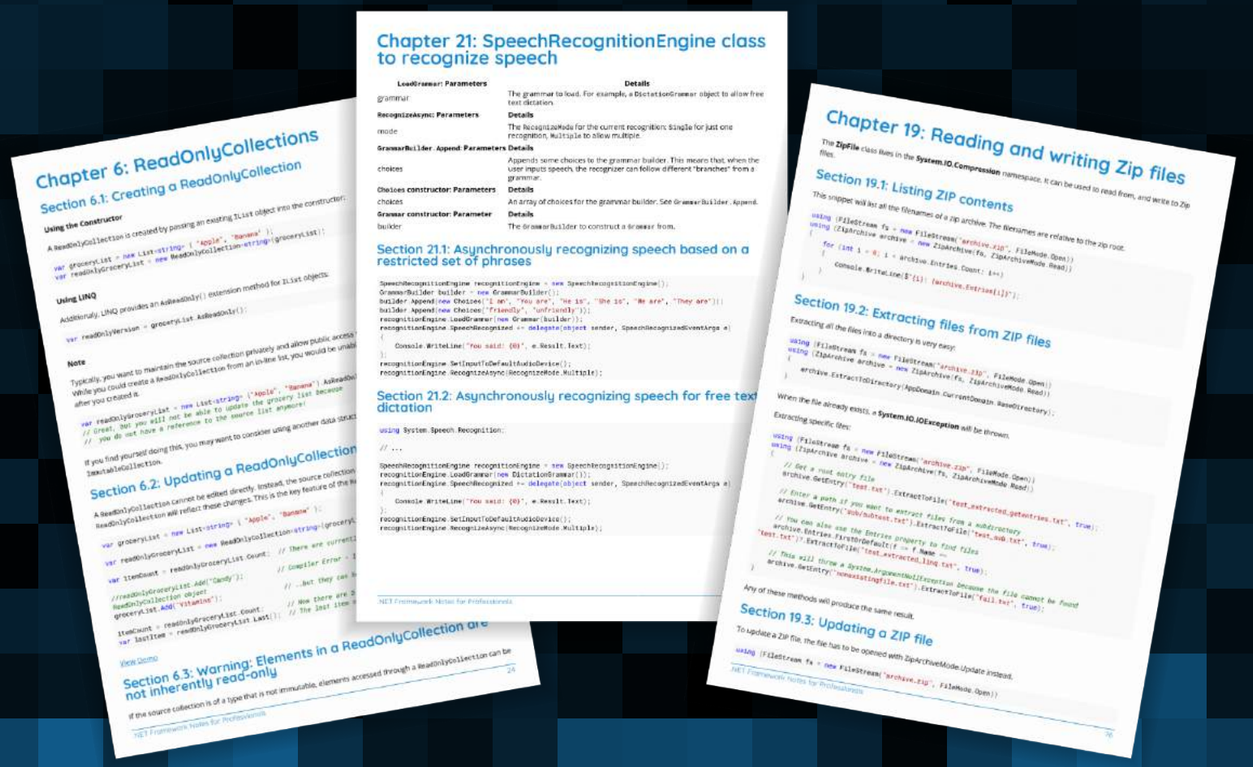
.NET 框架

.NET 框架

专业人士笔记



100 多页

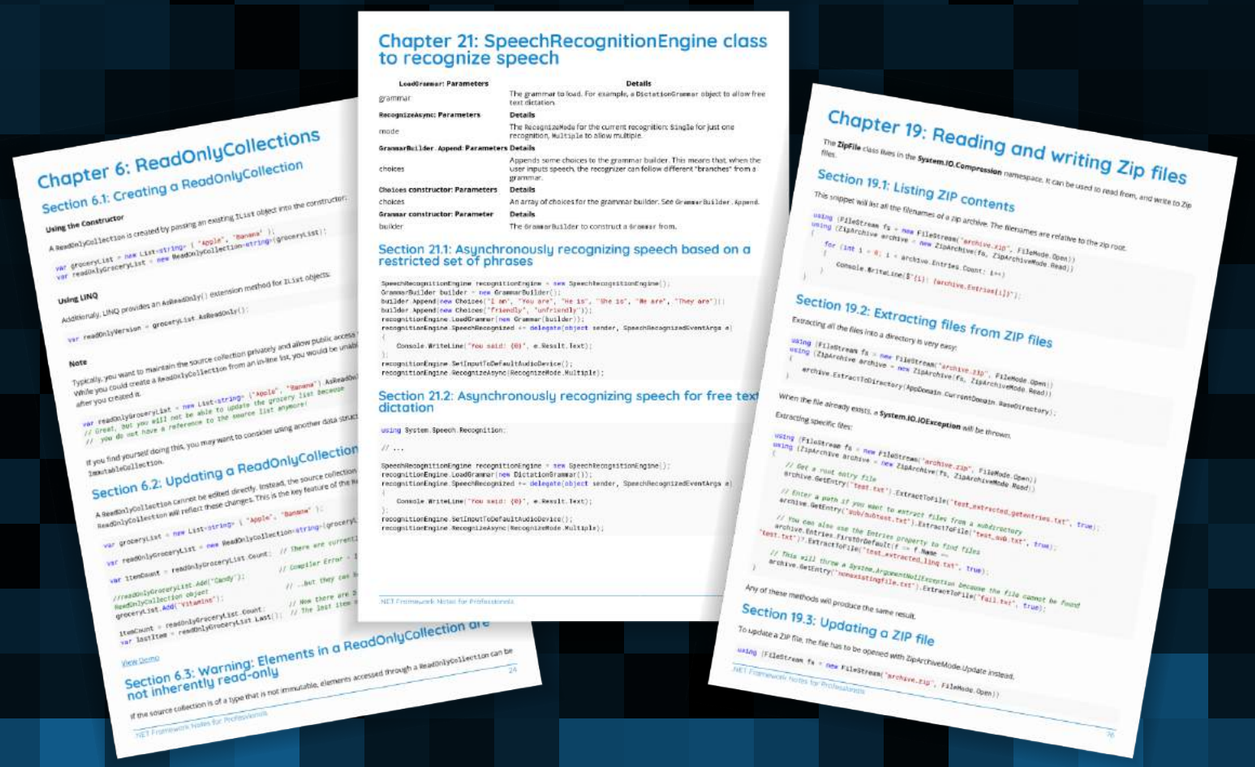
专业提示和技巧

GoalKicker.com

免费编程书籍

.NET Framework

Notes for Professionals



100+ pages

of professional hints and tricks

GoalKicker.com

Free Programming Books

目录

Contents

关于

欢迎随意免费分享此 PDF，
本书的最新版本可从以下网址下载：
https://goalkicker.com/DotNETFrameworkBook

请将反馈和更正发送至web@petercv.com

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/DotNETFrameworkBook

Please send feedback and corrections to web@petercv.com

第1章：开始使用.NET框架

.NET

版本 发布日期

1.0	2002-02-13
1.1	2003-04-24
2.0	2005-11-07
3.0	2006-11-06
3.5	2007-11-19
3.5 SP1	2008-08-11
4.0	2010-04-12
4.5	2012-08-15
4.5.1	2013-10-17
4.5.2	2014-05-05
4.6	2015-07-20
4.6.1	2015-11-17
4.6.2	2016-08-02
4.7	2017-04-05
4.7.1	2017-10-17

紧凑型框架

版本发布日期

1.0	2000-01-01
2.0	2005-10-01
3.5	2007-11-19
3.7	2009-01-01
3.9	2013-06-01

微型框架

版本发布日期

4.2	2011-10-04
4.3	2012-12-04
4.4	2015-10-20

第1.1节：C#中的Hello World

```
using System;

class Program
{
    // Main()函数是程序中第一个被执行的函数
    static void Main()
    {
        // 将字符串"Hello World"写入标准输出
Console.WriteLine("Hello World");
    }
}
```

`Console.WriteLine` 有多个重载。在此情况下，字符串 "Hello World" 是参数，执行时会将 "Hello World" 输出到标准输出流。其他重载可能会在写入流之前调用参数的 .ToString 方法。更多信息请参见 .NET Framework Documentation 。

Chapter 1: Getting started with .NET Framework

.NET

Version Release Date

1.0	2002-02-13
1.1	2003-04-24
2.0	2005-11-07
3.0	2006-11-06
3.5	2007-11-19
3.5 SP1	2008-08-11
4.0	2010-04-12
4.5	2012-08-15
4.5.1	2013-10-17
4.5.2	2014-05-05
4.6	2015-07-20
4.6.1	2015-11-17
4.6.2	2016-08-02
4.7	2017-04-05
4.7.1	2017-10-17

Compact Framework

Version Release Date

1.0	2000-01-01
2.0	2005-10-01
3.5	2007-11-19
3.7	2009-01-01
3.9	2013-06-01

Micro Framework

Version Release Date

4.2	2011-10-04
4.3	2012-12-04
4.4	2015-10-20

Section 1.1: Hello World in C#

```
using System;

class Program
{
    // The Main() function is the first function to be executed in a program
    static void Main()
    {
        // Write the string "Hello World to the standard out
        Console.WriteLine("Hello World");
    }
}
```

`Console.WriteLine` has several overloads. In this case, the string "Hello World" is the parameter, and it will output the "Hello World" to the standard out stream during execution. Other overloads may call the `.ToString` of the

有关详细信息，请参见 .NET Framework Documentation 。

在 .NET Fiddle 上的实时演示

C# 入门

第 1.2 节：F# 中的 Hello World

```
open System

[<EntryPoint>]
let main argv =
    printfn "Hello World"
    0
```

在 .NET Fiddle 上的实时演示

F# 入门

第1.3节：Visual Basic .NET中的Hello World

```
Imports System

Module Program
    Public Sub Main()
Console.WriteLine("Hello World")
    End Sub
End Module
```

.NET Fiddle上的实时演示

Visual Basic .NET简介

第1.4节：C++/CLI中的Hello World

```
using namespace System;

int main(array<String^>^ args)
{
Console::WriteLine("Hello World");
}
```

第1.5节：IL中的Hello World

```
.类 公共 自动 ansi 在字段初始化之前 Program
       继承自 [mscorlib]System.Object
{
  .方法 公共 hidebysig 静态 void  Main() cil 管理
  {
    .最大堆栈  8
IL_0000:  nop
IL_0001:  ldstr      "Hello World"
    IL_0006:  调用      void [mscorlib]System.Console::WriteLine(string)
   IL_000b:  nop
IL_000c:  返回
  }
```


argument before writing to the stream. See the .NET Framework Documentation for more information.

Live Demo in Action at .NET Fiddle

Introduction to C#

Section 1.2: Hello World in F#

```
open System

[<EntryPoint>]
let main argv =
    printfn "Hello World"
    0
```

Live Demo in Action at .NET Fiddle

Introduction to F#

Section 1.3: Hello World in Visual Basic .NET

```
Imports System

Module Program
    Public Sub Main()
        Console.WriteLine("Hello World")
    End Sub
End Module
```

Live Demo in Action at .NET Fiddle

Live Demo in Action at .NET Fiddle

Introduction to Visual Basic .NET

Section 1.4: Hello World in C++/CLI

```
using namespace System;

int main(array<String^>^ args)
{
    Console::WriteLine("Hello World");
}
```

Section 1.5: Hello World in IL

```
.class public auto ansi beforefieldinit Program
       extends [mscorlib]System.Object
{
  .method public hidebysig static void  Main() cil managed
  {
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr      "Hello World"
    IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ret
  }
```

```
  .方法 公共 hidebysig 特殊名称 rtspecialname
        实例 void  .ctor() cil 管理
  {
    .最大堆栈  8
IL_0000:  加载参数.0
IL_0001:  调用       实例 void [mscorlib]System.Object::.ctor()
    IL_0006:  返回
  }

}
```

第1.6节：PowerShell中的Hello World

```
Write-Host "Hello World"
```

PowerShell简介

第1.7节：Nemerle中的Hello World

```
System.Console.WriteLine("Hello World");
```

第1.8节：Python（IronPython）中的Hello World

```
print "Hello World"
```

```
import clr
from System import Console
Console.WriteLine("Hello World")
```

第1.9节：Oxygene中的Hello World

```
namespace HelloWorld;

interface

type
App = class
  public
    class method Main(args: array of String);
  end;

implementation

class method App.Main(args: array of String);
begin
Console.WriteLine('Hello World');
end;

end.
```

第1.10节：Boo语言中的Hello World

```
print "Hello World"
```



```
  .method public hidebysig specialname rtspecialname
        instance void  .ctor() cil managed
  {
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
  }

}
```

Section 1.6: Hello World in PowerShell

```
Write-Host "Hello World"
```

Introduction to PowerShell

Section 1.7: Hello World in Nemerle

```
System.Console.WriteLine("Hello World");
```

Section 1.8: Hello World in Python (IronPython)

```
print "Hello World"
```

```
import clr
from System import Console
Console.WriteLine("Hello World")
```

Section 1.9: Hello World in Oxygene

```
namespace HelloWorld;

interface

type
  App = class
  public
    class method Main(args: array of String);
  end;

implementation

class method App.Main(args: array of String);
begin
  Console.WriteLine('Hello World');
end;

end.
```

Section 1.10: Hello World in Boo

```
print "Hello World"
```

第2章：字符串

第2.1节：计数字符

如果你需要计算*字符*，那么，基于*备注*部分解释的原因，你不能简单地使用Length属性，因为它是`System.Char`数组的长度，这些不是字符而是代码单元（既不是Unicode代码点也不是字形）。正确的代码如下：

```
int length = text.EnumerateCharacters().Count();
```

一个小的优化可以专门为此目的重写EnumerateCharacters()扩展方法：

```
public static class StringExtensions
{
    public static int CountCharacters(this string text)
    {
        if (String.IsNullOrEmpty(text))
            return 0;

        int count = 0;
        var enumerator = StringInfo.GetTextElementEnumerator(text);
        while (enumerator.MoveNext())
            ++count;

        return count;
    }
}
```

第2.2节：计算不同字符数

如果你需要计算不同字符的数量，那么出于"备注"部分中解释的原因，你不能简单地使用`Length`属性，因为它是`System.Char`数组的长度，而这些不是字符，而是代码单元（既不是Unicode代码点，也不是字形）。例如，如果你简单地写`text.Distinct().Count()`，你将得到错误的结果，正确的代码是：

```
int distinctCharactersCount = text.EnumerateCharacters().Count();
```

更进一步的是**计算每个字符的出现次数**，如果性能不是问题，你可以简单地这样做（在此示例中不区分大小写）：

```
var frequencies = text.EnumerateCharacters()
    .GroupBy(x => x, StringComparer.CurrentCultureIgnoreCase)
    .Select(x => new { Character = x.Key, Count = x.Count() };
```

第2.3节：字符串与另一种编码的转换

.NET字符串包含`System.Char`（UTF-16代码单元）。如果你想保存（或管理）使用另一种编码的文本，你必须使用`System.Byte`数组。

转换由继承自`System.Text.Encoder`和`System.Text.Decoder`的类执行，它们共同可以实现与另一种编码之间的转换（从字节*X*编码的数组`byte[]`到UTF-16编码的`System.String`，反之亦然）。

因为编码器/解码器通常紧密配合工作，所以它们被组合在一个继承自

Chapter 2: Strings

Section 2.1: Count characters

If you need to count *characters* then, for the reasons explained in *Remarks* section, you can't simply use Length property because it's the length of the array of `System.Char` which are not characters but code-units (not Unicode code-points nor graphemes). Correct code is then:

```
int length = text.EnumerateCharacters().Count();
```

A small optimization may rewrite `EnumerateCharacters()` extension method specifically for this purpose:

```
public static class StringExtensions
{
    public static int CountCharacters(this string text)
    {
        if (String.IsNullOrEmpty(text))
            return 0;

        int count = 0;
        var enumerator = StringInfo.GetTextElementEnumerator(text);
        while (enumerator.MoveNext())
            ++count;

        return count;
    }
}
```

Section 2.2: Count distinct characters

If you need to count distinct characters then, for the reasons explained in *Remarks* section, you can't simply use `Length` property because it's the length of the array of `System.Char` which are not characters but code-units (not Unicode code-points nor graphemes). If, for example, you simply write `text.Distinct().Count()` you will get incorrect results, correct code:

```
int distinctCharactersCount = text.EnumerateCharacters().Count();
```

One step further is to **count occurrences of each character**, if performance aren't an issue you may simply do it like this (in this example regardless of case):

```
var frequencies = text.EnumerateCharacters()
    .GroupBy(x => x, StringComparer.CurrentCultureIgnoreCase)
    .Select(x => new { Character = x.Key, Count = x.Count() };
```

Section 2.3: Convert string to/from another encoding

.NET strings contain `System.Char` (UTF-16 code-units). If you want to save (or manage) text with another encoding you have to work with an array of `System.Byte`.

Conversions are performed by classes derived from `System.Text.Encoder` and `System.Text.Decoder` which, together, can convert to/from another encoding (from a byte *X* encoded array `byte[]` to an UTF-16 encoded `System.String` and vice-versa).

Because the encoder/decoder usually works very close to each other they're grouped together in a class derived

System.Text.Encoding的类中，派生类提供与流行编码（UTF-8、UTF-16等）之间的转换。

示例：
将字符串转换为UTF-8

```
byte[] data = Encoding.UTF8.GetBytes("这是我的文本");
```

将UTF-8数据转换为字符串

```
var text = Encoding.UTF8.GetString(data);
```

更改现有文本文件的编码

此代码将读取一个UTF-8编码的文本文件内容，并以UTF-16编码保存回去。请注意，如果文件较大，此代码并非最佳，因为它会将所有内容读入内存：

```
var content = File.ReadAllText(path, Encoding.UTF8);
File.WriteAllText(content, Encoding.UTF16);
```

第2.4节：字符串比较

尽管String是引用类型，==运算符比较的是字符串的值而非引用。

如你所知，string只是字符数组。但如果你认为字符串的相等检查和比较是逐字符进行的，那你就错了。此操作是特定于文化的（见下文备注）：某些字符序列可能根据culture被视为相等。

在通过比较两个字符串的Length属性来短路相等性检查之前请三思！

如果你需要更改默认行为，请使用接受额外StringComparison枚举值的String.Equals方法的重载。

第2.5节：计算字符出现次数

由于Remarks部分解释的原因，你不能简单地这样做（除非你想计算特定代码单元的出现次数）：

```
int count = text.Count(x => x == ch);
```

你需要一个更复杂的函数：

```
public static int CountOccurrencesOf(this string text, string character)
{
    return text.EnumerateCharacters()
        .Count(x => String.Equals(x, character, StringComparer.CurrentCulture));
}
```

请注意，字符串比较（与文化无关的字符比较不同）必须始终根据特定文化的规则进行。

第2.6节：将字符串拆分为固定长度的块

我们不能在任意位置拆分字符串（因为一个System.Char可能单独无效，因为它是一个组合字符或代理对的一部分），因此代码必须考虑到这一点（注意我所说的*length*是指

from System.Text.Encoding, derived classes offer conversions to/from popular encodings (UTF-8, UTF-16 and so on).

Examples:
Convert a string to UTF-8

```
byte[] data = Encoding.UTF8.GetBytes("This is my text");
```

Convert UTF-8 data to a string

```
var text = Encoding.UTF8.GetString(data);
```

Change encoding of an existing text file

This code will read content of an UTF-8 encoded text file and save it back encoded as UTF-16. Note that this code is not optimal if file is big because it will read all its content into memory:

```
var content = File.ReadAllText(path, Encoding.UTF8);
File.WriteAllText(content, Encoding.UTF16);
```

Section 2.4: Comparing strings

Despite String is a reference type == operator compares string values rather than references.

As you may know string is just an array of characters. But if you think that strings equality check and comparison is made character by character, you are mistaken. This operation is culture specific (see Remarks below): some character sequences can be treated as equal depending on the culture.

Think twice before short circuiting equality check by comparing Length properties of two strings!

Use overloads of String.Equals method which accept additional StringComparison enumeration value, if you need to change default behavior.

Section 2.5: Count occurrences of a character

Because of the reasons explained in *Remarks* section you can't simply do this (unless you want to count occurrences of a specific code-unit):

```
int count = text.Count(x => x == ch);
```

You need a more complex function:

```
public static int CountOccurrencesOf(this string text, string character)
{
    return text.EnumerateCharacters()
        .Count(x => String.Equals(x, character, StringComparer.CurrentCulture));
}
```

Note that string comparison (in contrast to character comparison which is culture invariant) must always be performed according to rules to a specific culture.

Section 2.6: Split string into fixed length blocks

We cannot break a string into arbitrary points (because a System.Char may not be valid alone because it's a combining character or part of a surrogate) then code must take that into account (note that with *length* I mean the

图形集的数量，而不是代码单元的数量）：

```
public static IEnumerable<string> Split(this string value, int desiredLength)
{
    var characters = StringInfo.GetTextElementEnumerator(value);
    while (characters.MoveNext())
        yield return String.Concat(Take(characters, desiredLength));
}

private static IEnumerable<string> Take(TextElementEnumerator enumerator, int count)
{
    for (int i = 0; i < count; ++i)
    {
        yield return (string)enumerator.Current;

        if (!enumerator.MoveNext())
            yield break;
    }
}
```

第2.7节：Object.ToString() 虚方法

.NET 中的所有内容都是对象，因此每种类型都有在 Object 类中定义的ToString() 方法，可以被重写。该方法的默认实现仅返回类型的名称：

```
public class Foo
{
}

var foo = new Foo();
Console.WriteLine(foo); // 输出 Foo
```

当将值与字符串连接时，ToString() 方法会被隐式调用：

```
public class Foo
{
    public override string ToString()
    {
        return "我是 Foo";
    }
}

var foo = new Foo();
Console.WriteLine("我是 bar 和 "+foo);// 输出 我是 bar 和 我是 Foo
```

该方法的结果也被调试工具广泛使用。如果出于某种原因，你不想重写此方法，但想自定义调试器显示你的类型值的方式，可以使用 DebuggerDisplayAttribute（MSDN）：

```
// [DebuggerDisplay("Person = FN {FirstName}, LN {LastName}")]
[DebuggerDisplay("Person = 名字 {"+nameof(Person.FirstName)+"}, 姓氏 {"+nameof(Person.LastName)+"}")]
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set;}
    // ...
}
```


number of *graphemes* not the number of *code-units*):

```
public static IEnumerable<string> Split(this string value, int desiredLength)
{
    var characters = StringInfo.GetTextElementEnumerator(value);
    while (characters.MoveNext())
        yield return String.Concat(Take(characters, desiredLength));
}

private static IEnumerable<string> Take(TextElementEnumerator enumerator, int count)
{
    for (int i = 0; i < count; ++i)
    {
        yield return (string)enumerator.Current;

        if (!enumerator.MoveNext())
            yield break;
    }
}
```

Section 2.7: Object.ToString() virtual method

Everything in .NET is an object, hence every type has `ToString()` method defined in `Object` class which can be overridden. Default implementation of this method just returns the name of the type:

```
public class Foo
{
}

var foo = new Foo();
Console.WriteLine(foo); // outputs Foo
```

`ToString()` is implicitly called when concatinating value with a string:

```
public class Foo
{
    public override string ToString()
    {
        return "I am Foo";
    }
}

var foo = new Foo();
Console.WriteLine("I am bar and "+foo);// outputs I am bar and I am Foo
```

The result of this method is also extensively used by debugging tools. If, for some reason, you do not want to override this method, but want to customize how debugger shows the value of your type, use DebuggerDisplay Attribute (MSDN):

```
// [DebuggerDisplay("Person = FN {FirstName}, LN {LastName}")]
[DebuggerDisplay("Person = FN {"+nameof(Person.FirstName)+"}, LN {"+nameof(Person.LastName)+"}")]
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set;}
    // ...
}
```

第2.8节：字符串的不可变性

字符串是不可变的。你无法更改已有的字符串。对字符串的任何操作都会创建一个具有新值的新字符串实例。这意味着如果你需要替换一个非常长字符串中的单个字符，内存将会为新值分配。

```
string veryLongString = ...
// 分配内存
string newString = veryLongString.Remove(0,1); // 移除字符串的第一个字符。
```

如果你需要对字符串值执行多次操作，请使用StringBuilder类，该类专为高效的字符串操作设计：

```
var sb = new StringBuilder(someInitialString);
foreach(var str in manyManyStrings)
{
sb.Append(str);
}
var finalString = sb.ToString();
```


Section 2.8: Immutability of strings

Strings are immutable. You just cannot change existing string. Any operation on the string crates a new instance of the string having new value. It means that if you need to replace a single character in a very long string, memory will be allocated for a new value.

```
string veryLongString = ...
// memory is allocated
string newString = veryLongString.Remove(0,1); // removes first character of the string.
```

If you need to perform many operations with string value, use `StringBuilder` class which is designed for efficient strings manipulation:

```
var sb = new StringBuilder(someInitialString);
foreach(var str in manyManyStrings)
{
    sb.Append(str);
}
var finalString = sb.ToString();
```

第3章：DateTime解析

第3.1节：ParseExact

```
var dateString = "2015-11-24";

var date = DateTime.ParseExact(dateString, "yyyy-MM-dd", null);
Console.WriteLine(date);
```

> 2015/11/24 0:00:00

注意，传入`CultureInfo.CurrentCulture`作为第三个参数与传入`null`是相同的。或者，你也可以传入特定的文化信息。

格式字符串

输入字符串可以是任何与格式字符串匹配的格式

```
var date = DateTime.ParseExact("24|201511", "dd|yyyyMM", null);
Console.WriteLine(date);
```

> 2015/11/24 0:00:00

任何非格式说明符的字符都被视为字面量

```
var date = DateTime.ParseExact("2015|11|24", "yyyy|MM|dd", null);
Console.WriteLine(date);
```

> 2015/11/24 0:00:00

格式说明符区分大小写

```
var date = DateTime.ParseExact("2015-01-24 11:11:30", "yyyy-mm-dd hh:MM:ss", null);
Console.WriteLine(date);
```

> 2015/11/24 11:01:30 上午

注意月份和分钟的值被解析到了错误的位置。

单字符格式字符串必须是标准格式之一

```
var date = DateTime.ParseExact("11/24/2015", "d", new CultureInfo("en-US"));
var date = DateTime.ParseExact("2015-11-24T10:15:45", "s", null);
var date = DateTime.ParseExact("2015-11-24 10:15:45Z", "u", null);
```

异常

ArgumentNullException

Chapter 3: DateTime parsing

Section 3.1: ParseExact

```
var dateString = "2015-11-24";

var date = DateTime.ParseExact(dateString, "yyyy-MM-dd", null);
Console.WriteLine(date);
```

> 11/24/2015 12:00:00 AM

Note that passing `CultureInfo.CurrentCulture` as the third parameter is identical to passing `null`. Or, you can pass a specific culture.

Format Strings

Input string can be in any format that matches the format string

```
var date = DateTime.ParseExact("24|201511", "dd|yyyyMM", null);
Console.WriteLine(date);
```

> 11/24/2015 12:00:00 AM

Any characters that are not format specifiers are treated as literals

```
var date = DateTime.ParseExact("2015|11|24", "yyyy|MM|dd", null);
Console.WriteLine(date);
```

> 11/24/2015 12:00:00 AM

Case matters for format specifiers

```
var date = DateTime.ParseExact("2015-01-24 11:11:30", "yyyy-mm-dd hh:MM:ss", null);
Console.WriteLine(date);
```

> 11/24/2015 11:01:30 AM

Note that the month and minute values were parsed into the wrong destinations.

Single-character format strings must be one of the standard formats

```
var date = DateTime.ParseExact("11/24/2015", "d", new CultureInfo("en-US"));
var date = DateTime.ParseExact("2015-11-24T10:15:45", "s", null);
var date = DateTime.ParseExact("2015-11-24 10:15:45Z", "u", null);
```

Exceptions

ArgumentNullException

```
var date = DateTime.ParseExact(null, "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", null, null);
```

FormatException

```
var date = DateTime.ParseExact("", "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", "", null);
var date = DateTime.ParseExact("2015-0C-24", "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", "yyyy-QQ-dd", null);

// 单字符格式字符串必须是标准格式之一
var date = DateTime.ParseExact("2015-11-24", "q", null);

// 格式字符串必须与输入完全匹配*（见下一节）
var date = DateTime.ParseExact("2015-11-24", "d", null); // 对大多数文化期望为 11/24/2015 或 24/11/2015
```

处理多种可能的格式

```
var date = DateTime.ParseExact("2015-11-24T10:15:45",
   new [] { "s", "t", "u", "yyyy-MM-dd" }, // 只要输入匹配其中一个格式就会成功
  CultureInfo.CurrentCulture, DateTimeStyles.None);
```

处理文化差异

```
var dateString = "10/11/2015";
var date = DateTime.ParseExact(dateString, "d", new CultureInfo("en-US"));
Console.WriteLine("Day: {0}; Month: {1}", date.Day, date.Month);
```

> Day: 11; Month: 10

```
date = DateTime.ParseExact(dateString, "d", new CultureInfo("en-GB"));
Console.WriteLine("Day: {0}; Month: {1}", date.Day, date.Month);
```

> Day: 10; Month: 11

第3.2节：TryParse

此方法接受一个字符串作为输入，尝试将其解析为DateTime类型，并返回一个布尔值，指示解析成功或失败。如果调用成功，作为out参数传入的变量将被赋值为解析结果。

如果解析失败，作为out参数传入的变量将被设置为默认值DateTime.MinValue。

TryParse(string, out DateTime)

```
DateTime parsedValue;

if (DateTime.TryParse("monkey", out parsedValue))
{
Console.WriteLine("显然，'monkey' 是一个日期/时间值。谁知道呢？");
}
```

此方法尝试根据系统区域设置和已知格式（如


```
var date = DateTime.ParseExact(null, "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", null, null);
```

FormatException

```
var date = DateTime.ParseExact("", "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", "", null);
var date = DateTime.ParseExact("2015-0C-24", "yyyy-MM-dd", null);
var date = DateTime.ParseExact("2015-11-24", "yyyy-QQ-dd", null);

// Single-character format strings must be one of the standard formats
var date = DateTime.ParseExact("2015-11-24", "q", null);

// Format strings must match the input exactly* (see next section)
var date = DateTime.ParseExact("2015-11-24", "d", null); // Expects 11/24/2015 or 24/11/2015 for most cultures
```

Handling multiple possible formats

```
var date = DateTime.ParseExact("2015-11-24T10:15:45",
  new [] { "s", "t", "u", "yyyy-MM-dd" }, // Will succeed as long as input matches one of these
  CultureInfo.CurrentCulture, DateTimeStyles.None);
```

Handling culture differences

```
var dateString = "10/11/2015";
var date = DateTime.ParseExact(dateString, "d", new CultureInfo("en-US"));
Console.WriteLine("Day: {0}; Month: {1}", date.Day, date.Month);
```

> Day: 11; Month: 10

```
date = DateTime.ParseExact(dateString, "d", new CultureInfo("en-GB"));
Console.WriteLine("Day: {0}; Month: {1}", date.Day, date.Month);
```

> Day: 10; Month: 11

Section 3.2: TryParse

This method accepts a string as input, attempts to parse it into a `DateTime`, and returns a Boolean result indicating success or failure. If the call succeeds, the variable passed as the `out` parameter is populated with the parsed result.

If the parse fails, the variable passed as the `out` parameter is set to the default value, `DateTime.MinValue`.

TryParse(string, out DateTime)

```
DateTime parsedValue;

if (DateTime.TryParse("monkey", out parsedValue))
{
   Console.WriteLine("Apparently, 'monkey' is a date/time value. Who knew?");
}
```

This method attempts to parse the input string based on the system regional settings and known formats such as

ISO 8601 及其他常见格式）解析输入字符串。

```
DateTime.TryParse("11/24/2015 14:28:42", out parsedValue); // true
DateTime.TryParse("2015-11-24 14:28:42", out parsedValue); // true
DateTime.TryParse("2015-11-24T14:28:42", out parsedValue); // true
DateTime.TryParse("Sat, 24 Nov 2015 14:28:42", out parsedValue); // true
```

由于此方法不接受文化信息，它使用系统区域设置。这可能导致意外结果。

```
// 系统设置为 en-US 文化
bool result = DateTime.TryParse("24/11/2015", out parsedValue);
Console.WriteLine(result);
```

> False

```
// 系统设置为 en-GB 文化
bool result = DateTime.TryParse("11/24/2015", out parsedValue);
Console.WriteLine(result);
```

> False

```
// 系统设置为 en-GB 文化
bool result = DateTime.TryParse("10/11/2015", out parsedValue);
Console.WriteLine(result);
```

> True

请注意，如果你在美国，你可能会惊讶地发现解析结果是11月10日，而不是10月11日。

TryParse(string, IFormatProvider, DateTimeStyles, out DateTime)

```
if (DateTime.TryParse(" monkey ", new CultureInfo("en-GB"),
    DateTimeStyles.AllowLeadingWhite | DateTimeStyles.AllowTrailingWhite, out parsedValue)
{
Console.WriteLine("显然，' monkey ' 是一个日期/时间值。谁能想到呢？");
}
```

与其同类方法不同，此重载允许指定特定的文化和样式。传递 null 给 IFormatProvider 参数将使用系统文化。

异常

请注意，在某些条件下此方法可能会抛出异常。这些条件与此重载引入的参数有关：IFormatProvider 和 DateTimeStyles。

- `NotSupportedException`: `IFormatProvider` 指定了中性文化
- `ArgumentException`: `DateTimeStyles` 不是有效选项，或包含不兼容的标志，如 `AssumeLocal` 和 `AssumeUniversal`。

ISO 8601 and other common formats.

```
DateTime.TryParse("11/24/2015 14:28:42", out parsedValue); // true
DateTime.TryParse("2015-11-24 14:28:42", out parsedValue); // true
DateTime.TryParse("2015-11-24T14:28:42", out parsedValue); // true
DateTime.TryParse("Sat, 24 Nov 2015 14:28:42", out parsedValue); // true
```

Since this method does not accept culture info, it uses the system locale. This can lead to unexpected results.

```
// System set to en-US culture
bool result = DateTime.TryParse("24/11/2015", out parsedValue);
Console.WriteLine(result);
```

> False

```
// System set to en-GB culture
bool result = DateTime.TryParse("11/24/2015", out parsedValue);
Console.WriteLine(result);
```

> False

```
// System set to en-GB culture
bool result = DateTime.TryParse("10/11/2015", out parsedValue);
Console.WriteLine(result);
```

> True

Note that if you are in the US, you might be surprised that the parsed result is November 10, not October 11.

TryParse(string, IFormatProvider, DateTimeStyles, out DateTime)

```
if (DateTime.TryParse(" monkey ", new CultureInfo("en-GB"),
    DateTimeStyles.AllowLeadingWhite | DateTimeStyles.AllowTrailingWhite, out parsedValue)
{
    Console.WriteLine("Apparently, ' monkey ' is a date/time value. Who knew?");
}
```

Unlike its sibling method, this overload allows a specific culture and style(s) to be specified. Passing `null` for the `IFormatProvider` parameter uses the system culture.

Exceptions

Note that it is possible for this method to throw an exception under certain conditions. These relate to the parameters introduced for this overload: `IFormatProvider` and `DateTimeStyles`.

- `NotSupportedException`: `IFormatProvider` specifies a neutral culture
- `ArgumentException`: `DateTimeStyles` is not a valid option, or contains incompatible flags such as `AssumeLocal` and `AssumeUniversal`.

第3.3节：TryParseExact

此方法的行为类似于TryParse和ParseExact的结合：它允许指定自定义格式，并且在解析失败时返回一个布尔值结果表示成功或失败，而不是抛出异常。

TryParseExact(string, string, IFormatProvider, DateTimeStyles, out DateTime)

此重载尝试根据特定格式解析输入字符串。输入字符串必须匹配该格式才能被解析。

```
DateTime.TryParseExact("11242015", "MMddyyyy", null, DateTimeStyles.None, out parsedValue); // true
```

TryParseExact(string, string[], IFormatProvider, DateTimeStyles, out DateTime)

此重载尝试根据一组格式数组解析输入字符串。输入字符串必须至少匹配其中一个格式才能被解析。

```
DateTime.TryParseExact("11242015", new [] { "yyyy-MM-dd", "MMddyyyy" }, null, DateTimeStyles.None,
out parsedValue); // true
```


Section 3.3: TryParseExact

This method behaves as a combination of `TryParse` and `ParseExact`: It allows custom format(s) to be specified, and returns a Boolean result indicating success or failure rather than throwing an exception if the parse fails.

TryParseExact(string, string, IFormatProvider, DateTimeStyles, out DateTime)

This overload attempts to parse the input string against a specific format. The input string must match that format in order to be parsed.

```
DateTime.TryParseExact("11242015", "MMddyyyy", null, DateTimeStyles.None, out parsedValue); // true
```

TryParseExact(string, string[], IFormatProvider, DateTimeStyles, out DateTime)

This overload attempts to parse the input string against an array of formats. The input string must match at least one format in order to be parsed.

```
DateTime.TryParseExact("11242015", new [] { "yyyy-MM-dd", "MMddyyyy" }, null, DateTimeStyles.None,
out parsedValue); // true
```

第4章：字典

第4.1节：使用集合初始化器初始化字典

```
// 翻译为 `dict.Add(1, "First")` 等。
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};

// 翻译为 `dict[1] = "First"` 等。
// 适用于 C# 6.0。
var dict = new Dictionary<int, string>()
{
    [1] = "First",
    [2] = "Second",
    [3] = "Third"
};
```

第4.2节：向字典添加元素

```
Dictionary<int, string> dict = new Dictionary<int, string>();
dict.Add(1, "First");
dict.Add(2, "Second");

// 安全添加元素（检查确保元素不存在，否则会抛出异常）
if(!dict.ContainsKey(3))
{
dict.Add(3, "Third");
}
```

或者也可以通过索引器添加/设置。（索引器在内部看起来像属性，具有 get 和 set，但接受一个在括号中指定的任意类型的参数）：

```
Dictionary<int, string> dict = new Dictionary<int, string>();
dict[1] = "First";
dict[2] = "Second";
dict[3] = "Third";
```

与Add方法不同，如果字典中已包含该键，Add 会抛出异常，而索引器则直接替换现有的值。

要使用线程安全的字典，请使用ConcurrentDictionary<TKey, TValue>：

```
var dict = new ConcurrentDictionary<int, string>();
dict.AddOrUpdate(1, "First", (oldKey, oldValue) => "First");
```

第4.3节：从字典中获取值

给定以下初始化代码：

```
var dict = new Dictionary<int, string>()
{
    { 1, "第一" },
```


Chapter 4: Dictionaries

Section 4.1: Initializing a Dictionary with a Collection Initializer

```
// Translates to `dict.Add(1, "First")` etc.
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};

// Translates to `dict[1] = "First"` etc.
// Works in C# 6.0.
var dict = new Dictionary<int, string>()
{
    [1] = "First",
    [2] = "Second",
    [3] = "Third"
};
```

Section 4.2: Adding to a Dictionary

```
Dictionary<int, string> dict = new Dictionary<int, string>();
dict.Add(1, "First");
dict.Add(2, "Second");

// To safely add items (check to ensure item does not already exist - would throw)
if(!dict.ContainsKey(3))
{
   dict.Add(3, "Third");
}
```

Alternatively they can be added/set via the an indexer. (An indexer internally looks like a property, having a get and set, but takes a parameter of any type which is specified between the brackets) :

```
Dictionary<int, string> dict = new Dictionary<int, string>();
dict[1] = "First";
dict[2] = "Second";
dict[3] = "Third";
```

Unlike the `Add` method which throws an exception, if a key is already contained in the dictionary, the indexer just replaces the existing value.

