = city;
    }
}
```

Bear in mind that having read-only collections does not respect immutability. For example,

```
public class Classroom
{
    public readonly List<Student> Students;

    public Classroom(List<Student> students)
    {
        Students = students;
    }
}
```

is not immutable, as the user of the object can alter the collection (add or remove elements from it). In order to make it immutable, one has either to use an interface like `IEnumerable`, which does not expose methods to add, or to make it a `ReadOnlyCollection`.

```

public class Classroom
{
    public readonly ReadOnlyCollection<Student> Students;

    public Classroom(ReadOnlyCollection<Student> students)
    {
        Students = students;
    }

    List<Students> list = new List<Student>();
    // 添加学生
    Classroom c = new Classroom(list.AsReadOnly());
}

```

使用不可变对象我们有以下好处：

- 它将处于已知状态（其他代码无法更改它）。
- 它是线程安全的。
- 构造函数提供了一个用于验证的单一位置。
- 知道对象不能被修改使代码更易于理解。

第122.5节：不可变集合

System.Collections.Immutable NuGet包提供了不可变集合类。

创建和添加项目

```

var stack = ImmutableStack.Create<int>();
var stack2 = stack.Push(1); // stack 仍然为空, stack2 包含 1
var stack3 = stack.Push(2); // stack2 仍然只包含 1, stack3 包含 2, 1

```

使用构建器创建

某些不可变集合具有一个Builder内部类，可用于廉价地构建大型不可变实例：

```

var builder = ImmutableList.CreateBuilder<int>(); // 返回 ImmutableList.Builder
builder.Add(1);
builder.Add(2);
var list = builder.ToImmutable();

```

从现有的 IEnumerable 创建

```

var numbers = Enumerable.Range(1, 5);
var list = ImmutableList.CreateRange<int>(numbers);

```

所有不可变集合类型的列表：

- [System.Collections.Immutable.ImmutableArray<T>](#)
- [System.Collections.Immutable.ImmutableDictionary< TKey, TValue >](#)
- [System.Collections.Immutable.ImmutableHashSet<T>](#)
- [System.Collections.Immutable.ImmutableList<T>](#)
- [System.Collections.Immutable.ImmutableQueue<T>](#)
- [System.Collections.Immutable.ImmutableSortedDictionary< TKey, TValue >](#)
- [System.Collections.Immutable.ImmutableSortedSet<T>](#)
- [System.Collections.Immutable.ImmutableStack<T>](#)

```

public class Classroom
{
    public readonly ReadOnlyCollection<Student> Students;

    public Classroom(ReadOnlyCollection<Student> students)
    {
        Students = students;
    }

    List<Students> list = new List<Student>();
    // add students
    Classroom c = new Classroom(list.AsReadOnly());
}

```

With the immutable object we have the following benefits:

- It will be in a known state (other code can't change it).
- It is thread safe.
- The constructor offers a single place for validation.
- Knowing that the object cannot be altered makes the code easier to understand.

Section 122.5: Immutable collections

The [System.Collections.Immutable](#) NuGet package provides immutable collection classes.

Creating and adding items

```

var stack = ImmutableStack.Create<int>();
var stack2 = stack.Push(1); // stack is still empty, stack2 contains 1
var stack3 = stack.Push(2); // stack2 still contains only one, stack3 has 2, 1

```

Creating using the builder

Certain immutable collections have a Builder inner class that can be used to cheaply build large immutable instances:

```

var builder = ImmutableList.CreateBuilder<int>(); // returns ImmutableList.Builder
builder.Add(1);
builder.Add(2);
var list = builder.ToImmutable();

```

Creating from an existing IEnumerable

```

var numbers = Enumerable.Range(1, 5);
var list = ImmutableList.CreateRange<int>(numbers);

```

List of all immutable collection types:

- [System.Collections.Immutable.ImmutableArray<T>](#)
- [System.Collections.Immutable.ImmutableDictionary< TKey, TValue >](#)
- [System.Collections.Immutable.ImmutableHashSet<T>](#)
- [System.Collections.Immutable.ImmutableList<T>](#)
- [System.Collections.Immutable.ImmutableQueue<T>](#)
- [System.Collections.Immutable.ImmutableSortedDictionary< TKey, TValue >](#)
- [System.Collections.Immutable.ImmutableSortedSet<T>](#)
- [System.Collections.Immutable.ImmutableStack<T>](#)

第123章：Func 委托

参数	详情
arg 或 arg1	方法的（第一个）参数
arg2	该方法的第二个参数
arg3	该方法的第三个参数
arg4	该方法的第四个参数
T 或 T1	该方法（第一个）参数的类型
T2	该方法第二个参数的类型
T3	该方法第三个参数的类型
T4	方法第四个参数的类型
TResult	方法的返回类型

第123.1节：无参数

此示例展示了如何创建一个委托来封装返回当前时间的方法

```
static DateTime UTCNow()
{
    return DateTime.UtcNow;
}

static DateTime LocalNow()
{
    return DateTime.Now;
}

static void Main(string[] args)
{
    Func<DateTime> method = UTCNow;
    // 方法指向 UTCNow 方法
    // 该方法返回当前 UTC 时间
    DateTime utcNow = method();

    method = LocalNow;
    // 现在方法指向 LocalNow 方法
    // 该方法返回本地时间
    DateTime localNow = method();
}
```

第 123.2 节：多个变量

```
static int Sum(int a, int b)
{
    return a + b;
}

static int Multiplication(int a, int b)
{
    return a * b;
}

static void Main(string[] args)
{
```

Chapter 123: Func delegates

Parameter	Details
arg or arg1	the (first) parameter of the method
arg2	the second parameter of the method
arg3	the third parameter of the method
arg4	the fourth parameter of the method
T or T1	the type of the (first) parameter of the method
T2	the type of the second parameter of the method
T3	the type of the third parameter of the method
T4	the type of the fourth parameter of the method
TResult	the return type of the method

Section 123.1: Without parameters

This example shows how to create a delegate that encapsulates the method that returns the current time

```
static DateTime UTCNow()
{
    return DateTime.UtcNow;
}

static DateTime LocalNow()
{
    return DateTime.Now;
}

static void Main(string[] args)
{
    Func<DateTime> method = UTCNow;
    // method points to the UTCNow method
    // that retuns current UTC time
    DateTime utcNow = method();

    method = LocalNow;
    // now method points to the LocalNow method
    // that returns local time
    DateTime localNow = method();
}
```

Section 123.2: With multiple variables

```
static int Sum(int a, int b)
{
    return a + b;
}

static int Multiplication(int a, int b)
{
    return a * b;
}

static void Main(string[] args)
{
```

```

Func<int, int, int> 方法 = Sum;
// 方法指向返回1个int变量并接受2个int变
量的Sum方法
int sum = 方法(1, 1);

方法 = Multiplication;
// 现在方法指向Multiplication方法

int multiplication = 方法(1, 1);
}

```

第123.3节：Lambda表达式和匿名方法

匿名方法可以赋值给任何期望委托的地方：

```
Func<int, int> square = delegate (int x) { return x * x; }
```

Lambda表达式也可以用来表达同样的内容：

```
Func<int, int> square = x => x * x;
```

无论哪种情况，我们现在都可以这样调用存储在 square中的方法：

```
var sq = square.Invoke(2);
```

或者作为简写：

```
var sq = square(2);
```

注意，为了使赋值类型安全，匿名方法的参数类型和返回类型必须与委托类型匹配：

```

Func<int, int> sum = delegate (int x, int y) { return x + y; } // 错误
Func<int, int> sum = (x, y) => x + y; // 错误

```

第123.4节：协变和逆变类型参数

Func 也支持 协变和逆变

```

// 简单的类层次结构。
public class Person { }
public class Employee : Person { }

class Program
{
    static Employee FindByTitle(String title)
    {
        // 这是一个返回具有指定职位的员工的方法的存根。
        return new Employee();
    }

    static void Test()
    {
        // 创建一个不使用协变的委托实例。
        Func<String, Employee> findEmployee = FindByTitle;
    }
}

```

```

Func<int, int, int> method = Sum;
// method points to the Sum method
// that retuns 1 int variable and takes 2 int variables
int sum = method(1, 1);

method = Multiplication;
// now method points to the Multiplication method

int multiplication = method(1, 1);
}

```

Section 123.3: Lambda & anonymous methods

An anonymous method can be assigned wherever a delegate is expected:

```
Func<int, int> square = delegate (int x) { return x * x; }
```

Lambda expressions can be used to express the same thing:

```
Func<int, int> square = x => x * x;
```

In either case, we can now invoke the method stored inside square like this:

```
var sq = square.Invoke(2);
```

Or as a shorthand:

```
var sq = square(2);
```

Notice that for the assignment to be type-safe, the parameter types and return type of the anonymous method must match those of the delegate type:

```

Func<int, int> sum = delegate (int x, int y) { return x + y; } // error
Func<int, int> sum = (x, y) => x + y; // error

```

Section 123.4: Covariant & Contravariant Type Parameters

Func also supports [Covariant & Contravariant](#)

```

// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }

class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;
    }
}

```

```
// 委托期望一个返回Person的方法,  
// 但你可以将一个返回Employee的方法赋给它。  
Func<String, Person> findPerson = FindByTitle;  
  
// 你也可以将一个返回更派生类型的方法的  
委托// 赋给一个返回较少派生类型的委托。  
  
findPerson = findEmployee;  
  
}  
}
```

belindoc.com

```
// The delegate expects a method to return Person,  
// but you can assign it a method that returns Employee.  
Func<String, Person> findPerson = FindByTitle;  
  
// You can also assign a delegate  
// that returns a more derived type  
// to a delegate that returns a less derived type.  
findPerson = findEmployee;  
  
}
```

第124章：具有多个返回值的函数

第124.1节：“匿名对象”+“dynamic关键字”解决方案

您可以从函数返回一个匿名对象

```
public static object FunctionWithUnknowReturnValues ()  
{  
    //匿名对象  
    return new { a = 1, b = 2 };  
}
```

并将结果赋值给dynamic对象，然后读取其中的值。

```
/// dynamic对象  
dynamic x = FunctionWithUnknowReturnValues();  
  
Console.WriteLine(x.a);  
Console.WriteLine(x.b);
```

第124.2节：元组解决方案

您可以从函数返回一个带有两个模板参数的Tuple类实例，形式为 Tuple<string, MyClass>：

```
public Tuple<string, MyClass> FunctionWith2ReturnValues ()  
{  
    return Tuple.Create("abc", new MyClass());  
}
```

并且像下面这样读取值：

```
Console.WriteLine(x.Item1);  
Console.WriteLine(x.Item2);
```

第124.3节：Ref 和 Out 参数

ref 关键字用于按引用传递参数。out 也具有与ref相同的功能，但调用函数前调用者不需要为其赋值。

Ref 参数：如果您想将变量作为ref参数传递，则需要在将其作为ref参数传递给方法之前对其进行初始化。

输出参数：如果你想将变量作为输出参数传递，则不需要在将其作为输出参数传递给方法之前进行初始化。

```
static void Main(string[] args)  
{  
    int a = 2;  
    int b = 3;  
    int add = 0;  
    int mult= 0;
```

Chapter 124: Function with multiple return values

Section 124.1: "anonymous object" + "dynamic keyword" solution

You can return an anonymous object from your function

```
public static object FunctionWithUnknowReturnValues ()  
{  
    //匿名对象  
    return new { a = 1, b = 2 };  
}
```

And assign the result to a dynamic object and read the values in it.

```
/// dynamic object  
dynamic x = FunctionWithUnknowReturnValues();  
  
Console.WriteLine(x.a);  
Console.WriteLine(x.b);
```

Section 124.2: Tuple solution

You can return an instance of Tuple class from your function with two template parameters as Tuple<string, MyClass>:

```
public Tuple<string, MyClass> FunctionWith2ReturnValues ()  
{  
    return Tuple.Create("abc", new MyClass());  
}
```

And read the values like below:

```
Console.WriteLine(x.Item1);  
Console.WriteLine(x.Item2);
```

Section 124.3: Ref and Out Parameters

The ref keyword is used to pass an Argument as Reference. out will do the same as ref but it does not require an assigned value by the caller prior to calling the function.

Ref Parameter: If you want to pass a variable as ref parameter then you need to initialize it before you pass it as ref parameter to method.

Out Parameter: If you want to pass a variable as out parameter you don't need to initialize it before you pass it as out parameter to method.

```
static void Main(string[] args)  
{  
    int a = 2;  
    int b = 3;  
    int add = 0;  
    int mult= 0;
```

```
AddOrMult(a, b, ref add, ref mult); //AddOrMult(a, b, out add, out mult);
Console.WriteLine(add); //5
Console.WriteLine(mult); //6
}

private static void AddOrMult(int a, int b, ref int add, ref int mult) //AddOrMult(int a, int b,
out int add, out int mult)
{
    add = a + b;
    mult = a * b;
}
```

```
AddOrMult(a, b, ref add, ref mult); //AddOrMult(a, b, out add, out mult);
Console.WriteLine(add); //5
Console.WriteLine(mult); //6
}

private static void AddOrMult(int a, int b, ref int add, ref int mult) //AddOrMult(int a, int b,
out int add, out int mult)
{
    add = a + b;
    mult = a * b;
}
```

belindoc.com

第125章：二进制序列化

第125.1节：使用属性控制序列化行为

如果你使用[NonSerialized]属性，那么该成员在反序列化后将始终保持其默认值（例如，int类型为0，string类型为null，bool类型为false等），无论对象本身进行了何种初始化（构造函数、声明等）。为此，提供了[OnDeserializing]（在反序列化之前调用）和[OnDeserialized]（在反序列化之后调用）这两个属性，以及它们的对应属性[OnSerializing]和[OnSerialized]。

假设我们想给我们的向量添加一个“评分”，并且我们想确保该值始终从1开始。下面的写法在反序列化后，值将变为0：

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal Rating = 1M;

    public Vector()
    {
        Rating = 1M;
    }

    public Vector(decimal initialRating)
    {
        Rating = initialRating;
    }
}
```

为了解决这个问题，我们可以简单地在类中添加以下方法，将其设置为1：

```
[OnDeserializing]
void OnDeserializing(StreamingContext context)
{
    Rating = 1M;
}
```

或者，如果我们想将其设置为计算值，可以等待反序列化完成后再设置它：

```
[OnDeserialized]
void OnDeserialized(StreamingContext context)
{
    Rating = 1 + ((X+Y+Z)/3);
}
```

同样，我们可以通过使用[OnSerializing]和[OnSerialized]来控制数据的写出方式。

第125.2节：序列化绑定器

绑定器让你有机会检查在应用程序域中加载的类型

Chapter 125: Binary Serialization

Section 125.1: Controlling serialization behavior with attributes

If you use the [NonSerialized] attribute, then that member will always have its default value after deserialization (ex. 0 for an `int`, null for `string`, false for a `bool`, etc.), regardless of any initialization done in the object itself (constructors, declarations, etc.). To compensate, the attributes [OnDeserializing] (called just BEFORE deserializing) and [OnDeserialized] (called just AFTER deserializing) together with their counterparts, [OnSerializing] and [OnSerialized] are provided.

Assume we want to add a "Rating" to our Vector and we want to make sure the value always starts at 1. The way it is written below, it will be 0 after being serialized:

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal Rating = 1M;

    public Vector()
    {
        Rating = 1M;
    }

    public Vector(decimal initialRating)
    {
        Rating = initialRating;
    }
}
```

To fix this problem, we can simply add the following method inside of the class to set it to 1:

```
[OnDeserializing]
void OnDeserializing(StreamingContext context)
{
    Rating = 1M;
}
```

Or, if we want to set it to a calculated value, we can wait for it to be finished deserializing and then set it:

```
[OnDeserialized]
void OnDeserialized(StreamingContext context)
{
    Rating = 1 + ((X+Y+Z)/3);
}
```

Similarly, we can control how things are written out by using [OnSerializing] and [OnSerialized].

Section 125.2: Serialization Binder

The binder gives you an opportunity to inspect what types are being loaded in your application domain

创建一个继承自 `SerializationBinder` 的类

```
class MyBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        if (typeName.Equals("BinarySerializationExample.Item"))
            return typeof(Item);
        return null;
    }
}
```

现在我们可以检查正在加载的类型，并据此决定我们真正想接收的内容

要使用绑定器，必须将其添加到 `BinaryFormatter` 中。

```
object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    binaryFormatter.Binder = new MyBinder();

    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

完整的解决方案

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerializationExample
{
    class MyBinder : SerializationBinder
    {
        public override Type BindToType(string assemblyName, string typeName)
        {
            if (typeName.Equals("BinarySerializationExample.Item"))
                return typeof(Item);
            return null;
        }
    }

    [Serializable]
    public class 项目
    {
        private string _名称;

        public string 名称
        {
            get { return _名称; }
            set { _名称 = value; }
        }
    }

    class 程序
    {
        static void Main(string[] args)
        {
    }
```

Create a class inherited from `SerializationBinder`

```
class MyBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        if (typeName.Equals("BinarySerializationExample.Item"))
            return typeof(Item);
        return null;
    }
}
```

Now we can check what types are loading and on this basis to decide what we really want to receive

For using a binder, you must add it to the `BinaryFormatter`.

```
object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    binaryFormatter.Binder = new MyBinder();

    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

The complete solution

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerializationExample
{
    class MyBinder : SerializationBinder
    {
        public override Type BindToType(string assemblyName, string typeName)
        {
            if (typeName.Equals("BinarySerializationExample.Item"))
                return typeof(Item);
            return null;
        }
    }

    [Serializable]
    public class Item
    {
        private string _name;

        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
    }
```

```

var 项目 = new 项目
{
    名称 = "Orange"
};

    var 字节 = SerializeData(项目);
    var 反序列化数据 = (项目)DeserializeData(字节);
}

private static byte[] SerializeData(object 对象)
{
    var 二进制格式化器 = new BinaryFormatter();
    using (var 内存流 = new MemoryStream())
    {
        二进制格式化器.Serialize(内存流, 对象);
        return 内存流.ToArray();
    }
}

private static object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    Binder = new MyBinder()
    {
        using (var memoryStream = new MemoryStream(bytes))
            return binaryFormatter.Deserialize(memoryStream);
    }
}

```

```

var item = new Item
{
    Name = "Orange"
};

var bytes = SerializeData(item);
var serializedData = (Item)DeserializeData(bytes);
}

private static byte[] SerializeData(object obj)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream())
    {
        binaryFormatter.Serialize(memoryStream, obj);
        return memoryStream.ToArray();
    }
}

private static object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    Binder = new MyBinder()
    {
        using (var memoryStream = new MemoryStream(bytes))
            return binaryFormatter.Deserialize(memoryStream);
    }
}

```

第125.3节：向后兼容性中的一些陷阱

这个小例子展示了如果你事先不注意，程序如何会失去向后兼容性。以及如何更好地控制序列化过程的方法

首先，我们将编写程序第一个版本的示例：

版本1

```

[Serializable]
class Data
{
    [OptionalField]
    private int _version;

    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }
}

```

现在，假设在程序的第二个版本中添加了一个新类。我们需要将它存储在一个数组中。

现在代码将如下所示：

Section 125.3: Some gotchas in backward compatibility

This small example shows how you can lose backward compatibility in your programs if you do not take care in advance about this. And ways to get more control of serialization process

At first, we will write an example of the first version of the program:

Version 1

```

[Serializable]
class Data
{
    [OptionalField]
    private int _version;

    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }
}

```

And now, let us assume that in the second version of the program added a new class. And we need to store it in an array.

Now code will look like this:

版本 2

```
[Serializable]
class NewItem
{
    [OptionalField]
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}

[Serializable]
class Data
{
    [OptionalField]
    private int _version;

    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }

    [OptionalField]
    private List<NewItem> _newItems;

    public List<NewItem> NewItems
    {
        get { return _newItems; }
        set { _newItems = value; }
    }
}
```

以及序列化和反序列化的代码

```
private static byte[] SerializeData(object 对象)
{
    var 二进制格式化器 = new BinaryFormatter();
    using (var 内存流 = new MemoryStream())
    {
        binaryFormatter.Serialize(memoryStream, obj);
        return memoryStream.ToArray();
    }
}

private static object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

那么，当你在v2版本的程序中序列化数据，然后尝试在v1版本的程序中反序列化这些数据时，会发生什么呢？

你会得到一个异常：

Version 2

```
[Serializable]
class NewItem
{
    [OptionalField]
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}

[Serializable]
class Data
{
    [OptionalField]
    private int _version;

    public int Version
    {
        get { return _version; }
        set { _version = value; }
    }

    [OptionalField]
    private List<NewItem> _newItems;

    public List<NewItem> NewItems
    {
        get { return _newItems; }
        set { _newItems = value; }
    }
}
```

And code for serialize and deserialize

```
private static byte[] SerializeData(object obj)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream())
    {
        binaryFormatter.Serialize(memoryStream, obj);
        return memoryStream.ToArray();
    }
}

private static object DeserializeData(byte[] bytes)
{
    var binaryFormatter = new BinaryFormatter();
    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
```

And so, what would happen when you serialize the data in the program of v2 and will try to deserialize them in the program of v1?

You get an exception:

System.Runtime.Serialization.SerializationException 未处理Message=ObjectManager
er发现了无效数量的修正。这通常表示Formatter中存在问题。Source=mscorlib

堆栈跟踪：

```
在 System.Runtime.Serialization.ObjectManager.DoFixups()
在 System.Runtime.Serialization.Formatters.Binary.ObjectReader.Deserialize(HeaderHandler handler,
__BinaryParser serParser, Boolean fCheck, Boolean isCrossAppDomain, IMethodCallMessage
methodCallMessage)
在 System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream
serializationStream, HeaderHandler handler, Boolean fCheck, Boolean isCrossAppDomain,
IMethodCallMessage methodCallMessage)
在 System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream
serializationStream)
在 Microsoft.Samples.TestV1.Main(String[] args) 中, 位于 c:\Users\andrew\Documents\Visual Studio
2013\Projects\vts\CS\V1 Application\TestV1Part2\TestV1Part2.cs 的第 29 行
在 System.AppDomain._nExecuteAssembly(Assembly assembly, String[] args)
在 Microsoft.VisualStudio.HostingProcess.HostProc.RunUsersAssembly()
在 System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback
callback, Object state)
在 System.Threading.ThreadHelper.ThreadStart()
```

为什么？

ObjectManager 对数组以及引用类型和值类型的依赖解析逻辑不同。我们添加了一个新的引用类型数组，而该类型在我们的程序集里不存在。

当 ObjectManager 尝试解析依赖时，它会构建依赖图。遇到数组时，无法立即解决，因此它创建了一个虚拟引用，随后再修正该数组。

由于该类型不在程序集内，依赖无法被修正。出于某种原因，它没有将该数组从需要修正的元素列表中移除，最终抛出异常“IncorrectNumberOfFixups”。

这是序列化过程中一些“陷阱”。出于某种原因，它仅对新引用类型的数组处理不正确。

注意：

如果不使用带有新类的数组，类似代码将能正常工作

修复并保持兼容性的第一种方法是什么？

- 使用一组新的结构体而不是类，或者使用字典（可能是类），因为字典是键值对的集合（它是结构体）
- 如果无法更改旧代码，请使用 ISerializable

第125.4节：使对象可序列化

添加 [Serializable] 属性以标记整个对象进行二进制序列化：

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal DontSerializeThis;
```

```
System.Runtime.Serialization.SerializationException was unhandled
Message=The ObjectManager found an invalid number of fixups. This usually indicates a problem in
the Formatter.Source=mscorlib
StackTrace:
at System.Runtime.Serialization.ObjectManager.DoFixups()
at System.Runtime.Serialization.Formatters.Binary.ObjectReader.Deserialize(HeaderHandler handler,
__BinaryParser serParser, Boolean fCheck, Boolean isCrossAppDomain, IMethodCallMessage
methodCallMessage)
at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream
serializationStream, HeaderHandler handler, Boolean fCheck, Boolean isCrossAppDomain,
IMethodCallMessage methodCallMessage)
at System.Runtime.Serialization.Formatters.Binary.BinaryFormatter.Deserialize(Stream
serializationStream)
at Microsoft.Samples.TestV1.Main(String[] args) in c:\Users\andrew\Documents\Visual Studio
2013\Projects\vts\CS\V1 Application\TestV1Part2\TestV1Part2.cs:line 29
at System.AppDomain._nExecuteAssembly(Assembly assembly, String[] args)
at Microsoft.VisualStudio.HostingProcess.HostProc.RunUsersAssembly()
at System.Threading.ExecutionContext.Run(ExecutionContext executionContext, ContextCallback
callback, Object state)
at System.Threading.ThreadHelper.ThreadStart()
```

Why?

The ObjectManager has a different logic to resolve dependencies for arrays and for reference and value types. We added an array of new the reference type which is absent in our assembly.

When ObjectManager attempts to resolve dependencies it builds the graph. When it sees the array, it can not fix it immediately, so that it creates a dummy reference and then fixes the array later.

And since this type is not in the assembly and dependencies can't be fixed. For some reason, it does not remove the array from the list of elements for the fixes and at the end, it throws an exception “IncorrectNumberOfFixups”.

It is some ‘gotchas’ in the process of serialization. For some reason, it does not work correctly only for arrays of new reference types.

A Note:

Similar code will work correctly if you do not use arrays with new classes

And the first way to fix it and maintain compatibility?

- Use a collection of new structures rather than classes or use a dictionary(possible classes), because a dictionary it's a collection of keyvaluepair(it's structure)
- Use ISerializable, if you can't change the old code

Section 125.4: Making an object serializable

Add the [Serializable] attribute to mark an entire object for binary serialization:

```
[Serializable]
public class Vector
{
    public int X;
    public int Y;
    public int Z;

    [NonSerialized]
    public decimal DontSerializeThis;
```

```

[OptionalField]
public string Name;
}

```

除非我们显式使用 [NonSerialized] 属性选择不序列化，否则所有成员都会被序列化。在我们的示例中，X、Y、Z 和 Name 都会被序列化。

除非标记为 [NonSerialized] 或 [OptionalField]，否则所有成员在反序列化时都必须存在。在我们的示例中，X、Y 和 Z 都是必需的，如果它们在数据流中不存在，反序列化将失败。DontSerializeThis 始终会被设置为 decimal 类型的默认值（即 0）。如果 Name 存在于数据流中，则会被设置为该值，否则会被设置为 string 类型的默认值（即 null）。其目的在于 [OptionalField] 是为了提供一定的版本容错性。

第125.5节：序列化代理（实现 ISerializationSurrogate）

实现一个序列化代理选择器，允许一个对象执行另一个对象的序列化和反序列化

同时允许正确序列化或反序列化一个本身不可序列化的类

实现ISerializationSurrogate接口

```

public class ItemSurrogate : ISerializationSurrogate
{
    public void GetObjectData(object obj, SerializationInfo info, StreamingContext context)
    {
        var item = (Item)obj;
        info.AddValue("_name", item.Name);
    }

    public object SetObjectData(object obj, SerializationInfo info, StreamingContext context,
ISurrogateSelector selector)
    {
        var item = (Item)obj;
        item.Name = (string)info.GetValue("_name", typeof(string));
        return item;
    }
}

```

然后你需要通过定义和初始化一个SurrogateSelector来让你的IFormatter知道这些代理，并将其分配给你的IFormatter

```

var surrogateSelector = new SurrogateSelector();
surrogateSelector.AddSurrogate(typeof(Item), new StreamingContext(StreamingContextStates.All), new
ItemSurrogate());
var binaryFormatter = new BinaryFormatter
{
    SurrogateSelector = surrogateSelector
};

```

即使该类未标记为可序列化。

```

//该类不可序列化
public class Item
{
    private string _name;
}

```

```

[OptionalField]
public string Name;
}

```

All members will be serialized unless we explicitly opt-out using the [NonSerialized] attribute. In our example, X, Y, Z, and Name are all serialized.

All members are required to be present on deserialization unless marked with [NonSerialized] or [OptionalField]. In our example, X, Y, and Z are all required and deserialization will fail if they are not present in the stream. DontSerializeThis will always be set to `default(decimal)` (which is 0). If Name is present in the stream, then it will be set to that value, otherwise it will be set to `default(string)` (which is null). The purpose of [OptionalField] is to provide a bit of version tolerance.

Section 125.5: Serialization surrogates (Implementing ISerializationSurrogate)

Implements a serialization surrogate selector that allows one object to perform serialization and deserialization of another

As well allows to properly serialize or deserialize a class that is not itself serializable

Implement ISerializationSurrogate interface

```

public class ItemSurrogate : ISerializationSurrogate
{
    public void GetObjectData(object obj, SerializationInfo info, StreamingContext context)
    {
        var item = (Item)obj;
        info.AddValue("_name", item.Name);
    }

    public object SetObjectData(object obj, SerializationInfo info, StreamingContext context,
ISurrogateSelector selector)
    {
        var item = (Item)obj;
        item.Name = (string)info.GetValue("_name", typeof(string));
        return item;
    }
}

```

Then you need to let your IFormatter know about the surrogates by defining and initializing a SurrogateSelector and assigning it to your IFormatter

```

var surrogateSelector = new SurrogateSelector();
surrogateSelector.AddSurrogate(typeof(Item), new StreamingContext(StreamingContextStates.All), new
ItemSurrogate());
var binaryFormatter = new BinaryFormatter
{
    SurrogateSelector = surrogateSelector
};

```

Even if the class is not marked serializable.

```

//this class is not serializable
public class Item
{
    private string _name;
}

```

```

public string Name
{
    get { return _名称; }
    set { _名称 = value; }
}

```

完整的解决方案

```

using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerializationExample
{
    class Item
    {
        private string _名称;

        public string 名称
        {
            get { return _名称; }
            set { _名称 = value; }
        }
    }

    class ItemSurrogate : ISerializationSurrogate
    {
        public void GetObjectData(object obj, SerializationInfo info, StreamingContext context)
        {
            var item = (Item)obj;
            info.AddValue("_name", item.Name);
        }

        public object SetObjectData(object obj, SerializationInfo info, StreamingContext context,
ISurrogateSelector selector)
        {
            var item = (Item)obj;
            item.Name = (string)info.GetValue("_name", typeof(string));
            return item;
        }
    }

    class 程序
    {
        static void Main(string[] args)
        {
            var 项目 = new 项目
            {
                名称 = "Orange"
            };

            var bytes = SerializeData(item);
            var deserializedData = (Item)DeserializeData(bytes);
        }

        private static byte[] SerializeData(object 对象)
        {
            var surrogateSelector = new SurrogateSelector();
            surrogateSelector.AddSurrogate(typeof(Item), new

```

```

public string Name
{
    get { return _name; }
    set { _name = value; }
}

```

The complete solution

```

using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinarySerializationExample
{
    class Item
    {
        private string _name;

        public string Name
        {
            get { return _name; }
            set { _name = value; }
        }
    }

    class ItemSurrogate : ISerializationSurrogate
    {
        public void GetObjectData(object obj, SerializationInfo info, StreamingContext context)
        {
            var item = (Item)obj;
            info.AddValue("_name", item.Name);
        }

        public object SetObjectData(object obj, SerializationInfo info, StreamingContext context,
ISurrogateSelector selector)
        {
            var item = (Item)obj;
            item.Name = (string)info.GetValue("_name", typeof(string));
            return item;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var item = new Item
            {
                Name = "Orange"
            };

            var bytes = SerializeData(item);
            var deserializedData = (Item)DeserializeData(bytes);
        }

        private static byte[] SerializeData(object obj)
        {
            var surrogateSelector = new SurrogateSelector();
            surrogateSelector.AddSurrogate(typeof(Item), new

```

```

StreamingContext(StreamingContextStates.All), new ItemSurrogate());

    var binaryFormatter = new BinaryFormatter
    {
        SurrogateSelector = surrogateSelector
    };

    using (var memoryStream = new MemoryStream())
    {
        二进制格式化器.Serialize(内存流, 对象);
        return 内存流.ToArray();
    }
}

private static object DeserializeData(byte[] bytes)
{
    var surrogateSelector = new SurrogateSelector();
    surrogateSelector.AddSurrogate(typeof(Item), new
StreamingContext(StreamingContextStates.All), new ItemSurrogate());

    var binaryFormatter = new BinaryFormatter
    {
        SurrogateSelector = surrogateSelector
    };

    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
}

```

第125.6节：通过实现

ISerializable来增加更多控制

这将获得对序列化的更多控制，如何保存和加载类型

实现ISerializable接口并创建一个空构造函数以便编译

```

[Serializable]
public class Item : ISerializable
{
    private string _名称;

    public string 名称
    {
        get { return _名称; }
        set { _名称 = value; }
    }

    public Item ()
    {

    }

    protected Item (SerializationInfo info, StreamingContext context)
    {
        _name = (string)info.GetValue("_name", typeof(string));
    }

    public void GetObjectData(SerializationInfo info, StreamingContext context)

```

```

StreamingContext(StreamingContextStates.All), new ItemSurrogate());

    var binaryFormatter = new BinaryFormatter
    {
        SurrogateSelector = surrogateSelector
    };

    using (var memoryStream = new MemoryStream())
    {
        binaryFormatter.Serialize(memoryStream, obj);
        return memoryStream.ToArray();
    }
}

private static object DeserializeData(byte[] bytes)
{
    var surrogateSelector = new SurrogateSelector();
    surrogateSelector.AddSurrogate(typeof(Item), new
StreamingContext(StreamingContextStates.All), new ItemSurrogate());

    var binaryFormatter = new BinaryFormatter
    {
        SurrogateSelector = surrogateSelector
    };

    using (var memoryStream = new MemoryStream(bytes))
        return binaryFormatter.Deserialize(memoryStream);
}
}

```

Section 125.6: Adding more control by implementing ISerializable

That would get more control over serialization, how to save and load types

Implement ISerializable interface and create an empty constructor to compile

```

[Serializable]
public class Item : ISerializable
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public Item ()
    {

    }

    protected Item (SerializationInfo info, StreamingContext context)
    {
        _name = (string)info.GetValue("_name", typeof(string));
    }

    public void GetObjectData(SerializationInfo info, StreamingContext context)

```

```
{  
    info.AddValue("_name", _name, typeof(string));  
}
```

对于数据序列化，您可以指定所需的名字和所需的类型

```
info.AddValue("_name", _name, typeof(string));
```

当数据被反序列化时，您将能够读取所需的类型

```
_name = (string)info.GetValue("_name", typeof(string));
```

```
{  
    info.AddValue("_name", _name, typeof(string));  
}
```

For data serialization, you can specify the desired name and the desired type

```
info.AddValue("_name", _name, typeof(string));
```

When the data is deserialized, you will be able to read the desired type

```
_name = (string)info.GetValue("_name", typeof(string));
```

belindoc.com

第126章 : ICloneable

第126.1节 : 在类中实现ICloneable

在类中实现ICloneable，但有些变化。公开一个类型安全的Clone()方法，并私有实现object Clone()。

```
public class Person : ICloneable
{
    // 类的内容
    public string Name { get; set; }
    public int Age { get; set; }

    // 构造函数
    public Person(string name, int age)
    {
        this.Name=name;
        this.Age=age;
    }

    // 复制构造函数
    public Person(Person other)
    {
        this.Name=other.Name;
        this.Age=other.Age;
    }

    #region ICloneable 成员
    // 类型安全的克隆
    public Person Clone() { return new Person(this); }
    // ICloneable 实现
    object ICloneable.Clone()
    {
        return Clone();
    }
    #endregion
}
```

后续用法如下：

```
{
Person bob=new Person("Bob", 25);
Person bob_clone=bob.Clone();
Debug.Assert(bob_clone.Name==bob.Name);

bob.Age=56;
Debug.Assert(bob.Age!=bob.Age);
}
```

注意，改变bob的年龄不会改变bob_clone的年龄。这是因为设计使用了克隆而不是赋值（引用）变量。

第126.2节 : 在结构体中实现ICloneable接口

结构体通常不需要实现ICloneable接口，因为结构体使用赋值运算符=进行成员逐个复制。但设计可能要求实现继承自

ICloneable的另一个接口。

另一个原因是如果结构体包含引用类型（或数组），也需要进行复制。

Chapter 126: ICloneable

Section 126.1: Implementing ICloneable in a class

Implement ICloneable in a class with a twist. Expose a public type safe Clone() and implement `object` Clone() privately.

```
public class Person : ICloneable
{
    // Contents of class
    public string Name { get; set; }
    public int Age { get; set; }

    // Constructor
    public Person(string name, int age)
    {
        this.Name=name;
        this.Age=age;
    }

    // Copy Constructor
    public Person(Person other)
    {
        this.Name=other.Name;
        this.Age=other.Age;
    }

    #region ICloneable Members
    // Type safe Clone
    public Person Clone() { return new Person(this); }
    // ICloneable implementation
    object ICloneable.Clone()
    {
        return Clone();
    }
    #endregion
}
```

Later to be used as follows:

```
{
Person bob=new Person("Bob", 25);
Person bob_clone=bob.Clone();
Debug.Assert(bob_clone.Name==bob.Name);

bob.Age=56;
Debug.Assert(bob.Age!=bob.Age);
}
```

Notice that changing the age of bob does not change the age of bob_clone. This is because the design uses cloning instead of assigning of (reference) variables.

Section 126.2: Implementing ICloneable in a struct

The implementation of ICloneable for a struct is not generally needed because structs do a memberwise copy with the assignment operator =. But the design might require the implementation of another interface that inherits from ICloneable.

Another reason would be if the struct contains a reference type (or an array) which would need copying also.

```

// 建议结构体为不可变对象
[ImmutableObject(true)]
public struct Person : ICloneable
{
    // 类的内容
    public string Name { get; private set; }
    public int Age { get; private set; }
    // 构造函数
    public Person(string name, int age)
    {
        this.Name=name;
        this.Age=age;
    }
    // 复制构造函数
    public Person(Person other)
    {
        // 赋值运算符会复制所有成员
        this=other;
    }

#region ICloneable 成员
// 类型安全的克隆
public Person Clone() { return new Person(this); }
// ICloneable 实现
object ICloneable.Clone()
{
    return Clone();
}
#endregion
}

```

后续用法如下：

```

static void Main(string[] args)
{
Person bob=new Person("Bob", 25);
    Person bob_clone=bob.Clone();
    Debug.Assert(bob_clone.Name==bob.Name);
}

```

```

// Structs are recommended to be immutable objects
[ImmutableObject(true)]
public struct Person : ICloneable
{
    // Contents of class
    public string Name { get; private set; }
    public int Age { get; private set; }
    // Constructor
    public Person(string name, int age)
    {
        this.Name=name;
        this.Age=age;
    }
    // Copy Constructor
    public Person(Person other)
    {
        // The assignment operator copies all members
        this=other;
    }

#region ICloneable Members
// Type safe Clone
public Person Clone() { return new Person(this); }
// ICloneable implementation
object ICloneable.Clone()
{
    return Clone();
}
#endregion
}

```

Later to be used as follows:

```

static void Main(string[] args)
{
    Person bob=new Person("Bob", 25);
    Person bob_clone=bob.Clone();
    Debug.Assert(bob_clone.Name==bob.Name);
}

```

第127章 : IComparable

第127.1节 : 排序版本

类：

```
public class Version : IComparable<Version>
{
    public int[] Parts { get; }

    public Version(string value)
    {
        if (value == null)
            throw new ArgumentNullException();
        if (!Regex.IsMatch(value, @"^[\d]+(\.[\d]+)*$"))
            throw new ArgumentException("格式无效");
        var parts = value.Split('.');
        Parts = new int[parts.Length];
        for (var i = 0; i < parts.Length; i++)
            Parts[i] = int.Parse(parts[i]);
    }

    public override string ToString()
    {
        return string.Join(".", Parts);
    }

    public int CompareTo(Version that)
    {
        if (that == null) return 1;
        var thisLength = this.Parts.Length;
        var thatLength = that.Parts.Length;
        var maxLength = Math.Max(thisLength, thatLength);
        for (var i = 0; i < maxLength; i++)
        {
            var thisPart = i < thisLength ? this.Parts[i] : 0;
            var thatPart = i < thatLength ? that.Parts[i] : 0;
            if (thisPart < thatPart) return -1;
            if (thisPart > thatPart) return 1;
        }
        return 0;
    }
}
```

测试：

```
版本 a, b;

a = new Version("4.2.1");
b = new Version("4.2.6");
a.CompareTo(b); // a < b : -1

a = new Version("2.8.4");
b = new Version("2.8.0");
a.CompareTo(b); // a > b : 1

a = new Version("5.2");
b = null;
a.CompareTo(b); // a > b : 1
```

Chapter 127: IComparable

Section 127.1: Sort versions

Class:

```
public class Version : IComparable<Version>
{
    public int[] Parts { get; }

    public Version(string value)
    {
        if (value == null)
            throw new ArgumentNullException();
        if (!Regex.IsMatch(value, @"^[\d]+(\.[\d]+)*$"))
            throw new ArgumentException("Invalid format");
        var parts = value.Split('.');
        Parts = new int[parts.Length];
        for (var i = 0; i < parts.Length; i++)
            Parts[i] = int.Parse(parts[i]);
    }

    public override string ToString()
    {
        return string.Join(".", Parts);
    }

    public int CompareTo(Version that)
    {
        if (that == null) return 1;
        var thisLength = this.Parts.Length;
        var thatLength = that.Parts.Length;
        var maxLength = Math.Max(thisLength, thatLength);
        for (var i = 0; i < maxLength; i++)
        {
            var thisPart = i < thisLength ? this.Parts[i] : 0;
            var thatPart = i < thatLength ? that.Parts[i] : 0;
            if (thisPart < thatPart) return -1;
            if (thisPart > thatPart) return 1;
        }
        return 0;
    }
}
```

Test:

```
Version a, b;

a = new Version("4.2.1");
b = new Version("4.2.6");
a.CompareTo(b); // a < b : -1

a = new Version("2.8.4");
b = new Version("2.8.0");
a.CompareTo(b); // a > b : 1

a = new Version("5.2");
b = null;
a.CompareTo(b); // a > b : 1
```

```

a = new Version("3");
b = new Version("3.6");
a.CompareTo(b); // a < b : -1

var versions = new List<Version>
{
    new Version("2.0"),
    new Version("1.1.5"),
    new Version("3.0.10"),
    new Version("1"),
    null,
    new Version("1.0.1")
};

versions.Sort();

foreach (var version in versions)
    Console.WriteLine(version?.ToString() ?? "NULL");

```

输出：

```

NULL
1
1.0.1
1.1.5
2.0
3.0.10

```

演示：

[Ideone 在线演示](#)

```

a = new Version("3");
b = new Version("3.6");
a.CompareTo(b); // a < b : -1

var versions = new List<Version>
{
    new Version("2.0"),
    new Version("1.1.5"),
    new Version("3.0.10"),
    new Version("1"),
    null,
    new Version("1.0.1")
};

versions.Sort();

foreach (var version in versions)
    Console.WriteLine(version?.ToString() ?? "NULL");

```

Output:

```

NULL
1
1.0.1
1.1.5
2.0
3.0.10

```

Demo:

[Live demo on Ideone](#)

第128章：访问数据库

第128.1节：连接字符串

连接字符串是一种字符串，用于指定有关特定数据源的信息以及如何通过存储凭据、位置和其他信息来连接它。

```
Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;
```

存储您的连接字符串

通常，连接字符串会存储在配置文件中（例如ASP.NET应用程序中的app.config或web.config）。以下是这些文件中本地连接字符串的示例：

```
<connectionStrings>
  <add name="WidgetsContext" providerName="System.Data.SqlClient"
    connectionString="Server=.\SQLEXPRESS;Database=Widgets;Integrated Security=True;" />
</connectionStrings>

<connectionStrings>
  <add name="WidgetsContext" providerName="System.Data.SqlClient"
    connectionString="Server=.\SQLEXPRESS;Database=Widgets;Integrated Security=SSPI;" />
</connectionStrings>
```

这将允许您的应用程序通过WidgetsContext以编程方式访问连接字符串。

虽然Integrated Security=SSPI和Integrated Security=True执行相同的功能；Integrated Security=SSPI更受推荐，因为它既适用于SQLClient也适用于OleDb提供程序，而Integrated Security=true在与OleDb提供程序一起使用时会抛出异常。

不同提供程序的不同连接

每个数据提供程序（SQL Server、MySQL、Azure等）都有其特定的连接字符串语法，并且暴露不同的可用属性。如果您不确定自己的连接字符串应该是什么样子，[ConnectionString.com](#)是一个非常有用的资源。

第128.2节：实体框架连接

实体框架（Entity Framework）提供了用于与底层数据库交互的抽象类，这些类以DbContext等类的形式存在。这些上下文通常包含DbSet<T>属性，用于暴露可查询的集合：

```
public class ExampleContext: DbContext
{
    public virtual DbSet<Widgets> Widgets { get; set; }
}
```

该DbContext本身将负责与数据库建立连接，通常会从配置中读取相应的连接字符串数据，以确定如何建立连接：

```
public class ExampleContext: DbContext
{
    // 传递给基类构造函数的参数表示连接字符串的名称
}
```

Chapter 128: Accessing Databases

Section 128.1: Connection Strings

A Connection String is a string that specifies information about a particular data source and how to go about connecting to it by storing credentials, locations, and other information.

```
Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;
```

Storing Your Connection String

Typically, a connection string will be stored within a configuration file (such as an app.config or web.config within ASP.NET applications). The following is an example of what a local connection might look like within one of these files :

```
<connectionStrings>
  <add name="WidgetsContext" providerName="System.Data.SqlClient"
    connectionString="Server=.\SQLEXPRESS;Database=Widgets;Integrated Security=True;" />
</connectionStrings>

<connectionStrings>
  <add name="WidgetsContext" providerName="System.Data.SqlClient"
    connectionString="Server=.\SQLEXPRESS;Database=Widgets;Integrated Security=SSPI;" />
</connectionStrings>
```

This will allow your application to access the connection string programmatically through WidgetsContext. Although both Integrated Security=SSPI and Integrated Security=True perform the same function; Integrated Security=SSPI is preferred since works with both SQLClient & OleDb provider where as Integrated Security=true throws an exception when used with the OleDb provider.

Different Connections for Different Providers

Each data provider (SQL Server, MySQL, Azure, etc.) all feature their own flavor of syntax for their connection strings and expose different available properties. [ConnectionString.com](#) is an incredibly useful resource if you are unsure about what yours should look like.

Section 128.2: Entity Framework Connections

Entity Framework exposes abstraction classes that are used to interact with underlying databases in the form of classes like DbContext. These contexts generally consist of DbSet<T> properties that expose the available collections that can be queried :

```
public class ExampleContext: DbContext
{
    public virtual DbSet<Widgets> Widgets { get; set; }
}
```

The DbContext itself will handle making the connections with the databases and will generally read the appropriate Connection String data from a configuration to determine how to establish the connections :

```
public class ExampleContext: DbContext
{
    // The parameter being passed in to the base constructor indicates the name of the
    // connection string
```

```

public ExampleContext() : base("ExampleContextEntities")
{
}

public virtual DbSet<Widgets> Widgets { get; set; }
}

```

执行实体框架查询

实际执行实体框架查询非常简单，只需创建上下文的实例，然后使用其可用属性来提取或访问数据

```

using(var context = new ExampleContext())
{
    // 获取数据库中所有的控件
    var data = context.Widgets.ToList();
}

```

Entity Framework 还提供了一个广泛的变更跟踪系统，可以通过简单调用SaveChanges()方法将更改推送到数据库，从而处理数据库中条目的更新：

```

using(var context = new ExampleContext())
{
    // 获取你想要更新的控件
    var widget = context.Widgets.Find(w => w.Id == id);
    // 如果存在，则更新它
    if(widget != null)
    {
        // 更新你的控件并保存更改
        widget.Updated = DateTime.UtcNow;
        context.SaveChanges();
    }
}

```

第128.3节：ADO.NET 连接

ADO.NET 连接是从 C# 应用程序连接数据库的最简单方式之一。它们依赖于提供程序和指向数据库的连接字符串来执行查询。

常用数据提供程序类

以下许多类是常用来查询数据库及其相关命名空间的类：

- SqlConnection,SqlCommand,SqlDataReader 来自 System.Data.SqlClient
- OleDbConnection,OleDbCommand,OleDbDataReader 来自 System.Data.OleDb
- MySqlConnection, MySqlCommand, MySqlDbDataReader 来自 [MySql.Data](#)

所有这些类都是通过C#访问数据时常用的，并且在构建以数据为中心的应用程序时会经常遇到。还有许多未提及的实现相同功能的类

FooConnection,FooCommand,FooDataReader 类也可以预期具有相同的行为。

ADO.NET连接的常见访问模式

通过ADO.NET连接访问数据时常用的模式可能如下所示：

```

public ExampleContext() : base("ExampleContextEntities")
{
}

public virtual DbSet<Widgets> Widgets { get; set; }
}

```

Executing Entity Framework Queries

Actually executing an Entity Framework query can be quite easy and simply requires you to create an instance of the context and then use the available properties on it to pull or access your data

```

using(var context = new ExampleContext())
{
    // Retrieve all of the Widgets in your database
    var data = context.Widgets.ToList();
}

```

Entity Framework also provides an extensive change-tracking system that can be used to handle updating entries within your database by simply calling the SaveChanges() method to push changes to the database :

```

using(var context = new ExampleContext())
{
    // Grab the widget you wish to update
    var widget = context.Widgets.Find(w => w.Id == id);
    // If it exists, update it
    if(widget != null)
    {
        // Update your widget and save your changes
        widget.Updated = DateTime.UtcNow;
        context.SaveChanges();
    }
}

```

Section 128.3: ADO.NET Connections

ADO.NET Connections are one of the simplest ways to connect to a database from a C# application. They rely on the use of a provider and a connection string that points to your database to perform queries against.

Common Data Provider Classes

Many of the following are classes that are commonly used to query databases and their related namespaces :

- SqlConnection,SqlCommand,SqlDataReader from System.Data.SqlClient
- OleDbConnection,OleDbCommand,OleDbDataReader from System.Data.OleDb
- MySqlConnection, MySqlCommand, MySqlDbDataReader from [MySql.Data](#)

All of these are commonly used to access data through C# and will be commonly encountered throughout building data-centric applications. Many other classes that are not mentioned that implement the same FooConnection,FooCommand,FooDataReader classes can be expected to behave the same way.

Common Access Pattern for ADO.NET Connections

A common pattern that can be used when accessing your data through an ADO.NET connection might look as follows :

```

// 这限定了连接的作用域 (具体类可能有所不同)
using(var connection = new SqlConnection("{your-connection-string}"))
{
    // 构建您的查询
    var query = "SELECT * FROM YourTable WHERE Property = @property";
    // 限定命令的执行范围
    using(var command = new SqlCommand(query, connection))
    {
        // 打开您的连接
        connection.Open();

        // 如有必要, 在此添加参数

        // 使用读取器执行查询 (同样使用 using 语句限定范围)
        using(var reader = command.ExecuteReader())
        {
            // 在此遍历您的结果
        }
    }
}

```

或者, 如果您只是执行一个简单的更新且不需要读取器, 基本概念同样适用 :

```

using(var connection = new SqlConnection("{your-connection-string}"))
{
    var query = "UPDATE YourTable SET Property = Value WHERE Foo = @foo";
    using(var command = new SqlCommand(query, connection))
    {
        connection.Open();

        // 在此添加参数

        // 执行更新操作
        command.ExecuteNonQuery();
    }
}

```

你甚至可以针对一组通用接口进行编程, 而无需担心特定提供程序的类。ADO.NET 提供的核心接口有 :

- IDbConnection - 用于管理数据库连接
- IDbCommand - 用于执行 SQL 命令
- IDbTransaction - 用于管理事务
- IDataReader - 用于读取命令返回的数据
- IDataAdapter - 用于在数据集之间传输数据

```

var connectionString = "{your-connection-string}";
var providerName = "{System.Data.SqlClient}"; // Oracle 使用 "Oracle.ManagedDataAccess.Client"
// 大多数情况下你会从 ConnectionStringSettings 对象获取以上两个值

var factory = DbProviderFactories.GetFactory(providerName);

using(var connection = new factory.CreateConnection()) {
    connection.ConnectionString = connectionString;
    connection.Open();

    using(var command = new connection.CreateCommand()) {
        command.CommandText = "{sql-query}"; // 这需要针对每个数据库系统进行调整

        using(var reader = command.ExecuteReader()) {

```

```

// This scopes the connection (your specific class may vary)
using(var connection = new SqlConnection("{your-connection-string}"))
{
    // Build your query
    var query = "SELECT * FROM YourTable WHERE Property = @property";
    // Scope your command to execute
    using(var command = new SqlCommand(query, connection))
    {
        // Open your connection
        connection.Open();

        // Add your parameters here if necessary

        // Execute your query as a reader (again scoped with a using statement)
        using(var reader = command.ExecuteReader())
        {
            // Iterate through your results here
        }
    }
}

```

Or if you were just performing a simple update and didn't require a reader, the same basic concept would apply :

```

using(var connection = new SqlConnection("{your-connection-string}"))
{
    var query = "UPDATE YourTable SET Property = Value WHERE Foo = @foo";
    using(var command = new SqlCommand(query, connection))
    {
        connection.Open();

        // Add parameters here

        // Perform your update
        command.ExecuteNonQuery();
    }
}

```

You can even program against a set of common interfaces and not have to worry about the provider specific classes. The core interfaces provided by ADO.NET are:

- IDbConnection - for managing database connections
- IDbCommand - for running SQL commands
- IDbTransaction - for managing transactions
- IDataReader - for reading data returned by a command
- IDataAdapter - for channeling data to and from datasets

```

var connectionString = "{your-connection-string}";
var providerName = "{System.Data.SqlClient}"; // for Oracle use "Oracle.ManagedDataAccess.Client"
// most likely you will get the above two from ConnectionStringSettings object

var factory = DbProviderFactories.GetFactory(providerName);

using(var connection = new factory.CreateConnection()) {
    connection.ConnectionString = connectionString;
    connection.Open();

    using(var command = new connection.CreateCommand()) {
        command.CommandText = "{sql-query}"; // this needs to be tailored for each database system

        using(var reader = command.ExecuteReader()) {

```

```
    while(reader.Read()) {  
        ...  
    }  
}
```

```
    while(reader.Read()) {  
        ...  
    }  
}
```

belindoc.com

第129章：在C#中使用SQLite

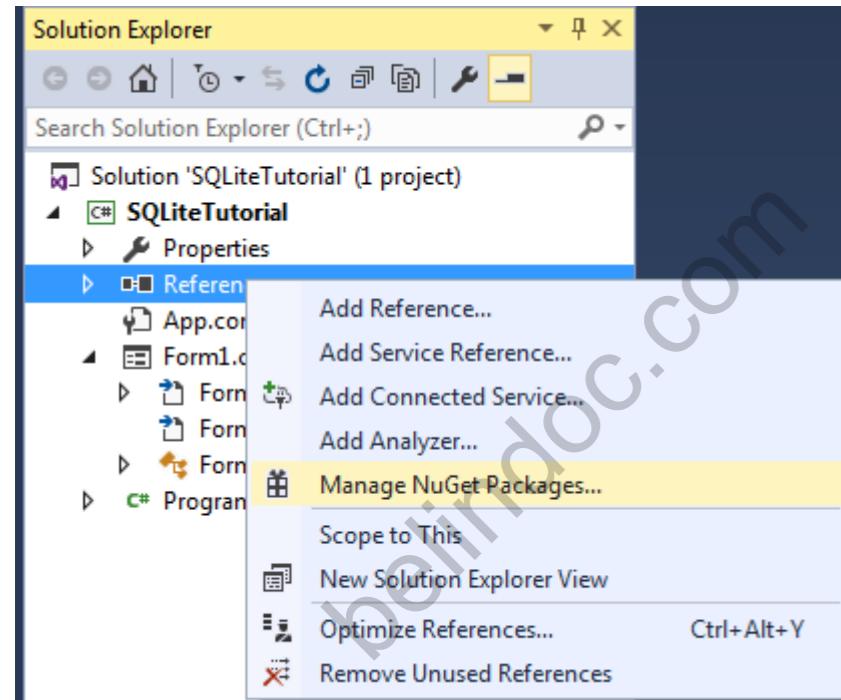
第129.1节：在C#中使用SQLite创建简单的CRUD

首先，我们需要为应用程序添加SQLite支持。有两种方法可以实现

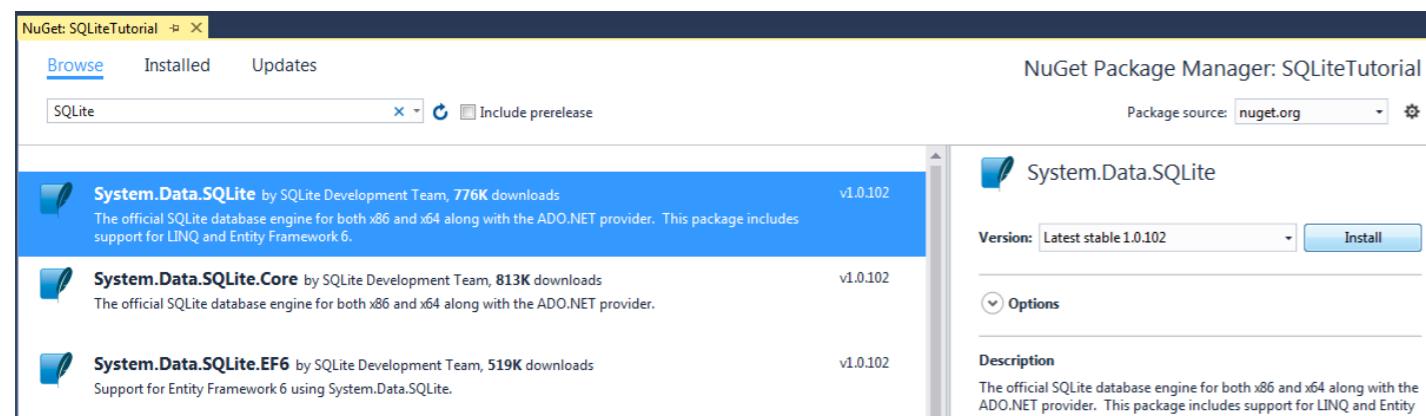
- 从[SQLite下载页面](#)下载适合您系统的DLL，然后手动添加到项目中
- 通过NuGet添加SQLite依赖

我们将采用第二种方法

首先打开NuGet菜单



搜索System.Data.SQLite，选择它并点击安装



安装也可以通过包管理控制台完成

```
PM> Install-Package System.Data.SQLite
```

或者仅安装核心功能

```
PM> Install-Package System.Data.SQLite.Core
```

Chapter 129: Using SQLite in C#

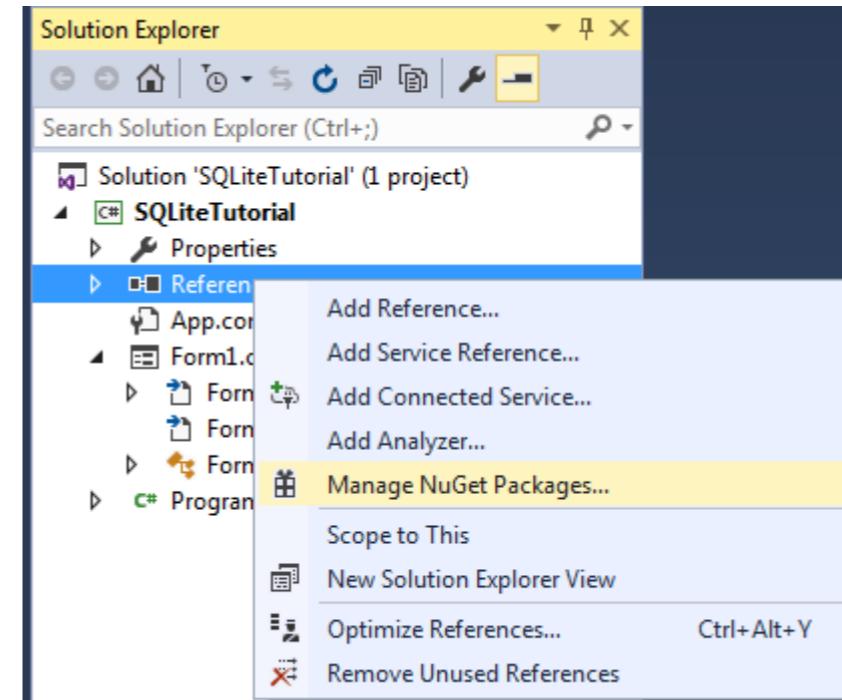
Section 129.1: Creating simple CRUD using SQLite in C#

First of all we need to add SQLite support to our application. There are two ways of doing that

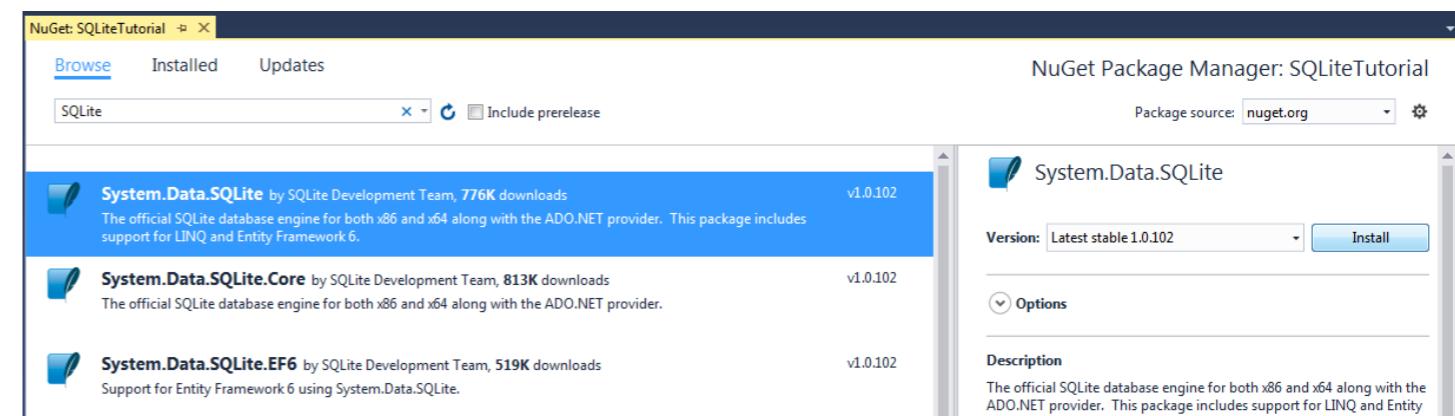
- Download DLL suiting your system from [SQLite download page](#) and then add to the project manually
- Add SQLite dependency via NuGet

We'll do it the second way

First open the NuGet menu



and search for **System.Data.SQLite**, select it and hit **Install**



Installation can also be done from [Package Manager Console](#) with

```
PM> Install-Package System.Data.SQLite
```

Or for only core features

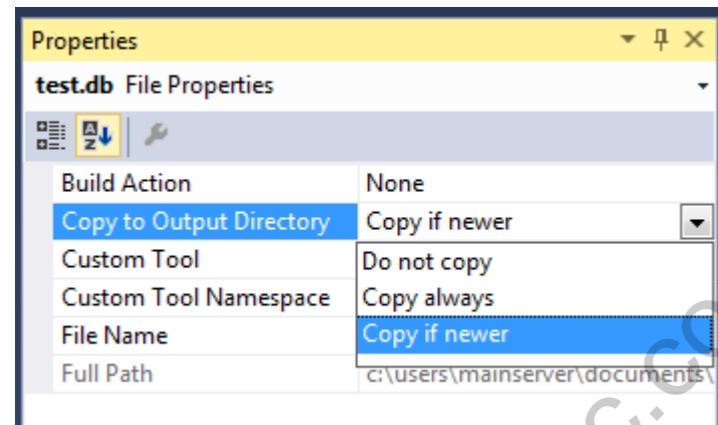
```
PM> Install-Package System.Data.SQLite.Core
```

下载就到这里，我们可以直接开始编码了。

首先创建一个简单的SQLite数据库，包含此表，并将其作为文件添加到项目中

```
CREATE TABLE User(
    Id INTEGER PRIMARY KEY AUTOINCREMENT,
    FirstName TEXT NOT NULL,
    LastName TEXT NOT NULL
);
```

还要记得根据需要将文件的复制到输出目录属性设置为如果较新则复制或始终复制



创建一个名为 User 的类，它将作为我们数据库的基础实体

```
private class User
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

我们将编写两个用于查询执行的方法，第一个用于插入、更新或删除数据库中的数据

```
private int ExecuteWrite(string query, Dictionary<string, object> args)
{
    int numberOfRowsAffected;

    //设置与数据库的连接
    using (var con = new SQLiteConnection("Data Source=test.db"))
    {
        con.Open();

        //打开一个新命令
        using (var cmd = new SQLiteCommand(query, con))
        {
            //设置查询中给定的参数
            foreach (var pair in args)
            {
                cmd.Parameters.AddWithValue(pair.Key, pair.Value);
            }

            //执行查询并获取受影响的行数
            numberOfRowsAffected = cmd.ExecuteNonQuery();
        }
    }

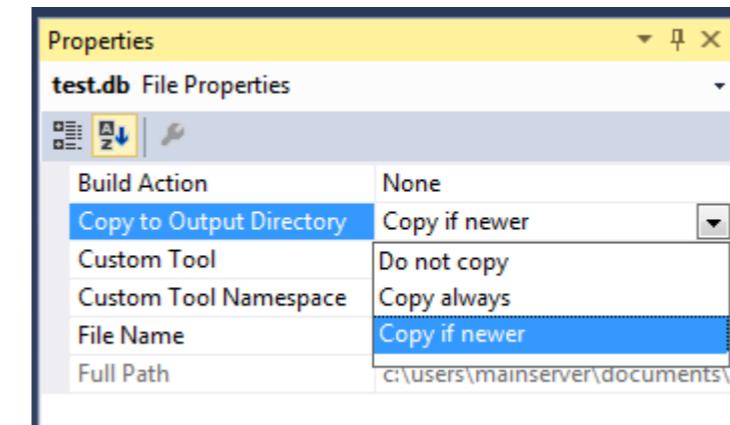
    return numberOfRowsAffected;
}
```

That's it for the download, so we can go right into coding.