For thread-safe dictionary use `ConcurrentDictionary<TKey, TValue>`:

```
var dict = new ConcurrentDictionary<int, string>();
dict.AddOrUpdate(1, "First", (oldKey, oldValue) => "First");
```

Section 4.3: Getting a value from a dictionary

Given this setup code:

```
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
```

```
    { 2, "Second" },
    { 3, "Third" }
};
```

您可能想读取键为1的条目的值。如果键不存在，获取值时会抛出KeyNotFoundException，因此您可能想先用ContainsKey检查一下：

```
if (dict.ContainsKey(1))
    Console.WriteLine(dict[1]);
```

这有一个缺点：您会对字典进行两次查找（一次检查是否存在，一次读取值）。对于大型字典，这可能影响性能。幸运的是，这两个操作可以合并执行：

```
string value;
if (dict.TryGetValue(1, out value))
    Console.WriteLine(value);
```

第4.4节：创建一个键不区分大小写的 Dictionary<string, T>

```
var MyDict = new Dictionary<string,T>(StringComparison.InvariantCultureIgnoreCase)
```

第4.5节：IEnumerable 转 Dictionary（.NET 3.5及以上）

从一个IEnumerable<T>创建一个Dictionary<TKey, TValue>：

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
public class 水果
{
    public int Id { get; set; }
    public string 名称 { get; set; }
}
```

```
var fruits = new[]
{
    new 水果 { Id = 8 , 名称 = "Apple" },
    new 水果 { Id = 3 , 名称 = "Banana" },
    new 水果 { Id = 7 , 名称 = "Mango" },
};


// Dictionary<int, string>                  键      值
var dictionary = fruits.ToDictionary(x => x.Id, x => x.名称);
```

第4.6节：枚举字典

你可以通过三种方式枚举字典：

使用键值对


```
    { 2, "Second" },
    { 3, "Third" }
};
```

You may want to read the value for the entry with key 1. If key doesn't exist getting a value will throw `KeyNotFoundException`, so you may want to first check for that with `ContainsKey`:

```
if (dict.ContainsKey(1))
    Console.WriteLine(dict[1]);
```

This has one disadvantage: you will search through your dictionary twice (once to check for existence and one to read the value). For a large dictionary this can impact performance. Fortunately both operations can be performed together:

```
string value;
if (dict.TryGetValue(1, out value))
    Console.WriteLine(value);
```

Section 4.4: Make a Dictionary<string, T> with Case-Insensivitve keys

```
var MyDict = new Dictionary<string,T>(StringComparison.InvariantCultureIgnoreCase)
```

Section 4.5: IEnumerable to Dictionary (≥ .NET 3.5)

Create a Dictionary<TKey, TValue> from an IEnumerable<T>:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
public class Fruits
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

```
var fruits = new[]
{
    new Fruits { Id = 8 , Name = "Apple" },
    new Fruits { Id = 3 , Name = "Banana" },
    new Fruits { Id = 7 , Name = "Mango" },
};


// Dictionary<int, string>                  key      value
var dictionary = fruits.ToDictionary(x => x.Id, x => x.Name);
```

Section 4.6: Enumerating a Dictionary

You can enumerate through a Dictionary in one of 3 ways:

Using KeyValue pairs

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(KeyValuePair<int, string> kvp in dict)
{
Console.WriteLine("Key : " + kvp.Key.ToString() + ", Value : " + kvp.Value);
}
```

使用键

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(int key in dict.Keys)
{
Console.WriteLine("Key : " + key.ToString() + ", Value : " + dict[key]);
}
```

使用值

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(string s in dict.Values)
{
Console.WriteLine("Value : " + s);
}
```

第4.7节：ConcurrentDictionary<TKey，TValue>（来自.NET 4.0）

> 表示一个线程安全的键/值对集合，可以被多个线程同时访问。

创建实例

创建实例的方式与Dictionary<TKey，TValue>基本相同，例如：

```
var dict = new ConcurrentDictionary<int, string>();
```

添加或更新

你可能会感到惊讶，没有Add方法，而是有两个重载的AddOrUpdate方法：

(1) AddOrUpdate(TKey key, TValue, Func<TKey, TValue, TValue> addValue) - 如果键不存在，则添加键/值对；如果键已存在，则使用指定的函数更新键/值对。

(2) AddOrUpdate(TKey key, Func<TKey, TValue> addValue, Func<TKey, TValue, TValue> updateValueFactory) *- 使用指定的函数，如果键不存在则添加键/值对，如果键已存在则更新键/值对。*

添加或更新值，无论给定键之前是否已有值（1）：

```
string addedValue = dict.AddOrUpdate(1, "First", (updateKey, valueOld) => "First");
```

添加或更新一个值，但现在基于之前的值在更新时修改该值（1）：

```
string addedValue2 = dict.AddOrUpdate(1, "First", (updateKey, valueOld) => $"{valueOld} Updated");
```



```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(KeyValuePair<int, string> kvp in dict)
{
   Console.WriteLine("Key : " + kvp.Key.ToString() + ", Value : " + kvp.Value);
}
```

Using Keys

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(int key in dict.Keys)
{
    Console.WriteLine("Key : " + key.ToString() + ", Value : " + dict[key]);
}
```

Using Values

```
Dictionary<int, string> dict = new Dictionary<int, string>();
foreach(string s in dict.Values)
{
    Console.WriteLine("Value : " + s);
}
```

Section 4.7: ConcurrentDictionary<TKey,TValue> (from .NET 4.0)

> Represents a thread-safe collection of key/value pairs that can be accessed by multiple threads concurrently.

Creating an instance

Creating an instance works pretty much the same way as with `Dictionary<TKey, TValue>`, e.g.:

```
var dict = new ConcurrentDictionary<int, string>();
```

Adding or Updating

You might be surprised, that there is no `Add` method, but instead there is `AddOrUpdate` with 2 overloads:

(1) `AddOrUpdate(TKey key, TValue, Func<TKey, TValue, TValue> addValue)` - *Adds a key/value pair if the key does not already exist, or updates a key/value pair by using the specified function if the key already exists.*

(2) `AddOrUpdate(TKey key, Func<TKey, TValue> addValue, Func<TKey, TValue, TValue> updateValueFactory)` *- Uses the specified functions to add a key/value pair to the if the key does not already exist, or to update a key/value pair if the key already exists.*

Adding or updating a value, no matter what was the value if it was already present for given key (1):

```
string addedValue = dict.AddOrUpdate(1, "First", (updateKey, valueOld) => "First");
```

Adding or updating a value, but now altering the value in update, based on the previous value (1):

```
string addedValue2 = dict.AddOrUpdate(1, "First", (updateKey, valueOld) => $"{valueOld} Updated");
```

使用重载方法 (2)，我们也可以使用工厂添加新值：

```
string addedValue3 = dict.AddOrUpdate(1, (key) => key == 1 ? "First" : "Not First", (updateKey, valueOld) => $"{valueOld} Updated");
```

获取值

获取值的方式与Dictionary<TKey,TValue>相同：

```
string value = null;
bool success = dict.TryGetValue(1, out value);
```

获取或添加值

有两个方法重载，可以以线程安全的方式获取或添加一个值。

获取键为2的值，如果键不存在则添加值“Second”：

```
string theValue = dict.GetOrAdd(2, "Second");
```

如果值不存在，使用工厂方法添加值：

```
string theValue2 = dict.GetOrAdd(2, (key) => key == 2 ? "Second" : "Not Second." );
```

第4.8节：字典转列表

创建 KeyValuePair 列表：

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<KeyValuePair<int, int>> list = new List<KeyValuePair<int, int>>();
list.AddRange(dictionary);
```

创建键列表：

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<int> list = new List<int>();
list.AddRange(dictionary.Keys);
```

创建一个数值列表：

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<int> list = new List<int>();
list.AddRange(dictionary.Values);
```

第4.9节：从字典中移除

给定以下初始化代码：

```
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};
```


Using the overload (2) we can also add new value using a factory:

```
string addedValue3 = dict.AddOrUpdate(1, (key) => key == 1 ? "First" : "Not First", (updateKey, valueOld) => $"{valueOld} Updated");
```

Getting value

Getting a value is the same as with the `Dictionary<TKey,TValue>`:

```
string value = null;
bool success = dict.TryGetValue(1, out value);
```

Getting or Adding a value

There are two mehod overloads, that will **get or add** a value in a thread-safe manner.

Get value with key 2, or add value "Second" if the key is not present:

```
string theValue = dict.GetOrAdd(2, "Second");
```

Using a factory for adding a value, if value is not present:

```
string theValue2 = dict.GetOrAdd(2, (key) => key == 2 ? "Second" : "Not Second." );
```

Section 4.8: Dictionary to List

Creating a list of KeyValuePair:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<KeyValuePair<int, int>> list = new List<KeyValuePair<int, int>>();
list.AddRange(dictionary);
```

Creating a list of keys:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<int> list = new List<int>();
list.AddRange(dictionary.Keys);
```

Creating a list of values:

```
Dictionary<int, int> dictionary = new Dictionary<int, int>();
List<int> list = new List<int>();
list.AddRange(dictionary.Values);
```

Section 4.9: Removing from a Dictionary

Given this setup code:

```
var dict = new Dictionary<int, string>()
{
    { 1, "First" },
    { 2, "Second" },
    { 3, "Third" }
};
```

使用Remove方法移除一个键及其关联的值。

```
bool wasRemoved = dict.Remove(2);
```

执行此代码会从字典中移除键2及其对应的值。Remove返回一个布尔值，指示是否找到并移除了指定的键。如果字典中不存在该键，字典不会被修改，且返回false（不会抛出异常）。

尝试通过将键的值设置为null来移除键是错误的。

```
dict[2] = null; // 错误的移除方式！
```

这不会移除该键，只会将之前的值替换为null。

要移除字典中的所有键和值，请使用Clear方法。

```
dict.Clear();
```

执行Clear后，字典的Count将为0，但内部容量保持不变。

第4.10节：ContainsKey(TKey)

要检查Dictionary中是否存在特定键，可以调用方法ContainsKey(TKey)，并提供TKey类型的键。当字典中存在该键时，该方法返回bool值。示例：

```
var dictionary = new Dictionary<string, Customer>()
{
   {"F1", new Customer() { FirstName = "Felipe", ... } },
   {"C2", new Customer() { FirstName = "Carl", ... } },
   {"J7", new Customer() { FirstName = "John", ... } },
   {"M5", new Customer() { FirstName = "Mary", ... } },
};
```

并检查C2是否存在于字典中：

```
if (dictionary.ContainsKey("C2"))
{
   // 存在
}
```

ContainsKey方法可用于泛型版本的Dictionary<TKey，TValue>。

第4.11节：使用Lazy'1增强的ConcurrentDictionary减少重复计算

问题

ConcurrentDictionary 在即时返回缓存中已存在的键时表现出色，主要是无锁的，并且在细粒度级别上进行竞争。但如果对象的创建非常昂贵，超过了上下文切换的成本，并且发生了一些缓存未命中的情况，该怎么办？

如果多个线程请求相同的键，碰撞操作产生的对象中最终只有一个会被添加到集合中，其他的对象将被丢弃，浪费了创建对象的 CPU 资源和临时存储对象的内存资源。其他资源也可能被浪费。这是非常糟糕的。

Use the `Remove` method to remove a key and its associated value.

```
bool wasRemoved = dict.Remove(2);
```

Executing this code removes the key 2 and it's value from the dictionary. `Remove` returns a boolean value indicating whether the specified key was found and removed from the dictionary. If the key does not exist in the dictionary, nothing is removed from the dictionary, and false is returned (no exception is thrown).

It's **incorrect** to try and remove a key by setting the value for the key to `null`.

```
dict[2] = null; // WRONG WAY TO REMOVE!
```

This will not remove the key. It will just replace the previous value with a value of `null`.

To remove all keys and values from a dictionary, use the `Clear` method.

```
dict.Clear();
```

After executing `Clear` the dictionary's `Count` will be 0, but the internal capacity remains unchanged.

Section 4.10: ContainsKey(TKey)

To check if a `Dictionary` has an specifique key, you can call the method `ContainsKey(TKey)` and provide the key of `TKey` type. The method returns a `bool` value when the key exists on the dictionary. For sample:

```
var dictionary = new Dictionary<string, Customer>()
{
   {"F1", new Customer() { FirstName = "Felipe", ... } },
   {"C2", new Customer() { FirstName = "Carl", ... } },
   {"J7", new Customer() { FirstName = "John", ... } },
   {"M5", new Customer() { FirstName = "Mary", ... } },
};
```

And check if a `C2` exists on the Dictionary:

```
if (dictionary.ContainsKey("C2"))
{
   // exists
}
```

The ContainsKey method is available on the generic version `Dictionary<TKey, TValue>`.

Section 4.11: ConcurrentDictionary augmented with Lazy'1 reduces duplicated computation

Problem

ConcurrentDictionary shines when it comes to instantly returning of existing keys from cache, mostly lock free, and contending on a granular level. But what if the object creation is really expensive, outweighing the cost of context switching, and some cache misses occur?

If the same key is requested from multiple threads, one of the objects resulting from colliding operations will be eventually added to the collection, and the others will be thrown away, wasting the CPU resource to create the object and memory resource to store the object temporarily. Other resources could be wasted as well. This is really bad.

解决方案

我们可以将ConcurrentDictionary<TKey, TValue>与Lazy<TValue>结合使用。其思路是，ConcurrentDictionary 的 GetOrAdd 方法只能返回实际添加到集合中的值。失败的 Lazy 对象也可能被浪费，但这没什么大问题，因为 Lazy 对象本身相对不昂贵。失败的 Lazy 的 Value 属性永远不会被请求，因为我们聪明地只请求实际添加到集合中的那个 Lazy 对象的 Value 属性——即从 GetOrAdd 方法返回的那个对象：

```
public static class ConcurrentDictionaryExtensions
{
    public static TValue GetOrCreateLazy<TKey, TValue>(
        this ConcurrentDictionary<TKey, Lazy<TValue>> d,
       TKey key,
Func<TKey, TValue> factory)
    {
        return
d.GetOrAdd(
                key,
key1 =>
                    new Lazy<TValue>(() => factory(key1),
                    LazyThreadSafetyMode.ExecutionAndPublication)).Value;
    }
}
```

XmlSerializer 对象的缓存可能特别昂贵，且在应用程序启动时也会有大量争用。此外，还有更重要的问题：如果这些是自定义序列化器，整个进程生命周期内还会存在内存泄漏。在这种情况下，ConcurrentDictionary 唯一的好处是，在进程的其余生命周期内不会有锁，但应用程序启动时间和内存使用将不可接受。这正是我们需要使用带有 Lazy 的 ConcurrentDictionary 的原因：

```
private ConcurrentDictionary<Type, Lazy<XmlSerializer>> _serializers =
    new ConcurrentDictionary<Type, Lazy<XmlSerializer>>();

public XmlSerializer GetSerialier(Type t)
{
    return _serializers.GetOrCreateLazy(t, BuildSerializer);
}

private XmlSerializer BuildSerializer(Type t)
{
    throw new NotImplementedException("and this is a homework");
}
```


Solution

We can combine `ConcurrentDictionary<TKey, TValue>` with `Lazy<TValue>`. The idea is that ConcurrentDictionary GetOrAdd method can only return the value which was actually added to the collection. The loosing Lazy objects could be wasted in this case too, but that's not much problem, as the Lazy object itself is relatively unexpensive. The Value property of the losing Lazy is never requested, because we are smart to only request the Value property of the one actually added to the collection - the one returned from the GetOrAdd method:

```
public static class ConcurrentDictionaryExtensions
{
    public static TValue GetOrCreateLazy<TKey, TValue>(
        this ConcurrentDictionary<TKey, Lazy<TValue>> d,
        TKey key,
        Func<TKey, TValue> factory)
    {
        return
            d.GetOrAdd(
                key,
                key1 =>
                    new Lazy<TValue>(() => factory(key1),
                    LazyThreadSafetyMode.ExecutionAndPublication)).Value;
    }
}
```

Caching of XmlSerializer objects can be particularly expensive, and there is a lot of contention at the application startup too. And there is more to this: if those are custom serializers, there will be a memory leak too for the rest of the process lifecycle. The only benefit of the ConcurrentDictionary in this case is that for the rest of the process lifecycle there will be no locks, but application startup and memory usage would be inacceptable. This is a job for our ConcurrentDictionary, augmented with Lazy:

```
private ConcurrentDictionary<Type, Lazy<XmlSerializer>> _serializers =
    new ConcurrentDictionary<Type, Lazy<XmlSerializer>>();

public XmlSerializer GetSerialier(Type t)
{
    return _serializers.GetOrCreateLazy(t, BuildSerializer);
}

private XmlSerializer BuildSerializer(Type t)
{
    throw new NotImplementedException("and this is a homework");
}
```

第5章：集合

第5.1节：使用集合初始化器

某些集合类型可以在声明时初始化。例如，以下语句创建并初始化了包含一些整数的numbers：

```
List<int> numbers = new List<int>(){10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1};
```

在内部，C# 编译器实际上将此初始化转换为对 Add 方法的一系列调用。因此，您只能对实际支持Add方法的集合使用此语法。

> Stack<T>和Queue<T>类不支持此语法。

对于像Dictionary<TKey, TValue>这样接受键/值对的复杂集合，您可以在初始化列表中将每个键/值对指定为匿名类型。

```
Dictionary<int, string> employee = new Dictionary<int, string>()
     {{44, "John"}, {45, "Bob"}, {47, "James"}, {48, "Franklin"}};
```

每对中的第一个元素是键，第二个是值。

第5.2节：栈

.Net 中有一个用于管理值的Stack集合，它采用LIFO（后进先出）的概念。栈的基本方法是Push(T item)，用于向栈中添加元素，以及Pop()，用于获取最后添加的元素并将其从栈中移除。泛型版本可以像下面的代码一样用于字符串队列。

首先，添加命名空间：

```
using System.Collections.Generic;
```

并使用它：

```
Stack<string> stack = new Stack<string>();
stack.Push("John");
stack.Push("Paul");
stack.Push("George");
stack.Push("Ringo");

string value;
value = stack.Pop(); // 返回 Ringo
value = stack.Pop(); // 返回 George
value = stack.Pop(); // 返回 Paul
value = stack.Pop(); // 返回 John
```

有一个非泛型版本的类型，它可以处理对象。

命名空间是：

```
using System.Collections;
```


Chapter 5: Collections

Section 5.1: Using collection initializers

Some collection types can be initialized at the declaration time. For example, the following statement creates and initializes the `numbers` with some integers:

```
List<int> numbers = new List<int>(){10, 9, 8, 7, 7, 6, 5, 10, 4, 3, 2, 1};
```

Internally, the C# compiler actually converts this initialization to a series of calls to the Add method. Consequently, you can use this syntax only for collections that actually support the `Add` method.

> The `Stack<T>` and `Queue<T>` classes do not support it.

For complex collections such as the `Dictionary<TKey, TValue>` class, that take key/value pairs, you can specify each key/value pair as an anonymous type in the initializer list.

```
Dictionary<int, string> employee = new Dictionary<int, string>()
     {{44, "John"}, {45, "Bob"}, {47, "James"}, {48, "Franklin"}};
```

The first item in each pair is the key, and the second is the value.

Section 5.2: Stack

There is a collection in .Net used to manage values in a `Stack` that uses the LIFO (last-in first-out) concept. The basics of stacks is the method `Push(T item)` which is used to add elements in the stack and `Pop()` which is used to get the last element added and remove it from the stack. The generic version can be used like the following code for a queue of strings.

First, add the namespace:

```
using System.Collections.Generic;
```

and use it:

```
Stack<string> stack = new Stack<string>();
stack.Push("John");
stack.Push("Paul");
stack.Push("George");
stack.Push("Ringo");

string value;
value = stack.Pop(); // return Ringo
value = stack.Pop(); // return George
value = stack.Pop(); // return Paul
value = stack.Pop(); // return John
```

There is a non generic version of the type, which works with objects.

The namespace is:

```
using System.Collections;
```

非泛型栈的代码示例：

```
Stack stack = new Stack();
stack.Push("Hello World"); // 字符串
stack.Push(5); // 整数
stack.Push(1d); // double
stack.Push(true); // bool
stack.Push(new Product()); // Product 对象

object value;
value = stack.Pop(); // 返回 Product（产品类型）
value = stack.Pop(); // 返回 true（布尔值）
value = stack.Pop(); // 返回 1d（双精度浮点数）
value = stack.Pop(); // 返回 5（整数）
value = stack.Pop(); // 返回 Hello World（字符串）
```

还有一种称为Peek()的方法，它返回最后添加的元素，但不会将其从Stack中移除。

```
Stack<int> stack = new Stack<int>();
stack.Push(10);
stack.Push(20);

var lastValueAdded = stack.Peek(); // 20
```

可以对栈中的元素进行迭代，并且会遵循栈的顺序（后进先出）。

```
Stack<int> stack = new Stack<int>();
stack.Push(10);
stack.Push(20);
stack.Push(30);
stack.Push(40);
stack.Push(50);

foreach (int element in stack)
{
Console.WriteLine(element);
}
```

输出（未移除时）：

```

```

想要初始化一个`List<Model>`，有几种不同的方式可以实现。

And a code sample of non generic stack:

```
Stack stack = new Stack();
stack.Push("Hello World"); // string
stack.Push(5); // int
stack.Push(1d); // double
stack.Push(true); // bool
stack.Push(new Product()); // Product object

object value;
value = stack.Pop(); // return Product (Product type)
value = stack.Pop(); // return true (bool)
value = stack.Pop(); // return 1d (double)
value = stack.Pop(); // return 5 (int)
value = stack.Pop(); // return Hello World (string)
```

There is also a method called Peek() which returns the last element added but without removing it from the `Stack`.

```
Stack<int> stack = new Stack<int>();
stack.Push(10);
stack.Push(20);

var lastValueAdded = stack.Peek(); // 20
```

It is possible to iterate on the elements on the stack and it will respect the order of the stack (LIFO).

```
Stack<int> stack = new Stack<int>();
stack.Push(10);
stack.Push(20);
stack.Push(30);
stack.Push(40);
stack.Push(50);

foreach (int element in stack)
{
   Console.WriteLine(element);
}
```

The output (without removing):

```
50
40
30
20
10
```

Section 5.3: Creating an initialized List with Custom Types

```
public class Model
{
    public string Name { get; set; }
    public bool? Selected { get; set; }
}
```

Here we have a Class with no constructor with two properties: `Name` and a nullable boolean property `Selected`. If we wanted to initialize a `List<Model>`, there are a few different ways to execute this.

```
var SelectedEmployees = new List<Model>
 {
      new Model() {Name = "Item1", Selected = true},
      new Model() {Name = "Item2", Selected = false},
      new Model() {Name = "Item3", Selected = false},
      new Model() {Name = "Item4"}
 };
```

这里，我们创建了多个new的Model类实例，并用数据初始化它们。如果我们添加一个构造函数呢？

```
public class Model
{

    public Model(string name, bool? selected = false)
    {
名称 = name;
       选中 = Selected;
    }
    public string 名称 { get; set; }
    public bool? 选中 { get; set; }
}
```

这使我们可以稍微不同地初始化我们的列表 a。

```
var SelectedEmployees = new List<Model>
{
    new Model("马克", true),
    new Model("亚历克西斯"),
    new Model("")
};
```

如果一个类的属性本身是一个类呢？

```
public class Model
{
    public string 名称 { get; set; }
    public bool? 选中 { get; set; }
}

public class 扩展模型 : Model
{
    public 扩展模型()
    {
基础模型 = new Model();
    }

    public Model 基础模型 { get; set; }
    public DateTime 出生日期 { get; set; }
}
```

注意我们将 Model 类的构造函数还原，以简化示例。

```
var SelectedWithBirthDate = new List<ExtendedModel>
{
    new ExtendedModel()
    {
BaseModel = new Model { Name = "马克", Selected = true},
        BirthDate = new DateTime(2015, 11, 23)
```



```
var SelectedEmployees = new List<Model>
 {
      new Model() {Name = "Item1", Selected = true},
      new Model() {Name = "Item2", Selected = false},
      new Model() {Name = "Item3", Selected = false},
      new Model() {Name = "Item4"}
 };
```

Here, we are creating several new instances of our Model class, and initializing them with data. What if we added a constructor?

```
public class Model
{

    public Model(string name, bool? selected = false)
    {
        Name = name;
        selected = Selected;
    }
    public string Name { get; set; }
    public bool? Selected { get; set; }
}
```

This allows us to initialize our List a *little* differently.

```
var SelectedEmployees = new List<Model>
{
    new Model("Mark", true),
    new Model("Alexis"),
    new Model("")
};
```

What about a Class where one of the properties is a class itself?

```
public class Model
{
    public string Name { get; set; }
    public bool? Selected { get; set; }
}

public class ExtendedModel : Model
{
    public ExtendedModel()
    {
        BaseModel = new Model();
    }

    public Model BaseModel { get; set; }
    public DateTime BirthDate { get; set; }
}
```

Notice we reverted the constructor on the Model class to simplify the example a little bit.

```
var SelectedWithBirthDate = new List<ExtendedModel>
{
    new ExtendedModel()
    {
        BaseModel = new Model { Name = "Mark", Selected = true},
        BirthDate = new DateTime(2015, 11, 23)
```

```
    },
                        new ExtendedModel()
    {
BaseModel = new Model { Name = "随机"},
        BirthDate = new DateTime(2015, 11, 23)
    }
};
```

注意，我们可以将 `List<ExtendedModel>` 与 `Collection<ExtendedModel>`、`ExtendedModel[]`、`object[]`，甚至简单的 `[]` 互换使用。

第5.4节：队列

.Net中有一个集合用于管理使用 FIFO（先进先出） 概念的 Queue。队列的基本方法是 Enqueue(T item)，用于向队列中添加元素，Dequeue() 用于获取第一个元素并将其从队列中移除。泛型版本可以像下面的代码一样用于字符串队列。

首先，添加命名空间：

```
using System.Collections.Generic;
```

并使用它：

```
Queue<string> queue = new Queue<string>();
queue.Enqueue("约翰");
queue.Enqueue("Paul");
queue.入队("George");
queue.入队("Ringo");

string 出队值;
出队值 = queue.出队(); // 返回 John
出队值 = queue.出队(); // 返回 Paul
出队值 = queue.出队(); // 返回 George
出队值 = queue.出队(); // 返回 Ringo
```

有一个非泛型版本的类型，它可以处理对象。

命名空间是：

```
using System.Collections;
```

以及一个非泛型队列的代码示例：

```
Queue 队列 = new Queue();
queue.入队("Hello World"); // 字符串
queue.入队(5); // 整数
queue.入队(1d); // 双精度浮点数
queue.入队(true); // 布尔值
queue.入队(new Product()); // Product 对象

object 出队值;
出队值 = queue.出队(); // 返回 Hello World（字符串）
出队值 = queue.出队(); // 返回 5（整数）
出队值 = queue.出队(); // 返回 1d（双精度浮点数）
出队值 = queue.出队(); // 返回 true（布尔值）
出队值 = queue.出队(); // 返回 Product（Product 类型）
```



```
    },
                    new ExtendedModel()
    {
        BaseModel = new Model { Name = "Random"},
        BirthDate = new DateTime(2015, 11, 23)
    }
};
```

Note that we can interchange our `List<ExtendedModel>` with `Collection<ExtendedModel>`, `ExtendedModel[]`, `object[]`, or even simply `[]`.

Section 5.4: Queue

There is a collection in .Net used to manage values in a Queue that uses the FIFO (first-in first-out) concept. The basics of queues is the method Enqueue(T item) which is used to add elements in the queue and Dequeue() which is used to get the first element and remove it from the queue. The generic version can be used like the following code for a queue of strings.

First, add the namespace:

```
using System.Collections.Generic;
```

and use it:

```
Queue<string> queue = new Queue<string>();
queue.Enqueue("John");
queue.Enqueue("Paul");
queue.Enqueue("George");
queue.Enqueue("Ringo");

string dequeueValue;
dequeueValue = queue.Dequeue(); // return John
dequeueValue = queue.Dequeue(); // return Paul
dequeueValue = queue.Dequeue(); // return George
dequeueValue = queue.Dequeue(); // return Ringo
```

There is a non generic version of the type, which works with objects.

The namespace is:

```
using System.Collections;
```

Adn a code sample fo non generic queue:

```
Queue queue = new Queue();
queue.Enqueue("Hello World"); // string
queue.Enqueue(5); // int
queue.Enqueue(1d); // double
queue.Enqueue(true); // bool
queue.Enqueue(new Product()); // Product object

object dequeueValue;
dequeueValue = queue.Dequeue(); // return Hello World (string)
dequeueValue = queue.Dequeue(); // return 5 (int)
dequeueValue = queue.Dequeue(); // return 1d (double)
dequeueValue = queue.Dequeue(); // return true (bool)
dequeueValue = queue.Dequeue(); // return Product (Product type)
```

还有一个名为Peek()的方法，它返回队列开头的对象，但不移除该元素。

```
Queue<int> queue = new Queue<int>();
queue.Enqueue(10);
queue.Enqueue(20);
queue.Enqueue(30);
queue.Enqueue(40);
queue.Enqueue(50);

foreach (int element in queue)
{
Console.WriteLine(i);
}
```

输出（未移除时）：

```
30
```

There is also a method called Peek() which returns the object at the beginning of the queue without removing it the elements.

```
Queue<int> queue = new Queue<int>();
queue.Enqueue(10);
queue.Enqueue(20);
queue.Enqueue(30);
queue.Enqueue(40);
queue.Enqueue(50);

foreach (int element in queue)
{
    Console.WriteLine(i);
}
```

The output (without removing):

```
10
20
30
40
50
```

第6章：只读集合（ReadOnlyCollections）

第6.1节：创建只读集合

使用构造函数

只读集合（ReadOnlyCollection）是通过将现有的 IList 对象传入构造函数来创建的：

```
var groceryList = new List<string> { "Apple", "Banana" };
var readOnlyGroceryList = new ReadOnlyCollection<string>(groceryList);
```

使用 LINQ

此外，LINQ 为 IList 对象提供了一个 AsReadOnly() 扩展方法：

```
var readOnlyVersion = groceryList.AsReadOnly();
```

注意

通常，你希望私下维护源集合，并允许公开访问ReadOnlyCollection。
虽然你可以从内联列表创建一个ReadOnlyCollection，但创建后将无法修改该集合。

```
var readOnlyGroceryList = new List<string> {"Apple", "Banana"}.AsReadOnly();
// 很好，但你将无法更新购物清单，因为
// 你不再拥有源列表的引用！
```

如果你发现自己这样做，可能需要考虑使用另一种数据结构，例如
ImmutableCollection。

第6.2节：更新ReadOnlyCollection

ReadOnlyCollection不能直接编辑。相反，应更新源集合，
ReadOnlyCollection将反映这些更改。这是ReadOnlyCollection的关键特性。

```
var groceryList = new List<string> { "Apple", "Banana" };

var readOnlyGroceryList = new ReadOnlyCollection<string>(groceryList);

var itemCount = readOnlyGroceryList.Count;  // 当前有2个项目//readOnlyGroceryList.Add("

Candy");        // 编译错误 - 不能向 ReadOnlyCollection 对象添加项目

groceryList.Add("Vitamins");                // ..但可以向原始集合添加

itemCount = readOnlyGroceryList.Count;      // 现在有3个项目
            var lastItem = readOnlyGroceryList.Last();  // 只读列表中的最后一个项目现在是 "Vitamins"
```

查看演示

第6.3节：警告：ReadOnlyCollection 中的元素本质上并非只读

如果源集合的类型不是不可变的，通过 ReadOnlyCollection 访问的元素可以被

Chapter 6: ReadOnlyCollections

Section 6.1: Creating a ReadOnlyCollection

Using the Constructor

A `ReadOnlyCollection` is created by passing an existing `IList` object into the constructor:

```
var groceryList = new List<string> { "Apple", "Banana" };
var readOnlyGroceryList = new ReadOnlyCollection<string>(groceryList);
```

Using LINQ

Additionaly, LINQ provides an `AsReadOnly()` extension method for `IList` objects:

```
var readOnlyVersion = groceryList.AsReadOnly();
```

Note

Typically, you want to maintain the source collection privately and allow public access to the `ReadOnlyCollection`. While you could create a `ReadOnlyCollection` from an in-line list, you would be unable to modify the collection after you created it.

```
var readOnlyGroceryList = new List<string> {"Apple", "Banana"}.AsReadOnly();
// Great, but you will not be able to update the grocery list because
//  you do not have a reference to the source list anymore!
```

If you find yourself doing this, you may want to consider using another data structure, such as an `ImmutableCollection`.

Section 6.2: Updating a ReadOnlyCollection

A `ReadOnlyCollection` cannot be edited directly. Instead, the source collection is updated and the `ReadOnlyCollection` will reflect these changes. This is the key feature of the `ReadOnlyCollection`.

```
var groceryList = new List<string> { "Apple", "Banana" };

var readOnlyGroceryList = new ReadOnlyCollection<string>(groceryList);

var itemCount = readOnlyGroceryList.Count;  // There are currently 2 items

//readOnlyGroceryList.Add("Candy");         // Compiler Error - Items cannot be added to a
ReadOnlyCollection object
groceryList.Add("Vitamins");                // ..but they can be added to the original collection

itemCount = readOnlyGroceryList.Count;      // Now there are 3 items
var lastItem = readOnlyGroceryList.Last();  // The last item on the read only list is now "Vitamins"
```

View Demo

Section 6.3: Warning: Elements in a ReadOnlyCollection are not inherently read-only

If the source collection is of a type that is not immutable, elements accessed through a `ReadOnlyCollection` can be

修改。

```
public class Item
{
    public string Name { get; set; }
    public decimal Price { get; set; }
}

public static void FillOrder()
{
    // 生成一个订单
    var order = new List<Item>
    {
        new Item { Name = "苹果", Price = 0.50m },
        new Item { Name = "香蕉", Price = 0.75m },
        new Item { Name = "维生素", Price = 5.50m }
    };

    // 当前小计是6.75美元
    var subTotal = order.Sum(item => item.Price);

    // 让客户预览他们的订单
    var customerPreview = new ReadOnlyCollection<Item>(order);

    // 客户不能添加或删除商品，但他们可以更改
    //   商品的价格，尽管这是一个只读集合
customerPreview.Last().Price = 0.25m;

    // 小计现在只有1.50美元！
subTotal = order.Sum(item => item.Price);
}
```

查看演示

modified.

```
public class Item
{
    public string Name { get; set; }
    public decimal Price { get; set; }
}

public static void FillOrder()
{
    // An order is generated
    var order = new List<Item>
    {
        new Item { Name = "Apple", Price = 0.50m },
        new Item { Name = "Banana", Price = 0.75m },
        new Item { Name = "Vitamins", Price = 5.50m }
    };

    // The current sub total is $6.75
    var subTotal = order.Sum(item => item.Price);

    // Let the customer preview their order
    var customerPreview = new ReadOnlyCollection<Item>(order);

    // The customer can't add or remove items, but they can change
    //   the price of an item, even though it is a ReadOnlyCollection
    customerPreview.Last().Price = 0.25m;

    // The sub total is now only $1.50!
    subTotal = order.Sum(item => item.Price);
}
```

View Demo

第7章：栈和堆

第7.1节：使用中的值类型

值类型仅包含一个值。

所有值类型都派生自System.ValueType类，这包括大多数内置类型。

创建新的值类型时，会使用称为栈的内存区域。
栈会根据声明类型的大小相应增长。例如，int类型总是在栈上分配32位内存。
当值类型不再在作用域内时，栈上的空间将被释放。

下面的代码演示了将值类型赋给新变量。结构体被用作创建自定义值类型的便捷方式（System.ValueType类无法被其他方式扩展）。

重要的是要理解，当赋值一个值类型时，值本身会被***复制***到新变量，
这意味着我们有两个独立的对象实例，彼此不会相互影响。

```
struct PersonAsValueType
{
    public string Name;
}

class Program
{
    static void Main()
    {
PersonAsValueType personA;

        personA.Name = "Bob";

        var personB = personA;

personA.Name = "Linda";

Console.WriteLine(                  // 输出 'False' - 因为
            object.ReferenceEquals(         // personA 和 personB 引用的是
                personA,                    // 不同的内存区域
                personB));

Console.WriteLine(personA.Name);   // 输出 'Linda'
        Console.WriteLine(personB.Name);   // 输出 'Bob'
    }
}
```

第7.2节：引用类型的使用

引用类型由指向内存区域的引用和存储在该区域内的值组成。
这类似于C/C++中的指针。

所有引用类型都存储在所谓的堆上。
堆只是一个受管理的内存区域，用于存储对象。当实例化一个新对象时，堆的一部分将被分配给该对象使用，并返回指向该堆位置的引用。堆由垃圾回收器管理和维护，不允许手动干预。

除了实例本身所需的内存空间外，还需要额外的空间来存储引用

Chapter 7: Stack and Heap

Section 7.1: Value types in use

Value types simply contain a ***value***.

All value types are derived from the System.ValueType class, and this includes most of the built in types.

When creating a new value type, the an area of memory called ***the stack*** is used.
The stack will grow accordingly, by the size the declared type. So for example, an int will always be allocated 32 bits of memory on the stack. When the value type is no longer in scope, the space on the stack will be deallocated.

The code below demonstrates a value type being assigned to a new variable. A struct is being used as a convenient way to create a custom value type (the System.ValueType class cannot be otherwise extended).

The important thing to understand is that when assigning a value type, the value itself ***copied*** to the new variable, meaning we have two distinct instances of the object, that cannot affect each other.

```
struct PersonAsValueType
{
    public string Name;
}

class Program
{
    static void Main()
    {
        PersonAsValueType personA;

        personA.Name = "Bob";

        var personB = personA;

        personA.Name = "Linda";

        Console.WriteLine(                  // Outputs 'False' - because
            object.ReferenceEquals(         // personA and personB are referencing
                personA,                    // different areas of memory
                personB));

        Console.WriteLine(personA.Name);   // Outputs 'Linda'
        Console.WriteLine(personB.Name);   // Outputs 'Bob'
    }
}
```

Section 7.2: Reference types in use

Reference types are comprised of both a ***reference*** to a memory area, and a ***value*** stored within that area.
This is analogous to pointers in C/C++.

All reference types are stored on what is known as ***the heap***.
The heap is simply a managed area of memory where objects are stored. When a new object is instantiated, a part of the heap will be allocated for use by that object, and a reference to that location of the heap will be returned. The heap is managed and maintained by the *garbage collector*, and does not allow for manual intervention.

In addition to the memory space required for the instance itself, additional space is required to store the reference

本身，以及.NET CLR所需的额外临时信息。

下面的代码演示了将引用类型赋值给一个新变量。在此示例中，我们使用的是一个类，所有类都是引用类型（即使是静态的）。

当引用类型被赋值给另一个变量时，复制的是对象的引用，而不是值本身。这是值类型和引用类型之间的重要区别。

这意味着我们现在有两个指向同一对象的引用。
对该对象内的值所做的任何更改都会反映在两个变量中。

```
class PersonAsReferenceType
{
    public string Name;
}

class Program
{
    static void Main()
    {
PersonAsReferenceType personA;

        personA = new PersonAsReferenceType { Name = "Bob" };

        var personB = personA;

personA.Name = "Linda";

Console.WriteLine(               // 输出 'True' - 因为
            object.ReferenceEquals(      // personA 和 personB 引用的是
                personA,                 // *相同* 的内存位置
             personB));

Console.WriteLine(personA.Name); // 输出 'Linda'
        Console.WriteLine(personB.Name); // 输出 'Linda'
    }
```


itself, along with additional temporary information required by the .NET CLR.

The code below demonstrates a reference type being assigned to a new variable. In this instance, we are using a class, all classes are reference types (even if static).

When a reference type is assigned to another variable, it is the ***reference*** to the object that is copied over, **not** the value itself. This is an important distinction between value types and reference types.

The implications of this are that we now have *two* references to the same object.
Any changes to the values within that object will be reflected by both variables.

```
class PersonAsReferenceType
{
    public string Name;
}

class Program
{
    static void Main()
    {
        PersonAsReferenceType personA;

        personA = new PersonAsReferenceType { Name = "Bob" };

        var personB = personA;

        personA.Name = "Linda";

        Console.WriteLine(               // Outputs 'True' - because
            object.ReferenceEquals(      // personA and personB are referencing
                personA,                 // the *same* memory location
                personB));

        Console.WriteLine(personA.Name); // Outputs 'Linda'
        Console.WriteLine(personB.Name); // Outputs 'Linda'
    }
```

第8章：LINQ

LINQ（语言集成查询）是一种从数据源检索数据的表达式。LINQ通过为各种类型的数据源和格式提供一致的操作模型，简化了这一过程。在LINQ查询中，你始终处理的是对象。你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集合以及任何有提供程序支持的其他格式中的数据。LINQ可以在C#和VB中使用。

第8.1节：SelectMany（扁平映射）

Enumerable.Select为每个输入元素返回一个输出元素。而Enumerable.SelectMany为每个输入元素生成可变数量的输出元素。这意味着输出序列中的元素数量可能多于或少于输入序列中的元素数量。

传递给Enumerable.Select的Lambda表达式必须返回单个项。传递给Enumerable.SelectMany的Lambda表达式必须生成一个子序列。该子序列对于输入序列中的每个元素可能包含不同数量的元素。

示例

```
class Invoice
{
    public int Id { get; set; }
}

class Customer
{
    public Invoice[] 发票 {get;set;}
}

var 客户 = new[] {
    new 客户 {
发票 = new[] {
                new 发票 {Id=1},
                new 发票 {Id=2},
            }
    },
    new 客户 {
发票 = new[] {
                new 发票 {Id=3},
                new 发票 {Id=4},
            }
    },
    new 客户 {
发票 = new[] {
                new 发票 {Id=5},
                new 发票 {Id=6},
            }
    }
};

var allInvoicesFromAllCustomers = customers.SelectMany(c => c.Invoices);

Console.WriteLine(
    string.Join(",", allInvoicesFromAllCustomers.Select(i => i.Id).ToArray()));
```

输出：

Chapter 8: LINQ

LINQ (Language Integrated Query) is an expression that retrieves data from a data source. LINQ simplifies this situation by offering a consistent model for working with data across various kinds of data sources and formats. In a LINQ query, you are always working with objects. 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 provider is available. LINQ can be used in C# and VB.

Section 8.1: SelectMany (flat map)

`Enumerable.Select` returns an output element for every input element. Whereas `Enumerable.SelectMany` produces a variable number of output elements for each input element. This means that the output sequence may contain more or fewer elements than were in the input sequence.

`Lambda expressions` passed to `Enumerable.Select` must return a single item. Lambda expressions passed to `Enumerable.SelectMany` must produce a child sequence. This child sequence may contain a varying number of elements for each element in the input sequence.

Example

```
class Invoice
{
    public int Id { get; set; }
}

class Customer
{
    public Invoice[] Invoices {get;set;}
}

var customers = new[] {
    new Customer {
        Invoices = new[] {
            new Invoice {Id=1},
            new Invoice {Id=2},
        }
    },
    new Customer {
        Invoices = new[] {
            new Invoice {Id=3},
            new Invoice {Id=4},
        }
    },
    new Customer {
        Invoices = new[] {
            new Invoice {Id=5},
            new Invoice {Id=6},
        }
    }
};

var allInvoicesFromAllCustomers = customers.SelectMany(c => c.Invoices);

Console.WriteLine(
    string.Join(",", allInvoicesFromAllCustomers.Select(i => i.Id).ToArray()));
```

Output:

> 1,2,3,4,5,6

查看演示

Enumerable.SelectMany 也可以通过使用两个连续的 from 子句的基于语法的查询来实现：

```
var allInvoicesFromAllCustomers
    = from customer in customers
      from invoice in customer.Invoices
      select invoice;
```

第8.2节：Where（过滤）

此方法返回一个 IEnumerable，包含所有满足 lambda 表达式的元素

示例

```
var personNames = new[]
{
    "Foo", "Bar", "Fizz", "Buzz"
};

var namesStartingWithF = personNames.Where(p => p.StartsWith("F"));
Console.WriteLine(string.Join(",", namesStartingWithF));
```

输出：

> Foo,Fizz

查看演示

第8.3节：Any

如果集合中有任何元素满足lambda表达式中的条件，则返回true：

```
var numbers = new[] {1,2,3,4,5};

var isNotEmpty = numbers.Any();
Console.WriteLine(isNotEmpty); //True

var anyNumberIsOne = numbers.Any(n => n == 1);
Console.WriteLine(anyNumberIsOne); //True

var anyNumberIsSix = numbers.Any(n => n == 6);
Console.WriteLine(anyNumberIsSix); //False

var anyNumberIsOdd = numbers.Any(n => (n & 1) == 1);
Console.WriteLine(anyNumberIsOdd); //True

var anyNumberIsNegative = numbers.Any(n => n < 0);
Console.WriteLine(anyNumberIsNegative); //False
```


> 1,2,3,4,5,6

View Demo

`Enumerable.SelectMany` can also be achieved with a syntax-based query using two consecutive `from` clauses:

```
var allInvoicesFromAllCustomers
    = from customer in customers
      from invoice in customer.Invoices
      select invoice;
```

Section 8.2: Where (filter)

This method returns an IEnumerable with all the elements that meets the lambda expression

Example

```
var personNames = new[]
{
    "Foo", "Bar", "Fizz", "Buzz"
};

var namesStartingWithF = personNames.Where(p => p.StartsWith("F"));
Console.WriteLine(string.Join(",", namesStartingWithF));
```

Output:

> Foo,Fizz

View Demo

Section 8.3: Any

Returns `true` if the collection has any elements that meets the condition in the lambda expression:

```
var numbers = new[] {1,2,3,4,5};

var isNotEmpty = numbers.Any();
Console.WriteLine(isNotEmpty); //True

var anyNumberIsOne = numbers.Any(n => n == 1);
Console.WriteLine(anyNumberIsOne); //True

var anyNumberIsSix = numbers.Any(n => n == 6);
Console.WriteLine(anyNumberIsSix); //False

var anyNumberIsOdd = numbers.Any(n => (n & 1) == 1);
Console.WriteLine(anyNumberIsOdd); //True

var anyNumberIsNegative = numbers.Any(n => n < 0);
Console.WriteLine(anyNumberIsNegative); //False
```

第8.4节：GroupJoin

```
class Developer
{
    public int Id { get; set; }
    public string 名称 { get; set; }
}

class Project
{
    public int DeveloperId { get; set; }
    public string Name { get; set; }
}

var developers = new[] {
    new Developer {
Id = 1,
名称 = "Foobuzz"
    },
    new 开发者 {
       Id = 2,
       名称 = "Barfizz"
    }
};

var 项目 = new[] {
    new 项目 {
开发者Id = 1,
       名称 = "Hello World 3D"
    },
    new 项目 {
开发者Id = 1,
       名称 = "Super Fizzbuzz Maker"
    },
    new 项目 {
开发者Id = 2,
       名称 = "Citizen Kane - The action game"
    },
    new 项目 {
开发者Id = 2,
       名称 = "Pro Pong 2016"
    }
};

var 分组 = 开发者.GroupJoin(
    inner: 项目,
outerKeySelector: dev => dev.Id,
    innerKeySelector: proj => proj.DeveloperId,
    resultSelector:
        (dev, projs) => new {
DeveloperName = dev.Name,
            ProjectNames = projs.Select(p => p.Name).ToArray()});

foreach(var item in grouped)
{
Console.WriteLine(
        "{0}'s projects: {1}",
item.DeveloperName,
        string.Join(", ", item.ProjectNames));
}
```


Section 8.4: GroupJoin

```
class Developer
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Project
{
    public int DeveloperId { get; set; }
    public string Name { get; set; }
}

var developers = new[] {
    new Developer {
        Id = 1,
        Name = "Foobuzz"
    },
    new Developer {
        Id = 2,
        Name = "Barfizz"
    }
};

var projects = new[] {
    new Project {
        DeveloperId = 1,
        Name = "Hello World 3D"
    },
    new Project {
        DeveloperId = 1,
        Name = "Super Fizzbuzz Maker"
    },
    new Project {
        DeveloperId = 2,
        Name = "Citizen Kane - The action game"
    },
    new Project {
        DeveloperId = 2,
        Name = "Pro Pong 2016"
    }
};

var grouped = developers.GroupJoin(
    inner: projects,
    outerKeySelector: dev => dev.Id,
    innerKeySelector: proj => proj.DeveloperId,
    resultSelector:
        (dev, projs) => new {
            DeveloperName = dev.Name,
            ProjectNames = projs.Select(p => p.Name).ToArray()});

foreach(var item in grouped)
{
    Console.WriteLine(
        "{0}'s projects: {1}",
        item.DeveloperName,
        string.Join(", ", item.ProjectNames));
}
```

```
//Foobuzz's projects: Hello World 3D, Super Fizzbuzz Maker
//Barfizz's projects: Citizen Kane - The action game, Pro Pong 2016
```

第8.5节：Except

```
var numbers = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var evenNumbersBetweenSixAndFourteen = new[] { 6, 8, 10, 12 };

var result = numbers.Except(evenNumbersBetweenSixAndFourteen);

Console.WriteLine(string.Join(",", result));

//1, 2, 3, 4, 5, 7, 9
```

第8.6节：Zip

.NET 版本 ≥ 4.0

```
var tens = new[] {10,20,30,40,50};
var units = new[] {1,2,3,4,5};

var sums = tens.Zip(units, (first, second) => first + second);

Console.WriteLine(string.Join(",", sums));

//11,22,33,44,55
```

第8.7节：聚合（fold）

在每一步生成一个新对象：

```
var elements = new[] {1,2,3,4,5};

var commaSeparatedElements = elements.Aggregate(
    seed: "",
func: (aggregate, element) => $"{aggregate}{element},");

Console.WriteLine(commaSeparatedElements);  //1,2,3,4,5,
```

在所有步骤中使用同一个对象：

```
var commaSeparatedElements2 = elements.Aggregate(
    seed: new StringBuilder(),
func: (seed, element) => seed.Append($"{element},"));

Console.WriteLine(commaSeparatedElements2.ToString());  //1,2,3,4,5,
```

使用结果选择器：

```
var commaSeparatedElements3 = elements.Aggregate(
    seed: new StringBuilder(),
func: (seed, element) => seed.Append($"{element},"),
    resultSelector: (seed) => seed.ToString());
Console.WriteLine(commaSeparatedElements3);  //1,2,3,4,5,
```

如果省略种子，首个元素将作为种子：


```
//Foobuzz's projects: Hello World 3D, Super Fizzbuzz Maker
//Barfizz's projects: Citizen Kane - The action game, Pro Pong 2016
```

Section 8.5: Except

```
var numbers = new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
var evenNumbersBetweenSixAndFourteen = new[] { 6, 8, 10, 12 };

var result = numbers.Except(evenNumbersBetweenSixAndFourteen);

Console.WriteLine(string.Join(",", result));

//1, 2, 3, 4, 5, 7, 9
```

Section 8.6: Zip

.NET Version ≥ 4.0

```
var tens = new[] {10,20,30,40,50};
var units = new[] {1,2,3,4,5};

var sums = tens.Zip(units, (first, second) => first + second);

Console.WriteLine(string.Join(",", sums));

//11,22,33,44,55
```

Section 8.7: Aggregate (fold)

Generating a new object in each step:

```
var elements = new[] {1,2,3,4,5};

var commaSeparatedElements = elements.Aggregate(
    seed: "",
    func: (aggregate, element) => $"{aggregate}{element},");

Console.WriteLine(commaSeparatedElements);  //1,2,3,4,5,
```

Using the same object in all steps:

```
var commaSeparatedElements2 = elements.Aggregate(
    seed: new StringBuilder(),
    func: (seed, element) => seed.Append($"{element},"));

Console.WriteLine(commaSeparatedElements2.ToString());  //1,2,3,4,5,
```

Using a result selector:

```
var commaSeparatedElements3 = elements.Aggregate(
    seed: new StringBuilder(),
    func: (seed, element) => seed.Append($"{element},"),
    resultSelector: (seed) => seed.ToString());
Console.WriteLine(commaSeparatedElements3);  //1,2,3,4,5,
```

If a seed is omitted, the first element becomes the seed:

```
var seedAndElements = elements.Select(n=>n.ToString());
var commaSeparatedElements4 = seedAndElements.Aggregate(
    func: (aggregate, element) => $"{aggregate}{element},");

Console.WriteLine(commaSeparatedElements4);  //12,3,4,5,
```

第8.8节：ToLookup

```
var persons = new[] {
    new { Name="Fizz", Job="Developer"},
    new { Name="Buzz", Job="Developer"},
    new { Name="Foo", Job="Astronaut"},
    new { Name="Bar", Job="Astronaut"},
};

var groupedByJob = persons.ToLookup(p => p.Job);

foreach(var theGroup in groupedByJob)
{
Console.WriteLine(
        "{0} 是 {1}们",
        string.Join(",", theGroup.Select(g => g.Name).ToArray()),
        theGroup.Key);
}

//Fizz，Buzz 是开发者
//Foo，Bar 是宇航员
```

第8.9节：交集

```
var numbers1to10 = new[] {1,2,3,4,5,6,7,8,9,10};
var numbers5to15 = new[] {5,6,7,8,9,10,11,12,13,14,15};

var numbers5to10 = numbers1to10.Intersect(numbers5to15);

Console.WriteLine(string.Join(",", numbers5to10));

//5,6,7,8,9,10
```

第8.10节：连接

```
var numbers1to5 = new[] {1, 2, 3, 4, 5};
var numbers4to8 = new[] {4, 5, 6, 7, 8};

var numbers1to8 = numbers1to5.Concat(numbers4to8);

Console.WriteLine(string.Join(",", numbers1to8));

//1,2,3,4,5,4,5,6,7,8
```

注意结果中会保留重复项。如果不希望如此，请使用Union代替。

第8.11节：All

```
var numbers = new[] {1,2,3,4,5};

var allNumbersAreOdd = numbers.All(n => (n & 1) == 1);
```



```
var seedAndElements = elements.Select(n=>n.ToString());
var commaSeparatedElements4 = seedAndElements.Aggregate(
    func: (aggregate, element) => $"{aggregate}{element},");

Console.WriteLine(commaSeparatedElements4);  //12,3,4,5,
```

Section 8.8: ToLookup

```
var persons = new[] {
    new { Name="Fizz", Job="Developer"},
    new { Name="Buzz", Job="Developer"},
    new { Name="Foo", Job="Astronaut"},
    new { Name="Bar", Job="Astronaut"},
};

var groupedByJob = persons.ToLookup(p => p.Job);

foreach(var theGroup in groupedByJob)
{
    Console.WriteLine(
        "{0} are {1}s",
        string.Join(",", theGroup.Select(g => g.Name).ToArray()),
        theGroup.Key);
}

//Fizz,Buzz are Developers
//Foo,Bar are Astronauts
```

Section 8.9: Intersect

```
var numbers1to10 = new[] {1,2,3,4,5,6,7,8,9,10};
var numbers5to15 = new[] {5,6,7,8,9,10,11,12,13,14,15};

var numbers5to10 = numbers1to10.Intersect(numbers5to15);

Console.WriteLine(string.Join(",", numbers5to10));

//5,6,7,8,9,10
```

Section 8.10: Concat

```
var numbers1to5 = new[] {1, 2, 3, 4, 5};
var numbers4to8 = new[] {4, 5, 6, 7, 8};

var numbers1to8 = numbers1to5.Concat(numbers4to8);

Console.WriteLine(string.Join(",", numbers1to8));

//1,2,3,4,5,4,5,6,7,8
```

Note that duplicates are kept in the result. If this is undesirable, use `Union` instead.

Section 8.11: All

```
var numbers = new[] {1,2,3,4,5};

var allNumbersAreOdd = numbers.All(n => (n & 1) == 1);
```

```
Console.WriteLine(allNumbersAreOdd); //False

var allNumbersArePositive = numbers.All(n => n > 0);
Console.WriteLine(allNumbersArePositive); //True
```

注意，All 方法的功能是检查第一个根据谓词评估为false的元素。因此，在集合为空的情况下，对于任何谓词，该方法都会返回true：

```
var numbers = new int[0];
var allNumbersArePositive = numbers.All(n => n > 0);
Console.WriteLine(allNumbersArePositive); //True
```

第8.12节：Sum

```
var numbers = new[] {1,2,3,4};

var sumOfAllNumbers = numbers.Sum();
Console.WriteLine(sumOfAllNumbers); //10

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var totalPopulation = cities.Sum(c => c.Population);
Console.WriteLine(totalPopulation); //7500
```

第8.13节：SequenceEqual

```
var numbers = new[] {1,2,3,4,5};
var sameNumbers = new[] {1,2,3,4,5};
var sameNumbersInDifferentOrder = new[] {5,1,4,2,3};

var equalIfSameOrder = numbers.SequenceEqual(sameNumbers);
Console.WriteLine(equalIfSameOrder); //True

var equalIfDifferentOrder = numbers.SequenceEqual(sameNumbersInDifferentOrder);
Console.WriteLine(equalIfDifferentOrder); //False
```

第8.14节：Min（最小值）

```
var numbers = new[] {1,2,3,4};

var minNumber = numbers.Min();
Console.WriteLine(minNumber); //1

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var minPopulation = cities.Min(c => c.Population);
Console.WriteLine(minPopulation); //1000
```



```
Console.WriteLine(allNumbersAreOdd); //False

var allNumbersArePositive = numbers.All(n => n > 0);
Console.WriteLine(allNumbersArePositive); //True
```

Note that the `All` method functions by checking for the first element to evaluate as `false` according to the predicate. Therefore, the method will return `true` for *any* predicate in the case that the set is empty:

```
var numbers = new int[0];
var allNumbersArePositive = numbers.All(n => n > 0);
Console.WriteLine(allNumbersArePositive); //True
```

Section 8.12: Sum

```
var numbers = new[] {1,2,3,4};

var sumOfAllNumbers = numbers.Sum();
Console.WriteLine(sumOfAllNumbers); //10

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var totalPopulation = cities.Sum(c => c.Population);
Console.WriteLine(totalPopulation); //7500
```

Section 8.13: SequenceEqual

```
var numbers = new[] {1,2,3,4,5};
var sameNumbers = new[] {1,2,3,4,5};
var sameNumbersInDifferentOrder = new[] {5,1,4,2,3};

var equalIfSameOrder = numbers.SequenceEqual(sameNumbers);
Console.WriteLine(equalIfSameOrder); //True

var equalIfDifferentOrder = numbers.SequenceEqual(sameNumbersInDifferentOrder);
Console.WriteLine(equalIfDifferentOrder); //False
```

Section 8.14: Min

```
var numbers = new[] {1,2,3,4};

var minNumber = numbers.Min();
Console.WriteLine(minNumber); //1

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var minPopulation = cities.Min(c => c.Population);
Console.WriteLine(minPopulation); //1000
```

第8.15节：Distinct（去重）

```
var numbers = new[] {1, 1, 2, 2, 3, 3, 4, 4, 5, 5};
var distinctNumbers = numbers.Distinct();

Console.WriteLine(string.Join(",", distinctNumbers));

//1,2,3,4,5
```

第8.16节：Count（计数）

```
IEnumerable<int> numbers = new[] {1,2,3,4,5,6,7,8,9,10};

var numbersCount = numbers.Count();
Console.WriteLine(numbersCount); //10

var evenNumbersCount = numbers.Count(n => (n & 1) == 0);
Console.WriteLine(evenNumbersCount); //5
```

第8.17节：Cast

`Cast` 与其他 `Enumerable` 方法不同，它是 `IEnumerable` 的扩展方法，而不是 `IEnumerable<T>` 的扩展方法。因此它可以用来将前者的实例转换为后者的实例。

这段代码无法编译，因为 ArrayList 没有实现 IEnumerable<T>：

```
var numbers = new ArrayList() {1,2,3,4,5};
Console.WriteLine(numbers.First());
```

这段代码按预期工作：

```
var numbers = new ArrayList() {1,2,3,4,5};
Console.WriteLine(numbers.Cast<int>().First()); //1
```

Cast 不执行转换投射。以下代码可以编译，但在运行时会抛出 InvalidCastException 异常：

```
var numbers = new int[] {1,2,3,4,5};
decimal[] numbersAsDecimal = numbers.Cast<decimal>().ToArray();
```

执行转换类型转换为集合的正确方法如下：

```
var numbers= new int[] {1,2,3,4,5};
decimal[] numbersAsDecimal = numbers.Select(n => (decimal)n).ToArray();
```

第8.18节：范围

Range的两个参数是第一个数字和要生成的元素数量（不是最后一个数字）。

```
// 输出 1,2,3,4,5,6,7,8,9,10
Console.WriteLine(string.Join(",", Enumerable.Range(1, 10)));

// 输出 10,11,12,13,14
Console.WriteLine(string.Join(",", Enumerable.Range(10, 5)));
```


Section 8.15: Distinct

```
var numbers = new[] {1, 1, 2, 2, 3, 3, 4, 4, 5, 5};
var distinctNumbers = numbers.Distinct();

Console.WriteLine(string.Join(",", distinctNumbers));

//1,2,3,4,5
```

Section 8.16: Count

```
IEnumerable<int> numbers = new[] {1,2,3,4,5,6,7,8,9,10};

var numbersCount = numbers.Count();
Console.WriteLine(numbersCount); //10

var evenNumbersCount = numbers.Count(n => (n & 1) == 0);
Console.WriteLine(evenNumbersCount); //5
```

Section 8.17: Cast

`Cast` is different from the other methods of `Enumerable` in that it is an extension method for `IEnumerable`, not for `IEnumerable<T>`. Thus it can be used to convert instances of the former into instances of the later.

This does not compile since `ArrayList` does not implement `IEnumerable<T>`:

```
var numbers = new ArrayList() {1,2,3,4,5};
Console.WriteLine(numbers.First());
```

This works as expected:

```
var numbers = new ArrayList() {1,2,3,4,5};
Console.WriteLine(numbers.Cast<int>().First()); //1
```

`Cast` does **not** perform conversion casts. The following compiles but throws `InvalidCastException` at runtime:

```
var numbers = new int[] {1,2,3,4,5};
decimal[] numbersAsDecimal = numbers.Cast<decimal>().ToArray();
```

The proper way to perform a converting cast to a collection is as follows:

```
var numbers= new int[] {1,2,3,4,5};
decimal[] numbersAsDecimal = numbers.Select(n => (decimal)n).ToArray();
```

Section 8.18: Range

The two parameters to `Range` are the *first* number and the *count* of elements to produce (not the last number).

```
// prints 1,2,3,4,5,6,7,8,9,10
Console.WriteLine(string.Join(",", Enumerable.Range(1, 10)));

// prints 10,11,12,13,14
Console.WriteLine(string.Join(",", Enumerable.Range(10, 5)));
```

第8.19节：ThenBy

ThenBy只能在OrderBy子句之后使用，允许使用多个条件进行排序

```
var persons = new[]
{
    new {Id = 1, Name = "Foo", Order = 1},
    new {Id = 1, Name = "FooTwo", Order = 2},
    new {Id = 2, Name = "Bar", Order = 2},
    new {Id = 2, Name = "BarTwo", Order = 1},
    new {Id = 3, Name = "Fizz", Order = 2},
    new {Id = 3, Name = "FizzTwo", Order = 1},
};

var personsSortedByName = persons.OrderBy(p => p.Id).ThenBy(p => p.Order);

Console.WriteLine(string.Join(",", personsSortedByName.Select(p => p.Name)));
//这将显示：
//Foo,FooTwo,BarTwo,Bar,FizzTwo,Fizz
```

第8.20节：重复

Enumerable.Repeat 生成一个重复值的序列。在此示例中，它生成了4次“Hello”。

```
var repeats = Enumerable.Repeat("Hello", 4);

foreach (var item in repeats)
{
Console.WriteLine(item);
}

/* 输出：
你好
    你好
    你好
    你好
*/
```

第8.21节：空

创建一个空的 int 类型 IEnumerable：

```
IEnumerable<int> emptyList = Enumerable.Empty<int>();
```

这个空的 IEnumerable 会为每个类型 T 缓存，因此：

```
Enumerable.Empty<decimal>() == Enumerable.Empty<decimal>(); // 这为真
Enumerable.Empty<int>() == Enumerable.Empty<decimal>();     // 这为假
```

第8.22节：Select（映射）

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
```


Section 8.19: ThenBy

`ThenBy` can only be used after a `OrderBy` clause allowing to order using multiple criteria

```
var persons = new[]
{
    new {Id = 1, Name = "Foo", Order = 1},
    new {Id = 1, Name = "FooTwo", Order = 2},
    new {Id = 2, Name = "Bar", Order = 2},
    new {Id = 2, Name = "BarTwo", Order = 1},
    new {Id = 3, Name = "Fizz", Order = 2},
    new {Id = 3, Name = "FizzTwo", Order = 1},
};

var personsSortedByName = persons.OrderBy(p => p.Id).ThenBy(p => p.Order);

Console.WriteLine(string.Join(",", personsSortedByName.Select(p => p.Name)));
//This will display :
//Foo,FooTwo,BarTwo,Bar,FizzTwo,Fizz
```

Section 8.20: Repeat

`Enumerable.Repeat` generates a sequence of a repeated value. In this example it generates "Hello" 4 times.

```
var repeats = Enumerable.Repeat("Hello", 4);

foreach (var item in repeats)
{
    Console.WriteLine(item);
}

/* output:
    Hello
    Hello
    Hello
    Hello
*/
```

Section 8.21: Empty

To create an empty IEnumerable of int:

```
IEnumerable<int> emptyList = Enumerable.Empty<int>();
```

This empty IEnumerable is cached for each Type T, so that:

```
Enumerable.Empty<decimal>() == Enumerable.Empty<decimal>(); // This is True
Enumerable.Empty<int>() == Enumerable.Empty<decimal>();     // This is False
```

Section 8.22: Select (map)

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
```

```
};

var names = persons.Select(p => p.Name);
Console.WriteLine(string.Join(",", names.ToArray()));

//Foo,Bar,Fizz,Buzz
```

这种类型的函数在函数式编程语言中通常称为map。

第8.23节：OrderBy

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var personsSortedByName = persons.OrderBy(p => p.Name);

Console.WriteLine(string.Join(",", personsSortedByName.Select(p => p.Id).ToArray()));

//2,4,3,1
```

第8.24节：OrderByDescending

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var personsSortedByNameDescending = persons.OrderByDescending(p => p.Name);

Console.WriteLine(string.Join(",", personsSortedByNameDescending.Select(p => p.Id).ToArray()));

//1,3,4,2
```

第8.25节：Contains

```
var numbers = new[] {1,2,3,4,5};
Console.WriteLine(numbers.Contains(3)); //True
Console.WriteLine(numbers.Contains(34)); //False
```

第8.26节：First（查找）

```
var numbers = new[] {1,2,3,4,5};

var firstNumber = numbers.First();
Console.WriteLine(firstNumber); //1

var firstEvenNumber = numbers.First(n => (n & 1) == 0);
Console.WriteLine(firstEvenNumber); //2
```



```
};

var names = persons.Select(p => p.Name);
Console.WriteLine(string.Join(",", names.ToArray()));

//Foo,Bar,Fizz,Buzz
```

This type of function is usually called `map` in functional programming languages.

Section 8.23: OrderBy

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var personsSortedByName = persons.OrderBy(p => p.Name);

Console.WriteLine(string.Join(",", personsSortedByName.Select(p => p.Id).ToArray()));

//2,4,3,1
```

Section 8.24: OrderByDescending

```
var persons = new[]
{
    new {Id = 1, Name = "Foo"},
    new {Id = 2, Name = "Bar"},
    new {Id = 3, Name = "Fizz"},
    new {Id = 4, Name = "Buzz"}
};

var personsSortedByNameDescending = persons.OrderByDescending(p => p.Name);

Console.WriteLine(string.Join(",", personsSortedByNameDescending.Select(p => p.Id).ToArray()));

//1,3,4,2
```

Section 8.25: Contains

```
var numbers = new[] {1,2,3,4,5};
Console.WriteLine(numbers.Contains(3)); //True
Console.WriteLine(numbers.Contains(34)); //False
```

Section 8.26: First (find)

```
var numbers = new[] {1,2,3,4,5};

var firstNumber = numbers.First();
Console.WriteLine(firstNumber); //1

var firstEvenNumber = numbers.First(n => (n & 1) == 0);
Console.WriteLine(firstEvenNumber); //2
```

以下代码会抛出InvalidOperationException，消息为“序列不包含匹配的元素”：

```
var firstNegativeNumber = numbers.First(n => n < 0);
```

第8.27节：Single

```
var oneNumber = new[] {5};
var theOnlyNumber = oneNumber.Single();
Console.WriteLine(theOnlyNumber);  //5

var numbers = new[] {1,2,3,4,5};

var theOnlyNumberSmallerThanTwo = numbers.Single(n => n < 2);
Console.WriteLine(theOnlyNumberSmallerThanTwo);  //1
```

以下代码会抛出InvalidOperationException，因为序列中有多个元素：

```
var theOnlyNumberInNumbers = numbers.Single();
var theOnlyNegativeNumber = numbers.Single(n => n < 0);
```

第8.28节：Last

```
var numbers = new[] {1,2,3,4,5};

var lastNumber = numbers.Last();
Console.WriteLine(lastNumber); //5

var lastEvenNumber = numbers.Last(n => (n & 1) == 0);
Console.WriteLine(lastEvenNumber); //4
```

以下代码会抛出InvalidOperationException：

```
var lastNegativeNumber = numbers.Last(n => n < 0);
```

第8.29节：LastOrDefault

```
var numbers = new[] {1,2,3,4,5};

var lastNumber = numbers.LastOrDefault();
Console.WriteLine(lastNumber); //5

var lastEvenNumber = numbers.LastOrDefault(n => (n & 1) == 0);
Console.WriteLine(lastEvenNumber); //4

var lastNegativeNumber = numbers.LastOrDefault(n => n < 0);
Console.WriteLine(lastNegativeNumber); //0

var words = new[] { "one", "two", "three", "four", "five" };

var lastWord = words.LastOrDefault();
Console.WriteLine(lastWord); // five

var lastLongWord = words.LastOrDefault(w => w.Length > 4);
Console.WriteLine(lastLongWord); // three

var lastMissingWord = words.LastOrDefault(w => w.Length > 5);
```


The following throws `InvalidOperationException` with message "Sequence contains no matching element":

```
var firstNegativeNumber = numbers.First(n => n < 0);
```

Section 8.27: Single

```
var oneNumber = new[] {5};
var theOnlyNumber = oneNumber.Single();
Console.WriteLine(theOnlyNumber);  //5

var numbers = new[] {1,2,3,4,5};

var theOnlyNumberSmallerThanTwo = numbers.Single(n => n < 2);
Console.WriteLine(theOnlyNumberSmallerThanTwo);  //1
```

The following throws `InvalidOperationException` since there is more than one element in the sequence:

```
var theOnlyNumberInNumbers = numbers.Single();
var theOnlyNegativeNumber = numbers.Single(n => n < 0);
```

Section 8.28: Last

```
var numbers = new[] {1,2,3,4,5};

var lastNumber = numbers.Last();
Console.WriteLine(lastNumber); //5

var lastEvenNumber = numbers.Last(n => (n & 1) == 0);
Console.WriteLine(lastEvenNumber); //4
```

The following throws `InvalidOperationException`:

```
var lastNegativeNumber = numbers.Last(n => n < 0);
```

Section 8.29: LastOrDefault

```
var numbers = new[] {1,2,3,4,5};

var lastNumber = numbers.LastOrDefault();
Console.WriteLine(lastNumber); //5

var lastEvenNumber = numbers.LastOrDefault(n => (n & 1) == 0);
Console.WriteLine(lastEvenNumber); //4

var lastNegativeNumber = numbers.LastOrDefault(n => n < 0);
Console.WriteLine(lastNegativeNumber); //0

var words = new[] { "one", "two", "three", "four", "five" };

var lastWord = words.LastOrDefault();
Console.WriteLine(lastWord); // five

var lastLongWord = words.LastOrDefault(w => w.Length > 4);
Console.WriteLine(lastLongWord); // three

var lastMissingWord = words.LastOrDefault(w => w.Length > 5);
```

```
Console.WriteLine(lastMissingWord); // null
```

第8.30节：SingleOrDefault

```
var oneNumber = new[] {5};
var theOnlyNumber = oneNumber.SingleOrDefault();
Console.WriteLine(theOnlyNumber);  //5

var numbers = new[] {1,2,3,4,5};

var 唯一小于二的数字 = numbers.SingleOrDefault(n => n < 2);
Console.WriteLine(唯一小于二的数字);  //1

var 唯一的负数 = numbers.SingleOrDefault(n => n < 0);
Console.WriteLine(唯一的负数);  //0
```

以下代码会抛出InvalidOperationException：

```
var numbers中唯一的数字 = numbers.SingleOrDefault();
```

第8.31节：FirstOrDefault

```
var numbers = new[] {1,2,3,4,5};

var 第一个数字 = numbers.FirstOrDefault();
Console.WriteLine(第一个数字); //1

var 第一个偶数 = numbers.FirstOrDefault(n => (n & 1) == 0);
Console.WriteLine(第一个偶数); //2

var 第一个负数 = numbers.FirstOrDefault(n => n < 0);
Console.WriteLine(第一个负数); //0

var words = new[] { "one", "two", "three", "four", "five" };

var 第一个单词 = words.FirstOrDefault();
Console.WriteLine(第一个单词); // one

var 第一个长单词 = words.FirstOrDefault(w => w.Length > 3);
Console.WriteLine(第一个长单词); // three

var 第一个缺失的单词 = words.FirstOrDefault(w => w.Length > 5);
Console.WriteLine(第一个缺失的单词); // null
```

第8.32节：Skip

Skip 会枚举前 N 个元素但不返回它们。一旦达到第 N+1 个元素，Skip 开始返回每个枚举的元素：

```
var numbers = new[] {1,2,3,4,5};

var allNumbersExceptFirstTwo = numbers.Skip(2);
Console.WriteLine(string.Join(",", allNumbersExceptFirstTwo.ToArray()));

//3,4,5
```



```
Console.WriteLine(lastMissingWord); // null
```

Section 8.30: SingleOrDefault

```
var oneNumber = new[] {5};
var theOnlyNumber = oneNumber.SingleOrDefault();
Console.WriteLine(theOnlyNumber);  //5

var numbers = new[] {1,2,3,4,5};

var theOnlyNumberSmallerThanTwo = numbers.SingleOrDefault(n => n < 2);
Console.WriteLine(theOnlyNumberSmallerThanTwo);  //1

var theOnlyNegativeNumber = numbers.SingleOrDefault(n => n < 0);
Console.WriteLine(theOnlyNegativeNumber);  //0
```

The following throws `InvalidOperationException`:

```
var theOnlyNumberInNumbers = numbers.SingleOrDefault();
```

Section 8.31: FirstOrDefault

```
var numbers = new[] {1,2,3,4,5};

var firstNumber = numbers.FirstOrDefault();
Console.WriteLine(firstNumber); //1

var firstEvenNumber = numbers.FirstOrDefault(n => (n & 1) == 0);
Console.WriteLine(firstEvenNumber); //2

var firstNegativeNumber = numbers.FirstOrDefault(n => n < 0);
Console.WriteLine(firstNegativeNumber); //0

var words = new[] { "one", "two", "three", "four", "five" };

var firstWord = words.FirstOrDefault();
Console.WriteLine(firstWord); // one

var firstLongWord = words.FirstOrDefault(w => w.Length > 3);
Console.WriteLine(firstLongWord); // three

var firstMissingWord = words.FirstOrDefault(w => w.Length > 5);
Console.WriteLine(firstMissingWord); // null
```

Section 8.32: Skip

Skip will enumerate the first N items without returning them. Once item number N+1 is reached, Skip starts returning every enumerated item:

```
var numbers = new[] {1,2,3,4,5};

var allNumbersExceptFirstTwo = numbers.Skip(2);
Console.WriteLine(string.Join(",", allNumbersExceptFirstTwo.ToArray()));

//3,4,5
```

第8.33节：Take

此方法从可枚举集合中获取前 n 个元素。

```
var numbers = new[] {1,2,3,4,5};

var threeFirstNumbers = numbers.Take(3);
Console.WriteLine(string.Join(",", threeFirstNumbers.ToArray()));

//1,2,3
```

第8.34节：Reverse

```
var numbers = new[] {1,2,3,4,5};
var reversed = numbers.Reverse();

Console.WriteLine(string.Join(",", reversed.ToArray()));

//5,4,3,2,1
```

第8.35节：OfType

```
var mixed = new object[] {1,"Foo",2,"Bar",3,"Fizz",4,"Buzz"};
var numbers = mixed.OfType<int>();

Console.WriteLine(string.Join(",", numbers.ToArray()));

//1,2,3,4
```

第8.36节：Max

```
var numbers = new[] {1,2,3,4};

var maxNumber = numbers.Max();
Console.WriteLine(maxNumber); //4

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var maxPopulation = cities.Max(c => c.Population);
Console.WriteLine(maxPopulation); //4000
```

第8.37节：Average

```
var numbers = new[] {1,2,3,4};

var averageNumber = numbers.Average();
Console.WriteLine(averageNumber);
// 2,5
```

此方法计算数字集合的平均值。

```
var cities = new[] {
```


Section 8.33: Take

This method takes the first n elements from an enumerable.