First create a simple SQLite database with this table and add it as a file to the project

```
CREATE TABLE User(
    Id INTEGER PRIMARY KEY AUTOINCREMENT,
    FirstName TEXT NOT NULL,
    LastName TEXT NOT NULL
);
```

Also do not forget to set the **Copy to Output Directory** property of the file to **Copy if newer** or **Copy always**, based on your needs



Create a class called User, which will be the base entity for our database

```
private class User
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

We'll write two methods for query execution, first one for inserting, updating or removing from database

```
private int ExecuteWrite(string query, Dictionary<string, object> args)
{
    int numberOfRowsAffected;

    //setup the connection to the database
    using (var con = new SQLiteConnection("Data Source=test.db"))
    {
        con.Open();

        //open a new command
        using (var cmd = new SQLiteCommand(query, con))
        {
            //set the arguments given in the query
            foreach (var pair in args)
            {
                cmd.Parameters.AddWithValue(pair.Key, pair.Value);
            }

            //execute the query and get the number of row affected
            numberOfRowsAffected = cmd.ExecuteNonQuery();
        }
    }

    return numberOfRowsAffected;
}
```

```
}
```

第二个用于从数据库读取

```
private DataTable Execute(string query)
{
    if (string.IsNullOrEmpty(query.Trim()))
        return null;

    using (var con = new SQLiteConnection("Data Source=test.db"))
    {
        con.Open();
        using (var cmd = new SQLiteCommand(query, con))
        {
            foreach (KeyValuePair<string, object> entry in args)
            {
                cmd.Parameters.AddWithValue(entry.Key, entry.Value);
            }

            var da = new SQLiteDataAdapter(cmd);
            var dt = new DataTable();
            da.Fill(dt);

            da.Dispose();
            return dt;
        }
    }
}
```

现在让我们进入我们的**CRUD**方法

添加用户

```
private int AddUser(User user)
{
    const string query = "INSERT INTO User(FirstName, LastName) VALUES(@firstName, @lastName)";

    //这里我们设置将在Execute方法中实际替换到查询中的参数值
    //
    var args = new Dictionary<string, object>
    {
        {"@firstName", user.FirstName},
        {"@lastName", user.LastName}
    };

    return ExecuteWrite(query, args);
}
```

编辑用户

```
private int EditUser(User user)
{
    const string query = "UPDATE User SET FirstName = @firstName, LastName = @lastName WHERE Id = @id";

    //这里我们设置将在Execute方法中实际替换到查询中的参数值
    //
    var args = new Dictionary<string, object>
```

```
}
```

and the second one for reading from database

```
private DataTable Execute(string query)
{
    if (string.IsNullOrEmpty(query.Trim()))
        return null;

    using (var con = new SQLiteConnection("Data Source=test.db"))
    {
        con.Open();
        using (var cmd = new SQLiteCommand(query, con))
        {
            foreach (KeyValuePair<string, object> entry in args)
            {
                cmd.Parameters.AddWithValue(entry.Key, entry.Value);
            }

            var da = new SQLiteDataAdapter(cmd);
            var dt = new DataTable();
            da.Fill(dt);

            da.Dispose();
            return dt;
        }
    }
}
```

Now lets get into our **CRUD** methods

Adding user

```
private int AddUser(User user)
{
    const string query = "INSERT INTO User(FirstName, LastName) VALUES(@firstName, @lastName)";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@firstName", user.FirstName},
        {"@lastName", user.LastName}
    };

    return ExecuteWrite(query, args);
}
```

Editing user

```
private int EditUser(User user)
{
    const string query = "UPDATE User SET FirstName = @firstName, LastName = @lastName WHERE Id = @id";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
```

```

    {
        {"@id", user.Id},
        {"@firstName", user.FirstName},
        {"@lastName", user.LastName}
   };

    return ExecuteWrite(query, args);
}

```

删除用户

```

private int DeleteUser(User user)
{
    const string query = "Delete from User WHERE Id = @id";

    //这里我们设置将在Execute方法中实际替换到查询中的参数值
    //
    var args = new Dictionary<string, object>
    {
        {"@id", user.Id}
    };

    return ExecuteWrite(query, args);
}

```

通过Id获取用户

```

private User GetUserById(int id)
{
    var query = "SELECT * FROM User WHERE Id = @id";

    var args = new Dictionary<string, object>
    {
        {"@id", id}
    };

    DataTable dt = ExecuteRead(query, args);

    if (dt == null || dt.Rows.Count == 0)
    {
        return null;
    }

    var user = new User
    {
        Id = Convert.ToInt32(dt.Rows[0]["Id"]),
        FirstName = Convert.ToString(dt.Rows[0]["FirstName"]),
        Lastname = Convert.ToString(dt.Rows[0]["LastName"])
    };

    return user;
}

```

第129.2节：执行查询

```

using (SQLiteConnection conn = new SQLiteConnection(@"Data
Source=data.db;Pooling=true;FailIfMissing=false"))
{
    conn.Open();
    using (SQLiteCommand cmd = new SQLiteCommand(conn))

```

```

    {
        {"@id", user.Id},
        {"@firstName", user.FirstName},
        {"@lastName", user.LastName}
   };

    return ExecuteWrite(query, args);
}

```

Deleting user

```

private int DeleteUser(User user)
{
    const string query = "Delete from User WHERE Id = @id";

    //here we are setting the parameter values that will be actually
    //replaced in the query in Execute method
    var args = new Dictionary<string, object>
    {
        {"@id", user.Id}
    };

    return ExecuteWrite(query, args);
}

```

Getting user by Id

```

private User GetUserById(int id)
{
    var query = "SELECT * FROM User WHERE Id = @id";

    var args = new Dictionary<string, object>
    {
        {"@id", id}
    };

    DataTable dt = ExecuteRead(query, args);

    if (dt == null || dt.Rows.Count == 0)
    {
        return null;
    }

    var user = new User
    {
        Id = Convert.ToInt32(dt.Rows[0]["Id"]),
        FirstName = Convert.ToString(dt.Rows[0]["FirstName"]),
        Lastname = Convert.ToString(dt.Rows[0]["LastName"])
    };

    return user;
}

```

Section 129.2: Executing Query

```

using (SQLiteConnection conn = new SQLiteConnection(@"Data
Source=data.db;Pooling=true;FailIfMissing=false"))
{
    conn.Open();
    using (SQLiteCommand cmd = new SQLiteCommand(conn))

```

```
{  
cmd.CommandText = "query";  
using (SqlDataReader dr = cmd.ExecuteReader())  
{  
    while(dr.Read())  
    {  
        //do stuff  
    }  
}  
}
```

注意: 将 FailIfMissing 设置为 true 会在缺少时创建文件 `data.db`, 但该文件将为空。因此, 任何所需的表都必须重新创建。

```
{  
cmd.CommandText = "query";  
using (SqlDataReader dr = cmd.ExecuteReader())  
{  
    while(dr.Read())  
    {  
        //do stuff  
    }  
}
```

Note: Setting `FailIfMissing` to true creates the file `data.db` if missing. However, the file will be empty. So, any required tables have to be recreated.

第130章：缓存

第130.1节：内存缓存

```
//获取缓存实例  
using System.Runtime.Caching;  
  
var cache = MemoryCache.Default;  
  
//检查缓存中是否包含某个项  
cache.Contains("CacheKey");  
  
//从缓存中获取项  
var item = cache.Get("CacheKey");  
  
//从缓存中获取项，若不存在则添加该项  
object list = MemoryCache.Default.AddOrGetExisting("CacheKey", "object to be stored",  
DateTime.Now.AddHours(12));  
  
//注意：如果项不存在，该方法会添加该项  
//但方法返回null
```

Chapter 130: Caching

Section 130.1: MemoryCache

```
//Get instance of cache  
using System.Runtime.Caching;  
  
var cache = MemoryCache.Default;  
  
//Check if cache contains an item with  
cache.Contains("CacheKey");  
  
//get item from cache  
var item = cache.Get("CacheKey");  
  
//get item from cache or add item if not existing  
object list = MemoryCache.Default.AddOrGetExisting("CacheKey", "object to be stored",  
DateTime.Now.AddHours(12));  
  
//note if item not existing the item is added by this method  
//but the method returns null
```

第131章：代码契约

第131.1节：后置条件

```
public double GetPaymentsTotal(string name)
{
    Contract.Ensures(Contract.Result<double>() >= 0);

    double total = 0.0;

    foreach (var payment in this._payments) {
        if (string.Equals(payment.Name, name)) {
            total += payment.Amount;
        }
    }

    return total;
}
```

第131.2节：不变量

```
namespace CodeContractsDemo
{
    using System;
    using System.Diagnostics.Contracts;

    public class Point
    {
        public int X { get; set; }
        public int Y { get; set; }

        public Point()
        {}

        public Point(int x, int y)
        {
            this.X = x;
            this.Y = y;
        }

        public void Set(int x, int y)
        {
            this.X = x;
            this.Y = y;
        }

        public void Test(int x, int y)
        {
            for (int dx = -x; dx <= x; dx++) {
                this.X = dx;
            }
            Console.WriteLine("当前 X = {0}", this.X);

            for (int dy = -y; dy <= y; dy++) {
                this.Y = dy;
            }
            Console.WriteLine("当前 Y = {0}", this.Y);
        }
    }
}
```

Chapter 131: Code Contracts

Section 131.1: Postconditions

```
public double GetPaymentsTotal(string name)
{
    Contract.Ensures(Contract.Result<double>() >= 0);

    double total = 0.0;

    foreach (var payment in this._payments) {
        if (string.Equals(payment.Name, name)) {
            total += payment.Amount;
        }
    }

    return total;
}
```

Section 131.2: Invariants

```
namespace CodeContractsDemo
{
    using System;
    using System.Diagnostics.Contracts;

    public class Point
    {
        public int X { get; set; }
        public int Y { get; set; }

        public Point()
        {}

        public Point(int x, int y)
        {
            this.X = x;
            this.Y = y;
        }

        public void Set(int x, int y)
        {
            this.X = x;
            this.Y = y;
        }

        public void Test(int x, int y)
        {
            for (int dx = -x; dx <= x; dx++) {
                this.X = dx;
                Console.WriteLine("Current X = {0}", this.X);
            }

            for (int dy = -y; dy <= y; dy++) {
                this.Y = dy;
                Console.WriteLine("Current Y = {0}", this.Y);
            }
        }
    }
}
```

```

Console.WriteLine("X = {0}", this.X);
Console.WriteLine("Y = {0}", this.Y);
}

[ContractInvariantMethod]
private void ValidateCoordinates()
{
    Contract.Invariant(this.X >= 0);
    Contract.Invariant(this.Y >= 0);
}
}

```

第131.3节：在接口上定义契约

```

[ContractClass(typeof(ValidationContract))]
interface IValidation
{
    string CustomerID{get;set;}
    string Password{get;set;}
}

[ContractClassFor(typeof(IValidation))]
sealed class ValidationContract:IValidation
{
    string IValidation.CustomerID
    {
        [Pure]
        get
        {
            return Contract.Result<string>();
        }
        设置
        {
            Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(value), "客户ID不能为空！！");
        }
    }

    string IValidation.Password
    {
        [Pure]
        get
        {
            return Contract.Result<string>();
        }
        设置
        {
            Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(value), "密码不能为空！！");
        }
    }

    class Validation:IValidation
    {
        public string GetCustomerPassword(string customerID)
        {
            Contract.Requires(!string.IsNullOrEmpty(customerID), "客户ID不能为空");
            Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(customerID), "异常！！");
            Contract.Ensures(Contract.Result<string>() != null);
        }
    }
}

```

```

Console.WriteLine("X = {0}", this.X);
Console.WriteLine("Y = {0}", this.Y);
}

[ContractInvariantMethod]
private void ValidateCoordinates()
{
    Contract.Invariant(this.X >= 0);
    Contract.Invariant(this.Y >= 0);
}
}

```

Section 131.3: Defining Contracts on Interface

```

[ContractClass(typeof(ValidationContract))]
interface IValidation
{
    string CustomerID{get;set;}
    string Password{get;set;}
}

[ContractClassFor(typeof(IValidation))]
sealed class ValidationContract:IValidation
{
    string IValidation.CustomerID
    {
        [Pure]
        get
        {
            return Contract.Result<string>();
        }
        set
        {
            Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(value), "Customer ID cannot be null!!!");
        }
    }

    string IValidation.Password
    {
        [Pure]
        get
        {
            return Contract.Result<string>();
        }
        set
        {
            Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(value), "Password cannot be null!!!");
        }
    }

    class Validation:IValidation
    {
        public string GetCustomerPassword(string customerID)
        {
            Contract.Requires(!string.IsNullOrEmpty(customerID), "Customer ID cannot be Null");
            Contract.Requires<ArgumentNullException>(!string.IsNullOrEmpty(customerID), "Exception!!");
            Contract.Ensures(Contract.Result<string>() != null);
        }
    }
}

```

```

        string password="AAA@1234";
        if (customerID!=null)
        {
            return password;
        }
        else
        {
            return null;
        }
    }

private string m_custID, m_PWD;

public string CustomerID
{
    get
    {
        return m_custID;
    }
    设置
    {
        m_custID = value;
    }
}

public string Password
{
    get
    {
        return m_PWD;
    }
    设置
    {
        m_PWD = value;
    }
}

```

在上述代码中，我们定义了一个名为IValidation的接口，并带有一个属性[ContractClass]。该属性接受一个类的地址，该类中实现了接口的契约。类ValidationContract利用接口中定义的属性，并使用Contract.Requires<T>来检查空值。T是一个异常类。

我们还用属性[Pure]标记了get访问器。Pure属性确保该方法或属性不会改变实现IValidation接口的类的实例状态。

第131.4节：前置条件

```

namespace CodeContractsDemo
{
    using System;
    using System.Collections.Generic;
    using System.Diagnostics.Contracts;

    public class PaymentProcessor
    {
        private List<Payment> _payments = new List<Payment>();

        public void Add(Payment payment)
    }
}

```

```

        string password="AAA@1234";
        if (customerID!=null)
        {
            return password;
        }
        else
        {
            return null;
        }
    }

private string m_custID, m_PWD;

public string CustomerID
{
    get
    {
        return m_custID;
    }
    set
    {
        m_custID = value;
    }
}

public string Password
{
    get
    {
        return m_PWD;
    }
    set
    {
        m_PWD = value;
    }
}

```

In the above code, we have defined an interface called IValidation with an attribute [ContractClass]. This attribute takes an address of a class where we have implemented a contract for an Interface. The class ValidationContract makes use of properties defined in the interface and checks for the null values using Contract.Requires<T>. T is an exception class.

We have also marked the get accessor with an attribute [Pure]. The pure attribute ensures that the method or a property does not change the instance state of a class in which IValidation interface is implemented.

Section 131.4: Preconditions

```

namespace CodeContractsDemo
{
    using System;
    using System.Collections.Generic;
    using System.Diagnostics.Contracts;

    public class PaymentProcessor
    {
        private List<Payment> _payments = new List<Payment>();

        public void Add(Payment payment)
    }
}

```

```
{  
    Contract.Requires(payment != null);  
    Contract.Requires(!string.IsNullOrEmpty(payment.Name));  
    Contract.Requires(payment.Date <= DateTime.Now);  
    Contract.Requires(payment.Amount > 0);  
  
    this._payments.Add(payment);  
}  
}  
}
```

```
{  
    Contract.Requires(payment != null);  
    Contract.Requires(!string.IsNullOrEmpty(payment.Name));  
    Contract.Requires(payment.Date <= DateTime.Now);  
    Contract.Requires(payment.Amount > 0);  
  
    this._payments.Add(payment);  
}  
}  
}
```

belindoc.com

第132章：代码契约和断言

第132.1节：用于检查逻辑应始终为真的断言

断言不是用来测试输入参数，而是用来验证程序流程是否正确——即你可以在某个时间点对代码做出某些假设。换句话说：

Debug.Assert 进行的测试应始终假设被测试的值为真。

Debug.Assert 仅在调试 (DEBUG) 版本中执行；在发布 (RELEASE) 版本中会被过滤掉。它应被视为除单元测试之外的调试工具，而不是代码契约或输入验证方法的替代品。

例如，这是一个好的断言：

```
var systemData = RetrieveSystemConfiguration();
Debug.Assert(systemData != null);
```

这里使用断言是一个好的选择，因为我们可以假设 RetrieveSystemConfiguration() 会返回一个有效值并且永远不会返回 null。

这是另一个好的例子：

```
UserData user = RetrieveUserData();
Debug.Assert(user != null);
Debug.Assert(user.Age > 0);
int year = DateTime.Today.Year - user.Age;
```

首先，我们可以假设 RetrieveUserData() 会返回一个有效值。然后，在使用 Age 属性之前，我们验证这个假设（应该总是成立），即用户的年龄严格大于零。

这是一个错误的断言示例：

```
string input = Console.ReadLine();
int age = Convert.ToInt32(input);
Debug.Assert(age > 16);
Console.WriteLine("Great, you are over 16");
```

Assert 不是用于输入验证，因为假设该断言总是成立是不正确的。你必须使用输入验证方法来实现这一点。在上述情况下，你还应该首先验证输入值是否为数字。

Chapter 132: Code Contracts and Assertions

Section 132.1: Assertions to check logic should always be true

Assertions are used not to perform testing of input parameters, but to verify that program flow is correct -- i.e., that you can make certain assumptions about your code at a certain point in time. In other words: a test done with Debug.Assert should *always* assume that the value tested is `true`.

Debug.Assert only executes in DEBUG builds; it is filtered out of RELEASE builds. It must be considered a debugging tool in addition to unit testing and not as a replacement of code contracts or input validation methods.

For instance, this is a good assertion:

```
var systemData = RetrieveSystemConfiguration();
Debug.Assert(systemData != null);
```

Here assert is a good choice because we can assume that RetrieveSystemConfiguration() will return a valid value and will never return null.

Here is another good example:

```
UserData user = RetrieveUserData();
Debug.Assert(user != null);
Debug.Assert(user.Age > 0);
int year = DateTime.Today.Year - user.Age;
```

First, we may assume that RetrieveUserData() will return a valid value. Then, before using the Age property, we verify the assumption (which should always be true) that the age of the user is strictly positive.

This is a bad example of assert:

```
string input = Console.ReadLine();
int age = Convert.ToInt32(input);
Debug.Assert(age > 16);
Console.WriteLine("Great, you are over 16");
```

Assert is not for input validation because it is incorrect to assume that this assertion will always be true. You must use input validation methods for that. In the case above, you should also verify that the input value is a number in the first place.

第133章：结构型设计模式

结构型设计模式是描述对象和类如何组合并形成大型结构的模式，通过识别实现实体之间关系的简单方式来简化设计。共有七种结构型模式，分别是：适配器（Adapter）、桥接（Bridge）、组合（Composite）、装饰（Decorator）、外观（Facade）、享元（Flyweight）和代理（Proxy）。

第133.1节：适配器设计模式

“[适配器（Adapter）](#)”顾名思义，是让两个互不兼容的接口能够相互通信的对象。

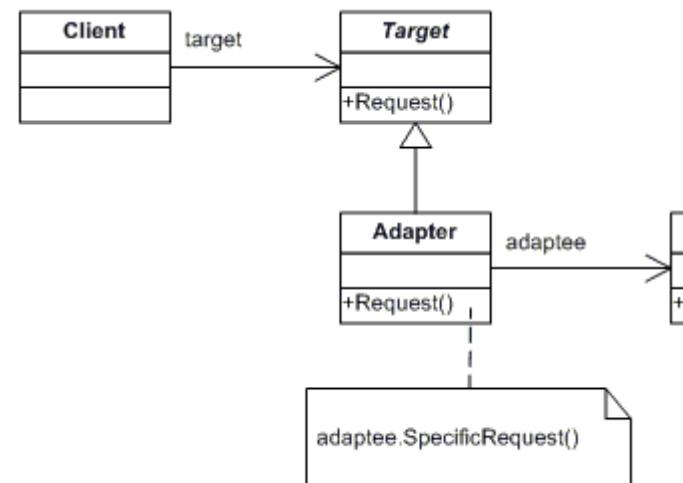
例如：如果你购买一部iPhone 8（或任何其他苹果产品），你需要很多适配器。因为默认接口不支持音频插孔或USB。通过这些适配器，你可以使用有线耳机，或者使用普通的以太网线。所以“两个互不兼容的接口能够相互通信”。

从技术角度讲，这意味着：将一个类的接口转换成客户端期望的另一个接口。适配器让因接口不兼容而无法协作的类能够一起工作。参与该模式的类和对象有：

适配器模式包含4个元素

1. **目标接口（ITarget）**：这是客户端用来实现功能的接口。
2. 适配者：这是客户端所需的功能，但其接口不兼容客户端。
3. 客户端：这是希望通过使用适配者（Adaptee）代码来实现某些功能的类。
4. 适配器：这是将实现ITarget接口并调用客户端想要调用的适配者代码的类。

UML



第一个代码示例（理论示例）。

Chapter 133: Structural Design Patterns

Structural design patterns are patterns that describe how objects and classes can be combined and form a large structure and that ease design by identifying a simple way to realize relationships between entities. There are seven structural patterns described. They are as follows: Adapter, Bridge, Composite, Decorator, Facade, Flyweight and Proxy

Section 133.1: Adapter Design Pattern

“[Adapter](#)” as the name suggests is the object which lets two mutually incompatible interfaces communicate with each other.

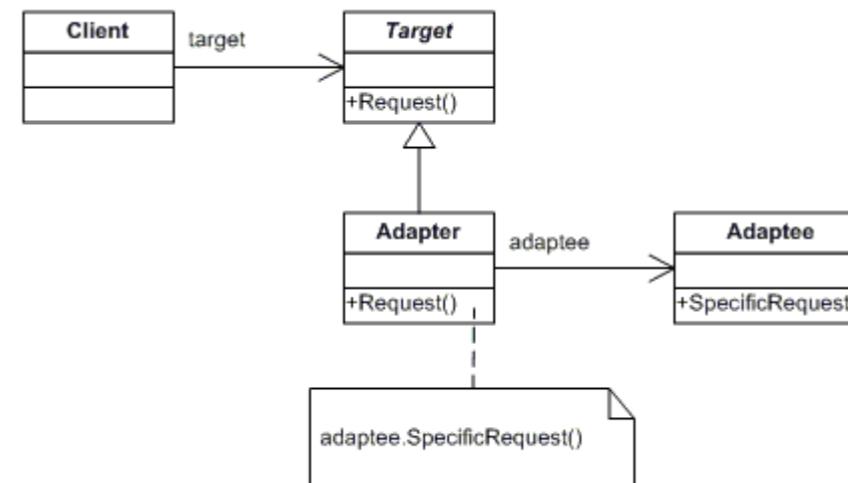
For example: if you buy a Iphone 8 (or any other Apple product) you need alot of adapters. Because the default interface does not support audio jac or USB. With these adapters you can use earphones with wires or you can use a normal Ethernet cable. So “two mutually incompatible interfaces communicate with each other”.

So in technical terms this means: Convert the interface of a class into another interface that a clients expect. Adapter let classes work together that couldn't otherwise because of incompatible interfaces. The classes and objects participating in this pattern are:

The adapter pattern exists out 4 elements

1. **ITarget**: This is the interface which is used by the client to achieve functionality.
2. **Adaptee**: This is the functionality which the client desires but its interface is not compatible with the client.
3. **Client**: This is the class which wants to achieve some functionality by using the adaptee's code.
4. **Adapter**: This is the class which would implement ITarget and would call the Adaptee code which the client wants to call.

UML



First code Example (Theoretical example).

```

public interface ITarget
{
    void MethodA();
}

public class Adaptee
{
    public void MethodB()
    {
        Console.WriteLine("MethodB() 被调用");
    }
}

public class 客户端
{
    private ITarget 目标;

    public 客户端(ITarget 目标)
    {
        this.目标 = 目标;
    }

    public void 发起请求()
    {
        目标.方法A();
    }
}

public class 适配器 : 被适配者, ITarget
{
    public void MethodA()
    {
        方法B();
    }
}

```

第二段代码示例（现实世界的实现）

```

/// <summary>
/// 接口：这是客户端用来实现功能的接口。
/// </summary>
public interface ITarget
{
    List<string> 获取员工列表();
}

/// <summary>
/// 适配者：这是客户端所需的功能，但其接口与客户端不兼容
/// </summary>
public class 公司员工
{
    public string[][] 获取员工()
    {
        string[][] 员工 = new string[4][];
        员工[0] = new string[] { "100", "Deepak", "团队负责人" };
        员工[1] = new string[] { "101", "Rohit", "开发人员" };
        员工[2] = new string[] { "102", "Gautam", "开发人员" };
        员工[3] = new string[] { "103", "Dev", "测试人员" };
    }
}

```

```

public interface ITarget
{
    void MethodA();
}

public class Adaptee
{
    public void MethodB()
    {
        Console.WriteLine("MethodB() is called");
    }
}

public class Client
{
    private ITarget target;

    public Client(ITarget target)
    {
        this.target = target;
    }

    public void MakeRequest()
    {
        target.MethodA();
    }
}

public class Adapter : Adaptee, ITarget
{
    public void MethodA()
    {
        MethodB();
    }
}

```

Second code example (Real world imlementation)

```

/// <summary>
/// Interface: This is the interface which is used by the client to achieve functionality.
/// </summary>
public interface ITarget
{
    List<string> GetEmployeeList();
}

/// <summary>
/// Adaptee: This is the functionality which the client desires but its interface is not compatible
/// with the client.
/// </summary>
public class CompanyEmployees
{
    public string[][] GetEmployees()
    {
        string[][] employees = new string[4][];
        employees[0] = new string[] { "100", "Deepak", "Team Leader" };
        employees[1] = new string[] { "101", "Rohit", "Developer" };
        employees[2] = new string[] { "102", "Gautam", "Developer" };
        employees[3] = new string[] { "103", "Dev", "Tester" };
    }
}

```

```

        return 员工;
    }

/// <summary>
/// 客户端：这是希望通过使用适配者代码（员工列表）来实现某些功能的类
/// </summary>
public class 第三方计费系统
{
    /*
     * 该类来自第三方，您无法控制它。
     * 但它需要一个员工列表来完成其工作
     */

    private ITarget employeeSource;

    public ThirdPartyBillingSystem(ITarget employeeSource)
    {
        this.employeeSource = employeeSource;
    }

    public void ShowEmployeeList()
    {
        // 调用接口中的客户列表
        List<string> employee = employeeSource.GetEmployeeList();

        Console.WriteLine("##### 员工列表 #####");
        foreach (var item in employee)
        {
            Console.Write(item);
        }
    }

    /// <summary>
    /// 适配器：这是实现ITarget接口并调用客户端想要调用的被适配者代码的类。
    /// </summary>
    public class EmployeeAdapter : CompanyEmployees, ITarget
    {
        public List<string> GetEmployeeList()
        {
            List<string> employeeList = new List<string>();
            string[][] employees = GetEmployees();
            foreach (string[] employee in employees)
            {
                employeeList.Add(employee[0]);
                employeeList.Add(",");
            }
            employeeList.Add(employee[1]);
            employeeList.Add(",");
            employeeList.Add(employee[2]);
            employeeList.Add("");
        }

        return employeeList;
    }

    /**
     * 演示
     */
}

```

```

        return employees;
    }

    /// <summary>
    /// Client: This is the class which wants to achieve some functionality by using the adaptee's code
    /// (list of employees).
    /// </summary>
    public class ThirdPartyBillingSystem
    {
        /*
         * This class is from a third party and you do'n have any control over it.
         * But it requires a Employee list to do its work
         */

        private ITarget employeeSource;

        public ThirdPartyBillingSystem(ITarget employeeSource)
        {
            this.employeeSource = employeeSource;
        }

        public void ShowEmployeeList()
        {
            // call the client list in the interface
            List<string> employee = employeeSource.GetEmployeeList();

            Console.WriteLine("##### Employee List #####");
            foreach (var item in employee)
            {
                Console.Write(item);
            }
        }

        /// <summary>
        /// Adapter: This is the class which would implement ITarget and would call the Adaptee code which
        /// the client wants to call.
        /// </summary>
        public class EmployeeAdapter : CompanyEmployees, ITarget
        {
            public List<string> GetEmployeeList()
            {
                List<string> employeeList = new List<string>();
                string[][] employees = GetEmployees();
                foreach (string[] employee in employees)
                {
                    employeeList.Add(employee[0]);
                    employeeList.Add(",");
                    employeeList.Add(employee[1]);
                    employeeList.Add(",");
                    employeeList.Add(employee[2]);
                    employeeList.Add("\n");
                }
            }

            return employeeList;
        }

        /**
         * Demo
         */
    }
}

```

```

/// 
class 程序
{
    static void Main(string[] args)
    {
        ITarget Itarget = new EmployeeAdapter();
        ThirdPartyBillingSystem client = new ThirdPartyBillingSystem(Itarget);
        client.ShowEmployeeList();
        Console.ReadKey();
    }
}

```

何时使用

- 允许一个系统使用另一个与其不兼容的系统的类。
- 允许新系统与已存在且相互独立的系统之间进行通信
- Ado.Net SqlDataAdapter、OracleAdapter、MySqlAdapter 是适配器模式的最佳示例。

belindoc.com

```

/// 
class Programs
{
    static void Main(string[] args)
    {
        ITarget Itarget = new EmployeeAdapter();
        ThirdPartyBillingSystem client = new ThirdPartyBillingSystem(Itarget);
        client.ShowEmployeeList();
        Console.ReadKey();
    }
}

```

When to use

- Allow a system to use classes of another system that is incompatible with it.
- Allow communication between new and already existing system which are independent to each other
- Ado.Net SqlDataAdapter, OracleAdapter, MySqlAdapter are best example of Adapter Pattern.

第134章：创建型设计模式

第134.1节：单例模式

单例模式旨在限制一个类只能创建一个唯一实例。

该模式适用于只需要一个实例的场景，例如：

- 一个协调其他对象交互的单一类，例如管理类
- 或者表示唯一单一资源的类，例如日志组件

实现单例模式最常见的方法之一是通过静态工厂方法，例如

`CreateInstance()` 或 `GetInstance()`（或在C#中使用静态属性`Instance`），该方法设计为始终返回同一个实例。

首次调用该方法或属性时会创建并返回单例实例。此后，该方法总是返回同一个实例。这样，单例对象始终只有一个实例。

通过将类的构造函数设为`private`，可以防止通过`new`创建实例。

以下是在C#中实现单例模式的典型代码示例：

```
class Singleton
{
    // 由于_instance成员被设为私有，获取唯一实例的唯一方式是通过下面的静态Instance属性。也可以用
    // GetInstance()方法代替该属性实现类似功能。
    private static Singleton _instance = null;

    // 将构造函数设为私有，防止通过类似Singleton s = new Singleton()的方式创建其他实
    // 例，避免误用。
    private Singleton()
    {
    }

    public static 单例实例
    {
        get
        {
            // 第一次调用将创建唯一的实例。
            if (_instance == null)
            {
                _instance = new 单例();
            }

            // 之后的每次调用都会返回上面创建的单一实例。
            return _instance;
        }
    }
}
```

为了进一步说明这一模式，下面的代码检查当多次调用`Instance`属性时，是否返回了相同的单例实例。

```
class 程序
{
```

Chapter 134: Creational Design Patterns

Section 134.1: Singleton Pattern

The Singleton pattern is designed to restrict creation of a class to exactly one single instance.

This pattern is used in a scenario where it makes sense to have only one of something, such as:

- a single class that orchestrates other objects' interactions, ex. Manager class
- or one class that represents a unique, single resource, ex. Logging component

One of the most common ways to implement the Singleton pattern is via a static **factory method** such as a `CreateInstance()` or `GetInstance()` (or a static property in C#, `Instance`), which is then designed to always return the same instance.

The first call to the method or property creates and returns the Singleton instance. Thereafter, the method always returns the same instance. This way, there is only ever one instance of the singleton object.

Preventing creation of instances via `new` can be accomplished by making the class constructor(s) `private`.

Here is a typical code example for implementing a Singleton pattern in C#:

```
class Singleton
{
    // Because the _instance member is made private, the only way to get the single
    // instance is via the static Instance property below. This can also be similarly
    // achieved with a GetInstance() method instead of the property.
    private static Singleton _instance = null;

    // Making the constructor private prevents other instances from being
    // created via something like Singleton s = new Singleton(), protecting
    // against unintentional misuse.
    private Singleton()
    {
    }

    public static Singleton Instance
    {
        get
        {
            // The first call will create the one and only instance.
            if (_instance == null)
            {
                _instance = new Singleton();
            }

            // Every call afterwards will return the single instance created above.
            return _instance;
        }
    }
}
```

To illustrate this pattern further, the code below checks whether an identical instance of the Singleton is returned when the `Instance` property is called more than once.

```
class Program
{
```

```

static void Main(string[] args)
{
    单例 s1 = 单例.Instance;
    单例 s2 = 单例.Instance;

    // 上述两个单例对象现在都应该引用同一个单例实例。
    if (Object.ReferenceEquals(s1, s2))
    {
        Console.WriteLine("单例模式正在运行");
    }
    else
    {
        // 否则，单例实例属性返回的不是唯一的单一实例。
        Console.WriteLine("单例模式已被破坏");
    }
}

```

注意：此实现不是线程安全的。

要查看更多示例，包括如何实现线程安全，请访问：[Singleton Implementation](#)

单例模式在概念上类似于全局变量，并引发类似的设计缺陷和问题。因此，单例模式被广泛认为是一种反模式。

[访问“单例模式有什么不好？”了解更多关于使用单例模式所带来的问题。](#)

在C#中，你可以将类定义为static，这会使所有成员成为静态的，且该类无法被实例化。基于此，通常会看到静态类被用来替代单例模式。

[关于两者的主要区别，请访问C# 单例模式与静态类。](#)

第134.2节：工厂方法模式

工厂方法是创建型设计模式之一。它用于解决创建对象时不指定具体结果类型的问题。本文档将教你如何正确使用工厂方法设计模式。

让我通过一个简单的例子向你解释这个想法。想象你在一家工厂工作，生产三种设备——电流表、电压表和电阻表。你正在为一台中央计算机编写程序，该程序将创建所选设备，但你不知道老板最终决定生产哪种设备。

让我们创建一个接口IDevice，包含所有设备共有的一些功能：

```

public interface IDevice
{
    int Measure();
    void TurnOff();
    void TurnOn();
}

```

现在，我们可以创建表示我们设备的类。这些类必须实现IDevice接口：

```

public class Ammeter : IDevice
{
    private Random r = null;
    public Ammeter()
    {

```

```

static void Main(string[] args)
{
    Singleton s1 = Singleton.Instance;
    Singleton s2 = Singleton.Instance;

    // Both Singleton objects above should now reference the same Singleton instance.
    if (Object.ReferenceEquals(s1, s2))
    {
        Console.WriteLine("Singleton is working");
    }
    else
    {
        // Otherwise, the Singleton Instance property is returning something
        // other than the unique, single instance when called.
        Console.WriteLine("Singleton is broken");
    }
}

```

Note: this implementation is not thread safe.

To see more examples, including how to make this thread-safe, visit: [Singleton Implementation](#)

Singletons are conceptually similar to a global value, and cause similar design flaws and concerns. Because of this, the Singleton pattern is widely regarded as an anti-pattern.

[Visit "What is so bad about Singletons?" for more information on the problems that arise with their use.](#)

In C#, you have the ability to make a class **static**, which makes all members static, and the class cannot be instantiated. Given this, it is common to see static classes used in place of the Singleton pattern.

[For key differences between the two, visit \[C# Singleton Pattern Versus Static Class\]\(#\).](#)

Section 134.2: Factory Method pattern

Factory Method is one of creational design patterns. It is used to deal with the problem of creating objects without specifying exact result type. This document will teach you how to use Factory Method DP properly.

Let me explain the idea of it to you on a simple example. Imagine you're working in a factory that produces three types of devices - Ammeter, Voltmeter and resistance meter. You are writing a program for a central computer that will create selected device, but you don't know final decision of your boss on what to produce.

Let's create an interface IDevice with some common functions that all devices have:

```

public interface IDevice
{
    int Measure();
    void TurnOff();
    void TurnOn();
}

```

Now, we can create classes that represent our devices. Those classes must implement IDevice interface:

```

public class Ammeter : IDevice
{
    private Random r = null;
    public Ammeter()
    {

```

```

r = new Random();
}
public int Measure() { return r.Next(-25, 60); }
public void TurnOff() { Console.WriteLine("电流表闪烁灯光表示再见！"); }
public void TurnOn() { Console.WriteLine("电流表开启中..."); }
}
public class 欧姆表 : IDevice
{
    private Random r = null;
    public OhmMeter()
    {
    }
    r = new Random();
}
public int Measure() { return r.Next(0, 0000000); }
public void TurnOff() { Console.WriteLine("OhmMeter 闪烁灯光说再见！"); }
public void TurnOn() { Console.WriteLine("OhmMeter 开机中..."); }
}
public class Voltmeter : IDevice
{
    private Random r = null;
    public Voltmeter()
    {
    }
    r = new Random();
}
public int Measure() { return r.Next(-230, 230); }
public void TurnOff() { Console.WriteLine("Voltmeter 闪烁灯光说再见！"); }
public void TurnOn() { Console.WriteLine("Voltmeter 开机中..."); }
}

```

现在我们必须定义工厂方法。让我们创建一个带有静态方法的DeviceFactory类：

```

public enum Device
{
    AM,
    VOLT,
    OHM
}
public class DeviceFactory
{
    public static IDevice CreateDevice(Device d)
    {
        switch(d)
        {
            case Device.AM: return new Ammeter();
            case Device.VOLT: return new Voltmeter();
            case Device.OHM: return new OhmMeter();
            default: return new Ammeter();
        }
    }
}

```

太好了！让我们测试一下代码：

```

public class Program
{
    static void Main(string[] args)
    {
        IDevice device = DeviceFactory.CreateDevice(Device.AM);
        device.TurnOn();
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
    }
}

```

```

r = new Random();
}
public int Measure() { return r.Next(-25, 60); }
public void TurnOff() { Console.WriteLine("Ammeter flashes lights saying good bye!"); }
public void TurnOn() { Console.WriteLine("Ammeter turns on..."); }
}
public class OhmMeter : IDevice
{
    private Random r = null;
    public OhmMeter()
    {
    }
    r = new Random();
}
public int Measure() { return r.Next(0, 1000000); }
public void TurnOff() { Console.WriteLine("OhmMeter flashes lights saying good bye!"); }
public void TurnOn() { Console.WriteLine("OhmMeter turns on..."); }
}
public class Voltmeter : IDevice
{
    private Random r = null;
    public Voltmeter()
    {
    }
    r = new Random();
}
public int Measure() { return r.Next(-230, 230); }
public void TurnOff() { Console.WriteLine("Voltmeter flashes lights saying good bye!"); }
public void TurnOn() { Console.WriteLine("Voltmeter turns on..."); }
}

```

Now we have to define factory method. Let's create DeviceFactory class with static method inside:

```

public enum Device
{
    AM,
    VOLT,
    OHM
}
public class DeviceFactory
{
    public static IDevice CreateDevice(Device d)
    {
        switch(d)
        {
            case Device.AM: return new Ammeter();
            case Device.VOLT: return new Voltmeter();
            case Device.OHM: return new OhmMeter();
            default: return new Ammeter();
        }
    }
}

```

Great! Let's test our code:

```

public class Program
{
    static void Main(string[] args)
    {
        IDevice device = DeviceFactory.CreateDevice(Device.AM);
        device.TurnOn();
        Console.WriteLine(device.Measure());
        Console.WriteLine(device.Measure());
    }
}

```

```

Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
device.TurnOff();
Console.WriteLine();

device = DeviceFactory.CreateDevice(Device.VOLT);
device.TurnOn();
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
device.TurnOff();
Console.WriteLine();

device = DeviceFactory.CreateDevice(Device.OHM);
device.TurnOn();
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
device.TurnOff();
Console.WriteLine();
}
}

```

这是运行此代码后您可能看到的示例输出：

```

电流表开启...
36
6
33
43
24

电流表闪烁灯光表示再见！

电压表开启...
102
-61
85
138
36

电压表闪烁灯光表示再见！

欧姆表开启.....

```

```

Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
device.TurnOff();
Console.WriteLine();

device = DeviceFactory.CreateDevice(Device.VOLT);
device.TurnOn();
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
device.TurnOff();
Console.WriteLine();

device = DeviceFactory.CreateDevice(Device.OHM);
device.TurnOn();
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
Console.WriteLine(device.Measure());
device.TurnOff();
Console.WriteLine();
}
}

```

This is the example output you might see after running this code:

```

AmMeter turns on...
36
6
33
43
24

AmMeter flashes lights saying good bye!

VoltMeter turns on...
102
-61
85
138
36

VoltMeter flashes lights saying good bye!

OhmMeter turns on...

```

723828

368536

685412

800266

578595

欧姆表闪烁灯光表示再见！

723828

368536

685412

800266

578595

OhmMeter flashes lights saying good bye!

第134.3节：抽象工厂模式

提供一个接口，用于创建一系列相关或相互依赖的对象，而无需指定它们的具体类。

本例演示了如何使用不同的工厂为电脑游戏创建不同的动物世界。

虽然大陆工厂创建的动物不同，但动物之间的互动保持不变。

```
using System;

namespace 四人帮.抽象工厂
{
    /// <summary>
    /// Real-World的MainApp启动类
    /// 抽象工厂设计模式。
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// 控制台应用程序的入口点。
        /// </summary>
        public static void Main()
        {
            // 创建并运行非洲动物世界
            ContinentFactory 非洲工厂 = new 非洲工厂();
            动物世界 world = new 动物世界(非洲工厂);
            world.运行食物链();

            // 创建并运行美洲动物世界
            ContinentFactory 美洲工厂 = new 美洲工厂();
            world = new 动物世界(美洲工厂);
            world.运行食物链();

            // 等待用户输入
            控制台.读取按键();
        }

        /// <summary>
        /// "抽象工厂"抽象类
        /// </summary>
        抽象类 大陆工厂
        {
            public abstract 食草动物 创建食草动物();
            public abstract 食肉动物 创建食肉动物();
        }
    }
}
```

Section 134.3: Abstract Factory Pattern

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

In this example demonstrates the creation of different animal worlds for a computer game using different factories. Although the animals created by the Continent factories are different, the interactions among the animals remain the same.

```
using System;

namespace GangOfFour.AbstractFactory
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Abstract Factory Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        public static void Main()
        {
            // Create and run the African animal world
            ContinentFactory africa = new AfricaFactory();
            AnimalWorld world = new AnimalWorld(africa);
            world.RunFoodChain();

            // Create and run the American animal world
            ContinentFactory america = new AmericaFactory();
            world = new AnimalWorld(america);
            world.RunFoodChain();

            // Wait for user input
            Console.ReadKey();
        }
    }

    /// <summary>
    /// The 'AbstractFactory' abstract class
    /// </summary>
    abstract class ContinentFactory
    {
        public abstract Herbivore CreateHerbivore();
        public abstract Carnivore CreateCarnivore();
    }
}
```

```

/// <summary>
/// "ConcreteFactory1" 类
///
class 非洲工厂 : 大陆工厂
{
    public override 创建草食动物()
    {
        return new 角马();
    }
    public override 创建肉食动物()
    {
        return new 狮子();
    }
}

/// <summary>
/// "具体工厂2"类
/// </summary>
class 美洲工厂 : 大陆工厂
{
    public override 创建草食动物()
    {
        return new 野牛();
    }
    public override 创建肉食动物()
    {
        return new 狼();
    }
}

/// <summary>
/// "抽象产品A"抽象类
/// </summary>
抽象类 食草动物
{
}

/// <summary>
/// 'AbstractProductB' 抽象类
/// </summary>
抽象类 食肉动物
{
    public abstract void 吃(食草动物 h);
}

/// <summary>
/// 'ProductA1' 类
/// </summary>
类 牛羚 : 食草动物
{
}

/// <summary>
/// 'ProductB1' 类
/// </summary>
类 狮子 : 食肉动物
{
    public override void 吃(食草动物 h)
    {
        // 吃牛羚
        Console.WriteLine(this.GetType().Name +

```

```

/// <summary>
/// The 'ConcreteFactory1' class
/// </summary>
class AfricaFactory : ContinentFactory
{
    public override Herbivore CreateHerbivore()
    {
        return new Wildebeest();
    }
    public override Carnivore CreateCarnivore()
    {
        return new Lion();
    }
}

/// <summary>
/// The 'ConcreteFactory2' class
/// </summary>
class AmericaFactory : ContinentFactory
{
    public override Herbivore CreateHerbivore()
    {
        return new Bison();
    }
    public override Carnivore CreateCarnivore()
    {
        return new Wolf();
    }
}

/// <summary>
/// The 'AbstractProductA' abstract class
/// </summary>
abstract class Herbivore
{
}

/// <summary>
/// The 'AbstractProductB' abstract class
/// </summary>
abstract class Carnivore
{
    public abstract void Eat(Herbivore h);
}

/// <summary>
/// The 'ProductA1' class
/// </summary>
class Wildebeest : Herbivore
{
}

/// <summary>
/// The 'ProductB1' class
/// </summary>
class Lion : Carnivore
{
    public override void Eat(Herbivore h)
    {
        // Eat Wildebeest
        Console.WriteLine(this.GetType().Name +

```

```

    " 吃 " + h.GetType().Name);
}

}

/// <summary>
/// "ProductA2" 类
/// </summary>
class 野牛 : 食草动物
{
}

/// <summary>
/// "ProductB2" 类
/// </summary>
class 狼 : 食肉动物
{
    public override void 吃(食草动物 h)
    {
        // 吃野牛
        Console.WriteLine(this.GetType().Name +
            " 吃 " + h.GetType().Name);
    }
}

/// <summary>
/// "客户端" 类
/// </summary>
class 动物世界
{
    private 食草动物 _herbivore;
    private 食肉动物 _carnivore;

    // 构造函数
    public 动物世界(大陆工厂 factory)
    {
        _carnivore = factory.创建食肉动物();
        _herbivore = factory.创建食草动物();
    }

    public void 运行食物链()
    {
        _carnivore.吃(_herbivore);
    }
}

```

输出：

狮子吃角马
狼吃野牛

第134.4节：建造者模式

将复杂对象的构建与其表示分离，使得相同的构建过程可以创建不同的表示，并且对对象的组装提供高度控制。

本例演示了建造者模式，其中不同的车辆按步骤组装

```

    " eats " + h.GetType().Name);
}

}

/// <summary>
/// The 'ProductA2' class
/// </summary>
class Bison : Herbivore
{
}

/// <summary>
/// The 'ProductB2' class
/// </summary>
class Wolf : Carnivore
{
    public override void Eat(Herbivore h)
    {
        // Eat Bison
        Console.WriteLine(this.GetType().Name +
            " eats " + h.GetType().Name);
    }
}

/// <summary>
/// The 'Client' class
/// </summary>
class AnimalWorld
{
    private Herbivore _herbivore;
    private Carnivore _carnivore;

    // Constructor
    public AnimalWorld(ContinentFactory factory)
    {
        _carnivore = factory.CreateCarnivore();
        _herbivore = factory.CreateHerbivore();
    }

    public void RunFoodChain()
    {
        _carnivore.Eat(_herbivore);
    }
}

```

Output:

Lion eats Wildebeest
Wolf eats Bison

Section 134.4: Builder Pattern

Separate the construction of a complex object from its representation so that the same construction process can create different representations and provides a high level of control over the assembly of the objects.

In this example demonstrates the Builder pattern in which different vehicles are assembled in a step-by-step

时尚。商店使用车辆制造者 (VehicleBuilders) 通过一系列连续步骤构建各种车辆。

```
using System;
using System.Collections.Generic;

namespace 四人帮.建造者模式
{
    /// <summary>
    /// 现实世界建造者设计模式的主程序启动类。
    /// </summary>
    public class 主程序
    {
        /// <summary>
        /// 控制台应用程序的入口点。
        /// </summary>
        public static void Main()
        {
            车辆建造者 builder;

            // 使用车辆建造者创建商店
            商店 shop = new 商店();

            // 构建并展示车辆
            builder = new 滑板车建造者();
            shop.构建(builder);
            builder.车辆.展示();

            builder = new 汽车建造者();
            shop.构建(builder);
            builder.车辆.展示();

            builder = new 摩托车建造者();
            shop.构建(builder);
            builder.车辆.展示();

            // 等待用户
            控制台.读取按键();
        }
    }

    /// <summary>
    /// “导演”类
    /// </summary>
    class 商店
    {
        // 建造者使用一系列复杂的步骤
        public void 构建(车辆建造者 vehicleBuilder)
        {
            vehicleBuilder.建造车架();
            vehicleBuilder.建造引擎();
            vehicleBuilder.建造车轮();
            vehicleBuilder.建造车门();
        }
    }

    /// <summary>
    /// “建造者”抽象类
    /// </summary>
    abstract class 车辆建造者
    {
        protected 车辆 vehicle;
```

fashion. The Shop uses VehicleBuilders to construct a variety of Vehicles in a series of sequential steps.

```
using System;
using System.Collections.Generic;

namespace GangOfFour.Builder
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Builder Design Pattern.
    /// </summary>
    public class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        public static void Main()
        {
            VehicleBuilder builder;

            // Create shop with vehicle builders
            Shop shop = new Shop();

            // Construct and display vehicles
            builder = new ScooterBuilder();
            shop.Construct(builder);
            builder.Vehicle.Show();

            builder = new CarBuilder();
            shop.Construct(builder);
            builder.Vehicle.Show();

            builder = new MotorCycleBuilder();
            shop.Construct(builder);
            builder.Vehicle.Show();

            // Wait for user
            Console.ReadKey();
        }
    }

    /// <summary>
    /// The 'Director' class
    /// </summary>
    class Shop
    {
        // Builder uses a complex series of steps
        public void Construct(VehicleBuilder vehicleBuilder)
        {
            vehicleBuilder.BuildFrame();
            vehicleBuilder.BuildEngine();
            vehicleBuilder.BuildWheels();
            vehicleBuilder.BuildDoors();
        }
    }

    /// <summary>
    /// The 'Builder' abstract class
    /// </summary>
    abstract class VehicleBuilder
    {
        protected Vehicle vehicle;
```

```

// 获取车辆实例
public Vehicle 车辆
{
    get { return 车辆; }
}

// 抽象构建方法
public abstract void 构建车架();
public abstract void 构建引擎();
public abstract void 构建车轮();
public abstract void 构建车门();
}

/// <summary>
/// 'ConcreteBuilder1' 类
/// </summary>
class 摩托车建造者 : 车辆建造者
{
    public 摩托车建造者()
    {
        车辆 = new 车辆("摩托车");
    }

    public override void 构建车架()
    {
        车辆["车架"] = "摩托车车架";
    }

    public override void BuildEngine()
    {
        vehicle["engine"] = "500 cc";
    }

    public override void BuildWheels()
    {
        vehicle["wheels"] = "2";
    }

    public override void BuildDoors()
    {
        vehicle["doors"] = "0";
    }
}

/// <summary>
/// 'ConcreteBuilder2' 类
/// </summary>
class CarBuilder : VehicleBuilder
{
    public CarBuilder()
    {
        vehicle = new Vehicle("Car");
    }

    public override void 构建车架()
    {
        vehicle["frame"] = "汽车车架";
    }

    public override void BuildEngine()

```

```

// Gets vehicle instance
public Vehicle Vehicle
{
    get { return vehicle; }
}

// Abstract build methods
public abstract void BuildFrame();
public abstract void BuildEngine();
public abstract void BuildWheels();
public abstract void BuildDoors();
}

/// <summary>
/// The 'ConcreteBuilder1' class
/// </summary>
class MotorCycleBuilder : VehicleBuilder
{
    public MotorCycleBuilder()
    {
        vehicle = new Vehicle("MotorCycle");
    }

    public override void BuildFrame()
    {
        vehicle["frame"] = "MotorCycle Frame";
    }

    public override void BuildEngine()
    {
        vehicle["engine"] = "500 cc";
    }

    public override void BuildWheels()
    {
        vehicle["wheels"] = "2";
    }

    public override void BuildDoors()
    {
        vehicle["doors"] = "0";
    }
}

/// <summary>
/// The 'ConcreteBuilder2' class
/// </summary>
class CarBuilder : VehicleBuilder
{
    public CarBuilder()
    {
        vehicle = new Vehicle("Car");
    }

    public override void BuildFrame()
    {
        vehicle["frame"] = "Car Frame";
    }

    public override void BuildEngine()

```

```

{
vehicle["engine"] = "2500 cc";
}

public override void BuildWheels()
{
vehicle["wheels"] = "4";
}

public override void BuildDoors()
{
vehicle["doors"] = "4";
}

/// <summary>
/// 'ConcreteBuilder3' 类
/// </summary>
class ScooterBuilder : VehicleBuilder
{
    public ScooterBuilder()
    {
vehicle = new Vehicle("滑板车");
    }

    public override void 构建车架()
    {
vehicle["frame"] = "滑板车车架";
    }

    public override void BuildEngine()
    {
vehicle["engine"] = "50 cc";
    }

    public override void BuildWheels()
    {
vehicle["wheels"] = "2";
    }

    public override void BuildDoors()
    {
vehicle["doors"] = "0";
    }

    /// <summary>
    /// "产品"类
    /// </summary>
    class Vehicle
    {
        private string _vehicleType;
        private Dictionary<string, string> _parts =
            new Dictionary<string, string>();

        // 构造函数
        public Vehicle(string vehicleType)
        {
            this._vehicleType = vehicleType;
        }

        // 索引器
    }
}

```

```

{
    vehicle["engine"] = "2500 cc";
}

public override void BuildWheels()
{
    vehicle["wheels"] = "4";
}

public override void BuildDoors()
{
    vehicle["doors"] = "4";
}

/// <summary>
/// The 'ConcreteBuilder3' class
/// </summary>
class ScooterBuilder : VehicleBuilder
{
    public ScooterBuilder()
    {
        vehicle = new Vehicle("Scooter");
    }

    public override void BuildFrame()
    {
        vehicle["frame"] = "Scooter Frame";
    }

    public override void BuildEngine()
    {
        vehicle["engine"] = "50 cc";
    }

    public override void BuildWheels()
    {
        vehicle["wheels"] = "2";
    }

    public override void BuildDoors()
    {
        vehicle["doors"] = "0";
    }

    /// <summary>
    /// The 'Product' class
    /// </summary>
    class Vehicle
    {
        private string _vehicleType;
        private Dictionary<string, string> _parts =
            new Dictionary<string, string>();

        // Constructor
        public Vehicle(string vehicleType)
        {
            this._vehicleType = vehicleType;
        }

        // Indexer
    }
}

```

```

public string this[string key]
{
    get { return _parts[key]; }
    set { _parts[key] = value; }
}

public void Show()
{
    Console.WriteLine("-----");
    Console.WriteLine("车辆类型: {0}", _vehicleType);
    Console.WriteLine(" 车架 : {0}", _parts["frame"]);
    Console.WriteLine(" 引擎 : {0}", _parts["engine"]);
    Console.WriteLine(" 车轮数: {0}", _parts["wheels"]);
    Console.WriteLine(" 车门数 : {0}", _parts["doors"]);
}

}

```

输出

```

车辆类型: 滑板车 车架 : 滑板车车架
引擎 : 无
车轮数: 2
车门数 : 0

车辆类型: 汽车
车架 : 汽车车架
引擎 : 2500 cc
车轮数: 4
车门数 : 4

车辆类型: 摩托车
车架 : 摩托车车架
引擎 : 500 cc
车轮数: 2
车门数 : 0

```

```

public string this[string key]
{
    get { return _parts[key]; }
    set { _parts[key] = value; }
}

public void Show()
{
    Console.WriteLine("\n-----");
    Console.WriteLine(" Vehicle Type: {0}", _vehicleType);
    Console.WriteLine(" Frame : {0}", _parts["frame"]);
    Console.WriteLine(" Engine : {0}", _parts["engine"]);
    Console.WriteLine(" #Wheels: {0}", _parts["wheels"]);
    Console.WriteLine(" #Doors : {0}", _parts["doors"]);
}
}

```

Output

```

Vehicle Type: Scooter Frame : Scooter Frame
Engine : none
#Wheels: 2
#Doors : 0

Vehicle Type: Car
Frame : Car Frame
Engine : 2500 cc
#Wheels: 4
#Doors : 4

Vehicle Type: MotorCycle
Frame : MotorCycle Frame
Engine : 500 cc
#Wheels: 2
#Doors : 0

```

第134.5节：原型模式

使用原型实例指定要创建的对象类型，并通过复制该原型来创建新对象。

本例演示了原型模式，其中新的颜色对象是通过复制预先存在的、用户定义的同类型颜色来创建的。

```

using System;
using System.Collections.Generic;

namespace 四人帮.原型模式
{
    /// <summary>
    /// Real-World 原型设计模式的主应用启动类。
    /// </summary>
    class MainApp
    {
        /// <summary>

```

Section 134.5: Prototype Pattern

Specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.

In this example demonstrates the Prototype pattern in which new Color objects are created by copying pre-existing, user-defined Colors of the same type.

```

using System;
using System.Collections.Generic;

namespace GangOfFour.Prototype
{
    /// <summary>
    /// MainApp startup class for Real-World
    /// Prototype Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>

```

```

/// 控制台应用程序的入口点。
/// </summary>
static void Main()
{
    ColorManager 颜色管理器 = new ColorManager();

    // 使用标准颜色初始化
    颜色管理器["red"] = new Color(255, 0, 0);
    颜色管理器["green"] = new Color(0, 255, 0);
    颜色管理器["blue"] = new Color(0, 0, 255);

    // 用户添加个性化颜色
    颜色管理器["angry"] = new Color(255, 54, 0);
    颜色管理器["peace"] = new Color(128, 211, 128);
    颜色管理器["flame"] = new Color(211, 34, 20);

    // 用户克隆选定的颜色
    Color color1 = colormanager["red"].Clone() as Color;
    Color color2 = colormanager["peace"].Clone() as Color;
    Color color3 = colormanager["flame"].Clone() as Color;

    // 等待用户
    控制台.读取按键();
}

/// <summary>
/// "原型"抽象类
/// 
abstract class ColorPrototype
{
    public abstract ColorPrototype Clone();
}

/// <summary>
/// "具体原型"类
/// 
class Color : ColorPrototype
{
    private int _red;
    private int _green;
    private int _blue;

    // 构造函数
    public Color(int red, int green, int blue)
    {
        this._red = red;
        this._green = green;
        this._blue = blue;
    }

    // 创建一个浅拷贝
    public override ColorPrototype Clone()
    {
        Console.WriteLine(
            "克隆颜色 RGB: {0},{1},{2}",
            _red, _green, _blue);

        return this.MemberwiseClone() as ColorPrototype;
    }
}

```

```

/// Entry point into console application.
/// </summary>
static void Main()
{
    ColorManager colormanager = new ColorManager();

    // Initialize with standard colors
    colormanager["red"] = new Color(255, 0, 0);
    colormanager["green"] = new Color(0, 255, 0);
    colormanager["blue"] = new Color(0, 0, 255);

    // User adds personalized colors
    colormanager["angry"] = new Color(255, 54, 0);
    colormanager["peace"] = new Color(128, 211, 128);
    colormanager["flame"] = new Color(211, 34, 20);

    // User clones selected colors
    Color color1 = colormanager["red"].Clone() as Color;
    Color color2 = colormanager["peace"].Clone() as Color;
    Color color3 = colormanager["flame"].Clone() as Color;

    // Wait for user
    Console.ReadKey();
}

/// <summary>
/// The 'Prototype' abstract class
/// </summary>
abstract class ColorPrototype
{
    public abstract ColorPrototype Clone();
}

/// <summary>
/// The 'ConcretePrototype' class
/// </summary>
class Color : ColorPrototype
{
    private int _red;
    private int _green;
    private int _blue;

    // Constructor
    public Color(int red, int green, int blue)
    {
        this._red = red;
        this._green = green;
        this._blue = blue;
    }

    // Create a shallow copy
    public override ColorPrototype Clone()
    {
        Console.WriteLine(
            "Cloning color RGB: {0},{1},{2}",
            _red, _green, _blue);

        return this.MemberwiseClone() as ColorPrototype;
    }
}

```

```
/// <summary>
/// 原型管理器
/// </summary>
class ColorManager
{
    private Dictionary<string, ColorPrototype> _colors =
        new Dictionary<string, ColorPrototype>();

    // 索引器
    public ColorPrototype this[string key]
    {
        get { return _colors[key]; }
        set { _colors.Add(key, value); }
    }
}
```

输出：

```
克隆颜色 RGB: 255, 0, 0
克隆颜色 RGB: 128,211,128
克隆颜色 RGB: 211, 34, 20
```

```
/// <summary>
/// Prototype manager
/// </summary>
class ColorManager
{
    private Dictionary<string, ColorPrototype> _colors =
        new Dictionary<string, ColorPrototype>();

    // Indexer
    public ColorPrototype this[string key]
    {
        get { return _colors[key]; }
        set { _colors.Add(key, value); }
    }
}
```

Output:

```
Cloning color RGB: 255, 0, 0
Cloning color RGB: 128,211,128
Cloning color RGB: 211, 34, 20
```

第135章：实现装饰器设计模式

第135.1节：模拟自助餐厅

装饰器是结构型设计模式之一。它用于添加、移除或改变对象的行为。本文档将教你如何正确使用装饰器设计模式。

让我通过一个简单的例子向你解释它的思想。想象你现在在星巴克，一家著名的咖啡公司。你可以点任何你想要的咖啡——加奶油和糖，加奶油和配料，甚至更多组合！但所有饮品的基础都是咖啡——一种浓郁、苦涩的饮料，你可以对其进行修改。让我们编写一个简单的程序来模拟咖啡机。

首先，我们需要创建一个抽象类来描述我们的基础饮品：

```
public abstract class AbstractCoffee
{
    protected AbstractCoffee k = null;

    public AbstractCoffee(AbstractCoffee k)
    {
        this.k = k;
    }

    public abstract string ShowCoffee();
}
```

现在，让我们创建一些额外的配料，比如糖、牛奶和配料。创建的类必须实现AbstractCoffee——它们将对其进行装饰：

```
public class Milk : AbstractCoffee
{
    public Milk(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null)
            return k.ShowCoffee() + " with Milk";
        else return "Milk";
    }
}

public class Sugar : AbstractCoffee
{
    public Sugar(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null) return k.ShowCoffee() + " with Sugar";
        else return "Sugar";
    }
}

public class Topping : AbstractCoffee
{
    public Topping(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null) return k.ShowCoffee() + " with Topping";
    }
}
```

Chapter 135: Implementing Decorator Design Pattern

Section 135.1: Simulating cafeteria

Decorator is one of structural design patterns. It is used to add, remove or change behaviour of object. This document will teach you how to use Decorator DP properly.

Let me explain the idea of it to you on a simple example. Imagine you're now in Starbobs, famous coffee company. You can place an order for any coffee you want - with cream and sugar, with cream and topping and much more combinations! But, the base of all drinks is coffee - dark, bitter drink, you can modify. Let's write a simple program that simulates coffee machine.

First, we need to create an abstract class that describes our base drink:

```
public abstract class AbstractCoffee
{
    protected AbstractCoffee k = null;

    public AbstractCoffee(AbstractCoffee k)
    {
        this.k = k;
    }

    public abstract string ShowCoffee();
}
```

Now, let's create some extras, like sugar, milk and topping. Created classes must implement AbstractCoffee - they will decorate it:

```
public class Milk : AbstractCoffee
{
    public Milk(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null)
            return k.ShowCoffee() + " with Milk";
        else return "Milk";
    }
}

public class Sugar : AbstractCoffee
{
    public Sugar(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null) return k.ShowCoffee() + " with Sugar";
        else return "Sugar";
    }
}

public class Topping : AbstractCoffee
{
    public Topping(AbstractCoffee c) : base(c) { }

    public override string ShowCoffee()
    {
        if (k != null) return k.ShowCoffee() + " with Topping";
    }
}
```

```
        else return "Topping";
    }
}
```

现在我们可以制作我们最喜欢的咖啡了：

```
public class Program
{
    public static void Main(string[] args)
    {
AbstractCoffee coffee = null; //我们不能实例化抽象类
    coffee = new Topping(coffee); //传入 null
coffee = new Sugar(coffee); //传入 topping 实例
    coffee = new Milk(coffee); //传入 sugar
Console.WriteLine("Coffee with " + coffee.ShowCoffee());
    }
}
```

运行代码将产生以下输出：

Coffee with Topping with Sugar with Milk

```
        else return "Topping";
    }
}
```

Now we can create our favourite coffee:

```
public class Program
{
    public static void Main(string[] args)
    {
AbstractCoffee coffee = null; //we cant create instance of abstract class
    coffee = new Topping(coffee); //passing null
coffee = new Sugar(coffee); //passing topping instance
    coffee = new Milk(coffee); //passing sugar
Console.WriteLine("Coffee with " + coffee.ShowCoffee());
    }
}
```

Running the code will produce the following output:

Coffee with Topping with Sugar with Milk

第136章：实现享元设计模式

第136.1节：在RPG游戏中实现地图

享元模式是结构型设计模式之一。它通过与相似对象共享尽可能多的数据来减少所使用的内存量。本文档将教你如何正确使用享元设计模式。

让我通过一个简单的例子向你解释它的思想。假设你正在开发一款角色扮演游戏，需要加载一个包含一些角色的大型文件。例如：

- # 是草地。你可以在上面行走。
 - \$ 是起点
 - @ 是岩石。你不能在上面行走。
 - % 是宝箱

地图示例：

由于这些对象具有相似的特征，你不需要为每个地图格子创建单独的对象。我将向你展示如何使用享元模式。

让我们定义一个接口，供我们的格子实现：

```
public interface IField
{
    string 名称 { get; }
    char 标记 { get; }
    bool 可行走 { get; }
    FieldType 类型 { get; }
}
```

现在我们可以创建表示我们地块的类。我们还必须以某种方式识别它们（我使用了一个枚举）：

```
public enum FieldType
{
    草地,
    岩石,
    起点,
    宝箱
}
public class 草地 : IField
{
    public string 名称 { get { return "草地"; } }
    public char 标记 { get { return '#'; } }
}
```

Chapter 136: Implementing Flyweight Design Pattern

Section 136.1: Implementing map in RPG game

Flyweight is one of structural design patterns. It is used to decrease the amount of used memory by sharing as much data as possible with similar objects. This document will teach you how to use Flyweight DP properly.

Let me explain the idea of it to you on a simple example. Imagine you're working on a RPG game and you need to load huge file that contains some characters. For example:

- # is grass. You can walk on it.
 - \$ is starting point
 - @ is rock. You can't walk on it.
 - % is treasure chest

Sample of a map:

Since those objects have similar characteristic, you don't need to create separate object for each map field. I will show you how to use flyweight.

Let's define an interface which our fields will implement:

```
public interface IField
{
    string Name { get; }
    char Mark { get; }
    bool CanWalk { get; }
    FieldType Type { get; }
}
```

Now we can create classes that represent our fields. We also have to identify them somehow (I used an enumeration):

```
public enum FieldType
{
    GRASS,
    ROCK,
    START,
    CHEST
}
public class Grass : IField
{
    public string Name { get { return "Grass"; } }
    public char Mark { get { return '#'; } }
}
```

```

public bool 可行走 { get { return true; } }
public FieldType 类型 { get { return FieldType.草地; } }
}
public class 起点 : IField
{
    public string 名称 { get { return "起点"; } }
    public char 标记 { get { return '$'; } }
    public bool CanWalk { get { return true; } }
    public FieldType Type { get { return FieldType.START; } }
}
public class Rock : IField
{
    public string Name { get { return "Rock"; } }
    public char Mark { get { return '@'; } }
    public bool CanWalk { get { return false; } }
    public FieldType Type { get { return FieldType.ROCK; } }
}
public class TreasureChest : IField
{
    public string Name { get { return "Treasure Chest"; } }
    public char Mark { get { return '%'; } }
    public bool CanWalk { get { return true; } } // 你可以接近它
    public FieldType Type { get { return FieldType.CHEST; } }
}

```

正如我所说，我们不需要为每个字段创建单独的实例。我们必须创建一个字段仓库（repository）。享元设计模式的本质是：只有当我们需要且仓库中还不存在该对象时，才动态创建该对象；如果已经存在，则返回它。让我们编写一个简单的类来处理这个问题：

```

public class FieldRepository
{
    private List<IField> lstFields = new List<IField>();

    private IField AddField(FieldType type)
    {
        IField f;
        switch(type)
        {
            case FieldType.GRASS: f = new Grass(); break;
            case FieldType.ROCK: f = new Rock(); break;
            case FieldType.START: f = new StartingPoint(); break;
            case FieldType.CHEST:
                default: f = new TreasureChest(); break;
        }
        lstFields.Add(f); // 将其添加到仓库
        Console.WriteLine("创建了新的实例 {0}", f.Name);
        return f;
    }

    public IField GetField(FieldType type)
    {
        IField f = lstFields.Find(x => x.Type == type);
        if (f != null) return f;
        else return AddField(type);
    }
}

```

太好了！现在我们可以测试我们的代码了：

```

public class Program
{

```

```

public bool CanWalk { get { return true; } }
public FieldType Type { get { return FieldType.GRASS; } }
}
public class StartingPoint : IField
{
    public string Name { get { return "Starting Point"; } }
    public char Mark { get { return '$'; } }
    public bool CanWalk { get { return true; } }
    public FieldType Type { get { return FieldType.START; } }
}
public class Rock : IField
{
    public string Name { get { return "Rock"; } }
    public char Mark { get { return '@'; } }
    public bool CanWalk { get { return false; } }
    public FieldType Type { get { return FieldType.ROCK; } }
}
public class TreasureChest : IField
{
    public string Name { get { return "Treasure Chest"; } }
    public char Mark { get { return '%'; } }
    public bool CanWalk { get { return true; } } // you can approach it
    public FieldType Type { get { return FieldType.CHEST; } }
}

```

Like I said, we don't need to create separate instance for each field. We have to create a **repository** of fields. The essence of Flyweight DP is that we dynamically create an object only if we need it and it doesn't exist yet in our repo, or return it if it already exists. Let's write simple class that will handle this for us:

```

public class FieldRepository
{
    private List<IField> lstFields = new List<IField>();

    private IField AddField(FieldType type)
    {
        IField f;
        switch(type)
        {
            case FieldType.GRASS: f = new Grass(); break;
            case FieldType.ROCK: f = new Rock(); break;
            case FieldType.START: f = new StartingPoint(); break;
            case FieldType.CHEST:
                default: f = new TreasureChest(); break;
        }
        lstFields.Add(f); // add it to repository
        Console.WriteLine("Created new instance of {0}", f.Name);
        return f;
    }

    public IField GetField(FieldType type)
    {
        IField f = lstFields.Find(x => x.Type == type);
        if (f != null) return f;
        else return AddField(type);
    }
}

```

Great! Now we can test our code:

```

public class Program
{

```

```
public static void Main(string[] args)
{
FieldRepository f = new FieldRepository();
    IField grass = f.GetField(FieldType.GRASS);
    grass = f.GetField(FieldType.ROCK);
grass = f.GetField(FieldType.GRASS);
}
}
```

控制台中的结果应该是：

创建了一个新的草实例

创建了一个新的石头实例

但是如果我们想要获取两次草，为什么草只出现了一次？这是因为第一次调用GetField草时，实例在我们的repository中不存在，所以它被创建了，但下一次我们需要草时，它已经存在了，所以我们只返回它。

belindoc.com

```
public static void Main(string[] args)
{
    FieldRepository f = new FieldRepository();
    IField grass = f.GetField(FieldType.GRASS);
    grass = f.GetField(FieldType.ROCK);
    grass = f.GetField(FieldType.GRASS);
}
}
```

The result in the console should be:

Created a new instance of Grass

Created a new instance of Rock

But why grass appears only one time if we wanted to get it twice? That's because first time we call GetField grass instance does not exist in our **repository**, so it's created, but next time we need grass it already exist, so we only return it.

第137章： System.Management.Automation

第137.1节：调用简单同步管道

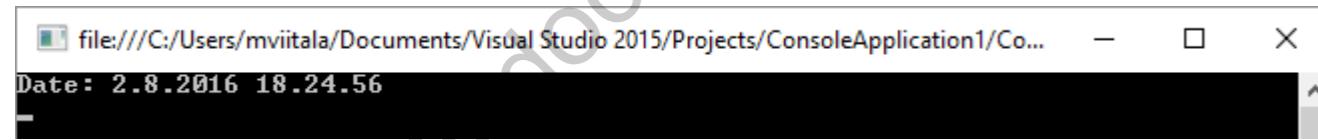
获取当前日期和时间。

```
public class Program
{
    static void Main()
    {
        // 创建空管道
PowerShell ps = PowerShell.Create();

        // 添加命令
ps.AddCommand("Get-Date");

        // 运行命令
Console.WriteLine("日期: {0}", ps.Invoke().First());

        Console.ReadLine();
    }
}
```



Chapter 137: System.Management.Automation

Section 137.1: Invoke simple synchronous pipeline

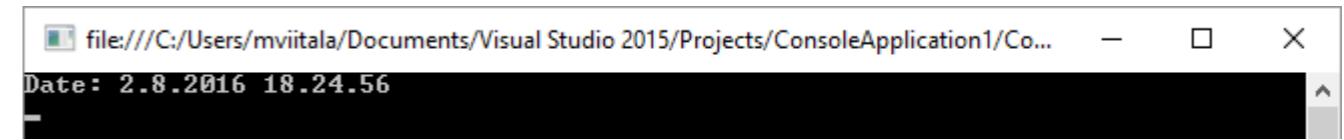
Get the current date and time.

```
public class Program
{
    static void Main()
    {
        // create empty pipeline
PowerShell ps = PowerShell.Create();

        // add command
ps.AddCommand("Get-Date");

        // run command(s)
Console.WriteLine("Date: {0}", ps.Invoke().First());

        Console.ReadLine();
    }
}
```



第138章： System.DirectoryServices.Protocols.LdapConnection

第138.1节：经过身份验证的SSL LDAP连接，SSL证书与反向DNS不匹配

为服务器和身份验证信息设置一些常量。假设使用LDAPv3，但更改起来也很简单。

```
// 身份验证及服务器名称。  
private const string LDAPUser = "cn=example:app:mygroup:accts,ou=Applications,dc=example,dc=com";  
private readonly char[] password = { 'p', 'a', 's', 's', 'w', 'o', 'r', 'd' };  
private const string TargetServer = "ldap.example.com";  
  
// 特定于您的公司。例如，可能以 "cn=manager" 开头，而不是 "ou=people"。  
private const string CompanyDN = "ou=people,dc=example,dc=com";
```

实际上使用三个部分创建连接：LdapDirectoryIdentifier（服务器）和 NetworkCredentials。

```
// 配置服务器和端口。使用 SSL 的 LDAP，即 LDAPS，端口为 636。  
// 如果没有 SSL，请不要使用 SSL 端口。  
LdapDirectoryIdentifier identifier = new LdapDirectoryIdentifier(TargetServer, 636);  
  
// 配置网络凭据（用户ID 和密码）  
var secureString = new SecureString();  
foreach (var character in password)  
    secureString.AppendChar(character);  
NetworkCredential creds = new NetworkCredential(LDAPUser, secureString);  
  
// 实际创建连接  
LdapConnection connection = new LdapConnection(identifier, creds)  
{  
    AuthType = AuthType.Basic,  
    SessionOptions =  
    {  
        ProtocolVersion = 3,  
        SecureSocketLayer = true  
    }  
};  
  
// 覆盖 SChannel 反向 DNS 查找。  
// 这让我们避免了“LDAP 服务器不可用”的异常  
// 可能是  
//     connection.SessionOptions.VerifyServerCertificate += { return true; };  
// 但某些证书验证可能是必要的。  
connection.SessionOptions.VerifyServerCertificate +=  
    (sender, certificate) => certificate.Subject.Contains(string.Format("CN={0}", TargetServer));
```

使用 LDAP 服务器，例如通过 userid 搜索某人，针对所有 objectClass 值。objectClass 的存在是为了演示复合搜索：& 符号是两个查询子句的布尔“与”操作符。

```
SearchRequest searchRequest = new SearchRequest(  
    CompanyDN,  
    string.Format((&(objectClass=*))(uid={0})), uid),  
    SearchScope.Subtree,
```

Chapter 138: System.DirectoryServices.Protocols.LdapConnection

Section 138.1: Authenticated SSL LDAP connection, SSL cert does not match reverse DNS

Set up some constants for the server and authentication information. Assuming LDAPv3, but it's easy enough to change that.

```
// Authentication, and the name of the server.  
private const string LDAPUser = "cn=example:app:mygroup:accts,ou=Applications,dc=example,dc=com";  
private readonly char[] password = { 'p', 'a', 's', 's', 'w', 'o', 'r', 'd' };  
private const string TargetServer = "ldap.example.com";  
  
// Specific to your company. Might start "cn=manager" instead of "ou=people", for example.  
private const string CompanyDN = "ou=people,dc=example,dc=com";
```

Actually create the connection with three parts: an LdapDirectoryIdentifier (the server), and NetworkCredentials.

```
// Configure server and port. LDAP w/ SSL, aka LDAPS, uses port 636.  
// If you don't have SSL, don't give it the SSL port.  
LdapDirectoryIdentifier identifier = new LdapDirectoryIdentifier(TargetServer, 636);  
  
// Configure network credentials (userid and password)  
var secureString = new SecureString();  
foreach (var character in password)  
    secureString.AppendChar(character);  
NetworkCredential creds = new NetworkCredential(LDAPUser, secureString);  
  
// Actually create the connection  
LdapConnection connection = new LdapConnection(identifier, creds)  
{  
    AuthType = AuthType.Basic,  
    SessionOptions =  
    {  
        ProtocolVersion = 3,  
        SecureSocketLayer = true  
    }  
};  
  
// Override SChannel reverse DNS lookup.  
// This gets us past the "The LDAP server is unavailable." exception  
// Could be  
//     connection.SessionOptions.VerifyServerCertificate += { return true; };  
// but some certificate validation is probably good.  
connection.SessionOptions.VerifyServerCertificate +=  
    (sender, certificate) => certificate.Subject.Contains(string.Format("CN={0}", TargetServer));
```

Use the LDAP server, e.g. search for someone by userid for all objectClass values. The objectClass is present to demonstrate a compound search: The ampersand is the boolean "and" operator for the two query clauses.

```
SearchRequest searchRequest = new SearchRequest(  
    CompanyDN,  
    string.Format((&(objectClass=*))(uid={0})), uid),  
    SearchScope.Subtree,
```

```

    null
);

// 查看你的结果
foreach (SearchResultEntry entry in searchResponse.Entries) {
    // 执行某些操作
}

```

第138.2节：超级简单的匿名LDAP

假设使用LDAPv3，但修改起来也很简单。这是匿名、未加密的LDAPv3 LdapConnection 创建。

```
private const string TargetServer = "ldap.example.com";
```

实际上使用三个部分创建连接：LdapDirectoryIdentifier（服务器）和 NetworkCredentials。

```

// 配置服务器和凭据
LdapDirectoryIdentifier identifier = new LdapDirectoryIdentifier(TargetServer);
NetworkCredential creds = new NetworkCredential();
LdapConnection connection = new LdapConnection(identifier, creds)
{
    AuthType=AuthType.Anonymous,
    SessionOptions =
    {
        ProtocolVersion = 3
    }
};

```

要使用该连接，类似这样的操作会获取姓氏为史密斯的人

```
SearchRequest searchRequest = new SearchRequest("dn=example,dn=com", "(sn=Smith)",
SearchScope.Subtree,null);
```

```

    null
);

// Look at your results
foreach (SearchResultEntry entry in searchResponse.Entries) {
    // do something
}

```

Section 138.2: Super Simple anonymous LDAP

Assuming LDAPv3, but it's easy enough to change that. This is anonymous, unencrypted LDAPv3 LdapConnection creation.

```
private const string TargetServer = "ldap.example.com";
```

Actually create the connection with three parts: an LdapDirectoryIdentifier (the server), and NetworkCredentials.

```

// Configure server and credentials
LdapDirectoryIdentifier identifier = new LdapDirectoryIdentifier(TargetServer);
NetworkCredential creds = new NetworkCredential();
LdapConnection connection = new LdapConnection(identifier, creds)
{
    AuthType=AuthType.Anonymous,
    SessionOptions =
    {
        ProtocolVersion = 3
    }
};

```

To use the connection, something like this would get people with the surname Smith

```
SearchRequest searchRequest = new SearchRequest("dn=example,dn=com", "(sn=Smith)",
SearchScope.Subtree,null);
```

第139章：C# 认证处理程序

第139.1节：认证处理程序

```
public class AuthenticationHandler : DelegatingHandler
{
    /// <summary>
    /// 保存包含令牌的请求头名称。
    /// </summary>
    private const string securityToken = "__RequestAuthToken";

    /// <summary>
    /// 默认重写方法，执行认证操作。
    /// </summary>
    /// <param name="request">HTTP请求消息。</param>/// <param name="cancellationToken">取消令牌。</param>/// <returns>异步返回类型为<see cref=" HttpResponseMessage"/> 类的HTTP响应消息。</returns>
    protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
    CancellationToken cancellationToken)
    {
        if (request.Headers.Contains(securityToken))
        {
            bool authorized = Authorize(request);
            if (!authorized)
            {
                return ApiHttpUtility.FromResult(request, false, HttpStatusCode.Unauthorized,
                MessageTypes.Error, Resource.UnAuthenticatedUser);
            }
            else
            {
                return ApiHttpUtility.FromResult(request, false, HttpStatusCode.BadRequest,
                MessageTypes.Error, Resource.UnAuthenticatedUser);
            }
        }

        return base.SendAsync(request, cancellationToken);
    }

    /// <summary>
    /// 通过验证令牌授权用户。
    /// </summary>
    /// <param name="requestMessage">授权上下文。</param>/// <returns>返回一个值，指示当前请求是否已通过身份验证。</returns>
    private bool Authorize(HttpRequestMessage requestMessage)
    {
        try
        {
            HttpRequest request = HttpContext.Current.Request;
            string token = request.Headers[securityToken];
            return SecurityUtility.IsTokenValid(token, request.UserAgent,
            HttpContext.Current.Server.MapPath("~/Content/"), requestMessage);
        }
        catch (Exception)
        {
            return false;
        }
    }
}
```

Chapter 139: C# Authentication handler

Section 139.1: Authentication handler

```
public class AuthenticationHandler : DelegatingHandler
{
    /// <summary>
    /// Holds request's header name which will contains token.
    /// </summary>
    private const string securityToken = "__RequestAuthToken";

    /// <summary>
    /// Default overridden method which performs authentication.
    /// </summary>
    /// <param name="request">Http request message.</param>
    /// <param name="cancellationToken">Cancellation token.</param>
    /// <returns>Returns http response message of type <see cref=" HttpResponseMessage"/> class asynchronously.</returns>
    protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
    CancellationToken cancellationToken)
    {
        if (request.Headers.Contains(securityToken))
        {
            bool authorized = Authorize(request);
            if (!authorized)
            {
                return ApiHttpUtility.FromResult(request, false, HttpStatusCode.Unauthorized,
                MessageTypes.Error, Resource.UnAuthenticatedUser);
            }
            else
            {
                return ApiHttpUtility.FromResult(request, false, HttpStatusCode.BadRequest,
                MessageTypes.Error, Resource.UnAuthenticatedUser);
            }
        }

        return base.SendAsync(request, cancellationToken);
    }

    /// <summary>
    /// Authorize user by validating token.
    /// </summary>
    /// <param name="requestMessage">Authorization context.</param>
    /// <returns>Returns a value indicating whether current request is authenticated or not.</returns>
    private bool Authorize(HttpRequestMessage requestMessage)
    {
        try
        {
            HttpRequest request = HttpContext.Current.Request;
            string token = request.Headers[securityToken];
            return SecurityUtility.IsTokenValid(token, request.UserAgent,
            HttpContext.Current.Server.MapPath("~/Content/"), requestMessage);
        }
        catch (Exception)
        {
            return false;
        }
    }
}
```

belindoc.com

第140章：指针

第140.1节：用于数组访问的指针

本示例演示了如何使用指针实现类似C语言的C#数组访问。

```
unsafe
{
    var buffer = new int[1024];
    fixed (int* p = &buffer[0])
    {
        for (var i = 0; i < buffer.Length; i++)
        {
            *(p + i) = i;
        }
    }
}
```

使用`unsafe`关键字是必须的，因为指针访问不会发出通常访问C#数组时所发出的边界检查。

使用`fixed`关键字告诉C#编译器发出指令，以异常安全的方式固定对象。固定是必要的，以确保垃圾回收器不会移动数组在内存中的位置，因为那样会使指向数组内部的指针失效。

第140.2节：指针运算

指针的加法和减法与整数不同。当指针递增或递减时，它所指向的地址会根据引用类型的大小增加或减少。

例如，类型`int`（`System.Int32`的别名）大小为4。如果一个`int`可以存储在地址0，后续的`int`可以存储在地址4，依此类推。代码示例：

```
var ptr = (int*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // 输出0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // 输出4
ptr++;
Console.WriteLine(new IntPtr(ptr)); // 输出8
```

类似地，类型`long`（`System.Int64`的别名）大小为8。如果一个`long`可以存储在地址0，后续的`long`可以存储在地址8，依此类推。代码示例：

```
var ptr = (long*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // 输出0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // 输出8
ptr++;
Console.WriteLine(new IntPtr(ptr)); // 输出16
```

类型`void`是特殊的，`void`指针也很特殊，当类型未知或不重要时，它们被用作通用指针。由于其大小无关的特性，`void`指针不能进行递增或递减：

```
var ptr = (void*)IntPtr.Zero;
```

Chapter 140: Pointers

Section 140.1: Pointers for array access

This example demonstrates how pointers can be used for C-like access to C# arrays.

```
unsafe
{
    var buffer = new int[1024];
    fixed (int* p = &buffer[0])
    {
        for (var i = 0; i < buffer.Length; i++)
        {
            *(p + i) = i;
        }
    }
}
```

The `unsafe` keyword is required because pointer access will not emit any bounds checks that are normally emitted when accessing C# arrays the regular way.

The `fixed` keyword tells the C# compiler to emit instructions to pin the object in an exception-safe way. Pinning is required to ensure that the garbage collector will not move the array in memory, as that would invalidate any pointers pointing within the array.

Section 140.2: Pointer arithmetic

Addition and subtraction in pointers works differently from integers. When a pointer is incremented or decremented, the address it points to is increased or decreased by the size of the referent type.

For example, the type `int` (alias for `System.Int32`) has a size of 4. If an `int` can be stored in address 0, the subsequent `int` can be stored in address 4, and so on. In code:

```
var ptr = (int*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // prints 0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 4
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 8
```

Similarly, the type `long` (alias for `System.Int64`) has a size of 8. If a `long` can be stored in address 0, the subsequent `long` can be stored in address 8, and so on. In code:

```
var ptr = (long*)IntPtr.Zero;
Console.WriteLine(new IntPtr(ptr)); // prints 0
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 8
ptr++;
Console.WriteLine(new IntPtr(ptr)); // prints 16
```

The type `void` is special and `void` pointers are also special and they are used as catch-all pointers when the type isn't known or doesn't matter. Due to their size-agnostic nature, `void` pointers cannot be incremented or decremented:

```
var ptr = (void*)IntPtr.Zero;
```

```
Console.WriteLine(new IntPtr(ptr));
ptr++; // 编译时错误
Console.WriteLine(new IntPtr(ptr));
ptr++; // 编译时错误
Console.WriteLine(new IntPtr(ptr));
```

第140.3节：星号是类型的一部分

在C和C++中，指针变量声明中的星号是被声明表达式的一部分。在C#中，声明中的星号是类型的一部分。

在C、C++和C#中，以下代码片段声明了一个int指针：

```
int* a;
```

在C和C++中，以下代码片段声明了一个int指针和一个int变量。在C#中，它声明了两个int指针：

```
int* a, b;
```

在C和C++中，以下代码片段声明了两个int指针。在C#中，这是无效的：

```
int *a, *b;
```

第140.4节：void*

C#继承了C和C++中void*作为类型无关且大小无关指针的用法。

```
void* ptr;
```

任何指针类型都可以通过隐式转换赋值给void*：

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
```

反向转换需要显式转换：

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
int* p2 = (int*)ptr;
```

第140.5节：使用 -> 进行成员访问

C# 继承了 C 和 C++ 中使用符号 -> 作为通过类型指针访问实例成员的方式。

考虑以下结构体：

```
struct Vector2
{
    public int X;
    public int Y;
}
```

这是使用->访问其成员的示例：

```
Console.WriteLine(new IntPtr(ptr));
ptr++; // compile-time error
Console.WriteLine(new IntPtr(ptr));
ptr++; // compile-time error
Console.WriteLine(new IntPtr(ptr));
```

Section 140.3: The asterisk is part of the type

In C and C++, the asterisk in the declaration of a pointer variable is *part of the expression* being declared. In C#, the asterisk in the declaration is *part of the type*.

In C, C++ and C#, the following snippet declares an `int` pointer:

```
int* a;
```

In C and C++, the following snippet declares an `int` pointer and an `int` variable. In C#, it declares two `int` pointers:

```
int* a, b;
```

In C and C++, the following snippet declares two `int` pointers. In C#, it is invalid:

```
int *a, *b;
```

Section 140.4: void*

C# inherits from C and C++ the usage of `void*` as a type-agnostic and size-agnostic pointer.

```
void* ptr;
```

Any pointer type can be assigned to `void*` using an implicit conversion:

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
```

The reverse requires an explicit conversion:

```
int* p1 = (int*)IntPtr.Zero;
void* ptr = p1;
int* p2 = (int*)ptr;
```

Section 140.5: Member access using ->

C# inherits from C and C++ the usage of the symbol `->` as a means of accessing the members of an instance through a typed pointer.

Consider the following struct:

```
struct Vector2
{
    public int X;
    public int Y;
}
```

This is an example of the usage of `->` to access its members:

```
Vector2 v;  
v.X = 5;  
v.Y = 10;  
  
Vector2* ptr = &v;  
int x = ptr->X;  
int y = ptr->Y;  
string s = ptr->ToString();  
  
Console.WriteLine(x); // 输出 5  
Console.WriteLine(y); // 输出 10  
Console.WriteLine(s); // 输出 Vector2
```

第140.6节：泛型指针

类型必须满足的支持指针的条件（参见备注）无法用泛型约束来表达。因此，任何尝试声明指向通过泛型类型参数提供的类型的指针的操作都会失败。

```
void P<T>(T obj)  
    where T : struct  
{  
    T* ptr = &obj; // 编译时错误  
}
```

```
Vector2 v;  
v.X = 5;  
v.Y = 10;  
  
Vector2* ptr = &v;  
int x = ptr->X;  
int y = ptr->Y;  
string s = ptr->ToString();  
  
Console.WriteLine(x); // prints 5  
Console.WriteLine(y); // prints 10  
Console.WriteLine(s); // prints Vector2
```

Section 140.6: Generic pointers

The criteria that a type must satisfy in order to support pointers (see *Remarks*) cannot be expressed in terms of generic constraints. Therefore, any attempt to declare a pointer to a type provided through a generic type parameter will fail.

```
void P<T>(T obj)  
    where T : struct  
{  
    T* ptr = &obj; // compile-time error  
}
```

第141章：指针与不安全代码

第141.1节：不安全代码简介

C# 允许在用unsafe修饰的代码块函数中使用指针变量。unsafe 代码或非托管代码是使用指针变量的代码块。

指针是一个变量，其值是另一个变量的地址，即内存位置的直接地址。类似于任何变量或常量，必须先声明指针，才能使用它来存储任何变量地址。

指针声明的一般形式是：

```
类型 *变量名;
```

以下是有效的指针声明：

```
int *ip; /* 指向整数的指针 */
double *dp; /* 指向双精度浮点数的指针 */
float *fp; /* 指向浮点数的指针 */
char *ch; /* 指向字符的指针 */
```

下面的示例演示了在C#中使用指针，使用了unsafe修饰符：

```
using System;
namespace UnsafeCodeApplication
{
    class 程序
    {
        static unsafe void Main(string[] args)
        {
            int var = 20;
            int* p = &var;
            Console.WriteLine("数据是: {0}", var);
            Console.WriteLine("地址是: {0}", (int)p);
            Console.ReadKey();
        }
    }
}
```

当上述代码被编译并执行时，产生以下结果：

```
数据是: 20
地址是: 99215364
```

你也可以只将代码的一部分声明为不安全，而不是将整个方法声明为不安全：

```
// 安全代码
unsafe
{
    // 你可以在这里使用指针
}
// 安全代码
```

Chapter 141: Pointers & Unsafe Code

Section 141.1: Introduction to unsafe code

C# allows using pointer variables in a function of code block when it is marked by the `unsafe` modifier. The unsafe code or the unmanaged code is a code block that uses a pointer variable.

A pointer is a variable whose value is the address of another variable i.e., the direct address of the memory location. similar to any variable or constant, you must declare a pointer before you can use it to store any variable address.

The general form of a pointer declaration is:

```
type *var-name;
```

Following are valid pointer declarations:

```
int *ip; /* pointer to an integer */
double *dp; /* pointer to a double */
float *fp; /* pointer to a float */
char *ch /* pointer to a character */
```

The following example illustrates use of pointers in C#, using the unsafe modifier:

```
using System;
namespace UnsafeCodeApplication
{
    class Program
    {
        static unsafe void Main(string[] args)
        {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0}", var);
            Console.WriteLine("Address is: {0}", (int)p);
            Console.ReadKey();
        }
    }
}
```

When the above code was compiled and executed, it produces the following result:

```
Data is: 20
Address is: 99215364
```

Instead of declaring an entire method as unsafe, you can also declare a part of the code as unsafe:

```
// safe code
unsafe
{
    // you can use pointers here
}
// safe code
```

第141.2节：使用指针访问数组元素

在C#中，数组名和指向与数组数据类型相同的数据的指针，不是相同的变量类型。例如，`int *p` 和 `int[] p`，不是相同类型。你可以递增指针变量 `p`，因为它在内存中不是固定的，但数组地址在内存中是固定的，且你不能递增它。

因此，如果你需要像传统的C或C++那样使用指针变量访问数组数据，你需要使用`fixed`关键字固定指针。

以下示例演示了这一点：

```
using System;
namespace UnsafeCodeApplication
{
    class TestPointer
    {
        public unsafe static void Main()
        {
            int[] list = {10, 100, 200};
            fixed(int *ptr = list)

            /* 让我们通过指针获取数组地址 */
            for (int i = 0; i < 3; i++)
            {
                Console.WriteLine("list[{0}] 的地址={1}", i, (int)(ptr + i));
                Console.WriteLine("list[{0}] 的值={1}", i, *(ptr + i));
            }

            Console.ReadKey();
        }
    }
}
```

当上述代码被编译并执行时，产生以下结果：

```
list[0]的地址 = 31627168
list[0]的值 = 10
list[1]的地址 = 31627172
list[1]的值 = 100
列表[2]的地址 = 31627176
列表[2]的值 = 200
```

第141.3节：编译不安全代码

要编译不安全代码，必须在命令行编译器中指定`/unsafe`命令行开关。

例如，要从命令行编译包含不安全代码的名为`prog1.cs`的程序，输入命令：

```
csc /unsafe prog1.cs
```

如果使用Visual Studio集成开发环境，则需要在项目属性中启用不安全代码的使用。

Section 141.2: Accessing Array Elements Using a Pointer

In C#, an array name and a pointer to a data type same as the array data, are not the same variable type. For example, `int *p` and `int[] p`, are not same type. You can increment the pointer variable `p` because it is not fixed in memory but an array address is fixed in memory, and you can't increment that.

Therefore, if you need to access an array data using a pointer variable, as we traditionally do in C, or C++, you need to fix the pointer using the `fixed` keyword.

The following example demonstrates this:

```
using System;
namespace UnsafeCodeApplication
{
    class TestPointer
    {
        public unsafe static void Main()
        {
            int[] list = {10, 100, 200};
            fixed(int *ptr = list)

            /* let us have array address in pointer */
            for (int i = 0; i < 3; i++)
            {
                Console.WriteLine("Address of list[{0}]={1}", i, (int)(ptr + i));
                Console.WriteLine("Value of list[{0}]={1}", i, *(ptr + i));
            }

            Console.ReadKey();
        }
    }
}
```

When the above code was compiled and executed, it produces the following result:

```
Address of list[0] = 31627168
Value of list[0] = 10
Address of list[1] = 31627172
Value of list[1] = 100
Address of list[2] = 31627176
Value of list[2] = 200
```

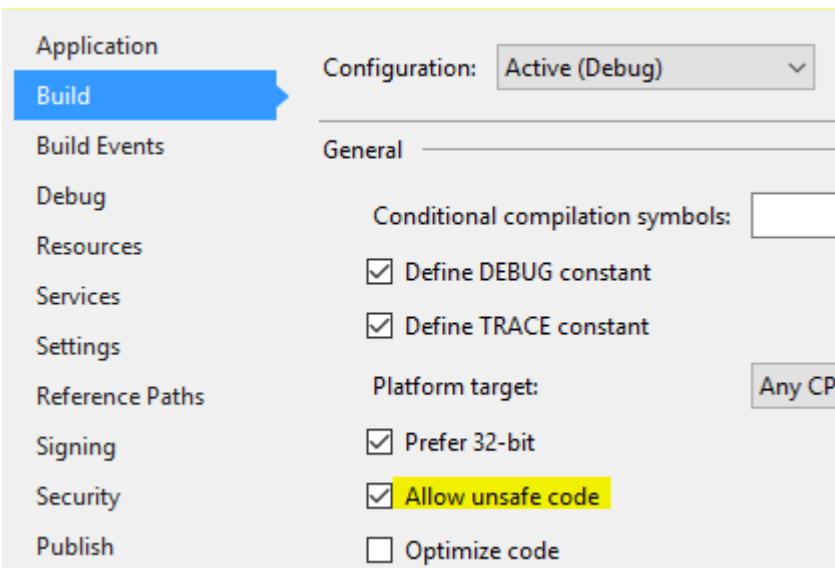
Section 141.3: Compiling Unsafe Code

For compiling unsafe code, you have to specify the `/unsafe` command-line switch with command-line compiler.

For example, to compile a program named `prog1.cs` containing unsafe code, from command line, give the command:

```
csc /unsafe prog1.cs
```

If you are using Visual Studio IDE then you need to enable use of unsafe code in the project properties.



操作步骤如下：

- 在解决方案资源管理器中双击属性节点以打开项目属性。
- 点击“生成”选项卡。
- 选择“允许不安全代码”选项

第141.4节：使用指针检索数据值

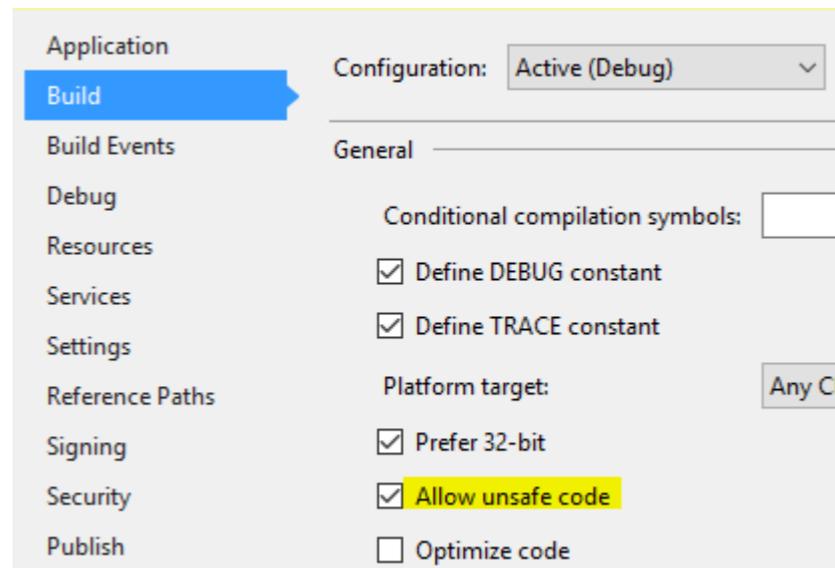
您可以使用ToString()方法检索指针变量所引用位置存储的数据。以下示例演示了这一点：

```
using System;
namespace UnsafeCodeApplication
{
    class 程序
    {
        public static void Main()
        {
            unsafe
            {
                int var = 20;
                int* p = &var;
                Console.WriteLine("数据是: {0} " , var);
                Console.WriteLine("数据是: {0} " , p->ToString());
                Console.WriteLine("地址是: {0} " , (int)p);
            }

            Console.ReadKey();
        }
    }
}
```

当上述代码被编译并执行时，产生以下结果：

```
数据是: 20
数据是: 20
地址是: 77128984
```



To do this:

- Open project properties by double clicking the properties node in the Solution Explorer.
- Click on the Build tab.
- Select the option "Allow unsafe code"

Section 141.4: Retrieving the Data Value Using a Pointer

You can retrieve the data stored at the located referenced by the pointer variable, using the ToString() method. The following example demonstrates this:

```
using System;
namespace UnsafeCodeApplication
{
    class Program
    {
        public static void Main()
        {
            unsafe
            {
                int var = 20;
                int* p = &var;
                Console.WriteLine("Data is: {0} " , var);
                Console.WriteLine("Data is: {0} " , p->ToString());
                Console.WriteLine("Address is: {0} " , (int)p);
            }

            Console.ReadKey();
        }
    }
}
```

When the above code was compiled and executed, it produces the following result:

```
Data is: 20
Data is: 20
Address is: 77128984
```

第141.5节：将指针作为参数传递给方法

您可以将指针变量作为参数传递给方法。以下示例说明了这一点：

```
using System;
namespace UnsafeCodeApplication
{
    class TestPointer
    {
        public unsafe void swap(int* p, int *q)
        {
            int temp = *p;
            *p = *q;
            *q = temp;
        }

        public unsafe static void Main()
        {
TestPointer p = new TestPointer();
            int var1 = 10;
            int var2 = 20;
            int* x = &var1;
            int* y = &var2;

Console.WriteLine("Before Swap: var1:{0}, var2: {1}", var1, var2);
            p.swap(x, y);

Console.WriteLine("After Swap: var1:{0}, var2: {1}", var1, var2);
            Console.ReadKey();
        }
    }
}
```

当上述代码被编译并执行时，产生以下结果：

```
Before Swap: var1: 10, var2: 20
After Swap: var1: 20, var2: 10
```

Section 141.5: Passing Pointers as Parameters to Methods

You can pass a pointer variable to a method as parameter. The following example illustrates this:

```
using System;
namespace UnsafeCodeApplication
{
    class TestPointer
    {
        public unsafe void swap(int* p, int *q)
        {
            int temp = *p;
            *p = *q;
            *q = temp;
        }

        public unsafe static void Main()
        {
TestPointer p = new TestPointer();
            int var1 = 10;
            int var2 = 20;
            int* x = &var1;
            int* y = &var2;

Console.WriteLine("Before Swap: var1:{0}, var2: {1}", var1, var2);
            p.swap(x, y);

Console.WriteLine("After Swap: var1:{0}, var2: {1}", var1, var2);
            Console.ReadKey();
        }
    }
}
```

When the above code is compiled and executed, it produces the following result:

```
Before Swap: var1: 10, var2: 20
After Swap: var1: 20, var2: 10
```

第142章：如何使用C#结构体创建联合类型（类似于C语言的联合体）

第142.1节：C#中的C风格联合体

联合类型在多种语言中使用，比如C语言，用于包含几种可以“重叠”的不同类型。换句话说，它们可能包含不同的字段，这些字段都从相同的内存偏移量开始，即使它们可能具有不同的长度和类型。这既节省了内存，又实现了自动转换的好处。

以IP地址为例。内部，IP地址表示为一个整数，但有时我们希望访问不同的字节部分，如Byte1.Byte2.Byte3.Byte4。这适用于任何值类型，无论是像Int32或long这样的基本类型，还是你自己定义的其他结构体。

我们可以通过使用显式布局结构体在C#中实现相同的效果。

```
using System;
using System.Runtime.InteropServices;

// 该结构体需要标注为“显式布局”
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
{
    // "FieldOffset"表示该整数的起始偏移量，单位为字节。
    // sizeof(int) 4, sizeof(byte) = 1
    [FieldOffset(0)] public int Address;
    [FieldOffset(0)] public byte Byte1;
    [FieldOffset(1)] public byte Byte2;
    [FieldOffset(2)] public byte Byte3;
    [FieldOffset(3)] public byte Byte4;

    public IpAddress(int address) : this()
    {
        // 当我们初始化整数时，字节也会随之改变。
        Address = address;
    }

    // 现在我们可以使用显式布局分别访问
    // 各个字节，而无需进行任何转换。
    public override string ToString() => $"{Byte1}.{Byte2}.{Byte3}.{Byte4}";
}
```

定义了我们的结构体后，我们可以像在C语言中使用联合体一样使用它。例如，创建一个作为随机整数的IP地址，然后通过将地址的第一个部分从'A.B.C.D'修改为'100.B.C.D'来更改它：

```
var ip = new IpAddress(new Random().Next());
Console.WriteLine($"{ip} = {ip.Address}");
ip.Byte1 = 100;
Console.WriteLine($"{ip} = {ip.Address}");
```

输出：

```
75.49.5.32 = 537211211
100.49.5.32 = 537211236
```

[查看演示](#)

Chapter 142: How to use C# Structs to create a Union type (Similar to C Unions)

Section 142.1: C-Style Unions in C#

Union types are used in several languages, like C-language, to contain several different types which can "overlap". In other words, they might contain different fields all of which start at the same memory offset, even when they might have different lengths and types. This has the benefit of both saving memory, and doing automatic conversion. Think of an IP address, as an example. Internally, an IP address is represented as an integer, but sometimes we want to access the different Byte component, as in Byte1.Byte2.Byte3.Byte4. This works for any value types, be it primitives like Int32 or long, or for other structs that you define yourself.

We can achieve the same effect in C# by using Explicit Layout Structs.

```
using System;
using System.Runtime.InteropServices;

// The struct needs to be annotated as "Explicit Layout"
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
{
    // The "FieldOffset" means that this Integer starts, an offset in bytes.
    // sizeof(int) 4, sizeof(byte) = 1
    [FieldOffset(0)] public int Address;
    [FieldOffset(0)] public byte Byte1;
    [FieldOffset(1)] public byte Byte2;
    [FieldOffset(2)] public byte Byte3;
    [FieldOffset(3)] public byte Byte4;

    public IpAddress(int address) : this()
    {
        // When we init the Int, the Bytes will change too.
        Address = address;
    }

    // Now we can use the explicit layout to access the
    // bytes separately, without doing any conversion.
    public override string ToString() => $"{Byte1}.{Byte2}.{Byte3}.{Byte4}";
}
```

Having defined out Struct in this way, we can use it as we would use a Union in C. For example, let's create an IP address as a Random Integer and then modify the first token in the address to '100', by changing it from 'A.B.C.D' to '100.B.C.D':

```
var ip = new IpAddress(new Random().Next());
Console.WriteLine($"{ip} = {ip.Address}");
ip.Byte1 = 100;
Console.WriteLine($"{ip} = {ip.Address}");
```

Output:

```
75.49.5.32 = 537211211
100.49.5.32 = 537211236
```

[View Demo](#)

第142.2节：C#中的联合类型也可以包含结构体字段

除了基本类型外，C#中的显式布局结构体（联合体）也可以包含其他结构体。只要字段是值类型而非引用类型，就可以包含在联合体中：

```
using System;
using System.Runtime.InteropServices;

// 该结构体需要标注为“显式布局”
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
{
    // 与上面示例中相同的IpAddress定义
}

// 现在让我们看看是否能将整个 URL 放入一行

// 让我们定义一个简短的枚举来保存协议
enum 协议 : short { Http,Https,Ftp,Sftp,Tcp }

// Service 结构体将包含地址、端口和协议
[StructLayout(LayoutKind.Explicit)]
struct Service
{
    [FieldOffset(0)] public IpAddress 地址;
    [FieldOffset(4)] public ushort 端口;
    [FieldOffset(6)] public 协议 应用协议;
    [FieldOffset(0)] public long 负载;

    public Service(IpAddress 地址, ushort 端口, 协议 协议)
    {
        负载 = 0;
        地址 = 地址;
        端口 = 端口;
        应用协议 = 协议;
    }

    public Service(long 负载)
    {
        地址 = new IpAddress(0);
        端口 = 80;
        应用协议 = 协议.Http;
        负载 = 负载;
    }

    public Service Copy() => new Service(Payload);

    public override string ToString() => $"{AppProtocol}/{Address}:{Port}/";
}
```

我们现在可以验证整个 Service 联合体是否适合一个 long (8 字节) 的大小。

```
var ip = new IpAddress(new Random().Next());
Console.WriteLine($"大小: {Marshal.SizeOf(ip)} 字节。值: {ip.Address} = {ip}.");

var s1 = new Service(ip, 8080, Protocol.Https);
var s2 = new Service(s1.Payload);
s2.Address.Byte1 = 100;
s2.AppProtocol = Protocol.Ftp;
```

Section 142.2: Union Types in C# can also contain Struct fields

Apart from primitives, the Explicit Layout structs (Unions) in C#, can also contain other Structs. As long as a field is a Value type and not a Reference, it can be contained in a Union:

```
using System;
using System.Runtime.InteropServices;

// The struct needs to be annotated as "Explicit Layout"
[StructLayout(LayoutKind.Explicit)]
struct IpAddress
{
    // Same definition of IpAddress, from the example above
}

// Now let's see if we can fit a whole URL into a long

// Let's define a short enum to hold protocols
enum Protocol : short { Http,Https,Ftp,Sftp,Tcp }

// The Service struct will hold the Address, the Port and the Protocol
[StructLayout(LayoutKind.Explicit)]
struct Service
{
    [FieldOffset(0)] public IpAddress Address;
    [FieldOffset(4)] public ushort Port;
    [FieldOffset(6)] public Protocol AppProtocol;
    [FieldOffset(0)] public long Payload;

    public Service(IpAddress address, ushort port, Protocol protocol)
    {
        Payload = 0;
        Address = address;
        Port = port;
        AppProtocol = protocol;
    }

    public Service(long payload)
    {
        Address = new IpAddress(0);
        Port = 80;
        AppProtocol = Protocol.Http;
        Payload = payload;
    }

    public Service Copy() => new Service(Payload);

    public override string ToString() => $"{AppProtocol}/{Address}:{Port}/";
}
```

We can now verify that the whole Service Union fits into the size of a long (8 bytes).

```
var ip = new IpAddress(new Random().Next());
Console.WriteLine($"Size: {Marshal.SizeOf(ip)} bytes. Value: {ip.Address} = {ip}.");

var s1 = new Service(ip, 8080, Protocol.Https);
var s2 = new Service(s1.Payload);
s2.Address.Byte1 = 100;
s2.AppProtocol = Protocol.Ftp;
```

```
Console.WriteLine($"大小: {Marshal.SizeOf(s1)} 字节。值: {s1.Address} = {s1}.");
Console.WriteLine($"大小: {Marshal.SizeOf(s2)} 字节。值: {s2.Address} = {s2}.");
```

[查看演示](#)

belindoc.com

```
Console.WriteLine($"Size: {Marshal.SizeOf(s1)} bytes. Value: {s1.Address} = {s1}.");
Console.WriteLine($"Size: {Marshal.SizeOf(s2)} bytes. Value: {s2.Address} = {s2}.");
```

[View Demo](#)

第143章：响应式扩展 (Rx)

第143.1节：观察 TextBox 的 TextChanged 事件

一个可观察对象是由文本框的 TextChanged 事件创建的。只有当输入与上一次输入不同且在 0.5 秒内没有新的输入时，输入才会被选中。此示例中的输出被发送到控制台。

可观察对象

```
.FromEventPattern(textBoxInput, "TextChanged")
.Select(s => ((TextBox) s.Sender).Text)
.Throttle(TimeSpan.FromSeconds(0.5))
.DistinctUntilChanged()
.Subscribe(text => Console.WriteLine(text));
```

第143.2节：使用 Observable

Observable从数据库流式传输数据

假设有一个返回`IEnumerable<T>`的方法，例如

```
private IEnumerable<T> GetData()
{
    try
    {
        // 从数据库返回结果
    }
    catch(Exception exception)
    {
        throw;
    }
}
```

创建一个Observable并异步启动一个方法。SelectMany将集合扁平化，订阅通过Buffer每200个元素触发一次。

```
int bufferSize = 200;

Observable
    .Start(() => GetData())
    .SelectMany(s => s)
    .Buffer(bufferSize)
    .ObserveOn(SynchronizationContext.Current)
    .Subscribe(items =>
    {
        Console.WriteLine("Loaded {0} elements", items.Count);

        // 在界面上执行某些操作，比如增加进度条
    },
    () => Console.WriteLine("Completed loading"));
```

Chapter 143: Reactive Extensions (Rx)

Section 143.1: Observing TextChanged event on a TextBox

An observable is created from the TextChanged event of the TextBox. Also any input is only selected if it's different from the last input and if there was no input within 0.5 seconds. The output in this example is sent to the console.