```
var numbers = new[] {1,2,3,4,5};

var threeFirstNumbers = numbers.Take(3);
Console.WriteLine(string.Join(",", threeFirstNumbers.ToArray()));

//1,2,3
```

Section 8.34: Reverse

```
var numbers = new[] {1,2,3,4,5};
var reversed = numbers.Reverse();

Console.WriteLine(string.Join(",", reversed.ToArray()));

//5,4,3,2,1
```

Section 8.35: OfType

```
var mixed = new object[] {1,"Foo",2,"Bar",3,"Fizz",4,"Buzz"};
var numbers = mixed.OfType<int>();

Console.WriteLine(string.Join(",", numbers.ToArray()));

//1,2,3,4
```

Section 8.36: Max

```
var numbers = new[] {1,2,3,4};

var maxNumber = numbers.Max();
Console.WriteLine(maxNumber); //4

var cities = new[] {
    new {Population = 1000},
    new {Population = 2500},
    new {Population = 4000}
};

var maxPopulation = cities.Max(c => c.Population);
Console.WriteLine(maxPopulation); //4000
```

Section 8.37: Average

```
var numbers = new[] {1,2,3,4};

var averageNumber = numbers.Average();
Console.WriteLine(averageNumber);
// 2,5
```

This method calculates the average of enumerable of numbers.

```
var cities = new[] {
```

```
    new {Population = 1000},
    new {Population = 2000},
    new {Population = 4000}
};

var averagePopulation = cities.Average(c => c.Population);
Console.WriteLine(averagePopulation);
// 2333,33
```

此方法使用委托函数计算可枚举集合的平均值。

第8.38节：GroupBy（分组）

```
var persons = new[] {
    new { Name="Fizz", Job="Developer"},
    new { Name="Buzz", Job="Developer"},
    new { Name="Foo", Job="Astronaut"},
    new { Name="Bar", Job="Astronaut"},
};

var groupedByJob = persons.GroupBy(p => p.Job);

foreach(var theGroup in groupedByJob)
{
Console.WriteLine(
        "{0} 是 {1}们",
        string.Join(",", theGroup.Select(g => g.Name).ToArray()),
        theGroup.Key);
}

//Fizz，Buzz 是开发者
//Foo，Bar 是宇航员
```

按国家对发票进行分组，生成包含记录数、总支付金额和平均支付金额的新对象

```
var a = db.Invoices.GroupBy(i => i.Country)
          .Select(g => new { Country = g.Key,
                             Count = g.Count(),
                             Total = g.Sum(i => i.Paid),
                             Average = g.Average(i => i.Paid) });
```

如果我们只想要总计，不要分组

```
var a = db.Invoices.GroupBy(i => 1)
          .Select(g => new { Count = g.Count(),
                             Total = g.Sum(i => i.Paid),
                             Average = g.Average(i => i.Paid) });
```

如果我们需要多个计数

```
var a = db.Invoices.GroupBy(g => 1)
          .Select(g => new { High = g.Count(i => i.Paid >= 1000),
                             Low = g.Count(i => i.Paid < 1000),
                             Sum = g.Sum(i => i.Paid) });
```

第8.39节：ToDictionary

使用提供的keySelector函数从源IEnumerable返回一个新的字典，用于确定键。


```
    new {Population = 1000},
    new {Population = 2000},
    new {Population = 4000}
};

var averagePopulation = cities.Average(c => c.Population);
Console.WriteLine(averagePopulation);
// 2333,33
```

This method calculates the average of enumerable using delegated function.

Section 8.38: GroupBy

```
var persons = new[] {
    new { Name="Fizz", Job="Developer"},
    new { Name="Buzz", Job="Developer"},
    new { Name="Foo", Job="Astronaut"},
    new { Name="Bar", Job="Astronaut"},
};

var groupedByJob = persons.GroupBy(p => p.Job);

foreach(var theGroup in groupedByJob)
{
    Console.WriteLine(
        "{0} are {1}s",
        string.Join(",", theGroup.Select(g => g.Name).ToArray()),
        theGroup.Key);
}

//Fizz,Buzz are Developers
//Foo,Bar are Astronauts
```

Group invoices by country, generating a new object with the number of record, total paid, and average paid

```
var a = db.Invoices.GroupBy(i => i.Country)
          .Select(g => new { Country = g.Key,
                             Count = g.Count(),
                             Total = g.Sum(i => i.Paid),
                             Average = g.Average(i => i.Paid) });
```

If we want only the totals, no group

```
var a = db.Invoices.GroupBy(i => 1)
          .Select(g => new { Count = g.Count(),
                             Total = g.Sum(i => i.Paid),
                             Average = g.Average(i => i.Paid) });
```

If we need several counts

```
var a = db.Invoices.GroupBy(g => 1)
          .Select(g => new { High = g.Count(i => i.Paid >= 1000),
                             Low = g.Count(i => i.Paid < 1000),
                             Sum = g.Sum(i => i.Paid) });
```

Section 8.39: ToDictionary

Returns a new dictionary from the source `IEnumerable` using the provided keySelector function to determine keys.

如果keySelector不是单射（对源集合的每个成员返回唯一值），将抛出ArgumentException。还有重载允许指定存储的值以及键。

```
var persons = new[] {
    new { Name="Fizz", Id=1},
    new { Name="Buzz", Id=2},
    new { Name="Foo", Id=3},
    new { Name="Bar", Id=4},
};
```

仅指定键选择器函数将创建一个Dictionary<TKey,TVal>，TKey为键选择器的返回类型，TVal为原始对象类型，原始对象作为存储值。

```
var personsById = persons.ToDictionary(p => p.Id);
// personsById 是一个 Dictionary<int,object>

Console.WriteLine(personsById[1].Name); //Fizz
Console.WriteLine(personsById[2].Name); //Buzz
```

指定一个值选择器函数也会创建一个Dictionary<TKey,TVal>，其中TKey仍然是键选择器的返回类型，但TVal现在是值选择器函数的返回类型，返回值作为存储的值。

```
var namesById = persons.ToDictionary(p => p.Id, p => p.Name);
//namesById 是一个 Dictionary<int,string>

Console.WriteLine(namesById[3]); //Foo
Console.WriteLine(namesById[4]); //Bar
```

如上所述，键选择器返回的键必须是唯一的。以下代码将抛出异常。

```
var persons = new[] {
    new { Name="Fizz", Id=1},
    new { Name="Buzz", Id=2},
    new { Name="Foo", Id=3},
    new { Name="Bar", Id=4},
    new { Name="Oops", Id=4}
};

var willThrowException = persons.ToDictionary(p => p.Id)
```

如果无法为源集合提供唯一键，请考虑改用 ToLookup。从表面上看，ToLookup 的行为类似于 ToDictionary，然而，在生成的 Lookup 中，每个键都与具有匹配键的值集合配对。

第8.40节：并集（Union）

```
var numbers1to5 = new[] {1,2,3,4,5};
var numbers4to8 = new[] {4,5,6,7,8};

var numbers1to8 = numbers1to5.Union(numbers4to8);

Console.WriteLine(string.Join(",", numbers1to8));

//1,2,3,4,5,6,7,8
```

注意结果中会去除重复项。如果不希望去重，请使用Concat代替。

Will throw an `ArgumentException` if keySelector is not injective(returns a unique value for each member of the source collection.) There are overloads which allow one to specify the value to be stored as well as the key.

```
var persons = new[] {
    new { Name="Fizz", Id=1},
    new { Name="Buzz", Id=2},
    new { Name="Foo", Id=3},
    new { Name="Bar", Id=4},
};
```

Specifying just a key selector function will create a `Dictionary<TKey,TVal>` with `TKey` the return Type of the key selector, `TVal` the original object Type, and the original object as the stored value.

```
var personsById = persons.ToDictionary(p => p.Id);
// personsById is a Dictionary<int,object>

Console.WriteLine(personsById[1].Name); //Fizz
Console.WriteLine(personsById[2].Name); //Buzz
```

Specifying a value selector function as well will create a `Dictionary<TKey,TVal>` with `TKey` still the return type of the key selector, but `TVal` now the return type of the value selector function, and the returned value as the stored value.

```
var namesById = persons.ToDictionary(p => p.Id, p => p.Name);
//namesById is a Dictionary<int,string>

Console.WriteLine(namesById[3]); //Foo
Console.WriteLine(namesById[4]); //Bar
```

As stated above, the keys returned by the key selector must be unique. The following will throw an exception.

```
var persons = new[] {
    new { Name="Fizz", Id=1},
    new { Name="Buzz", Id=2},
    new { Name="Foo", Id=3},
    new { Name="Bar", Id=4},
    new { Name="Oops", Id=4}
};

var willThrowException = persons.ToDictionary(p => p.Id)
```

If a unique key can not be given for the source collection, consider using ToLookup instead. On the surface, ToLookup behaves similarly to ToDictionary, however, in the resulting Lookup each key is paired with a collection of values with matching keys.

Section 8.40: Union

```
var numbers1to5 = new[] {1,2,3,4,5};
var numbers4to8 = new[] {4,5,6,7,8};

var numbers1to8 = numbers1to5.Union(numbers4to8);

Console.WriteLine(string.Join(",", numbers1to8));

//1,2,3,4,5,6,7,8
```

Note that duplicates are removed from the result. If this is undesirable, use `Concat` instead.

第8.41节：转换为数组（ToArray）

```
var numbers = new[] {1,2,3,4,5,6,7,8,9,10};
var someNumbers = numbers.Where(n => n < 6);

Console.WriteLine(someNumbers.GetType().Name);
//WhereArrayIterator`1

var someNumbersArray = someNumbers.ToArray();

Console.WriteLine(someNumbersArray.GetType().Name);
//Int32[]
```

第8.42节：ToList

```
var numbers = new[] {1,2,3,4,5,6,7,8,9,10};
var someNumbers = numbers.Where(n => n < 6);

Console.WriteLine(someNumbers.GetType().Name);
//WhereArrayIterator`1

var someNumbersList = someNumbers.ToList();

Console.WriteLine(
someNumbersList.GetType().Name + " - " +
    someNumbersList.GetType().GetGenericArguments()[0].Name);
//List`1 - Int32
```

第8.43节：ElementAt

```
var names = new[] {"Foo","Bar","Fizz","Buzz"};

var thirdName = names.ElementAt(2);
Console.WriteLine(thirdName);  //Fizz

//以下代码会抛出ArgumentOutOfRangeException异常

var minusOnethName = names.ElementAt(-1);
var fifthName = names.ElementAt(4);
```

第8.44节：ElementAtOrDefault

```
var names = new[] {"Foo","Bar","Fizz","Buzz"};

var thirdName = names.ElementAtOrDefault(2);
Console.WriteLine(thirdName); //Fizz

var minusOnethName = names.ElementAtOrDefault(-1);
Console.WriteLine(minusOnethName); //null

var fifthName = names.ElementAtOrDefault(4);
Console.WriteLine(fifthName); //null
```

第8.45节：SkipWhile

```
var numbers = new[] {2,4,6,8,1,3,5,7};
```


Section 8.41: ToArray

```
var numbers = new[] {1,2,3,4,5,6,7,8,9,10};
var someNumbers = numbers.Where(n => n < 6);

Console.WriteLine(someNumbers.GetType().Name);
//WhereArrayIterator`1

var someNumbersArray = someNumbers.ToArray();

Console.WriteLine(someNumbersArray.GetType().Name);
//Int32[]
```

Section 8.42: ToList

```
var numbers = new[] {1,2,3,4,5,6,7,8,9,10};
var someNumbers = numbers.Where(n => n < 6);

Console.WriteLine(someNumbers.GetType().Name);
//WhereArrayIterator`1

var someNumbersList = someNumbers.ToList();

Console.WriteLine(
    someNumbersList.GetType().Name + " - " +
    someNumbersList.GetType().GetGenericArguments()[0].Name);
//List`1 - Int32
```

Section 8.43: ElementAt

```
var names = new[] {"Foo","Bar","Fizz","Buzz"};

var thirdName = names.ElementAt(2);
Console.WriteLine(thirdName); //Fizz

//The following throws ArgumentOutOfRangeException

var minusOnethName = names.ElementAt(-1);
var fifthName = names.ElementAt(4);
```

Section 8.44: ElementAtOrDefault

```
var names = new[] {"Foo","Bar","Fizz","Buzz"};

var thirdName = names.ElementAtOrDefault(2);
Console.WriteLine(thirdName); //Fizz

var minusOnethName = names.ElementAtOrDefault(-1);
Console.WriteLine(minusOnethName); //null

var fifthName = names.ElementAtOrDefault(4);
Console.WriteLine(fifthName); //null
```

Section 8.45: SkipWhile

```
var numbers = new[] {2,4,6,8,1,3,5,7};
```

```
var oddNumbers = numbers.SkipWhile(n => (n & 1) == 0);

Console.WriteLine(string.Join(",", oddNumbers.ToArray()));

//1,3,5,7
```

第8.46节：TakeWhile

```
var numbers = new[] {2,4,6,1,3,5,7,8};

var evenNumbers = numbers.TakeWhile(n => (n & 1) == 0);

Console.WriteLine(string.Join(",", evenNumbers.ToArray()));

//2,4,6
```

第8.47节：DefaultIfEmpty

```
var numbers = new[] {2,4,6,8,1,3,5,7};

var numbersOrDefault = numbers.DefaultIfEmpty();
Console.WriteLine(numbers.SequenceEqual(numbersOrDefault)); //True

var noNumbers = new int[0];

var noNumbersOrDefault = noNumbers.DefaultIfEmpty();
Console.WriteLine(noNumbersOrDefault.Count()); //1
Console.WriteLine(noNumbersOrDefault.Single()); //0

var noNumbersOrExplicitDefault = noNumbers.DefaultIfEmpty(34);
Console.WriteLine(noNumbersOrExplicitDefault.Count()); //1
Console.WriteLine(noNumbersOrExplicitDefault.Single()); //34
```

第8.48节：连接（Join）

```
class Developer
{
    public int Id { get; set; }
    public string 名称 { get; set; }
}

class Project
{
    public int DeveloperId { get; set; }
    public string Name { get; set; }
}

var developers = new[] {
    new Developer {
Id = 1,
名称 = "Foobuzz"
    },
    new 开发者 {
       Id = 2,
       名称 = "Barfizz"
    }
};

var projects = new[] {
```



```
var oddNumbers = numbers.SkipWhile(n => (n & 1) == 0);

Console.WriteLine(string.Join(",", oddNumbers.ToArray()));

//1,3,5,7
```

Section 8.46: TakeWhile

```
var numbers = new[] {2,4,6,1,3,5,7,8};

var evenNumbers = numbers.TakeWhile(n => (n & 1) == 0);

Console.WriteLine(string.Join(",", evenNumbers.ToArray()));

//2,4,6
```

Section 8.47: DefaultIfEmpty

```
var numbers = new[] {2,4,6,8,1,3,5,7};

var numbersOrDefault = numbers.DefaultIfEmpty();
Console.WriteLine(numbers.SequenceEqual(numbersOrDefault)); //True

var noNumbers = new int[0];

var noNumbersOrDefault = noNumbers.DefaultIfEmpty();
Console.WriteLine(noNumbersOrDefault.Count()); //1
Console.WriteLine(noNumbersOrDefault.Single()); //0

var noNumbersOrExplicitDefault = noNumbers.DefaultIfEmpty(34);
Console.WriteLine(noNumbersOrExplicitDefault.Count()); //1
Console.WriteLine(noNumbersOrExplicitDefault.Single()); //34
```

Section 8.48: Join

```
class Developer
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Project
{
    public int DeveloperId { get; set; }
    public string Name { get; set; }
}

var developers = new[] {
    new Developer {
        Id = 1,
        Name = "Foobuzz"
    },
    new Developer {
        Id = 2,
        Name = "Barfizz"
    }
};

var projects = new[] {
```

```
    new 项目 {
开发者Id = 1,
        名称 = "Hello World 3D"
    },
    new 项目 {
开发者Id = 1,
        名称 = "Super Fizzbuzz Maker"
    },
    new 项目 {
开发者Id = 2,
        名称 = "Citizen Kane - The action game"
    },
    new 项目 {
开发者Id = 2,
        名称 = "Pro Pong 2016"
    }
};

var denormalized = developers.Join(
    inner: projects,
outerKeySelector: dev => dev.Id,
    innerKeySelector: proj => proj.DeveloperId,
    resultSelector:
        (dev, proj) => new {
            ProjectName = proj.Name,
            DeveloperName = dev.Name});

foreach(var item in denormalized)
{
Console.WriteLine("{0} by {1}", item.ProjectName, item.DeveloperName);
}

//Hello World 3D by Foobuzz
//Super Fizzbuzz Maker by Foobuzz
//Citizen Kane - The action game by Barfizz
//Pro Pong 2016 by Barfizz
```

第8.49节：左外连接（Left Outer Join）

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

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

public static void Main(string[] args)
{
    var magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    var terry = new Person { FirstName = "Terry", LastName = "Adams" };

    var barley = new Pet { Name = "Barley", Owner = terry };

    var people = new[] { magnus, terry };
    var pets = new[] { barley };
```



```
    new Project {
        DeveloperId = 1,
        Name = "Hello World 3D"
    },
    new Project {
        DeveloperId = 1,
        Name = "Super Fizzbuzz Maker"
    },
    new Project {
        DeveloperId = 2,
        Name = "Citizen Kane - The action game"
    },
    new Project {
        DeveloperId = 2,
        Name = "Pro Pong 2016"
    }
};

var denormalized = developers.Join(
    inner: projects,
    outerKeySelector: dev => dev.Id,
    innerKeySelector: proj => proj.DeveloperId,
    resultSelector:
        (dev, proj) => new {
            ProjectName = proj.Name,
            DeveloperName = dev.Name});

foreach(var item in denormalized)
{
    Console.WriteLine("{0} by {1}", item.ProjectName, item.DeveloperName);
}

//Hello World 3D by Foobuzz
//Super Fizzbuzz Maker by Foobuzz
//Citizen Kane - The action game by Barfizz
//Pro Pong 2016 by Barfizz
```

Section 8.49: Left Outer Join

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

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

public static void Main(string[] args)
{
    var magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };
    var terry = new Person { FirstName = "Terry", LastName = "Adams" };

    var barley = new Pet { Name = "Barley", Owner = terry };

    var people = new[] { magnus, terry };
    var pets = new[] { barley };
```

```
    var query =
        from person in people
        join pet in pets on person equals pet.Owner into gj
        from subpet in gj.DefaultIfEmpty()
        select new
        {
person.FirstName,
PetName = subpet?.Name ?? "-" // 如果他没有宠物则使用 -
        };

    foreach (var p in query)
Console.WriteLine($"{p.FirstName}: {p.PetName}");
}
```

```
    var query =
        from person in people
        join pet in pets on person equals pet.Owner into gj
        from subpet in gj.DefaultIfEmpty()
        select new
        {
            person.FirstName,
            PetName = subpet?.Name ?? "-" // Use - if he has no pet
        };

    foreach (var p in query)
        Console.WriteLine($"{p.FirstName}: {p.PetName}");
}
```

第9章：ForEach

第9.1节：IEnumerable的扩展方法

ForEach() 定义在 List<T> 类上，但不定义在 IQueryable<T> 或 IEnumerable<T> 上。在这些情况下你有两个选择：

先调用ToList

枚举（或查询）将被执行，结果复制到一个新列表或调用数据库。然后对每个项目调用该方法。

```
IEnumerable<Customer> customers = new List<Customer>();

customers.ToList().ForEach(c => c.SendEmail());
```

该方法存在明显的内存使用开销，因为会创建一个中间列表。

扩展方法

编写一个扩展方法：

```
public static void ForEach<T>(this IEnumerable<T> enumeration, Action<T> action)
{
    foreach(T item in enumeration)
    {
action(item);
    }
}
```

用法：

```
IEnumerable<Customer> customers = new List<Customer>();

customers.ForEach(c => c.SendEmail());
```

注意：框架的 LINQ 方法设计时意图保持纯粹性，即它们不会产生副作用。ForEach方法的唯一目的是产生副作用，这一点与其他方法不同。你可以考虑直接使用普通的foreach循环。

第9.2节：在列表中的对象上调用方法

```
public class Customer {
   public void SendEmail()
   {
       // 发送邮件代码
   }
}

List<Customer> customers = new List<Customer>();

customers.Add(new Customer());
customers.Add(new Customer());

customers.ForEach(c => c.SendEmail());
```


Chapter 9: ForEach

Section 9.1: Extension method for IEnumerable

`ForEach()` is defined on the `List<T>` class, but not on `IQueryable<T>` or `IEnumerable<T>`. You have two choices in those cases:

ToList first

The enumeration (or query) will be evaluated, copying the results into a new list or calling the database. The method is then called on each item.

```
IEnumerable<Customer> customers = new List<Customer>();

customers.ToList().ForEach(c => c.SendEmail());
```

This method has obvious memory usage overhead, as an intermediate list is created.

Extension method

Write an extension method:

```
public static void ForEach<T>(this IEnumerable<T> enumeration, Action<T> action)
{
    foreach(T item in enumeration)
    {
        action(item);
    }
}
```

Use:

```
IEnumerable<Customer> customers = new List<Customer>();

customers.ForEach(c => c.SendEmail());
```

Caution: The Framework's LINQ methods have been designed with the intention of being *pure*, which means they do not produce side effects. The `ForEach` method's only purpose is to produce side effects, and deviates from the other methods in this aspect. You may consider just using a plain `foreach` loop instead.

Section 9.2: Calling a method on an object in a list

```
public class Customer {
   public void SendEmail()
   {
       // Sending email code here
   }
}

List<Customer> customers = new List<Customer>();

customers.Add(new Customer());
customers.Add(new Customer());

customers.ForEach(c => c.SendEmail());
```

第10章：反射

第10.1节：什么是程序集？

程序集是任何公共语言运行时（CLR）应用程序的构建块。你定义的每个类型，
连同其方法、属性及其字节码，都会被编译并打包到程序集内。

```
using System.Reflection;
```

```
Assembly assembly = this.GetType().Assembly;
```

程序集是自我描述的：它们不仅包含类型、方法及其IL代码，还包含在编译和运行时
检查和使用它们所需的元数据：

```
Assembly assembly = Assembly.GetExecutingAssembly();

foreach (var type in assembly.GetTypes())
{
Console.WriteLine(type.FullName);
}
```

程序集有描述其完整唯一身份的名称：

```
Console.WriteLine(typeof(int).Assembly.FullName);
// 将打印："mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
```

如果此名称包含PublicKeyToken，则称为强名称。为程序集强命名是通过使用与程序集一起分发的公钥对应的私钥
创建签名的过程。此签名被添加到程序集清单中，清单包含组成程序集的所有文件的名称和哈希值，其PublicKeyToken
成为名称的一部分。具有相同强名称的程序集应当是相同的；强名称用于版本控制和防止程序集冲突。

第10.2节：使用反射比较两个对象

```
public class Equatable
{
    public string field1;

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

        var type = obj.GetType();
        if (GetType() != type)
            return false;

        var fields = type.GetFields(BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.Public);
        foreach (var field in fields)
            if (field.GetValue(this) != field.GetValue(obj))
                return false;

        return true;
    }
```


Chapter 10: Reflection

Section 10.1: What is an Assembly?

Assemblies are the building block of any Common Language Runtime (CLR) application. Every type you define, together with its methods, properties and their bytecode, is compiled and packaged inside an Assembly.

```
using System.Reflection;
```

```
Assembly assembly = this.GetType().Assembly;
```

Assemblies are self-documenting: they do not only contain types, methods and their IL code, but also the Metadata necessary to inspect and consume them, both at compile and runtime:

```
Assembly assembly = Assembly.GetExecutingAssembly();

foreach (var type in assembly.GetTypes())
{
    Console.WriteLine(type.FullName);
}
```

Assemblies have names which describes their full, unique identity:

```
Console.WriteLine(typeof(int).Assembly.FullName);
// Will print: "mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
```

If this name includes a `PublicKeyToken`, it is called a *strong name*. Strong-naming an assembly is the process of creating a signature by using the private key that corresponds to the public key distributed with the assembly. This signature is added to the Assembly manifest, which contains the names and hashes of all the files that make up the assembly, and its `PublicKeyToken` becomes part of the name. Assemblies that have the same strong name should be identical; strong names are used in versioning and to prevent assembly conflicts.

Section 10.2: Compare two objects with reflection

```
public class Equatable
{
    public string field1;

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

        var type = obj.GetType();
        if (GetType() != type)
            return false;

        var fields = type.GetFields(BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.Public);
        foreach (var field in fields)
            if (field.GetValue(this) != field.GetValue(obj))
                return false;

        return true;
    }
```

```
    public override int GetHashCode()
    {
        var accumulator = 0;
        var fields = GetType().GetFields(BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.Public);
        foreach (var field in fields)
accumulator = unchecked ((accumulator * 937) ^ field.GetValue(this).GetHashCode());

        return accumulator;
    }
}
```

注意： 这个示例为了简化，进行了基于字段的比较（忽略静态字段和属性）

第10.3节：使用反射创建对象和设置属性

假设我们有一个类Classy，具有属性Propertua

```
public class Classy
{
    public string Propertua {get; set;}
}
```

使用反射设置Propertua：

```
var typeOfClassy = typeof (Classy);
var classy = new Classy();
var prop = typeOfClassy.GetProperty("Propertua");
prop.SetValue(classy, "Value");
```

第10.4节：如何使用反射创建T类型的对象

使用默认构造函数

```
T变量 = Activator.CreateInstance(typeof(T));
```

使用带参数的构造函数

```
T 变量 = Activator.CreateInstance(typeof(T), arg1, arg2);
```

第10.5节：使用反射获取枚举的属性（并进行缓存）

属性对于标注枚举的元数据非常有用。获取其值可能较慢，因此缓存结果非常重要。

```
    private static Dictionary<object, object> attributeCache = new Dictionary<object, object>();

    public static T GetAttribute<T, V>(this V value)
        where T : Attribute
        where V : struct
    {
        object temp;

        // 尝试从静态缓存中获取值。
```



```
    public override int GetHashCode()
    {
        var accumulator = 0;
        var fields = GetType().GetFields(BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.Public);
        foreach (var field in fields)
            accumulator = unchecked ((accumulator * 937) ^ field.GetValue(this).GetHashCode());

        return accumulator;
    }
}
```

Note: this example do a field based comparasion (ignore static fields and properties) for simplicity

Section 10.3: Creating Object and setting properties using reflection

Lets say we have a class `Classy` that has property Propertua

```
public class Classy
{
    public string Propertua {get; set;}
}
```

to set `Propertua` using reflection:

```
var typeOfClassy = typeof (Classy);
var classy = new Classy();
var prop = typeOfClassy.GetProperty("Propertua");
prop.SetValue(classy, "Value");
```

Section 10.4: How to create an object of T using Reflection

Using the default constructor

```
T variable = Activator.CreateInstance(typeof(T));
```

Using parameterized constructor

```
T variable = Activator.CreateInstance(typeof(T), arg1, arg2);
```

Section 10.5: Getting an attribute of an enum with reflection (and caching it)

Attributes can be useful for denoting metadata on enums. Getting the value of this can be slow, so it is important to cache results.

```
    private static Dictionary<object, object> attributeCache = new Dictionary<object, object>();

    public static T GetAttribute<T, V>(this V value)
        where T : Attribute
        where V : struct
    {
        object temp;

        // Try to get the value from the static cache.
```

```
        if (attributeCache.TryGetValue(value, out temp))
        {
            return (T) temp;
        }
        否则
        {
            // 获取传入结构体的类型。
Type type = value.GetType();
            FieldInfo fieldInfo = type.GetField(value.ToString());

            // 获取结构体上指定类型的自定义属性。
T[] attribs = (T[])fieldInfo.GetCustomAttributes(typeof(T), false);

            // 如果匹配则返回第一个。
            var result = attribs.Length > 0 ? attribs[0] : null;

            // 缓存结果，以便后续检查无需反射。
attributeCache.Add(value, result);

            return result;
        }
    }
```



```
        if (attributeCache.TryGetValue(value, out temp))
        {
            return (T) temp;
        }
        else
        {
            // Get the type of the struct passed in.
            Type type = value.GetType();
            FieldInfo fieldInfo = type.GetField(value.ToString());

            // Get the custom attributes of the type desired found on the struct.
            T[] attribs = (T[])fieldInfo.GetCustomAttributes(typeof(T), false);

            // Return the first if there was a match.
            var result = attribs.Length > 0 ? attribs[0] : null;

            // Cache the result so future checks won't need reflection.
            attributeCache.Add(value, result);

            return result;
        }
    }
```

第11章：表达式树

第11.1节：构建形式为字段 == 值的谓词

在运行时构建类似 `_ => _.字段 == "值"` 的表达式。

给定一个谓词 `_ => _.字段` 和一个字符串值 "值"，创建一个表达式来测试该谓词是否为真。

该表达式适用于：

- IQueryable<T>，IEnumerable<T> 用于测试谓词。
- 实体框架或 Linq 到 SQL 用于创建测试谓词的 Where 子句。

此方法将构建一个合适的 Equal 表达式，用于测试 字段 是否等于 "值"。

```
public static Expression<Func<T, bool>> BuildEqualPredicate<T>(
    Expression<Func<T, string>> memberAccessor,
    string term)
{
    var toString = Expression.Convert(Expression.Constant(term), typeof(string));
    Expression expression = Expression.Equal(memberAccessor.Body, toString);
    var predicate = Expression.Lambda<Func<T, bool>>(
表达式,
        memberAccessor.参数);
    返回 谓词;
}
```

该谓词可以通过在Where扩展方法中包含谓词来使用。

```
var predicate = PredicateExtensions.BuildEqualPredicate<Entity>(
    _ => _.Field,
    "VALUE");
var results = context.Entity.Where(predicate).ToList();
```

第11.2节：由C#编译器生成的简单表达式树

考虑以下C#代码

```
Expression<Func<int, int>> expression = a => a + 1;
```

因为C#编译器看到该lambda表达式被赋值给Expression类型而非委托类型，它生成了一个大致等同于以下代码的表达式树

```
ParameterExpression parameterA = Expression.Parameter(typeof(int), "a");
var expression = (Expression<Func<int, int>>)Expression.Lambda(
Expression.Add(
                                        parameterA,
                                                Expression.Constant(1)),
                                            parameterA);
```

树的根是包含主体和参数列表的lambda表达式。该lambda有1个名为“a”的参数。主体是一个CLR类型为BinaryExpression且NodeType为Add的单一表达式。该表达式表示加法。它有两个子表达式，分别称为Left和Right。Left是

Chapter 11: Expression Trees

Section 11.1: building a predicate of form field == value

To build up an expression like `_ => _.Field == "VALUE"` at runtime.

Given a predicate `_ => _.Field` and a string value `"VALUE"`, create an expression that tests whether or not the predicate is true.

The expression is suitable for:

- `IQueryable<T>`, `IEnumerable<T>` to test the predicate.
- entity framework or `Linq` to `SQL` to create a `Where` clause that tests the predicate.

This method will build an appropriate `Equal` expression that tests whether or not `Field` equals `"VALUE"`.

```
public static Expression<Func<T, bool>> BuildEqualPredicate<T>(
    Expression<Func<T, string>> memberAccessor,
    string term)
{
    var toString = Expression.Convert(Expression.Constant(term), typeof(string));
    Expression expression = Expression.Equal(memberAccessor.Body, toString);
    var predicate = Expression.Lambda<Func<T, bool>>(
        expression,
        memberAccessor.Parameters);
    return predicate;
}
```

The predicate can be used by including the predicate in a `Where` extension method.

```
var predicate = PredicateExtensions.BuildEqualPredicate<Entity>(
    _ => _.Field,
    "VALUE");
var results = context.Entity.Where(predicate).ToList();
```

Section 11.2: Simple Expression Tree Generated by the C# Compiler

Consider the following C# code

```
Expression<Func<int, int>> expression = a => a + 1;
```

Because the C# compiler sees that the lambda expression is assigned to an Expression type rather than a delegate type it generates an expression tree roughly equivalent to this code

```
ParameterExpression parameterA = Expression.Parameter(typeof(int), "a");
var expression = (Expression<Func<int, int>>)Expression.Lambda(
                                                Expression.Add(
                                                    parameterA,
                                                    Expression.Constant(1)),
                                                parameterA);
```

The root of the tree is the lambda expression which contains a body and a list of parameters. The lambda has 1 parameter called "a". The body is a single expression of CLR type BinaryExpression and NodeType of Add. This expression represents addition. It has two subexpressions denoted as Left and Right. Left is the

参数“a”的ParameterExpression，Right是值为1的ConstantExpression。

该表达式最简单的用法是打印它：

```
Console.WriteLine(expression); //打印 a => (a + 1)
```

它打印出等价的C#代码。

表达式树可以编译成C#委托并由CLR执行

```
Func<int, int> lambda = expression.Compile();
Console.WriteLine(lambda(2)); //打印 3
```

通常表达式会被翻译成其他语言如SQL，但也可以用来调用公共或非公共类型的私有、受保护和内部成员，作为反射的替代方案。

第11.3节：用于获取静态字段的表达式

有如下示例类型：

```
public TestClass
{
    public static string StaticPublicField = "StaticPublicFieldValue";
}
```

我们可以获取 StaticPublicField 的值：

```
var fieldExpr = Expression.Field(null, typeof(TestClass), "StaticPublicField");
var labmda = Expression.Lambda<Func<string>>(fieldExpr);
```

然后它可以被编译成委托，用于获取字段值。

```
Func<string> retriever = lambda.Compile();
var fieldValue = retriever();
```

//fieldValue 的结果是 StaticPublicFieldValue

第11.4节：InvocationExpression 类

InvocationExpression 类允许调用同一 Expression 树中作为部分的其他 lambda 表达式。

你可以使用静态方法 Expression.Invoke 来创建它们。

问题 我们想要获取描述中包含“car”的项目。在搜索字符串之前需要先检查是否为 null，但我们不希望过度调用它，因为计算可能很昂贵。

```
using System;
using System.Linq;
using System.Linq.Expressions;

public class Program
{
    public static void Main()
    {
        var elements = new[] {
```


ParameterExpression for the parameter "a" and Right is a ConstantExpression with the value 1.

The simplest usage of this expression is printing it:

```
Console.WriteLine(expression); //prints a => (a + 1)
```

Which prints the equivalent C# code.

The expression tree can be compiled into a C# delegate and executed by the CLR

```
Func<int, int> lambda = expression.Compile();
Console.WriteLine(lambda(2)); //prints 3
```

Usually expressions are translated to other languages like SQL, but can be also used to invoke private, protected and internal members of public or non-public types as alternative to Reflection.

Section 11.3: Expression for retrieving a static field

Having example type like this:

```
public TestClass
{
    public static string StaticPublicField = "StaticPublicFieldValue";
}
```

We can retrieve value of StaticPublicField:

```
var fieldExpr = Expression.Field(null, typeof(TestClass), "StaticPublicField");
var labmda = Expression.Lambda<Func<string>>(fieldExpr);
```

It can be then i.e. compiled into a delegate for retrieving field value.

```
Func<string> retriever = lambda.Compile();
var fieldValue = retriever();
```

//fieldValue result is StaticPublicFieldValue

Section 11.4: InvocationExpression Class

InvocationExpression class allows invocation of other lambda expressions that are parts of the same Expression tree.

You create them with static `Expression.Invoke` method.

Problem We want to get on the items which have "car" in their description. We need to check it for null before searching for a string inside but we don't want it to be called excessively, as the computation could be expensive.