Observable

```
.FromEventPattern(textBoxInput, "TextChanged")
.Select(s => ((TextBox) s.Sender).Text)
.Throttle(TimeSpan.FromSeconds(0.5))
.DistinctUntilChanged()
.Subscribe(text => Console.WriteLine(text));
```

Section 143.2: Streaming Data from Database with Observable

Assume having a method returning `IEnumerable<T>`, f.e.

```
private IEnumerable<T> GetData()
{
    try
    {
        // return results from database
    }
    catch(Exception exception)
    {
        throw;
    }
}
```

Creates an Observable and starts a method asynchronously. SelectMany flattens the collection and the subscription is fired every 200 elements through Buffer.

```
int bufferSize = 200;

Observable
    .Start(() => GetData())
    .SelectMany(s => s)
    .Buffer(bufferSize)
    .ObserveOn(SynchronizationContext.Current)
    .Subscribe(items =>
    {
        Console.WriteLine("Loaded {0} elements", items.Count);

        // do something on the UI like incrementing a ProgressBar
    },
    () => Console.WriteLine("Completed loading"));
```

第144章：AssemblyInfo.cs 示例

第144.1节：全局和局部 AssemblyInfo

拥有一个全局变量可以更好地实现代码复用（DRY原则），你只需将不同的值放入有差异的项目的 AssemblyInfo.cs 中。此用法假设你的产品包含多个 Visual Studio 项目。

GlobalAssemblyInfo.cs

```
using System.Reflection;
using System.Runtime.InteropServices;
// 使用 Stackoverflow 域名作为虚构示例// 通常且大多数情况下，
```

使用一个 GlobalAssemblyInfo.cs 文件，并作为链接添加到同一产品的多个项目中，详情如下

```
// 在本地程序集信息中更改这些属性值以修改信息。
[assembly: AssemblyProduct("Stackoverflow 问答")]
[assembly: AssemblyCompany("Stackoverflow")]
[assembly: AssemblyCopyright("版权所有 © Stackoverflow 2016")]

// 如果此项目暴露给 COM，以下 GUID 用于 typelib 的 ID
[assembly: Guid("4e4f2d33-aaab-48ea-a63d-1f0a8e3c935f")]
[assembly: ComVisible(false)] // 不会暴露 ;)

// 程序集的版本信息由以下四个值组成：
// 大致翻译自我认为这是为 SO 准备的，注意他们很可能是动态生成此文件的

// 主版本号 - 6 年即 2016 年
// 次版本号 - 月份
// 日期编号 - 月中的日期
// 修订 - 构建编号
// 你可以指定所有的值，也可以使用 '*' 来默认生成构建号和修订号，示例如下：[assembly: AssemblyVersion("year.month.day.*")]
[assembly: AssemblyVersion("2016.7.00.00")]
[assembly: AssemblyFileVersion("2016.7.27.3839")]
```

AssemblyInfo.cs - 每个项目一个

```
//然后以下内容可能会根据项目放入单独的汇编文件中，例如
[assembly: AssemblyTitle("Stackoverflow.Redis")]
```

您可以使用以下步骤将 GlobalAssemblyInfo.cs 添加到本地项目中：

1. 在项目的上下文菜单中选择“添加/现有项...”
2. 选择 GlobalAssemblyInfo.cs
3. 点击右侧那个小向下箭头，展开添加按钮
4. 在按钮下拉列表中选择“作为链接添加”

第144.2节：[AssemblyVersion]

此属性为程序集应用版本。

```
[assembly: AssemblyVersion("1.0.*")]
```

星号*字符用于在每次编译时自动递增版本号的一部分（通常用于“构建”号）

Chapter 144: AssemblyInfo.cs Examples

Section 144.1: Global and local AssemblyInfo

Having a global allows for better DRYness, you need only put values that are different into AssemblyInfo.cs for projects that have variance. This use assumes your product has more than one visual studio project.

GlobalAssemblyInfo.cs

```
using System.Reflection;
using System.Runtime.InteropServices;
//using Stackoverflow domain as a made up example
```

```
// It is common, and mostly good, to use one GlobalAssemblyInfo.cs that is added
// as a link to many projects of the same product, details below
// Change these attribute values in local assembly info to modify the information.
[assembly: AssemblyProduct("Stackoverflow Q&A")]
[assembly: AssemblyCompany("Stackoverflow")]
[assembly: AssemblyCopyright("Copyright © Stackoverflow 2016")]
```

```
// The following GUID is for the ID of the typelib if this project is exposed to COM
[assembly: Guid("4e4f2d33-aaab-48ea-a63d-1f0a8e3c935f")]
[assembly: ComVisible(false)] //not going to expose ;)
```

```
// Version information for an assembly consists of the following four values:
// roughly translated from I reckon it is for SO, note that they most likely
// dynamically generate this file
// Major Version - Year 6 being 2016
// Minor Version - The month
// Day Number - Day of month
// Revision - Build number
// You can specify all the values or you can default the Build and Revision Numbers
// by using the '*' as shown below: [assembly: AssemblyVersion("year.month.day.*")]
[assembly: AssemblyVersion("2016.7.00.00")]
[assembly: AssemblyFileVersion("2016.7.27.3839")]
```

AssemblyInfo.cs - one for each project

```
//then the following might be put into a separate Assembly file per project, e.g.
[assembly: AssemblyTitle("Stackoverflow.Redis")]
```

You can add the GlobalAssemblyInfo.cs to the local project using the [following procedure](#):

1. Select Add/Existing Item... in the context menu of the project
2. Select GlobalAssemblyInfo.cs
3. Expand the Add-Button by clicking on that little down-arrow on the right hand
4. Select "Add As Link" in the buttons drop down list

Section 144.2: [AssemblyVersion]

This attribute applies a version to the assembly.

```
[assembly: AssemblyVersion("1.0.*")]
```

The * character is used to auto-increment a portion of the version automatically every time you compile (often used for the "build" number)

第144.3节：自动版本控制

您的源代码控制中版本号要么是默认的（SVN ID或Git SHA1哈希），要么是显式的（Git标签）。与其手动更新AssemblyInfo.cs中的版本号，不如使用构建时的流程将源代码控制系统中的版本写入AssemblyInfo.cs文件，从而写入程序集。

GitVersionTask或SemVer.Git.Fody NuGet包是上述方法的示例。例如，使用GitVersionTask时，在项目中安装该包后，需从AssemblyInfo.cs文件中移除Assembly*Version属性。这将使GitVersionTask负责管理程序集的版本控制。

请注意，语义化版本控制（Semantic Versioning）正日益成为事实上的标准，因此这些方法建议使用遵循SemVer的源代码控制标签。

第144.4节：常用字段

填写AssemblyInfo的默认字段是良好实践。这些信息可能会被安装程序读取，并在使用“程序和功能”（Windows 10）卸载或更改程序时显示。

最少应包括：

- AssemblyTitle - 通常是命名空间，*i.e.* MyCompany.MySolution.MyProject
- AssemblyCompany - 法人实体的全称
- AssemblyProduct - 市场部门可能对此有看法
- AssemblyCopyright - 保持更新，否则看起来很乱

“AssemblyTitle”在查看 DLL 的属性详细信息标签时变为“文件描述”。

第144.5节：[AssemblyTitle]

此属性用于为该特定程序集命名。

```
[assembly: AssemblyTitle("MyProduct")]
```

第144.6节：[AssemblyProduct]

此属性用于描述该特定程序集所属的产品。多个程序集可以是同一产品的组件，在这种情况下，它们可以共享此属性的相同值。

```
[assembly: AssemblyProduct("MyProduct")]
```

第144.7节：[InternalsVisibleTo]

如果你想让一个程序集的internal类或函数可以被另一个程序集访问，你可以通过声明InternalsVisibleTo和允许访问的程序集名称来实现。

在这个示例代码中，程序集MyAssembly允许UnitTests调用MyAssembly中的internal元素。

```
[assembly: InternalsVisibleTo("MyAssembly.UnitTests")]
```

这对于单元测试特别有用，可以避免不必要的public声明。

Section 144.3: Automated versioning

Your code in source control has version numbers either by default (SVN ids or Git SHA1 hashes) or explicitly (Git tags). Rather than manually updating versions in AssemblyInfo.cs you can use a build time process to write the version from your source control system into your AssemblyInfo.cs files and thus onto your assemblies.

The [GitVersionTask](#) or [SemVer.Git.Fody](#) NuGet packages are examples of the above. To use GitVersionTask, for instance, after installing the package in your project remove the Assembly*Version attributes from your AssemblyInfo.cs files. This puts GitVersionTask in charge of versioning your assemblies.

Note that Semantic Versioning is increasingly the *de facto* standard so these methods recommend using source control tags that follow SemVer.

Section 144.4: Common fields

It's good practice to complete your AssemblyInfo's default fields. The information may be picked up by installers and will then appear when using Programs and Features (Windows 10) to uninstall or change a program.

The minimum should be:

- AssemblyTitle - usually the namespace, *i.e.* MyCompany.MySolution.MyProject
- AssemblyCompany - the legal entities full name
- AssemblyProduct - marketing may have a view here
- AssemblyCopyright - keep it up to date as it looks scruffy otherwise

'AssemblyTitle' becomes the 'File description' when examining the DLL's Properties Details tab.

Section 144.5: [AssemblyTitle]

This attribute is used to give a name to this particular assembly.

```
[assembly : AssemblyTitle ("MyProduct")]
```

Section 144.6: [AssemblyProduct]

This attribute is used to describe the product that this particular assembly is for. Multiple assemblies can be components of the same product, in which case they can all share the same value for this attribute.

```
[assembly : AssemblyProduct ("MyProduct")]
```

Section 144.7: [InternalsVisibleTo]

If you want to make `internal` classes or functions of an assembly accessible from another assembly you declare this by InternalsVisibleTo and the assembly name that is allowed to access.

In this example code in the assembly MyAssembly.UnitTests is allowed to call `internal` elements from MyAssembly.

```
[assembly : InternalsVisibleTo ("MyAssembly.UnitTests")]
```

This is especially useful for unit testing to prevent unnecessary `public` declarations.

第144.8节 : [AssemblyConfiguration]

AssemblyConfiguration : AssemblyConfiguration属性必须包含用于构建程序集的配置。使用条件编译来正确包含不同的程序集配置。使用类似下面示例的代码块。根据常用的配置添加多个不同的配置。

```
#if (DEBUG)  
[assembly: AssemblyConfiguration("Debug")]  
  
#else  
[assembly: AssemblyConfiguration("Release")]  
  
#endif
```

第144.9节 : [AssemblyKeyFile]

每当我们希望程序集安装到全局程序集缓存 (GAC) 时，必须拥有强名称。要为程序集创建强名称，我们必须生成一个公钥。用于生成.snk文件。

创建强名称密钥文件

1. 使用具有管理员权限的VS2015开发者命令提示符
2. 在命令提示符下，输入 cd C:\目录名称 并按回车。
3. 在命令提示符下，输入 sn -k KeyFileName.snk，然后按回车。

一旦在指定目录创建了 KeyFileName.snk 文件，就在项目中引用它。给 AssemblyKeyFileAttribute属性指定 snk文件的路径，以便在构建类库时生成密钥。

属性 -> AssemblyInfo.cs

```
[assembly: AssemblyKeyFile(@"c:\目录名称\KeyFileName.snk")]
```

这将在构建后创建一个强名称程序集。创建强名称程序集后，您就可以将其安装到 GAC中。

编程愉快 :)

第144.10节 : 读取程序集属性

使用.NET丰富的反射API，您可以访问程序集的元数据。例如，您可以使用以下代码获取此程序集的标题属性

```
using System.Linq;  
using System.Reflection;  
  
...  
  
Assembly assembly = typeof(this).Assembly;  
var titleAttribute = assembly.GetCustomAttributes<AssemblyTitleAttribute>().FirstOrDefault();
```

Section 144.8: [AssemblyConfiguration]

AssemblyConfiguration: The AssemblyConfiguration attribute must have the configuration that was used to build the assembly. Use conditional compilation to properly include different assembly configurations. Use the block similar to the example below. Add as many different configurations as you commonly use.

```
#if (DEBUG)  
[assembly: AssemblyConfiguration("Debug")]  
  
#else  
[assembly: AssemblyConfiguration("Release")]  
  
#endif
```

Section 144.9: [AssemblyKeyFile]

Whenever we want our assembly to install in GAC then it is must to have a strong name. For strong naming assembly we have to create a public key. To generate the .snk file.

To create a strong name key file

1. Developers command prompt for VS2015 (with administrator Access)
2. At the command prompt, type cd C:\Directory_Name and press ENTER.
3. At the command prompt, type sn -k KeyFileName.snk, and then press ENTER.

once the keyFileName.snk is created at specified directory then give reference in your project . give AssemblyKeyFileAttribute attribute the path to snk file to generate the key when we build our class library.

Properties -> AssemblyInfo.cs

```
[assembly: AssemblyKeyFile(@"c:\Directory_Name\KeyFileName.snk")]
```

This will create a strong name assembly after build. After creating your strong name assembly you can then install it in GAC

Happy Coding :)

Section 144.10: Reading Assembly Attributes

Using .NET's rich reflection APIs, you can gain access to an assembly's metadata. For example, you can get this assembly's title attribute with the following code

```
using System.Linq;  
using System.Reflection;  
  
...  
  
Assembly assembly = typeof(this).Assembly;  
var titleAttribute = assembly.GetCustomAttributes<AssemblyTitleAttribute>().FirstOrDefault();
```

```
Console.WriteLine($"此程序集的标题是 {titleAttribute?.Title}");
```

```
Console.WriteLine($"This assembly title is {titleAttribute?.Title}");
```

belindoc.com

第145章：使用纯文本编辑器和C#编译器（csc.exe）创建控制台应用程序

第145.1节：使用纯文本编辑器和C#编译器创建控制台应用程序

为了使用纯文本编辑器创建用C#编写的控制台应用程序，您需要C#编译器。

C#编译器（csc.exe）可以在以下位置找到：

%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe

注意：根据您系统上安装的.NET Framework版本，您可能需要相应地更改上述路径。

保存代码

本主题的目的不是教您如何编写控制台应用程序，而是教您如何仅使用C#编译器和任何纯文本编辑器（如记事本）来编译一个控制台应用程序[生成单个可执行文件]。

1. 使用键盘快捷键打开运行对话框 **Windows键** + **R**
2. 输入notepad，然后按下**回车**
3. 将下面的示例代码粘贴到记事本中
4. 通过点击“文件”→“另存为...”，将文件保存为ConsoleApp.cs，然后在“文件名”文本框中输入ConsoleApp.cs
然后选择所有文件作为文件类型。
5. 点击保存

编译源代码

1. 使用以下方法打开运行对话框 **Windows键** + **R**
2. 输入：

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe /t:exe  
/out:"C:\Users\yourUserName\Documents\ConsoleApp.exe"  
"C:\Users\yourUserName\Documents\ConsoleApp.cs"
```

现在，返回到你最初保存ConsoleApp.cs文件的位置。你现在应该能看到一个可执行文件（ConsoleApp.exe）。双击ConsoleApp.exe打开它。

就是这样！你的控制台应用程序已经编译完成。一个可执行文件已经创建，现在你有了一个可运行的控制台应用程序。

```
using System;  
  
namespace ConsoleApp  
{  
    class 程序  
    {  
        private static string input = String.Empty;  
  
        static void Main(string[] args)  
        {  
            goto DisplayGreeting;  
        }  
    }  
}
```

Chapter 145: Creating a Console Application using a Plain-Text Editor and the C# Compiler (csc.exe)

Section 145.1: Creating a Console application using a Plain-Text Editor and the C# Compiler

In order to use a plain-text editor to create a Console application that is written in C#, you'll need the C# Compiler.

The C# Compiler (csc.exe), can be found at the following location:

%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe

N.B. Depending upon which version of the .NET Framework that is installed on your system, you may need to change the path above, accordingly.

Saving the Code

The purpose of this topic is not to teach you *how* to write a Console application, but to teach you how to *compile* one [to produce a single executable file], with nothing other than the C# Compiler and any Plain-Text Editor (such as Notepad).

1. Open the Run dialog, by using the keyboard shortcut **Windows Key** + **R**
2. Type notepad, then hit **Enter**
3. Paste the example code below, into Notepad
4. Save the file as ConsoleApp.cs, by going to **File** → **Save As...**, then entering ConsoleApp.cs in the 'File Name' text field, then selecting All Files as the file-type.
5. Click Save

Compiling the Source Code

1. Open the Run dialog, using **Windows Key** + **R**
2. Enter:

```
%WINDIR%\Microsoft.NET\Framework64\v4.0.30319\csc.exe /t:exe  
/out:"C:\Users\yourUserName\Documents\ConsoleApp.exe"  
"C:\Users\yourUserName\Documents\ConsoleApp.cs"
```

Now, go back to where you originally saved your ConsoleApp.cs file. You should now see an executable file (ConsoleApp.exe). Double-click ConsoleApp.exe to open it.

That's it! Your console application has been compiled. An executable file has been created and you now have a working Console app.

```
using System;  
  
namespace ConsoleApp  
{  
    class Program  
    {  
        private static string input = String.Empty;  
  
        static void Main(string[] args)  
        {  
            goto DisplayGreeting;  
        }  
    }  
}
```

```

DisplayGreeting:
{
    Console.WriteLine("你好！你叫什么名字？");

    input = Console.ReadLine();

    if (input.Length >= 1)
    {
        Console.WriteLine(
            "Hello, " +
        输入 +
            ", 随时输入 'Exit' 以退出此应用。");

        goto 等待进一步指令;
    }
    else
    {
        goto 显示问候;
    }
}

等待进一步指令:
{
    输入 = Console.ReadLine();

    if(输入.ToLower() == "exit")
    {
        输入 = String.Empty;

        Environment.Exit(0);
    }
    else
    {
        goto 等待进一步指令;
    }
}
}

```

```

DisplayGreeting:
{
    Console.WriteLine("Hello! What is your name?");

    input = Console.ReadLine();

    if (input.Length >= 1)
    {
        Console.WriteLine(
            "Hello, " +
            input +
            ", enter 'Exit' at any time to exit this app.");

        goto AwaitFurtherInstruction;
    }
    else
    {
        goto DisplayGreeting;
    }
}

AwaitFurtherInstruction:
{
    input = Console.ReadLine();

    if(input.ToLower() == "exit")
    {
        input = String.Empty;

        Environment.Exit(0);
    }
    else
    {
        goto AwaitFurtherInstruction;
    }
}
}

```

第146章：CLSCCompliantAttribute

构造函数

CLSCCompliantAttribute(Boolean) 使用布尔值初始化 CLSCCompliantAttribute 类的实例
指示所示程序元素是否符合 CLS 规范。

参数

第146.1节：适用 CLS 规则的访问修饰符

```
using System;

[assembly:CLSCCompliant(true)]
namespace CLSDoc
{

    public class Cat
    {
        internal UInt16 _age = 0;
        private UInt16 _daysTillVaccination = 0;

        //警告 CS3003 'Cat.DaysTillVaccination' 的类型不符合 CLS 规范
        protected UInt16 DaysTillVaccination
        {
            get { return _daysTillVaccination; }
        }

        //警告 CS3003 'Cat.Age' 的类型不符合 CLS 规范
        public UInt16 年龄
        { get { return _age; } }

        //符合 CLS 规则的有效行为
        public int 增加年龄()
        {
            int 增加后的年龄 = (int)_age + 1;

            return 增加后的年龄;
        }
    }
}
```

CLS 合规规则仅适用于公共/受保护的组件。

第146.2节：违反 CLS 规则：无符号类型 / sbyte

```
using System;

[assembly:CLSCCompliant(true)]
namespace CLSDoc
{

    public class 汽车
    {
        internal UInt16 _制造年份 = 0;

        //警告 CS3008 标识符 '_numberOfDoors' 不符合 CLS 规范
        //警告 CS3003 'Car._numberOfDoors' 的类型不符合 CLS 规范
        public UInt32 _车门数量 = 0;
    }
}
```

Chapter 146: CLSCCompliantAttribute

Constructor

CLSCCompliantAttribute(Boolean) Initializes an instance of the CLSCCompliantAttribute class with a Boolean value indicating whether the indicated program element is CLS-compliant.

Parameter

```
using System;

[assembly:CLSCCompliant(true)]
namespace CLSDoc
{

    public class Cat
    {
        internal UInt16 _age = 0;
        private UInt16 _daysTillVaccination = 0;

        //Warning CS3003 Type of 'Cat.DaysTillVaccination' is not CLS-compliant
        protected UInt16 DaysTillVaccination
        {
            get { return _daysTillVaccination; }
        }

        //Warning CS3003 Type of 'Cat.Age' is not CLS-compliant
        public UInt16 Age
        { get { return _age; } }

        //valid behaviour by CLS-compliant rules
        public int IncreaseAge()
        {
            int increasedAge = (int)_age + 1;

            return increasedAge;
        }
    }
}
```

The rules for CLS compliance apply only to a public/protected components.

Section 146.2: Violation of CLS rule: Unsigned types / sbyte

```
using System;

[assembly:CLSCCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        internal UInt16 _yearOfCreation = 0;

        //Warning CS3008 Identifier '_numberOfDoors' is not CLS-compliant
        //Warning CS3003 Type of 'Car._numberOfDoors' is not CLS-compliant
        public UInt32 _numberOfDoors = 0;
    }
}
```

```

//警告 CS3003 'Car.YearOfCreation' 的类型不符合 CLS 规范
public UInt16 创建年份
{
    get { return _yearOfCreation; }
}

//警告 CS3002 'Car.CalculateDistance()' 的返回类型不符合 CLS 规范
public UInt64 计算距离()
{
    return 0;
}

//警告 CS3002 'Car.TestDummyUnsignedPointerMethod()' 的返回类型不符合 CLS 规范
public UIntPtr 测试虚拟无符号指针方法()
{
    int[] 数组 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    UIntPtr 指针 = (UIntPtr)数组[0];

    return 指针;
}

//警告 CS3003 类型 'Car.age' 不符合 CLS 规范
public sbyte age = 120;
}

```

第146.3节：违反 CLS 规则：命名相同

```

using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

public class 汽车
{
    //警告 CS3005 标识符 'Car.CALCULATEAge()' 仅大小写不同，不符合 CLS 规范

    public int CalculateAge()
    {
        return 0;
    }

    public int CALCULATEAge()
    {
        return 0;
    }
}

```

Visual Basic 不区分大小写

```

//Warning CS3003 Type of 'Car.YearOfCreation' is not CLS-compliant
public UInt16 YearOfCreation
{
    get { return _yearOfCreation; }
}

//Warning CS3002 Return type of 'Car.CalculateDistance()' is not CLS-compliant
public UInt64 CalculateDistance()
{
    return 0;
}

//Warning CS3002 Return type of 'Car.TestDummyUnsignedPointerMethod()' is not CLS-compliant
public UIntPtr TestDummyUnsignedPointerMethod()
{
    int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    UIntPtr ptr = (UIntPtr)arr[0];

    return ptr;
}

//Warning CS3003 Type of 'Car.age' is not CLS-compliant
public sbyte age = 120;
}

```

Section 146.3: Violation of CLS rule: Same naming

```

using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        //Warning CS3005 Identifier 'Car.CALCULATEAge()' differing only in case is not CLS-compliant
        public int CalculateAge()
        {
            return 0;
        }

        public int CALCULATEAge()
        {
            return 0;
        }
    }
}

```

Visual Basic is not case sensitive

第146.4节：违反 CLS 规则：标识符 _

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class 汽车
    {
        //警告 CS3008 标识符 '_age' 不符合 CLS 规范
        public int _age = 0;
    }

}
```

变量名不能以下划线开头

第146.5节：违反CLS规则：继承自非CLS兼容类

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    [CLSCompliant(false)]
    public class Animal
    {
        public int age = 0;
    }

    //警告 CS3009 'Dog': 基类 'Animal' 不是CLS兼容的
    public class Dog : Animal
    {
    }

}
```

Section 146.4: Violation of CLS rule: Identifier _

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    public class Car
    {
        //Warning CS3008 Identifier '_age' is not CLS-compliant
        public int _age = 0;
    }

}
```

You can not start variable with _

Section 146.5: Violation of CLS rule: Inherit from non CLSComplaint class

```
using System;

[assembly:CLSCompliant(true)]
namespace CLSDoc
{

    [CLSCompliant(false)]
    public class Animal
    {
        public int age = 0;
    }

    //Warning CS3009 'Dog': base type 'Animal' is not CLS-compliant
    public class Dog : Animal
    {
    }

}
```

第147章 : ObservableCollection<T>

第147.1节 : 初始化ObservableCollection<T>

ObservableCollection 是一种类型为 T 的集合，类似于 List<T>，意味着它包含类型为 T 的对象。

从文档中我们了解到：

ObservableCollection表示一个动态数据集合，当项目被添加、移除或整个列表被刷新时，会提供通知。

与其他集合的主要区别在于ObservableCollection实现了接口INotifyCollectionChanged和INotifyPropertyChanged，并且在新对象被添加或移除以及集合被清空时，立即触发通知事件。

这对于连接应用程序的UI和后端特别有用，无需编写额外代码
因为当对象被添加到或从可观察集合中移除时，UI会自动更新。

使用它的第一步是包含

```
using System.Collections.ObjectModel
```

你可以创建一个空的集合实例，例如类型为string

```
ObservableCollection<string> collection = new ObservableCollection<string>();
```

或者创建一个已填充数据的实例

```
ObservableCollection<string> collection = new ObservableCollection<string>()
{
    "First_String", "Second_String"
};
```

请记住，和所有 IList 集合一样，索引从 0 开始 (IList.Item 属性)。

Chapter 147: ObservableCollection<T>

Section 147.1: Initialize ObservableCollection<T>

ObservableCollection is a collection of type T like List<T> which means that it holds objects of type T.

From documentation we read that :

ObservableCollectionrepresents a dynamic data collection that provides notifications when items get added, removed, or when the entire list is refreshed.

The key difference from other collections is that ObservableCollection implements the interfaces INotifyCollectionChanged and INotifyPropertyChanged and immediately raise notification event when a new object is added or removed and when collection is cleared.

This is especially useful for connecting the UI and backend of an application without having to write extra code because when an object is added to or removed from an observable collection, the UI is automatically updated.

The first step in order to use it is to include

```
using System.Collections.ObjectModel
```

You can either create an empty instance of a collection for example of type string

```
ObservableCollection<string> collection = new ObservableCollection<string>();
```

or an instance that is filled with data

```
ObservableCollection<string> collection = new ObservableCollection<string>()
{
    "First_String", "Second_String"
};
```

Remember as in all IList collection, index starts from 0 ([IList.Item Property](#)).

第 148 章：哈希函数

第 148.1 节：用于密码哈希的 PBKDF2

PBKDF2 (“基于密码的密钥派生函数 2”) 是推荐用于密码哈希的哈希函数之一。它是 rfc-2898 的一部分。

.NET 的 Rfc2898DeriveBytes 类基于 HMACSHA1。

```
using System.Security.Cryptography;  
...  
  
public const int SALT_SIZE = 24; // 字节大小  
public const int HASH_SIZE = 24; // 字节大小  
public const int ITERATIONS = 100000; // pbkdf2 迭代次数  
  
public static byte[] CreateHash(string input)  
{  
    // 生成盐值  
    RNGCryptoServiceProvider provider = new RNGCryptoServiceProvider();  
    byte[] salt = new byte[SALT_SIZE];  
    provider.GetBytes(salt);  
  
    // 生成哈希值  
    Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(input, salt, ITERATIONS);  
    return pbkdf2.GetBytes(HASH_SIZE);  
}
```

PBKDF2 需要一个盐值和迭代次数。

迭代次数：

较高的迭代次数会减慢算法速度，从而使密码破解变得更加困难。因此，建议使用较高的迭代次数。例如，PBKDF2 的速度比 MD5 慢几个数量级。

盐值：

盐值可以防止在彩虹表中查找哈希值。它必须与密码哈希一起存储。

建议每个密码使用一个盐值（而不是一个全局盐值）。

第148.2节：使用

Pbkdf2 完整的密码哈希解决方案

```
using System;  
using System.Linq;  
using System.Security.Cryptography;  
  
namespace YourCryptoNamespace  
{  
    /// <summary>  
    /// 使用带盐的 PBKDF2-SHA1 密码哈希。  
    /// 兼容性：.NET 3.0 及更高版本。  
    /// </summary>  
    /// <remarks>有关密码哈希的更多信息，请参见 http://crackstation.net/hashing-security.htm。</remarks>  
  
    public static class PasswordHashProvider
```

Chapter 148: Hash Functions

Section 148.1: PBKDF2 for Password Hashing

PBKDF2 ("Password-Based Key Derivation Function 2") is one of the recommended hash-functions for password-hashing. It is part of [rfc-2898](#).

.NET's Rfc2898DeriveBytes-Class is based upon HMACSHA1.

```
using System.Security.Cryptography;  
...  
  
public const int SALT_SIZE = 24; // size in bytes  
public const int HASH_SIZE = 24; // size in bytes  
public const int ITERATIONS = 100000; // number of pbkdf2 iterations  
  
public static byte[] CreateHash(string input)  
{  
    // Generate a salt  
    RNGCryptoServiceProvider provider = new RNGCryptoServiceProvider();  
    byte[] salt = new byte[SALT_SIZE];  
    provider.GetBytes(salt);  
  
    // Generate the hash  
    Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(input, salt, ITERATIONS);  
    return pbkdf2.GetBytes(HASH_SIZE);  
}
```

PBKDF2 requires a [salt](#) and the number of iterations.

Iterations:

A high number of iterations will slow the algorithm down, which makes password cracking a lot harder. A high number of iterations is therefore recommended. PBKDF2 is orders of magnitude slower than MD5 for example.

Salt:

A salt will prevent the lookup of hash values in [rainbow tables](#). It has to be stored alongside the password hash. One salt per password (not one global salt) is recommended.

Section 148.2: Complete Password Hashing Solution using Pbkdf2

```
using System;  
using System.Linq;  
using System.Security.Cryptography;  
  
namespace YourCryptoNamespace  
{  
    /// <summary>  
    /// Salted password hashing with PBKDF2-SHA1.  
    /// Compatibility: .NET 3.0 and later.  
    /// </summary>  
    /// <remarks>See http://crackstation.net/hashing-security.htm for much more on password hashing.</remarks>  
    public static class PasswordHashProvider
```

```

{
    /// <summary>
    /// 盐的字节大小, 64 长度确保安全, 但可以增加或减少
    /// </summary>
    private const int SaltByteSize = 64;
    /// <summary>
    /// 哈希的字节大小,
    /// </summary>
    private const int HashByteSize = 64;
    /// <summary>
    /// 高迭代次数不易被破解
    /// 
    private const int Pbkdf2Iterations = 10000;

    /// <summary>
    /// 创建密码的带盐PBKDF2哈希。
    /// </summary>
    /// <remarks>
    /// 盐值和哈希必须并排保存以用于密码。它们可以作为字节保存, 或者使用下一个类中的便捷方法将byte[]转换为字符串, 之后在执行密码验证时再转换回字节。
    /// </remarks>
    /// <param name="password">要哈希的密码。</param>
    /// <returns>密码的哈希值。</returns>
    public static PasswordHashContainer CreateHash(string password)
    {
        // 生成随机盐值
        using (var csprng = new RNGCryptoServiceProvider())
        {
            // 为每个密码哈希创建唯一盐值, 以防止彩虹表和字典攻击

            var salt = new byte[SaltByteSize];
            csprng.GetBytes(salt);

            // 对密码进行哈希并编码参数
            var hash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);

            return new PasswordHashContainer(hash, salt);
        }
    }
    /// <summary>
    /// 根据传入的密码字符串和存储的盐重新创建密码哈希
    /// 
    /// <param name="password">要检查的密码。</param>
    /// <param name="salt">现有的盐。</param>
    /// <returns>基于密码和盐生成的哈希</returns>
    public static byte[] CreateHash(string password, byte[] salt)
    {
        // 从哈希中提取参数
        return Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
    }

    /// <summary>
    /// 验证给定密码是否与正确密码的哈希匹配。
    /// </summary>
    /// <param name="password">要检查的密码。</param>
    /// <param name="salt">现有存储的盐值。</param>
    /// <param name="correctHash">现有密码的哈希值。</param>
    /// <returns><c>true</c> 如果密码正确。否则返回 <c>false</c> 。</returns>
    public static bool ValidatePassword(string password, byte[] salt, byte[] correctHash)
    {
        // 从哈希中提取参数
    }
}

```

```

{
    /// <summary>
    /// The salt byte size, 64 length ensures safety but could be increased / decreased
    /// </summary>
    private const int SaltByteSize = 64;
    /// <summary>
    /// The hash byte size,
    /// </summary>
    private const int HashByteSize = 64;
    /// <summary>
    /// High iteration count is less likely to be cracked
    /// </summary>
    private const int Pbkdf2Iterations = 10000;

    /// <summary>
    /// Creates a salted PBKDF2 hash of the password.
    /// </summary>
    /// <remarks>
    /// The salt and the hash have to be persisted side by side for the password. They could be
    persisted as bytes or as a string using the convenience methods in the next class to convert from
    byte[] to string and later back again when executing password validation.
    /// </remarks>
    /// <param name="password">The password to hash.</param>
    /// <returns>The hash of the password.</returns>
    public static PasswordHashContainer CreateHash(string password)
    {
        // Generate a random salt
        using (var csprng = new RNGCryptoServiceProvider())
        {
            // create a unique salt for every password hash to prevent rainbow and dictionary based
            attacks
            var salt = new byte[SaltByteSize];
            csprng.GetBytes(salt);

            // Hash the password and encode the parameters
            var hash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);

            return new PasswordHashContainer(hash, salt);
        }
    }
    /// <summary>
    /// Recreates a password hash based on the incoming password string and the stored salt
    /// </summary>
    /// <param name="password">The password to check.</param>
    /// <param name="salt">The salt existing.</param>
    /// <returns>the generated hash based on the password and salt</returns>
    public static byte[] CreateHash(string password, byte[] salt)
    {
        // Extract the parameters from the hash
        return Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
    }

    /// <summary>
    /// Validates a password given a hash of the correct one.
    /// </summary>
    /// <param name="password">The password to check.</param>
    /// <param name="salt">The existing stored salt.</param>
    /// <param name="correctHash">The hash of the existing password.</param>
    /// <returns><c>true</c> if the password is correct. <c>false</c> otherwise. </returns>
    public static bool ValidatePassword(string password, byte[] salt, byte[] correctHash)
    {
        // Extract the parameters from the hash
    }
}

```

```

byte[] testHash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
return CompareHashes(correctHash, testHash);
}
/// <summary>
/// 比较两个字节数组 (哈希)
///
/// <param name="array1">数组1。</param>
/// <param name="array2">数组2。</param>
/// <returns>如果相同则返回<c>true</c>, 否则返回<c>false</c></returns>
public static bool CompareHashes(byte[] array1, byte[] array2)
{
    if (array1.Length != array2.Length) return false;
    return !array1.Where((t, i) => t != array2[i]).Any();
}

/// <summary>
/// 计算密码的 PBKDF2-SHA1 哈希值。
/// </summary>
/// <param name="password">要哈希的密码。</param>
/// <param name="salt">盐值。</param>
/// <param name="iterations">PBKDF2 迭代次数。</param>
/// <param name="outputBytes">要生成的哈希长度, 单位为字节。</param>
/// <returns>密码的哈希值。</returns>
private static byte[] Pbkdf2(string password, byte[] salt, int iterations, int outputBytes)
{
    using (var pbkdf2 = new Rfc2898DeriveBytes(password, salt))
    {
        pbkdf2.IterationCount = iterations;
        return pbkdf2.GetBytes(outputBytes);
    }
}

/// <summary>
/// 密码哈希、盐值和迭代次数的容器。
/// </summary>
public sealed class PasswordHashContainer
{
    /// <summary>
    /// 获取哈希后的密码。
    /// </summary>
    public byte[] HashedPassword { get; private set; }

    /// <summary>
    /// 获取盐值。
    /// </summary>
    public byte[] Salt { get; private set; }

    /// <summary>
    /// 初始化 <see cref="PasswordHashContainer" /> 类的新实例。
    /// </summary>
    /// <param name="hashedPassword">哈希密码。</param>
    /// <param name="salt">盐值。</param>
    public PasswordHashContainer(byte[] hashedPassword, byte[] salt)
    {
        this.HashedPassword = hashedPassword;
        this.Salt = salt;
    }

    /// <summary>
    /// 方便的方法, 用于在十六进制字符串和字节数组之间转换。
    /// </summary>

```

```

byte[] testHash = Pbkdf2(password, salt, Pbkdf2Iterations, HashByteSize);
return CompareHashes(correctHash, testHash);
}
/// <summary>
/// Compares two byte arrays (hashes)
/// </summary>
/// <param name="array1">The array1.</param>
/// <param name="array2">The array2.</param>
/// <returns><c>true</c> if they are the same, otherwise <c>false</c></returns>
public static bool CompareHashes(byte[] array1, byte[] array2)
{
    if (array1.Length != array2.Length) return false;
    return !array1.Where((t, i) => t != array2[i]).Any();
}

/// <summary>
/// Computes the PBKDF2-SHA1 hash of a password.
/// </summary>
/// <param name="password">The password to hash.</param>
/// <param name="salt">The salt.</param>
/// <param name="iterations">The PBKDF2 iteration count.</param>
/// <param name="outputBytes">The length of the hash to generate, in bytes.</param>
/// <returns>A hash of the password.</returns>
private static byte[] Pbkdf2(string password, byte[] salt, int iterations, int outputBytes)
{
    using (var pbkdf2 = new Rfc2898DeriveBytes(password, salt))
    {
        pbkdf2.IterationCount = iterations;
        return pbkdf2.GetBytes(outputBytes);
    }
}

/// <summary>
/// Container for password hash and salt and iterations.
/// </summary>
public sealed class PasswordHashContainer
{
    /// <summary>
    /// Gets the hashed password.
    /// </summary>
    public byte[] HashedPassword { get; private set; }

    /// <summary>
    /// Gets the salt.
    /// </summary>
    public byte[] Salt { get; private set; }

    /// <summary>
    /// Initializes a new instance of the <see cref="PasswordHashContainer" /> class.
    /// </summary>
    /// <param name="hashedPassword">The hashed password.</param>
    /// <param name="salt">The salt.</param>
    public PasswordHashContainer(byte[] hashedPassword, byte[] salt)
    {
        this.HashedPassword = hashedPassword;
        this.Salt = salt;
    }

    /// <summary>
    /// Convenience methods for converting between hex strings and byte array.
    /// </summary>

```

```

public static class ByteConverter
{
    /// <summary>
    /// 将十六进制表示的字符串转换为字节数组
    /// </summary>
    /// <param name="hexedString">十六进制字符串。</param>
    /// <returns></returns>
    public static byte[] GetHexBytes(string hexedString)
    {
        var bytes = new byte[hexedString.Length / 2];
        for (var i = 0; i < bytes.Length; i++)
        {
            var strPos = i * 2;
            var chars = hexedString.Substring(strPos, 2);
            bytes[i] = Convert.ToByte(chars, 16);
        }
        return bytes;
    }
    /// <summary>
    /// 获取传入字节数组的十六进制字符串表示。
    /// </summary>
    /// <param name="bytes">字节数组。</param>
    public static string GetHexString(byte[] bytes)
    {
        return BitConverter.ToString(bytes).Replace("-", "").ToUpper();
    }
}

/*
* 使用 PBKDF2 的密码哈希 (http://crackstation.net/hashing-security.htm)。
* 版权所有 (c) 2013, Taylor Hornby
* 保留所有权利。
*
* 允许在符合以下条件的前提下，以源代码和二进制形式进行再分发和使用，无论是否经过
修改：
*
* 1. 源代码的再分发必须保留上述版权声明。
* 本条件列表及以下免责声明。
*
* 2. 以二进制形式再分发时，必须在随发行版提供的文档和/或其他材料中复制上述版权声明、* 本条件
列表及以下免责声明。
*
* 本软件由版权持有人和贡献者按“原样”提供，* 不提供任何明示或暗示的保证，包括但不限于 * 对
适销性和特定用途适用性的暗示保证。* 在任何情况下，版权持有人或贡献者均不对任何直接、间接
、附带、特殊、示范性或
*
* 后果性损害（包括但不限于采购替代商品或服务；使用、数据或利润的损失；或业务 * 中断
）承担责任，无论其责任理论为何，* 无论是合同责任、严格责任还是侵权责任（包括过失或其他
原因），* 即使已被告知发生此类损害的可能性，亦不承担任何责任。
*/

```

请参阅此优秀资源[Crackstation - Salted Password Hashing - Doing it Right](http://crackstation.net/hashing-security.htm)了解更多信息。
该解决方案的一部分（哈希函数）基于该网站的代码。

```

public static class ByteConverter
{
    /// <summary>
    /// Converts the hex representation string to an array of bytes
    /// </summary>
    /// <param name="hexedString">The hexed string.</param>
    /// <returns></returns>
    public static byte[] GetHexBytes(string hexedString)
    {
        var bytes = new byte[hexedString.Length / 2];
        for (var i = 0; i < bytes.Length; i++)
        {
            var strPos = i * 2;
            var chars = hexedString.Substring(strPos, 2);
            bytes[i] = Convert.ToByte(chars, 16);
        }
        return bytes;
    }
    /// <summary>
    /// Gets a hex string representation of the byte array passed in.
    /// </summary>
    /// <param name="bytes">The bytes.</param>
    public static string GetHexString(byte[] bytes)
    {
        return BitConverter.ToString(bytes).Replace("-", "").ToUpper();
    }
}

/*
* Password Hashing With PBKDF2 (http://crackstation.net/hashing-security.htm).
* Copyright (c) 2013, Taylor Hornby
* All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions are met:
*
* 1. Redistributions of source code must retain the above copyright notice,
* this list of conditions and the following disclaimer.
*
* 2. Redistributions in binary form must reproduce the above copyright notice,
* this list of conditions and the following disclaimer in the documentation
* and/or other materials provided with the distribution.
*
* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
* AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
* LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
* CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
* SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
* INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
* CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
* POSSIBILITY OF SUCH DAMAGE.
*/

```

Please see this excellent resource [Crackstation - Salted Password Hashing - Doing it Right](http://crackstation.net/hashing-security.htm) for more information.
Part of this solution (the hashing function) was based on the code from that site.

第148.3节：MD5

哈希函数将任意长度的二进制字符串映射为固定长度的小二进制字符串。

MD5是一种广泛使用的哈希函数，生成128位哈希值（16字节，32个十六进制字符）。

System.Security.Cryptography.MD5 类的 ComputeHash 方法返回一个长度为16字节的哈希数组。

示例：

```
using System;
using System.Security.Cryptography;
using System.Text;

internal class Program
{
    private static void Main()
    {
        var source = "Hello World!";

        // 创建 MD5 哈希算法默认实现的实例。
        using (var md5Hash = MD5.Create())
        {
            // 源字符串的字节数组表示
            var sourceBytes = Encoding.UTF8.GetBytes(source);

            // 为输入数据生成哈希值（字节数组）
            var hashBytes = md5Hash.ComputeHash(sourceBytes);

            // 将哈希字节数组转换为字符串
            var hash = BitConverter.ToString(hashBytes).Replace("-", string.Empty);

            // 输出 MD5 哈希值
            Console.WriteLine("字符串 " + source + " 的 MD5 哈希值是: " + hash);
        }
    }
}
```

输出：字符串 Hello World! 的 MD5 哈希值是: ED076287532E86365E841E92BFC50D8C

安全问题：

像大多数哈希函数一样，MD5 既不是加密也不是编码。它可以被暴力破解，并且存在广泛的碰撞和原像攻击漏洞。

第 148.4 节：SHA1

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class 程序
    {
```

Section 148.3: MD5

Hash functions map binary strings of an arbitrary length to small binary strings of a fixed length.

The [MD5](#) algorithm is a widely used hash function producing a 128-bit hash value (16 Bytes, 32 Hexdecimal characters).

The [ComputeHash](#) method of the [System.Security.Cryptography.MD5](#) class returns the hash as an array of 16 bytes.

Example:

```
using System;
using System.Security.Cryptography;
using System.Text;

internal class Program
{
    private static void Main()
    {
        var source = "Hello World!";

        // Creates an instance of the default implementation of the MD5 hash algorithm.
        using (var md5Hash = MD5.Create())
        {
            // Byte array representation of source string
            var sourceBytes = Encoding.UTF8.GetBytes(source);

            // Generate hash value(Byte Array) for input data
            var hashBytes = md5Hash.ComputeHash(sourceBytes);

            // Convert hash byte array to string
            var hash = BitConverter.ToString(hashBytes).Replace("-", string.Empty);

            // Output the MD5 hash
            Console.WriteLine("The MD5 hash of " + source + " is: " + hash);
        }
    }
}
```

Output: The MD5 hash of Hello World! is: ED076287532E86365E841E92BFC50D8C

Security Issues:

Like most hash functions, MD5 is neither encryption nor encoding. It can be reversed by brute-force attack and suffers from extensive vulnerabilities against collision and preimage attacks.

Section 148.4: SHA1

```
using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
```

```

static void Main(string[] args)
{
    string source = "Hello World!";
    using (SHA1 sha1Hash = SHA1.Create())
    {
        //从字符串到字节数组
        byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
        byte[] hashBytes = sha1Hash.ComputeHash(sourceBytes);
        string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

        Console.WriteLine("字符串 " + source + " 的 SHA1 哈希值是: " + hash);
    }
}

```

输出：

Hello Word! 的 SHA1 哈希值是: 2EF7BDE608CE5404E97D5F042F95F89F1C232871

第148.5节：SHA256

```

using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class 程序
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA256 sha256Hash = SHA256.Create())
            {
                //从字符串到字节数组
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha256Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("字符串 " + source + " 的 SHA256 哈希值是: " + hash);
            }
        }
    }
}

```

输出：

Hello World! 的 SHA256 哈希值是:
7F83B1657FF1FC53B92DC18148A1D65DFC2D4B1FA3D677284ADDD200126D9069

第148.6节：SHA384

```

using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{

```

```

static void Main(string[] args)
{
    string source = "Hello World!";
    using (SHA1 sha1Hash = SHA1.Create())
    {
        //From String to byte array
        byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
        byte[] hashBytes = sha1Hash.ComputeHash(sourceBytes);
        string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

        Console.WriteLine("The SHA1 hash of " + source + " is: " + hash);
    }
}

```

Output:

The SHA1 hash of Hello Word! is: 2EF7BDE608CE5404E97D5F042F95F89F1C232871

Section 148.5: SHA256

```

using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA256 sha256Hash = SHA256.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha256Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

                Console.WriteLine("The SHA256 hash of " + source + " is: " + hash);
            }
        }
    }
}

```

Output:

The SHA256 hash of Hello World! is:
7F83B1657FF1FC53B92DC18148A1D65DFC2D4B1FA3D677284ADDD200126D9069

Section 148.6: SHA384

```

using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{

```

```

class 程序
{
    static void Main(string[] args)
    {
        string source = "Hello World!";
        using (SHA384 sha384Hash = SHA384.Create())
        {
            //从字符串到字节数组
            byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
            byte[] hashBytes = sha384Hash.ComputeHash(sourceBytes);
            string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

Console.WriteLine("'" + source + "' 的 SHA384 哈希值是: " + hash);
        }
    }
}

```

输出：

"Hello World!" 的 SHA384 哈希值是：
BFD76C0EBBD006FEE583410547C1887B0292BE76D582D96C242D2A792723E3FD6FD061F9D5CFD13B8F961358E6A
DBA4A

第148.7节：SHA512

```

using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class 程序
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA512 sha512Hash = SHA512.Create())
            {
                //从字符串到字节数组
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha512Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

Console.WriteLine("'" + source + "' 的 SHA512 哈希值是: " + hash);
            }
        }
    }
}

```

输出: Hello World! 的 SHA512 哈希值是：
861844D6704E8573FEC34D967E20BCFEF3D424CF48BE04E6DC08F2BD58C729743371015EAD891CC3CF1C9D34B49
264B510751B1FF9E537937BC46B5D6FF4ECC8

```

class Program
{
    static void Main(string[] args)
    {
        string source = "Hello World!";
        using (SHA384 sha384Hash = SHA384.Create())
        {
            //From String to byte array
            byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
            byte[] hashBytes = sha384Hash.ComputeHash(sourceBytes);
            string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

Console.WriteLine("The SHA384 hash of " + source + " is: " + hash);
        }
    }
}

```

Output:

The SHA384 hash of Hello World! is:
BFD76C0EBBD006FEE583410547C1887B0292BE76D582D96C242D2A792723E3FD6FD061F9D5CFD13B8F961358E6A
DBA4A

Section 148.7: SHA512

```

using System;
using System.Security.Cryptography;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string source = "Hello World!";
            using (SHA512 sha512Hash = SHA512.Create())
            {
                //From String to byte array
                byte[] sourceBytes = Encoding.UTF8.GetBytes(source);
                byte[] hashBytes = sha512Hash.ComputeHash(sourceBytes);
                string hash = BitConverter.ToString(hashBytes).Replace("-", String.Empty);

Console.WriteLine("The SHA512 hash of " + source + " is: " + hash);
            }
        }
    }
}

```

Output: The SHA512 hash of Hello World! is:
861844D6704E8573FEC34D967E20BCFEF3D424CF48BE04E6DC08F2BD58C729743371015EAD891CC3CF1C9D34B49
264B510751B1FF9E537937BC46B5D6FF4ECC8

第149章：在 C# 中生成随机数

参数	详情
种子	用于生成随机数的值。如果未设置，默认值由当前系统时间决定。
最小值	生成的数字不会小于此值。如果未设置，默认值为0。
最大值	生成的数字将小于此值。如果未设置，默认值为Int32.MaxValue。
返回值	返回一个随机值的数字。

第149.1节：生成随机整数

此示例生成0到2147483647之间的随机值。

```
Random rnd = new Random();
int randomNumber = rnd.Next();
```

第149.2节：在给定范围内生成随机整数

生成一个介于minValue和maxValue - 1之间的随机数。

```
Random rnd = new Random();
var randomBetween10And20 = rnd.Next(10, 20);
```

第149.3节：重复生成相同的随机数序列

使用相同种子创建Random实例时，将生成相同的数字。

```
int seed = 5;
for (int i = 0; i < 2; i++)
{
    Console.WriteLine("随机实例 " + i);
    Random rnd = new Random(seed);
    for (int j = 0; j < 5; j++)
    {
        Console.Write(rnd.Next());
        Console.Write(" ");
    }
    Console.WriteLine();
}
```

输出：

```
随机实例 0
726643700 610783965 564707973 1342984399 995276750
随机实例 1
726643700 610783965 564707973 1342984399 995276750
```

Chapter 149: Generating Random Numbers in C#

Parameters	Details
Seed	A value for generating random numbers. If not set, the default value is determined by the current system time.
minValue	Generated numbers won't be smaller than this value. If not set, the default value is 0.
maxValue	Generated numbers will be smaller than this value. If not set, the default value is Int32.MaxValue.
return value	Returns a number with random value.

Section 149.1: Generate a random int

This example generates random values between 0 and 2147483647.

```
Random rnd = new Random();
int randomNumber = rnd.Next();
```

Section 149.2: Generate a random int in a given range

Generate a random number between minValue and maxValue - 1.

```
Random rnd = new Random();
var randomBetween10And20 = rnd.Next(10, 20);
```

Section 149.3: Generating the same sequence of random numbers over and over again

When creating Random instances with the same seed, the same numbers will be generated.

```
int seed = 5;
for (int i = 0; i < 2; i++)
{
    Console.WriteLine("Random instance " + i);
    Random rnd = new Random(seed);
    for (int j = 0; j < 5; j++)
    {
        Console.Write(rnd.Next());
        Console.Write(" ");
    }
    Console.WriteLine();
}
```

Output:

```
Random instance 0
726643700 610783965 564707973 1342984399 995276750
Random instance 1
726643700 610783965 564707973 1342984399 995276750
```

第149.4节：同时使用不同种子创建多个随机类

同时创建的两个 Random 类将具有相同的种子值。

使用 System.Guid.NewGuid().GetHashCode() 即使在同一时间也能获得不同的种子。

```
Random rnd1 = new Random();
Random rnd2 = new Random();
Console.WriteLine("rnd1 中的前 5 个随机数");
for (int i = 0; i < 5; i++)
Console.WriteLine(rnd1.Next());

Console.WriteLine("rnd2 中的前 5 个随机数");
for (int i = 0; i < 5; i++)
Console.WriteLine(rnd2.Next());

rnd1 = new Random(Guid.NewGuid().GetHashCode());
rnd2 = new Random(Guid.NewGuid().GetHashCode());
Console.WriteLine("使用 Guid 的 rnd1 中的前 5 个随机数");
for (int i = 0; i < 5; i++)
Console.WriteLine(rnd1.Next());
Console.WriteLine("使用 Guid 的 rnd2 中的前 5 个随机数");
for (int i = 0; i < 5; i++)
Console.WriteLine(rnd2.Next());
```

实现不同种子的另一种方法是使用另一个 Random 实例来获取种子值。

```
Random rndSeeds = new Random();
Random rnd1 = new Random(rndSeeds.Next());
Random rnd2 = new Random(rndSeeds.Next());
```

这也使得通过仅设置 rndSeeds 的种子值来控制所有 Random 实例的结果成为可能。所有其他实例都将从该单一种子值确定性地派生。

第149.5节：生成一个随机双精度数

生成一个介于0和1.0之间的随机数。（不包括1.0）

```
Random rnd = new Random();
var randomDouble = rnd.NextDouble();
```

第149.6节：生成一个随机字符

使用带有指定数字范围的Next()重载，生成一个介于'a'和'z'之间的随机字母，然后将得到的 int 转换为 char

```
Random rnd = new Random();
char randomChar = (char)rnd.Next('a', 'z');
// 'a' 和 'z' 作为 Next() 参数时被解释为 int 类型
```

第149.7节：生成一个最大值的百分比数

生成随机数的一个常见需求是生成某个最大值的X%。可以通过将NextDouble()的结果视为百分比来实现：

Section 149.4: Create multiple random class with different seeds simultaneously

Two Random class created at the same time will have the same seed value.

Using System.Guid.NewGuid().GetHashCode() can get a different seed even in the same time.

```
Random rnd1 = new Random();
Random rnd2 = new Random();
Console.WriteLine("First 5 random number in rnd1");
for (int i = 0; i < 5; i++)
Console.WriteLine(rnd1.Next());

Console.WriteLine("First 5 random number in rnd2");
for (int i = 0; i < 5; i++)
Console.WriteLine(rnd2.Next());

rnd1 = new Random(Guid.NewGuid().GetHashCode());
rnd2 = new Random(Guid.NewGuid().GetHashCode());
Console.WriteLine("First 5 random number in rnd1 using Guid");
for (int i = 0; i < 5; i++)
Console.WriteLine(rnd1.Next());
Console.WriteLine("First 5 random number in rnd2 using Guid");
for (int i = 0; i < 5; i++)
Console.WriteLine(rnd2.Next());
```

Another way to achieve different seeds is to use another Random instance to retrieve the seed values.

```
Random rndSeeds = new Random();
Random rnd1 = new Random(rndSeeds.Next());
Random rnd2 = new Random(rndSeeds.Next());
```

This also makes it possible to control the result of all the Random instances by setting only the seed value for the rndSeeds. All the other instances will be deterministically derived from that single seed value.

Section 149.5: Generate a Random double

Generate a random number between 0 and 1.0. (not including 1.0)

```
Random rnd = new Random();
var randomDouble = rnd.NextDouble();
```

Section 149.6: Generate a random character

Generate a random letter between a and z by using the Next() overload for a given range of numbers, then converting the resulting int to a char

```
Random rnd = new Random();
char randomChar = (char)rnd.Next('a', 'z');
// 'a' and 'z' are interpreted as ints for parameters for Next()
```

Section 149.7: Generate a number that is a percentage of a max value

A common need for random numbers is to generate a number that is X% of some max value. This can be done by treating the result of NextDouble() as a percentage:

```
var rnd = new Random();
var maxValue = 5000;
var percentage = rnd.NextDouble();
var result = maxValue * percentage;
//假设 NextDouble() 返回 0.65, result 将保存 5000 的 65%: 3250.
```

```
var rnd = new Random();
var maxValue = 5000;
var percentage = rnd.NextDouble();
var result = maxValue * percentage;
//suppose NextDouble() returns .65, result will hold 65% of 5000: 3250.
```

belindoc.com

第150章：密码学 (System.Security.Cryptography)

第150.1节：字符串对称认证加密的现代示例

密码学非常复杂，在花费大量时间阅读不同示例并看到引入某种漏洞是多么容易之后，我找到了一个由 @jbtule 最初撰写的答案，我认为非常好。请享用阅读：

对称加密的一般最佳实践是使用带关联数据的认证加密（AEAD），但这不是标准 .net 加密库的一部分。因此，第一个示例使用AES256然后是HMAC256，采用两步先加密再MAC的方式，这需要更多的开销和更多的密钥。

第二个示例使用更简单的 AES256-GCM，利用开源的 [Bouncy Castle](#) (通过 nuget)。

两个示例都有一个主函数，接受秘密消息字符串、密钥和一个可选的非秘密负载，并返回一个认证加密的字符串，选项地在前面加上非秘密数据。理想情况下，你会使用随机生成的 256 位密钥，参见NewKey()。

两个示例也都有辅助方法，使用字符串密码生成密钥。这些辅助方法作为方便与其他示例匹配而提供，但它们安全性远低于因为密码的强度远弱于256位密钥。

更新：添加了byte[]重载，且只有Gist包含完整格式，带4个空格缩进和API文档，因StackOverflow答案限制。

.NET 内置加密 (AES) -然后-MAC (HMAC) [[Gist](#)]

```
/*
 * 本作品 (James Tuley 的字符串现代加密 C#) ,
 * 由 James Tuley 识别, 已无已知版权限制。
 * https://gist.github.com/4336842
 * http://creativecommons.org/publicdomain/mark/1.0/
 */

using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;

namespace Encryption
{
    public static class AESThenHMAC
    {
        private static readonly RandomNumberGenerator Random = RandomNumberGenerator.Create();

        //预配置的加密参数
        public static readonly int BlockBitSize = 128;
        public static readonly int KeyBitSize = 256;

        //预配置的密码密钥派生参数
        public static readonly int SaltBitSize = 64;
        public static readonly int Iterations = 10000;
        public static readonly int MinPasswordLength = 12;
    }
}
```

Chapter 150: Cryptography (System.Security.Cryptography)

Section 150.1: Modern Examples of Symmetric Authenticated Encryption of a string

Cryptography is something very hard and after spending a lot of time reading different examples and seeing how easy it is to introduce some form of vulnerability I found an answer originally written by @jbtule that I think is very good. Enjoy reading:

"The general best practice for symmetric encryption is to use Authenticated Encryption with Associated Data (AEAD), however this isn't a part of the standard .net crypto libraries. So the first example uses [AES256](#) and then [HMAC256](#), a two step [Encrypt then MAC](#), which requires more overhead and more keys.

The second example uses the simpler practice of AES256-GCM using the open source Bouncy Castle (via nuget).

Both examples have a main function that takes secret message string, key(s) and an optional non-secret payload and return and authenticated encrypted string optionally prepended with the non-secret data. Ideally you would use these with 256bit key(s) randomly generated see NewKey().

Both examples also have a helper methods that use a string password to generate the keys. These helper methods are provided as a convenience to match up with other examples, however they are *far less secure* because the strength of the password is going to be *far weaker than a 256 bit key*.

Update: Added `byte[]` overloads, and only the [Gist](#) has the full formatting with 4 spaces indent and api docs due to StackOverflow answer limits."

.NET Built-in Encrypt(AES)-Then-MAC(HMAC) [[Gist](#)]

```
/*
 * This work (Modern Encryption of a String C#, by James Tuley),
 * identified by James Tuley, is free of known copyright restrictions.
 * https://gist.github.com/4336842
 * http://creativecommons.org/publicdomain/mark/1.0/
 */

using System;
using System.IO;
using System.Security.Cryptography;
using System.Text;

namespace Encryption
{
    public static class AESThenHMAC
    {
        private static readonly RandomNumberGenerator Random = RandomNumberGenerator.Create();

        //Preconfigured Encryption Parameters
        public static readonly int BlockBitSize = 128;
        public static readonly int KeyBitSize = 256;

        //Preconfigured Password Key Derivation Parameters
        public static readonly int SaltBitSize = 64;
        public static readonly int Iterations = 10000;
        public static readonly int MinPasswordLength = 12;
    }
}
```

```

/// <summary>
/// 每次调用时生成随机密钥的辅助方法。
/// </summary>
/// <returns></returns>
public static byte[] NewKey()
{
    var key = new byte[KeyBitSize / 8];
    Random.GetBytes(key);
    return key;
}

/// <summary>
/// 简单加密 (AES) 然后对UTF8消息进行认证 (HMAC) 。
/// </summary>
/// <param name="secretMessage">秘密消息。</param>
/// <param name="cryptKey">加密密钥。</param>
/// <param name="authKey">认证密钥。</param>
/// <param name="nonSecretPayload">(可选) 非秘密负载。</param>
/// <returns></returns>
/// 加密消息
/// </returns>
/// <exception cref="System.ArgumentException">需要秘密消息 ! ;secretMessage</exception>
/// <remarks></remarks>
/// 增加的开销为 (可选负载 + 块大小 (16) + 消息填充至块大小 + HMac标签 (32) ) * 1.33 Base64
/// <remarks>
public static string SimpleEncrypt(string secretMessage, byte[] cryptKey, byte[] authKey,
                                   byte[] nonSecretPayload = null)
{
    if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("需要秘密消息 ! ", "secretMessage");

    var plainText = Encoding.UTF8.GetBytes(secretMessage);
    var cipherText = SimpleEncrypt(plainText, cryptKey, authKey, nonSecretPayload);
    return Convert.ToString(cipherText);
}

/// <summary>
/// 简单认证 (HMAC) 然后解密 (AES) 用于秘密的UTF8消息。
/// </summary>
/// <param name="encryptedMessage">加密消息。</param>
/// <param name="cryptKey">加密密钥。</param>
/// <param name="authKey">认证密钥。</param>
/// <param name="nonSecretPayloadLength">非秘密负载长度。</param>
/// <returns>
/// 解密后的消息
/// </returns>
/// <exception cref="System.ArgumentException">需要加密消息 ! ;encryptedMessage</exception>
public static string SimpleDecrypt(string encryptedMessage, byte[] cryptKey, byte[] authKey,
                                   int nonSecretPayloadLength = 0)
{
    if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("需要加密消息 ! ", "encryptedMessage");

    var cipherText = Convert.FromBase64String(encryptedMessage);
    var plainText = SimpleDecrypt(cipherText, cryptKey, authKey, nonSecretPayloadLength);
    return plainText == null ? null : Encoding.UTF8.GetString(plainText);
}

/// <summary>
/// 对UTF8消息进行简单加密 (AES) 然后认证 (HMAC)

```

```

/// <summary>
/// Helper that generates a random key on each call.
/// </summary>
/// <returns></returns>
public static byte[] NewKey()
{
    var key = new byte[KeyBitSize / 8];
    Random.GetBytes(key);
    return key;
}

/// <summary>
/// Simple Encryption (AES) then Authentication (HMAC) for a UTF8 Message.
/// </summary>
/// <param name="secretMessage">The secret message.</param>
/// <param name="cryptKey">The crypt key.</param>
/// <param name="authKey">The auth key.</param>
/// <param name="nonSecretPayload">(Optional) Non-Secret Payload.</param>
/// <returns>
/// Encrypted Message
/// </returns>
/// <exception cref="System.ArgumentException">Secret Message Required! ;secretMessage</exception>
/// <remarks>
/// Adds overhead of (Optional-Payload + BlockSize(16) + Message-Padded-To-Blocksize + HMac-Tag(32)) * 1.33 Base64
/// </remarks>
public static string SimpleEncrypt(string secretMessage, byte[] cryptKey, byte[] authKey,
                                   byte[] nonSecretPayload = null)
{
    if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required! ", "secretMessage");

    var plainText = Encoding.UTF8.GetBytes(secretMessage);
    var cipherText = SimpleEncrypt(plainText, cryptKey, authKey, nonSecretPayload);
    return Convert.ToString(cipherText);
}

/// <summary>
/// Simple Authentication (HMAC) then Decryption (AES) for a secrets UTF8 Message.
/// </summary>
/// <param name="encryptedMessage">The encrypted message.</param>
/// <param name="cryptKey">The crypt key.</param>
/// <param name="authKey">The auth key.</param>
/// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
/// <returns>
/// Decrypted Message
/// </returns>
/// <exception cref="System.ArgumentException">Encrypted Message Required! ;encryptedMessage</exception>
public static string SimpleDecrypt(string encryptedMessage, byte[] cryptKey, byte[] authKey,
                                   int nonSecretPayloadLength = 0)
{
    if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required! ", "encryptedMessage");

    var cipherText = Convert.FromBase64String(encryptedMessage);
    var plainText = SimpleDecrypt(cipherText, cryptKey, authKey, nonSecretPayloadLength);
    return plainText == null ? null : Encoding.UTF8.GetString(plainText);
}

/// <summary>
/// Simple Encryption (AES) then Authentication (HMAC) of a UTF8 message

```

```

/// 使用从密码派生的密钥 (PBKDF2)。
/// </summary>
/// <param name="secretMessage">秘密消息。</param>
/// <param name="password">密码。</param>
/// <param name="nonSecretPayload">非秘密负载。</param>
/// <returns>
/// 加密消息
/// </returns>
/// <exception cref="System.ArgumentException">密码异常</exception>
/// <remarks>
/// 安全性明显低于使用随机二进制密钥。
/// 为密钥生成参数添加额外的非秘密负载。
/// </remarks>
public static string SimpleEncryptWithPassword(string secretMessage, string password,
                                              byte[] nonSecretPayload = null)
{
    if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("需要秘密消息！", "secretMessage");

    var plainText = Encoding.UTF8.GetBytes(secretMessage);
    var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
    return Convert.ToBase64String(cipherText);
}

/// <summary>
/// 使用从密码派生的密钥 (PBKDF2) 对UTF8消息进行简单认证 (HMAC) 然后解密 (AES)。
/// </summary>
/// <param name="encryptedMessage">加密消息。</param>
/// <param name="password">密码。</param>
/// <param name="nonSecretPayloadLength">非秘密负载的长度。</param>
/// <returns>
/// 解密后的消息
/// </returns>
/// <exception cref="System.ArgumentException">加密消息
必填! ;encryptedMessage</exception>
/// <remarks>
/// 安全性明显低于使用随机二进制密钥。
/// </remarks>
public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
                                                int nonSecretPayloadLength = 0)
{
    if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("需要加密消息！", "encryptedMessage");

    var cipherText = Convert.FromBase64String(encryptedMessage);
    var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
    return plainText == null ? null : Encoding.UTF8.GetString(plainText);
}

public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] cryptKey, byte[] authKey,
                                  byte[] nonSecretPayload = null)
{
    // 用户错误检查
    if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("密钥需要是 {0} 位！", KeyBitSize),
                                  "cryptKey");

    if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("密钥需要是 {0} 位！", KeyBitSize),
                                  "authKey");
}

```

```

/// using Keys derived from a Password (PBKDF2).
/// </summary>
/// <param name="secretMessage">The secret message.</param>
/// <param name="password">The password.</param>
/// <param name="nonSecretPayload">The non secret payload.</param>
/// <returns>
/// Encrypted Message
/// </returns>
/// <exception cref="System.ArgumentException">password</exception>
/// <remarks>
/// Significantly less secure than using random binary keys.
/// Adds additional non secret payload for key generation parameters.
/// </remarks>
public static string SimpleEncryptWithPassword(string secretMessage, string password,
                                               byte[] nonSecretPayload = null)
{
    if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

    var plainText = Encoding.UTF8.GetBytes(secretMessage);
    var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
    return Convert.ToBase64String(cipherText);
}

/// <summary>
/// Simple Authentication (HMAC) and then Decryption (AES) of a UTF8 Message
/// using keys derived from a password (PBKDF2).
/// </summary>
/// <param name="encryptedMessage">The encrypted message.</param>
/// <param name="password">The password.</param>
/// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
/// <returns>
/// Decrypted Message
/// </returns>
/// <exception cref="System.ArgumentException">Encrypted Message
Required! ;encryptedMessage</exception>
/// <remarks>
/// Significantly less secure than using random binary keys.
/// </remarks>
public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
                                               int nonSecretPayloadLength = 0)
{
    if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

    var cipherText = Convert.FromBase64String(encryptedMessage);
    var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
    return plainText == null ? null : Encoding.UTF8.GetString(plainText);
}

public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] cryptKey, byte[] authKey,
                                  byte[] nonSecretPayload = null)
{
    //User Error Checks
    if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
                                  "cryptKey");

    if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize),
                                  "authKey");
}

```

```

if (secretMessage == null || secretMessage.Length < 1)
    throw new ArgumentException("Secret Message Required!", "secretMessage");

//非秘密负载可选
nonSecretPayload = nonSecretPayload ?? new byte[] { };

byte[] cipherText;
byte[] iv;

using (var aes = new AesManaged
{
    KeySize = KeyBitSize,
    BlockSize = BlockBitSize,
    Mode = CipherMode.CBC,
    Padding = PaddingMode.PKCS7
})
{
    //使用随机IV
    aes.GenerateIV();
    iv = aes.IV;

    using (var encrypter = aes.CreateEncryptor(cryptKey, iv))
    using (var cipherStream = new MemoryStream())
    {
        using (var cryptoStream = new CryptoStream(cipherStream, encrypter,
            CryptoStreamMode.Write))
        使用 (var binaryWriter = new BinaryWriter(cryptoStream))
        {
            //加密数据
            binaryWriter.Write(secretMessage);
        }

        cipherText = cipherStream.ToArray();
    }
}

//组装加密消息并添加认证
使用 (var hmac = new HMACSHA256(authKey))
使用 (var encryptedStream = new MemoryStream())
{
    使用 (var binaryWriter = new BinaryWriter(encryptedStream))
    {
        //如果有，预先写入非秘密负载
        binaryWriter.Write(nonSecretPayload);
        //预先写入初始化向量 (IV)
        binaryWriter.Write(iv);
        //写入密文
        binaryWriter.Write(cipherText);
        binaryWriter.Flush();

        //验证所有数据
        var tag = hmac.ComputeHash(encryptedStream.ToArray());
        //附加标签
        binaryWriter.Write(tag);
    }
    return encryptedStream.ToArray();
}

```

```

if (secretMessage == null || secretMessage.Length < 1)
    throw new ArgumentException("Secret Message Required!", "secretMessage");

//non-secret payload optional
nonSecretPayload = nonSecretPayload ?? new byte[] { };

byte[] cipherText;
byte[] iv;

using (var aes = new AesManaged
{
    KeySize = KeyBitSize,
    BlockSize = BlockBitSize,
    Mode = CipherMode.CBC,
    Padding = PaddingMode.PKCS7
})
{
    //Use random IV
    aes.GenerateIV();
    iv = aes.IV;

    using (var encrypter = aes.CreateEncryptor(cryptKey, iv))
    using (var cipherStream = new MemoryStream())
    {
        using (var cryptoStream = new CryptoStream(cipherStream, encrypter,
            CryptoStreamMode.Write))
        using (var binaryWriter = new BinaryWriter(cryptoStream))
        {
            //Encrypt Data
            binaryWriter.Write(secretMessage);
        }

        cipherText = cipherStream.ToArray();
    }
}

//Assemble encrypted message and add authentication
using (var hmac = new HMACSHA256(authKey))
using (var encryptedStream = new MemoryStream())
{
    using (var binaryWriter = new BinaryWriter(encryptedStream))
    {
        //Prepend non-secret payload if any
        binaryWriter.Write(nonSecretPayload);
        //Prepend IV
        binaryWriter.Write(iv);
        //Write Ciphertext
        binaryWriter.Write(cipherText);
        binaryWriter.Flush();

        //Authenticate all data
        var tag = hmac.ComputeHash(encryptedStream.ToArray());
        //Postpend tag
        binaryWriter.Write(tag);
    }
    return encryptedStream.ToArray();
}

```

```

public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] cryptKey, byte[] authKey,
int nonSecretPayloadLength = 0)
{
    //基本使用错误检查
    if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("CryptKey 需要是 {0} 位!", KeyBitSize),
"cryptKey");

    if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("AuthKey 需要是 {0} 位!", KeyBitSize),
"authKey");

    if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("需要加密消息!", "encryptedMessage");

    using (var hmac = new HMACSHA256(authKey))
    {
        var sentTag = new byte[hmac.HashSize / 8];
        //计算标签
        var calcTag = hmac.ComputeHash(encryptedMessage, 0, encryptedMessage.Length -
sentTag.Length);
        var ivLength = (BlockBitSize / 8);

        //如果消息长度太小, 则直接返回 null
        if (encryptedMessage.Length < sentTag.Length + nonSecretPayloadLength + ivLength)
            return null;

        //获取发送的标签
        Array.Copy(encryptedMessage, encryptedMessage.Length - sentTag.Length, sentTag, 0,
sentTag.Length);

        // 使用常量时间比较标签
        var compare = 0;
        for (var i = 0; i < sentTag.Length; i++)
            compare |= sentTag[i] ^ calcTag[i];

        // 如果消息未通过认证则返回 null
        if (compare != 0)
            return null;

        using (var aes = new AesManaged
        {
            KeySize = KeyBitSize,
            BlockSize = BlockBitSize,
            Mode = CipherMode.CBC,
            Padding = PaddingMode.PKCS7
        })
        {

            // 从消息中获取 IV
            var iv = new byte[ivLength];
            Array.Copy(encryptedMessage, nonSecretPayloadLength, iv, 0, iv.Length);

            using (var decrypter = aes.CreateDecryptor(cryptKey, iv))
            using (var plainTextStream = new MemoryStream())
            {
                using (var 解密流 = new CryptoStream(明文流, 解密器,
CryptoStreamMode.Write))
                using (var 二进制写入器 = new BinaryWriter(解密流))
                {
                    //从消息中解密密文

```

```

public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] cryptKey, byte[] authKey,
int nonSecretPayloadLength = 0)
{
    //Basic Usage Error Checks
    if (cryptKey == null || cryptKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("CryptKey needs to be {0} bit!", KeyBitSize),
"cryptKey");

    if (authKey == null || authKey.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("AuthKey needs to be {0} bit!", KeyBitSize),
"authKey");

    if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

    using (var hmac = new HMACSHA256(authKey))
    {
        var sentTag = new byte[hmac.HashSize / 8];
        //Calculate Tag
        var calcTag = hmac.ComputeHash(encryptedMessage, 0, encryptedMessage.Length -
sentTag.Length);
        var ivLength = (BlockBitSize / 8);

        //if message length is too small just return null
        if (encryptedMessage.Length < sentTag.Length + nonSecretPayloadLength + ivLength)
            return null;

        //Grab Sent Tag
        Array.Copy(encryptedMessage, encryptedMessage.Length - sentTag.Length, sentTag, 0,
sentTag.Length);

        //Compare Tag with constant time comparison
        var compare = 0;
        for (var i = 0; i < sentTag.Length; i++)
            compare |= sentTag[i] ^ calcTag[i];

        //if message doesn't authenticate return null
        if (compare != 0)
            return null;

        using (var aes = new AesManaged
        {
            KeySize = KeyBitSize,
            BlockSize = BlockBitSize,
            Mode = CipherMode.CBC,
            Padding = PaddingMode.PKCS7
        })
        {

            //Grab IV from message
            var iv = new byte[ivLength];
            Array.Copy(encryptedMessage, nonSecretPayloadLength, iv, 0, iv.Length);

            using (var decrypter = aes.CreateDecryptor(cryptKey, iv))
            using (var plainTextStream = new MemoryStream())
            {
                using (var decrypterStream = new CryptoStream(plainTextStream, decrypter,
CryptoStreamMode.Write))
                using (var binaryWriter = new BinaryWriter(decrypterStream))
                {
                    //Decrypt Cipher Text from Message

```

```

二进制写入器.Write(
    加密消息,
    非秘密负载长度 + iv.Length,
    加密消息.Length - 非秘密负载长度 - iv.Length - 发送标签.Length
);
}

//返回明文
return 明文流.ToArray();
}

}

public static byte[] SimpleEncryptWithPassword(byte[] 秘密消息, string 密码, byte[]
非秘密负载 = null)
{
    nonSecretPayload = nonSecretPayload ?? new byte[] {};

    // 用户错误检查
    if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("密码长度必须至少为 {0} 个字符 !", MinPasswordLength), "password");

    if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("需要秘密信息 !", "secretMessage");

    var payload = new byte[((SaltBitSize / 8) * 2) + nonSecretPayload.Length];

    Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
    int payloadIndex = nonSecretPayload.Length;

    byte[] cryptKey;
    byte[] authKey;
    //使用随机盐以防止预生成的弱密码攻击。
    using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
    {
        var salt = generator.Salt;

        //生成密钥
        cryptKey = generator.GetBytes(KeyBitSize / 8);

        //创建非秘密负载
        Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
        payloadIndex += salt.Length;
    }

    //派生独立密钥，可能效率低于使用HKDF，//但现在兼容RNEncryptor，其线格式非常相似且比HKDF需要更少代码。
    using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
    {
        var salt = generator.Salt;

        //生成密钥
        authKey = generator.GetBytes(KeyBitSize / 8);

        //创建其余非秘密负载
        Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
    }

    return SimpleEncrypt(secretMessage, cryptKey, authKey, payload);
}

```

```

binaryWriter.Write(
    encryptedMessage,
    nonSecretPayloadLength + iv.Length,
    encryptedMessage.Length - nonSecretPayloadLength - iv.Length - sentTag.Length
);
}

//Return Plain Text
return plainTextStream.ToArray();
}

}

public static byte[] SimpleEncryptWithPassword(byte[] secretMessage, string password, byte[]
nonSecretPayload = null)
{
    nonSecretPayload = nonSecretPayload ?? new byte[] {};

    //User Error Checks
    if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0} characters!", MinPasswordLength), "password");

    if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

    var payload = new byte[((SaltBitSize / 8) * 2) + nonSecretPayload.Length];

    Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
    int payloadIndex = nonSecretPayload.Length;

    byte[] cryptKey;
    byte[] authKey;
    //Use Random Salt to prevent pre-generated weak password attacks.
    using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
    {
        var salt = generator.Salt;

        //Generate Keys
        cryptKey = generator.GetBytes(KeyBitSize / 8);

        //Create Non Secret Payload
        Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
        payloadIndex += salt.Length;
    }

    //Deriving separate key, might be less efficient than using HKDF,
    //but now compatible with RNEncryptor which had a very similar wireformat and requires less code than HKDF.
    using (var generator = new Rfc2898DeriveBytes(password, SaltBitSize / 8, Iterations))
    {
        var salt = generator.Salt;

        //Generate Keys
        authKey = generator.GetBytes(KeyBitSize / 8);

        //Create Rest of Non Secret Payload
        Array.Copy(salt, 0, payload, payloadIndex, salt.Length);
    }

    return SimpleEncrypt(secretMessage, cryptKey, authKey, payload);
}

```

```

    public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password, int
nonSecretPayloadLength = 0)
{
    // 用户错误检查
    if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("密码长度必须至少为 {0} 个字符 !", MinPasswordLength), "password");

    if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

    var cryptSalt = new byte[SaltBitSize / 8];
    var authSalt = new byte[SaltBitSize / 8];

    //从非机密有效载荷中获取盐值
    Array.Copy(encryptedMessage, nonSecretPayloadLength, cryptSalt, 0, cryptSalt.Length);
    Array.Copy(encryptedMessage, nonSecretPayloadLength + cryptSalt.Length, authSalt, 0,
authSalt.Length);

    byte[] cryptKey;
    byte[] authKey;

    //生成加密密钥
    using (var generator = new Rfc2898DeriveBytes(password, cryptSalt, Iterations))
    {
        cryptKey = generator.GetBytes(KeyBitSize / 8);
    }
    //生成认证密钥
    using (var generator = new Rfc2898DeriveBytes(password, authSalt, Iterations))
    {
        authKey = generator.GetBytes(KeyBitSize / 8);
    }

    return SimpleDecrypt(encryptedMessage, cryptKey, authKey, cryptSalt.Length + authSalt.Length
+nonSecretPayloadLength);
}
}
}

```

Bouncy Castle AES-GCM [Gist]

```

/*
* 本作品 (James Tuley 的字符串现代加密 C#) ,
* 由 James Tuley 识别, 已无已知版权限制。
* https://gist.github.com/4336842
* http://creativecommons.org/publicdomain/mark/1.0/
*/
using System;
using System.IO;
using System.Text;
using Org.BouncyCastle.Crypto;
using Org.BouncyCastle.Crypto.Engines;
using Org.BouncyCastle.Crypto.Generators;
using Org.BouncyCastle.Crypto.Modes;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Security;
namespace Encryption
{

    public static class AESGCM

```

```

    public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password, int
nonSecretPayloadLength = 0)
{
    //User Error Checks
    if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0} characters!", MinPasswordLength), "password");

    if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

    var cryptSalt = new byte[SaltBitSize / 8];
    var authSalt = new byte[SaltBitSize / 8];

    //Grab Salt from Non-Secret Payload
    Array.Copy(encryptedMessage, nonSecretPayloadLength, cryptSalt, 0, cryptSalt.Length);
    Array.Copy(encryptedMessage, nonSecretPayloadLength + cryptSalt.Length, authSalt, 0,
authSalt.Length);

    byte[] cryptKey;
    byte[] authKey;

    //Generate crypt key
    using (var generator = new Rfc2898DeriveBytes(password, cryptSalt, Iterations))
    {
        cryptKey = generator.GetBytes(KeyBitSize / 8);
    }
    //Generate auth key
    using (var generator = new Rfc2898DeriveBytes(password, authSalt, Iterations))
    {
        authKey = generator.GetBytes(KeyBitSize / 8);
    }

    return SimpleDecrypt(encryptedMessage, cryptKey, authKey, cryptSalt.Length + authSalt.Length
+ nonSecretPayloadLength);
}
}
}

```

Bouncy Castle AES-GCM [Gist]

```

/*
* This work (Modern Encryption of a String C#, by James Tuley),
* identified by James Tuley, is free of known copyright restrictions.
* https://gist.github.com/4336842
* http://creativecommons.org/publicdomain/mark/1.0/
*/
using System;
using System.IO;
using System.Text;
using Org.BouncyCastle.Crypto;
using Org.BouncyCastle.Crypto.Engines;
using Org.BouncyCastle.Crypto.Generators;
using Org.BouncyCastle.Crypto.Modes;
using Org.BouncyCastle.Crypto.Parameters;
using Org.BouncyCastle.Security;
namespace Encryption
{

    public static class AESGCM

```

```

{
    private static readonly SecureRandom Random = new SecureRandom();

    //预配置的加密参数
    public static readonly int NonceBitSize = 128;
    public static readonly int MacBitSize = 128;
    public static readonly int KeyBitSize = 256;

    //预配置的密码密钥派生参数
    public static readonly int SaltBitSize = 128;
    public static readonly int Iterations = 10000;
    public static readonly int MinPasswordLength = 12;

    /// <summary>
    /// 辅助方法，每次调用生成一个随机新密钥。
    /// </summary>
    /// <returns></returns>
    public static byte[] NewKey()
    {
        var key = new byte[KeyBitSize / 8];
        Random.NextBytes(key);
        return key;
    }

    /// <summary>
    /// 对UTF8字符串进行简单加密和认证 (AES-GCM)。
    /// </summary>
    /// <param name="secretMessage">秘密信息。</param>
    /// <param name="key">密钥。</param>
    /// <param name="nonSecretPayload">可选的非秘密负载。</param>
    /// <returns>
    /// 加密消息
    /// </returns>
    /// <exception cref="System.ArgumentException">需要秘密消息！;secretMessage</exception>
    /// <remarks></remarks>
    /// 添加开销为 (可选负载 + 块大小(16) + 消息 + HMac标签(16)) * 1.33 Base64
    /// </remarks>
    public static string SimpleEncrypt(string secretMessage, byte[] key, byte[] nonSecretPayload =
null)
    {
        if (string.IsNullOrEmpty(secretMessage))
            throw new ArgumentException("需要秘密信息！", "secretMessage");

        var plainText = Encoding.UTF8.GetBytes(secretMessage);
        var cipherText = SimpleEncrypt(plainText, key, nonSecretPayload);
        return Convert.ToString(cipherText);
    }

    /// <summary>
    /// 简单解密与认证 (AES-GCM) 针对UTF8消息
    /// </summary>
    /// <param name="encryptedMessage">加密消息。</param>
    /// <param name="key">密钥。</param>
    /// <param name="nonSecretPayloadLength">可选非秘密负载的长度。</param>
    /// <returns>解密后的消息</returns>
    public static string SimpleDecrypt(string encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
    {
        if (string.IsNullOrEmpty(encryptedMessage))
            throw new ArgumentException("需要加密消息！", "encryptedMessage");
    }
}

```

```

{
    private static readonly SecureRandom Random = new SecureRandom();

    //Preconfigured Encryption Parameters
    public static readonly int NonceBitSize = 128;
    public static readonly int MacBitSize = 128;
    public static readonly int KeyBitSize = 256;

    //Preconfigured Password Key Derivation Parameters
    public static readonly int SaltBitSize = 128;
    public static readonly int Iterations = 10000;
    public static readonly int MinPasswordLength = 12;

    /// <summary>
    /// Helper that generates a random new key on each call.
    /// </summary>
    /// <returns></returns>
    public static byte[] NewKey()
    {
        var key = new byte[KeyBitSize / 8];
        Random.NextBytes(key);
        return key;
    }

    /// <summary>
    /// Simple Encryption And Authentication (AES-GCM) of a UTF8 string.
    /// </summary>
    /// <param name="secretMessage">The secret message.</param>
    /// <param name="key">The key.</param>
    /// <param name="nonSecretPayload">Optional non-secret payload.</param>
    /// <returns>
    /// Encrypted Message
    /// </returns>
    /// <exception cref="System.ArgumentException">Secret Message Required!;secretMessage</exception>
    /// <remarks>
    /// Adds overhead of (Optional-Payload + BlockSize(16) + Message + HMac-Tag(16)) * 1.33 Base64
    /// </remarks>
    public static string SimpleEncrypt(string secretMessage, byte[] key, byte[] nonSecretPayload =
null)
    {
        if (string.IsNullOrEmpty(secretMessage))
            throw new ArgumentException("Secret Message Required!", "secretMessage");

        var plainText = Encoding.UTF8.GetBytes(secretMessage);
        var cipherText = SimpleEncrypt(plainText, key, nonSecretPayload);
        return Convert.ToString(cipherText);
    }

    /// <summary>
    /// Simple Decryption & Authentication (AES-GCM) of a UTF8 Message
    /// </summary>
    /// <param name="encryptedMessage">The encrypted message.</param>
    /// <param name="key">The key.</param>
    /// <param name="nonSecretPayloadLength">Length of the optional non-secret payload.</param>
    /// <returns>Decrypted Message</returns>
    public static string SimpleDecrypt(string encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
    {
        if (string.IsNullOrEmpty(encryptedMessage))
            throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");
    }
}

```

```

var cipherText = Convert.FromBase64String(encryptedMessage);
var plainText = SimpleDecrypt(cipherText, key, nonSecretPayloadLength);
return plainText == null ? null : Encoding.UTF8.GetString(plainText);
}

/// <summary>
/// 使用从密码派生的密钥 (PBKDF2) 对UTF8字符串进行简单加密和认证 (AES-GCM)
///
/// </summary>
/// <param name="secretMessage">秘密消息。</param>
/// <param name="password">密码。</param>
/// <param name="nonSecretPayload">非秘密负载。</param>
/// <returns>
/// 加密消息
/// </returns>
/// <remarks>
/// 安全性明显低于使用随机二进制密钥。
/// 为密钥生成参数添加额外的非秘密负载。
/// </remarks>
public static string SimpleEncryptWithPassword(string secretMessage, string password,
byte[] nonSecretPayload = null)
{
    if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("需要秘密消息！", "secretMessage");

    var plainText = Encoding.UTF8.GetBytes(secretMessage);
    var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
    return Convert.ToString(cipherText);
}

/// <summary>
/// 使用从密码派生的密钥 (PBKDF2) 对UTF8消息进行简单解密和认证 (AES-GCM) 。
///
/// </summary>
/// <param name="encryptedMessage">加密消息。</param>
/// <param name="password">密码。</param>
/// <param name="nonSecretPayloadLength">非秘密负载的长度。</param>
/// <returns>
/// 解密后的消息
/// </returns>
/// <exception cref="System.ArgumentException">加密消息
必填！;encryptedMessage</exception>
/// <remarks>
/// 安全性明显低于使用随机二进制密钥。
/// </remarks>
public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
int nonSecretPayloadLength = 0)
{
    if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("需要加密消息！", "encryptedMessage");

    var cipherText = Convert.FromBase64String(encryptedMessage);
    var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
    return plainText == null ? null : Encoding.UTF8.GetString(plainText);
}

public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] key, byte[] nonSecretPayload =
null)
{
    // 用户错误检查
    if (key == null || key.Length != KeyBitSize / 8)
}

```

```

var cipherText = Convert.FromBase64String(encryptedMessage);
var plainText = SimpleDecrypt(cipherText, key, nonSecretPayloadLength);
return plainText == null ? null : Encoding.UTF8.GetString(plainText);
}

/// <summary>
/// Simple Encryption And Authentication (AES-GCM) of a UTF8 String
/// using key derived from a password (PBKDF2).
/// </summary>
/// <param name="secretMessage">The secret message.</param>
/// <param name="password">The password.</param>
/// <param name="nonSecretPayload">The non secret payload.</param>
/// <returns>
/// Encrypted Message
/// </returns>
/// <remarks>
/// Significantly less secure than using random binary keys.
/// Adds additional non secret payload for key generation parameters.
/// </remarks>
public static string SimpleEncryptWithPassword(string secretMessage, string password,
byte[] nonSecretPayload = null)
{
    if (string.IsNullOrEmpty(secretMessage))
        throw new ArgumentException("Secret Message Required!", "secretMessage");

    var plainText = Encoding.UTF8.GetBytes(secretMessage);
    var cipherText = SimpleEncryptWithPassword(plainText, password, nonSecretPayload);
    return Convert.ToString(cipherText);
}

/// <summary>
/// Simple Decryption and Authentication (AES-GCM) of a UTF8 message
/// using a key derived from a password (PBKDF2)
/// </summary>
/// <param name="encryptedMessage">The encrypted message.</param>
/// <param name="password">The password.</param>
/// <param name="nonSecretPayloadLength">Length of the non secret payload.</param>
/// <returns>
/// Decrypted Message
/// </returns>
/// <exception cref="System.ArgumentException">Encrypted Message
Required!;encryptedMessage</exception>
/// <remarks>
/// Significantly less secure than using random binary keys.
/// </remarks>
public static string SimpleDecryptWithPassword(string encryptedMessage, string password,
int nonSecretPayloadLength = 0)
{
    if (string.IsNullOrWhiteSpace(encryptedMessage))
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

    var cipherText = Convert.FromBase64String(encryptedMessage);
    var plainText = SimpleDecryptWithPassword(cipherText, password, nonSecretPayloadLength);
    return plainText == null ? null : Encoding.UTF8.GetString(plainText);
}

public static byte[] SimpleEncrypt(byte[] secretMessage, byte[] key, byte[] nonSecretPayload =
null)
{
    //User Error Checks
    if (key == null || key.Length != KeyBitSize / 8)
}

```

```

throw new ArgumentException(String.Format("密钥需要是 {0} 位!", KeyBitSize), "key");

if (secretMessage == null || secretMessage.Length == 0)
    throw new ArgumentException("需要秘密信息!", "secretMessage");

//非秘密负载可选
nonSecretPayload = nonSecretPayload ?? new byte[] { };

//使用足够大的随机随机数以避免重复
var nonce = new byte[NonceBitSize / 8];
Random.NextBytes(nonce, 0, nonce.Length);

var cipher = new GcmBlockCipher(new AesFastEngine());
var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
cipher.Init(true, parameters);

//生成带认证标签的密文
var cipherText = new byte[cipher.GetOutputSize(secretMessage.Length)];
var len = cipher.ProcessBytes(secretMessage, 0, secretMessage.Length, cipherText, 0);
cipher.DoFinal(cipherText, len);

//组装消息
using (var combinedStream = new MemoryStream())
{
    using (var binaryWriter = new BinaryWriter(combinedStream))
    {
        //添加认证负载
        binaryWriter.Write(nonSecretPayload);
        //添加随机数
        binaryWriter.Write(nonce);
        //写入密文
        binaryWriter.Write(cipherText);
    }
    return combinedStream.ToArray();
}

public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
{
    // 用户错误检查
    if (key == null || key.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize), "key");

    if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

    using (var cipherStream = new MemoryStream(encryptedMessage))
    using (var cipherReader = new BinaryReader(cipherStream))
    {
        //抓取有效载荷
        var nonSecretPayload = cipherReader.ReadBytes(nonSecretPayloadLength);

        //获取随机数 (Nonce)
        var nonce = cipherReader.ReadBytes(NonceBitSize / 8);

        var cipher = new GcmBlockCipher(new AesFastEngine());
        var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
        cipher.Init(false, parameters);
    }
}

```

```

throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize), "key");

if (secretMessage == null || secretMessage.Length == 0)
    throw new ArgumentException("Secret Message Required!", "secretMessage");

//Non-secret Payload Optional
nonSecretPayload = nonSecretPayload ?? new byte[] { };

//Using random nonce large enough not to repeat
var nonce = new byte[NonceBitSize / 8];
Random.NextBytes(nonce, 0, nonce.Length);

var cipher = new GcmBlockCipher(new AesFastEngine());
var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
cipher.Init(true, parameters);

//Generate Cipher Text With Auth Tag
var cipherText = new byte[cipher.GetOutputSize(secretMessage.Length)];
var len = cipher.ProcessBytes(secretMessage, 0, secretMessage.Length, cipherText, 0);
cipher.DoFinal(cipherText, len);

//Assemble Message
using (var combinedStream = new MemoryStream())
{
    using (var binaryWriter = new BinaryWriter(combinedStream))
    {
        //Prepend Authenticated Payload
        binaryWriter.Write(nonSecretPayload);
        //Prepend Nonce
        binaryWriter.Write(nonce);
        //Write Cipher Text
        binaryWriter.Write(cipherText);
    }
    return combinedStream.ToArray();
}

public static byte[] SimpleDecrypt(byte[] encryptedMessage, byte[] key, int
nonSecretPayloadLength = 0)
{
    //User Error Checks
    if (key == null || key.Length != KeyBitSize / 8)
        throw new ArgumentException(String.Format("Key needs to be {0} bit!", KeyBitSize), "key");

    if (encryptedMessage == null || encryptedMessage.Length == 0)
        throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

    using (var cipherStream = new MemoryStream(encryptedMessage))
    using (var cipherReader = new BinaryReader(cipherStream))
    {
        //Grab Payload
        var nonSecretPayload = cipherReader.ReadBytes(nonSecretPayloadLength);

        //Grab Nonce
        var nonce = cipherReader.ReadBytes(NonceBitSize / 8);

        var cipher = new GcmBlockCipher(new AesFastEngine());
        var parameters = new AeadParameters(new KeyParameter(key), MacBitSize, nonce,
nonSecretPayload);
        cipher.Init(false, parameters);
    }
}

```

```

//解密密文
var cipherText = cipherReader.ReadBytes(encryptedMessage.Length - nonSecretPayloadLength -
nonce.Length);
var plainText = new byte[cipher.GetOutputSize(cipherText.Length)];

try
{
    var len = cipher.ProcessBytes(cipherText, 0, cipherText.Length, plainText, 0);
    cipher.DoFinal(plainText, len);
}

catch (InvalidCipherTextException)
{
    //如果未通过认证则返回null
    return null;
}

return plainText;
}

public static byte[] SimpleEncryptWithPassword(byte[] 秘密消息, string 密码, byte[]
非秘密负载 = null)
{
    nonSecretPayload = nonSecretPayload ?? new byte[] {};

    // 用户错误检查
    if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("密码长度必须至少为 {0} 个字符 !", MinPasswordLength), "password");

    if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("需要密文消息 !", "secretMessage");

    var generator = new Pkcs5S2ParametersGenerator();

    //使用随机盐以减少预生成弱密码攻击。
    var salt = new byte[SaltBitSize / 8];
    Random.NextBytes(salt);

    generator.Init(
        PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
        salt,
        Iterations);

    //生成密钥
    var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

    //创建完整的非秘密负载
    var payload = new byte[salt.Length + nonSecretPayload.Length];
    Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
    Array.Copy(salt, 0, payload, nonSecretPayload.Length, salt.Length);

    return SimpleEncrypt(secretMessage, key.GetKey(), payload);
}

public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password, int
nonSecretPayloadLength = 0)
{
    // 用户错误检查
    if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)

```

```

//Decrypt Cipher Text
var cipherText = cipherReader.ReadBytes(encryptedMessage.Length - nonSecretPayloadLength -
nonce.Length);
var plainText = new byte[cipher.GetOutputSize(cipherText.Length)];

try
{
    var len = cipher.ProcessBytes(cipherText, 0, cipherText.Length, plainText, 0);
    cipher.DoFinal(plainText, len);
}

catch (InvalidCipherTextException)
{
    //Return null if it doesn't authenticate
    return null;
}

return plainText;
}

public static byte[] SimpleEncryptWithPassword(byte[] secretMessage, string password, byte[]
nonSecretPayload = null)
{
    nonSecretPayload = nonSecretPayload ?? new byte[] {};

    //User Error Checks
    if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)
        throw new ArgumentException(String.Format("Must have a password of at least {0} characters!", MinPasswordLength), "password");

    if (secretMessage == null || secretMessage.Length == 0)
        throw new ArgumentException("Secret Message Required!", "secretMessage");

    var generator = new Pkcs5S2ParametersGenerator();

    //Use Random Salt to minimize pre-generated weak password attacks.
    var salt = new byte[SaltBitSize / 8];
    Random.NextBytes(salt);

    generator.Init(
        PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
        salt,
        Iterations);

    //Generate Key
    var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

    //Create Full Non Secret Payload
    var payload = new byte[salt.Length + nonSecretPayload.Length];
    Array.Copy(nonSecretPayload, payload, nonSecretPayload.Length);
    Array.Copy(salt, 0, payload, nonSecretPayload.Length, salt.Length);

    return SimpleEncrypt(secretMessage, key.GetKey(), payload);
}

public static byte[] SimpleDecryptWithPassword(byte[] encryptedMessage, string password, int
nonSecretPayloadLength = 0)
{
    //User Error Checks
    if (string.IsNullOrWhiteSpace(password) || password.Length < MinPasswordLength)

```

```

        throw new ArgumentException(String.Format("密码长度必须至少为 {0} 
字符！", MinPasswordLength), "password");

if (encryptedMessage == null || encryptedMessage.Length == 0)
    throw new ArgumentException("需要加密消息！", "encryptedMessage");

var generator = new Pkcs5S2ParametersGenerator();

//从负载中获取盐值
var salt = new byte[SaltBitSize / 8];
Array.Copy(encryptedMessage, nonSecretPayloadLength, salt, 0, salt.Length);

generator.Init(
PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
salt,
Iterations);

//生成密钥
var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

return SimpleDecrypt(encryptedMessage, key.GetKey(), salt.Length + nonSecretPayloadLength);
}
}
}

```

第150.2节：对称加密和非对称加密简介

您可以通过实施加密技术来提高数据传输或存储的安全性。基本上，使用System.Security.Cryptography时有两种方法：对称加密和非对称加密。

对称加密

此方法使用私钥来执行数据转换。

优点：

- 对称算法消耗的资源较少，速度比非对称算法快。
- 您可以加密的数据量没有限制。

缺点：

- 加密和解密使用相同的密钥。如果密钥被泄露，别人就能解密您的数据。
- 如果您选择为不同数据使用不同的密钥，可能会导致需要管理许多不同的密钥。

在System.Security.Cryptography中，有不同的类执行对称加密，它们被称为分组密码：

- [AesManaged \(AES算法\)](#)。
- [AesCryptoServiceProvider \(AES算法符合FIPS 140-2标准\)](#)。
- [DESCryptoServiceProvider \(DES算法\)](#)。
- [RC2CryptoServiceProvider \(Rivest Cipher 2算法\)](#)。
- [RijndaelManaged \(AES算法\)](#)。注意：RijndaelManaged 不符合FIPS-197标准。
- [TripleDES \(TripleDES算法\)](#)。

```

        throw new ArgumentException(String.Format("Must have a password of at least {0} 
characters！", MinPasswordLength), "password");

if (encryptedMessage == null || encryptedMessage.Length == 0)
    throw new ArgumentException("Encrypted Message Required!", "encryptedMessage");

var generator = new Pkcs5S2ParametersGenerator();

//Grab Salt from Payload
var salt = new byte[SaltBitSize / 8];
Array.Copy(encryptedMessage, nonSecretPayloadLength, salt, 0, salt.Length);

generator.Init(
PbeParametersGenerator.Pkcs5PasswordToBytes(password.ToCharArray()),
salt,
Iterations);

//Generate Key
var key = (KeyParameter)generator.GenerateDerivedMacParameters(KeyBitSize);

return SimpleDecrypt(encryptedMessage, key.GetKey(), salt.Length + nonSecretPayloadLength);
}
}
}

```

Section 150.2: Introduction to Symmetric and Asymmetric Encryption

You can improve the security for data transit or storing by implementing encrypting techniques. Basically there are two approaches when using *System.Security.Cryptography*: **symmetric** and **asymmetric**.

Symmetric Encryption

This method uses a private key in order to perform the data transformation.

Pros:

- Symmetric algorithms consume less resources and are faster than asymmetric ones.
- The amount of data you can encrypt is unlimited.

Cons:

- Encryption and decryption use the same key. Someone will be able to decrypt your data if the key is compromised.
- You could end up with many different secret keys to manage if you choose to use a different secret key for different data.

Under System.Security.Cryptography you have different classes that perform symmetric encryption, they are known as [block ciphers](#):

- [AesManaged \(AES algorithm\)](#).
- [AesCryptoServiceProvider \(AES algorithm FIPS 140-2 compliant\)](#).
- [DESCryptoServiceProvider \(DES algorithm\)](#).
- [RC2CryptoServiceProvider \(Rivest Cipher 2 algorithm\)](#).
- [RijndaelManaged \(AES algorithm\)](#). Note: RijndaelManaged is **not** FIPS-197 compliant.
- [TripleDES \(TripleDES algorithm\)](#).

非对称加密

该方法使用公钥和私钥的组合来执行数据转换。

优点：

- 它使用比对称算法更大的密钥，因此不易被暴力破解。
- 由于依赖两个密钥（公钥和私钥），因此更容易保证谁能够加密和解密数据。

缺点：

- 您可以加密的数据量是有限制的。该限制因算法而异，通常与算法的密钥大小成正比。例如，密钥长度为1024位的RSACryptoServiceProvider对象只能加密小于128字节的消息。
- 与对称算法相比，非对称算法非常慢。

在System.Security.Cryptography下，您可以访问执行非对称加密的不同类：

- [DSACryptoServiceProvider \(数字签名算法算法\)](#)
- [RSACryptoServiceProvider \(RSA算法算法\)](#)

第150.3节：简单的对称文件加密

以下代码示例演示了使用AES对称加密算法快速简便地加密和解密文件的方法。

代码每次加密文件时都会随机生成盐值和初始化向量，这意味着使用相同密码加密相同文件时，输出总是不同的。盐值和初始化向量会写入输出文件，因此解密时只需密码即可。

```
public static void ProcessFile(string inputPath, string password, bool encryptMode, string outputPath)
{
    using (var cypher = new AesManaged())
    using (var fsIn = new FileStream(inputPath, FileMode.Open))
    using (var fsOut = new FileStream(outputPath, FileMode.Create))
    {
        const int saltLength = 256;
        var salt = new byte[saltLength];
        var iv = new byte[cypher.BlockSize / 8];

        if (encryptMode)
        {
            // 生成随机盐值和初始化向量 (IV)，然后写入文件
            using (var rng = new RNGCryptoServiceProvider())
            {
                rng.GetBytes(salt);
                rng.GetBytes(iv);
            }
            fsOut.Write(salt, 0, salt.Length);
            fsOut.Write(iv, 0, iv.Length);
        }
        else
        {
            // 从文件中读取盐值和初始化向量 (IV)
            fsIn.Read(salt, 0, saltLength);
        }
    }
}
```

Asymmetric Encryption

This method uses a combination of public and private keys in order to perform the data transformation.

Pros:

- It uses larger keys than symmetric algorithms, thus they are less susceptible to being cracked by using brute force.
- It is easier to guarantee who is able to encrypt and decrypt the data because it relies on two keys (public and private).

Cons:

- There is a limit on the amount of data that you can encrypt. The limit is different for each algorithm and is typically proportional with the key size of the algorithm. For example, an RSACryptoServiceProvider object with a key length of 1,024 bits can only encrypt a message that is smaller than 128 bytes.
- Asymmetric algorithms are very slow in comparison to symmetric algorithms.

Under System.Security.Cryptography you have access to different classes that perform asymmetric encryption:

- [DSACryptoServiceProvider \(Digital Signature Algorithm algorithm\)](#)
- [RSACryptoServiceProvider \(RSA Algorithm algorithm\)](#)

Section 150.3: Simple Symmetric File Encryption

The following code sample demonstrates a quick and easy means of encrypting and decrypting files using the AES symmetric encryption algorithm.

The code randomly generates the Salt and Initialization Vectors each time a file is encrypted, meaning that encrypting the same file with the same password will always lead to different output. The salt and IV are written to the output file so that only the password is required to decrypt it.

```
public static void ProcessFile(string inputPath, string password, bool encryptMode, string outputPath)
{
    using (var cypher = new AesManaged())
    using (var fsIn = new FileStream(inputPath, FileMode.Open))
    using (var fsOut = new FileStream(outputPath, FileMode.Create))
    {
        const int saltLength = 256;
        var salt = new byte[saltLength];
        var iv = new byte[cypher.BlockSize / 8];

        if (encryptMode)
        {
            // Generate random salt and IV, then write them to file
            using (var rng = new RNGCryptoServiceProvider())
            {
                rng.GetBytes(salt);
                rng.GetBytes(iv);
            }
            fsOut.Write(salt, 0, salt.Length);
            fsOut.Write(iv, 0, iv.Length);
        }
        else
        {
            // Read the salt and IV from the file
            fsIn.Read(salt, 0, saltLength);
        }
    }
}
```

```

fsIn.Read(iv, 0, iv.Length);
}

// 基于提供的密码和盐值生成安全密码
var pdb = new Rfc2898DeriveBytes(password, salt);
var key = pdb.GetBytes(cypher.KeySize / 8);

// 加密或解密文件
using (var cryptoTransform = encryptMode
    ? cypher.CreateEncryptor(key, iv)
    : cypher.CreateDecryptor(key, iv))
using (var cs = new CryptoStream(fsOut, cryptoTransform, CryptoStreamMode.Write))
{
    fsIn.CopyTo(cs);
}
}
}

```

第150.4节：密码学安全的随机数据

有时框架的Random()类可能不被认为足够随机，因为它基于伪随机数生成器。然而，框架的Crypto类确实提供了更强大的功能，即RNGCryptoServiceProvider。

以下代码示例演示了如何生成密码学安全的字节数组、字符串和数字。

随机字节数组

```

public static byte[] GenerateRandomData(int length)
{
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    return rnd;
}

```

随机整数（均匀分布）

```

public static int GenerateRandomInt(int minValue=0, int maxValue=100)
{
    var rnd = new byte[4];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    var i = Math.Abs(BitConverter.ToInt32(rnd, 0));
    return Convert.ToInt32(i % (maxValue - minValue + 1) + minValue);
}

```

随机字符串

```

public static string GenerateRandomString(int length, string allowableChars=null)
{
    if (string.IsNullOrEmpty(allowableChars))
        allowableChars = @"ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    // 生成随机数据
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
}

```

```

fsIn.Read(iv, 0, iv.Length);
}

// Generate a secure password, based on the password and salt provided
var pdb = new Rfc2898DeriveBytes(password, salt);
var key = pdb.GetBytes(cypher.KeySize / 8);

// Encrypt or decrypt the file
using (var cryptoTransform = encryptMode
    ? cypher.CreateEncryptor(key, iv)
    : cypher.CreateDecryptor(key, iv))
using (var cs = new CryptoStream(fsOut, cryptoTransform, CryptoStreamMode.Write))
{
    fsIn.CopyTo(cs);
}
}
}

```

Section 150.4: Cryptographically Secure Random Data

There are times when the framework's Random() class may not be considered random enough, given that it is based on a psuedo-random number generator. The framework's Crypto classes do, however, provide something more robust in the form of RNGCryptoServiceProvider.

The following code samples demonstrate how to generate Cryptographically Secure byte arrays, strings and numbers.

Random Byte Array

```

public static byte[] GenerateRandomData(int length)
{
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    return rnd;
}

```

Random Integer (with even distribution)

```

public static int GenerateRandomInt(int minValue=0, int maxValue=100)
{
    var rnd = new byte[4];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
    var i = Math.Abs(BitConverter.ToInt32(rnd, 0));
    return Convert.ToInt32(i % (maxValue - minValue + 1) + minValue);
}

```

Random String

```

public static string GenerateRandomString(int length, string allowableChars=null)
{
    if (string.IsNullOrEmpty(allowableChars))
        allowableChars = @"ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    // Generate random data
    var rnd = new byte[length];
    using (var rng = new RNGCryptoServiceProvider())
        rng.GetBytes(rnd);
}

```

```
// 生成输出字符串
var allowable = allowableChars.ToCharArray();
var l = allowable.Length;
var chars = new char[length];
for (var i = 0; i < length; i++)
    chars[i] = allowable[rnd[i] % l];

return new string(chars);
}
```

第150.5节：密码哈希

密码绝不应以明文形式存储！应使用随机生成的盐（以防止彩虹表攻击）和缓慢的密码哈希算法进行哈希。可以使用较高的迭代次数 (> 10k) 来减缓暴力破解攻击。对用户登录来说，约100毫秒的延迟是可以接受的，但这会使破解长密码变得困难。在选择迭代次数时，应使用应用程序可容忍的最大值，并随着计算机性能的提升而增加。还需要考虑阻止可能被用作拒绝服务 (DoS) 攻击的重复请求。

首次哈希时，可以为您生成盐，生成的哈希值和盐随后可以存储到文件中。

```
private void firstHash(string userName, string userPassword, int numberofIterations)
{
    Rfc2898DeriveBytes PBKDF2 = new Rfc2898DeriveBytes(userPassword, 8, numberofIterations);
    //使用8字节盐值对密码进行哈希处理
    byte[] hashedPassword = PBKDF2.GetBytes(20);    //返回一个20字节的哈希值
    byte[] salt = PBKDF2.Salt;
    writeHashToFile(userName, hashedPassword, salt, numberofIterations); //将哈希密码、盐值和迭代次数存储到文件
    中，以便未来密码验证使用
}
```

检查现有用户的密码，从文件中读取其哈希和盐值，并与输入密码的哈希进行比较

```
private bool checkPassword(string userName, string userPassword, int numberofIterations)
{
    byte[] usersHash = getUserHashFromFile(userName);
    byte[] userSalt = getUserSaltFromFile(userName);
    Rfc2898DeriveBytes PBKDF2 = new Rfc2898DeriveBytes(userPassword, userSalt,
    numberofIterations);    //使用用户的盐值对密码进行哈希
    byte[] hashedPassword = PBKDF2.GetBytes(20);    //返回一个20字节的哈希值
    bool passwordsMatch = comparePasswords(usersHash, hashedPassword);    //比较字节数组
    return passwordsMatch;
}
```

第150.6节：快速非对称文件加密

非对称加密通常被认为比对称加密更适合向其他方传输消息。这主要是因为它消除了共享密钥交换相关的许多风险，并确保虽然任何拥有公钥的人都可以预期接收者加密消息，但只有该接收者能够解密。不幸的是，非对称加密算法的主要缺点是它们比对称算法显著更慢。因此，文件的非对称加密，尤其是大文件，通常是一个计算量很大的过程。

为了同时提供安全性和性能，可以采用混合方法。这包括

```
// Generate the output string
var allowable = allowableChars.ToCharArray();
var l = allowable.Length;
var chars = new char[length];
for (var i = 0; i < length; i++)
    chars[i] = allowable[rnd[i] % l];

return new string(chars);
}
```

Section 150.5: Password Hashing

Passwords should never be stored as plain text! They should be hashed with a randomly generated salt (to defend against rainbow table attacks) using a slow password hashing algorithm. A high number of iterations (> 10k) can be used to slow down brute force attacks. A delay of ~100ms is acceptable to a user logging in, but makes breaking a long password difficult. When choosing a number of iterations you should use the maximum tolerable value for your application and increase it as computer performance improves. You will also need to consider stopping repeated requests which could be used as a DoS attack.

When hashing for the first time a salt can be generated for you, the resulting hash and salt can then be stored to a file.

```
private void firstHash(string userName, string userPassword, int numberofIterations)
{
    Rfc2898DeriveBytes PBKDF2 = new Rfc2898DeriveBytes(userPassword, 8, numberofIterations);
    //Hash the password with a 8 byte salt
    byte[] hashedPassword = PBKDF2.GetBytes(20);    //Returns a 20 byte hash
    byte[] salt = PBKDF2.Salt;
    writeHashToFile(userName, hashedPassword, salt, numberofIterations); //Store the hashed
    password with the salt and number of iterations to check against future password entries
}
```

Checking an existing users password, read their hash and salt from a file and compare to the hash of the entered password

```
private bool checkPassword(string userName, string userPassword, int numberofIterations)
{
    byte[] usersHash = getUserHashFromFile(userName);
    byte[] userSalt = getUserSaltFromFile(userName);
    Rfc2898DeriveBytes PBKDF2 = new Rfc2898DeriveBytes(userPassword, userSalt,
    numberofIterations);    //Hash the password with the users salt
    byte[] hashedPassword = PBKDF2.GetBytes(20);    //Returns a 20 byte hash
    bool passwordsMatch = comparePasswords(usersHash, hashedPassword);    //Compares byte arrays
    return passwordsMatch;
}
```

Section 150.6: Fast Asymmetric File Encryption

Asymmetric encryption is often regarded as preferable to Symmetric encryption for transferring messages to other parties. This is mainly because it negates many of the risks related to the exchange of a shared key and ensures that whilst anyone with the public key can encrypt a message for the intended recipient, only that recipient can decrypt it. Unfortunately the major down-side of asymmetric encryption algorithms is that they are significantly slower than their symmetric cousins. As such the asymmetric encryption of files, especially large ones, can often be a very computationally intensive process.

In order to provide both security AND performance, a hybrid approach can be taken. This entails the

对称加密的密钥和初始化向量的密码学随机生成。这些值随后使用非对称算法加密并写入输出文件，然后用于对源数据进行对称加密并附加到输出中。

这种方法在性能和安全性方面都提供了很高的保障，因为数据使用对称算法加密（速度快），而密钥和初始化向量均为随机生成（安全），并通过非对称算法加密（安全）。它还有一个额外的优点，即同一负载在不同时间加密时，密文会有很大差异，因为对称密钥每次都是随机生成的。

下面的类演示了字符串和字节数组的非对称加密，以及混合文件加密。

```
public static class 非对称提供者
{
    #region 密钥生成
    public class 密钥对
    {
        public string 公钥 { get; set; }
        public string 私钥 { get; set; }
    }

    public static 密钥对 生成新密钥对(int 密钥大小 = 4096)
    {
        // 密钥大小以位为单位。1024是默认值，2048更好，4096更强大但生成时间较长。
        using (var rsa = new RSACryptoServiceProvider(密钥大小))
        {
            return new 密钥对 {公钥 = rsa.ToXmlString(false), 私钥 =
rsa.ToXmlString(true)};
        }
    }

    #endregion

    #region 非对称数据加密与解密

    public static byte[] EncryptData(byte[] data, string publicKey)
    {
        using (var asymmetricProvider = new RSACryptoServiceProvider())
        {
            asymmetricProvider.FromXmlString(publicKey);
            return asymmetricProvider.Encrypt(data, true);
        }
    }

    public static byte[] DecryptData(byte[] data, string publicKey)
    {
        using (var asymmetricProvider = new RSACryptoServiceProvider())
        {
            asymmetricProvider.FromXmlString(publicKey);
            if (asymmetricProvider.PublicOnly)
                throw new Exception("The key provided is a public key and does not contain the
private key elements required for decryption");
            return asymmetricProvider.Decrypt(data, true);
        }
    }

    public static string EncryptString(string value, string publicKey)
    {
        return Convert.ToString(EncryptData(Encoding.UTF8.GetBytes(value), publicKey));
    }
}
```

cryptographically random generation of a key and initialization vector for *Symmetric* encryption. These values are then encrypted using an *Asymmetric* algorithm and written to the output file, before being used to encrypt the source data *Symmetrically* and appending it to the output.

This approach provides a high degree of both performance and security, in that the data is encrypted using a symmetric algorithm (fast) and the key and iv, both randomly generated (secure) are encrypted by an asymmetric algorithm (secure). It also has the added advantage that the same payload encrypted on different occasions will have very different ciphertext, because the symmetric keys are randomly generated each time.

The following class demonstrates asymmetric encryption of strings and byte arrays, as well as hybrid file encryption.