```
using System;
using System.Linq;
using System.Linq.Expressions;

public class Program
{
    public static void Main()
    {
        var elements = new[] {
```

```
            new Element { Description = "car" },
            new Element { Description = "cargo" },
            new Element { Description = "wheel" },
            new Element { Description = null },
            new Element { Description = "Madagascar" },
        };

        var elementIsInterestingExpression = CreateSearchPredicate(
            searchTerm: "car",
whereToSearch: (Element e) => e.Description);

        Console.WriteLine(elementIsInterestingExpression.ToString());

        var elementIsInteresting = elementIsInterestingExpression.Compile();
        var interestingElements = elements.Where(elementIsInteresting);
        foreach (var e in interestingElements)
        {
Console.WriteLine(e.Description);
        }

        var countExpensiveComputations = 0;
        Action incCount = () => countExpensiveComputations++;
       elements
            .其中(
CreateSearchPredicate(
                    "car",
                    (Element e) => ExpensivelyComputed(
                    e, incCount
                    )
                ).Compile()
            )
            .Count();

Console.WriteLine("属性提取器被调用了 {0} 次。", countExpensiveComputations);
    }

    private class Element
    {
        public string Description { get; set; }
    }

    private static string ExpensivelyComputed(Element source, Action count)
    {
count();
        return source.Description;
    }

    private static Expression<Func<T, bool>> CreateSearchPredicate<T>(
            string searchTerm,
Expression<Func<T, string>> whereToSearch)
    {
        var extracted = Expression.Parameter(typeof(string), "extracted");

Expression<Func<string, bool>> coalesceNullCheckWithSearch =
            Expression.Lambda<Func<string, bool>>(
Expression.AndAlso(
Expression.Not(
Expression.Call(typeof(string), "IsNullOrEmpty", null, extracted)
                    ),
Expression.Call(extracted, "Contains", null, Expression.Constant(searchTerm))
                ),
extracted);
```



```
            new Element { Description = "car" },
            new Element { Description = "cargo" },
            new Element { Description = "wheel" },
            new Element { Description = null },
            new Element { Description = "Madagascar" },
        };

        var elementIsInterestingExpression = CreateSearchPredicate(
            searchTerm: "car",
            whereToSearch: (Element e) => e.Description);

        Console.WriteLine(elementIsInterestingExpression.ToString());

        var elementIsInteresting = elementIsInterestingExpression.Compile();
        var interestingElements = elements.Where(elementIsInteresting);
        foreach (var e in interestingElements)
        {
            Console.WriteLine(e.Description);
        }

        var countExpensiveComputations = 0;
        Action incCount = () => countExpensiveComputations++;
        elements
            .Where(
                CreateSearchPredicate(
                    "car",
                    (Element e) => ExpensivelyComputed(
                        e, incCount
                    )
                ).Compile()
            )
            .Count();

        Console.WriteLine("Property extractor is called {0} times.", countExpensiveComputations);
    }

    private class Element
    {
        public string Description { get; set; }
    }

    private static string ExpensivelyComputed(Element source, Action count)
    {
        count();
        return source.Description;
    }

    private static Expression<Func<T, bool>> CreateSearchPredicate<T>(
            string searchTerm,
            Expression<Func<T, string>> whereToSearch)
    {
        var extracted = Expression.Parameter(typeof(string), "extracted");

        Expression<Func<string, bool>> coalesceNullCheckWithSearch =
            Expression.Lambda<Func<string, bool>>(
                Expression.AndAlso(
                    Expression.Not(
                        Expression.Call(typeof(string), "IsNullOrEmpty", null, extracted)
                    ),
                    Expression.Call(extracted, "Contains", null, Expression.Constant(searchTerm))
                ),
                extracted);
```

```
        var elementParameter = Expression.Parameter(typeof(T), "element");

        return Expression.Lambda<Func<T, bool>>(
Expression.Invoke(
coalesceNullCheckWithSearch,
                Expression.Invoke(whereToSearch, elementParameter)
            ),
elementParameter
        );
    }
}
```

输出

```
element => Invoke(extracted => (Not(IsNullOrEmpty(extracted)) AndAlso extracted.Contains("car")),
Invoke(e => e.Description, element))
汽车
货物
马达加斯加
谓词被调用了5次。
```

首先要注意的是实际的属性访问，是被封装在一个 Invoke 中的：

```
Invoke(e => e.Description, element)
```

，这是唯一接触到 `e.Description` 的部分，取而代之的是一个类型为 string 的 extracted 参数被传递给下一个部分：

```
(Not(IsNullOrEmpty(extracted)) AndAlso extracted.Contains("car"))
```

这里另一个重要的点是 AndAlso。它只计算左边部分，如果第一部分返回“false”。常见的错误是使用按位运算符“And”代替它，后者总是计算两部分，在本例中会导致 NullReferenceException 错误。


```
        var elementParameter = Expression.Parameter(typeof(T), "element");

        return Expression.Lambda<Func<T, bool>>(
            Expression.Invoke(
                coalesceNullCheckWithSearch,
                Expression.Invoke(whereToSearch, elementParameter)
            ),
            elementParameter
        );
    }
}
```

Output

```
element => Invoke(extracted => (Not(IsNullOrEmpty(extracted)) AndAlso extracted.Contains("car")),
Invoke(e => e.Description, element))
car
cargo
Madagascar
Predicate is called 5 times.
```

First thing to note is how the actual propery access, wrapped in an Invoke:

```
Invoke(e => e.Description, element)
```

, and this is the only part that touches `e.Description`, and in place of it, `extracted` parameter of type `string` is passed to the next one:

```
(Not(IsNullOrEmpty(extracted)) AndAlso extracted.Contains("car"))
```

Another important thing to note here is `AndAlso`. It computes only the left part, if the first part returns 'false'. It's a common mistake to use the bitwise operator 'And' instead of it, which always computes both parts, and would fail with a NullReferenceException in this example.

第12章：自定义类型

第12.1节：结构体定义

结构体继承自 System.ValueType，是值类型，存储在栈上。当值类型作为参数传递时，是按值传递的。

```
结构体 MyStruct
{
    public int x;
    public int y;
}
```

按值传递 意味着参数的值被复制给方法，且在方法中对参数所做的任何更改都不会反映到方法外部。例如，考虑以下代码，它调用了一个名为AddNumbers的方法，传入了类型为int的变量 a和 b，这是一种值类型。

```
int a = 5;
int b = 6;

AddNumbers(a,b);

public AddNumbers(int x, int y)
{
    int z = x + y; // z 变为 11
    x = x + 5; // 现在我们将 x 改为 10
    z = x + y; // 现在 z 变为 16
}
```

尽管我们在方法内部给 x 加了 5， a 的值仍然保持不变，因为它是值类型，
这意味着 x 是 *a* 值的副本，而不是真正的 a。

记住，值类型存储在栈上，并且是按值传递的。

第12.2节：类定义

类继承自 System.Object，是引用类型，存储在堆上。当引用类型作为参数传递时，按引用传递。

```
public Class MyClass
{
    public int a;
    public int b;
}
```

通过引用传递 意味着将参数的引用传递给方法，对参数的任何更改都会在方法返回后反映到方法外部，因为该引用指向内存中完全相同的对象。让我们使用之前的相同示例，但先将int包装在一个类中。

```
MyClass instanceOfMyClass = new MyClass();
instanceOfMyClass.a = 5;
instanceOfMyClass.b = 6;

AddNumbers(instanceOfMyClass);

public AddNumbers(MyClass sample)
{
```


Chapter 12: Custom Types

Section 12.1: Struct Definition

Structs inherit from System.ValueType, are value types, and live on the stack. When value types are passed as a parameter, they are passed by value.

```
Struct MyStruct
{
    public int x;
    public int y;
}
```

Passed by value means that the value of the parameter is *copied* for the method, and any changes made to the parameter in the method are not reflected outside of the method. For instance, consider the following code, which calls a method named `AddNumbers`, passing in the variables `a` and `b`, which are of type `int`, which is a Value type.

```
int a = 5;
int b = 6;

AddNumbers(a,b);

public AddNumbers(int x, int y)
{
    int z = x + y; // z becomes 11
    x = x + 5; // now we changed x to be 10
    z = x + y; // now z becomes 16
}
```

Even though we added 5 to `x` inside the method, the value of `a` remains unchanged, because it's a Value type, and that means `x` was a *copy* of `a`'s value, but not actually `a`.

Remember, Value types live on the stack, and are passed by value.

Section 12.2: Class Definition

Classes inherit from System.Object, are reference types, and live on the heap. When reference types are passed as a parameter, they are passed by reference.

```
public Class MyClass
{
    public int a;
    public int b;
}
```

Passed by reference means that a *reference* to the parameter is passed to the method, and any changes to the parameter will be reflected outside of the method when it returns, because the reference is *to the exact same object in memory*. Let's use the same example as before, but we'll "wrap" the `int`s in a class first.

```
MyClass instanceOfMyClass = new MyClass();
instanceOfMyClass.a = 5;
instanceOfMyClass.b = 6;

AddNumbers(instanceOfMyClass);

public AddNumbers(MyClass sample)
{
```

```
    int z = sample.a + sample.b; // z 变为 11
    sample.a = sample.a + 5; // 现在我们将 a 改为 10
    z = sample.a + sample.b; // 现在 z 变为 16
}
```

这次，当我们将 sample.a 改为 10 时，instanceOfMyClass.a 的值也发生了变化，因为它是通过引用传递的。通过引用传递意味着传递给方法的是对象的引用（有时也称为指针），而不是对象本身的副本。

请记住，引用类型存储在堆上，并且是通过引用传递的。


```
    int z = sample.a + sample.b; // z becomes 11
    sample.a = sample.a + 5; // now we changed a to be 10
    z = sample.a + sample.b; // now z becomes 16
}
```

This time, when we changed `sample.a` to `10`, the value of `instanceOfMyClass.a` *also* changes, because it was *passed by reference*. Passed by reference means that a *reference* (also sometimes called a *pointer*) to the object was passed into the method, instead of a copy of the object itself.

Remember, Reference types live on the heap, and are passed by reference.

第13章：代码契约

第13.1节：接口的契约

使用代码契约可以将契约应用于接口。这是通过声明一个实现接口的抽象类来完成的。接口应标记为ContractClassAttribute，契约定义（抽象类）应标记为ContractClassForAttribute

C# 示例...

```
[ContractClass(typeof(MyInterfaceContract))]
public interface IMyInterface
{
    string DoWork(string input);
}
//切勿继承此合约定义类
[ContractClassFor(typeof(IMyInterface))]
internal abstract class MyInterfaceContract : IMyInterface
{
    private MyInterfaceContract() { }

    public string DoWork(string input)
    {
Contract.Requires(!string.IsNullOrEmpty(input));
        Contract.Ensures(!string.IsNullOrEmpty(Contract.Result<string>()));
        throw new NotSupportedException();
    }
}
public class MyInterfaceImplmentation : IMyInterface
{
    public string DoWork(string input)
    {
        return input;
    }
}
```

静态分析结果...

```
var m = new MyInterfaceImplmentation();
var ret = m.DoWork(null);
```

CodeContracts: requires is false: !string.IsNullOrEmpty(input)

第13.2节：安装和启用代码合约

虽然System.Diagnostics.Contracts包含在.Net框架中。要使用代码契约，您必须安装Visual Studio扩展。

在扩展和更新中搜索代码契约，然后安装代码契约工具

Chapter 13: Code Contracts

Section 13.1: Contracts for Interfaces

Using Code Contracts it is possible to apply a contract to an interface. This is done by declaring an abstract class that implments the interfaces. The interface should be tagged with the `ContractClassAttribute` and the contract definition (the abstract class) should be tagged with the `ContractClassForAttribute`

C# Example...

```
[ContractClass(typeof(MyInterfaceContract))]
public interface IMyInterface
{
    string DoWork(string input);
}
//Never inherit from this contract defintion class
[ContractClassFor(typeof(IMyInterface))]
internal abstract class MyInterfaceContract : IMyInterface
{
    private MyInterfaceContract() { }

    public string DoWork(string input)
    {
        Contract.Requires(!string.IsNullOrEmpty(input));
        Contract.Ensures(!string.IsNullOrEmpty(Contract.Result<string>()));
        throw new NotSupportedException();
    }
}
public class MyInterfaceImplmentation : IMyInterface
{
    public string DoWork(string input)
    {
        return input;
    }
}
```

Static Analysis Result...

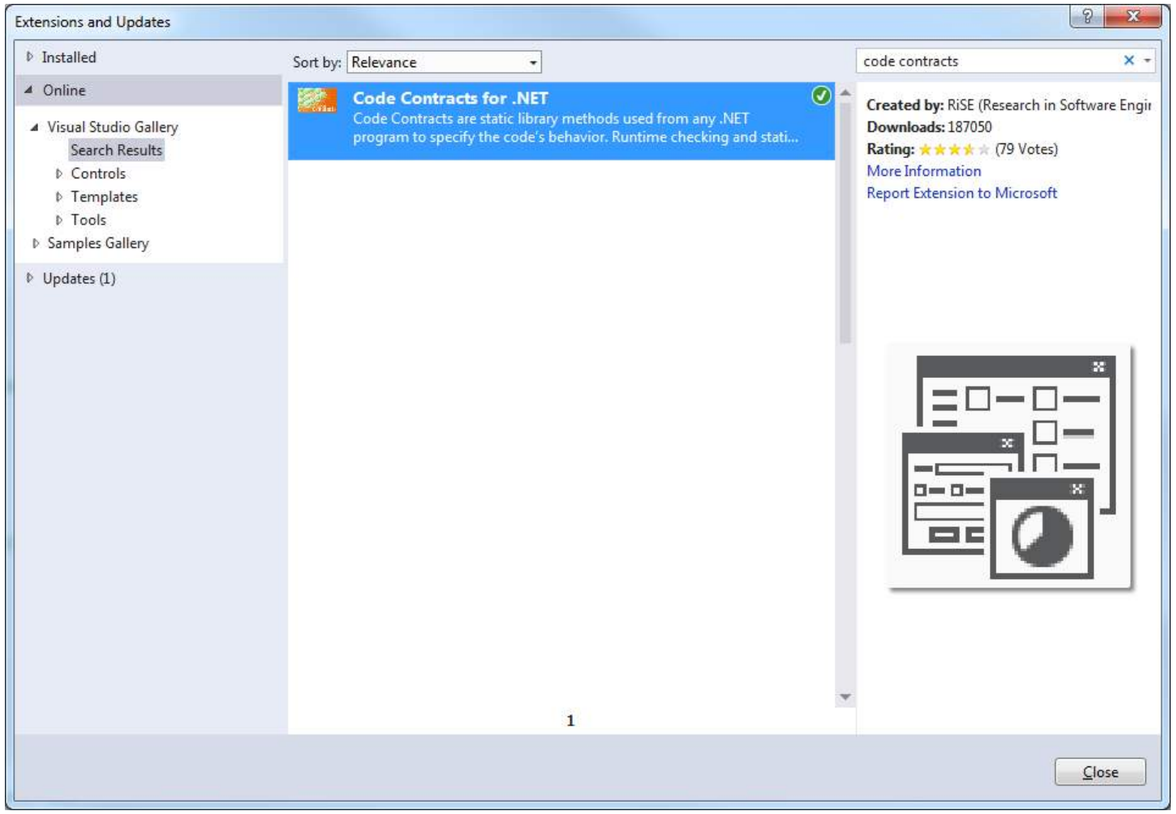
```
var m = new MyInterfaceImplmentation();
var ret = m.DoWork(null);
```

CodeContracts: requires is false: !string.IsNullOrEmpty(input)

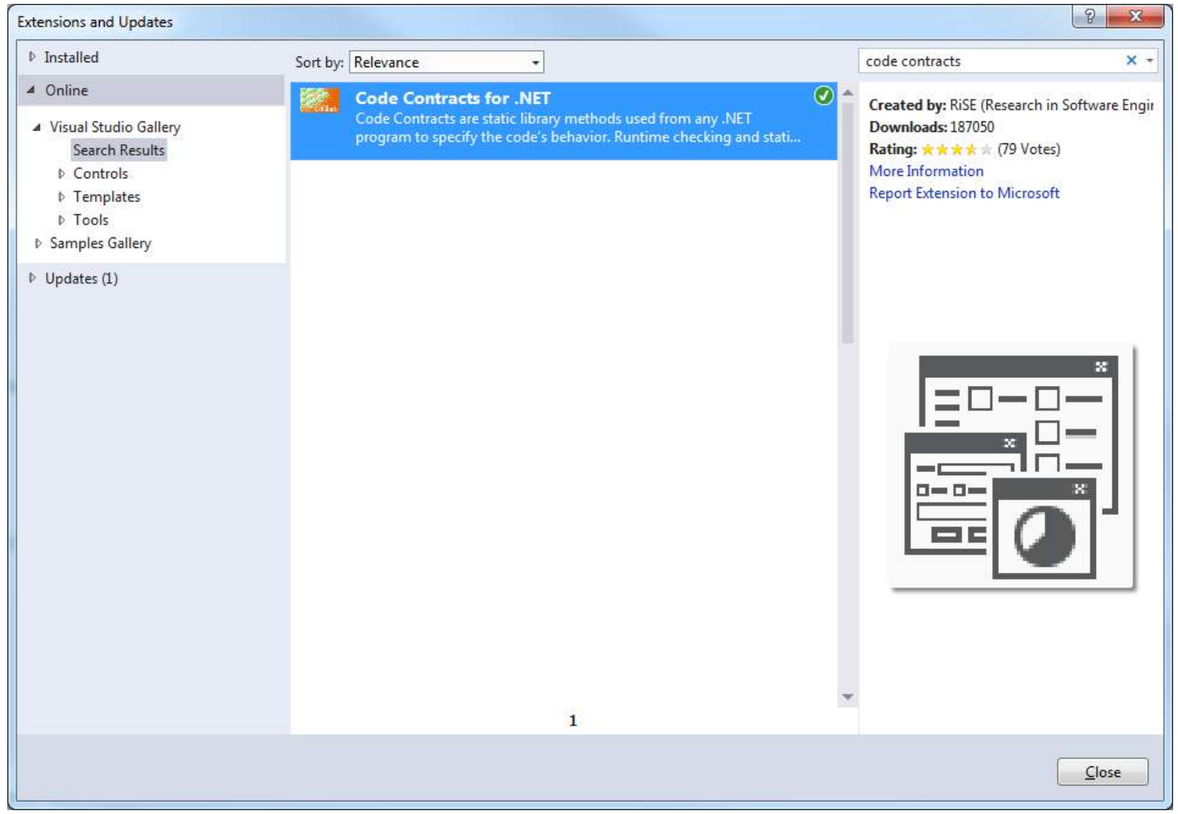
Section 13.2: Installing and Enabling Code Contracts

While `System.Diagnostics.Contracts` is included within the .Net Framework. To use Code Contracts you must install the Visual Studio extensions.

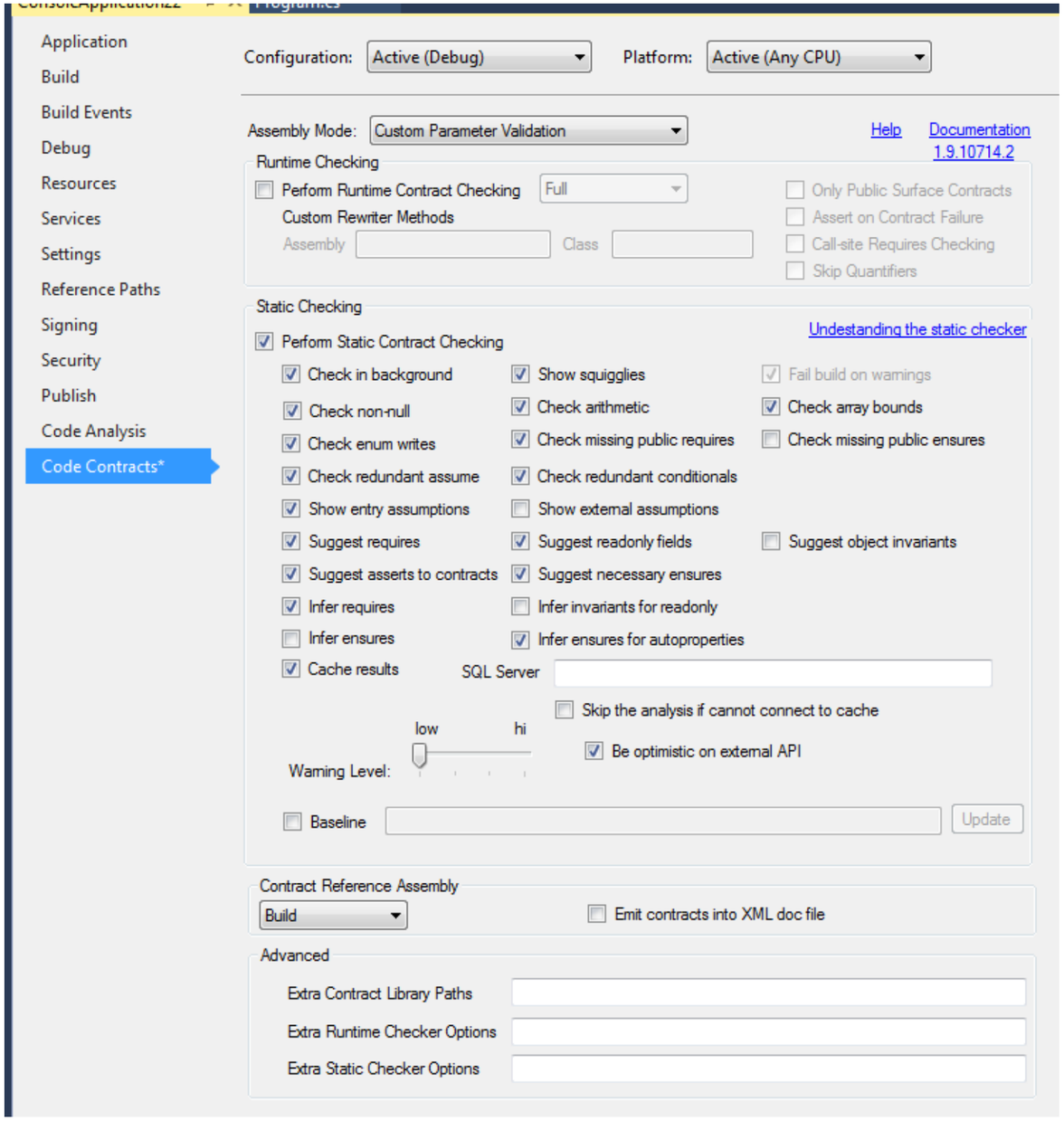
Under `Extensions and Updates` search for `Code Contracts` then install the `Code Contracts Tools`



工具安装完成后，您必须在项目解决方案中启用代码契约。至少您可能想启用静态检查（构建后检查）。如果您正在实现一个将被其他解决方案使用的库，您可能还想考虑启用运行时检查。



After the tools are installed you must enable `Code Contracts` within your Project solution. At the minimum you probably want to enable the `Static Checking` (check after build). If you are implementing a library that will be used by other solutions you may want to consider also enabling `Runtime Checking`.



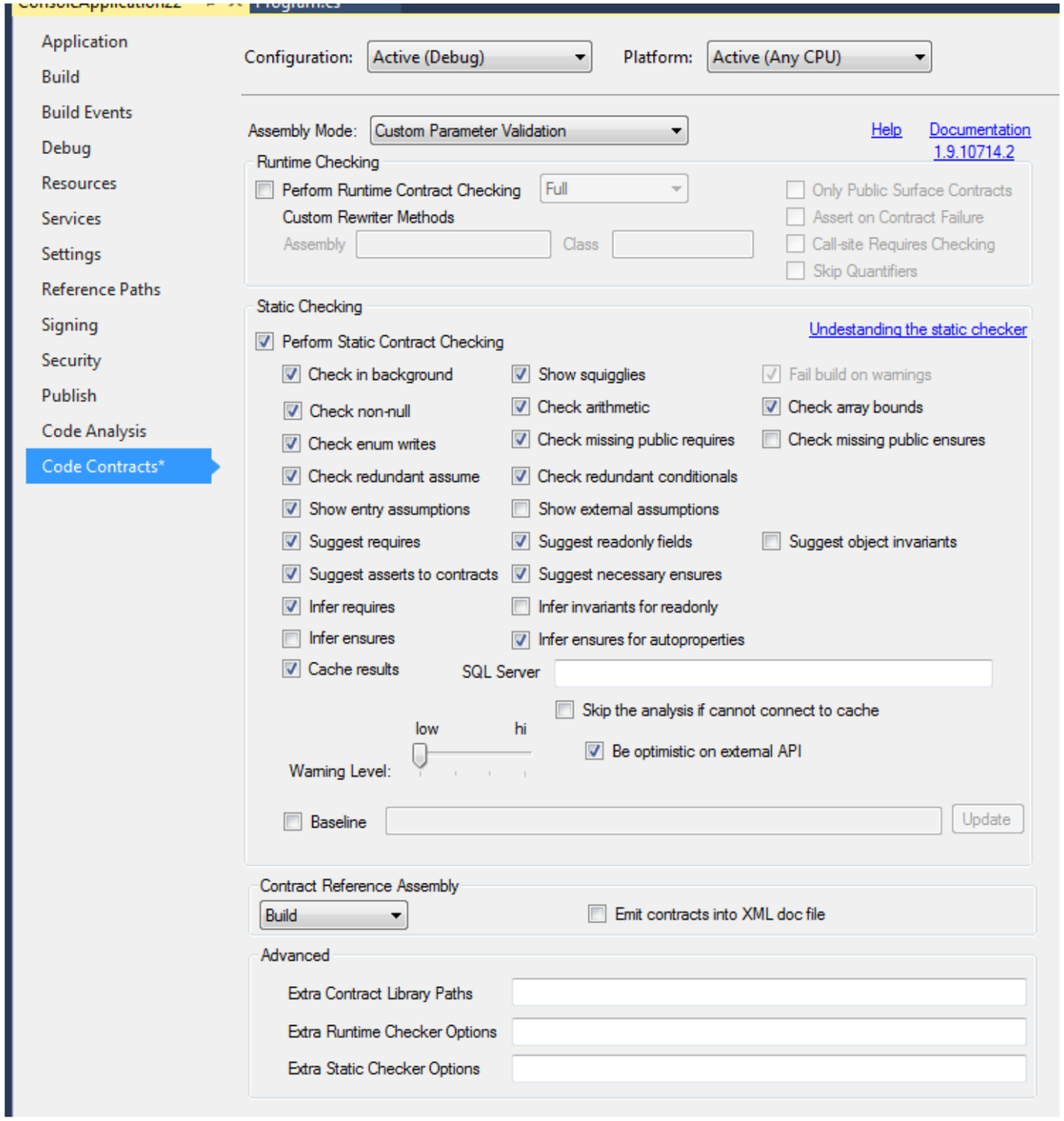
第13.3节：前置条件

前置条件允许方法为输入参数提供最低要求值

示例...

```
void DoWork(string input)
{
Contract.Requires(!string.IsNullOrEmpty(input));

    //执行工作
}
```

Section 13.3: Preconditions

Preconditions allows methods to provide minimum required values for input parameters

Example...

```
void DoWork(string input)
{
    Contract.Requires(!string.IsNullOrEmpty(input));

    //do work
}
```

静态分析结果...

```
string s = null;
p.DoWork(s);
```

CodeContracts: requires is false: !string.IsNullOrEmpty(input)

第13.4节：后置条件

后置条件确保方法返回的结果将符合所提供的定义。这为调用者提供了预期结果的定义。后置条件允许简化的实现，因为某些可能的结果可以由静态分析器提供。

示例...

```
string GetValue()
{
Contract.Ensures(Contract.Result<string>() != null);

    return null;
}
```

静态分析结果...

```
string GetValue()
{
    Contract.Ensures(Contract.Result<string>() != null);

    return null;
}
```

CodeContracts: Invoking method 'GetValue' will always lead to an error. If this is wanted, consider adding Contract.Requires(false) to document it

Static Analysis Result...

```
string s = null;
p.DoWork(s);
```

CodeContracts: requires is false: !string.IsNullOrEmpty(input)

Section 13.4: Postconditions

Postconditions ensure that the returned results from a method will match the provided definition. This provides the caller with a definition of the expected result. Postconditions may allowed for simplied implmentations as some possible outcomes can be provided by the static analyizer.

Example...

```
string GetValue()
{
    Contract.Ensures(Contract.Result<string>() != null);

    return null;
}
```

Static Analyis Result...

```
string GetValue()
{
    Contract.Ensures(Contract.Result<string>() != null);

    return null;
}
```

CodeContracts: Invoking method 'GetValue' will always lead to an error. If this is wanted, consider adding Contract.Requires(false) to document it

第14章：设置

第14.1节：来自.NET 1.x中ConfigurationSettings的AppSettings

已弃用的用法

ConfigurationSettings类是.NET 1.0和1.1中检索程序集设置的原始方式。它已被ConfigurationManager类和WebConfigurationManager类取代。

如果配置文件的appSettings部分中有两个同名的键，则使用最后一个。

app.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="keyName" value="anything, as a string"/>
    <add key="keyNames" value="123"/>
    <add key="keyNames" value="234"/>
  </appSettings>
</configuration>
```

Program.cs

```
using System;
using System.Configuration;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            string keyValue = ConfigurationSettings.AppSettings["keyName"];
            Debug.Assert("anything, as a string".Equals(keyValue));

            string twoKeys = ConfigurationSettings.AppSettings["keyNames"];
            Debug.Assert("234".Equals(twoKeys));

            Console.ReadKey();
        }
    }
}
```

第14.2节：在.NET 2.0及更高版本中从ConfigurationManager读取AppSettings

ConfigurationManager 类支持 AppSettings 属性，该属性允许你继续以 .NET 1.x 支持的方式从配置文件的 appSettings 部分读取设置。

app.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
```


Chapter 14: Settings

Section 14.1: AppSettings from ConfigurationSettings in .NET 1.x

Deprecated usage

The ConfigurationSettings class was the original way to retrieve settings for an assembly in .NET 1.0 and 1.1. It has been superseded by the ConfigurationManager class and the WebConfigurationManager class.

If you have two keys with the same name in the `appSettings` section of the configuration file, the last one is used.

app.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="keyName" value="anything, as a string"/>
    <add key="keyNames" value="123"/>
    <add key="keyNames" value="234"/>
  </appSettings>
</configuration>
```

Program.cs

```
using System;
using System.Configuration;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            string keyValue = ConfigurationSettings.AppSettings["keyName"];
            Debug.Assert("anything, as a string".Equals(keyValue));

            string twoKeys = ConfigurationSettings.AppSettings["keyNames"];
            Debug.Assert("234".Equals(twoKeys));

            Console.ReadKey();
        }
    }
}
```

Section 14.2: Reading AppSettings from ConfigurationManager in .NET 2.0 and later

The ConfigurationManager class supports the `AppSettings` property, which allows you to continue reading settings from the `appSettings` section of a configuration file the same way as .NET 1.x supported.

app.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
```

```
  <appSettings>
    <add key="keyName" value="anything, as a string"/>
    <add key="keyNames" value="123"/>
    <add key="keyNames" value="234"/>
  </appSettings>
</configuration>
```

Program.cs

```
using System;
using System.Configuration;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            string keyValue = ConfigurationManager.AppSettings["keyName"];
            Debug.Assert("anything, as a string".Equals(keyValue));

            var twoKeys = ConfigurationManager.AppSettings["keyNames"];
            Debug.Assert("234".Equals(twoKeys));

            Console.ReadKey();
        }
    }
}
```

第14.3节：Visual Studio中强类型应用程序和用户设置支持简介

Visual Studio 帮助管理用户和应用程序设置。使用这种方法相比使用
配置文件的 appSettings 部分有以下优点。

1. 设置可以是强类型的。任何可以序列化的类型都可以用作设置值。
2. 应用程序设置可以轻松地与用户设置分开。应用程序设置存储在单个配置文件中：Web 站点和 Web 应用程序使用 web.config，app.config 会被重命名为 assembly.exe.config，其中 assembly 是可执行文件的名称。用户设置（Web 项目不使用）存储在用户的应用程序数据文件夹中的 user.config 文件中（该文件夹因操作系统版本而异）。
3. 来自类库的应用程序设置可以合并到单个配置文件中而不会发生名称冲突，因为每个类库可以有自己的自定义设置部分。

在大多数项目类型中，项目属性设计器具有一个设置选项卡，这是创建自定义应用程序和用户设置的起点。最初，设置选项卡将是空白的，只有一个链接用于创建默认的设置文件。点击该链接会导致以下更改：

1. 如果项目不存在配置文件（app.config 或 web.config），将会创建一个。
2. 设置标签将被一个网格控件取代，该控件使您能够创建、编辑和删除单个设置条目。
3. 在解决方案资源管理器中，Properties特殊文件夹下添加了一个Settings.settings项。打开此项


```
  <appSettings>
    <add key="keyName" value="anything, as a string"/>
    <add key="keyNames" value="123"/>
    <add key="keyNames" value="234"/>
  </appSettings>
</configuration>
```

Program.cs

```
using System;
using System.Configuration;
using System.Diagnostics;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            string keyValue = ConfigurationManager.AppSettings["keyName"];
            Debug.Assert("anything, as a string".Equals(keyValue));

            var twoKeys = ConfigurationManager.AppSettings["keyNames"];
            Debug.Assert("234".Equals(twoKeys));

            Console.ReadKey();
        }
    }
}
```


Section 14.3: Introduction to strongly-typed application and user settings support from Visual Studio

Visual Studio helps manage user and application settings. Using this approach has these benefits over using the `appSettings` section of the configuration file.

1. Settings can be made strongly typed. Any type which can be serialized can be used for a settings value.
2. Application settings can be easily separated from user settings. Application settings are stored in a single configuration file: `web.config` for Web sites and Web applications, and app.config, renamed as *assembly*.exe.config, where *assembly* is the name of the executable. User settings (not used by Web projects) are stored in a `user.config` file in the user's Application Data folder (which varies with the operating system version).
3. Application settings from class libraries can be combined into a single configuration file without risk of name collisions, since each class library can have its own custom settings section.

In most project types, the Project Properties Designer has a Settings tab which is the starting point for creating custom application and user settings. Initially, the Settings tab will be blank, with a single link to create a default settings file. Clicking the link results in these changes:

1. If a configuration file (`app.config` or `web.config`) does not exist for the project, one will be created.
2. The Settings tab will be replaced with a grid control which enables you to create, edit, and delete individual settings entries.
3. In Solution Explorer, a `Settings.settings` item is added under the Properties special folder. Opening this

该项目将打开“设置”标签页。

4. 在项目文件夹的Properties文件夹下添加了一个带有新部分类的新文件。这个新文件是命名为Settings.Designer.__（.cs、.vb 等），类名为Settings。该类是代码生成的，因此不应编辑，但该类是一个部分类，因此您可以通过将额外的成员放在单独的文件中来扩展该类。此外，该类使用单例模式实现，通过名为Default的属性公开单例实例。

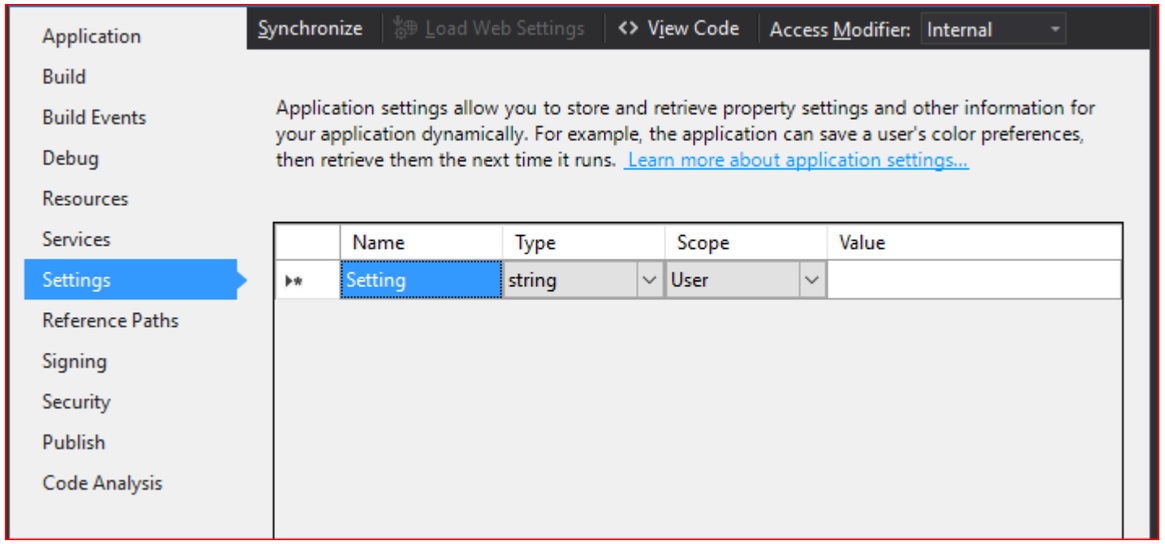
当您向“设置”选项卡添加每个新条目时，Visual Studio 会执行以下两项操作：

1. 将设置保存在配置文件中，在一个由 Settings 类管理的自定义配置节中。

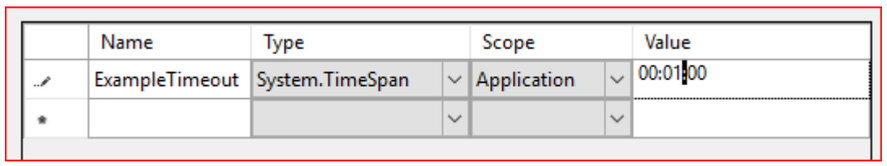
2. 在 Settings 类中创建一个新成员，用于读取、写入并以特定类型显示从 Settings 选项卡中选择的设置。

第14.4节：从配置文件的自定义节读取强类型设置

从一个新的 Settings 类和自定义配置节开始：



添加一个名为 ExampleTimeout 的应用程序设置，使用 System.Timespan 类型，并将值设置为1分钟：



保存项目属性，这将保存 Settings 选项卡中的条目，同时重新生成自定义 Settings 类并更新项目配置文件。

在代码中使用该设置（C#）：

Program.cs

item will open the Settings tab.

4. A new file with a new partial class is added under the `Properties` folder in the project folder. This new file is named `Settings.Designer.__` (.cs, .vb, etc.), and the class is named `Settings`. The class is code-generated, so it should not be edited, but the class is a partial class, so you can extend the class by putting additional members in a separate file. Furthermore, the class is implemented using the Singleton Pattern, exposing the singleton instance with the property named `Default`.

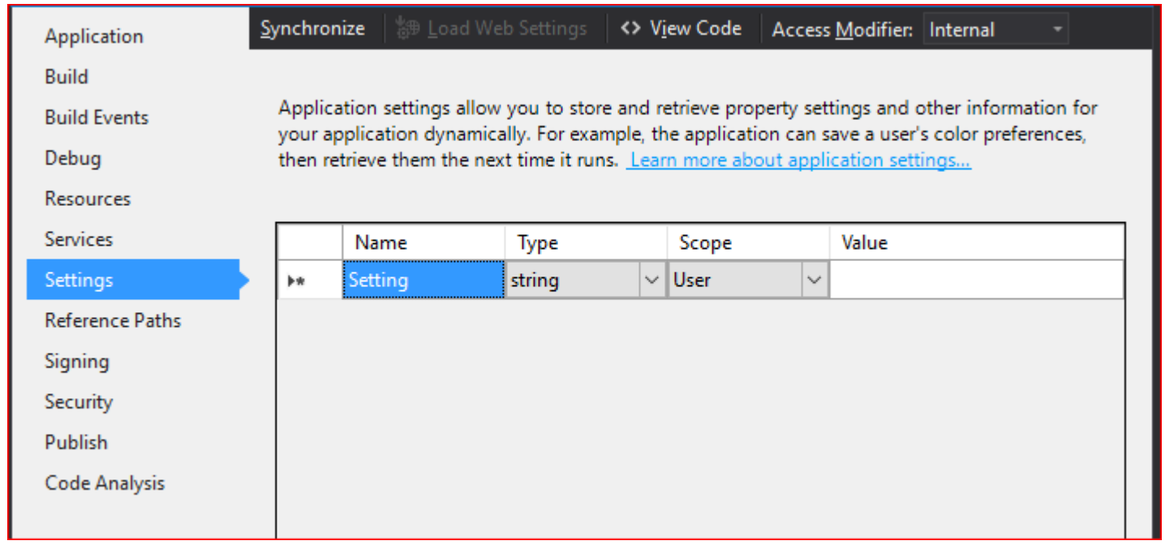
As you add each new entry to the Settings tab, Visual Studio does these two things:

1. Saves the setting in the configuration file, in a custom configuration section designed to be managed by the Settings class.

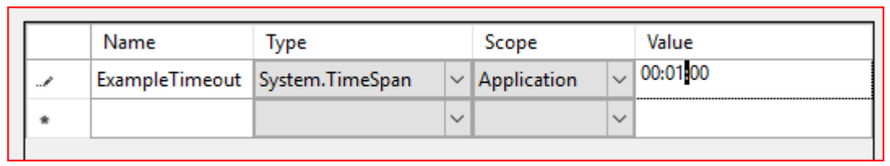
2. Creates a new member in the Settings class to read, write, and present the setting in the specific type selected from the Settings tab.

Section 14.4: Reading strongly-typed settings from custom section of configuration file

Starting from a new Settings class and custom configuration section:



Add an application setting named ExampleTimeout, using the time System.Timespan, and set the value to 1 minute:



Save the Project Properties, which saves the Settings tab entries, as well as re-generates the custom Settings class and updates the project configuration file.

Use the setting from code (C#):

Program.cs

```
using System;
using System.Diagnostics;
using ConsoleApplication1.Properties;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
TimeSpan exampleTimeout = Settings.Default.ExampleTimeout;
            Debug.Assert(TimeSpan.FromMinutes(1).Equals(exampleTimeout));

            Console.ReadKey();
        }
    }
}
```

在幕后

查看项目配置文件，了解应用程序设置条目是如何创建的：

app.config（Visual Studio 会自动更新此文件）

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <sectionGroup name="applicationSettings" type="System.Configuration.ApplicationSettingsGroup,
System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" >
      <section name="ConsoleApplication1.Properties.Settings"
type="System.Configuration.ClientSettingsSection, System, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" requirePermission="false" />
    </sectionGroup>
  </configSections>
  <appSettings />
  <applicationSettings>
    <ConsoleApplication1.Properties.Settings>
      <setting name="ExampleTimeout" serializeAs="String">
        <value>00:01:00</value>
      </setting>
    </ConsoleApplication1.Properties.Settings>
  </applicationSettings>
</configuration>
```

注意，appSettings部分未被使用。applicationSettings部分包含一个自定义命名空间限定的部分，其中每个条目都有一个setting元素。值的类型未存储在配置文件中；它仅由Settings类知道。

查看Settings类，了解它如何使用ConfigurationManager类读取此自定义部分。

Settings.designer.cs（适用于C#项目）

```
...
    [global::System.Configuration.ApplicationScopedSettingAttribute()]
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.Configuration.DefaultSettingValueAttribute("00:01:00")]
    public global::System.TimeSpan 示例超时 {
        get {
            return ((global::System.TimeSpan)(this["ExampleTimeout"]));
```



```
using System;
using System.Diagnostics;
using ConsoleApplication1.Properties;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            TimeSpan exampleTimeout = Settings.Default.ExampleTimeout;
            Debug.Assert(TimeSpan.FromMinutes(1).Equals(exampleTimeout));

            Console.ReadKey();
        }
    }
}
```

Under the covers

Look in the project configuration file to see how the application setting entry has been created:

app.config (Visual Studio updates this automatically)

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <sectionGroup name="applicationSettings" type="System.Configuration.ApplicationSettingsGroup,
System, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" >
      <section name="ConsoleApplication1.Properties.Settings"
type="System.Configuration.ClientSettingsSection, System, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" requirePermission="false" />
    </sectionGroup>
  </configSections>
  <appSettings />
  <applicationSettings>
    <ConsoleApplication1.Properties.Settings>
      <setting name="ExampleTimeout" serializeAs="String">
        <value>00:01:00</value>
      </setting>
    </ConsoleApplication1.Properties.Settings>
  </applicationSettings>
</configuration>
```

Notice that the `appSettings` section is not used. The `applicationSettings` section contains a custom namespace-qualified section that has a `setting` element for each entry. The type of the value is not stored in the configuration file; it is only known by the `Settings` class.

Look in the `Settings` class to see how it uses the `ConfigurationManager` class to read this custom section.

Settings.designer.cs (for C# projects)

```
...
    [global::System.Configuration.ApplicationScopedSettingAttribute()]
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.Configuration.DefaultSettingValueAttribute("00:01:00")]
    public global::System.TimeSpan ExampleTimeout {
        get {
            return ((global::System.TimeSpan)(this["ExampleTimeout"]));
```

```
        }
    }
...
```

请注意，创建了一个DefaultSettingValueAttribute，用于存储在项目属性设计器的设置选项卡中输入的值。如果配置文件中缺少该条目，则使用此默认值。


```
        }
    }
...
```

Notice that a `DefaultSettingValueAttribute` was created to stored the value entered in the Settings tab of the Project Properties Designer. If the entry is missing from the configuration file, this default value is used instead.

第15章：正则表达式 (System.Text.RegularExpressions)

第15.1节：检查模式是否匹配输入

```
public bool Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. W.rld!";

    // true
    return Regex.IsMatch(input, pattern);
}
```

第15.2节：从字符串中移除非字母数字字符

```
public string Remove()
{
    string 输入 = "Hello./!";

    return Regex.Replace(input, "[^a-zA-Z0-9]", "");
}
```

第15.3节：传递选项

```
public bool Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. W.rld!";

    // true
    return Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase | RegexOptions.Singleline);
}
```

第15.4节：分组匹配

```
public string Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. (?<Subject>W.rld)!";

Match match = Regex.Match(input, pattern);

    // World
    return match.Groups["Subject"].Value;
}
```

第15.5节：查找所有匹配项

使用

```
using System.Text.RegularExpressions;
```

代码

Chapter 15: Regular Expressions (System.Text.RegularExpressions)

Section 15.1: Check if pattern matches input

```
public bool Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. W.rld!";

    // true
    return Regex.IsMatch(input, pattern);
}
```

Section 15.2: Remove non alphanumeric characters from string

```
public string Remove()
{
    string input = "Hello./!";

    return Regex.Replace(input, "[^a-zA-Z0-9]", "");
}
```

Section 15.3: Passing Options

```
public bool Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. W.rld!";

    // true
    return Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase | RegexOptions.Singleline);
}
```

Section 15.4: Match into groups

```
public string Check()
{
    string input = "Hello World!";
    string pattern = @"H.ll. (?<Subject>W.rld)!";

    Match match = Regex.Match(input, pattern);

    // World
    return match.Groups["Subject"].Value;
}
```

Section 15.5: Find all matches

Using

```
using System.Text.RegularExpressions;
```

Code

```
static void Main(string[] args)
{
    string input = "胡萝卜 香蕉 苹果 樱桃 克莱门汀 葡萄";
    // 查找以大写字母 'C' 开头的单词
    string pattern = @"\bC\w*\b";

MatchCollection matches = Regex.Matches(input, pattern);
    foreach (Match m in matches)
Console.WriteLine(m.Value);
}
```

输出

```
胡萝卜
樱桃
克莱门汀
```

第15.6节：简单匹配与替换

```
public string Check()
{
    string input = "Hello World!";
    string pattern = @"W.rld";

    // Hello Stack Overflow!
    return Regex.Replace(input, pattern, "Stack Overflow");
}
```



```
static void Main(string[] args)
{
    string input = "Carrot Banana Apple Cherry Clementine Grape";
    // Find words that start with uppercase 'C'
    string pattern = @"\bC\w*\b";

    MatchCollection matches = Regex.Matches(input, pattern);
    foreach (Match m in matches)
        Console.WriteLine(m.Value);
}
```

Output

```
Carrot
Cherry
Clementine
```

Section 15.6: Simple match and replace

```
public string Check()
{
    string input = "Hello World!";
    string pattern = @"W.rld";

    // Hello Stack Overflow!
    return Regex.Replace(input, pattern, "Stack Overflow");
}
```

第16章：文件输入/输出

参数	详情
string path	要检查的文件路径。（相对路径或完整路径）

第16.1节：C# File.Exists()

```
using System;
using System.IO;

public class Program
{
    public static void Main()
    {
        string filePath = "somePath";

        if(File.Exists(filePath))
        {
Console.WriteLine("存在");
        }
        否则
        {
Console.WriteLine("不存在");
        }
    }
}
```

也可以用于三元运算符。

```
Console.WriteLine(File.Exists(pathToFile) ? "存在" : "不存在");
```

第16.2节：VB WriteAllText

```
Imports System.IO

Dim filename As String = "c:\patho\file.txt"
File.WriteAllText(filename, "要写入的文本" & vbCrLf)
```

第16.3节：VB StreamWriter

```
Dim filename As String = "c:\patho\file.txt"
If System.IO.File.Exists(filename) Then
     Dim writer As New System.IO.StreamWriter(filename)
    writer.Write("要写入的文本" & vbCrLf) '添加换行符
    writer.close()
结束如果
```

第16.4节：C# StreamWriter

```
using System.Text;
using System.IO;

string filename = "c:\patho\file.txt";
//'using'结构允许正确释放流。
using (StreamWriter writer = new StreamWriter(filename"))
```


Chapter 16: File Input/Output

Parameter	Details
string path	Path of the file to check. (relative or fully qualified)

Section 16.1: C# File.Exists()

```
using System;
using System.IO;

public class Program
{
    public static void Main()
    {
        string filePath = "somePath";

        if(File.Exists(filePath))
        {
            Console.WriteLine("Exists");
        }
        else
        {
            Console.WriteLine("Does not exist");
        }
    }
}
```

Can also be used in a ternary operator.

```
Console.WriteLine(File.Exists(pathToFile) ? "Exists" : "Does not exist");
```

Section 16.2: VB WriteAllText

```
Imports System.IO

Dim filename As String = "c:\path\to\file.txt"
File.WriteAllText(filename, "Text to write" & vbCrLf)
```

Section 16.3: VB StreamWriter

```
Dim filename As String = "c:\path\to\file.txt"
If System.IO.File.Exists(filename) Then
    Dim writer As New System.IO.StreamWriter(filename)
    writer.Write("Text to write" & vbCrLf) 'Add a newline
    writer.close()
End If
```

Section 16.4: C# StreamWriter

```
using System.Text;
using System.IO;

string filename = "c:\path\to\file.txt";
//'using' structure allows for proper disposal of stream.
using (StreamWriter writer = new StreamWriter(filename"))
```

```
{
writer.WriteLine("Text to Write");}
```

第16.5节：C# WriteAllText()

```
using System.IO;
using System.Text;

string filename = "c:\patho\file.txt";File.writeAllTe
xt(filename, "Text to write");
```



```
{
    writer.WriteLine("Text to Write\n");
}
```

Section 16.5: C# WriteAllText()

```
using System.IO;
using System.Text;

string filename = "c:\path\to\file.txt";
File.writeAllText(filename, "Text to write\n");
```

第17章：System.IO

第17.1节：使用StreamReader读取文本文件

```
string fullOrRelativePath = "testfile.txt";

string fileData;

using (var reader = new StreamReader(fullOrRelativePath))
{
fileData = reader.ReadToEnd();
}
```

请注意，此StreamReader构造函数重载会进行一些自动encoding检测，但可能与文件实际使用的编码不符。

请注意，System.IO.File类中有一些方便的方法可用于读取文件中的所有文本，即File.ReadAllText(path)和File.ReadAllLines(path)。

第17.2节：使用System.IO.SerialPorts的串口

遍历已连接的串口

```
using System.IO.Ports;
string[] ports = SerialPort.GetPortNames();
for (int i = 0; i < ports.Length; i++)
{
Console.WriteLine(ports[i]);
}
```

实例化一个 System.IO.SerialPort 对象

```
using System.IO.Ports;
SerialPort port = new SerialPort();
SerialPort port = new SerialPort("COM 1"); ;
SerialPort port = new SerialPort("COM 1", 9600);
```

注意：这些只是 SerialPort 类型构造函数的七个重载中的三个。

通过 SerialPort 读写数据

最简单的方法是使用 `SerialPort.Read` 和 `SerialPort.Write` 方法。不过你也可以获取一个 `System.IO.Stream` 对象，用于通过 SerialPort 流式传输数据。要做到这一点，使用 SerialPort.BaseStream。

读取

```
int length = port.BytesToRead;
//注意，如果你愿意，可以用 char 数组替换 byte 数组。
byte[] buffer = new byte[length];
port.Read(buffer, 0, length);
```

你也可以读取所有可用的数据：

```
string curData = port.ReadExisting();
```


Chapter 17: System.IO

Section 17.1: Reading a text file using StreamReader

```
string fullOrRelativePath = "testfile.txt";

string fileData;

using (var reader = new StreamReader(fullOrRelativePath))
{
    fileData = reader.ReadToEnd();
}
```

Note that this `StreamReader` constructor overload does some auto encoding detection, which may or may not conform to the actual encoding used in the file.

Please note that there are some convenience methods that read all text from file available on the `System.IO.File` class, namely `File.ReadAllText(path)` and `File.ReadAllLines(path)`.

Section 17.2: Serial Ports using System.IO.SerialPorts

Iterating over connected serial ports

```
using System.IO.Ports;
string[] ports = SerialPort.GetPortNames();
for (int i = 0; i < ports.Length; i++)
{
    Console.WriteLine(ports[i]);
}
```

Instantiating a System.IO.SerialPort object

```
using System.IO.Ports;
SerialPort port = new SerialPort();
SerialPort port = new SerialPort("COM 1"); ;
SerialPort port = new SerialPort("COM 1", 9600);
```

NOTE: Those are just three of the seven overloads of the constructor for the SerialPort type.

Reading/Writing data over the SerialPort

The simplest way is to use the `SerialPort.Read` and `SerialPort.Write` methods. However you can also retrieve a `System.IO.Stream` object which you can use to stream data over the SerialPort. To do this, use `SerialPort.BaseStream`.

Reading

```
int length = port.BytesToRead;
//Note that you can swap out a byte-array for a char-array if you prefer.
byte[] buffer = new byte[length];
port.Read(buffer, 0, length);
```

You can also read all data available:

```
string curData = port.ReadExisting();
```

或者简单地读取直到接收到的第一个换行符为止的数据：

```
string line = port.ReadLine();
```

写入

通过 SerialPort 写入数据最简单的方法是：

```
port.Write("here is some text to be sent over the serial port.");
```

不过在需要时，你也可以这样发送数据：

```
//注意你可以根据需要将字节数组替换为字符数组。
byte[] data = new byte[1] { 255 };
port.Write(data, 0, data.Length);
```

第17.3节：使用System.IO.File读取/写入数据

首先，让我们来看三种从文件中提取数据的不同方法。

```
string fileText = File.ReadAllText(file);
string[] fileLines = File.ReadAllLines(file);
byte[] fileBytes = File.ReadAllBytes(file);
```

- 在第一行，我们将文件中的所有数据读取为一个字符串。
- 在第二行，我们将文件中的数据读取到一个字符串数组中。文件中的每一行都成为数组中的一个元素。

- 在第三行，我们读取文件中的字节。

接下来，让我们来看三种向文件追加数据的不同方法。如果指定的文件不存在，每种方法都会在尝试追加数据之前自动创建该文件。

```
File.AppendAllText(file, "Here is some data that isappended to the file."); File.AppendAllLines(fil
e, new string[2] { "Here is some data that is", "appended to the file." });using (StreamWriter stream = File.AppendText(
 file))
 {
stream.WriteLine("Here is some data that is");
     stream.Write("appended to the file.");
 }
```

- 在第一行，我们简单地将一个字符串添加到指定文件的末尾。
- 在第二行，我们将数组的每个元素添加到文件中的新行上。
- 最后在第三行，我们使用File.AppendText来打开一个StreamWriter，它会将写入的数据追加到文件中。

最后，让我们看看三种不同的写入数据到文件的方法。追加和写入的区别在于，写入会覆盖文件中的数据，而追加则是向文件中的数据添加内容。如果你指定的文件不存在，每种方法都会在尝试写入数据之前自动创建该文件。

```
File.WriteAllText(file, "here is some datain this file.");File.WriteAllLines(file,
 new string[2] { "here is some data", "in this file" });File.WriteAllBytes(file, new byte[2] { 0, 255
});
```

- 第一行将一个字符串写入文件。
- 第二行将数组中的每个字符串分别写入文件的不同行。

Or simply read to the first newline encountered in the incoming data:

```
string line = port.ReadLine();
```

Writing

The easiest way to write data over the SerialPort is:

```
port.Write("here is some text to be sent over the serial port.");
```

However you can also send data over like this when needed:

```
//Note that you can swap out the byte-array with a char-array if you so choose.
byte[] data = new byte[1] { 255 };
port.Write(data, 0, data.Length);
```

Section 17.3: Reading/Writing Data Using System.IO.File

First, let's see three different ways of extracting data from a file.

```
string fileText = File.ReadAllText(file);
string[] fileLines = File.ReadAllLines(file);
byte[] fileBytes = File.ReadAllBytes(file);
```

- On the first line, we read all the data in the file as a string.
- On the second line, we read the data in the file into a string-array. Each line in the file becomes an element in the array.
- On the third we read the bytes from the file.

Next, let's see three different methods of **appending** data to a file. If the file you specify doesn't exist, each method will automatically create the file before attempting to append the data to it.

```
File.AppendAllText(file, "Here is some data that is\nappended to the file.");
File.AppendAllLines(file, new string[2] { "Here is some data that is", "appended to the file." });
using (StreamWriter stream = File.AppendText(file))
{
    stream.WriteLine("Here is some data that is");
    stream.Write("appended to the file.");
}
```

- On the first line we simply add a string to the end of the specified file.
- On the second line we add each element of the array onto a new line in the file.
- Finally on the third line we use `File.AppendText` to open up a streamwriter which will append whatever data is written to it.

And lastly, let's see three different methods of **writing** data to a file. The difference between *appending* and *writing* being that writing **over-writes** the data in the file while appending **adds** to the data in the file. If the file you specify doesn't exist, each method will automatically create the file before attempting to write the data to it.

```
File.WriteAllText(file, "here is some data\nin this file.");
File.WriteAllLines(file, new string[2] { "here is some data", "in this file" });
File.WriteAllBytes(file, new byte[2] { 0, 255 });
```

- The first line writes a string to the file.
- The second line writes each string in the array on it's own line in the file.

- 第三行允许你将一个字节数组写入文件。

- And the third line allows you to write a byte array to the file.

第18章：System.IO.File 类

参数	详情
source	要移动到另一个位置的文件。
destination	您想要移动的目录 source 到（该变量还应包含名称文件的名称（和文件扩展名）。

第18.1节：删除文件

删除文件（如果您有相应权限）非常简单：

```
File.Delete(path);
```

但是可能会出现许多问题：

- 您没有所需权限（会抛出UnauthorizedAccessException异常）。
- 文件可能正被其他人使用（会抛出IOException异常）。
- 由于底层错误或介质为只读，文件无法删除（会抛出IOException异常）。
- 文件已不存在（会抛出IOException异常）。

注意，最后一点（文件不存在）通常通过如下代码片段来规避：

```
if (File.Exists(path))
    File.Delete(path);
```

但是这不是一个原子操作，文件可能在调用File.Exists()和File.Delete()之间被其他人删除。正确处理I/O操作的方法需要异常处理（假设操作失败时可以采取替代措施）：

```
if (File.Exists(path))
{
    try
    {
File.Delete(path);
    }
    catch (IOException exception)
    {
        if (!File.Exists(path))
            return; // 其他人已删除此文件

        // 出现了其他错误...
    }
    catch (UnauthorizedAccessException exception)
    {
        // 我没有所需的权限
    }
}
```

请注意，这些I/O错误有时是暂时性的（例如文件正在使用中），如果涉及网络连接，可能会自动恢复，无需我们采取任何操作。通常会retry几次I/O操作，每次尝试之间有短暂延迟：

```
public static void Delete(string path)
{
    if (!File.Exists(path))
```


Chapter 18: System.IO.File class

Parameter	Details
source	The file that is to be moved to another location.
destination	The directory in which you would like to move source to (this variable should also contain the name (and file extension) of the file.

Section 18.1: Delete a file

To delete a file (if you have required permissions) is as simple as:

```
File.Delete(path);
```

However many things may go wrong:

- You do not have required permissions (`UnauthorizedAccessException` is thrown).
- File may be in use by someone else (`IOException` is thrown).
- File cannot be deleted because of low level error or media is read-only (`IOException` is thrown).
- File does not exist anymore (`IOException` is thrown).

Note that last point (file does not exist) is usually *circumvented* with a code snippet like this:

```
if (File.Exists(path))
    File.Delete(path);
```

However it's not an atomic operation and file may be delete by someone else between the call to `File.Exists()` and before `File.Delete()`. Right approach to handle I/O operation requires exception handling (assuming an alternative course of actions may be taken when operation fails):

```
if (File.Exists(path))
{
    try
    {
        File.Delete(path);
    }
    catch (IOException exception)
    {
        if (!File.Exists(path))
            return; // Someone else deleted this file

        // Something went wrong...
    }
    catch (UnauthorizedAccessException exception)
    {
        // I do not have required permissions
    }
}
```

Note that this I/O errors sometimes are transitory (file in use, for example) and if a network connection is involved then it may automatically recover without any action from our side. It's then common to *retry* an I/O operation few times with a small delay between each attempt:

```
public static void Delete(string path)
{
    if (!File.Exists(path))
```

```
        return;

    for (int i=1; ; ++i)
    {
        try
        {
File.Delete(path);
            return;
        }
        catch (IOException e)
        {
            if (!File.Exists(path))
                return;

            if (i == NumberOfAttempts)
                throw;

Thread.Sleep(DelayBetweenEachAttempt);
        }

        // 你可以处理 UnauthorizedAccessException，但这个问题
        // 可能在几秒内无法解决...
    }
}

private const int NumberOfAttempts = 3;
private const int DelayBetweenEachAttempt = 1000; // 毫秒
```

注意：在Windows环境中，当调用此函数时，文件不会被真正删除，如果其他人使用FileShare.Delete打开该文件，则文件可以被删除，但实际上只有当所有者关闭该文件时，删除操作才会生效。

第18.2节：从文本文件中剔除不需要的行

修改文本文件并不容易，因为其内容必须被移动。对于小型文件，最简单的方法是将其内容读入内存，然后写回修改后的文本。

在此示例中，我们读取文件中的所有行，删除所有空白行，然后写回到原路径：

```
File.WriteAllLines(path,
File.ReadAllLines(path).Where(x => !String.IsNullOrWhiteSpace(x)));
```

如果文件太大无法加载到内存中，且输出路径与输入路径不同：

```
File.WriteAllLines(outputPath,
File.ReadLines(inputPath).Where(x => !String.IsNullOrWhiteSpace(x)));
```

第18.3节：转换文本文件编码

文本以编码形式保存（另见字符串主题），有时您可能需要更改其编码，本示例
假设（为简便起见）文件不太大，可以全部读入内存：

```
public static void ConvertEncoding(string path, Encoding from, Encoding to)
{
File.WriteAllText(path, File.ReadAllText(path, from), to);
}
```



```
        return;

    for (int i=1; ; ++i)
    {
        try
        {
            File.Delete(path);
            return;
        }
        catch (IOException e)
        {
            if (!File.Exists(path))
                return;

            if (i == NumberOfAttempts)
                throw;

            Thread.Sleep(DelayBetweenEachAttempt);
        }

        // You may handle UnauthorizedAccessException but this issue
        // will probably won't be fixed in few seconds...
    }
}

private const int NumberOfAttempts = 3;
private const int DelayBetweenEachAttempt = 1000; // ms
```

Note: in Windows environment file will not be really deleted when you call this function, if someone else open the file using `FileShare.Delete` then file can be deleted but it will effectively happen only when owner will close the file.

Section 18.2: Strip unwanted lines from a text file

To change a text file is not easy because its content must be moved around. For *small* files easiest method is to read its content in memory and then write back modified text.

In this example we read all lines from a file and drop all blank lines then we write back to original path:

```
File.WriteAllLines(path,
    File.ReadAllLines(path).Where(x => !String.IsNullOrWhiteSpace(x)));
```

If file is too big to load it in memory and output path is different from input path:

```
File.WriteAllLines(outputPath,
    File.ReadLines(inputPath).Where(x => !String.IsNullOrWhiteSpace(x)));
```

Section 18.3: Convert text file encoding

Text is saved encoded (see also Strings topic) then sometimes you may need to change its encoding, this example assumes (for simplicity) that file is not too big and it can be entirely read in memory:

```
public static void ConvertEncoding(string path, Encoding from, Encoding to)
{
    File.WriteAllText(path, File.ReadAllText(path, from), to);
}
```

执行转换时不要忘记文件可能包含BOM（字节顺序标记），要更好地理解
它是如何被管理的，请参考Encoding.UTF8.GetString不考虑Preamble/BOM。

第18.4节：枚举比指定时间更早的文件

此代码片段是一个辅助函数，用于枚举所有比指定时间更早的文件，例如当你需要删除旧日志文件或旧缓存数据时非常有用。

```
static IEnumerable<string> EnumerateAllFilesOlderThan(
                           TimeSpan maximumAge,
                                string path,
                                string searchPattern = "*.*",
SearchOption options = SearchOption.TopDirectoryOnly)
{
DateTime oldestWriteTime = DateTime.Now - maximumAge;

    return Directory.EnumerateFiles(path, searchPattern, options)
        .Where(x => Directory.GetLastWriteTime(x) < oldestWriteTime);
}
```

用法如下：

```
var oldFiles = EnumerateAllFilesOlderThan(TimeSpan.FromDays(7), @"c:\log", "*.log");
```

需要注意的几点：

- 搜索是使用Directory.EnumerateFiles()而不是Directory.GetFiles()进行的。枚举是*实时的*，因此你不需要等待所有文件系统条目被获取完毕。
- 我们检查的是最后写入时间，但你也可以使用创建时间或最后访问时间（例如删除*未使用的*缓存文件，注意访问时间可能被禁用）。
- 这些属性（写入时间、访问时间、创建时间）的粒度并不统一，具体细节请查阅MSDN。

第18.5节：将文件从一个位置移动到另一个位置

File.Move
为了将文件从一个位置移动到另一个位置，一行简单的代码即可实现：

```
File.Move(@"C:\TemporaryFile.txt", @"C:\TemporaryFiles\TemporaryFile.txt");
```

然而，这个简单操作可能会出现许多问题。例如，如果运行你程序的用户没有标记为“C”的驱动器怎么办？如果他们有——但他们决定将其重命名为“B”或“M”怎么办？

如果源文件（您想要移动的文件）在您不知情的情况下已被移动——或者根本不存在，该怎么办？

可以先检查源文件是否存在来避免这个问题：

```
string source = @"C:\TemporaryFile.txt", destination = @"C:\TemporaryFiles\TemporaryFile.txt";
if(File.Exists("C:\TemporaryFile.txt"))
{
File.Move(source, destination);
}
```

这将确保在那一刻，文件确实存在，并且可以被移动到另一个位置。可能会有

When performing conversions do not forget that file may contain BOM (Byte Order Mark), to better understand how it's managed refer to Encoding.UTF8.GetString doesn't take into account the Preamble/BOM.

Section 18.4: Enumerate files older than a specified amount

This snippet is an helper function to enumerate all files older than a specified age, it's useful - for example - when you have to delete old log files or old cached data.

```
static IEnumerable<string> EnumerateAllFilesOlderThan(
                                TimeSpan maximumAge,
                                string path,
                                string searchPattern = "*.*",
                                SearchOption options = SearchOption.TopDirectoryOnly)
{
    DateTime oldestWriteTime = DateTime.Now - maximumAge;

    return Directory.EnumerateFiles(path, searchPattern, options)
        .Where(x => Directory.GetLastWriteTime(x) < oldestWriteTime);
}
```

Used like this:

```
var oldFiles = EnumerateAllFilesOlderThan(TimeSpan.FromDays(7), @"c:\log", "*.log");
```

Few things to note:

- Search is performed using `Directory.EnumerateFiles()` instead of `Directory.GetFiles()`. Enumeration is *alive* then you won't need to wait until all file system entries have been fetched.
- We're checking for last write time but you may use creation time or last access time (for example to delete *unused* cached files, note that access time may be disabled).
- Granularity isn't uniform for all those properties (write time, access time, creation time), check MSDN for details about this.

Section 18.5: Move a File from one location to another

File.Move
In order to move a file from one location to another, one simple line of code can achieve this:

```
File.Move(@"C:\TemporaryFile.txt", @"C:\TemporaryFiles\TemporaryFile.txt");
```

However, there are many things that could go wrong with this simple operation. For instance, what if the user running your program does not have a Drive that is labelled 'C'? What if they did - but they decided to rename it to 'B', or 'M'?

What if the Source file (the file in which you would like to move) has been moved without your knowing - or what if it simply doesn't exist.

This can be circumvented by first checking to see whether the source file does exist:

```
string source = @"C:\TemporaryFile.txt", destination = @"C:\TemporaryFiles\TemporaryFile.txt";
if(File.Exists("C:\TemporaryFile.txt"))
{
    File.Move(source, destination);
}
```

This will ensure that at that very moment, the file does exist, and can be moved to another location. There may be

某些情况下，简单调用File.Exists可能不够。如果不够，请再次检查，告知用户操作失败——或者处理异常。

您可能遇到的不仅仅是FileNotFoundException异常。

请参见下方可能出现的异常：

异常类型	描述
输入输出异常	文件已存在或找不到源文件。
ArgumentNullException	源和/或目标参数的值为 null。
参数异常	源和/或目标参数的值为空，或包含无效字符。
未经授权的访问异常	您没有执行此操作所需的权限。
路径过长异常	源、目标或指定路径超过最大长度。在 Windows 上，路径长度必须小于 248 个字符，文件名必须小于 260 个字符。
目录未找到异常	找不到指定的目录。
NotSupportedException	源路径或目标路径或文件名格式无效。

times where a simple call to `File.Exists` won't be enough. If it isn't, check again, convey to the user that the operation failed - or handle the exception.

A `FileNotFoundException` is not the only exception you are likely to encounter.

See below for possible exceptions:

Exception Type	Description
IOException	The file already exists or the source file could not be found.
ArgumentNullException	The value of the Source and/or Destination parameters is null.
ArgumentException	The value of the Source and/or Destination parameters are empty, or contain invalid characters.
UnauthorizedAccessException	You do not have the required permissions in order to perform this action.
PathTooLongException	The Source, Destination or specified path(s) exceed the maximum length. On Windows, a Path's length must be less than 248 characters, while File names must be less than 260 characters.
DirectoryNotFoundException	The specified directory could not be found.
NotSupportedException	The Source or Destination paths or file names are in an invalid format.

第19章：读取和写入Zip文件

ZipFile 类位于 **System.IO.Compression** 命名空间中。它可用于读取和写入Zip文件。

第19.1节：列出ZIP内容

此代码片段将列出zip存档中的所有文件名。文件名相对于zip根目录。

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    for (int i = 0; i < archive.Entries.Count; i++)
    {
Console.WriteLine($"{i}: {archive.Entries[i]}");
    }
}
```

第19.2节：从ZIP文件中提取文件

将所有文件提取到目录中非常简单：

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
archive.ExtractToDirectory(AppDomain.CurrentDomain.BaseDirectory);
}
```

当文件已存在时，将抛出System.IO.IOException异常。

提取特定文件：

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    // 获取根目录下的文件
archive.GetEntry("test.txt").ExtractToFile("test_extracted_getentries.txt", true);

    // 如果想从子目录提取文件，请输入路径
archive.GetEntry("sub/subtest.txt").ExtractToFile("test_sub.txt", true);

    // 你也可以使用 Entries 属性来查找文件
archive.Entries.FirstOrDefault(f => f.Name ==
"test.txt")?.ExtractToFile("test_extracted_linq.txt", true);

    // 由于找不到文件，这将抛出 System.ArgumentNullException 异常
archive.GetEntry("nonexistingfile.txt").ExtractToFile("fail.txt", true);
}
```

这些方法中的任何一种都会产生相同的结果。

第19.3节：更新ZIP文件

要更新ZIP文件，必须使用ZipArchiveMode.Update模式打开该文件。

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
```


Chapter 19: Reading and writing Zip files

The **ZipFile** class lives in the **System.IO.Compression** namespace. It can be used to read from, and write to Zip files.

Section 19.1: Listing ZIP contents

This snippet will list all the filenames of a zip archive. The filenames are relative to the zip root.

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    for (int i = 0; i < archive.Entries.Count; i++)
    {
        Console.WriteLine($"{i}: {archive.Entries[i]}");
    }
}
```

Section 19.2: Extracting files from ZIP files

Extracting all the files into a directory is very easy:

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    archive.ExtractToDirectory(AppDomain.CurrentDomain.BaseDirectory);
}
```

When the file already exists, a **System.IO.IOException** will be thrown.

Extracting specific files:

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Read))
{
    // Get a root entry file
    archive.GetEntry("test.txt").ExtractToFile("test_extracted_getentries.txt", true);

    // Enter a path if you want to extract files from a subdirectory
    archive.GetEntry("sub/subtest.txt").ExtractToFile("test_sub.txt", true);

    // You can also use the Entries property to find files
    archive.Entries.FirstOrDefault(f => f.Name ==
"test.txt")?.ExtractToFile("test_extracted_linq.txt", true);

    // This will throw a System.ArgumentNullException because the file cannot be found
    archive.GetEntry("nonexistingfile.txt").ExtractToFile("fail.txt", true);
}
```

Any of these methods will produce the same result.

Section 19.3: Updating a ZIP file

To update a ZIP file, the file has to be opened with ZipArchiveMode.Update instead.

```
using (FileStream fs = new FileStream("archive.zip", FileMode.Open))
```

```
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Update))
{
    // 添加文件到根目录
archive.CreateEntryFromFile("test.txt", "test.txt");

    // 添加文件到子文件夹
archive.CreateEntryFromFile("test.txt", "symbols/test.txt");
}
```

还有一个选项是直接写入归档中的文件：

```
var entry = archive.CreateEntry("createentry.txt");
using(var writer = new StreamWriter(entry.Open()))
{
writer.WriteLine("Test line");
}
```



```
using (ZipArchive archive = new ZipArchive(fs, ZipArchiveMode.Update))
{
    // Add file to root
    archive.CreateEntryFromFile("test.txt", "test.txt");

    // Add file to subfolder
    archive.CreateEntryFromFile("test.txt", "symbols/test.txt");
}
```

There is also the option to write directly to a file within the archive:

```
var entry = archive.CreateEntry("createentry.txt");
using(var writer = new StreamWriter(entry.Open()))
{
    writer.WriteLine("Test line");
}
```

第20章：托管可扩展性框架

第20.1节：连接（基础）

请参见上面的其他（基础）示例。

```
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;

namespace Demo
{
    public static class Program
    {
        public static void Main()
        {
            using (var catalog = new ApplicationCatalog())
            using (var exportProvider = new CatalogExportProvider(catalog))
            using (var container = new CompositionContainer(exportProvider))
            {
exportProvider.SourceProvider = container;

                UserWriter writer = new UserWriter();

                // 此时，writer 的 userProvider 字段为 null
container.ComposeParts(writer);

                // 现在，它应该是非空的（否则会抛出异常）。
writer.PrintAllUsers();
            }
        }
    }
}
```

只要应用程序的程序集搜索路径中的某个位置有 `[Export(typeof(IUserProvider))]`，`UserWriter` 对应的导入就会被满足，用户信息将被打印出来。

可以使用其他类型的目录（例如 `DirectoryCatalog`）来替代（或作为补充）`ApplicationCatalog`，以在其他位置查找满足导入的导出。

第20.2节：导出类型（基础）

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel.Composition;

namespace Demo
{
    [Export(typeof(IUserProvider))]
    public sealed class UserProvider : IUserProvider
    {
        public ReadOnlyCollection<User> GetAllUsers()
        {
            return new List<User>
            {
                new User(0, "admin"),
                new User(1, "Dennis"),
```


Chapter 20: Managed Extensibility Framework

Section 20.1: Connecting (Basic)

See the other (Basic) examples above.

```
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;

namespace Demo
{
    public static class Program
    {
        public static void Main()
        {
            using (var catalog = new ApplicationCatalog())
            using (var exportProvider = new CatalogExportProvider(catalog))
            using (var container = new CompositionContainer(exportProvider))
            {
                exportProvider.SourceProvider = container;

                UserWriter writer = new UserWriter();

                // at this point, writer's userProvider field is null
                container.ComposeParts(writer);

                // now, it should be non-null (or an exception will be thrown).
                writer.PrintAllUsers();
            }
        }
    }
}
```

As long as something in the application's assembly search path has `[Export(typeof(IUserProvider))]`, `UserWriter`'s corresponding import will be satisfied and the users will be printed.

Other types of catalogs (e.g., `DirectoryCatalog`) can be used instead of (or in addition to) `ApplicationCatalog`, to look in other places for exports that satisfy the imports.

Section 20.2: Exporting a Type (Basic)

```
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel.Composition;

namespace Demo
{
    [Export(typeof(IUserProvider))]
    public sealed class UserProvider : IUserProvider
    {
        public ReadOnlyCollection<User> GetAllUsers()
        {
            return new List<User>
            {
                new User(0, "admin"),
                new User(1, "Dennis"),
```

```
                new User(2, "Samantha"),
            }.AsReadOnly();
        }
    }
}
```

这可以在几乎任何地方定义；关键是应用程序知道在哪里查找它（通过它创建的ComposablePartCatalogs）。

第20.3节：导入（基础）

```
using System;
using System.ComponentModel.Composition;

namespace Demo
{
    public sealed class UserWriter
    {
        [Import(typeof(IUserProvider))]
        private IUserProvider userProvider;

        public void PrintAllUsers()
        {
            foreach (User user in this.userProvider.GetAllUsers())
            {
Console.WriteLine(user);
            }
        }
    }
}
```

这是一个依赖于IUserProvider的类型，IUserProvider可以定义在任何地方。和之前的示例一样，关键是应用程序知道在哪里查找匹配的导出（通过它创建的 ComposablePartCatalogs）。


```
                new User(2, "Samantha"),
            }.AsReadOnly();
        }
    }
}
```

This could be defined virtually anywhere; all that matters is that the application knows where to look for it (via the ComposablePartCatalogs it creates).

Section 20.3: Importing (Basic)

```
using System;
using System.ComponentModel.Composition;

namespace Demo
{
    public sealed class UserWriter
    {
        [Import(typeof(IUserProvider))]
        private IUserProvider userProvider;

        public void PrintAllUsers()
        {
            foreach (User user in this.userProvider.GetAllUsers())
            {
                Console.WriteLine(user);
            }
        }
    }
}
```

This is a type that has a dependency on an `IUserProvider`, which could be defined anywhere. Like the previous example, all that matters is that the application knows where to look for the matching export (via the ComposablePartCatalogs it creates).

第21章：SpeechRecognitionEngine类用于识别语音

LoadGrammar：参数	详细信息
grammar	要加载的语法。例如，一个`DictationGrammar`对象，用于允许自由文本听写。
RecognizeAsync：参数	**详情**
mode	当前识别的RecognizeMode：Single表示仅识别一次，Multiple表示允许多次识别。
GrammarBuilder.Append：参数	**详细信息**
choices	向语法构建器追加一些选项。这意味着，当用户输入语音时，识别器可以从语法中遵循不同的“分支”。
Choices构造函数：参数	**详情**
选项	用于语法构建器的选项数组。参见GrammarBuilder.Append。
Grammar 构造函数：参数	**详情**
构建器	用于从中构建Grammar的GrammarBuilder。

第21.1节：基于有限短语集的异步语音识别

```
SpeechRecognitionEngine recognitionEngine = new SpeechRecognitionEngine();
GrammarBuilder builder = new GrammarBuilder();
builder.Append(new Choices("我", "你", "他", "她", "我们", "他们"));
builder.Append(new Choices("友好", "不友好"));
recognitionEngine.LoadGrammar(new Grammar(builder));
recognitionEngine.SpeechRecognized += delegate(object sender, SpeechRecognizedEventArgs e)
{
Console.WriteLine("你说的是：{0}", e.Result.Text);
};
recognitionEngine.SetInputToDefaultAudioDevice();
recognitionEngine.RecognizeAsync(RecognizeMode.Multiple);
```

第21.2节：异步识别自由文本听写的语音

```
使用 System.Speech.Recognition;

// ...