```
public static class AsymmetricProvider
{
    #region Key Generation
    public class KeyPair
    {
        public string PublicKey { get; set; }
        public string PrivateKey { get; set; }
    }

    public static KeyPair GenerateNewKeyPair(int keySize = 4096)
    {
        // KeySize is measured in bits. 1024 is the default, 2048 is better, 4096 is more robust but
        takes a fair bit longer to generate.
        using (var rsa = new RSACryptoServiceProvider(keySize))
        {
            return new KeyPair {PublicKey = rsa.ToXmlString(false), PrivateKey =
rsa.ToXmlString(true)};
        }
    }

    #endregion

    #region Asymmetric Data Encryption and Decryption

    public static byte[] EncryptData(byte[] data, string publicKey)
    {
        using (var asymmetricProvider = new RSACryptoServiceProvider())
        {
            asymmetricProvider.FromXmlString(publicKey);
            return asymmetricProvider.Encrypt(data, true);
        }
    }

    public static byte[] DecryptData(byte[] data, string publicKey)
    {
        using (var asymmetricProvider = new RSACryptoServiceProvider())
        {
            asymmetricProvider.FromXmlString(publicKey);
            if (asymmetricProvider.PublicOnly)
                throw new Exception("The key provided is a public key and does not contain the
private key elements required for decryption");
            return asymmetricProvider.Decrypt(data, true);
        }
    }

    public static string EncryptString(string value, string publicKey)
    {
        return Convert.ToString(EncryptData(Encoding.UTF8.GetBytes(value), publicKey));
    }
}
```

```

}

public static string DecryptString(string value, string privateKey)
{
    return Encoding.UTF8.GetString(EncryptData(Convert.FromBase64String(value), privateKey));
}

#endregion

#region 混合文件加密和解密

public static void EncryptFile(string inputFilePath, string outputPath, string publicKey)
{
    using (var symmetricCypher = new AesManaged())
    {
        // 生成用于对称加密的随机密钥和初始化向量 (IV)
        var key = new byte[symmetricCypher.KeySize / 8];
        var iv = new byte[symmetricCypher.BlockSize / 8];
        using (var rng = new RNGCryptoServiceProvider())
        {
            rng.GetBytes(key);
            rng.GetBytes(iv);
        }

        // 加密对称密钥和初始化向量 (IV)
        var buf = new byte[key.Length + iv.Length];
        Array.Copy(key, buf, key.Length);
        Array.Copy(iv, 0, buf, key.Length, iv.Length);
        buf = EncryptData(buf, publicKey);

        var bufLen = BitConverter.GetBytes(buf.Length);

        // 对数据进行对称加密并写入文件，同时写入加密后的密钥
        using (var cypherKey = symmetricCypher.CreateEncryptor(key, iv))
        using (var fsIn = new FileStream(inputFilePath, FileMode.Open))
        using (var fsOut = new FileStream(outputFilePath, FileMode.Create))
        using (var cs = new CryptoStream(fsOut, cypherKey, CryptoStreamMode.Write))
        {
            fsOut.Write(bufLen, 0, bufLen.Length);
            fsOut.Write(buf, 0, buf.Length);
            fsIn.CopyTo(cs);
        }
    }
}

public static void 解密文件(string 输入文件路径, string 输出文件路径, string 私钥)
{
    using (var 对称加密器 = new AesManaged())
    using (var fsIn = new FileStream(输入文件路径, FileMode.Open))
    {
        // 确定加密密钥和初始化向量的长度
        var buf = new byte[sizeof(int)];
        fsIn.Read(buf, 0, buf.Length);
        var bufLen = BitConverter.ToInt32(buf, 0);

        // 从文件中读取加密的密钥和初始化向量数据，并使用非对称算法解密
        buf = new byte[bufLen];
        fsIn.Read(buf, 0, buf.Length);
        buf = 解密数据(buf, 私钥);
    }
}

```

```

}

public static string DecryptString(string value, string privateKey)
{
    return Encoding.UTF8.GetString(EncryptData(Convert.FromBase64String(value), privateKey));
}

#endregion

#region Hybrid File Encryption and Decryption

public static void EncryptFile(string inputFilePath, string outputPath, string publicKey)
{
    using (var symmetricCypher = new AesManaged())
    {
        // Generate random key and IV for symmetric encryption
        var key = new byte[symmetricCypher.KeySize / 8];
        var iv = new byte[symmetricCypher.BlockSize / 8];
        using (var rng = new RNGCryptoServiceProvider())
        {
            rng.GetBytes(key);
            rng.GetBytes(iv);
        }

        // Encrypt the symmetric key and IV
        var buf = new byte[key.Length + iv.Length];
        Array.Copy(key, buf, key.Length);
        Array.Copy(iv, 0, buf, key.Length, iv.Length);
        buf = EncryptData(buf, publicKey);

        var bufLen = BitConverter.GetBytes(buf.Length);

        // Symmetrically encrypt the data and write it to the file, along with the encrypted key
        and iv
        using (var cypherKey = symmetricCypher.CreateEncryptor(key, iv))
        using (var fsIn = new FileStream(inputFilePath, FileMode.Open))
        using (var fsOut = new FileStream(outputFilePath, FileMode.Create))
        using (var cs = new CryptoStream(fsOut, cypherKey, CryptoStreamMode.Write))
        {
            fsOut.Write(bufLen, 0, bufLen.Length);
            fsOut.Write(buf, 0, buf.Length);
            fsIn.CopyTo(cs);
        }
    }
}

public static void DecryptFile(string inputFilePath, string outputPath, string privateKey)
{
    using (var symmetricCypher = new AesManaged())
    using (var fsIn = new FileStream(inputFilePath, FileMode.Open))
    {
        // Determine the length of the encrypted key and IV
        var buf = new byte[sizeof(int)];
        fsIn.Read(buf, 0, buf.Length);
        var bufLen = BitConverter.ToInt32(buf, 0);

        // Read the encrypted key and IV data from the file and decrypt using the asymmetric
        algorithm
        buf = new byte[bufLen];
        fsIn.Read(buf, 0, buf.Length);
        buf = DecryptData(buf, privateKey);
    }
}

```

```

var key = new byte[对称加密器.KeySize / 8];
var iv = new byte[对称加密器.BlockSize / 8];
Array.Copy(buf, key, key.Length);
Array.Copy(buf, key.Length, iv, 0, iv.Length);

// 使用对称算法解密文件数据
using (var cypherKey = symmetricCypher.CreateDecryptor(key, iv))
using (var fsOut = new FileStream(outputFilePath, FileMode.Create))
using (var cs = new CryptoStream(fsOut, cypherKey, CryptoStreamMode.Write))
{
    fsIn.CopyTo(cs);
}
}

#endregion

#region 密钥存储

public static void 写入公钥(string publicKeyFilePath, string publicKey)
{
File.WriteAllText(publicKeyFilePath, publicKey);
}
public static string 读取公钥(string publicKeyFilePath)
{
    return File.ReadAllText(publicKeyFilePath);
}

private const string 对称盐值 = "Stack_Overflow!"; // 请修改！

public static string 读取私钥(string privateKeyFilePath, string password)
{
    var salt = Encoding.UTF8.GetBytes(对称盐值);
    var cypherText = File.ReadAllBytes(privateKeyFilePath);

    using (var cypher = new AesManaged())
    {
        var pdb = new Rfc2898DeriveBytes(password, salt);
        var key = pdb.GetBytes(cypher.KeySize / 8);
        var iv = pdb.GetBytes(cypher.BlockSize / 8);

        using (var decryptor = cypher.CreateDecryptor(key, iv))
        using (var msDecrypt = new MemoryStream(cypherText))
        using (var csDecrypt = new CryptoStream(msDecrypt, decryptor, CryptoStreamMode.Read))
        using (var srDecrypt = new StreamReader(csDecrypt))
        {
            return srDecrypt.ReadToEnd();
        }
    }
}

public static void WritePrivateKey(string privateKeyFilePath, string privateKey, string password)
{
    var salt = Encoding.UTF8.GetBytes(SymmetricSalt);
    using (var cypher = new AesManaged())
    {
        var pdb = new Rfc2898DeriveBytes(password, salt);
        var key = pdb.GetBytes(cypher.KeySize / 8);
        var iv = pdb.GetBytes(cypher.BlockSize / 8);

        using (var encryptor = cypher.CreateEncryptor(key, iv))

```

```

var key = new byte[symmetricCypher.KeySize / 8];
var iv = new byte[symmetricCypher.BlockSize / 8];
Array.Copy(buf, key, key.Length);
Array.Copy(buf, key.Length, iv, 0, iv.Length);

// Decrypt the file data using the symmetric algorithm
using (var cypherKey = symmetricCypher.CreateDecryptor(key, iv))
using (var fsOut = new FileStream(outputFilePath, FileMode.Create))
using (var cs = new CryptoStream(fsOut, cypherKey, CryptoStreamMode.Write))
{
    fsIn.CopyTo(cs);
}
}

#endregion

#region Key Storage

public static void WritePublicKey(string publicKeyFilePath, string publicKey)
{
    File.WriteAllText(publicKeyFilePath, publicKey);
}
public static string ReadPublicKey(string publicKeyFilePath)
{
    return File.ReadAllText(publicKeyFilePath);
}

private const string SymmetricSalt = "Stack_Overflow!"; // Change me!

public static string ReadPrivateKey(string privateKeyFilePath, string password)
{
    var salt = Encoding.UTF8.GetBytes(SymmetricSalt);
    var cypherText = File.ReadAllBytes(privateKeyFilePath);

    using (var cypher = new AesManaged())
    {
        var pdb = new Rfc2898DeriveBytes(password, salt);
        var key = pdb.GetBytes(cypher.KeySize / 8);
        var iv = pdb.GetBytes(cypher.BlockSize / 8);

        using (var decryptor = cypher.CreateDecryptor(key, iv))
        using (var msDecrypt = new MemoryStream(cypherText))
        using (var csDecrypt = new CryptoStream(msDecrypt, decryptor, CryptoStreamMode.Read))
        using (var srDecrypt = new StreamReader(csDecrypt))
        {
            return srDecrypt.ReadToEnd();
        }
    }
}

public static void WritePrivateKey(string privateKeyFilePath, string privateKey, string password)
{
    var salt = Encoding.UTF8.GetBytes(SymmetricSalt);
    using (var cypher = new AesManaged())
    {
        var pdb = new Rfc2898DeriveBytes(password, salt);
        var key = pdb.GetBytes(cypher.KeySize / 8);
        var iv = pdb.GetBytes(cypher.BlockSize / 8);

        using (var encryptor = cypher.CreateEncryptor(key, iv))

```

```

        using (var fsEncrypt = new FileStream(privateKeyFilePath, FileMode.Create))
        using (var csEncrypt = new CryptoStream(fsEncrypt, encryptor, CryptoStreamMode.Write))
        using (var swEncrypt = new StreamWriter(csEncrypt))
        {
            swEncrypt.Write(privateKey);
        }
    }

#endregion
}

```

使用示例：

```

private static void HybridCryptoTest(string privateKeyPath, string privateKeyPassword, string
inputPath)
{
    // 设置测试
    var publicKeyPath = Path.ChangeExtension(privateKeyPath, ".public");
    var outputPath = Path.Combine(Path.ChangeExtension(inputPath, ".enc"));
    var testPath = Path.Combine(Path.ChangeExtension(inputPath, ".test"));

    if (!File.Exists(privateKeyPath))
    {
        var keys = AsymmetricProvider.GenerateNewKeyPair(2048);
        AsymmetricProvider.WritePublicKey(publicKeyPath, keys.PublicKey);
        AsymmetricProvider.WritePrivateKey(privateKeyPath, keys.PrivateKey, privateKeyPassword);
    }

    // 加密文件
    var publicKey = AsymmetricProvider.ReadPublicKey(publicKeyPath);
    AsymmetricProvider.EncryptFile(inputPath, outputPath, publicKey);

    // 再次解密以与源文件进行比较
    var privateKey = AsymmetricProvider.ReadPrivateKey(privateKeyPath, privateKeyPassword);
    AsymmetricProvider.DecryptFile(outputPath, testPath, privateKey);

    // 检查两个文件是否匹配
    var source = File.ReadAllBytes(inputPath);
    var dest = File.ReadAllBytes(testPath);

    if (source.Length != dest.Length)
        throw new Exception("长度不匹配");

    if (source.Where((t, i) => t != dest[i]).Any())
        throw new Exception("数据不匹配");
}

```

```

        using (var fsEncrypt = new FileStream(privateKeyFilePath, FileMode.Create))
        using (var csEncrypt = new CryptoStream(fsEncrypt, encryptor, CryptoStreamMode.Write))
        using (var swEncrypt = new StreamWriter(csEncrypt))
        {
            swEncrypt.Write(privateKey);
        }
    }

#endregion
}

```

Example of use:

```

private static void HybridCryptoTest(string privateKeyPath, string privateKeyPassword, string
inputPath)
{
    // Setup the test
    var publicKeyPath = Path.ChangeExtension(privateKeyPath, ".public");
    var outputPath = Path.Combine(Path.ChangeExtension(inputPath, ".enc"));
    var testPath = Path.Combine(Path.ChangeExtension(inputPath, ".test"));

    if (!File.Exists(privateKeyPath))
    {
        var keys = AsymmetricProvider.GenerateNewKeyPair(2048);
        AsymmetricProvider.WritePublicKey(publicKeyPath, keys.PublicKey);
        AsymmetricProvider.WritePrivateKey(privateKeyPath, keys.PrivateKey, privateKeyPassword);
    }

    // Encrypt the file
    var publicKey = AsymmetricProvider.ReadPublicKey(publicKeyPath);
    AsymmetricProvider.EncryptFile(inputPath, outputPath, publicKey);

    // Decrypt it again to compare against the source file
    var privateKey = AsymmetricProvider.ReadPrivateKey(privateKeyPath, privateKeyPassword);
    AsymmetricProvider.DecryptFile(outputPath, testPath, privateKey);

    // Check that the two files match
    var source = File.ReadAllBytes(inputPath);
    var dest = File.ReadAllBytes(testPath);

    if (source.Length != dest.Length)
        throw new Exception("Length does not match");

    if (source.Where((t, i) => t != dest[i]).Any())
        throw new Exception("Data mismatch");
}

```

第151章：ASP.NET 身份认证

关于asp.net身份认证的教程，如用户管理、角色管理、创建令牌等。

第151.1节：如何使用用户管理器在asp.net身份认证中实现密码重置令牌

1. 创建一个名为MyClasses的新文件夹，并创建并添加以下类

```
public class GmailEmailService:SmtpClient
{
    // Gmail 用户名
    public string UserName { get; set; }

    public GmailEmailService() :
        base(ConfigurationManager.AppSettings["GmailHost"],
        Int32.Parse(ConfigurationManager.AppSettings["GmailPort"]))
    {
        // 从web.config文件获取值：
        this.UserName = ConfigurationManager.AppSettings["GmailUserName"];
        this.EnableSsl = Boolean.Parse(ConfigurationManager.AppSettings["GmailSsl"]);
        this.UseDefaultCredentials = false;
        this.Credentials = new System.Net.NetworkCredential(this.UserName,
        ConfigurationManager.AppSettings["GmailPassword"]);
    }
}
```

2. 配置您的身份类

```
public async Task SendAsync(IdentityMessage message)
{
    MailMessage email = new MailMessage(new MailAddress("youremailadress@domain.com", "(any
    subject here"),
    new MailAddress(message.Destination));
    email.Subject = message.Subject;
    email.Body = message.Body;

    email.IsBodyHtml = true;

    GmailEmailService mailClient = new GmailEmailService();
    await mailClient.SendMailAsync(email);
}
```

3. 将您的凭据添加到 web.config。我在这部分没有使用 Gmail，因为我的工作场所屏蔽了 Gmail，但它仍然能完美工作。

```
<add key="GmailUserName" value="youremail@yourdomain.com"/> <add key="GmailPassword"
value="yourPassword"/> <add key="GmailHost" value="yourServer"/> <add key="GmailPort"
value="yourPort"/> <add key="GmailSsl" value="chooseTrueOrFalse"/> <!--Smtp Server (confirmations
emails)-->
```

4. 对您的账户控制器进行必要的更改。添加以下高亮代码。

Chapter 151: ASP.NET Identity

Tutorials concerning asp.net Identity such as user management, role management, creating tokens and more.

Section 151.1: How to implement password reset token in asp.net identity using user manager

1. Create a new folder called MyClasses and create and add the following class

```
public class GmailEmailService:SmtpClient
{
    // Gmail user-name
    public string UserName { get; set; }

    public GmailEmailService() :
        base(ConfigurationManager.AppSettings["GmailHost"],
        Int32.Parse(ConfigurationManager.AppSettings["GmailPort"]))
    {
        //Get values from web.config file:
        this.UserName = ConfigurationManager.AppSettings["GmailUserName"];
        this.EnableSsl = Boolean.Parse(ConfigurationManager.AppSettings["GmailSsl"]);
        this.UseDefaultCredentials = false;
        this.Credentials = new System.Net.NetworkCredential(this.UserName,
        ConfigurationManager.AppSettings["GmailPassword"]);
    }
}
```

2. Configure your Identity Class

```
public async Task SendAsync(IdentityMessage message)
{
    MailMessage email = new MailMessage(new MailAddress("youremailadress@domain.com", "(any
    subject here"),
    new MailAddress(message.Destination));
    email.Subject = message.Subject;
    email.Body = message.Body;

    email.IsBodyHtml = true;

    GmailEmailService mailClient = new GmailEmailService();
    await mailClient.SendMailAsync(email);
}
```

3. Add your credentials to the web.config. I did not use gmail in this portion because the use of gmail is blocked in my workplace and it still works perfectly.

```
<add key="GmailUserName" value="youremail@yourdomain.com"/> <add key="GmailPassword"
value="yourPassword"/> <add key="GmailHost" value="yourServer"/> <add key="GmailPort"
value="yourPort"/> <add key="GmailSsl" value="chooseTrueOrFalse"/> <!--Smtp Server (confirmations
emails)-->
```

4. Make necessary changes to your Account Controller. Add the following highlighted code.

```

using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.DataProtection;
using System;
using System.Linq;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;
using MYPROJECT.Models;

namespace MYPROJECT.Controllers
{
    [Authorize]
    public class AccountController : Controller
    {
        private ApplicationSignInManager _signInManager;
        private ApplicationUserManager _userManager;
        ApplicationDbContext _context;
        DpapiDataProtectionProvider provider = new DpapiDataProtectionProvider("MYPROJECT");
        EmailService EmailService = new EmailService();

        public AccountController()
            : this(new UserManager<ApplicationUser>(new UserStore<ApplicationUser>(new ApplicationDbContext())))
        {
            _context = new ApplicationDbContext();
        }

        public AccountController(UserManager< ApplicationUser > userManager, ApplicationSignInManager signInManager)
        {
            UserManager = userManager;
            SignInManager = signInManager;
            UserManager.UserTokenProvider = new DataProtectorTokenProvider< ApplicationUser >(
                provider.Create("EmailConfirmation"));
        }

        private AccountController(UserManager< ApplicationUser > userManager)
        {
            UserManager = userManager;
            UserManager.UserTokenProvider = new DataProtectorTokenProvider< ApplicationUser >(
                provider.Create("EmailConfirmation"));
        }

        public UserManager< ApplicationUser > UserManager { get; private set; }

        public ApplicationSignInManager SignInManager
        {
            get
            {
                return _signInManager ?? HttpContext.GetOwinContext().Get< ApplicationSignInManager >();
            }
            private set
            {
                _signInManager = value;
            }
        }
    }
}

```

```

using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin.Security;
using Microsoft.Owin.Security.DataProtection;
using System;
using System.Linq;
using System.Threading.Tasks;
using System.Web;
using System.Web.Mvc;
using MYPROJECT.Models;

namespace MYPROJECT.Controllers
{
    [Authorize]
    public class AccountController : Controller
    {
        private ApplicationSignInManager _signInManager;
        private ApplicationUserManager _userManager;
        ApplicationDbContext _context;
        DpapiDataProtectionProvider provider = new DpapiDataProtectionProvider("MYPROJECT");
        EmailService EmailService = new EmailService();

        public AccountController()
            : this(new UserManager< ApplicationUser >(new UserStore< ApplicationUser >(new ApplicationDbContext())))
        {
            _context = new ApplicationDbContext();
        }

        public AccountController(UserManager< ApplicationUser > userManager, ApplicationSignInManager signInManager)
        {
            UserManager = userManager;
            SignInManager = signInManager;
            UserManager.UserTokenProvider = new DataProtectorTokenProvider< ApplicationUser >(
                provider.Create("EmailConfirmation"));
        }

        private AccountController(UserManager< ApplicationUser > userManager)
        {
            UserManager = userManager;
            UserManager.UserTokenProvider = new DataProtectorTokenProvider< ApplicationUser >(
                provider.Create("EmailConfirmation"));
        }

        public UserManager< ApplicationUser > UserManager { get; private set; }

        public ApplicationSignInManager SignInManager
        {
            get
            {
                return _signInManager ?? HttpContext.GetOwinContext().Get< ApplicationSignInManager >();
            }
            private set
            {
                _signInManager = value;
            }
        }
    }
}

```

```

// GET: /Account/ForgotPassword
[AllowAnonymous]
public ActionResult ForgotPassword()
{
    return View();
}

// POST: /Account/ForgotPassword
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> ForgotPassword(ForgotPasswordViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = await UserManager.FindByNameAsync(model.UserName);

        if (user == null || !(await UserManager.IsEmailConfirmedAsync(user.Id)))
        {
            // Don't reveal that the user does not exist or is not confirmed
            return View("ForgotPasswordConfirmation");
        }

        // For more information on how to enable account confirmation and password reset please visit http://go.microsoft.com/fwlink/?LinkId=320771
        // Send an email with this link
        string code = await UserManager.GeneratePasswordResetTokenAsync(user.Id);
        code = HttpUtility.UrlEncode(code);
        var callbackUrl = Url.Action("ResetPassword", "Account", new { userId = user.Id, code = HttpUtility.UrlDecode(code) }, protocol: Request.Url.Scheme);
        await EmailService.SendAsync(new IdentityMessage
        {
            Body = "Please reset your password by clicking <a href=\"" + callbackUrl + "\">here</a>",
            Destination = user.Email,
            Subject = "Reset Password"
        });
        //await UserManager.SendEmailAsync(user.Id, "Reset Password", "Please reset your password by clicking <a href=\"" + callbackUrl + "\">here</a>");
        return RedirectToAction("ForgotPasswordConfirmation", "Account");
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}

```

编译然后运行。干杯！

belindoc.com

```

// GET: /Account/ForgotPassword
[AllowAnonymous]
public ActionResult ForgotPassword()
{
    return View();
}

// POST: /Account/ForgotPassword
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public async Task<ActionResult> ForgotPassword(ForgotPasswordViewModel model)
{
    if (ModelState.IsValid)
    {
        var user = await UserManager.FindByNameAsync(model.UserName);

        if (user == null || !(await UserManager.IsEmailConfirmedAsync(user.Id)))
        {
            // Don't reveal that the user does not exist or is not confirmed
            return View("ForgotPasswordConfirmation");
        }

        // For more information on how to enable account confirmation and password reset please visit http://go.microsoft.com/fwlink/?LinkId=320771
        // Send an email with this link
        string code = await UserManager.GeneratePasswordResetTokenAsync(user.Id);
        code = HttpUtility.UrlEncode(code);
        var callbackUrl = Url.Action("ResetPassword", "Account", new { userId = user.Id, code = HttpUtility.UrlDecode(code) }, protocol: Request.Url.Scheme);
        await EmailService.SendAsync(new IdentityMessage
        {
            Body = "Please reset your password by clicking <a href=\"" + callbackUrl + "\">here</a>",
            Destination = user.Email,
            Subject = "Reset Password"
        });
        //await UserManager.SendEmailAsync(user.Id, "Reset Password", "Please reset your password by clicking <a href=\"" + callbackUrl + "\">here</a>");
        return RedirectToAction("ForgotPasswordConfirmation", "Account");
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}

```

Compile then run. Cheers!

第152章：.NET中的不安全代码

第152.1节：使用不安全代码操作数组

使用指针访问数组时，没有边界检查，因此不会抛出`IndexOutOfRangeException`异常。这使代码运行更快。

使用指针给数组赋值：

```
class 程序
{
    static void Main(string[] args)
    {
        unsafe
        {
            int[] array = new int[1000];
            fixed (int* ptr = array)
            {
                for (int i = 0; i < array.Length; i++)
                {
                    *(ptr+i) = i; //用指针赋值
                }
            }
        }
    }
}
```

安全且正常的对应代码如下：

```
class 程序
{
    static void Main(string[] args)
    {
        int[] array = new int[1000];

        for (int i = 0; i < array.Length; i++)
        {
            array[i] = i;
        }
    }
}
```

不安全的部分通常会更快，性能差异取决于数组中元素的复杂性以及对每个元素应用的逻辑。尽管可能更快，但应谨慎使用，因为它更难维护且更容易出错。

第152.2节：在字符串中使用unsafe

```
var s = "Hello";      // 变量's'引用的字符串通常是不可变的，但
                      // 由于它是内存，如果能以不安全的方式访问它，
                      // 我们可以修改它。

unsafe           // 允许写入内存；System.String上的方法不允许这样做
{
    fixed (char* c = s) // 获取指向原本存储在只读内存中的字符串的指针
    for (int i = 0; i < s.Length; i++)
        c[i] = 'a';    // 修改为原字符串"Hello"分配的内存中的数据
}
```

Chapter 152: Unsafe Code in .NET

Section 152.1: Using unsafe with arrays

When accessing arrays with pointers, there are no bounds check and therefore no `IndexOutOfRangeException` will be thrown. This makes the code faster.

Assigning values to an array with a pointer:

```
class Program
{
    static void Main(string[] args)
    {
        unsafe
        {
            int[] array = new int[1000];
            fixed (int* ptr = array)
            {
                for (int i = 0; i < array.Length; i++)
                {
                    *(ptr+i) = i; //assigning the value with the pointer
                }
            }
        }
    }
}
```

While the safe and normal counterpart would be:

```
class Program
{
    static void Main(string[] args)
    {
        int[] array = new int[1000];

        for (int i = 0; i < array.Length; i++)
        {
            array[i] = i;
        }
    }
}
```

The unsafe part will generally be faster and the difference in performance can vary depending on the complexity of the elements in the array as well as the logic applied to each one. Even though it may be faster, it should be used with care since it is harder to maintain and easier to break.

Section 152.2: Using unsafe with strings

```
var s = "Hello";      // The string referenced by variable 's' is normally immutable, but
                      // since it is memory, we could change it if we can access it in an
                      // unsafe way.

unsafe           // allows writing to memory; methods on System.String don't allow this
{
    fixed (char* c = s) // get pointer to string originally stored in read only memory
    for (int i = 0; i < s.Length; i++)
        c[i] = 'a';    // change data in memory allocated for original string "Hello"
}
```

```
}
```

```
Console.WriteLine(s); // 变量 's' 仍然指向内存中相同的 System.String// 值，但该位置的内容已被上面  
                     // 的不安全写入更改。
```

```
// 显示: "aaaaa"
```

第152.3节：不安全的数组索引

```
void Main()  
{  
    unsafe  
    {  
        int[] a = {1, 2, 3};  
        fixed(int* b = a)  
        {  
Console.WriteLine(b[4]);  
        }  
    }  
}
```

运行此代码会创建一个长度为3的数组，但随后尝试获取第5个元素（索引4）。在我的机器上，这打印了1910457872，但该行为未定义。

没有unsafe代码块，您无法使用指针，因此无法访问数组末尾之后的值，否则会导致抛出异常。

```
}
```

```
Console.WriteLine(s); // The variable 's' still refers to the same System.String  
                     // value in memory, but the contents at that location were  
                     // changed by the unsafe write above.  
                     // Displays: "aaaaa"
```

Section 152.3: Unsafe Array Index

```
void Main()  
{  
    unsafe  
    {  
        int[] a = {1, 2, 3};  
        fixed(int* b = a)  
        {  
Console.WriteLine(b[4]);  
        }  
    }  
}
```

Running this code creates an array of length 3, but then tries to get the 5th item (index 4). On my machine, this printed 1910457872, but the behavior is not defined.

Without the `unsafe` block, you cannot use pointers, and therefore cannot access values past the end of an array without causing an exception to be thrown.

第153章：C# 脚本

第153.1节：简单代码评估

您可以评估任何有效的 C# 代码：

```
int value = await CSharpScript.EvaluateAsync<int>("15 * 89 + 95");
var span = await CSharpScript.EvaluateAsync<TimeSpan>("new DateTime(2016,1,1) - DateTime.Now");
```

如果未指定类型，结果为object：

```
object value = await CSharpScript.EvaluateAsync("15 * 89 + 95");
```

Chapter 153: C# Script

Section 153.1: Simple code evaluation

You can evaluate any valid C# code:

```
int value = await CSharpScript.EvaluateAsync<int>("15 * 89 + 95");
var span = await CSharpScript.EvaluateAsync<TimeSpan>("new DateTime(2016,1,1) - DateTime.Now");
```

If type is not specified, the result is `object`:

```
object value = await CSharpScript.EvaluateAsync("15 * 89 + 95");
```

belindoc.com

第154章：运行时编译

第154.1节：RoslynScript

Microsoft.CodeAnalysis.CSharp.Scripting.CSharpScript 是一个新的 C# 脚本引擎。

```
var code = "(1 + 2).ToString()";
var run = await CSharpScript.RunAsync(code, ScriptOptions.Default);
var result = (string)run.ReturnValue;
Console.WriteLine(result); //输出 3
```

您可以编译并运行任何语句、变量、方法、类或任何代码段。

第154.2节：CSharpCodeProvider

Microsoft.CSharp.CSharpCodeProvider 可用于编译C#类。

```
var code = @"
    public class Abc {
public string Get() { return ""abc""; }
    }

var options = new CompilerParameters();
options.GenerateExecutable = false;
options.GenerateInMemory = false;

var provider = new CSharpCodeProvider();
var compile = provider.CompileAssemblyFromSource(options, code);

var type = compile.CompiledAssembly.GetType("Abc");
var abc = Activator.CreateInstance(type);

var method = type.GetMethod("Get");
var result = method.Invoke(abc, null);

Console.WriteLine(result); //输出：abc
```

Chapter 154: Runtime Compile

Section 154.1: RoslynScript

Microsoft.CodeAnalysis.CSharp.Scripting.CSharpScript is a new C# script engine.

```
var code = "(1 + 2).ToString()";
var run = await CSharpScript.RunAsync(code, ScriptOptions.Default);
var result = (string)run.ReturnValue;
Console.WriteLine(result); //output 3
```

You can compile and run any statements, variables, methods, classes or any code segments.

Section 154.2: CSharpCodeProvider

Microsoft.CSharp.CSharpCodeProvider can be used to compile C# classes.

```
var code = @"
    public class Abc {
        public string Get() { return ""abc""; }
    }

var options = new CompilerParameters();
options.GenerateExecutable = false;
options.GenerateInMemory = false;

var provider = new CSharpCodeProvider();
var compile = provider.CompileAssemblyFromSource(options, code);

var type = compile.CompiledAssembly.GetType("Abc");
var abc = Activator.CreateInstance(type);

var method = type.GetMethod("Get");
var result = method.Invoke(abc, null);

Console.WriteLine(result); //output: abc
```

第155章：互操作性

第155.1节：从非托管C++ DLL导入函数

以下是如何导入定义在非托管C++ DLL中的函数的示例。在“myDLL.dll”的C++源代码中，定义了函数add：

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
{
    return a + b;
}
```

然后可以按如下方式将其包含到C#程序中：

```
class 程序
{
    // 这行代码将导入C++方法。
    // DllImport属性中指定的名称必须是DLL名称。
    // 参数名称无关紧要，但类型必须正确。
    [DllImport("myDLL.dll")]
    private static extern int add(int left, int right);

    static void Main(string[] args)
    {
        // extern 方法可以像其他任何 C# 方法一样调用。
        Console.WriteLine(add(1, 2));
    }
}
```

有关为什么需要 `extern "C"` 和 `__stdcall` 的解释，请参见调用约定和 C++ 名称修饰。

查找动态库

当 `extern` 方法首次被调用时，C# 程序将搜索并加载相应的 DLL。有关搜索 DLL 的位置以及如何影响搜索位置的更多信息，请参见 [thisstackoverflow question](#)。

第 155.2 节：调用约定

调用函数有多种约定，指定谁（调用者或被调用者）从栈中弹出参数，参数如何传递以及传递顺序。C++ 默认使用 `Cdecl` 调用约定，但 C# 期望 `StdCall`，通常用于 Windows API。你需要更改其中之一：

- 在 C++ 中将调用约定更改为 `StdCall`：

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
[DllImport("myDLL.dll")]
```

- 或者，在 C# 中将调用约定更改为 `Cdecl`：

```
extern "C" __declspec(dllexport) int /*__cdecl*/ add(int a, int b)
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl)]
```

Chapter 155: Interoperability

Section 155.1: Import function from unmanaged C++ DLL

Here is an example of how to import a function that is defined in an unmanaged C++ DLL. In the C++ source code for "myDLL.dll", the function `add` is defined:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
{
    return a + b;
}
```

Then it can be included into a C# program as follows:

```
class Program
{
    // This line will import the C++ method.
    // The name specified in the DllImport attribute must be the DLL name.
    // The names of parameters are unimportant, but the types must be correct.
    [DllImport("myDLL.dll")]
    private static extern int add(int left, int right);

    static void Main(string[] args)
    {
        //The extern method can be called just as any other C# method.
        Console.WriteLine(add(1, 2));
    }
}
```

See Calling conventions and C++ name mangling for explanations about why `extern "C"` and `__stdcall` are necessary.

Finding the dynamic library

When the `extern` method is first invoked the C# program will search for and load the appropriate DLL. For more information about where is searched to find the DLL, and how you can influence the search locations see [this stackoverflow question](#).

Section 155.2: Calling conventions

There're several conventions of calling functions, specifying who (caller or callee) pops arguments from the stack, how arguments are passed and in what order. C++ uses `Cdecl` calling convention by default, but C# expects `StdCall`, which is usually used by Windows API. You need to change one or the other:

- Change calling convention to `StdCall` in C++:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
[DllImport("myDLL.dll")]
```

- Or, change calling convention to `Cdecl` in C#:

```
extern "C" __declspec(dllexport) int /*__cdecl*/ add(int a, int b)
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl)]
```

如果你想使用带有 Cdecl 调用约定且名称被修饰的函数，代码将如下所示：

```
__declspec(dllexport) int add(int a, int b)
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl,
EntryPoint = "?add@@YAHHH@Z")]
```

- **thiscall(_thiscall)** 主要用于类的成员函数。
- 当函数使用 **thiscall(_thiscall)** 时，会将指向类的指针作为第一个参数传递下去。

第155.3节：C++ 名称修饰

C++ 编译器在导出函数的名称中编码了额外的信息，如参数类型，以支持不同参数的重载。这个过程称为名称修饰。这会导致在 C# 中导入函数（以及与其他语言的互操作）时出现问题，因为函数 `int add(int a, int b)` 的名称不再是 `add`，可能是 `?add@@YAHHH@Z`、`_add@8` 或其他名称，具体取决于编译器和调用约定。

解决名称改编问题有几种方法：

- 使用 `extern "C"` 导出函数以切换到使用 C 名称修饰的 C 外部链接：

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
[DllImport("myDLL.dll")]
```

函数名仍然会被修饰 (`_add@8`)，但 `StdCall+extern "C"` 的名称修饰被 C# 编译器识别。

- 在 `myDLL.def` 模块定义文件中指定导出函数名：

```
EXPORTS
add

int __stdcall add(int a, int b)
[DllImport("myDLL.dll")]
```

在这种情况下，函数名将是纯粹的 `add`。

- 导入修饰名称。你需要使用某些 DLL 查看器查看修饰名称，然后可以显式指定它：

```
__declspec(dllexport) int __stdcall add(int a, int b)
[DllImport("myDLL.dll", EntryPoint = "?add@@YGHHH@Z")]
```

第 155.4 节：非托管

DLL 的动态加载和卸载

使用 `DllImport` 属性时，必须在编译时知道正确的 dll 和方法名称。如果想更灵活一些，在运行时决定加载哪个 dll 和方法，可以使用 Windows API 方法 `LoadLibrary()`、`GetProcAddress()` 和 `FreeLibrary()`。如果所使用的库依赖于运行时条件，这会很有帮助

If you want to use a function with Cdecl calling convention and a mangled name, your code will look like this:

```
__declspec(dllexport) int add(int a, int b)
[DllImport("myDLL.dll", CallingConvention = CallingConvention.Cdecl,
EntryPoint = "?add@@YAHHH@Z")]
```

- **thiscall(_thiscall)** 主要用于类的成员函数。
- When a function uses **thiscall(_thiscall)**，a pointer to the class is passed down as the first parameter.

Section 155.3: C++ name mangling

C++ compilers encode additional information in the names of exported functions, such as argument types, to make overloads with different arguments possible. This process is called [name mangling](#). This causes problems with importing functions in C# (and interop with other languages in general), as the name of `int add(int a, int b)` function is no longer `add`, it can be `?add@@YAHHH@Z`, `_add@8` or anything else, depending on the compiler and the calling convention.

There're several ways of solving the problem of name mangling:

- Exporting functions using `extern "C"` to switch to C external linkage which uses C name mangling:

```
extern "C" __declspec(dllexport) int __stdcall add(int a, int b)
[DllImport("myDLL.dll")]
```

Function name will still be mangled (`_add@8`), but `StdCall+extern "C"` name mangling is recognized by C# compiler.

- Specifying exported function names in `myDLL.def` module definition file:

```
EXPORTS
add

int __stdcall add(int a, int b)
[DllImport("myDLL.dll")]
```

The function name will be pure `add` in this case.

- Importing mangled name. You'll need some DLL viewer to see the mangled name, then you can specify it explicitly:

```
__declspec(dllexport) int __stdcall add(int a, int b)
[DllImport("myDLL.dll", EntryPoint = "?add@@YGHHH@Z")]
```

Section 155.4: Dynamic loading and unloading of unmanaged DLLs

When using the `DllImport` attribute you have to know the correct dll and method name at *compile time*. If you want to be more flexible and decide at *runtime* which dll and methods to load, you can use the Windows API methods `LoadLibrary()`, [GetProcAddress\(\)](#) and `FreeLibrary()`. This can be helpful if the library to use depends on runtime

条件。

GetProcAddress()返回的指针可以使用 [Marshal.GetDelegateForFunctionPointer\(\)](#) 转换为委托。

下面的代码示例演示了如何使用之前示例中的myDLL.dll：

```
class 程序
{
    // 导入必要的API, 如其他示例所示
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr LoadLibrary(string lib);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern void FreeLibrary(IntPtr module);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr GetProcAddress(IntPtr module, string proc);

    // 声明具有所需签名的委托
    private delegate int AddDelegate(int a, int b);

    private static void Main()
    {
        // 加载dll
        IntPtr module = LoadLibrary("myDLL.dll");
        if (module == IntPtr.Zero) // 错误处理
        {
            Console.WriteLine($"无法加载库: {Marshal.GetLastWin32Error()}");
            return;
        }

        // 获取方法的“指针”
        IntPtr method = GetProcAddress(module, "add");
        if (method == IntPtr.Zero) // 错误处理
        {
            Console.WriteLine($"无法加载方法: {Marshal.GetLastWin32Error()}");
            FreeLibrary(module); // 卸载库
            return;
        }

        // 将“指针”转换为委托
        AddDelegate add = (AddDelegate)Marshal.GetDelegateForFunctionPointer(method,
        typeof(AddDelegate));

        // 使用函数
        int result = add(750, 300);

        // 卸载库
        FreeLibrary(module);
    }
}
```

第155.5节：使用Marshal读取结构体

Marshal类包含一个名为**PtrToStructure**的函数，该函数使我们能够通过非托管指针读取结构体。

PtrToStructure函数有多个重载，但它们的意图都是相同的。

泛型PtrToStructure：

conditions.

The pointer returned by GetProcAddress() can be casted into a delegate using [Marshal.GetDelegateForFunctionPointer\(\)](#).

The following code sample demonstrates this with the myDLL.dll from the previous examples:

```
class Program
{
    // import necessary API as shown in other examples
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr LoadLibrary(string lib);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern void FreeLibrary(IntPtr module);
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern IntPtr GetProcAddress(IntPtr module, string proc);

    // declare a delegate with the required signature
    private delegate int AddDelegate(int a, int b);

    private static void Main()
    {
        // load the dll
        IntPtr module = LoadLibrary("myDLL.dll");
        if (module == IntPtr.Zero) // error handling
        {
            Console.WriteLine($"Could not load library: {Marshal.GetLastWin32Error()}");
            return;
        }

        // get a "pointer" to the method
        IntPtr method = GetProcAddress(module, "add");
        if (method == IntPtr.Zero) // error handling
        {
            Console.WriteLine($"Could not load method: {Marshal.GetLastWin32Error()}");
            FreeLibrary(module); // unload library
            return;
        }

        // convert "pointer" to delegate
        AddDelegate add = (AddDelegate)Marshal.GetDelegateForFunctionPointer(method,
        typeof(AddDelegate));

        // use function
        int result = add(750, 300);

        // unload library
        FreeLibrary(module);
    }
}
```

Section 155.5: Reading structures with Marshal

Marshal class contains a function named **PtrToStructure**, this function gives us the ability of reading structures by an unmanaged pointer.

PtrToStructure function got many overloads, but they all have the same intention.

Generic PtrToStructure:

```
public static T PtrToStructure<T>(IntPtr ptr);
```

T - 结构体类型。

ptr - 指向非托管内存块的指针。

示例：

```
NATIVE_STRUCT 结果 = Marshal.PtrToStructure<NATIVE_STRUCT>(ptr);
```

- 如果在读取本机结构体时处理托管对象，别忘了固定你的对象 :)

```
T Read<T>(byte[] buffer)
{
    T result = default(T);

    var gch = GCHandle.Alloc(buffer, GCHandleType.Pinned);

    try
    {
        result = Marshal.PtrToStructure<T>(gch.AddrOfPinnedObject());
    }
    finally
    {
        gch.Free();
    }

    return result;
}
```

第155.6节：处理Win32错误

使用互操作方法时，可以使用GetLastError API获取有关API调用的更多信息。

DllImport 属性 SetLastError 属性

SetLastError=true

表示被调用方将调用 SetLastError (Win32 API函数)。

SetLastError=false

表示被调用方**不会**调用 SetLastError (Win32 API 函数)，因此您将无法获得错误信息。

- 当未设置 SetLastError 时，默认值为 false。
- 您可以使用 Marshal.GetLastWin32Error 方法获取错误代码：

示例：

```
[DllImport("kernel32.dll", SetLastError=true)]
public static extern IntPtr OpenMutex(uint access, bool handle, string lpName);
```

如果您尝试打开不存在的互斥体，GetLastError 将返回ERROR_FILE_NOT_FOUND。

```
var lastErrorCode = Marshal.GetLastWin32Error();
```

```
public static T PtrToStructure<T>(IntPtr ptr);
```

T - 结构体类型。

ptr - A pointer to an unmanaged block of memory.

Example:

```
NATIVE_STRUCT result = Marshal.PtrToStructure<NATIVE_STRUCT>(ptr);
```

- If you dealing with managed objects while reading native structures, don't forget to pin your object :)

```
T Read<T>(byte[] buffer)
{
    T result = default(T);

    var gch = GCHandle.Alloc(buffer, GCHandleType.Pinned);

    try
    {
        result = Marshal.PtrToStructure<T>(gch.AddrOfPinnedObject());
    }
    finally
    {
        gch.Free();
    }

    return result;
}
```

Section 155.6: Dealing with Win32 Errors

When using interop methods, you can use **GetLastError** API to get additional information on you API calls.

DllImport Attribute SetLastError Attribute

SetLastError=true

Indicates that the callee will call SetLastError (Win32 API function).

SetLastError=false

Indicates that the callee **will not** call SetLastError (Win32 API function), therefore you will not get an error information.

- When SetLastError isn't set, it is set to false (Default value).
- You can obtain the error code using Marshal.GetLastWin32Error Method:

Example:

```
[DllImport("kernel32.dll", SetLastError=true)]
public static extern IntPtr OpenMutex(uint access, bool handle, string lpName);
```

If you trying to open mutex which does not exist, GetLastError will return **ERROR_FILE_NOT_FOUND**.

```
var lastErrorCode = Marshal.GetLastWin32Error();
```

```

if (lastErrorCode == (uint)ERROR_FILE_NOT_FOUND)
{
    //处理错误
}

```

系统错误代码可在此处找到：

[https://msdn.microsoft.com/en-us/library/windows/desktop/ms681382\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms681382(v=vs.85).aspx)

GetLastError API

还有一个本地的GetLastError API 你也可以使用：

```

[DllImport("coredll.dll", SetLastError=true)]
static extern Int32 GetLastError();

```

- 从托管代码调用 Win32 API 时，必须始终使用Marshal.GetLastWin32Error。

原因如下：

在你的 Win32 调用设置错误（调用 SetLastError）之后，CLR 可能会调用其他 Win32 调用，这些调用也可能调用 SetLastError，这种行为可能会覆盖你的错误值。在这种情况下，如果你调用GetLastError，可能会获得无效的错误码。

设置SetLastError = true，确保 CLR 在执行其他 Win32 调用之前检索错误代码。

第 155.7 节：固定对象

GC（垃圾回收器）负责清理我们的垃圾。

当GC清理我们的垃圾时，它会从托管堆中移除未使用的对象，这些对象会导致堆碎片。当GC完成移除后，它会执行堆压缩（碎片整理），这涉及到在堆上移动对象。

由于GC不是确定性的，当将托管对象引用/指针传递给本地代码时，GC可能随时启动，如果它发生在Interop调用之后，很有可能对象（传递给本地的引用）会在托管堆上被移动——结果，我们在托管端得到一个无效的引用。

在这种情况下，您应该在将对象传递给本地代码之前pin该对象。

固定对象

固定对象是指不允许被GC移动的对象。

GC固定句柄

您可以使用Gc.Alloc方法创建一个固定对象

```
GCHandle handle = GCHandle.Alloc(yourObject, GCHandleType.Pinned);
```

- 获取托管对象的固定GCHandle会将该特定对象标记为GC不能移动的对象，直到释放该句柄为止
GC，直到释放该句柄

示例：

```
[DllImport("kernel32.dll", SetLastError = true)]
```

```

if (lastErrorCode == (uint)ERROR_FILE_NOT_FOUND)
{
    //Deal with error
}

```

System Error Codes can be found here:

[https://msdn.microsoft.com/en-us/library/windows/desktop/ms681382\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms681382(v=vs.85).aspx)

GetLastError API

There is a native **GetLastError** API which you can use as well :

```

[DllImport("coredll.dll", SetLastError=true)]
static extern Int32 GetLastError();

```

- When calling Win32 API from managed code, you must always use the **Marshal.GetLastWin32Error**.

Here's why:

Between your Win32 call which sets the error (calls SetLastError), the CLR can call other Win32 calls which could call **SetLastError** as well, this behavior can override your error value. In this scenario, if you call **GetLastError** you can obtain an invalid error.

Setting **SetLastError = true**, makes sure that the CLR retrieves the error code before it executes other Win32 calls.

Section 155.7: Pinned Object

GC (Garbage Collector) is responsible for cleaning our garbage.

While **GC** cleans our garbage, he removes the unused objects from the managed heap which cause heap fragmentation. When **GC** is done with the removal, it performs a heap compression (defragmentation) which involves moving objects on the heap.

Since **GC** isn't deterministic, when passing managed object reference/pointer to native code, **GC** can kick in at any time, if it occurs just after Interop call, there is a very good possibility that object (which reference passed to native) will be moved on the managed heap - as a result, we get an invalid reference on managed side.

In this scenario, you should **pin** the object before passing it to native code.

Pinned Object

Pinned object is an object that is not allowed to move by GC.

Gc Pinned Handle

You can create a pin object using **Gc.Alloc** method

```
GCHandle handle = GCHandle.Alloc(yourObject, GCHandleType.Pinned);
```

- Obtaining a pinned **GCHandle** to managed object marks a specific object as one that cannot be moved by **GC**, until freeing the handle

Example:

```
[DllImport("kernel32.dll", SetLastError = true)]
```

```

public static extern void EnterCriticalSection(IntPtr ptr);

[DllImport("kernel32.dll", SetLastError = true)]
public static extern void LeaveCriticalSection(IntPtr ptr);

public void EnterCriticalSection(CRITICAL_SECTION section)
{
    try
    {
        GCHandle handle = GCHandle.Alloc(section, GCHandleType.Pinned);
        EnterCriticalSection(handle.AddrOfPinnedObject());
        //执行一些关键工作
        LeaveCriticalSection(handle.AddrOfPinnedObject());
    }
    finally
    {
        handle.Free();
    }
}

```

注意事项

- 在固定对象（尤其是大型对象）时，尽量尽快释放固定的**GCHandle**，因为它会中断堆的碎片整理。
- 如果你忘记释放**GCHandle**，将不会有任何释放。请在安全代码段（例如 `finally`）中执行此操作

第155.8节：用于COM暴露类的简单代码

```

using System;
using System.Runtime.InteropServices;

命名空间 ComLibrary
{
    [ComVisible(true)]
    公共接口 IMainType
    {
        int GetInt();

        void StartTime();

        int StopTime();
    }

    [ComVisible(true)]
    [ClassInterface(ClassInterfaceType.None)]
    公共类 MainType : IMainType
    {
        私有 Stopwatch stopWatch;

        公共 int GetInt()
        {
            return 0;
        }

        公共 void StartTime()
        {
stopWatch= new Stopwatch();
            stopWatch.Start();
        }
    }
}

```

```

public static extern void EnterCriticalSection(IntPtr ptr);

[DllImport("kernel32.dll", SetLastError = true)]
public static extern void LeaveCriticalSection(IntPtr ptr);

public void EnterCriticalSection(CRITICAL_SECTION section)
{
    try
    {
        GCHandle handle = GCHandle.Alloc(section, GCHandleType.Pinned);
        EnterCriticalSection(handle.AddrOfPinnedObject());
        //Do Some Critical Work
        LeaveCriticalSection(handle.AddrOfPinnedObject());
    }
    finally
    {
        handle.Free();
    }
}

```

Precautions

- When pinning (especially large ones) object try to release the pinned **GCHandle** as fast as possible, since it interrupt heap defragmentation.
- If you forget to free **GCHandle** nothing will. Do it in a safe code section (such as `finally`)

Section 155.8: Simple code to expose class for com

```

using System;
using System.Runtime.InteropServices;

namespace ComLibrary
{
    [ComVisible(true)]
    public interface IMainType
    {
        int GetInt();

        void StartTime();

        int StopTime();
    }

    [ComVisible(true)]
    [ClassInterface(ClassInterfaceType.None)]
    public class MainType : IMainType
    {
        private Stopwatch stopWatch;

        public int GetInt()
        {
            return 0;
        }

        public void StartTime()
        {
            stopWatch= new Stopwatch();
            stopWatch.Start();
        }
    }
}

```

```
public int StopTime()
{
    return (int)stopWatch.ElapsedMilliseconds;
}
}
```

```
public int StopTime()
{
    return (int)stopWatch.ElapsedMilliseconds;
}
}
```

belindoc.com

第156章：.NET 编译器平台 (Roslyn)

第156.1节：语义模型

语义模型 (Semantic Model) 相比语法树提供了更深层次的代码解释和洞察。语法树可以告诉变量的名称，而语义模型还提供类型和所有引用。语法树能识别方法调用，但语义模型能给出方法声明的精确位置引用（在应用重载解析之后）。

```
var workspace = Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace.Create();
var sln = await workspace.OpenSolutionAsync(solutionFilePath);
var project = sln.Projects.First();
var compilation = await project.GetCompilationAsync();

foreach (var syntaxTree in compilation.SyntaxTrees)
{
    var root = await syntaxTree.GetRootAsync();

    var declaredIdentifiers = root.DescendantNodes()
        .Where(an => an is VariableDeclaratorSyntax)
        .Cast<VariableDeclaratorSyntax>();

    foreach (var di in declaredIdentifiers)
    {
        Console.WriteLine(di.Identifier);
        // => "root"

        var variableSymbol = compilation
            .GetSemanticModel(syntaxTree)
            .GetDeclaredSymbol(di) as ILocalSymbol;

        Console.WriteLine(variableSymbol.Type);
        // => "Microsoft.CodeAnalysis.SyntaxNode"

        var references = await SymbolFinder.FindReferencesAsync(variableSymbol, sln);
        foreach (var reference in references)
        {
            foreach (var loc in reference.Locations)
            {
                Console.WriteLine(loc.Location.SourceSpan);
                // => "[1375..1379]"
            }
        }
    }
}
```

这会使用语法树输出本地变量列表。然后它会查询语义模型以获取完整的类型名称，并查找每个变量的所有引用。

第156.2节：语法树

语法树 (Syntax Tree) 是一种不可变的数据结构，表示程序为由名称、命令和标记组成的树状结构（如编辑器中先前配置的那样）。

例如，假设已配置了一个名为 `compilation` 的 `Microsoft.CodeAnalysis.Compilation` 实例。

Chapter 156: .NET Compiler Platform (Roslyn)

Section 156.1: Semantic model

A **Semantic Model** offers a deeper level of interpretation and insight of code compare to a syntax tree. Where syntax trees can tell the names of variables, semantic models also give the type and all references. Syntax trees notice method calls, but semantic models give references to the precise location the method is declared (after overload resolution has been applied.)

```
var workspace = Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace.Create();
var sln = await workspace.OpenSolutionAsync(solutionFilePath);
var project = sln.Projects.First();
var compilation = await project.GetCompilationAsync();

foreach (var syntaxTree in compilation.SyntaxTrees)
{
    var root = await syntaxTree.GetRootAsync();

    var declaredIdentifiers = root.DescendantNodes()
        .Where(an => an is VariableDeclaratorSyntax)
        .Cast<VariableDeclaratorSyntax>();

    foreach (var di in declaredIdentifiers)
    {
        Console.WriteLine(di.Identifier);
        // => "root"

        var variableSymbol = compilation
            .GetSemanticModel(syntaxTree)
            .GetDeclaredSymbol(di) as ILocalSymbol;

        Console.WriteLine(variableSymbol.Type);
        // => "Microsoft.CodeAnalysis.SyntaxNode"

        var references = await SymbolFinder.FindReferencesAsync(variableSymbol, sln);
        foreach (var reference in references)
        {
            foreach (var loc in reference.Locations)
            {
                Console.WriteLine(loc.Location.SourceSpan);
                // => "[1375..1379]"
            }
        }
    }
}
```

This outputs a list of local variables using a syntax tree. Then it consults the semantic model to get the full type name and find all references of every variable.

Section 156.2: Syntax tree

A **Syntax Tree** is an immutable data structure representing the program as a tree of names, commands and marks (as previously configured in the editor.)

For example, assume a `Microsoft.CodeAnalysis.Compilation` instance named `compilation` has been configured.

有多种方法可以列出已加载代码中声明的每个变量的名称。简单地做法是，获取每个文档中的所有语法片段（使用 DescendantNodes 方法），并用 Linq 选择描述变量声明的节点：

```
foreach (var syntaxTree in compilation.SyntaxTrees)
{
    var root = await syntaxTree.GetRootAsync();
    var declaredIdentifiers = root.DescendantNodes()
        .Where(an => an is VariableDeclaratorSyntax)
        .Cast<VariableDeclaratorSyntax>()
        .Select(vd => vd.Identifier);

    foreach (var di in declaredIdentifiers)
    {
        Console.WriteLine(di);
    }
}
```

语法树中将包含每种对应类型的 C# 结构。要快速查找特定类型，可以使用 Visual Studio 中的语法可视化器（Syntax Visualizer）窗口。它会将当前打开的文档解释为 Roslyn 语法树。

第156.3节：从MSBuild项目创建工作区

首先获取 Microsoft.CodeAnalysis.CSharp.Workspaces nuget 后继续。

```
var workspace = Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace.Create();
var project = await workspace.OpenProjectAsync(projectFilePath);
var compilation = await project.GetCompilationAsync();

foreach (var diagnostic in compilation.GetDiagnostics()
    .Where(d => d.Severity == Microsoft.CodeAnalysis.DiagnosticSeverity.Error))
{
    Console.WriteLine(diagnostic);
}
```

将现有代码加载到工作区，编译并报告错误。之后代码将驻留在内存中。
从这里开始，语法和语义两方面都可以进行操作。

There are multiple ways to list the names of every variable declared in the loaded code. To do so naively, take all pieces of syntax in every document (the DescendantNodes method) and use Linq to select nodes that describe variable declaration:

```
foreach (var syntaxTree in compilation.SyntaxTrees)
{
    var root = await syntaxTree.GetRootAsync();
    var declaredIdentifiers = root.DescendantNodes()
        .Where(an => an is VariableDeclaratorSyntax)
        .Cast<VariableDeclaratorSyntax>()
        .Select(vd => vd.Identifier);

    foreach (var di in declaredIdentifiers)
    {
        Console.WriteLine(di);
    }
}
```

Every type of C# construct with a corresponding type will exist in the syntax tree. To quickly find specific types, use the Syntax Visualizer window from Visual Studio. This will interpret the current opened document as a Roslyn syntax tree.