SpeechRecognitionEngine recognitionEngine = new SpeechRecognitionEngine();
recognitionEngine.LoadGrammar(new DictationGrammar());
recognitionEngine.SpeechRecognized += delegate(object sender, SpeechRecognizedEventArgs e)
{
Console.WriteLine("你说的是：{0}", e.Result.Text);
};
recognitionEngine.SetInputToDefaultAudioDevice();
recognitionEngine.RecognizeAsync(RecognizeMode.Multiple);
```


Chapter 21: SpeechRecognitionEngine class to recognize speech

`LoadGrammar`: Parameters	Details
grammar	The grammar to load. For example, a `DictationGrammar` object to allow free text dictation.
`RecognizeAsync`: Parameters	**Details**
mode	The `RecognizeMode` for the current recognition: `Single` for just one recognition, `Multiple` to allow multiple.
`GrammarBuilder.Append`: Parameters	**Details**
choices	Appends some choices to the grammar builder. This means that, when the user inputs speech, the recognizer can follow different "branches" from a grammar.
`Choices` constructor: Parameters	**Details**
choices	An array of choices for the grammar builder. See `GrammarBuilder.Append`.
`Grammar` constructor: Parameter	**Details**
builder	The `GrammarBuilder` to construct a `Grammar` from.

Section 21.1: Asynchronously recognizing speech based on a restricted set of phrases

```
SpeechRecognitionEngine recognitionEngine = new SpeechRecognitionEngine();
GrammarBuilder builder = new GrammarBuilder();
builder.Append(new Choices("I am", "You are", "He is", "She is", "We are", "They are"));
builder.Append(new Choices("friendly", "unfriendly"));
recognitionEngine.LoadGrammar(new Grammar(builder));
recognitionEngine.SpeechRecognized += delegate(object sender, SpeechRecognizedEventArgs e)
{
    Console.WriteLine("You said: {0}", e.Result.Text);
};
recognitionEngine.SetInputToDefaultAudioDevice();
recognitionEngine.RecognizeAsync(RecognizeMode.Multiple);
```

Section 21.2: Asynchronously recognizing speech for free text dictation

```
using System.Speech.Recognition;

// ...

SpeechRecognitionEngine recognitionEngine = new SpeechRecognitionEngine();
recognitionEngine.LoadGrammar(new DictationGrammar());
recognitionEngine.SpeechRecognized += delegate(object sender, SpeechRecognizedEventArgs e)
{
    Console.WriteLine("You said: {0}", e.Result.Text);
};
recognitionEngine.SetInputToDefaultAudioDevice();
recognitionEngine.RecognizeAsync(RecognizeMode.Multiple);
```

第22章：System.Runtime.Caching.MemoryCache (ObjectCache)

第22.1节：向缓存添加项（Set）

Set函数通过使用CacheItem实例提供缓存项的键和值，将缓存项插入缓存中。

此函数重写了 ObjectCache.Set(CacheItem, CacheItemPolicy)

```
private static bool SetToCache()
{
    string key = "Cache_Key";
    string value = "Cache_Value";

    //获取默认 MemoryCache 实例的引用。
    var cacheContainer = MemoryCache.Default;

    var policy = new CacheItemPolicy()
    {
AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(DEFAULT_CACHE_EXPIRATION_MINUTES)
     };
     var itemToCache = new CacheItem(key, value); //值的类型是 object。
cacheContainer.Set(itemToCache, policy);
}
```

第 22.2 节：System.Runtime.Caching.MemoryCache (ObjectCache)

此函数从缓存中获取现有项，如果缓存中不存在该项，则会根据
valueFetchFactory 函数获取该项。

```
    public static TValue GetExistingOrAdd<TValue>(string key, double minutesForExpiration,
Func<TValue> valueFetchFactory)
    {
        尝试
        {
            // Lazy 类提供延迟初始化，只有当缓存中不存在该项时才会计算 valueFetchFactory
             的值。
            var newValue = new Lazy<TValue>(valueFetchFactory);

            // 设置缓存策略，如果该项将被保存回缓存。
CacheItemPolicy policy = new CacheItemPolicy()
            {
AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(minutesForExpiration)
            };

                            // 返回缓存中已有的项，或者如果不存在则添加新值。
            var cachedItem = _cacheContainer.AddOrGetExisting(key, newValue, policy) as
Lazy<TValue>;

            return (cachedItem ?? newValue).Value;
        }
        catch (Exception excep)
        {
```


Chapter 22: System.Runtime.Caching.MemoryCache (ObjectCache)

Section 22.1: Adding Item to Cache (Set)

Set function inserts a cache entry into the cache by using a CacheItem instance to supply the key and value for the cache entry.

This function Overrides `ObjectCache.Set(CacheItem, CacheItemPolicy)`

```
private static bool SetToCache()
{
    string key = "Cache_Key";
    string value = "Cache_Value";

    //Get a reference to the default MemoryCache instance.
    var cacheContainer = MemoryCache.Default;

    var policy = new CacheItemPolicy()
    {
        AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(DEFAULT_CACHE_EXPIRATION_MINUTES)
     };
     var itemToCache = new CacheItem(key, value); //Value is of type object.
     cacheContainer.Set(itemToCache, policy);
}
```

Section 22.2: System.Runtime.Caching.MemoryCache (ObjectCache)

This function gets existing item form cache, and if the item don't exist in cache, it will fetch item based on the valueFetchFactory function.

```
    public static TValue GetExistingOrAdd<TValue>(string key, double minutesForExpiration,
Func<TValue> valueFetchFactory)
    {
        try
        {
            //The Lazy class provides Lazy initialization which will evaluate
            //the valueFetchFactory only if item is not in the cache.
            var newValue = new Lazy<TValue>(valueFetchFactory);

            //Setup the cache policy if item will be saved back to cache.
            CacheItemPolicy policy = new CacheItemPolicy()
            {
                AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(minutesForExpiration)
            };

            //returns existing item form cache or add the new value if it does not exist.
            var cachedItem = _cacheContainer.AddOrGetExisting(key, newValue, policy) as
Lazy<TValue>;

            return (cachedItem ?? newValue).Value;
        }
        catch (Exception excep)
        {
```

```
            return default(TValue);
        }
    }
```



```
            return default(TValue);
        }
    }
```

第23章：System.Reflection.Emit命名空间

第23.1节：动态创建程序集

```
using System;
using System.Reflection;
using System.Reflection.Emit;

class DemoAssemblyBuilder
{
    public static void Main()
    {
        // 一个程序集由一个或多个模块组成，每个模块
        // 包含零个或多个类型。此代码创建了一个单模块
        // 程序集，这是最常见的情况。该模块包含一个类型，
        // 名为“MyDynamicType”，它有一个私有字段，
        // 一个获取和设置该私有字段的属性，构造函数
        // 用于初始化私有字段，以及一个方法，
        // 将用户提供的数字乘以私有字段的值并返回
        // 结果。在C#中，该类型可能如下所示：
        /*
public class MyDynamicType
       {
private int m_number;

           public MyDynamicType() : this(42) {}
           public MyDynamicType(int initNumber)
           {
m_number = initNumber;
           }

public int Number
           {
get { return m_number; }
               set { m_number = value; }
           }

public int MyMethod(int multiplier)
           {
return m_number * multiplier;
           }
}
       */

AssemblyName aName = new AssemblyName("DynamicAssemblyExample");
        AssemblyBuilder ab =
AppDomain.CurrentDomain.DefineDynamicAssembly(
            aName,
AssemblyBuilderAccess.RunAndSave);

        // 对于单模块程序集，模块名称通常是
        // 程序集名称加上扩展名。
ModuleBuilder mb =
ab.DefineDynamicModule(aName.Name, aName.Name + ".dll");

        TypeBuilder tb = mb.DefineType(
            "MyDynamicType",
TypeAttributes.Public);
```


Chapter 23: System.Reflection.Emit namespace

Section 23.1: Creating an assembly dynamically

```
using System;
using System.Reflection;
using System.Reflection.Emit;

class DemoAssemblyBuilder
{
    public static void Main()
    {
        // An assembly consists of one or more modules, each of which
        // contains zero or more types. This code creates a single-module
        // assembly, the most common case. The module contains one type,
        // named "MyDynamicType", that has a private field, a property
        // that gets and sets the private field, constructors that
        // initialize the private field, and a method that multiplies
        // a user-supplied number by the private field value and returns
        // the result. In C# the type might look like this:
        /*
        public class MyDynamicType
        {
            private int m_number;

            public MyDynamicType() : this(42) {}
            public MyDynamicType(int initNumber)
            {
                m_number = initNumber;
            }

            public int Number
            {
                get { return m_number; }
                set { m_number = value; }
            }

            public int MyMethod(int multiplier)
            {
                return m_number * multiplier;
            }
        }
        */

        AssemblyName aName = new AssemblyName("DynamicAssemblyExample");
        AssemblyBuilder ab =
            AppDomain.CurrentDomain.DefineDynamicAssembly(
                aName,
                AssemblyBuilderAccess.RunAndSave);

        // For a single-module assembly, the module name is usually
        // the assembly name plus an extension.
        ModuleBuilder mb =
            ab.DefineDynamicModule(aName.Name, aName.Name + ".dll");

        TypeBuilder tb = mb.DefineType(
            "MyDynamicType",
             TypeAttributes.Public);
```

```
        // 添加一个类型为 int (Int32) 的私有字段。
FieldBuilder fbNumber = tb.DefineField(
            "m_number",
            typeof(int),
FieldAttributes.Private);

        // 接下来，我们创建一个简单的密封方法。
MethodBuilder mbMyMethod = tb.DefineMethod(
            "MyMethod",
MethodAttributes.Public,
            typeof(int),
            new[] { typeof(int) });

ILGenerator il = mbMyMethod.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // 加载 this - 始终是任何实例方法的第一个参数
       il.Emit(OpCodes.Ldfld, fbNumber);
il.Emit(OpCodes.Ldarg_1); // 加载整数参数
        il.Emit(OpCodes.Mul); // 乘以两个数字，不进行溢出检查
        il.Emit(OpCodes.Ret); // 返回

        // 接下来，我们构建属性。这包括构建属性本身，以及getter和setter方法。

PropertyBuilder pbNumber = tb.DefineProperty(
            "Number", // 名称
PropertyAttributes.None,
            typeof(int), // 属性的类型
            new Type[0]); // 索引的类型（如果有）

MethodBuilder mbSetNumber = tb.DefineMethod(
            "set_Number", // 名称 - 按惯例，setter为set_Property
            // setter是特殊方法，我们不希望它在C#调用者中出现
              MethodAttributes.PrivateScope | MethodAttributes.HideBySig | MethodAttributes.Public |
MethodAttributes.SpecialName,
            typeof(void), // setter不返回值
            new[] { typeof(int) }); // 有一个类型为System.Int32的参数

        // 生成方法体，我们需要一个IL生成器
il = mbSetNumber.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // 加载this
        il.Emit(OpCodes.Ldarg_1); // 加载新值
        il.Emit(OpCodes.Stfld, fbNumber); // 将新值保存到this.m_number
        il.Emit(OpCodes.Ret); // 返回

        // 最后，将方法链接到我们属性的设置器
pbNumber.SetSetMethod(mbSetNumber);

        MethodBuilder mbGetNumber = tb.DefineMethod(
            "get_Number",
MethodAttributes.PrivateScope | MethodAttributes.HideBySig | MethodAttributes.Public |
MethodAttributes.SpecialName,
            typeof(int),
            new Type[0]);

il = mbGetNumber.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // 加载 this
        il.Emit(OpCodes.Ldfld, fbNumber); // 加载 this.m_number 的值
        il.Emit(OpCodes.Ret); // 返回值

pbNumber.SetGetMethod(mbGetNumber);

        // 最后，我们添加两个构造函数。
        // 构造函数需要调用父类的构造函数，或者同一类中的另一个构造函数
```



```
        // Add a private field of type int (Int32).
        FieldBuilder fbNumber = tb.DefineField(
            "m_number",
            typeof(int),
            FieldAttributes.Private);

        // Next, we make a simple sealed method.
        MethodBuilder mbMyMethod = tb.DefineMethod(
            "MyMethod",
            MethodAttributes.Public,
            typeof(int),
            new[] { typeof(int) });

        ILGenerator il = mbMyMethod.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this - always the first argument of any instance method
        il.Emit(OpCodes.Ldfld, fbNumber);
        il.Emit(OpCodes.Ldarg_1); // Load the integer argument
        il.Emit(OpCodes.Mul); // Multiply the two numbers with no overflow checking
        il.Emit(OpCodes.Ret); // Return

        // Next, we build the property. This involves building the property itself, as well as the
        // getter and setter methods.
        PropertyBuilder pbNumber = tb.DefineProperty(
            "Number", // Name
            PropertyAttributes.None,
            typeof(int), // Type of the property
            new Type[0]); // Types of indices, if any

        MethodBuilder mbSetNumber = tb.DefineMethod(
            "set_Number", // Name - setters are set_Property by convention
            // Setter is a special method and we don't want it to appear to callers from C#
            MethodAttributes.PrivateScope | MethodAttributes.HideBySig | MethodAttributes.Public |
MethodAttributes.SpecialName,
            typeof(void), // Setters don't return a value
            new[] { typeof(int) }); // We have a single argument of type System.Int32

        // To generate the body of the method, we'll need an IL generator
        il = mbSetNumber.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this
        il.Emit(OpCodes.Ldarg_1); // Load the new value
        il.Emit(OpCodes.Stfld, fbNumber); // Save the new value to this.m_number
        il.Emit(OpCodes.Ret); // Return

        // Finally, link the method to the setter of our property
        pbNumber.SetSetMethod(mbSetNumber);

        MethodBuilder mbGetNumber = tb.DefineMethod(
            "get_Number",
            MethodAttributes.PrivateScope | MethodAttributes.HideBySig | MethodAttributes.Public |
MethodAttributes.SpecialName,
            typeof(int),
            new Type[0]);

        il = mbGetNumber.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // Load this
        il.Emit(OpCodes.Ldfld, fbNumber); // Load the value of this.m_number
        il.Emit(OpCodes.Ret); // Return the value

        pbNumber.SetGetMethod(mbGetNumber);

        // Finally, we add the two constructors.
        // Constructor needs to call the constructor of the parent class, or another constructor in
```

```
同一类中的另一个构造函数
ConstructorBuilder intConstructor = tb.DefineConstructor(
            MethodAttributes.Public, CallingConventions.Standard | CallingConventions.HasThis,
new[] { typeof(int) });
il = intConstructor.GetILGenerator();
il.Emit(OpCodes.Ldarg_0); // this（当前实例）
il.Emit(OpCodes.Call, typeof(object).GetConstructor(new Type[0])); // 调用父类构造函数

il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Ldarg_1); // 我们的 int 参数
        il.Emit(OpCodes.Stfld, fbNumber); // 将参数存储到 this.m_number
       il.Emit(OpCodes.Ret);

        var parameterlessConstructor = tb.DefineConstructor(
            MethodAttributes.Public, CallingConventions.Standard | CallingConventions.HasThis, new
Type[0]);
il = parameterlessConstructor.GetILGenerator();
       il.Emit(OpCodes.Ldarg_0); // this（当前实例）
il.Emit(OpCodes.Ldc_I4_S, (byte)42); // 加载整数常量 42
        il.Emit(OpCodes.Call, intConstructor); // 调用 this(42)
il.Emit(OpCodes.Ret);

        // 并确保类型已创建
Type ourType = tb.CreateType();

        // 可以直接使用反射来使用程序集中的类型，或者我们可以保存程序集以供参考

        object ourInstance = Activator.CreateInstance(ourType);
        Console.WriteLine(ourType.GetProperty("Number").GetValue(ourInstance)); // 42

        // 保存程序集以便在其他地方使用。这对于调试非常有用——例如你可以使用
ILSpy 来查看等效的 IL/C# 代码。
ab.Save(@"DynamicAssemblyExample.dll");
        // 使用新创建的类型
        var myDynamicType = tb.CreateType();
        var myDynamicTypeInstance = Activator.CreateInstance(myDynamicType);

Console.WriteLine(myDynamicTypeInstance.GetType()); // MyDynamicType

        var numberField = myDynamicType.GetField("m_number", BindingFlags.NonPublic |
BindingFlags.Instance);
numberField.SetValue (myDynamicTypeInstance, 10);

        Console.WriteLine(numberField.GetValue(myDynamicTypeInstance)); // 10
    }
}
```



```
the same class
        ConstructorBuilder intConstructor = tb.DefineConstructor(
            MethodAttributes.Public, CallingConventions.Standard | CallingConventions.HasThis,
new[] { typeof(int) });
        il = intConstructor.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Call, typeof(object).GetConstructor(new Type[0])); // call parent's
constructor
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Ldarg_1); // our int argument
        il.Emit(OpCodes.Stfld, fbNumber); // store argument in this.m_number
        il.Emit(OpCodes.Ret);

        var parameterlessConstructor = tb.DefineConstructor(
            MethodAttributes.Public, CallingConventions.Standard | CallingConventions.HasThis, new
Type[0]);
        il = parameterlessConstructor.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0); // this
        il.Emit(OpCodes.Ldc_I4_S, (byte)42); // load 42 as an integer constant
        il.Emit(OpCodes.Call, intConstructor); // call this(42)
        il.Emit(OpCodes.Ret);

        // And make sure the type is created
        Type ourType = tb.CreateType();

        // The types from the assembly can be used directly using reflection, or we can save the
assembly to use as a reference
        object ourInstance = Activator.CreateInstance(ourType);
        Console.WriteLine(ourType.GetProperty("Number").GetValue(ourInstance)); // 42

        // Save the assembly for use elsewhere. This is very useful for debugging - you can use e.g.
ILSpy to look at the equivalent IL/C# code.
        ab.Save(@"DynamicAssemblyExample.dll");
        // Using newly created type
        var myDynamicType = tb.CreateType();
        var myDynamicTypeInstance = Activator.CreateInstance(myDynamicType);

        Console.WriteLine(myDynamicTypeInstance.GetType()); // MyDynamicType

        var numberField = myDynamicType.GetField("m_number", BindingFlags.NonPublic |
BindingFlags.Instance);
        numberField.SetValue (myDynamicTypeInstance, 10);

        Console.WriteLine(numberField.GetValue(myDynamicTypeInstance)); // 10
    }
}
```

第24章：.NET Core

.NET Core 是由微软和 .NET 社区在 GitHub 上维护的通用开发平台。它是跨平台的，支持 Windows、macOS 和 Linux，可用于设备、云以及嵌入式/物联网场景。

当你想到 .NET Core 时，应联想到以下内容（灵活部署、跨平台、命令行工具、开源）。

另一个优点是，即使它是开源的，微软仍在积极支持它。

第24.1节：基础控制台应用程序

```
public class Program
{
    public static void Main(string[] args)
    {
Console.WriteLine("你叫什么名字？ ");var name = Console.Read
        Line();
        var date = DateTime.Now;
Console.WriteLine("你好，{0}，今天是 {1:d}，时间是 {1:t}", name, date);        Console.Wr
ite("按任意键退出...");
Console.ReadKey(true);
    }
}
```


Chapter 24: .NET Core

.NET Core is a general purpose development platform maintained by Microsoft and the .NET community on GitHub. It is cross-platform, supporting Windows, macOS and Linux, and can be used in device, cloud, and embedded/IoT scenarios.

When you think of .NET Core the following should come to mind (flexible deployment, cross-platform, command-line tools, open source).

Another great thing is that even if it's open source Microsoft is actively supporting it.

Section 24.1: Basic Console App

```
public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("\nWhat is your name? ");
        var name = Console.ReadLine();
        var date = DateTime.Now;
        Console.WriteLine("\nHello, {0}, on {1:d} at {1:t}", name, date);
        Console.Write("\nPress any key to exit...");
        Console.ReadKey(true);
    }
}
```

第25章：ADO.NET

ADO（ActiveX数据对象）.Net 是微软提供的一个工具，通过其组件可以访问诸如SQLServer、Oracle和XML等数据源。前端.Net应用程序一旦通过ADO.Net以适当权限连接到数据源，就可以检索、创建和操作数据。

ADO.Net 提供了无连接架构。这是一种与数据库交互的安全方法，因为连接不必在整个会话期间保持。

第25.1节：最佳实践 - 执行Sql语句

```
public void SaveNewEmployee(Employee newEmployee)
{
    // 最佳实践 - 将所有数据库连接包装在using块中，以确保即使发生异常也能始终关闭和释放连接

    // 最佳实践 - 从app.config或web.config中按名称检索连接字符串（取决于应用程序类型）（注意，这需要引用System.configuration程序集）

    using(SqlConnection con = new SqlConnection(System.Configuration.ConfigurationManager.ConnectionStrings["MyConnectionName"].ConnectionString))
    {
        // 最佳实践 - 在INSERT语句中使用列名，这样就不依赖于sql模式中的列顺序

        // 最佳实践 - 始终使用参数以避免 SQL 注入攻击和错误，尤其是在使用格式错误的文本时，例如包含单引号，这在 SQL 中相当于转义或开始一个字符串（varchar/nvarchar）

        // 最佳实践 - 给你的参数起有意义的名字，就像你给代码中的变量命名一样

        using(SqlCommand sc = new SqlCommand("INSERT INTO employee (FirstName, LastName, DateOfBirth /*etc*/) VALUES (@firstName, @lastName, @dateOfBirth /*etc*/)", con))
        {
            // 最佳实践 - 始终指定你使用的列的数据库数据类型
            // 最佳实践 - 在代码中检查有效值和/或使用数据库约束，如果插入NULL则使用System.DbNull.Value
            sc.Parameters.Add(new SqlParameter("@firstName", SqlDbType.VarChar, 200){Value = newEmployee.FirstName ?? (object) System.DBNull.Value});
            sc.Parameters.Add(new SqlParameter("@lastName", SqlDbType.VarChar, 200){Value = newEmployee.LastName ?? (object) System.DBNull.Value});

            // 最佳实践 - 指定参数时始终使用正确的类型，Value赋值为DateTime实例，而不是日期的字符串表示

            sc.Parameters.Add(new SqlParameter("@dateOfBirth", SqlDbType.Date){ Value = newEmployee.DateOfBirth });

            // 最佳实践 - 尽可能晚打开连接，除非你需要验证数据库连接有效且不会失败，并且后续代码执行时间较长（此处不适用）

            con.Open();
            sc.ExecuteNonQuery();
        }

        // using块结束时会关闭并释放SqlConnection
        // 最佳实践 - 尽早结束using块以释放数据库连接
    }
}

// 用作参数的辅助类示例
public class Employee
```


Chapter 25: ADO.NET

ADO(ActiveX Data Objects).Net is a tool provided by Microsoft which provides access to data sources such as SQL Server, Oracle, and XML through its components. .Net front-end applications can retrieve, create, and manipulate data, once they are connected to a data source through ADO.Net with appropriate privileges.

ADO.Net provides a connection-less architecture. It is a secure approach to interact with a database, since, the connection doesn't have to be maintained during the entire session.

Section 25.1: Best Practices - Executing Sql Statements

```
public void SaveNewEmployee(Employee newEmployee)
{
    // best practice - wrap all database connections in a using block so they are always closed & disposed even in the event of an Exception
    // best practice - retrieve the connection string by name from the app.config or web.config (depending on the application type) (note, this requires an assembly reference to System.configuration)
    using(SqlConnection con = new SqlConnection(System.Configuration.ConfigurationManager.ConnectionStrings["MyConnectionName"].ConnectionString))
    {
        // best practice - use column names in your INSERT statement so you are not dependent on the sql schema column order
        // best practice - always use parameters to avoid sql injection attacks and errors if malformed text is used like including a single quote which is the sql equivalent of escaping or starting a string (varchar/nvarchar)
        // best practice - give your parameters meaningful names just like you do variables in your code
        using(SqlCommand sc = new SqlCommand("INSERT INTO employee (FirstName, LastName, DateOfBirth /*etc*/) VALUES (@firstName, @lastName, @dateOfBirth /*etc*/)", con))
        {
            // best practice - always specify the database data type of the column you are using
            // best practice - check for valid values in your code and/or use a database constraint, if inserting NULL then use System.DbNull.Value
            sc.Parameters.Add(new SqlParameter("@firstName", SqlDbType.VarChar, 200){Value = newEmployee.FirstName ?? (object) System.DBNull.Value});
            sc.Parameters.Add(new SqlParameter("@lastName", SqlDbType.VarChar, 200){Value = newEmployee.LastName ?? (object) System.DBNull.Value});

            // best practice - always use the correct types when specifying your parameters, Value is assigned to a DateTime instance and not a string representation of a Date
            sc.Parameters.Add(new SqlParameter("@dateOfBirth", SqlDbType.Date){ Value = newEmployee.DateOfBirth });

            // best practice - open your connection as late as possible unless you need to verify that the database connection is valid and won't fail and the proceeding code execution takes a long time (not the case here)
            con.Open();
            sc.ExecuteNonQuery();
        }

        // the end of the using block will close and dispose the SqlConnection
        // best practice - end the using block as soon as possible to release the database connection
    }
}

// supporting class used as parameter for example
public class Employee
```

```
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
}
```

使用ADO.NET的最佳实践

- 经验法则是尽量缩短连接打开时间。过程执行完毕后应显式关闭连接，这样会将连接对象返回到连接池。默认连接池最大大小 = 100。由于连接池提升了与SQL的物理连接性能

  服务器。SQL Server中的连接池
- 将所有数据库连接包装在using块中，这样即使发生
  异常，连接也会被始终关闭和释放。更多关于using语句的信息请参见using Statement (C# Reference)
- 从app.config或web.config（取决于应用类型）中按名称检索连接字符串
  - 这需要引用程序集System.configuration有关如何构建配置文件的
  - 更多信息，请参见Connection Strings and Configuration Files
- 始终对传入值使用参数，以
  - 避免SQL注入攻击
  - 如果使用格式错误的文本（例如包含单引号，单引号在SQL中相当于转义或字符串（varchar/nvarchar）的起始）时，避免出现错误
  - 允许数据库提供程序重用查询计划（并非所有数据库提供程序都支持），这可以提高效率
- 使用参数时
  - SQL 参数类型和大小不匹配是插入/更新/查询失败的常见原因。给你的 SQL 参数起有意义的名
  - 字，就像你给代码中的变量命名一样
  - 指定你使用的列的数据库数据类型，这可以确保不会使用错误的参数类型，从而避免出现意外结果
  - 在将参数传入命令之前验证传入的参数（俗话说，“垃圾进，垃圾出”）。尽早在调用栈中验证传入的值
  - 为参数赋值时使用正确的类型，例如：不要将 DateTime 的字符串值赋给参数，而是将实际的 DateTime 实例赋给参数的值
  - 指定字符串类型参数的大小。因为如果参数的类型和大小匹配，SQL Server 可以重用执行计划。使用 -1 表示 MAX
  - 不要使用AddWithValue方法，主要原因是很容易忘记在需要时指定参数类型或精度/小数位。更多信息请参见Can we stop using AddWithValue already?
- 使用数据库连接时
  - 尽可能晚打开连接，尽可能早关闭连接。这是使用任何外部资源时的一般指导原则
  - 切勿共享数据库连接实例（例如：让单例托管一个共享的SqlConnection实例）。你的代码应始终在需要时创建新的数据库连接实例，随后由调用代码在使用完毕后释放并“丢弃”它。原因是大多数数据库提供程序都有某种连接池机制，因此创建新的托管连接开销很小。
    -
    - 如果代码开始使用多线程，这样做可以避免未来出现任何错误。

第25.2节：作为命令执行SQL语句

```
// 使用Windows身份验证。将Trusted_Connection参数替换为
```



```
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
}
```


Best practice for working with ADO.NET

- Rule of thumb is to open connection for minimal time. Close the connection explicitly once your procedure execution is over this will return the connection object back to connection pool. Default connection pool max size = 100. As connection pooling enhances the performance of physical connection to SQL Server.Connection Pooling in SQL Server
- Wrap all database connections in a using block so they are always closed & disposed even in the event of an Exception. See using Statement (C# Reference) for more information on using statements
- Retrieve the connection strings by name from the app.config or web.config (depending on the application type)
  - This requires an assembly reference to `System.configuration`
  - See Connection Strings and Configuration Files for additional information on how to structure your configuration file
- Always use parameters for incoming values to
  - Avoid sql injection attacks
  - Avoid errors if malformed text is used like including a single quote which is the sql equivalent of escaping or starting a string (varchar/nvarchar)
  - Letting the database provider reuse query plans (not supported by all database providers) which increases efficiency
- When working with parameters
  - Sql parameters type and size mismatch is a common cause of insert/ updated/ select failure
  - Give your Sql parameters meaningful names just like you do variables in your code
  - Specify the database data type of the column you are using, this ensures the wrong parameter types is not used which could lead to unexpected results
  - Validate your incoming parameters before you pass them into the command (as the saying goes, "garbage in, garbage out"). Validate incoming values as early as possible in the stack
  - Use the correct types when assigning your parameter values, example: do not assign the string value of a DateTime, instead assign an actual DateTime instance to the value of the parameter
  - Specify the size of string-type parameters. This is because SQL Server can re-use execution plans if the parameters match in type *and* size. Use -1 for MAX
  - Do not use the method AddWithValue, the main reason is it is very easy to forget to specify the parameter type or the precision/scale when needed. For additional information see Can we stop using AddWithValue already?
- When using database connections
  - Open the connection as late as possible and close it as soon as possible. This is a general guideline when working with any external resource
  - Never share database connection instances (example: having a singleton host a shared instance of type `SqlConnection`). Have your code always create a new database connection instance when needed and then have the calling code dispose of it and "throw it away" when it is done. The reason for this is
    - Most database providers have some sort of connection pooling so creating new managed connections is cheap
    - It eliminates any future errors if the code starts working with multiple threads

Section 25.2: Executing SQL statements as a command

```
// Uses Windows authentication. Replace the Trusted_Connection parameter with
```

```
// User Id=...;Password=...; 以改用SQL Server身份验证。你可能需要找到适合你服务器的连接字符串
。
string connectionString =
@"Server=myServer\myInstance;Database=myDataBase;Trusted_Connection=True;"

string sql = "INSERT INTO myTable (myDateTimeField, myIntField) " +
    "VALUES (@someDateTime, @someInt);";

// 大多数ADO.NET对象都是可释放的，因此需要使用using关键字。
using (var connection = new SqlConnection(connectionString))
using (var command = new SqlCommand(sql, connection))
{
    // 使用参数而非字符串拼接来添加用户提供的值，以避免SQL注入和格式问题。明确指定数
    据类型。

    // System.Data.SqlDbType 是一个枚举。参见注释1
command.Parameters.Add("@someDateTime", SqlDbType.DateTime).Value = myDateTimeVariable;
    command.Parameters.Add("@someInt", SqlDbType.Int).Value = myInt32Variable;

    // 执行 SQL 语句。对于返回结果的查询，请使用 ExecuteScalar 和 ExecuteReader// （或者
    参见更具体的示例，一旦添加后）。


connection.Open();
    command.ExecuteNonQuery();
}
```

注释 1： 请参见 SqlDbType 枚举，了解微软 SQL Server 特定的变体。

注释 2： 请参见 MySqlDbType 枚举，了解 MySQL 特定的变体。

第 25.3 节：使用通用接口抽象供应商特定类

```
var providerName = "System.Data.SqlClient";    //Oracle.ManagedDataAccess.Client, IBM.Data.DB2
var connectionString = "{your-connection-string}";
                    // 你可能会从 .config 文件中的 ConnectionStringSettings 对象获取上述两个值

var factory = DbProviderFactories.GetFactory(providerName);
using(var connection = factory.CreateConnection()) {    //IDbConnection
    connection.ConnectionString = connectionString;
connection.Open();

    using(var command = connection.CreateCommand()) {    //IDbCommand
        command.CommandText = "{query}";

        using(var reader = command.ExecuteReader()) {    //IDataReader
            while(reader.Read()) {
                ...
            }
        }
    }
}
```



```
// User Id=...;Password=...; to use SQL Server authentication instead. You may
// want to find the appropriate connection string for your server.
string connectionString =
@"Server=myServer\myInstance;Database=myDataBase;Trusted_Connection=True;"

string sql = "INSERT INTO myTable (myDateTimeField, myIntField) " +
    "VALUES (@someDateTime, @someInt);";

// Most ADO.NET objects are disposable and, thus, require the using keyword.
using (var connection = new SqlConnection(connectionString))
using (var command = new SqlCommand(sql, connection))
{
    // Use parameters instead of string concatenation to add user-supplied
    // values to avoid SQL injection and formatting issues. Explicitly supply datatype.

    // System.Data.SqlDbType is an enumeration. See Note1
    command.Parameters.Add("@someDateTime", SqlDbType.DateTime).Value = myDateTimeVariable;
    command.Parameters.Add("@someInt", SqlDbType.Int).Value = myInt32Variable;

    // Execute the SQL statement. Use ExecuteScalar and ExecuteReader instead
    // for query that return results (or see the more specific examples, once
    // those have been added).

    connection.Open();
    command.ExecuteNonQuery();
}
```

Note 1: Please see SqlDbType Enumeration for the MSFT SQL Server-specific variation.

Note 2: Please see MySqlDbType Enumeration for the MySQL-specific variation.

Section 25.3: Using common interfaces to abstract away vendor specific classes

```
var providerName = "System.Data.SqlClient";    //Oracle.ManagedDataAccess.Client, IBM.Data.DB2
var connectionString = "{your-connection-string}";
//you will probably get the above two values in the ConnectionStringSettings object from .config file

var factory = DbProviderFactories.GetFactory(providerName);
using(var connection = factory.CreateConnection()) {    //IDbConnection
    connection.ConnectionString = connectionString;
    connection.Open();

    using(var command = connection.CreateCommand()) {    //IDbCommand
        command.CommandText = "{query}";

        using(var reader = command.ExecuteReader()) {    //IDataReader
            while(reader.Read()) {
                ...
            }
        }
    }
}
```

第26章：依赖注入

第26.1节：依赖注入如何使单元测试更简单

这基于之前的`Greeter`类示例，该类有两个依赖项，`IGreetingProvider`和`IGreetingWriter`。

IGreetingProvider的实际实现可能是从API调用或数据库中获取字符串。IGreetingWriter的实现可能是在控制台显示问候语。但由于Greeter将其依赖项注入到构造函数中，因此很容易编写单元测试，注入这些接口的模拟版本。在实际中，我们可能会使用像Moq这样的框架，但在本例中我将编写这些模拟实现。

```
public class TestGreetingProvider : IGreetingProvider
{
    public const string TestGreeting = "Hello!";

    public string GetGreeting()
    {
        return TestGreeting;
    }
}

public class TestGreetingWriter : List<string>, IGreetingWriter
{
    public void WriteGreeting(string greeting)
    {
        Add(greeting);
    }
}

[TestClass]
public class GreeterTests
{
    [TestMethod]
    public void Greeter_WritesGreeting()
    {
        var greetingProvider = new TestGreetingProvider();
        var greetingWriter = new TestGreetingWriter();
        var greeter = new Greeter(greetingProvider, greetingWriter);
        greeter.Greet();
Assert.AreEqual(greetingWriter[0], TestGreetingProvider.TestGreeting);
    }
}
```

IGreetingProvider 和 IGreetingWriter 的行为与本测试无关。我们想测试的是 Greeter 获取问候语并写出它。Greeter 的设计（使用依赖注入）允许我们注入模拟的依赖项，而无需任何复杂的操作。我们测试的只是 Greeter 是否按预期与这些依赖项交互。

第26.2节：依赖注入 - 简单示例

这个类被称为`Greeter`。它的职责是输出问候语。它有两个*依赖项*。它需要某种能够提供要输出的问候语的东西，然后还需要一种输出该问候语的方式。这些依赖项都被描述为接口，分别是`IGreetingProvider`和`IGreetingWriter`。在这个例子中，这两个依赖项被“注入”到`Greeter`中。（示例后有进一步说明。）

Chapter 26: Dependency Injection

Section 26.1: How Dependency Injection Makes Unit Testing Easier

This builds on the previous example of the `Greeter` class which has two dependencies, `IGreetingProvider` and `IGreetingWriter`.

The actual implementation of `IGreetingProvider` might retrieve a string from an API call or a database. The implementation of `IGreetingWriter` might display the greeting in the console. But because `Greeter` has its dependencies injected into its constructor, it's easy to write a unit test that injects mocked versions of those interfaces. In real life we might use a framework like Moq, but in this case I'll write those mocked implementations.

```
public class TestGreetingProvider : IGreetingProvider
{
    public const string TestGreeting = "Hello!";

    public string GetGreeting()
    {
        return TestGreeting;
    }
}

public class TestGreetingWriter : List<string>, IGreetingWriter
{
    public void WriteGreeting(string greeting)
    {
        Add(greeting);
    }
}

[TestClass]
public class GreeterTests
{
    [TestMethod]
    public void Greeter_WritesGreeting()
    {
        var greetingProvider = new TestGreetingProvider();
        var greetingWriter = new TestGreetingWriter();
        var greeter = new Greeter(greetingProvider, greetingWriter);
        greeter.Greet();
        Assert.AreEqual(greetingWriter[0], TestGreetingProvider.TestGreeting);
    }
}
```

The behavior of `IGreetingProvider` and `IGreetingWriter` are not relevant to this test. We want to test that `Greeter` gets a greeting and writes it. The design of `Greeter` (using dependency injection) allows us to inject mocked dependencies without any complicated moving parts. All we're testing is that `Greeter` interacts with those dependencies as we expect it to.

Section 26.2: Dependency Injection - Simple example

This class is called `Greeter`. Its responsibility is to output a greeting. It has two *dependencies*. It needs something that will give it the greeting to output, and then it needs a way to output that greeting. Those dependencies are both described as interfaces, `IGreetingProvider` and `IGreetingWriter`. In this example, those two dependencies are "injected" into `Greeter`. (Further explanation following the example.)