Section 156.3: Create workspace from MSBuild project

First obtain the Microsoft.CodeAnalysis.CSharp.Workspaces nuget before continuing.

```
var workspace = Microsoft.CodeAnalysis.MSBuild.MSBuildWorkspace.Create();
var project = await workspace.OpenProjectAsync(projectFilePath);
var compilation = await project.GetCompilationAsync();

foreach (var diagnostic in compilation.GetDiagnostics()
    .Where(d => d.Severity == Microsoft.CodeAnalysis.DiagnosticSeverity.Error))
{
    Console.WriteLine(diagnostic);
}
```

To load existing code to the workspace, compile and report errors. Afterwards the code will be located in memory.
From here, both the syntactic and semantic side will be available to work with.

第157章：IL生成器

第157.1节：创建包含 UnixTimestamp辅助方法的动态程序集

此示例展示了ILGenerator的用法，通过生成代码来使用已存在和新创建的成员以及基本的异常处理。以下代码生成了一个动态程序集，该程序集包含与以下C#代码等效的内容：

```
public static class UnixTimeHelper
{
    private readonly static DateTime EpochTime = new DateTime(1970, 1, 1);

    public static int UnixTimestamp(DateTime input)
    {
        int totalSeconds;
        try
        {
            totalSeconds = checked((int)input.Subtract(UnixTimeHelper.EpochTime).TotalSeconds);
        }
        catch (OverflowException overflowException)
        {
            throw new InvalidOperationException("Int32 时间戳太晚了。",
overflowException);
        }
        return totalSeconds;
    }
}

//获取所需方法
var dateTimeCtor = typeof(DateTime)
    .GetConstructor(new[] {typeof(int), typeof(int), typeof(int)});
var dateTimeSubtract = typeof(DateTime)
    .GetMethod(nameof(DateTime.Subtract), new[] {typeof(DateTime)});
var timeSpanSecondsGetter = typeof(TimeSpan)
    .GetProperty(nameof(TimeSpan.TotalSeconds)).GetGetMethod();
var invalidOperationCtor = typeof(InvalidOperationException)
    .GetConstructor(new[] {typeof(string), typeof(Exception)});

if (dateTimeCtor == null || dateTimeSubtract == null ||
    timeSpanSecondsGetter == null || invalidOperationCtor == null)
{
    throw new Exception("找不到所需方法，无法创建程序集。");
}

//设置所需成员
var an = new AssemblyName("UnixTimeAsm");
var dynAsm = AppDomain.CurrentDomain.DefineDynamicAssembly(an, AssemblyBuilderAccess.RunAndSave);
var dynMod = dynAsm.DefineDynamicModule(an.Name, an.Name + ".dll");

var dynType = dynMod.DefineType("UnixTimeHelper",
    TypeAttributes.Abstract | TypeAttributes.Sealed | TypeAttributes.Public);

var epochTimeField = dynType.DefineField("EpochStartTime", typeof(DateTime),
    FieldAttributes.Private | FieldAttributes.Static | FieldAttributes.InitOnly);

var cctor =
    dynType.DefineConstructor(
```

Chapter 157: ILGenerator

Section 157.1: Creates a DynamicAssembly that contains a UnixTimestamp helper method

This example shows the usage of the ILGenerator by generating code that makes use of already existing and new created members as well as basic Exception handling. The following code emits a DynamicAssembly that contains an equivalent to this c# code:

```
public static class UnixTimeHelper
{
    private readonly static DateTime EpochTime = new DateTime(1970, 1, 1);

    public static int UnixTimestamp(DateTime input)
    {
        int totalSeconds;
        try
        {
            totalSeconds = checked((int)input.Subtract(UnixTimeHelper.EpochTime).TotalSeconds);
        }
        catch (OverflowException overflowException)
        {
            throw new InvalidOperationException("It's too late for an Int32 timestamp.",
overflowException);
        }
        return totalSeconds;
    }
}

//Get the required methods
var dateTimeCtor = typeof(DateTime)
    .GetConstructor(new[] {typeof(int), typeof(int), typeof(int)});
var dateTimeSubtract = typeof(DateTime)
    .GetMethod(nameof(DateTime.Subtract), new[] {typeof(DateTime)});
var timeSpanSecondsGetter = typeof(TimeSpan)
    .GetProperty(nameof(TimeSpan.TotalSeconds)).GetGetMethod();
var invalidOperationCtor = typeof(InvalidOperationException)
    .GetConstructor(new[] {typeof(string), typeof(Exception)});

if (dateTimeCtor == null || dateTimeSubtract == null ||
    timeSpanSecondsGetter == null || invalidOperationCtor == null)
{
    throw new Exception("Could not find a required Method, can not create Assembly.");
}

//Setup the required members
var an = new AssemblyName("UnixTimeAsm");
var dynAsm = AppDomain.CurrentDomain.DefineDynamicAssembly(an, AssemblyBuilderAccess.RunAndSave);
var dynMod = dynAsm.DefineDynamicModule(an.Name, an.Name + ".dll");

var dynType = dynMod.DefineType("UnixTimeHelper",
    TypeAttributes.Abstract | TypeAttributes.Sealed | TypeAttributes.Public);

var epochTimeField = dynType.DefineField("EpochStartTime", typeof(DateTime),
    FieldAttributes.Private | FieldAttributes.Static | FieldAttributes.InitOnly);

var cctor =
    dynType.DefineConstructor(
```

```

MethodAttributes.Private | MethodAttributes.HideBySig | MethodAttributes.SpecialName |
MethodAttributes.RTSpecialName | MethodAttributes.Static, CallingConventions.Standard,
Type.EmptyTypes);

var cctorGen = cctor.GetILGenerator();
cctorGen.Emit(OpCodes.Ldc_I4, 1970); //将 DateTime 构造函数的参数加载到堆栈上
cctorGen.Emit(OpCodes.Ldc_I4_1);
cctorGen.Emit(OpCodes.Ldc_I4_1);
cctorGen.Emit(OpCodes.Newobj, dateTimeCtor); //调用构造函数
cctorGen.Emit(OpCodes.Stsfld, epochTimeField); //将对象存储到静态字段中
cctorGen.Emit(OpCodes.Ret);

var unixTimestampMethod = dynType.DefineMethod("UnixTimestamp",
    MethodAttributes.Public | MethodAttributes.HideBySig | MethodAttributes.Static,
    CallingConventions.Standard, typeof(int), new[] {typeof(DateTime)});

unixTimestampMethod.DefineParameter(1, ParameterAttributes.None, "input");

var methodGen = unixTimestampMethod.GetILGenerator();
methodGen.DeclareLocal(typeof(TimeSpan));
methodGen.DeclareLocal(typeof(int));
methodGen.DeclareLocal(typeof(OverflowException));

methodGen.BeginExceptionBlock(); //开始 try 块
methodGen.Emit(OpCodes.Ldarga_S, (byte)0); //要调用结构体上的方法，需要加载它的地址

methodGen.Emit(OpCodes.Ldsfld, epochTimeField);
    //加载我们创建的静态字段对象，作为后续调用的参数
methodGen.Emit(OpCodes.Call, dateTimeSubtract); //调用输入 DateTime 的减法方法
methodGen.Emit(OpCodes.Stloc_0); //将结果 TimeSpan 存储到局部变量
methodGen.Emit(OpCodes.Ldloca_S, (byte)0); //加载局部变量地址以调用其方法
methodGen.Emit(OpCodes.Call, timeSpanSecondsGetter); //调用 TimeSpan 的 TotalSeconds 获取方法

methodGen.Emit(OpCodes.Conv_Ovf_I4); //将结果转换为 Int32；溢出时抛出异常
methodGen.Emit(OpCodes.Stloc_1); //存储结果以便稍后返回
    //跳转到catch块后面的leave指令将自动生成
methodGen.BeginCatchBlock(typeof(OverflowException)); //开始catch块
    //当执行到这里时，抛出了一个OverflowException，该异常现在在线上
methodGen.Emit(OpCodes.Stloc_2); //将异常存储到局部变量中。
methodGen.Emit(OpCodes.Ldstr, "Int32时间戳已经太晚了。");
    //将错误信息加载到线上
methodGen.Emit(OpCodes.Ldloc_2); //再次加载异常
methodGen.Emit(OpCodes.Newobj, invalidOperationCtor);
    //使用我们的消息和内部异常创建一个InvalidOperationException
methodGen.Emit(OpCodes.Throw); //抛出创建的异常
methodGen.EndExceptionBlock(); //结束catch块
    //当执行到这里时，一切正常
methodGen.Emit(OpCodes.Ldloc_1); //加载结果值
methodGen.Emit(OpCodes.Ret); //返回结果

dynType.CreateType();

dynAsm.Save(an.Name + ".dll");

```

第157.2节：创建方法重写

此示例展示了如何在生成的类中重写ToString方法

```

// 创建一个程序集和新类型
var name = new AssemblyName("MethodOverriding");
var dynAsm = AppDomain.CurrentDomain.DefineDynamicAssembly(name, AssemblyBuilderAccess.RunAndSave);

```

```

MethodAttributes.Private | MethodAttributes.HideBySig | MethodAttributes.SpecialName |
MethodAttributes.RTSpecialName | MethodAttributes.Static, CallingConventions.Standard,
Type.EmptyTypes);

var cctorGen = cctor.GetILGenerator();
cctorGen.Emit(OpCodes.Ldc_I4, 1970); //Load the DateTime constructor arguments onto the stack
cctorGen.Emit(OpCodes.Ldc_I4_1);
cctorGen.Emit(OpCodes.Ldc_I4_1);
cctorGen.Emit(OpCodes.Newobj, dateTimeCtor); //Call the constructor
cctorGen.Emit(OpCodes.Stsfld, epochTimeField); //Store the object in the static field
cctorGen.Emit(OpCodes.Ret);

var unixTimestampMethod = dynType.DefineMethod("UnixTimestamp",
    MethodAttributes.Public | MethodAttributes.HideBySig | MethodAttributes.Static,
    CallingConventions.Standard, typeof(int), new[] {typeof(DateTime)});

unixTimestampMethod.DefineParameter(1, ParameterAttributes.None, "input");

var methodGen = unixTimestampMethod.GetILGenerator();
methodGen.DeclareLocal(typeof(TimeSpan));
methodGen.DeclareLocal(typeof(int));
methodGen.DeclareLocal(typeof(OverflowException));

methodGen.BeginExceptionBlock(); //Begin the try block
methodGen.Emit(OpCodes.Ldarga_S, (byte)0); //To call a method on a struct we need to load the
    //address of it
methodGen.Emit(OpCodes.Ldsfld, epochTimeField);
    //Load the object of the static field we created as argument for the following call
methodGen.Emit(OpCodes.Call, dateTimeSubtract); //Call the subtract method on the input DateTime
methodGen.Emit(OpCodes.Stloc_0); //Store the resulting TimeSpan in a local
methodGen.Emit(OpCodes.Ldloca_S, (byte)0); //Load the locals address to call a method on it
methodGen.Emit(OpCodes.Call, timeSpanSecondsGetter); //Call the TotalSeconds Get method on the
    //TimeSpan
methodGen.Emit(OpCodes.Conv_Ovf_I4); //Convert the result to Int32; throws an exception on overflow
methodGen.Emit(OpCodes.Stloc_1); //store the result for returning later
    //The leave instruction to jump behind the catch block will be automatically emitted
methodGen.BeginCatchBlock(typeof(OverflowException)); //Begin the catch block
    //When we are here, an OverflowException was thrown, that is now on the stack
methodGen.Emit(OpCodes.Stloc_2); //Store the exception in a local.
methodGen.Emit(OpCodes.Ldstr, "It's too late for an Int32 timestamp.");
    //Load our error message onto the stack
methodGen.Emit(OpCodes.Ldloc_2); //Load the exception again
methodGen.Emit(OpCodes.Newobj, invalidOperationCtor);
    //Create an InvalidOperationException with our message and inner Exception
methodGen.Emit(OpCodes.Throw); //Throw the created exception
methodGen.EndExceptionBlock(); //End the catch block
    //When we are here, everything is fine
methodGen.Emit(OpCodes.Ldloc_1); //Load the result value
methodGen.Emit(OpCodes.Ret); //Return it

dynType.CreateType();

dynAsm.Save(an.Name + ".dll");

```

Section 157.2: Create method override

This example shows how to override ToString method in generated class

```

// create an Assembly and new type
var name = new AssemblyName("MethodOverriding");
var dynAsm = AppDomain.CurrentDomain.DefineDynamicAssembly(name, AssemblyBuilderAccess.RunAndSave);

```

```

var dynModule = dynAsm.DefineDynamicModule(name.Name, $"{name.Name}.dll");
var typeBuilder = dynModule.DefineType("MyClass", TypeAttributes.Public | TypeAttributes.Class);

// 定义一个新方法
var toStr = typeBuilder.DefineMethod(
    "ToString", // 名称
    MethodAttributes.Public | MethodAttributes.Virtual, // 修饰符
    typeof(string), // 返回类型
    Type.EmptyTypes); // 参数类型
var ilGen = toStr.GetILGenerator();
ilGen.Emit(OpCodes.Ldstr, "Hello, world!");
ilGen.Emit(OpCodes.Ret);

// 将此方法设置为 object.ToString 的重写
typeBuilder.DefineMethodOverride(toStr, typeof(object).GetMethod("ToString"));
var type = typeBuilder.CreateType();

// 现在测试它：
var instance = Activator.CreateInstance(type);
Console.WriteLine(instance.ToString());

```

```

var dynModule = dynAsm.DefineDynamicModule(name.Name, $"{name.Name}.dll");
var typeBuilder = dynModule.DefineType("MyClass", TypeAttributes.Public | TypeAttributes.Class);

// define a new method
var toStr = typeBuilder.DefineMethod(
    "ToString", // name
    MethodAttributes.Public | MethodAttributes.Virtual, // modifiers
    typeof(string), // return type
    Type.EmptyTypes); // argument types
var ilGen = toStr.GetILGenerator();
ilGen.Emit(OpCodes.Ldstr, "Hello, world!");
ilGen.Emit(OpCodes.Ret);

// set this method as override of object.ToString
typeBuilder.DefineMethodOverride(toStr, typeof(object).GetMethod("ToString"));
var type = typeBuilder.CreateType();

// now test it:
var instance = Activator.CreateInstance(type);
Console.WriteLine(instance.ToString());

```

第158章：T4代码生成

第158.1节：运行时代码生成

```
<#@ template language="C#" #> // 您项目的语言  
<#@ assembly name="System.Core" #>  
<#@ import namespace="System.Linq" #>  
<#@ import namespace="System.Text" #>  
<#@ import namespace="System.Collections.Generic" #>
```

belindoc.com

Chapter 158: T4 Code Generation

Section 158.1: Runtime Code Generation

```
<#@ template language="C#" #> //Language of your project  
<#@ assembly name="System.Core" #>  
<#@ import namespace="System.Linq" #>  
<#@ import namespace="System.Text" #>  
<#@ import namespace="System.Collections.Generic" #>
```

第159章：在Windows窗体应用程序中创建自定义消息框

首先我们需要了解什么是消息框（MessageBox）...

消息框控件显示带有指定文本的消息，并且可以通过指定自定义图像、标题和按钮组进行自定义（这些按钮组允许用户选择比基本的是/否更多的选项）。

通过创建我们自己的消息框，我们可以通过使用生成的dll或复制包含该类的文件，在任何新应用程序中重复使用该消息框控件。

第159.1节：如何在另一个Windows窗体应用程序中使用自己创建的消息框控件

要找到现有的.cs文件，请在Visual Studio中右键点击项目，然后选择“在文件资源管理器中打开文件夹”。

1. Visual Studio --> 当前项目（Windows窗体） --> 解决方案资源管理器 --> 项目名称 --> 右键点击 --> 添加 --> 现有项 --> 然后定位到你的现有.cs文件。
2. 现在还有最后一步，为了使用该控件，需要在代码中添加using语句，以便程序集知道其依赖项。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
```

//这是我们依赖项的using语句。

3. 要显示消息框，只需使用以下代码...

```
CustomMsgBox.Show("您的消息内容...","MSG","确定");
```

第159.2节：创建自定义消息框控件

要创建我们自己的消息框控件，只需按照以下指南操作...

1. 打开您的Visual Studio实例（VS 2008/2010/2012/2015/2017）
2. 点击顶部工具栏的文件 -> 新建项目 --> Windows窗体应用程序 --> 输入项目名称，然后点击确定。
3. 加载完成后，从工具箱（左侧）拖放一个按钮控件到窗体上（如下图所示）。

Chapter 159: Creating Own MessageBox in Windows Form Application

First we need to know what a MessageBox is...

The MessageBox control displays a message with specified text, and can be customised by specifying a custom image, title and button sets (These button sets allow the user to choose more than a basic yes/no answer).

By creating our own MessageBox we can re-use that MessageBox Control in any new applications just by using the generated dll, or copying the file containing the class.

Section 159.1: How to use own created MessageBox control in another Windows Form application

To find your existing .cs files, right click on the project in your instance of Visual Studio, and click Open Folder in File Explorer.

1. Visual Studio --> Your current project (Windows Form) --> Solution Explorer --> Project Name --> Right Click --> Add --> Existing Item --> Then locate your existing .cs file.
2. Now there's one last thing to do in order to use the control. Add a using statement to your code, so that your assembly knows about its dependencies.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;

using CustomMsgBox; //Here's the using statement for our dependency.
```

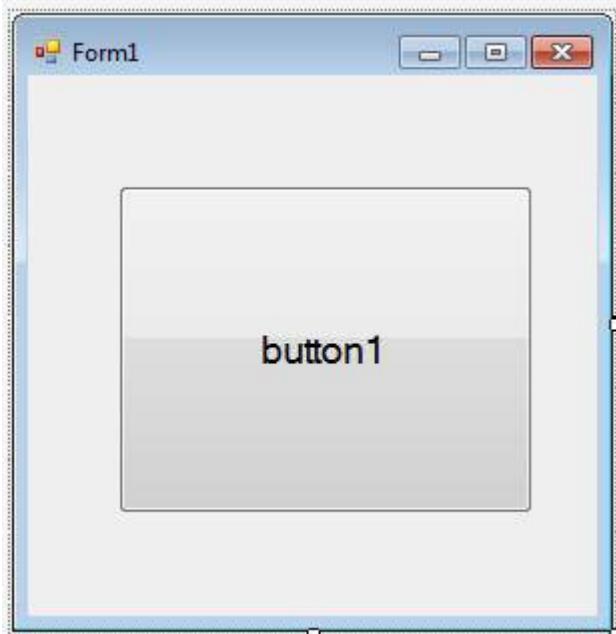
3. To display the messagebox, simply use the following...

```
CustomMsgBox.Show("Your Message for Message Box...","MSG","OK");
```

Section 159.2: Creating Own MessageBox Control

To create our own MessageBox control simply follow the guide below...

1. Open up your instance of Visual Studio (VS 2008/2010/2012/2015/2017)
2. Go to the toolbar at the top and click File -> New Project --> Windows Forms Application --> Give the project a name and then click ok.
3. Once loaded, drag and drop a button control from the Toolbox (found on the left) onto the form (as shown below).



4. 双击按钮，集成开发环境将自动为您生成点击事件处理程序。

5. 编辑窗体代码，使其如下所示（您可以右键点击窗体，选择编辑代码）：

```
namespace MsgBoxExample {
    public partial class MsgBoxExampleForm : Form {
        // 构造函数，在类初始化时调用。
        public MsgBoxExampleForm() {
            InitializeComponent();
        }

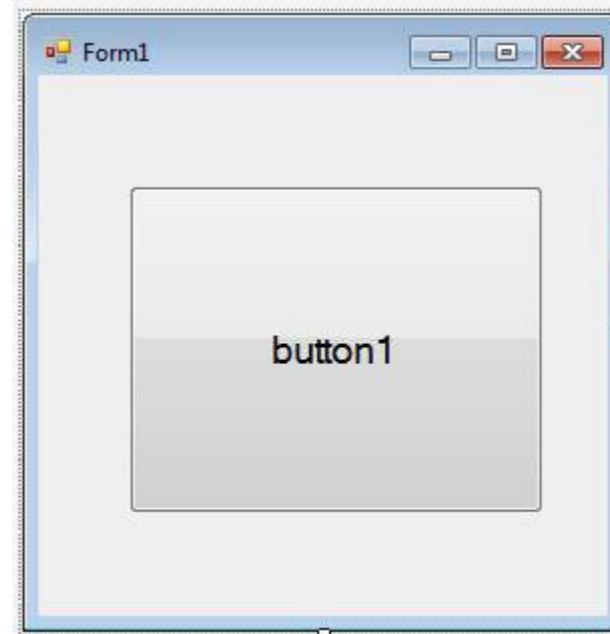
        // 每当按钮被点击时调用。
        private void btnShowMessageBox_Click(object sender, EventArgs e) {
            CustomMsgBox.Show($"I'm a {nameof(CustomMsgBox)}!", "MSG", "OK");
        }
    }
}
```

6. 解决方案资源管理器 -> 右键点击你的项目 --> 添加 --> Windows 窗体，命名为 "CustomMsgBox.cs"

7. 从工具箱拖入一个按钮和标签控件到窗体（完成后窗体大致如下所示）：



8. 现在将以下代码写入新创建的窗体：



4. Double click the button and the Integrated Development Environment will automatically generate the click event handler for you.

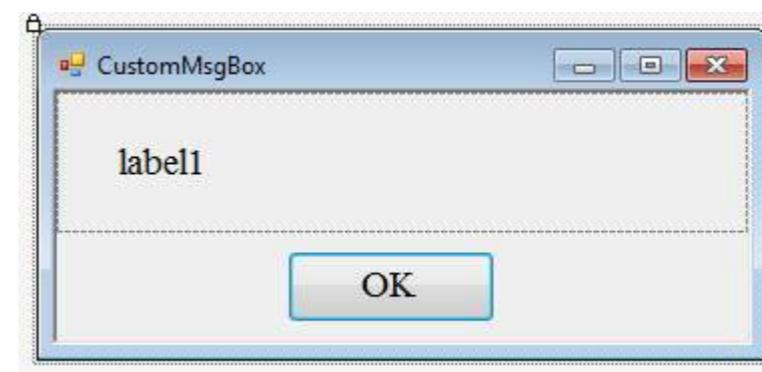
5. Edit the code for the form so that it looks like the following (You can right-click the form and click Edit Code):

```
namespace MsgBoxExample {
    public partial class MsgBoxExampleForm : Form {
        //Constructor, called when the class is initialised.
        public MsgBoxExampleForm() {
            InitializeComponent();
        }

        //Called whenever the button is clicked.
        private void btnShowMessageBox_Click(object sender, EventArgs e) {
            CustomMsgBox.Show($"I'm a {nameof(CustomMsgBox)}!", "MSG", "OK");
        }
    }
}
```

6. Solution Explorer -> Right Click on your project --> Add --> Windows Form and set the name as "CustomMsgBox.cs"

7. Drag in a button & label control from the Toolbox to the form (It'll look something like the form below after doing it):



8. Now write out the code below into the newly created form:

```

private DialogResult result = DialogResult.No;
public static DialogResult Show(string text, string caption, string btnOkText) {
    var msgBox = new CustomMsgBox();
    msgBox.lblText.Text = text; //标签的文本...
    msgBox.Text = caption; //窗体标题
    msgBox.btnOk.Text = btnOkText; //按钮上的文本
    //此方法是阻塞的，只有当用户
    //点击确定或关闭窗体时才会返回。
    msgBox.ShowDialog();
    return result;
}

private void btnOk_Click(object sender, EventArgs e) {
    result = DialogResult.Yes;
    MsgBox.Close();
}

```

9. 现在只需按F5键运行程序。恭喜，你已经制作了一个可重用控件。

belindoc.com

```

private DialogResult result = DialogResult.No;
public static DialogResult Show(string text, string caption, string btnOkText) {
    var msgBox = new CustomMsgBox();
    msgBox.lblText.Text = text; //The text for the label...
    msgBox.Text = caption; //Title of form
    msgBox.btnOk.Text = btnOkText; //Text on the button
    //This method is blocking, and will only return once the user
    //clicks ok or closes the form.
    msgBox.ShowDialog();
    return result;
}

private void btnOk_Click(object sender, EventArgs e) {
    result = DialogResult.Yes;
    MsgBox.Close();
}

```

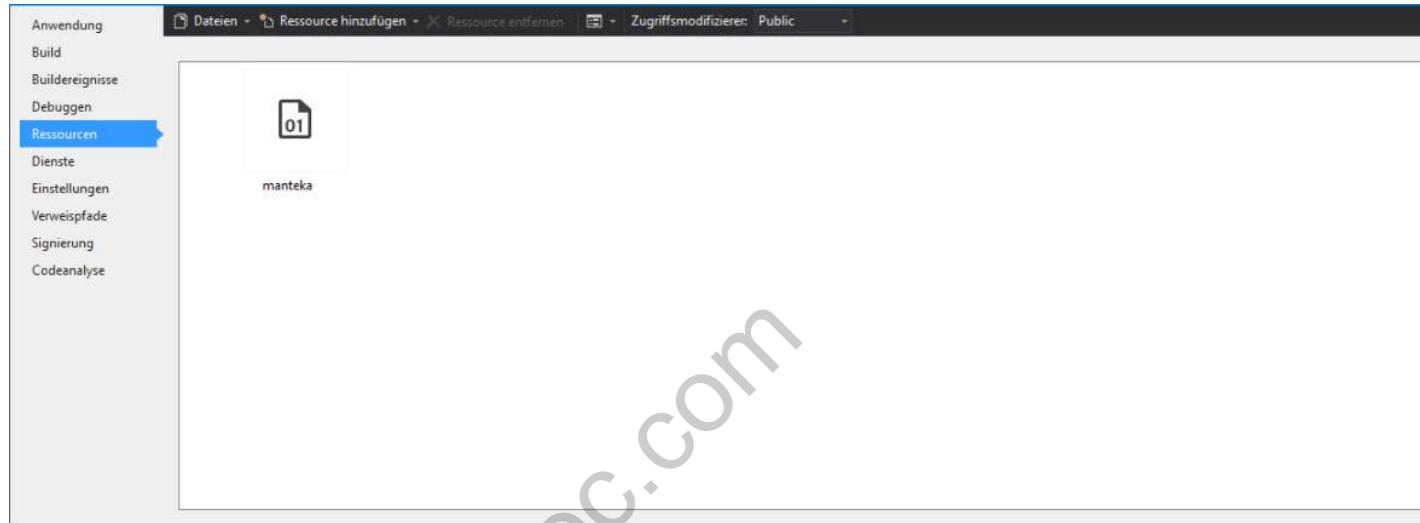
9. Now run the program by just pressing F5 Key. Congratulations, you've made a reusable control.

第160章：包含字体资源

参数 详情
fontbytes 从二进制 .ttf 文件的字节数组

第160.1节：从资源实例化 'Fontfamily'

```
public FontFamily Maneteke = GetResourceFontFamily(Properties.Resources.manteka);
```



第160.2节：集成方法

```
public static FontFamily GetResourceFontFamily(byte[] fontbytes)
{
    PrivateFontCollection pfc = new PrivateFontCollection();
    IntPtr fontMemPointer = Marshal.AllocCoTaskMem(fontbytes.Length);
    Marshal.Copy(fontbytes, 0, fontMemPointer, fontbytes.Length);
    pfc.AddMemoryFont(fontMemPointer, fontbytes.Length);
    Marshal.FreeCoTaskMem(fontMemPointer);
    return pfc.Families[0];
}
```

第160.3节：与“按钮”的使用

```
public static class Res
{
    /// <summary>
    /// URL: https://www.behance.net/gallery/2846011/Manteka
    /// </summary>
    public static FontFamily Maneteke = GetResourceFontFamily(Properties.Resources.manteka);

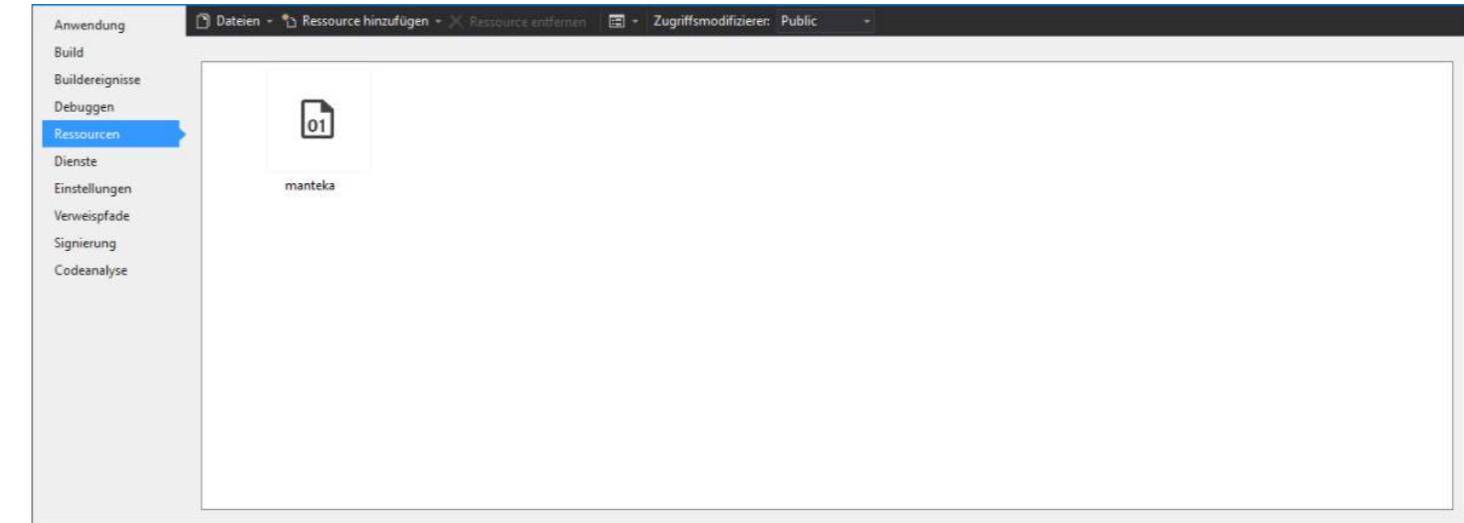
    public static FontFamily GetResourceFontFamily(byte[] fontbytes)
    {
        PrivateFontCollection pfc = new PrivateFontCollection();
        IntPtr fontMemPointer = Marshal.AllocCoTaskMem(fontbytes.Length);
        Marshal.Copy(fontbytes, 0, fontMemPointer, fontbytes.Length);
        pfc.AddMemoryFont(fontMemPointer, fontbytes.Length);
        Marshal.FreeCoTaskMem(fontMemPointer);
        return pfc.Families[0];
    }
}
```

Chapter 160: Including Font Resources

Parameter Details
fontbytes byte array from the binary .ttf

Section 160.1: Instantiate 'Fontfamily' from Resources

```
public FontFamily Maneteke = GetResourceFontFamily(Properties.Resources.manteka);
```



Section 160.2: Integration method

```
public static FontFamily GetResourceFontFamily(byte[] fontbytes)
{
    PrivateFontCollection pfc = new PrivateFontCollection();
    IntPtr fontMemPointer = Marshal.AllocCoTaskMem(fontbytes.Length);
    Marshal.Copy(fontbytes, 0, fontMemPointer, fontbytes.Length);
    pfc.AddMemoryFont(fontMemPointer, fontbytes.Length);
    Marshal.FreeCoTaskMem(fontMemPointer);
    return pfc.Families[0];
}
```

Section 160.3: Usage with a 'Button'

```
public static class Res
{
    /// <summary>
    /// URL: https://www.behance.net/gallery/2846011/Manteka
    /// </summary>
    public static FontFamily Maneteke = GetResourceFontFamily(Properties.Resources.manteka);

    public static FontFamily GetResourceFontFamily(byte[] fontbytes)
    {
        PrivateFontCollection pfc = new PrivateFontCollection();
        IntPtr fontMemPointer = Marshal.AllocCoTaskMem(fontbytes.Length);
        Marshal.Copy(fontbytes, 0, fontMemPointer, fontbytes.Length);
        pfc.AddMemoryFont(fontMemPointer, fontbytes.Length);
        Marshal.FreeCoTaskMem(fontMemPointer);
        return pfc.Families[0];
    }
}
```

```
public class FlatButton : Button
{
    public FlatButton() : base()
    {
        Font = new Font(Res.Maneteke, Font.Size);
    }

    protected override void OnFontChanged(EventArgs e)
    {
        base.OnFontChanged(e);
        this.Font = new Font(Res.Maneteke, this.Font.Size);
    }
}
```

```
public class FlatButton : Button
{
    public FlatButton() : base()
    {
        Font = new Font(Res.Maneteke, Font.Size);
    }

    protected override void OnFontChanged(EventArgs e)
    {
        base.OnFontChanged(e);
        this.Font = new Font(Res.Maneteke, this.Font.Size);
    }
}
```

belindoc.com

第161章：导入谷歌联系人

第161.1节：需求

要在ASP.NET MVC应用程序中导入Google (Gmail) 联系人，首先下载“Google API设置”这将授予以下引用：

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
```

将这些添加到相关的应用程序中。

第161.2节：控制器中的源代码

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net;
using System.Text;
using System.Web;
using System.Web.Mvc;

namespace GoogleContactImport.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult Import()
        {
            string clientId = ""; // 这里需要添加你的 Google 客户端 ID
            string redirectUrl = "http://localhost:1713/Home/AddGoogleContacts"; // 这里是你的
            重定向操作方法 注意：你需要在 Google 控制台配置相同的 URL
            Response.Redirect("https://accounts.google.com/o/oauth2/auth?redirect_uri=" +
                redirectUrl + "&response_type=code&&client_id=" + clientId +
                "&scope=https://www.google.com/m8/feeds/&approval_prompt=force&&access_type=offline");
            return View();
        }

        public ActionResult AddGoogleContacts()
        {
            string code = Request.QueryString["code"];
            if (!string.IsNullOrEmpty(code))
            {
                var contacts = GetAccessToken().ToArray();
                if (contacts.Length > 0)
                {

```

Chapter 161: Import Google Contacts

Section 161.1: Requirements

To Import Google(Gmail) contacts in ASP.NET MVC application, first [download "Google API setup"](#) This will grant the following references:

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
```

Add these to the relevant application.

Section 161.2: Source code in the controller

```
using Google.Contacts;
using Google.GData.Client;
using Google.GData.Contacts;
using Google.GData.Extensions;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net;
using System.Text;
using System.Web;
using System.Web.Mvc;

namespace GoogleContactImport.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult Import()
        {
            string clientId = ""; // here you need to add your google client id
            string redirectUrl = "http://localhost:1713/Home/AddGoogleContacts"; // here your
            redirect action method NOTE: you need to configure same url in google console
            Response.Redirect("https://accounts.google.com/o/oauth2/auth?redirect_uri=" +
                redirectUrl + "&response_type=code&&client_id=" + clientId +
                "&scope=https://www.google.com/m8/feeds/&approval_prompt=force&&access_type=offline");
            return View();
        }

        public ActionResult AddGoogleContacts()
        {
            string code = Request.QueryString["code"];
            if (!string.IsNullOrEmpty(code))
            {
                var contacts = GetAccessToken().ToArray();
                if (contacts.Length > 0)
                {

```

```

    // 你将在这里获取所有联系人
    return View("Index", contacts);
}
else
{
    return RedirectToAction("Index", "Home");
}
else
{
    return RedirectToAction("Index", "Home");
}
}

public List<GmailContacts> GetAccessToken()
{
    string code = Request.QueryString["code"];
    string google_client_id = ""; //你的 Google 客户端 ID
    string google_client_sceret = ""; //你的 Google 密钥
    string google_redirect_url = "http://localhost:1713/MyContact/AddGoogleContacts";

HttpWebRequest webRequest =
(HttpWebRequest)WebRequest.Create("https://accounts.google.com/o/oauth2/token");
    webRequest.Method = "POST";
    string parameters = "code=" + code + "&client_id=" + google_client_id +
"&client_secret=" + google_client_sceret + "&redirect_uri=" + google_redirect_url +
"&grant_type=authorization_code";
    byte[] byteArray = Encoding.UTF8.GetBytes(parameters);
    webRequest.ContentType = "application/x-www-form-urlencoded";
    webRequest.ContentLength = byteArray.Length;
Stream postStream = webRequest.GetRequestStream();
    // 将 POST 数据添加到 Web 请求中
postStream.Write(byteArray, 0, byteArray.Length);
    postStream.Close();
WebResponse response = webRequest.GetResponse();
    postStream = response.GetResponseStream();
StreamReader reader = new StreamReader(postStream);
    string responseFromServer = reader.ReadToEnd();
    GooglePlusAccessToken serStatus =
JsonConvert.DeserializeObject<GooglePlusAccessToken>(responseFromServer);
/*End*/
    return GetContacts(serStatus);
}

public List<GmailContacts> GetContacts(GooglePlusAccessToken serStatus)
{
    string google_client_id = ""; //客户端ID
    string google_client_sceret = ""; //密钥
    /*从访问令牌和刷新令牌获取谷歌联系人*/
    // string refreshToken = serStatus.refresh_token;
    string accessToken = serStatus.access_token;
    string scopes = "https://www.googleapis.com/auth/contacts.readonly";
    OAuth2Parameters oAuthparameters = new OAuth2Parameters()
    {
ClientId = google_client_id,
        ClientSecret = google_client_sceret,
        RedirectUri = "http://localhost:1713/Home/AddGoogleContacts",
        Scope = scopes,
AccessToken = accessToken,
        // RefreshToken = refreshToken
    };
}

```

```

    // You will get all contacts here
    return View("Index", contacts);
}
else
{
    return RedirectToAction("Index", "Home");
}
else
{
    return RedirectToAction("Index", "Home");
}
}

public List<GmailContacts> GetAccessToken()
{
    string code = Request.QueryString["code"];
    string google_client_id = ""; //your google client Id
    string google_client_sceret = ""; // your google secret key
    string google_redirect_url = "http://localhost:1713/MyContact/AddGoogleContacts";

HttpWebRequest webRequest =
(HttpWebRequest)WebRequest.Create("https://accounts.google.com/o/oauth2/token");
    webRequest.Method = "POST";
    string parameters = "code=" + code + "&client_id=" + google_client_id +
"&client_secret=" + google_client_sceret + "&redirect_uri=" + google_redirect_url +
"&grant_type=authorization_code";
    byte[] byteArray = Encoding.UTF8.GetBytes(parameters);
    webRequest.ContentType = "application/x-www-form-urlencoded";
    webRequest.ContentLength = byteArray.Length;
Stream postStream = webRequest.GetRequestStream();
    // Add the post data to the web request
postStream.Write(byteArray, 0, byteArray.Length);
    postStream.Close();
WebResponse response = webRequest.GetResponse();
    postStream = response.GetResponseStream();
StreamReader reader = new StreamReader(postStream);
    string responseFromServer = reader.ReadToEnd();
    GooglePlusAccessToken serStatus =
JsonConvert.DeserializeObject<GooglePlusAccessToken>(responseFromServer);
/*End*/
    return GetContacts(serStatus);
}

public List<GmailContacts> GetContacts(GooglePlusAccessToken serStatus)
{
    string google_client_id = ""; //client id
    string google_client_sceret = ""; //secret key
    /*Get Google Contacts From Access Token and Refresh Token*/
    // string refreshToken = serStatus.refresh_token;
    string accessToken = serStatus.access_token;
    string scopes = "https://www.googleapis.com/auth/contacts.readonly";
    OAuth2Parameters oAuthparameters = new OAuth2Parameters()
    {
        ClientId = google_client_id,
        ClientSecret = google_client_sceret,
        RedirectUri = "http://localhost:1713/Home/AddGoogleContacts",
        Scope = scopes,
        AccessToken = accessToken,
        // RefreshToken = refreshToken
    };
}

```

```

RequestSettings settings = new RequestSettings("App Name", oAuthparameters);
    ContactsRequest cr = new ContactsRequest(settings);
ContactsQuery query = new ContactsQuery(ContactsQuery.CreateContactsUri("default"));
    query.NumberToRetrieve = 5000;
Feed<Contact> ContactList = cr.GetContacts();

    List<GmailContacts> olist = new List<GmailContacts>();
    foreach (Contact contact in ContactList.Entries)
    {
        foreach (EMail email in contact.Emails)
        {
            GmailContacts gc = new GmailContacts();
            gc.EmailID = email.Address;
            var a = contact.Name.FullName;
            olist.Add(gc);
        }
    }
    return olist;
}

public class GmailContacts
{
    public string EmailID
    {
        get { return _EmailID; }
        set { _EmailID = value; }
    }
    private string _EmailID;
}

public class GooglePlusAccessToken
{

    public GooglePlusAccessToken()
    {

        public string access_token
        {
            get { return _access_token; }
            set { _access_token = value; }
        }
        private string _access_token;

        public string token_type
        {
            get { return _token_type; }
            set { _token_type = value; }
        }
        private string _token_type;

        public string expires_in
        {
            get { return _expires_in; }
            set { _expires_in = value; }
        }
        private string _expires_in;
    }
}

```

```

RequestSettings settings = new RequestSettings("App Name", oAuthparameters);
    ContactsRequest cr = new ContactsRequest(settings);
ContactsQuery query = new ContactsQuery(ContactsQuery.CreateContactsUri("default"));
    query.NumberToRetrieve = 5000;
Feed<Contact> ContactList = cr.GetContacts();

    List<GmailContacts> olist = new List<GmailContacts>();
    foreach (Contact contact in ContactList.Entries)
    {
        foreach (EMail email in contact.Emails)
        {
            GmailContacts gc = new GmailContacts();
            gc.EmailID = email.Address;
            var a = contact.Name.FullName;
            olist.Add(gc);
        }
    }
    return olist;
}

public class GmailContacts
{
    public string EmailID
    {
        get { return _EmailID; }
        set { _EmailID = value; }
    }
    private string _EmailID;
}

public class GooglePlusAccessToken
{

    public GooglePlusAccessToken()
    {

        public string access_token
        {
            get { return _access_token; }
            set { _access_token = value; }
        }
        private string _access_token;

        public string token_type
        {
            get { return _token_type; }
            set { _token_type = value; }
        }
        private string _token_type;

        public string expires_in
        {
            get { return _expires_in; }
            set { _expires_in = value; }
        }
        private string _expires_in;
    }
}

```

第161.3节：视图中的源代码

您需要添加的唯一操作方法是添加下面的操作链接

```
<a href="@Url.Action("Import", "Home")">导入谷歌联系人</a>
```

belindoc.com

Section 161.3: Source code in the view

The only action method you need to add is to add an action link present below

```
<a href="@Url.Action("Import", "Home")">Import Google Contacts</a>
```

第162章：.Net中的垃圾回收器

第162.1节：弱引用

在.NET中，当对象没有任何引用时，垃圾回收器（GC）会回收这些对象。因此，只要代码仍然可以访问该对象（存在强引用），GC就不会回收该对象。如果存在大量大型对象，这可能会成为问题。

弱引用是一种引用，它允许GC回收对象，同时仍然可以访问该对象。弱引用仅在没有强引用存在时、对象被回收之前的这段不确定时间内有效。当您使用弱引用时，应用程序仍然可以获得该对象的强引用，从而防止其被回收。因此，弱引用对于持有那些初始化代价高昂但如果被动使用时应允许垃圾回收的大型对象非常有用。

简单用法：

```
WeakReference 引用 = new WeakReference(new object(), false);  
GC.Collect();  
  
object 目标 = 引用.Target;  
if (目标 != null)  
    执行某操作(目标);
```

因此，弱引用可以用来维护例如对象缓存。然而，重要的是要记住，总是存在垃圾回收器在强引用重新建立之前回收对象的风险。

弱引用对于避免内存泄漏也很有用。一个典型的使用场景是事件处理。

假设我们有一个事件源的事件处理程序：

```
源.事件 += new EventHandler(处理程序)
```

这段代码注册了一个事件处理程序，并从事件源创建了一个指向监听对象的强引用。如果源对象的生命周期比监听器长，并且当监听器不再需要事件且没有其他引用时，使用普通的.NET事件会导致内存泄漏：源对象会在内存中持有本应被垃圾回收的监听器对象。

在这种情况下，使用弱事件模式可能是个好主意。

类似于：

```
public static class WeakEventManager  
{  
    public static void SetHandler<S, TArgs>(  
        Action<EventHandler<TArgs>> add,  
        Action<EventHandler<TArgs>> remove,  
        S subscriber,  
        Action<S, TArgs> action)  
    where TArgs : EventArgs  
    where S : class  
    {  
        var subscrWeakRef = new WeakReference(subscriber);  
        EventHandler<TArgs> handler = null;
```

Chapter 162: Garbage Collector in .Net

Section 162.1: Weak References

In .NET, the GC allocates objects when there are no references left to them. Therefore, while an object can still be reached from code (there is a strong reference to it), the GC will not allocate this object. This can become a problem if there are a lot of large objects.

A weak reference is a reference, that allows the GC to collect the object while still allowing to access the object. A weak reference is valid only during the indeterminate amount of time until the object is collected when no strong references exist. When you use a weak reference, the application can still obtain a strong reference to the object, which prevents it from being collected. So weak references can be useful for holding on to large objects that are expensive to initialize, but should be available for garbage collection if they are not actively in use.

Simple usage:

```
WeakReference reference = new WeakReference(new object(), false);  
GC.Collect();  
  
object target = reference.Target;  
if (target != null)  
    DoSomething(target);
```

So weak references could be used to maintain, for example, a cache of objects. However, it is important to remember that there is always the risk that the garbage collector will get to the object before a strong reference is reestablished.

Weak references are also handy for avoiding memory leaks. A typical use case is with events.

Suppose we have some handler to an event on a source:

```
Source.Event += new EventHandler(Handler)
```

This code registers an event handler and creates a strong reference from the event source to the listening object. If the source object has a longer lifetime than the listener, and the listener doesn't need the event anymore when there are no other references to it, using normal .NET events causes a memory leak: the source object holds listener objects in memory that should be garbage collected.

In this case, it may be a good idea to use the [Weak Event Pattern](#).

Something like:

```
public static class WeakEventManager  
{  
    public static void SetHandler<S, TArgs>(  
        Action<EventHandler<TArgs>> add,  
        Action<EventHandler<TArgs>> remove,  
        S subscriber,  
        Action<S, TArgs> action)  
    where TArgs : EventArgs  
    where S : class  
    {  
        var subscrWeakRef = new WeakReference(subscriber);  
        EventHandler<TArgs> handler = null;
```

```

handler = (s, e) =>
{
    var subscrStrongRef = subscrWeakRef.Target as S;
    if (subscrStrongRef != null)
    {
        action(subscrStrongRef, e);
    }
    else
    {
        remove(handler);
        handler = null;
    }
};

add(handler);
}

```

并且这样使用：

```

EventSource s = new EventSource();
Subscriber subscriber = new Subscriber();
WeakEventManager.SetHandler<Subscriber, SomeEventArgs>(a => s.Event += a, r => s.Event -= r,
subscriber, (s,e) => { s.HandleEvent(e); });

```

在这种情况下，当然我们有一些限制——事件必须是一个

```
public event EventHandler<SomeEventArgs> Event;
```

正如[MSDN](#)所建议的：

- 仅在必要时使用长弱引用，因为对象在终结后状态是不可预测的。
- 避免对小对象使用弱引用，因为指针本身可能与对象大小相当或更大。
- 避免将弱引用作为内存管理问题的自动解决方案。相反，应制定有效的缓存策略来处理应用程序的对象。

第162.2节：大对象堆压缩

默认情况下，大对象堆不会被压缩，不同于经典对象堆，这可能导致内存碎片化，并且进一步[可能导致OutOfMemoryException](#)异常

从 .NET 4.5.1 开始，有一个[选项](#)可以显式压缩大型对象堆（以及进行垃圾回收）：

```
GCSettings.LargeObjectHeapCompactionMode = GCLargeObjectHeapCompactionMode.CompactOnce;
GC.Collect();
```

正如任何显式的垃圾回收请求（称为请求是因为 CLR 并不强制执行它）一样，请谨慎使用，默认情况下如果可以，尽量避免使用，因为它可能会使GC的统计数据失准，从而降低其性能。

```

handler = (s, e) =>
{
    var subscrStrongRef = subscrWeakRef.Target as S;
    if (subscrStrongRef != null)
    {
        action(subscrStrongRef, e);
    }
    else
    {
        remove(handler);
        handler = null;
    }
};

add(handler);
}

```

and used like this:

```

EventSource s = new EventSource();
Subscriber subscriber = new Subscriber();
WeakEventManager.SetHandler<Subscriber, SomeEventArgs>(a => s.Event += a, r => s.Event -= r,
subscriber, (s,e) => { s.HandleEvent(e); });

```

In this case of course we have some restrictions - the event must be a

```
public event EventHandler<SomeEventArgs> Event;
```

As [MSDN](#) suggests:

- Use long weak references only when necessary as the state of the object is unpredictable after finalization.
- Avoid using weak references to small objects because the pointer itself may be as large or larger.
- Avoid using weak references as an automatic solution to memory management problems. Instead, develop an effective caching policy for handling your application's objects.

Section 162.2: Large Object Heap compaction

By default the Large Object Heap is not compacted unlike the classic Object Heap which [can lead to memory fragmentation](#) and further, can lead to [OutOfMemoryExceptions](#)

Starting with .NET 4.5.1 there is [an option](#) to explicitly compact the Large Object Heap (along with a garbage collection):

```
GCSettings.LargeObjectHeapCompactionMode = GCLargeObjectHeapCompactionMode.CompactOnce;
GC.Collect();
```

Just as any explicit garbage collection request (it's called request because the CLR is not forced to conduct it) use with care and by default avoid it if you can since it can de-calibrate GCs statistics, decreasing its performance.

第163章： Microsoft.Exchange.WebServices

第163.1节：检索指定用户的外出设置

首先让我们创建一个ExchangeManager对象，其构造函数将为我们连接到服务。它还有一个GetOofSettings方法，该方法将返回指定电子邮件地址的OofSettings对象：

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;

namespace SetOutOfOffice
{
    class ExchangeManager
    {
        private ExchangeService Service;

        public ExchangeManager()
        {
            var password = WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
            Connect("exchangeadmin", password);
        }

        private void Connect(string username, string password)
        {
            var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
            service.Credentials = new WebCredentials(username, password);
            service.AutodiscoverUrl("autodiscoveremail@domain.com",
RedirectionUrlValidationCallback);

            Service = service;
        }

        private static bool RedirectionUrlValidationCallback(string redirectionUrl)
        {
            return redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
        }

        public OofSettings GetOofSettings(string email)
        {
            return Service.GetUserOofSettings(email);
        }
    }
}
```

我们现在可以在其他地方这样调用：

```
var em = new ExchangeManager();
var oofSettings = em.GetOofSettings("testemail@domain.com");
```

第163.2节：更新特定用户的外出设置

使用以下类，我们可以连接到Exchange，然后使用UpdateUserOof：

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;
```

Chapter 163: Microsoft.Exchange.WebServices

Section 163.1: Retrieve Specified User's Out of Office Settings

First let's create an ExchangeManager object, where the constructor will connect to the services for us. It also has a GetOofSettings method, which will return the OofSettings object for the specified email address :

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;

namespace SetOutOfOffice
{
    class ExchangeManager
    {
        private ExchangeService Service;

        public ExchangeManager()
        {
            var password = WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
            Connect("exchangeadmin", password);
        }

        private void Connect(string username, string password)
        {
            var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
            service.Credentials = new WebCredentials(username, password);
            service.AutodiscoverUrl("autodiscoveremail@domain.com",
RedirectionUrlValidationCallback);

            Service = service;
        }

        private static bool RedirectionUrlValidationCallback(string redirectionUrl)
        {
            return redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
        }

        public OofSettings GetOofSettings(string email)
        {
            return Service.GetUserOofSettings(email);
        }
    }
}
```

We can now call this elsewhere like this:

```
var em = new ExchangeManager();
var oofSettings = em.GetOofSettings("testemail@domain.com");
```

Section 163.2: Update Specific User's Out of Office Settings

Using the class below, we can connect to Exchange and then set a specific user's out of office settings with UpdateUserOof:

```
using System;
using System.Web.Configuration;
using Microsoft.Exchange.WebServices.Data;
```

```

class ExchangeManager
{
    private ExchangeService Service;

    public ExchangeManager()
    {
        var password = WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
        Connect("exchangeadmin", password);
    }
    private void Connect(string username, string password)
    {
        var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
        service.Credentials = new WebCredentials(username, password);
        service.AutodiscoverUrl("autodiscoveremail@domain.com", RedirectionUrlValidationCallback);

        Service = service;
    }
    private static bool RedirectionUrlValidationCallback(string redirectionUrl)
    {
        return redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
    }
    /// <summary>
    /// 使用给定的详细信息更新指定用户的外出设置
    /// </summary>
    public void UpdateUserOof(int oofstate, DateTime starttime, DateTime endtime, int
externalaudience, string internalmsg, string externalmsg, string emailaddress)
    {
        var newSettings = new OofSettings
        {
            状态 = (OofState)oofstate,
            持续时间 = new TimeWindow(starttime, endtime),
            外部受众 = (OfExternalAudience)externalaudience,
            内部回复 = internalmsg,
            外部回复 = externalmsg
        };

        Service.SetUserOofSettings(emailaddress, newSettings);
    }
}

```

使用以下内容更新用户设置：

```

var oofState = 1;
var startDate = new DateTime(01,08,2016);
var endDate = new DateTime(15,08,2016);
var externalAudience = 1;
var internalMessage = "我不在办公室！";
var externalMessage = "我不在办公室 <strong>你也不在！</strong>";
var theUser = "theuser@domain.com";

var em = new ExchangeManager();
em.UpdateUserOof(oofstate, startDate, endDate, externalAudience, internalMessage, externalMessage,
theUser);

```

注意，你可以使用标准的 html 标签来格式化消息。

```

class ExchangeManager
{
    private ExchangeService Service;

    public ExchangeManager()
    {
        var password = WebConfigurationManager.ConnectionStrings["Password"].ConnectionString;
        Connect("exchangeadmin", password);
    }
    private void Connect(string username, string password)
    {
        var service = new ExchangeService(ExchangeVersion.Exchange2010_SP2);
        service.Credentials = new WebCredentials(username, password);
        service.AutodiscoverUrl("autodiscoveremail@domain.com", RedirectionUrlValidationCallback);

        Service = service;
    }
    private static bool RedirectionUrlValidationCallback(string redirectionUrl)
    {
        return redirectionUrl.Equals("https://mail.domain.com/autodiscover/autodiscover.xml");
    }
    /// <summary>
    /// Updates the given user's Oof settings with the given details
    /// </summary>
    public void UpdateUserOof(int oofstate, DateTime starttime, DateTime endtime, int
externalaudience, string internalmsg, string externalmsg, string emailaddress)
    {
        var newSettings = new OofSettings
        {
            State = (OofState)oofstate,
            Duration = new TimeWindow(starttime, endtime),
            ExternalAudience = (OfExternalAudience)externalaudience,
            InternalReply = internalmsg,
            ExternalReply = externalmsg
        };

        Service.SetUserOofSettings(emailaddress, newSettings);
    }
}

```

Update the user settings with the following:

```

var oofState = 1;
var startDate = new DateTime(01,08,2016);
var endDate = new DateTime(15,08,2016);
var externalAudience = 1;
var internalMessage = "I am not in the office!";
var externalMessage = "I am not in the office <strong>and neither are you!</strong>";
var theUser = "theuser@domain.com";

var em = new ExchangeManager();
em.UpdateUserOof(oofstate, startDate, endDate, externalAudience, internalMessage, externalMessage,
theUser);

```

Note that you can format the messages using standard html tags.

第164章：Windows通信基础设施

Windows通信基础设施（WCF）是一个用于构建面向服务应用程序的框架。使用WCF，您可以将数据作为异步消息从一个服务端点发送到另一个服务端点。服务端点可以是由IIS托管的持续可用服务的一部分，也可以是托管在应用程序中的服务。消息可以简单到作为XML发送的单个字符或单词，也可以复杂到二进制数据流。

第164.1节：入门示例

服务在其公开为元数据的服务契约中描述其执行的操作。

```
// 定义服务契约。  
[ServiceContract(Namespace="http://StackOverflow.ServiceModel.Samples")]  
public interface ICalculator  
{  
    [OperationContract]  
    double Add(double n1, double n2);  
}
```

服务实现计算并返回相应结果，如以下示例代码所示。

```
// 实现服务契约的服务类。  
public class CalculatorService : ICalculator  
{  
    public double Add(double n1, double n2)  
    {  
        return n1 + n2;  
    }  
}
```

该服务通过一个端点与外界通信，该端点使用配置文件（Web.config）定义，如下示例配置所示。

```
<services>  
    <service  
        name="StackOverflow.ServiceModel.Samples.CalculatorService"  
        behaviorConfiguration="CalculatorServiceBehavior">  
        <!-- ICalculator 在主机提供的基础地址上公开：  
        http://localhost/servicemodelsamples/service.svc。 -->  
        <endpoint address=""  
            binding="wsHttpBinding"  
            contract="StackOverflow.ServiceModel.Samples.ICalculator" />  
        ...  
    </service>  
</services>
```

框架默认不公开元数据。因此，服务启用了 ServiceMetadataBehavior 并在 <http://localhost/servicemodelsamples/service.svc/mex> 公开了一个元数据交换（MEX）端点。以下配置演示了这一点。

```
<system.serviceModel>  
    <services>  
        <service  
            name="StackOverflow.ServiceModel.Samples.CalculatorService"
```

Chapter 164: Windows Communication Foundation

Windows Communication Foundation (WCF) is a framework for building service-oriented applications. Using WCF, you can send data as asynchronous messages from one service endpoint to another. A service endpoint can be part of a continuously available service hosted by IIS, or it can be a service hosted in an application. The messages can be as simple as a single character or word sent as XML, or as complex as a stream of binary data.

Section 164.1: Getting started sample

The service describes the operations it performs in a service contract that it exposes publicly as metadata.

```
// Define a service contract.  
[ServiceContract(Namespace="http://StackOverflow.ServiceModel.Samples")]  
public interface ICalculator  
{  
    [OperationContract]  
    double Add(double n1, double n2);  
}
```

The service implementation calculates and returns the appropriate result, as shown in the following example code.

```
// Service class that implements the service contract.  
public class CalculatorService : ICalculator  
{  
    public double Add(double n1, double n2)  
    {  
        return n1 + n2;  
    }  
}
```

The service exposes an endpoint for communicating with the service, defined using a configuration file (Web.config), as shown in the following sample configuration.

```
<services>  
    <service  
        name="StackOverflow.ServiceModel.Samples.CalculatorService"  
        behaviorConfiguration="CalculatorServiceBehavior">  
        <!-- ICalculator is exposed at the base address provided by  
        host: http://localhost/servicemodelsamples/service.svc. -->  
        <endpoint address=""  
            binding="wsHttpBinding"  
            contract="StackOverflow.ServiceModel.Samples.ICalculator" />  
        ...  
    </service>  
</services>
```

The framework does not expose metadata by default. As such, the service turns on the ServiceMetadataBehavior and exposes a metadata exchange (MEX) endpoint at <http://localhost/servicemodelsamples/service.svc/mex>. The following configuration demonstrates this.

```
<system.serviceModel>  
    <services>  
        <service  
            name="StackOverflow.ServiceModel.Samples.CalculatorService"
```

```

behaviorConfiguration="CalculatorServiceBehavior">
...
<!-- mex 端点公开于
http://localhost/servicemodelsamples/service.svc/mex -->
<endpoint address="mex"
binding="mexHttpBinding"
contract="IMetadataExchange" />
</service>
</services>

<!--调试时将 includeExceptionDetailInFaults
属性设置为 true-->
<behaviors>
<serviceBehaviors>
<behavior name="CalculatorServiceBehavior">
<serviceMetadata httpGetEnabled="True"/>
<serviceDebug includeExceptionDetailInFaults="False" />
</behavior>
</serviceBehaviors>
</behaviors>
</system.serviceModel>

```

客户端通过使用由 ServiceModel 元数据工具 (Svccutil.exe) 生成的客户端类，使用指定的契约类型进行通信。

在客户端目录的 SDK 命令提示符下运行以下命令以生成类型化代理：

```

svccutil.exe /n:"http://StackOverflow.ServiceModel.Samples,StackOverflow.ServiceModel.Samples"
http://localhost/servicemodelsamples/service.svc/mex /out:generatedClient.cs

```

与服务端类似，客户端使用配置文件 (App.config) 来指定其想要通信的终结点。客户端终结点配置包括服务终结点的绝对地址、绑定和契约，如下例所示。

```

<client>
<endpoint
address="http://localhost/servicemodelsamples/service.svc"
binding="wsHttpBinding"
contract="StackOverflow.ServiceModel.Samples.ICalculator" />
</client>

```

客户端实现实例化客户端并使用类型化接口开始与服务通信，如下示例代码所示。

```

// 创建客户端。
CalculatorClient client = new CalculatorClient();

// 调用 Add 服务操作。
double value1 = 100.00D;
double value2 = 15.99D;
double result = client.Add(value1, value2);
Console.WriteLine("Add({0},{1}) = {2}", value1, value2, result);

//关闭客户端释放所有通信资源。
client.Close();

```

```

behaviorConfiguration="CalculatorServiceBehavior">
...
<!-- the mex endpoint is exposed at
http://localhost/servicemodelsamples/service.svc/mex -->
<endpoint address="mex"
binding="mexHttpBinding"
contract="IMetadataExchange" />
</service>
</services>

<!--For debugging purposes set the includeExceptionDetailInFaults
attribute to true-->
<behaviors>
<serviceBehaviors>
<behavior name="CalculatorServiceBehavior">
<serviceMetadata httpGetEnabled="True"/>
<serviceDebug includeExceptionDetailInFaults="False" />
</behavior>
</serviceBehaviors>
</behaviors>
</system.serviceModel>

```

The client communicates using a given contract type by using a client class that is generated by the ServiceModel Metadata Utility Tool (Svccutil.exe).

Run the following command from the SDK command prompt in the client directory to generate the typed proxy:

```

svccutil.exe /n:"http://StackOverflow.ServiceModel.Samples,StackOverflow.ServiceModel.Samples"
http://localhost/servicemodelsamples/service.svc/mex /out:generatedClient.cs

```

Like the service, the client uses a configuration file (App.config) to specify the endpoint with which it wants to communicate. The client endpoint configuration consists of an absolute address for the service endpoint, the binding, and the contract, as shown in the following example.

```

<client>
<endpoint
address="http://localhost/servicemodelsamples/service.svc"
binding="wsHttpBinding"
contract="StackOverflow.ServiceModel.Samples.ICalculator" />
</client>

```

The client implementation instantiates the client and uses the typed interface to begin communicating with the service, as shown in the following example code.

```

// Create a client.
CalculatorClient client = new CalculatorClient();

// Call the Add service operation.
double value1 = 100.00D;
double value2 = 15.99D;
double result = client.Add(value1, value2);
Console.WriteLine("Add({0},{1}) = {2}", value1, value2, result);

//Closing the client releases all communication resources.
client.Close();

```

鸣谢

非常感谢所有来自Stack Overflow Documentation的人员帮助提供此内容，
更多更改可发送至web@petercv.com以发布或更新新内容

Dan	第48章
4444	第156章
亚伦·阿诺迪德	第72章
亚伦·哈登	第20、22、27、48、89、96、114和120章
aashishkoirala	第7章
阿巴斯·加利亚科特瓦拉	第139章
阿卜杜勒·雷赫曼·赛义德	第13、22、31和76章
阿卜杜勒拉赫曼·安萨里	第19章
abishekshivan	第1章
Abob	第73章
abto	第94章
亚当	第96、111和124章
亚当·克利福德	第66章
亚当·霍尔兹沃思	第3章和第60章
亚当·西尔斯	第39章
阿德·斯特林格	第1章、第52章、第66章和第105章
阿迪·莱斯特	第39章、第43章、第48章、第52章、第66章、第68章、第89章、第98章、第148章和第149章
阿迪尔·马马多夫	第3章、第48章、第66章、第71章和第72章
阿迪提亚·科尔蒂	第52章
阿德里亚诺·雷佩蒂	第39章、第52章和第72章
AFT	第21章
AGB	第59章和第72章
艾哈迈德·詹·艾德米尔	第149章
Al.	第52章
阿克谢·阿南德	第3章和第81章
艾伦·麦克比	第3章和第10章
亚历克斯·安德烈耶夫	第83章和第157章
阿列克谢·L.	第66章
亚历克斯	第127章
亚历克斯·维泽	第60章
亚历山大·曼特	第4章、第56章、第95章和第149章
亚历山大·帕查	第52章
阿列克谢·科普佳耶夫	第66章
阿列克谢·格罗舍夫	第20章和第94章
阿利亚克谢·富特林	第130章
阿利森	第28章
阿洛克·辛格	第1章
阿尔法	第8章
阿曼·沙尔马	第63章
阿梅亚·德什潘德	第4章和第144章
阿米尔·普尔曼德	第31章
پورمند	第20章
阿纳斯·塔萨杜克	第48章、第66章和第72章
阿那克西曼德	第46章和第93章
安德烈	第122章
安德烈·埃普雷	第28章、第39章、第43章、第52章和第162章
安德烈·里内亚	

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

Dan	Chapter 48
4444	Chapter 156
Aaron Anodide	Chapter 72
Aaron Hudon	Chapters 20, 22, 27, 48, 89, 96, 114 and 120
aashishkoirala	Chapter 7
Abbas Galiyakotwala	Chapter 139
Abdul Rehman Sayed	Chapters 13, 22, 31 and 76
AbdulRahman Ansari	Chapter 19
abishekshivan	Chapter 1
Abob	Chapter 73
abto	Chapter 94
Adam	Chapters 96, 111 and 124
Adam Clifford	Chapter 66
Adam Houldsworth	Chapters 3 and 60
Adam Sills	Chapter 39
Ade Stringer	Chapters 1, 52, 66 and 105
Adi Lester	Chapters 39, 43, 48, 52, 66, 68, 89, 98, 148 and 149
Adil Mammadov	Chapters 3, 48, 66, 71 and 72
Aditya Korti	Chapter 52
Adriano Repetti	Chapters 39, 52 and 72
AFT	Chapter 21
AGB	Chapters 59 and 72
Ahmet Can Aydemir	Chapter 149
Al.	Chapter 52
Akshay Anand	Chapters 3 and 81
Alan McBee	Chapters 3 and 10
Aleks Andreev	Chapters 83 and 157
Aleksey L.	Chapter 66
alex	Chapter 127
Alex Wiese	Chapter 60
Alexander Mandt	Chapters 4, 56, 95 and 149
Alexander Pacha	Chapter 52
Alexey Koptyaev	Chapter 66
Alexey Groshev	Chapters 20 and 94
Aliaksei Futryn	Chapter 130
Alisson	Chapter 28
Alok Singh	Chapter 1
Alpha	Chapter 8
Aman Sharma	Chapter 63
Ameya Deshpande	Chapters 4 and 144
Amir Pourmand	Chapter 31
پورمند	Chapter 20
Anas Tasadduq	Chapters 48, 66 and 72
anaximander	Chapters 46 and 93
Andrei	Chapter 122
Andrei Epure	Chapters 28, 39, 43, 52 and 162
Andrei Rînea	

安德鲁·戴蒙德	第39、71、76和95章	Andrew Diamond	Chapters 39, 71, 76 and 95
安德鲁·皮利塞尔	第152章	Andrew Piliser	Chapter 152
安德鲁·斯托拉克	第138章	Andrew Stollak	Chapter 138
andre_ss6	第58、59和60章	andre_ss6	Chapters 58, 59 and 60
安德里乌斯	第12章	Andrius	Chapter 12
安吉拉	第52章	Angela	Chapter 52
安基特·拉纳	第7章	Ankit Rana	Chapter 7
安基特·维贾伊	第66章	Ankit Vijay	Chapter 66
安娜	第52章	Anna	Chapter 52
蚂蚁_P	第114章	Ant P	Chapter 114
远日点	第26章和第48章	Aphelion	Chapters 26 and 48
阿拉文德·苏雷什	第1章	Aravind Suresh	Chapter 1
战神	第72章	Ares	Chapter 72
阿尔扬·恩布	第10章和第15章	Arjan Einbu	Chapters 10 and 15
亚瑟·里佐	第72章	Arthur Rizzo	Chapter 72
阿西姆·高塔姆	第12章、第63章和第76章	Aseem Gautam	Chapters 12, 63 and 76
ASH	第26章	ASH	Chapter 26
ATechieThought	第48章	ATechieThought	Chapter 48
Athafoud	第76章	Athafoud	Chapter 76
Athari		Athari	Chapters 3, 16, 26, 48, 52, 57, 71, 72, 74, 76, 89, 153 and 155
奥斯汀·T·弗伦奇		Austin T French	Chapter 112
阿维·特纳		Avi Turner	Chapters 3 and 71
阿维亚		Avia	Chapters 30, 39, 41 and 52
avs099		avs099	Chapter 15
阿克萨里达克斯		Axarydax	Chapter 63
A_阿诺德		A Arnold	Chapters 1, 52, 72 and 107
差		Bad	Chapter 57
巴伦·丹尼		Balen Danny	Chapter 155
BanksySan		BanksySan	Chapters 39 and 97
巴西		Bassie	Chapter 163
本·阿伦森		Ben Aaronson	Chapters 48, 55, 59, 72, 96 and 120
本·福格尔		Ben Fogel	Chapter 3
本·詹金森		Ben Jenkinson	Chapter 45
本杰明·霍奇森		Benjamin Hodgson	Chapters 38, 39, 59, 94 and 96
本乔尔		Benjol	Chapters 12, 39, 48, 58, 59, 60, 84 and 89
宾基		binki	Chapters 48 and 62
比约恩		Bjørn	Chapter 52
布拉赫什玛		Blachshma	Chapters 1, 11, 14, 22 and 97
布洛格比尔德		Blorgbeard	Chapter 87
布拉伯盖22		Blubberguy22	Chapter 3
bmadtiger		bmadtiger	Chapter 66
bob0the0mighty		bob0the0mighty	Chapter 75
BOBS		BOBS	Chapter 66
鲍布森		Bobson	Chapter 59
博金		Boggin	Chapters 122 and 144
鲍里斯·卡伦斯		Boris Callens	Chapter 60
博通德·巴拉兹		Botond Balázs	Chapters 16, 24, 28, 40, 54, 57, 86, 108, 122 and 141
Botz3000		Botz3000	Chapters 47, 52 and 94
博瓦兹		Bovaz	Chapters 23, 116 and 155
bpoiss		bpoiss	Chapter 72
布拉德利·格兰杰		Bradley Grainger	Chapter 60
布拉德利·乌夫纳		Bradley Uffner	Chapters 60, 72 and 96
布兰登	第117章	Brandon	Chapter 117

布雷迪	第38章	Braydie	Chapter 38
布伦丹·L	第15章	Brendan L	Chapter 15
brijber	第52章	brijber	Chapter 52
BrunoLM	第12章和第114章	BrunoLM	Chapters 12 and 114
布莱恩·克罗斯比	第10章和第62章	Bryan Crosby	Chapters 10 and 62
bumbeishvili	第66章	bumbeishvili	Chapter 66
BunkerMentality	第52章	BunkerMentality	Chapter 52
BurnsBA	第52章	BurnsBA	Chapter 52
C4u	第18章和第19章	C4u	Chapters 18 and 19
卡勒姆·沃特金斯	第13章、第48章、第55章、第72章和第86章	Callum Watkins	Chapters 13, 48, 55, 72 and 86
卡洛斯·穆尼奥斯	第39章	Carlos Muñoz	Chapter 39
卡明	第129章	Carmine	Chapter 129
卡斯滕	第59章	Carsten	Chapter 59
cbale	第66章	cbale	Chapter 66
查德	第9章	Chad	Chapter 9
查德·莱维	第1章和第72章	Chad Levy	Chapters 1 and 72
查德·麦格拉斯	第48章和第66章	Chad McGrath	Chapters 48 and 66
查里斯J	第11章	CharithJ	Chapter 11
查理·H	第72章	Charlie H	Chapter 72
查温	第23章	Chawin	Chapter 23
维格姆警长	第11章	Chief Wiggum	Chapter 11
克里斯	第52章	Chris	Chapter 52
克里斯·H.	第66章	Chris H.	Chapter 66
克里斯·杰斯特	第3章	Chris Jester	Chapter 3
克里斯·马里西奇	第60章	Chris Marisic	Chapter 60
克里斯·奥尔德伍德	第22章	Chris Oldwood	Chapter 22
克里斯托弗·柯伦斯	第119章	Christopher Currens	Chapter 119
克里斯托弗·罗宁	第1章	Christopher Ronning	Chapter 1
克里斯·伍	第60章和第62章	ChrisWue	Chapters 60 and 62
时间灭绝	第48章	Chronocide	Chapter 48
楚	第52章	Chuu	Chapter 52
西加诺·莫里森·门德斯	第63章	Cigano Morrison Mendez	Chapter 63
希汉·亚卡尔	第49章、第52章和第119章	Cihan Yakar	Chapters 49, 52 and 119
codeape	第115章	codeape	Chapter 115
CodeCaster	第26章	CodeCaster	Chapter 26
codekaizen	第152章	codekaizen	Chapter 152
CodeWarrior	第96章	CodeWarrior	Chapter 96
Configure	第1章	Configure	Chapter 1
coyote	第20章和第66章	coyote	Chapters 20 and 66
克雷格·布雷特	第31章和第114章	Craig Brett	Chapters 31 and 114
Crowcoder	第70章	Crowcoder	Chapter 70
cubrr	第16章和第48章	cubrr	Chapters 16 and 48
CyberFox	第26章和第41章	CyberFox	Chapters 26 and 41
cyberj0g	第11章	cyberj0g	Chapter 11
陈岱尔	第71章	Dale Chen	Chapter 71
丹·赫尔姆	第39章	Dan Hulme	Chapter 39
丹尼尔	第52章	Daniel	Chapter 52
丹尼尔·阿尔盖列斯	第66章	Daniel Argüelles	Chapter 66
丹尼尔·J.G.	第52章	Daniel J.G.	Chapter 52
丹尼尔·斯特拉多夫斯基	第72章	Daniel Stradowski	Chapter 72
丹尼·陈	第9章和第49章	Danny Chen	Chapters 9 and 49
丹尼·瓦罗德	第152章	Danny Varod	Chapter 152
达伦·戴维斯	第52章	Darren Davies	Chapter 52

达里尔
[dasblinkenlight](#)
[das_keyboard](#)
戴夫·扎伊奇
[Dave Zych](#)
大卫
[David](#)
大卫·派恩
[David Pine](#)
大卫·斯托金格
[David Stockinger](#)
[dav_i](#)
[DAXaholic](#)
[dazerdude](#)
[da_sann](#)
[dbmuller](#)
[dcastro](#)
[deadManN](#)
迪帕克·古普塔
[Deepak gupta](#)
删除我
[delete me](#)
[deloreyk](#)
[demonplus](#)
丹尼斯·埃尔霍夫
[Denis Elkhov](#)
[Dialecticus](#)
迪吉里杜管
[Didgeridoo](#)
老鼠
[die maus](#)
迪利普
[Dileep](#)
勤奋的按键者
[Diligent Key Presser](#)
磁盘破坏者
[Disk Crash](#)
除以零
[DividedByZero](#)
[DJCubed](#)
[dlatikay](#)
[DLeh](#)
德米特里·比琴科
[Dmitry Bychenko](#)
德米特里·叶戈罗夫
[Dmitry Egorov](#)
[DmitryG](#)
[docesam](#)
多兹迪·扎库马
[Dodzi Dzakuma](#)
唐纳德·韦布
[Donald Webb](#)
多鲁克
[Doruk](#)
医生
[dotctor](#)
鸽子
[dove](#)
[DPenner1](#)
德拉肯
[Draken](#)
德鲁·肯尼迪
[Drew Kennedy](#)
德鲁乔丹
[DrewJordan](#)
德罗里托斯
[Droritos](#)
不知道
[Dunno](#)
杜尔格帕尔·辛格
[Durgpal Singh](#)
[DVJex](#)
德威伯利
[Dweeberly](#)
[DWright](#)
埃蒙·查尔斯
[Eamon Charles](#)
[ebattulga](#)
埃德·吉布斯
[Ed Gibbs](#)
[EFrank](#)
埃赫桑·萨贾德
[Ehsan Sajjad](#)

第35章和第52章
第90章
第20章
第15章
第35、52、66、72、94和125章
第7、48、52、66、71、72、84和120章
第60章
第60章
第1章和第52章
第8章和第60章
第63章和第162章
第97章
第89章
第52章
第48章和第72章
第145章
第11章和第58章
第7章
第7章和第67章
第28章
第71章
第66章
第90章
第22章、第71章和第119章
第43章
第1章
第28章和第97章
第22章
第3、8、10、15、26、39、48、100、142和149章
第22、48、52、60和66章
第12和66章
第38章
第31章
第144章
第114章
第13章和第95章
第3章、第66章、第111章和第120章
第22章和第144章
第52章
第116章
第52章
第52章
第121章
第91章
第1章和第124章
第63章
第13章
第1章、第48章和第134章
第7章和第162章
第12章
第16章
第23章

第1章、第7章、第10章、第15章、第36章、第39章、第43章、第48章、第52章、第58章、第59章、第62章、第63章、第66章、第72章和第120章

Daryl
[dasblinkenlight](#)
[das_keyboard](#)
[Dave Zych](#)
[David](#)
[David Pine](#)
[David Stockinger](#)
[dav_i](#)
[DAXaholic](#)
[dazerdude](#)
[da_sann](#)
[dbmuller](#)
[dcastro](#)
[deadManN](#)
[Deepak gupta](#)
[delete me](#)
[deloreyk](#)
[demonplus](#)
[Denis Elkhov](#)
[Dialecticus](#)
[Didgeridoo](#)
[die maus](#)
[Dileep](#)
[Diligent Key Presser](#)
[Disk Crash](#)
[DividedByZero](#)
[DJCubed](#)
[dlatikay](#)
[DLeh](#)
[Dmitry Bychenko](#)
[Dmitry Egorov](#)
[DmitryG](#)
[docesam](#)
[Dodzi Dzakuma](#)
[Donald Webb](#)
[Doruk](#)
[dotctor](#)
[dove](#)
[DPenner1](#)
[Draken](#)
[Drew Kennedy](#)
[DrewJordan](#)
[Droritos](#)
[Dunno](#)
[Durgpal Singh](#)
[DVJex](#)
[Dweeberly](#)
[DWright](#)
[Eamon Charles](#)
[ebattulga](#)
[Ed Gibbs](#)
[EFrank](#)
[Ehsan Sajjad](#)
Chapters 35 and 52
Chapter 90
Chapter 20
Chapter 15
Chapters 35, 52, 66, 72, 94 and 125
Chapters 7, 48, 52, 66, 71, 72, 84 and 120
Chapter 60
Chapter 60
Chapters 1 and 52
Chapters 8 and 60
Chapters 63 and 162
Chapter 97
Chapter 89
Chapter 52
Chapters 48 and 72
Chapter 145
Chapters 11 and 58
Chapter 7
Chapters 7 and 67
Chapter 28
Chapter 71
Chapter 66
Chapter 90
Chapters 22, 71 and 119
Chapter 43
Chapter 1
Chapters 28 and 97
Chapter 22
Chapters 3, 8, 10, 15, 26, 39, 48, 100, 142 and 149
Chapters 22, 48, 52, 60 and 66
Chapters 12 and 66
Chapter 38
Chapter 31
Chapter 144
Chapter 114
Chapters 13 and 95
Chapters 3, 66, 111 and 120
Chapters 22 and 144
Chapter 52
Chapter 116
Chapter 52
Chapter 52
Chapter 121
Chapter 91
Chapters 1 and 124
Chapter 63
Chapter 13
Chapters 1, 48 and 134
Chapters 7 and 162
Chapter 12
Chapter 16
Chapter 23
Chapters 1, 7, 10, 15, 36, 39, 43, 48, 52, 58, 59, 62, 63, 66, 72 and 120

埃约书亚S
[ekolis](#)
[el2iot2](#)
[埃尔德·拉赫米](#)
[埃尔达尔·多尔吉耶夫](#)
[elibyy](#)
[埃尔顿](#)
[enrico.bacis](#)
[eouw0o83hf](#)
[ephtee](#)
[埃里克](#)
[埃里克·谢尔布姆](#)
[EvenPrime](#)
[fabriciorissetto](#)
[Facebamm](#)
[fahadash](#)
[法索](#)
[费德里科·阿洛卡蒂](#)
[Feelbad Soussi Wolfgang](#)
[DZ](#)
[费利佩·奥里亚尼](#)
[fernacolo](#)
[费尔南多·松本](#)
[Finickyflame](#)
[弗洛里安·科赫](#)
[弗洛林·M](#)
[flq](#)
[forsvarir](#)
[FortyTwo](#)
[Freelex](#)
[fubo](#)
[F_V](#)
[G](#)
[加杰德拉](#)
[ganchito55](#)
[加文·格林沃尔特](#)
[gbellmann](#)
[gdyrrahitis](#)
[gdziadkiewicz](#)
[乔治·达克特](#)
[GeralexGR](#)
[吉拉德·格林](#)
[吉拉德·纳曼](#)
[姜头](#)
[glaubergft](#)
[戈登·贝尔](#)
[goric](#)
[granmirupa](#)
[GregC](#)
[吉列尔梅·德·热苏斯·桑托斯](#)
[第19章](#)
[guntbert](#)
[H. 保威伦](#)
[haim770](#)

第4章、第52章和第114章
第48章
第48章
第52章和第94章
第112章
第24章
第48章
第48章和第72章
第7章
第54章、第110章和第116章
第66章
第60章、第66章和第72章
第52章、第66章和第148章
第15章和第66章
第160章
第48章
第66章和第72章
第84章
第97章
第19章
第52章
第3章、第20章、第34章、第48章和第52章
第66章
第1章
第66章
第76章
第52章
第65章
第28章
第47章和第66章
第32章、第47章和第52章
第1章
第120章和第155章
第94章、第105章和第148章
第52章和第111章
第66章
第114章
第48章
第8章、第76章、第90章、第119章和第120章
第147章
第66章
第52章和第76章
第72章
第150章
第1章和第99章
第34章
第20章
第59章
第114章
第4、12、15、22、40、72和75章
第84章

[EJoshuaS](#)
[ekolis](#)
[el2iot2](#)
[Elad Lachmi](#)
[Eldar Dordzhiev](#)
[elibyy](#)
[Elton](#)
[enrico.bacis](#)
[eouw0o83hf](#)
[ephtee](#)
[Erick](#)
[Erik Schierboom](#)
[EvenPrime](#)
[fabriciorissetto](#)
[Facebamm](#)
[fahadash](#)
[faso](#)
[Federico Allocati](#)
[Feelbad Soussi Wolfgang](#)
[DZ](#)
[Felipe Oriani](#)
[fernacolo](#)
[Fernando Matsumoto](#)
[Finickyflame](#)
[Florian Koch](#)
[Florin M](#)
[flq](#)
[forsvarir](#)
[FortyTwo](#)
[Freelex](#)
[fubo](#)
[F_V](#)
[G](#)
[Gajendra](#)
[ganchito55](#)
[Gavin Greenwalt](#)
[gbellmann](#)
[gdyrrahitis](#)
[gdziadkiewicz](#)
[George Duckett](#)
[GeralexGR](#)
[Gilad Green](#)
[Gilad Naaman](#)
[GingerHead](#)
[glaubergft](#)
[Gordon Bell](#)
[goric](#)
[granmirupa](#)
[GregC](#)
[Guilherme de Jesus Santos](#)
[Chapter 19](#)
[guntbert](#)
[H. Pauwelyn](#)
[haim770](#)

[hankide](#)
[哈里·普拉萨德](#)
[harryott](#)
[哈西布·阿西夫](#)
[hatcyl](#)
[海登](#)
[hellyale](#)
[亨里克·H](#)
[HimBromBeere](#)
[iaminvincible](#)
[伊恩](#)
[冰冷的蔑视](#)
[伊戈尔](#)
[iliketocode](#)
[intox](#)
[约安尼斯·卡拉迪马斯](#)
[以撒](#)
[伊万·尤尔琴科](#)
[J. 斯廷](#)
[j3soon](#)
[ja72](#)
[杰克](#)
[雅各布·林尼](#)
[Jacobr365](#)
[杰克](#)
[杰克·法利](#)
[詹姆斯](#)
[詹姆斯·埃利斯](#)
[詹姆斯·休斯](#)
[杰米·里斯](#)
[杰米·特威尔斯](#)
[扬·博科夫斯基](#)
[扬·佩尔德·伊莫夫斯基](#)
[jao](#)
[雅罗斯拉夫·卡德莱茨](#)
[贾罗德·迪克森](#)
[贾斯敏·索兰基](#)
[jaycer](#)
[贾伊迪普·贾达夫](#)
[Jcoffman](#)
[让](#)
[杰弗伦](#)
[耶普·斯蒂格·尼尔森](#)
[杰里米·加藤](#)
[杰罗米·欧文](#)
[杰西·威廉姆斯](#)
[jHilscher](#)
[吉姆](#)
[若昂·洛伦索](#)
[乔德雷尔](#)
[乔](#)
[乔·阿门塔](#)
[乔尔·马丁内斯](#)

[hankide](#)
[Hari Prasad](#)
[harryott](#)
[Haseeb Asif](#)
[hatcyl](#)
[Hayden](#)
[hellyale](#)
[Henrik H](#)
[HimBromBeere](#)
[iaminvincible](#)
[Ian](#)
[Icy Defiance](#)
[Igor](#)
[iliketocode](#)
[intox](#)
[Ioannis Karadimas](#)
[Isac](#)
[Ivan Yurchenko](#)
[J. Steen](#)
[j3soon](#)
[ja72](#)
[Jack](#)
[Jacob Linney](#)
[Jacobr365](#)
[Jake](#)
[Jake Farley](#)
[James](#)
[James Ellis](#)
[James Hughes](#)
[Jamie Rees](#)
[Jamie Twells](#)
[Jan Bońkowski](#)
[Jan Peldřimovský](#)
[jao](#)
[Jaroslav Kadlec](#)
[Jarrod Dixon](#)
[Jasmin Solanki](#)
[jaycer](#)
[Jaydip Jadhav](#)
[Jcoffman](#)
[Jean](#)
[Jephron](#)
[Jeppe Stig Nielsen](#)
[Jeremy Kato](#)
[Jeremy Irvine](#)
[Jesse Williams](#)
[jHilscher](#)
[Jim](#)
[João Lourenço](#)
[Jodrell](#)
[Joe](#)
[Joe Amenta](#)
[Joel Martinez](#)

约翰
约翰·迈耶
约翰·彼得斯
约翰·斯莱格斯
约翰机器人
约翰·L·贝文
约翰尼·斯科夫达尔
乔乔德莫
第72章
第63章
第72章
第66章
第48章
第58章
第66章
乔恩·施奈德
乔恩·斯基特
乔纳斯·S
琼斯城
约书亚·贝伦斯
J.T.
尤尔根·D
尤哈·帕洛马克
朱利安·隆卡利亚
[just.another.programmer](#)
just.ru
卡米尔克
凯恩
karaken12
卡尔蒂克
肯·基南
ken2k
凯文·迪特拉格利亚
凯文·格林
凯文·蒙特罗斯
基拉祖尔
Kimmax
Kit
凯尔坦
科比
Konamiman
康斯坦丁·弗多夫金
Koopakiller
克里科尔·艾兰吉安
krimog
Kritner
Kroltan
Krzyslerious
克日什托夫·布拉尼茨基
凯尔·特劳伯曼
基里洛·M
拉塞·沃格塞瑟·卡尔森
[LegionMammal978](#)
leondepedlaw
levininja
Lijo

第40章和第86章
第76章
第48章
第48章和第52章
第72章
第3章、第61章、第86章和第112章
第72章
第66章
第48章
第58章
第66章
第7、11、13、14、17、20、22、27、28、38、47、52、66、70和99章
第52和60章
第3、10、26、45、48和72章
第7和60章
第52章
第60章
第11章
第27章
第39章和第120章
第90章
第120章
第39章
第9章
第66章
第105章
第60章
第1章和第72章
第48章
第65章
第1章和第114章
第10章和第35章
第3章和第71章
第61章
第52章
第8、40和60章
第11、39、52、58、63和72章
第38章
第66章
第72章
第77章
第1章和第71章
第46章
第1章
第43章
第60章
第113章
第60章
第112章
第52章
第29章和第36章

John
John Meyer
John Peters
John Slegers
Johnbot
JohnLBevan
Johny Skovdal
Jojodmo
Jon Bates
Jon Erickson
Jon Ericson
Jon G
Jon Schneider
Jon Skeet
Jonas S
Jonesopolis
JoshuaBehrens
J.T.
Juergen d
Juha Palomäki
Julien Roncaglia
[just.another.programmer](#)
just.ru
kamilk
Kane
karaken12
Karthik
Ken Keenan
ken2k
Kevin DiTraglia
Kevin Green
Kevin Montrose
Kilazur
Kimmax
Kit
Kjartan
Kobi
Konamiman
Konstantin Vdovkin
Koopakiller
Krikor Ailanjian
krimog
Kritner
Kroltan
Krzyslerious
Krzysztof Branicki
Kyle Traberman
Kyrylo M
Lasse Vågsæther Karlsen
[LegionMammal978](#)
leondepedlaw
levininja
Lijo

Chapters 40 and 86
Chapter 76
Chapter 48
Chapters 48 and 52
Chapter 72
Chapters 3, 61, 86 and 112
Chapter 63
Chapter 72
Chapter 66
Chapter 48
Chapter 58
Chapter 66
Chapters 7, 11, 13, 14, 17, 20, 22, 27, 28, 38, 47, 52, 66, 70 and 99
Chapters 52 and 60
Chapters 3, 10, 26, 45, 48 and 72
Chapters 7 and 60
Chapter 52
Chapter 60
Chapter 11
Chapter 27
Chapters 39 and 120
Chapter 90
Chapter 120
Chapter 39
Chapter 9
Chapter 66
Chapter 105
Chapter 60
Chapters 1 and 72
Chapter 48
Chapter 65
Chapters 1 and 114
Chapters 10 and 35
Chapters 3 and 71
Chapter 61
Chapter 52
Chapters 8, 40 and 60
Chapters 11, 39, 52, 58, 63 and 72
Chapter 38
Chapter 66
Chapter 72
Chapter 77
Chapters 1 and 71
Chapter 46
Chapter 1
Chapter 43
Chapter 60
Chapter 113
Chapters 60 and 72
Chapter 60
Chapter 112
Chapter 52
Chapters 29 and 36

[lloyd](#)
[LMK](#)
[洛希塔·帕拉吉里](#)
[洛克什·拉姆](#)
[洛库斯金](#)
[洛伦茨·韦德勒](#)
[洛斯马诺斯](#)
[洛斯拉里亚斯](#)
[洛维](#)
[ltiveron](#)
[卢卡兰斯基](#)
[卢卡斯·科莱茨基](#)
[卢克·瑞安](#)
[M·莫尼斯·艾哈迈德·汗](#)
[MaddinTribleD](#)
[maf](#)
[Mafii](#)
[马基恩](#)
[玛姆塔·D](#)
[芒果·黄](#)
[马涅罗](#)
[曼塞尔D](#)
[马克·格拉维尔](#)
[马克·维特曼](#)
[马克E](#)
[马尔钦·尤拉谢克](#)
[马尔科](#)
[马尔科·斯卡比奥洛](#)
[马雷克·穆谢拉克](#)
[马克·谢甫琴科](#)
[马丁](#)
[马丁·克林克](#)
[马丁·齐克蒙德](#)
[马塔斯·瓦伊特凯维丘斯](#)
[马廷·乌尔哈克](#)
[马修·甘东](#)
[马特](#)
[马特·罗兰](#)
[马特·托马斯](#)
[matteeyah](#)
[马修·怀特德](#)
[马蒂斯·韦塞尔斯](#)
[马克斯](#)
[马克西姆](#)
[马克西姆](#)
[马克西米连·阿斯特](#)
[mbrdev](#)
[mburleigh](#)
[MCronin](#)
[MDTech.us MAN](#)
[梅德尼·巴伊卡尔](#)
[mehrandvd](#)
[meJustAndrew](#)

第158章
第48章
第66章
第104章
第3章和第52章
第60章
第95章
第43章和第48章
第63章
第39章
第48章和第156章
第95章
第30章和第80章
第1章
第76章
第71章
第52章、第66章和第72章
第72章
第52章和第112章
第52章
第11章
第159章
第71章
第155章
第41章和第48章
第52章、第71章和第72章
第47章
第52章
第12章
第12章、第43章、第72章、第99章、第115章和第134章
第52章
第52章
第7、22、48、52、60和66章
第3、12、19、48、52、58、59、62、63、66、72、76、84、90和93章
第1、52和66章
第52章
第1、1、66、72、72、76和96章
第71章
第71章
第76、89和90章
第52章
第112章
第8和52章
第125章
第22、51、62和66章
第38章、第52章和第105章
第72章和第88章
第66章
第42章
第48章
第52章
第49章和第153章
第3章、第6章、第48章、第59章、第89章和第90章

[lloyd](#)
[LMK](#)
[Lohitha Palagiri](#)
[Lokesh_Ram](#)
[lokusking](#)
[Lorentz Vedeler](#)
[LosManos](#)
[lothlarias](#)
[Lovy](#)
[ltiveron](#)
[Lukáš Lánský](#)
[Lukas Kolletzki](#)
[Luke Ryan](#)
[M Monis Ahmed Khan](#)
[MaddinTribleD](#)
[maf](#)
[Mafii](#)
[Makyen](#)
[Mamta D](#)
[Mango Wong](#)
[Maniero](#)
[ManselD](#)
[Marc Gravell](#)
[Marc Wittmann](#)
[MarcE](#)
[MarcinJuraszek](#)
[Marco](#)
[Marco Scabbio](#)
[Marek Musielak](#)
[Mark Shevchenko](#)
[Martin](#)
[Martin Klinke](#)
[Martin Zikmund](#)
[Matas Vaitkevicius](#)
[Mateen Ulhaq](#)
[Mathieu Guindon](#)
[Matt](#)
[Matt Rowland](#)
[Matt Thomas](#)
[matteeyah](#)
[Matthew Whited](#)
[Matthijs Wessels](#)
[Max](#)
[Maxim](#)
[Maxime](#)
[Maximilian Ast](#)
[mbrdev](#)
[mburleigh](#)
[MCronin](#)
[MDTech.us MAN](#)
[Medeni Baykal](#)
[mehrandvd](#)
[meJustAndrew](#)

[Mellow](#)
[MGB](#)
[迈克尔·B](#)
[迈克尔·本福德](#)
[迈克尔·布兰登·莫里斯](#)
[迈克尔·迈雷格尔](#)
[迈克尔·理查森](#)
[迈克尔·索伦斯](#)
[米歇尔·塞奥](#)
[米歇尔·范·奥斯特豪特](#)
[迈克Z](#)
[MikeS159](#)
[米科·维塔拉](#)
[米兰·桑切斯](#)
[米尔顿·埃尔南德斯](#)
[米奇·塔尔马奇](#)
[莫·法拉格](#)
[莫辛·汗](#)
[mok](#)
[莫赫塔尔·阿舒尔](#)
[Mostafiz](#)
[MotKohn](#)
[Mourndark](#)
[Mindor先生](#)
[MrDKOz](#)
[MSE](#)
[MSL](#)
[穆罕默德·阿尔巴尔马维](#)
[MuiBienCarlota](#)
[穆贾西尔·纳西尔](#)
[穆尔德](#)
[穆尔图扎·沃赫拉](#)
[mybirthname](#)
[内特·巴贝蒂尼](#)
[内森·塔吉](#)
[纳文·戈吉内尼](#)
[内哈·贾因](#)
[尼奥·维杰](#)
[尼古拉斯·赛泽](#)
[尼古劳斯·劳森](#)
[尼克](#)
[尼克·拉尔森](#)
[nickguletskii](#)
[尼科](#)
[nietras](#)
[nik](#)
[nikchi](#)
[尼基尔·瓦尔塔克](#)
[尼基塔](#)
[尼古拉·孔德拉季耶夫](#)
[Noctis](#)
[nollidge](#)
[数据不足](#)

第72章和第112章
第12章
第1章
第48章
第54章
第22章、第27章、第31章和第52章
第27章、第39章、第48章、第52章、第72章、第114章和第120章
第87章
第31章
第64章
第72章
第150章
第3、38、51、66、97、119和137章
第1章
第142章
第72章
第31章
第102章
第66章
第117章
第19章
第9章和第71章
第63章
第72章
第57章
第37章、第41章和第152章
第47章
第23章
第72章和第147章
第59章
第54章
第117章
第146章
第3章、第15章、第35章、第66章、第72章和第114章
第90章
第100章
第66章
第82章
第8和52章
第52章和第98章
第1章
第48章
第114章
第3章、第48章、第66章和第71章
第71章
第113章
第12章
第86章
第23章和第27章
第22、71、90、94、95、96和129章
第66章
第72章
第2、52、84和89章

[Mellow](#)
[MGB](#)
[Michael B](#)
[Michael Benford](#)
[Michael Brandon Morris](#)
[Michael Mairegger](#)
[Michael Richardson](#)
[Michael Sorens](#)
[Michele Ceo](#)
[Michiel van Oosterhout](#)
[mike z](#)
[MikeS159](#)
[Mikko Viitala](#)
[Millan Sanchez](#)
[Milton Hernandez](#)
[Mitch Talmadge](#)
[Moe Farag](#)
[Mohsin khan](#)
[mok](#)
[Mokhtar Ashour](#)
[Mostafiz](#)
[MotKohn](#)
[Mourndark](#)
[Mr.Mindor](#)
[MrDKOz](#)
[MSE](#)
[MSL](#)
[Muhammad Albarmawi](#)
[MuiBienCarlota](#)
[Mujassir Nasir](#)
[Mulder](#)
[Murtuza Vohra](#)
[mybirthname](#)
[Nate Barbettini](#)
[Nathan Tuggy](#)
[Naveen Gogineni](#)
[Neha Jain](#)
[Neo Vijay](#)
[Nicholas Sizer](#)
[Nicholaus Lawson](#)
[Nick](#)
[Nick Larsen](#)
[nickguletskii](#)
[Nico](#)
[nietras](#)
[nik](#)
[nikchi](#)
[Nikhil Vartak](#)
[Nikita](#)
[Nikolay Kondratyev](#)
[Noctis](#)
[nollidge](#)
[NotEnoughData](#)

Chapters 72 and 112
Chapter 12
Chapter 1
Chapter 48
Chapter 54
Chapters 22, 27, 31 and 52
Chapters 27, 39, 48, 52, 72, 114 and 120
Chapter 87
Chapter 31
Chapter 64
Chapter 72
Chapter 150
Chapters 3, 38, 51, 66, 97, 119 and 137
Chapter 1
Chapter 142
Chapter 72
Chapter 31
Chapter 102
Chapter 66
Chapter 117
Chapter 19
Chapters 9 and 71
Chapter 63
Chapter 72
Chapter 57
Chapters 37, 41 and 152
Chapter 47
Chapter 23
Chapters 72 and 147
Chapter 59
Chapter 54
Chapter 117
Chapter 146
Chapters 3, 15, 35, 66, 72 and 114
Chapter 90
Chapter 100
Chapter 66
Chapter 82
Chapters 8 and 52
Chapters 52 and 98
Chapter 1
Chapter 48
Chapter 114
Chapters 3, 48, 66 and 71
Chapter 71
Chapter 113
Chapter 12
Chapter 86
Chapters 23 and 27
Chapters 22, 71, 90, 94, 95, 96 and 129
Chapter 66
Chapter 72
Chapters 2, 52, 84 and 89

不是我自己
[NtFreX](#)
[numaroth](#)
努里·塔斯德米尔
奥格拉斯
奥贡
奥仁
奥利弗·梅莱特
奥利维耶·德·穆尔德
奥卢瓦费米
奥利
奥兰多·威廉
潘
帕尔斯·帕特尔
[pascalhein](#)
帕特里克·霍夫曼
保罗·韦兰德
[paulius_I](#)
帕维尔·杜罗夫
帕维尔·马约罗夫
帕维尔·帕亚·哈尔比赫
帕维尔·萨佩欣
帕维尔·沃罗宁
帕维尔·叶尔马洛维奇
[Pawel_Hemperek](#)
帕维乌·马赫
佩德罗
佩德罗·索基
皮特
皮特·呃
皮特利兹
彼得·戈登
彼得·霍梅尔
彼得·K
彼得·L。
彼得·胡德ček
[petrzjunior](#)
菲利普·C
[pid](#)
皮埃尔·特亚特
[pinkfloydx33](#)
[PMF](#)
普拉蒂克
普罗克西玛
[PSGuy](#)
普什潘德拉
派瑞蒂
[qjake](#)
雷米
拉西尔·希兰
拉胡尔·尼卡特
[raidensan](#)
[Raidri](#)

第1章
第164章
第52章
第48、52、61、63、71、72、81、109和119章
第150章
第20章
第1章
第72章
第30章
第13、19、28、106和119章
第3章、第39章、第59章和第84章
第72章
第54章
第48、72、87和134章
第3章和第52章
第32和38章
第11章
第89章
第155章
第66章和第71章
第52章和第76章
第39章和第95章
第3章、第59章、第72章、第76章和第114章
第19章和第52章
第66章
第52章
第66章
第85章和第134章
第150章
第58章
第3章和第7章
第16章
第39章
第60章
第52章
第16章和第91章
第23章
第9章和第12章
第60章
第134章
第9章、第38章、第39章、第48章、第66章和第114章
第41、52、55和76章
第1章
第74章
第89章
第109章
第71章
第66、71、72和93章
第66章
第2、3、9、11、28、66和72章
第7、12、33、45和108章
第76章
第12章

[NotMyself](#)
[NtFreX](#)
[numaroth](#)
[Nuri_Tasdemir](#)
[OgglaS](#)
[Ogoun](#)
[Ojen](#)
[Oliver_Mellet](#)
[Olivier_De_Meulder](#)
[Oluwafemi](#)
[Oly](#)
[Orlando_William](#)
[Pan](#)
[Parth_Patel](#)
[pascalhein](#)
[Patrick_Hofman](#)
[Paul_Weiland](#)
[paulius_I](#)
[Pavel_Durov](#)
[Pavel_Mayorov](#)
[Pavel_Pája_Halbich](#)
[Pavel_Sapehin](#)
[Pavel_Voronin](#)
[Pavel_Yermalovich](#)
[Pawel_Hemperek](#)
[Pawel_Mach](#)
[Pedro](#)
[PedroSouki](#)
[Pete](#)
[Pete_Uh](#)
[petelids](#)
[Peter_Gordon](#)
[Peter_Hommel](#)
[Peter_K](#)
[Peter_L.](#)
[Petr_Hudeček](#)
[petrzjunior](#)
[Philip_C](#)
[pid](#)
[Pierre_Theate](#)
[pinkfloydx33](#)
[PMF](#)
[Prateek](#)
[Proxima](#)
[PSGuy](#)
[Pushpendra](#)
[Pyritie](#)
[qjake](#)
[Rémi](#)
[Racil_Hilan](#)
[Rahul_Nikate](#)
[raidensan](#)
[Raidri](#)

拉杰·拉奥	第1章	Raj Rao	Chapter 1
拉吉普特	第1章和第96章	Rajput	Chapters 1 and 96
拉基蒂ć	第1章和第52章	Rakitić	Chapters 1 and 52
拉尔夫·伯宁	第144章	Ralf Bönnig	Chapter 144
拉斐尔·潘塔莱昂	第65章	Raphael Pantaleão	Chapter 65
拉文德拉	第128章	ravindra	Chapter 128
雷耶尔	第23章	RaYell	Chapter 23
拉赞	第52章	Razan	Chapter 52
RBT	第52章	RBT	Chapter 52
rdans	第66章和第72章	rdans	Chapters 66 and 72
递归	第22章	recursive	Chapter 22
雷内·沃格特	第63章和第155章	René Vogt	Chapters 63 and 155
雷南·杰米尼亞尼	第52章	Renan Gemignani	Chapter 52
雷扎·阿加伊	第89章	Reza Aghaei	Chapter 89
瑞斯O	第3章	RhysO	Chapter 3
里卡多·阿莫雷斯	第10章和第120章	Ricardo Amores	Chapters 10 and 120
里查·加尔格	第22章、第59章、第72章和第96章	Richa Garg	Chapters 22, 59, 72 and 96
理查德	第12、12、72和119章	Richard	Chapters 12, 12, 72 and 119
理查德	第122章	richard	Chapter 122
林吉尔	第52章	Ringil	Chapter 52
里昂·威廉姆斯	第1、7、10、12、15、48、52、58、60、66、72、120和128章	Rion Williams	Chapters 1, 7, 10, 12, 15, 48, 52, 58, 60, 66, 72, 120 and 128
罗布	第26、52、72和119章	Rob	Chapters 26, 52, 72 and 119
罗班	第72章	Robban	Chapter 72
罗伯特·哥伦比亚	第3、9、17、20、39、48、52、54、58、72和76章	Robert Columbia	Chapters 3, 9, 17, 20, 39, 48, 52, 54, 58, 72 and 76
RobSiklos	第52章	RobSiklos	Chapter 52
rocky	第120章	rocky	Chapter 120
RoelF	第11章	RoelF	Chapter 11
Rokey Ge	第69章	Rokey Ge	Chapter 69
罗曼·马鲁西克	第131章	Roman Marusyk	Chapter 131
RomCoo	第1、52和155章	RomCoo	Chapters 1, 52 and 155
罗伊·迪克图斯	第54章和第132章	Roy Dictus	Chapters 54 and 132
RubberDuck	第13章	RubberDuck	Chapter 13
鲁本斯·法里亚斯	第69章	Rubens Farias	Chapter 69
瑞安·阿博特	第60章	Ryan Abbott	Chapter 60
瑞安·希尔伯特	第72章	Ryan Hilbert	Chapter 72
ryanyuyu	第12章、第52章和第72章	ryanyuyu	Chapters 12, 52 and 72
S. 兰格利	第66章	S. Rangeley	Chapter 66
S. 阿克巴里	第66章	S.Akbari	Chapter 66
S. 戴夫	第47章	S.Dav	Chapter 47
S.L. 巴斯	第79章和第84章	S.L. Barth	Chapters 79 and 84
萨钦·查万	第63章	Sachin Chavan	Chapter 63
萨尔瓦多·鲁比奥·马丁内斯	第66章	Salvador Rubio Martinez	Chapter 66
萨姆	第3章、第52章和第59章	Sam	Chapters 3, 52 and 59
萨姆·阿克斯	第72章	Sam Axe	Chapter 72
samuelesque	第71章	samuelesque	Chapter 71
桑杰·拉达迪亚	第52章和第66章	Sanjay Radadiya	Chapters 52 and 66
萨蒂什·亚达夫	第66章	Satish Yadav	Chapter 66
scher	第6章	scher	Chapter 6
斯科特	第52章	Scott	Chapter 52
斯科特·科兰德	第60章和第76章	Scott Koland	Chapters 60 and 76
肖恩	第60章	Sean	Chapter 60
sebingel	第113章	sebingel	Chapter 113
SeeuD1	第1章	SeeuD1	Chapter 1

谢尔格·罗戈夫采夫
[塞尔吉奥·多明格斯](#)
[sferencik](#)
[shawty](#)
[Shelby115](#)
[Shoe](#)
[Shuffler](#)
[Shyju](#)
[Sibeesh Venu](#)
[西瓦南塔姆·帕迪卡苏](#)
[Sjoerd222888](#)
[船长](#)
[Sklivvz](#)
[骷髅狂热](#)
[SlaterCodes](#)
[slawekwin](#)
[slinzerthegod](#)
[smead](#)
[snickro](#)
[Snipzwolf](#)
[Snympo](#)
[Sobieck](#)
[sohnryang](#)
[solidcell](#)
[somebody](#)
[Sompom](#)
[Sondre](#)
[索菲·杰克逊](#)
[松鼠](#)
[stackptr](#)
[Stavm](#)
[斯特凡·斯泰格](#)
[stefankmitph](#)
[斯特法诺·丹东尼奥](#)
[史蒂文](#)
[斯蒂尔加尔](#)
[斯特里普林勇士](#)
[斯图](#)
[苏杰·萨尔马](#)
[桑尼·R·古普塔](#)
[苏普拉杰·V](#)
[苏伦·斯拉皮扬](#)
[苏亚什·库马尔·辛格](#)
[斯沃尔格赫](#)
[协同编码者](#)
[塔格克](#)
[塔米尔·维雷德](#)
[坦纳·斯韦特](#)
[塔拉斯](#)
[塔尔卡达尔](#)
[特多夫](#)
[特奥·范·科特](#)
[测试123](#)

[Serg Rogovtsev](#)
[Sergio Domínguez](#)
[sferencik](#)
[shawty](#)
[Shelby115](#)
[Shoe](#)
[Shuffler](#)
[Shyju](#)
[Sibeesh Venu](#)
[Sivanantham Padikkasu](#)
[Sjoerd222888](#)
[Skipper](#)
[Sklivvz](#)
[Skullomania](#)
[SlaterCodes](#)
[slawekwin](#)
[slinzerthegod](#)
[smead](#)
[snickro](#)
[Snipzwolf](#)
[Snympo](#)
[Sobieck](#)
[sohnryang](#)
[solidcell](#)
[somebody](#)
[Sompom](#)
[Sondre](#)
[Sophie Jackson](#)
[Squirrel](#)
[stackptr](#)
[Stavm](#)
[Stefan Steiger](#)
[stefankmitph](#)
[Stefano d'Antonio](#)
[Steven](#)
[Stilgar](#)
[StriplingWarrior](#)
[Stu](#)
[Sujay Sarma](#)
[Sunny R Gupta](#)
[Supraj v](#)
[Suren Srapyan](#)
[Suyash Kumar Singh](#)
[Sworgkh](#)
[SynerCoder](#)
[Tagc](#)
[Tamir Vered](#)
[Tanner Swett](#)
[Taras](#)
[TarkaDaal](#)
[tehDorf](#)
[teo van kot](#)
[Testing123](#)

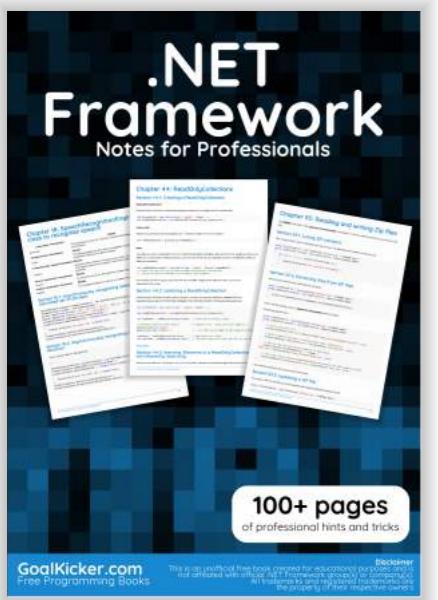
th1rdey3	第10章、第52章、第66章、第129章和第130章	th1rdey3	Chapters 10, 52, 66, 129 and 130
索登	第40章	Thaoden	Chapter 40
theB	第93章	theB	Chapter 93
thehenny	第157章	thehenny	Chapter 157
theinarasu	第19章	theinarasu	Chapter 19
特纳拉桑	第62章和第110章	Thennarasan	Chapters 62 and 110
西奥多罗斯	第22、31、52、59、66、74、123、140和144章	Theodoros	Chapters 22, 31, 52, 59, 66, 74, 123, 140 and 144
哈齐吉安纳基斯	第11、13、14、20、24、25、42、50、51和128章	Chatzigiannakis	Chapters 11, 13, 14, 20, 24, 25, 42, 50, 51 and 128
局外人	第20章	The_Outsider	Chapter 20
托马斯·韦勒	第52章	Thomas_Weller	Chapter 52
托斯滕·迪特马尔	第63和84章	Thorsten_Dittmar	Chapters 63 and 84
斯里格尔	第19、76和96章	Thriggle	Chapters 19, 76 and 96
图拉尼·奇万迪克瓦	第11章	Thulani_Chivandikwa	Chapter 11
tiedied61	第24章、第52章和第106章	tiedied61	Chapters 24, 52 and 106
蒂姆·埃本泽尔	第66章	Tim_Ebenezer	Chapter 66
蒂姆·艾尔斯	第39章	Tim_Iles	Chapter 39
蒂姆·梅多拉	第69章、第103章和第133章	Tim_Medora	Chapters 69, 103 and 133
蒂蒙·波斯特	第48章	Timon_Post	Chapter 48
蒂莫西·拉舍尔	第8章	Timothy_Rascher	Chapter 8
蒂莫西·希尔兹	第48章和第95章	Timothy_Shields	Chapters 48 and 95
TKharaishvili	第1章	TKharaishvili	Chapter 1
Tofix	第28章、第97章和第124章	Tofix	Chapters 28, 97 and 124
托尔加·埃夫西门	第1章	Tolga_Evcimen	Chapter 1
汤姆·鲍尔斯	第60章和第72章	Tom_Bowers	Chapters 60 and 72
汤姆·德罗斯特	第154章	Tom_Droste	Chapter 154
汤米	第16章、第20章、第52章和第81章	Tommy	Chapters 16, 20, 52 and 81
tonirush	第52章	tonirush	Chapter 52
topolm	第20章	topolm	Chapter 20
托尔莫德·豪根	第52章	Tormod_Haugene	Chapter 52
托特·扎姆	第1章、第66章和第72章	Tot_Zam	Chapters 1, 66 and 72
特拉维斯·J	第72章	Travis_J	Chapter 72
特伦特	第114章	Trent	Chapter 114
曾	第56章	Tseng	Chapter 56
tsjnsn	第1章、第15章、第48章、第72章和第134章	tsjnsn	Chapters 1, 15, 48, 72 and 134
图沙尔·帕特尔	第3章	Tushar_patel	Chapter 3
泰·科布	第67章	TyCobb	Chapter 67
乌阿里	第11、47和78章	Uali	Chapters 11, 47 and 78
未定义	第141章	un	Chapter 141
未定义	第113章	undefined	Chapter 113
下划线	第71章和第72章	Underscore	Chapters 71 and 72
乌里尔	第71、114和128章	Uriil	Chapters 71, 114 and 128
用户2321864	第1章、第48章、第52章、第59章、第66章和第71章	user2321864	Chapters 1, 48, 52, 59, 66 and 71
user3185569	第48章和第72章	user3185569	Chapters 48 and 72
user5389107	第39章	user5389107	Chapter 39
usr	第71章和第72章	usr	Chapters 71 and 72
uTeisT	第5章	uTeisT	Chapter 5
瓦迪姆·马尔蒂诺夫	第159章	Vadim_Martynov	Chapter 159
Vaibhav_Welcomes_You	第52章、第66章和第123章	Vaibhav_Welcomes_You	Chapters 52, 66 and 123
瓦伦丁	第66章	Valentin	Chapter 66
varocarbas	第89章	varocarbas	Chapter 89
vbnet3d	第52章和第72章	vbnet3d	Chapters 52 and 72
vcsjones	第1章、第48章和第72章	vcsjones	Chapters 1, 48 and 72
Ven		Ven	

[vesi](#)
维克多·托迈利
[VictorB](#)
[viggity](#)
[VirusParadox](#)
维塔利·费多琴科
[VitorCioletti](#)
弗拉德
[void](#)
[volvis](#)
[wablab](#)
李伟哈
[芥末迷](#)
[wertzui](#)
维克托·Dębski
威尔
威尔·雷
[门洛巫师](#)
[木材切碎机](#)
[沃尔特](#)
[WQYeo](#)
怀克
[桑德·卢西亚诺](#)
[Xandrmore](#)
[Xiaoy312](#)
雅各布·埃利斯
亚奈
亚沙尔·阿里亚巴西
约塔姆·萨尔蒙
[yumaikas](#)
[伊夫·谢尔普](#)
[扎洛蒙](#)
[ZenLulz](#)
齐亚德·阿基基
佐巴
[佐哈尔·佩莱德](#)

第76章
第1章和第48章
第59章和第66章
第1章、第6章、第48章和第72章
第58章
第38章和第66章
第82章
第88章和第120章
第24章、第52章和第66章
第93章
第63章
第32、71和72章
第8、9、15、20、22、52、71、97和120章
第27章
第44章
第116章
第7章、第27章和第51章
第11章
第52章
第39章
第20章
第118章
第98章
第52章
第3、42和66章
第66章
第22章
第53章
第20章
第92章
第54章
第52章
第108章
第1章
第112章
第52章和第86章

[vesi](#)
[Victor Tomaili](#)
[VictorB](#)
[viggity](#)
[VirusParadox](#)
Vitaliy Fedorchenco
[VitorCioletti](#)
[Vlad](#)
[void](#)
[volvis](#)
[wablab](#)
Wai Ha Lee
Wasabi Fan
[wertzui](#)
[Wiktor Dębski](#)
[Will](#)
Will Ray
[WizardOfMenlo](#)
[Woodchipper](#)
[Wouter](#)
[WQYeo](#)
[Wyck](#)
[Xander Luciano](#)
[Xandrmore](#)
[Xiaoy312](#)
Yaakov Ellis
[Yanai](#)
[Yashar Aliabasi](#)
[Yotam Salmon](#)
[yumaikas](#)
[Yves Schelpe](#)
[Zalomon](#)
[ZenLulz](#)
[Ziad Akiki](#)
[Zoba](#)
[Zohar Peled](#)

你可能也喜欢



You may also like