```
public class Greeter
{
    private readonly IGreetingProvider _greetingProvider;
    private readonly IGreetingWriter _greetingWriter;

    public Greeter(IGreetingProvider greetingProvider, IGreetingWriter greetingWriter)
    {
_greetingProvider = greetingProvider;
        _greetingWriter = greetingWriter;
    }

    public void Greet()
    {
        var greeting = _greetingProvider.GetGreeting();
        _greetingWriter.WriteGreeting(greeting);
    }
}

public interface IGreetingProvider
{
    string GetGreeting();
}

public interface IGreetingWriter
{
    void WriteGreeting(string greeting);
}
```

Greeting 类依赖于 IGreetingProvider 和 IGreetingWriter，但它不负责创建任一实例。相反，它在构造函数中需要它们。无论是什么创建了 Greeting 的实例，都必须提供这两个依赖项。我们可以称之为“注入”依赖项。

因为依赖项是在类的构造函数中提供的，这也称为“构造函数注入”。

一些常见的约定：

- 构造函数将依赖项保存为private字段。一旦类被实例化，这些依赖项就可以被类的所有其他非静态方法使用。
- 这些private字段是readonly的。一旦在构造函数中设置，它们就不能被更改。这表明这些字段不应该（也不能）在构造函数外被修改。这进一步确保了这些依赖项在类的生命周期内始终可用。

- 依赖项是接口。这不是严格必要的，但很常见，因为这使得用另一个实现替换依赖项变得更容易。它还允许为单元测试目的提供接口的模拟版本。

第26.3节：我们为什么使用依赖注入容器（IoC容器）

依赖注入意味着编写类时不控制它们的依赖项——相反，依赖项是被提供给它们的（“注入”）。

这与使用依赖注入框架（通常称为“DI容器”、“IoC容器”或简称“容器”）如Castle Windsor、Autofac、SimpleInjector、Ninject、Unity等不同。

容器只是让依赖注入更容易。例如，假设你编写了多个依赖依赖注入的类。一个类依赖多个接口，实现这些接口的类又依赖其他接口，依此类推。有些依赖于特定的值。并且为了趣味，有些类


```
public class Greeter
{
    private readonly IGreetingProvider _greetingProvider;
    private readonly IGreetingWriter _greetingWriter;

    public Greeter(IGreetingProvider greetingProvider, IGreetingWriter greetingWriter)
    {
        _greetingProvider = greetingProvider;
        _greetingWriter = greetingWriter;
    }

    public void Greet()
    {
        var greeting = _greetingProvider.GetGreeting();
        _greetingWriter.WriteGreeting(greeting);
    }
}

public interface IGreetingProvider
{
    string GetGreeting();
}

public interface IGreetingWriter
{
    void WriteGreeting(string greeting);
}
```

The `Greeting` class depends on both `IGreetingProvider` and `IGreetingWriter`, but it is not responsible for creating instances of either. Instead it requires them in its constructor. Whatever creates an instance of `Greeting` must provide those two dependencies. We can call that "injecting" the dependencies.

Because dependencies are provided to the class in its constructor, this is also called "constructor injection."

A few common conventions:

- The constructor saves the dependencies as `private` fields. As soon as the class is instantiated, those dependencies are available to all other non-static methods of the class.
- The `private` fields are `readonly`. Once they are set in the constructor they cannot be changed. This indicates that those fields should not (and cannot) be modified outside of the constructor. That further ensures that those dependencies will be available for the lifetime of the class.
- The dependencies are interfaces. This is not strictly necessary, but is common because it makes it easier to substitute one implementation of the dependency with another. It also allows providing a mocked version of the interface for unit testing purposes.

Section 26.3: Why We Use Dependency Injection Containers (IoC Containers)

Dependency injection means writing classes so that they do not control their dependencies - instead, their dependencies are provided to them ("injected.")

This is not the same thing as using a dependency injection framework (often called a "DI container", "IoC container", or just "container") like Castle Windsor, Autofac, SimpleInjector, Ninject, Unity, or others.

A container just makes dependency injection easier. For example, suppose you write a number of classes that rely on dependency injection. One class depends on several interfaces, the classes that implement those interfaces depend on other interfaces, and so on. Some depend on specific values. And just for fun, some of those classes

实现了IDisposable接口，需要被释放。

每个单独的类都写得很好且易于测试。但现在出现了一个不同的问题：创建一个类的实例变得更加复杂。假设我们正在创建一个CustomerService类的实例。它有依赖项，而它的依赖项又有依赖项。构造一个实例可能看起来像这样：

```
public CustomerData GetCustomerData(string customerNumber)
{
    var customerApiEndpoint = ConfigurationManager.AppSettings["customerApi:customerApiEndpoint"];
    var logFilePath = ConfigurationManager.AppSettings["logwriter:logFilePath"];
    var authConnectionString =
ConfigurationManager.ConnectionStrings["authorization"].ConnectionString;
    using(var logWriter = new LogWriter(logFilePath ))
    {
        using(var customerApiClient = new CustomerApiClient(customerApiEndpoint))
        {
            var customerService = new CustomerService(
                new SqlAuthorizationRepository(authorizationConnectionString, logWriter),
                new CustomerDataRepository(customerApiClient, logWriter),
logWriter
            );

            // 仅仅是为了创建一个CustomerService的实例！
            return customerService.GetCustomerData(string customerNumber);
        }
    }
}
```

你可能会想，为什么不把整个庞大的构造放在一个单独的函数中，只返回CustomerService？其中一个原因是因为每个类的依赖项都被注入到它里面，类本身不需要负责知道这些依赖项是否是IDisposable类型或负责释放它们。它只是使用它们。所以如果我们有一个GetCustomerService()函数返回一个完全构造好的CustomerService，这个类可能包含许多可释放的资源，但没有办法访问或释放它们。

除了释放IDisposable之外，谁还想调用一系列嵌套的构造函数呢？这只是一个简短的例子，情况可能会变得更糟。再次强调，这并不意味着我们写的类是错误的。
这些类本身可能是完美的。挑战在于如何将它们组合在一起。

依赖注入容器简化了这个过程。它允许我们指定使用哪个类或值来满足每个依赖项。这个稍微简化的例子使用了Castle Windsor：

```
var container = new WindsorContainer()
container.Register(
Component.For<CustomerService>(),
Component.For<ILogWriter, LogWriter>()
        .DependsOn(Dependency.OnAppSettingsValue("logFilePath", "logWriter:logFilePath")),
    Component.For<IAuthorizationRepository, SqlAuthorizationRepository>()
        .DependsOn(Dependency.OnValue(connectionString,
ConfigurationManager.ConnectionStrings["authorization"].ConnectionString)),
    Component.For<ICustomerDataProvider, CustomerApiClient>()
         .DependsOn(Dependency.OnAppSettingsValue("apiEndpoint",
"customerApi:customerApiEndpoint"))
);
```

我们称之为“注册依赖”或“配置容器”。换句话说，这告诉我们的WindsorContainer：

- 如果某个类需要ILogWriter，就创建一个LogWriter的实例。 LogWriter需要一个文件路径。使用AppSettings中的该值。

implement `IDisposable` and need to be disposed.

Each individual class is well-written and easy to test. But now there's a different problem: Creating an instance of a class has become much more complicated. Suppose we're creating an instance of a `CustomerService` class. It has dependencies and its dependencies have dependencies. Constructing an instance might look something like this:

```
public CustomerData GetCustomerData(string customerNumber)
{
    var customerApiEndpoint = ConfigurationManager.AppSettings["customerApi:customerApiEndpoint"];
    var logFilePath = ConfigurationManager.AppSettings["logwriter:logFilePath"];
    var authConnectionString =
ConfigurationManager.ConnectionStrings["authorization"].ConnectionString;
    using(var logWriter = new LogWriter(logFilePath ))
    {
        using(var customerApiClient = new CustomerApiClient(customerApiEndpoint))
        {
            var customerService = new CustomerService(
                new SqlAuthorizationRepository(authorizationConnectionString, logWriter),
                new CustomerDataRepository(customerApiClient, logWriter),
                logWriter
            );

            // All this just to create an instance of CustomerService!
            return customerService.GetCustomerData(string customerNumber);
        }
    }
}
```

You might wonder, why not put the whole giant construction in a separate function that just returns `CustomerService`? One reason is that because the dependencies for each class are injected into it, a class isn't responsible for knowing whether those dependencies are `IDisposable` or disposing them. It just uses them. So if a we had a `GetCustomerService()` function that returned a fully-constructed `CustomerService`, that class might contain a number of disposable resources and no way to access or dispose them.

And aside from disposing `IDisposable`, who wants to call a series of nested constructors like that, ever? That's a short example. It could get much, much worse. Again, that doesn't mean that we wrote the classes the wrong way. The classes might be individually perfect. The challenge is composing them together.

A dependency injection container simplifies that. It allows us to specify which class or value should be used to fulfill each dependency. This slightly oversimplified example uses Castle Windsor:

```
var container = new WindsorContainer()
container.Register(
    Component.For<CustomerService>(),
    Component.For<ILogWriter, LogWriter>()
        .DependsOn(Dependency.OnAppSettingsValue("logFilePath", "logWriter:logFilePath")),
    Component.For<IAuthorizationRepository, SqlAuthorizationRepository>()
        .DependsOn(Dependency.OnValue(connectionString,
ConfigurationManager.ConnectionStrings["authorization"].ConnectionString)),
    Component.For<ICustomerDataProvider, CustomerApiClient>()
         .DependsOn(Dependency.OnAppSettingsValue("apiEndpoint",
"customerApi:customerApiEndpoint"))
);
```

We call this "registering dependencies" or "configuring the container." Translated, this tells our `WindsorContainer`:

- If a class requires `ILogWriter`, create an instance of `LogWriter`. `LogWriter` requires a file path. Use this value from `AppSettings`.

- 如果某个类需要IAuthorizationRepository，就创建一个SqlAuthorizationRepository的实例。它需要一个连接字符串。使用ConnectionStrings部分中的该值。
- 如果某个类需要ICustomerDataProvider，就创建一个CustomerApiClient，并提供它从AppSettings中需要的字符串。

当我们从容器请求依赖时，我们称之为“解析”依赖。直接使用容器进行解析是不好的做法，但那是另一个话题。为了演示目的，我们现在可以这样做：

```
var customerService = container.Resolve<CustomerService>();
var data = customerService.GetCustomerData(customerNumber);
container.Release(customerService);
```

容器知道CustomerService依赖于IAuthorizationRepository和ICustomerDataProvider。它知道需要创建哪些类来满足这些需求。这些类反过来又有更多的依赖，容器知道如何满足这些依赖。它会创建所有需要的类，直到能够返回一个CustomerService的实例。

如果遇到某个类需要一个我们未注册的依赖，比如IDoesSomethingElse，那么当我们尝试解析CustomerService时，它会抛出一个明确的异常，告诉我们没有注册任何内容来满足该需求。

每个依赖注入（DI）框架的行为略有不同，但通常它们会让我们对某些类的实例化方式有一定的控制。例如，我们是希望它创建一个LogWriter实例并提供给所有依赖ILogWriter的类，还是希望每次都创建一个新的实例？大多数容器都有指定这一点的方法。

实现IDisposable接口的类怎么办？这就是为什么我们在最后调用container.Release(customerService);的原因。大多数容器（包括Windsor）会回溯所有创建的依赖项，并Dispose那些需要释放的对象。如果CustomerService实现了IDisposable接口，它也会被释放。

如上所示注册依赖项可能看起来只是多写了些代码。但当我们有很多类且依赖项众多时，这样做确实很有价值。如果我们不得不编写那些不使用依赖注入的类，那么同样包含大量类的应用程序将变得难以维护和测试。

这只是简单介绍了为什么我们使用依赖注入容器。至于如何配置我们的应用程序以使用依赖注入容器（并正确使用），这不仅是一个话题——而是多个话题，因为不同容器的使用说明和示例各不相同。

- If a class requires `IAuthorizationRepository`, create an instance of `SqlAuthorizationRepository`. It requires a connection string. Use this value from the `ConnectionStrings` section.
- If a class requires `ICustomerDataProvider`, create a `CustomerApiClient` and provide the string it needs from `AppSettings`.

When we request a dependency from the container we call that "resolving" a dependency. It's bad practice to do that directly using the container, but that's a different story. For demonstration purposes, we could now do this:

```
var customerService = container.Resolve<CustomerService>();
var data = customerService.GetCustomerData(customerNumber);
container.Release(customerService);
```

The container knows that `CustomerService` depends on `IAuthorizationRepository` and `ICustomerDataProvider`. It knows what classes it needs to create to fulfill those requirements. Those classes, in turn, have more dependencies, and the container knows how to fulfill those. It will create every class it needs to until it can return an instance of `CustomerService`.

If it gets to a point where a class requires a dependency that we haven't registered, like `IDoesSomethingElse`, then when we try to resolve `CustomerService` it will throw a clear exception telling us that we haven't registered anything to fulfill that requirement.

Each DI framework behaves a little differently, but typically they give us some control over how certain classes are instantiated. For example, do we want it to create one instance of `LogWriter` and provide it to every class that depends on `ILogWriter`, or do we want it to create a new one every time? Most containers have a way to specify that.

What about classes that implement `IDisposable`? That's why we call `container.Release(customerService);` at the end. Most containers (including Windsor) will step back through all of the dependencies created and `Dispose` the ones that need disposing. If `CustomerService` is `IDisposable` it will dispose that too.

Registering dependencies as seen above might just look like more code to write. But when we have lots of classes with lots of dependencies then it really pays off. And if we had to write those same classes *without* using dependency injection then that same application with lots of classes would become difficult to maintain and test.

This scratches the surface of *why* we use dependency injection containers. *How* we configure our application to use one (and use it correctly) is not just one topic - it's a number of topics, as the instructions and examples vary from one container to the next.

第27章：平台调用

第27.1节：结构体的封送处理

简单结构体

C++签名：

```
typedef struct _PERSON
{
     int age;
     char name[32];
} PERSON, *LP_PERSON;

void GetSpouse(PERSON person, LP_PERSON spouse);
```

C# 定义

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct PERSON
{
    public int age;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
    public string name;
}

[DllImport("family.dll", CharSet = CharSet.Auto)]
public static extern bool GetSpouse(PERSON person, ref PERSON spouse);
```

包含未知大小数组字段的结构体。传入

C++ 函数签名

```
typedef struct
{
     int length;
     int *data;
}  VECTOR ;

void SetVector(VECTOR &vector);
```

当从托管代码传递到非托管代码时，

该data数组应定义为 IntPtr，并且应显式分配内存，方法是
使用 Marshal.AllocHGlobal()（之后用Marshal.FreeHGlobal()释放）：

```
[StructLayout(LayoutKind.Sequential)]
public struct VECTOR : IDisposable
{
    int length;
    IntPtr dataBuf;

    public int[] data
    {
        set
        {
FreeDataBuf();
```


Chapter 27: Platform Invoke

Section 27.1: Marshaling structs

Simple struct

C++ signature:

```
typedef struct _PERSON
{
    int age;
    char name[32];
} PERSON, *LP_PERSON;

void GetSpouse(PERSON person, LP_PERSON spouse);
```

C# definition

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct PERSON
{
    public int age;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
    public string name;
}

[DllImport("family.dll", CharSet = CharSet.Auto)]
public static extern bool GetSpouse(PERSON person, ref PERSON spouse);
```

Struct with unknown size array fields. Passing in

C++ signature

```
typedef struct
{
    int length;
    int *data;
} VECTOR;

void SetVector(VECTOR &vector);
```

When passed from managed to unmanaged code, this

The data array should be defined as IntPtr and memory should be explicitly allocated with
Marshal.AllocHGlobal() (and freed with Marshal.FreeHGlobal() afterwords):

```
[StructLayout(LayoutKind.Sequential)]
public struct VECTOR : IDisposable
{
    int length;
    IntPtr dataBuf;

    public int[] data
    {
        set
        {
            FreeDataBuf();
```

```
            if (value != null && value.Length > 0)
            {
dataBuf = Marshal.AllocHGlobal(value.Length * Marshal.SizeOf(value[0]));
                Marshal.Copy(value, 0, dataBuf, value.Length);
length = value.Length;
            }
        }
    }
    void FreeDataBuf()
    {
        if (dataBuf != IntPtr.Zero)
        {
Marshal.FreeHGlobal(dataBuf);
            dataBuf = IntPtr.Zero;
        }
    }
    public void Dispose()
    {
FreeDataBuf();
    }
}

[DllImport("vectors.dll")]
public static extern void SetVector([In]ref VECTOR vector);
```

包含未知大小数组字段的结构体。接收中

C++签名：

```
typedef struct
{
    char *name;
}  USER ;

bool   GetCurrentUser (USER *user);
```

当此类数据从非托管代码传出且内存由非托管函数分配时，托管调用者应将其接收为一个 IntPtr 变量，并将缓冲区转换为托管数组。对于字符串，有一个方便的 Marshal.PtrToStringAnsi() 方法：

```
[StructLayout(LayoutKind.Sequential)]
public struct USER
{
IntPtr nameBuffer;
    public string name { get { return Marshal.PtrToStringAnsi(nameBuffer); } }
}

[DllImport("users.dll")]
public static extern bool GetCurrentUser(out USER user);
```

第27.2节：联合体的封送处理

仅限值类型字段

C++ 声明

```
typedef union
{
    char c;
```



```
            if (value != null && value.Length > 0)
            {
                dataBuf = Marshal.AllocHGlobal(value.Length * Marshal.SizeOf(value[0]));
                Marshal.Copy(value, 0, dataBuf, value.Length);
                length = value.Length;
            }
        }
    }
    void FreeDataBuf()
    {
        if (dataBuf != IntPtr.Zero)
        {
            Marshal.FreeHGlobal(dataBuf);
            dataBuf = IntPtr.Zero;
        }
    }
    public void Dispose()
    {
        FreeDataBuf();
    }
}

[DllImport("vectors.dll")]
public static extern void SetVector([In]ref VECTOR vector);
```

Struct with unknown size array fields. Receiving

C++ signature:

```
typedef struct
{
    char *name;
} USER;

bool GetCurrentUser(USER *user);
```

When such data is passed out of unmanaged code and memory is allocated by the unmanaged functions, the managed caller should receive it into an `IntPrt` variable and convert the buffer to a managed array. In case of strings there is a convenient `Marshal.PtrToStringAnsi()` method:

```
[StructLayout(LayoutKind.Sequential)]
public struct USER
{
    IntPtr nameBuffer;
    public string name { get { return Marshal.PtrToStringAnsi(nameBuffer); } }
}

[DllImport("users.dll")]
public static extern bool GetCurrentUser(out USER user);
```

Section 27.2: Marshaling unions

Value-type fields only

C++ declaration

```
typedef union
{
    char c;
```

```
    int i;
} CharOrInt;
```

C# 声明

```
[StructLayout(LayoutKind.Explicit)]
public struct CharOrInt
{
    [FieldOffset(0)]
    public byte c;
    [FieldOffset(0)]
    public int i;
}
```

混合值类型和引用字段

不允许将引用值与值类型重叠，因此不能简单地使用 `FieldOffset(0) text; FieldOffset(0) i;` 这段代码将无法编译

```
typedef union
{
    char text[128];
    int i;
} TextOrInt;
```

通常情况下，你需要使用自定义封送处理。然而，在像这种特定情况下，可以使用更简单的技术：

```
[StructLayout(LayoutKind.Sequential)]
public struct TextOrInt
{
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 128)]
    public byte[] text;
    public int i { get { return BitConverter.ToInt32(text, 0); } }
}
```

第27.3节：调用Win32 dll函数

```
using System.Runtime.InteropServices;

class PInvokeExample
{
    [DllImport("user32.dll", CharSet = CharSet.Auto)]
    public static extern uint MessageBox(IntPtr hWnd, String text, String caption, int options);

    public static void test()
    {
MessageBox(IntPtr.Zero, "Hello!", "Message", 0);
    }
}
```

将函数声明为static extern类型，使用DllImportAttribute并将其Value属性设置为.dll名称。别忘了使用System.Runtime.InteropServices命名空间。然后像调用普通静态方法一样调用它。

平台调用服务将负责加载.dll并找到所需的函数。P/Invoke在最简单的情况下还会将参数和返回值在.dll和.NET之间进行封送处理（即从.NET数据类型转换为Win32类型，反之亦然）。


```
    int i;
} CharOrInt;
```

C# declaration

```
[StructLayout(LayoutKind.Explicit)]
public struct CharOrInt
{
    [FieldOffset(0)]
    public byte c;
    [FieldOffset(0)]
    public int i;
}
```

Mixing value-type and reference fields

Overlapping a reference value with a value type one is not allowed so you cannot simply use the `FieldOffset(0) text; FieldOffset(0) i;` will not compile for

```
typedef union
{
    char text[128];
    int i;
} TextOrInt;
```

and generally you would have to employ custom marshaling. However, in particular cases like this simpler technics may be used:

```
[StructLayout(LayoutKind.Sequential)]
public struct TextOrInt
{
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 128)]
    public byte[] text;
    public int i { get { return BitConverter.ToInt32(text, 0); } }
}
```

Section 27.3: Calling a Win32 dll function

```
using System.Runtime.InteropServices;

class PInvokeExample
{
    [DllImport("user32.dll", CharSet = CharSet.Auto)]
    public static extern uint MessageBox(IntPtr hWnd, String text, String caption, int options);

    public static void test()
    {
        MessageBox(IntPtr.Zero, "Hello!", "Message", 0);
    }
}
```

Declare a function as `static extern` stting `DllImportAttribute` with its `Value` property set to .dll name. Don't forget to use `System.Runtime.InteropServices` namespace. Then call it as an regular static method.

The Platform Invocation Services will take care of loading the .dll and finding the desired finction. The P/Invoke in most simple cases will also marshal parameters and return value to and from the .dll (i.e. convert from .NET datatypes to Win32 ones and vice versa).

第27.4节：使用Windows API

使用pinvoke.net。

在代码中声明extern Windows API函数之前，建议先在pinvoke.net上查找。那里很可能已经有合适的声明，包含所有支持类型和良好的示例。

第27.5节：数组的封送处理

简单类型的数组

```
[DllImport("Example.dll")]
static extern void SetArray(
    [MarshalAs(UnmanagedType.LPArray, SizeConst = 128)]
    byte[] data);
```

字符串数组

```
[DllImport("Example.dll")]
static extern void SetStrArray(string[] textLines);
```


Section 27.4: Using Windows API

Use pinvoke.net.

Before declaring an `extern` Windows API function in your code, consider looking for it on pinvoke.net. They most likely already have a suitable declaration with all supporting types and good examples.

Section 27.5: Marshalling arrays

Arrays of simple type

```
[DllImport("Example.dll")]
static extern void SetArray(
    [MarshalAs(UnmanagedType.LPArray, SizeConst = 128)]
    byte[] data);
```

Arrays of string

```
[DllImport("Example.dll")]
static extern void SetStrArray(string[] textLines);
```

第28章：NuGet打包系统

第28.1节：从解决方案中的一个项目卸载包

```
PM> Uninstall-Package -ProjectName MyProjectB EntityFramework
```

第28.2节：安装特定版本的包

```
PM> Install-Package EntityFramework -Version 6.1.2
```

第28.3节：添加包源（MyGet、Klondike等）

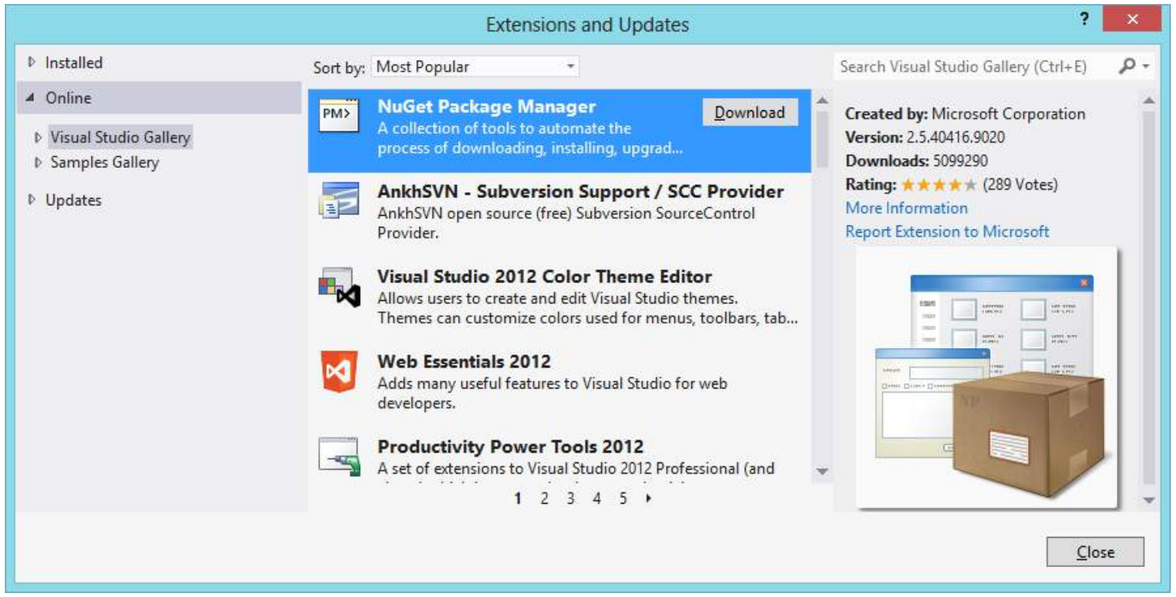
```
nuget sources add -name feedname -source http://sourcefeedurl
```

第28.4节：安装NuGet包管理器

为了能够管理项目的包，您需要NuGet包管理器。这是一个Visual Studio扩展，官方文档中有说明：安装和更新NuGet客户端。

从Visual Studio 2012开始，NuGet包含在每个版本中，可以通过以下路径使用：工具 -> NuGet包管理器 -> 包管理器控制台。

你可以通过 Visual Studio 的“工具”菜单，点击“扩展和更新”来操作：



这会安装图形用户界面（GUI）：

- 可通过点击项目或其“引用”文件夹上的“管理 NuGet 包...”来访问

以及包管理器控制台：

Chapter 28: NuGet packaging system

Section 28.1: Uninstalling a package from one project in a solution

```
PM> Uninstall-Package -ProjectName MyProjectB EntityFramework
```

Section 28.2: Installing a specific version of a package

```
PM> Install-Package EntityFramework -Version 6.1.2
```

Section 28.3: Adding a package source feed (MyGet, Klondike, ect)

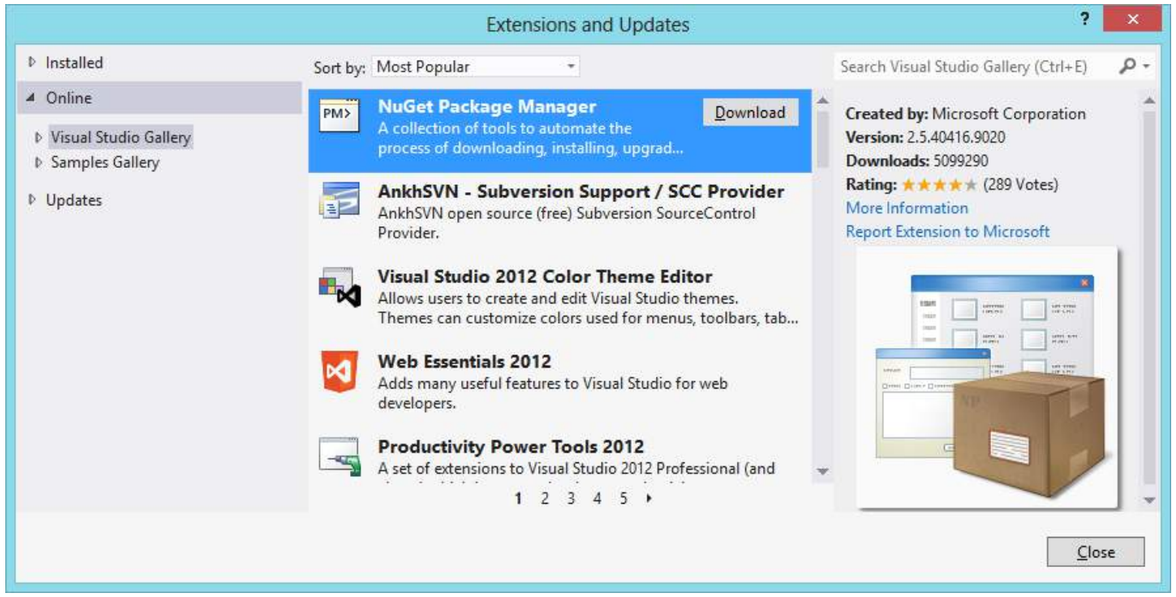
```
nuget sources add -name feedname -source http://sourcefeedurl
```

Section 28.4: Installing the NuGet Package Manager

In order to be able to manage your projects' packages, you need the NuGet Package Manager. This is a Visual Studio Extension, explained in the official docs: Installing and Updating NuGet Client.

Starting with Visual Studio 2012, NuGet is included in every edition, and can be used from: Tools -> NuGet Package Manager -> Package Manager Console.

You do so through the Tools menu of Visual Studio, clicking Extensions and Updates:



This installs both the GUI:

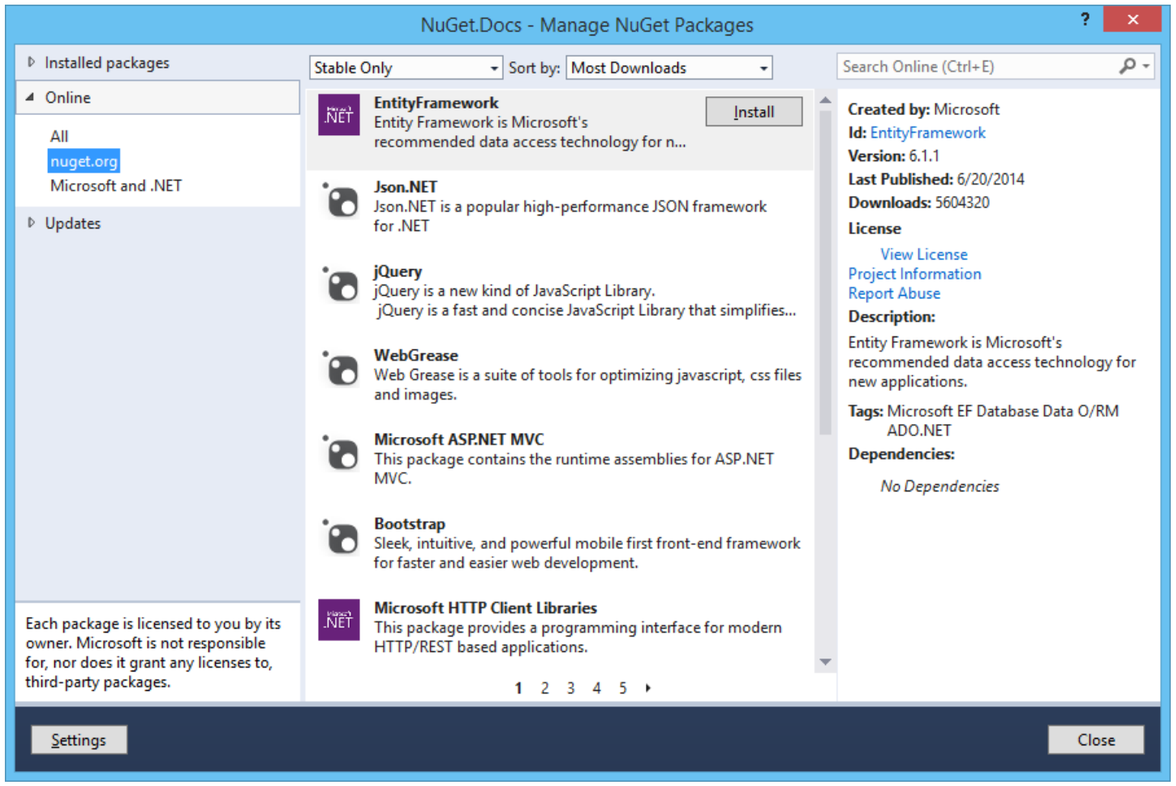
- Available through clicking "Manage NuGet Packages..." on a project or its References folder

And the Package Manager Console:

- 工具 -> NuGet 包管理器 -> 包管理器控制台。

第28.5节：通过用户界面管理包

当你右键点击项目（或其“引用”文件夹）时，可以点击“管理 NuGet 包...”选项。这会显示包管理器对话框。



第28.6节：通过控制台管理包

点击菜单 工具 -> NuGet 包管理器 -> 包管理器控制台，在你的集成开发环境中显示控制台。
官方文档在这里。

在这里，您可以执行包括install-package命令在内的操作，该命令会将输入的软件包安装到当前选定的“默认项目”中：

```
Install-Package Elmah
```

您也可以指定要安装包的项目，覆盖“默认项目”下拉列表中选定的项目：

```
Install-Package Elmah -ProjectName MyFirstWebsite
```

第28.7节：更新包

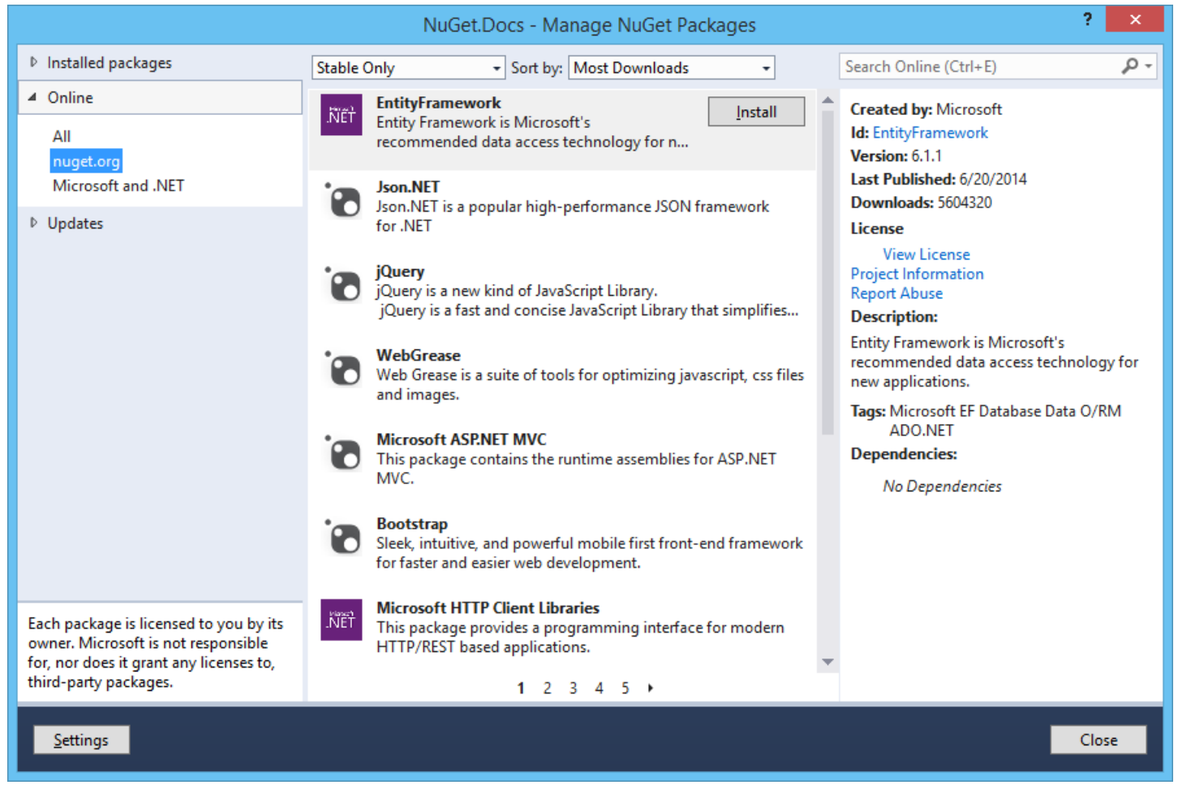
要更新包，请使用以下命令：

```
PM> Update-Package EntityFramework
```


- Tools -> NuGet Package Manager -> Package Manager Console.

Section 28.5: Managing Packages through the UI

When you right-click a project (or its References folder), you can click the "Manage NuGet Packages..." option. This shows the Package Manager Dialog.



Section 28.6: Managing Packages through the console

Click the menus Tools -> NuGet Package Manager -> Package Manager Console to show the console in your IDE.
Official documentation here.

Here you can issue, amongst others, `install-package` commands which installs the entered package into the currently selected "Default project":

```
Install-Package Elmah
```

You can also provide the project to install the package to, overriding the selected project in the "Default project" dropdown:

```
Install-Package Elmah -ProjectName MyFirstWebsite
```

Section 28.7: Updating a package

To update a package use the following command:

```
PM> Update-Package EntityFramework
```

其中 EntityFramework 是要更新的包的名称。请注意，更新命令会对所有项目运行，因此不同于Install-Package EntityFramework，该命令只会安装到“默认项目”。

您也可以显式指定单个项目：

```
PM> Update-Package EntityFramework -ProjectName MyFirstWebsite
```

第28.8节：卸载包

```
PM> 卸载包 EntityFramework
```

第28.9节：卸载特定版本的包

```
PM> 卸载包 EntityFramework -版本 6.1.2
```


where EntityFramework is the name of the package to be updated. Note that update will run for all projects, and so is different from `Install-Package EntityFramework` which would install to "Default project" only.

You can also specify a single project explicitly:

```
PM> Update-Package EntityFramework -ProjectName MyFirstWebsite
```

Section 28.8: Uninstalling a package

```
PM> Uninstall-Package EntityFramework
```

Section 28.9: Uninstall a specific version of package

```
PM> uninstall-Package EntityFramework -Version 6.1.2
```

第29章：ASP.NET MVC中的全球化使用智能国际化进行ASP.NET

第29.1节：基本配置和设置

1. 将I18N nuget包添加到你的MVC项目中。
2. 在web.config中，将 i18n.LocalizingModule添加到你的<httpModules>或<modules>部分。

```
<!-- IIS 6 -->
<httpModules>
  <add name="i18n.LocalizingModule" type="i18n.LocalizingModule, i18n" />
</httpModules>

<!-- IIS 7 -->
<system.webServer>
  <modules>
    <add name="i18n.LocalizingModule" type="i18n.LocalizingModule, i18n" />
  </modules>
</system.webServer>
```

3. 在网站根目录下添加一个名为“locale”的文件夹。为每个希望支持的语言文化创建一个子文件夹。例如，/locale/fr/。
4. 在每个特定语言文化的文件夹中，创建一个名为messages.po的文本文件。
5. 为了测试，在你的messages.po文件中输入以下文本行：

```
#: Translation test
msgid "Hello, world!"
msgstr "Bonjour le monde!"
```

6. 向您的项目添加一个控制器，该控制器返回一些需要翻译的文本。

```
using System.Web.Mvc;

namespace I18nDemo.Controllers
{
    public class DefaultController : Controller
    {
        public ActionResult Index()
        {
            // [[[三重括号]]] 内的文本必须与您的 .po 文件中的 msgid 完全匹配。

            return Content("[[[Hello, world!]]]");
        }
    }
}
```

7. 运行您的 MVC 应用程序并浏览到对应于您的控制器操作的路由，例如 http://localhost:[yourportnumber]/default。
   观察 URL 已更改以反映您的默认文化，例如 http://localhost:[yourportnumber]/en/default。
8. 将 URL 中的 /en/ 替换为 /fr/（或您选择的任何文化）。页面现在应显示您的文本的翻译版本。
9. 将浏览器的语言设置更改为偏好您的备用文化，然后再次浏览到/default。注意URL已更改以反映您的备用文化，并且显示了翻译后的文本。

Chapter 29: Globalization in ASP.NET MVC using Smart internationalization for ASP.NET

Section 29.1: Basic configuration and setup

1. Add the I18N nuget package to your MVC project.
2. In web.config, add the `i18n.LocalizingModule` to your **`<httpModules>`** or **`<modules>`** section.

```
<!-- IIS 6 -->
<httpModules>
  <add name="i18n.LocalizingModule" type="i18n.LocalizingModule, i18n" />
</httpModules>

<!-- IIS 7 -->
<system.webServer>
  <modules>
    <add name="i18n.LocalizingModule" type="i18n.LocalizingModule, i18n" />
  </modules>
</system.webServer>
```

3. Add a folder named "locale" to the root of your site. Create a subfolder for each culture you wish to support. For example, `/locale/fr/`.
4. In each culture-specific folder, create a text file named `messages.po`.
5. For testing purposes, enter the following lines of text in your `messages.po` file:

```
#: Translation test
msgid "Hello, world!"
msgstr "Bonjour le monde!"
```

6. Add a controller to your project which returns some text to translate.

```
using System.Web.Mvc;

namespace I18nDemo.Controllers
{
    public class DefaultController : Controller
    {
        public ActionResult Index()
        {
            // Text inside [[[triple brackets]]] must precisely match
            // the msgid in your .po file.
            return Content("[[[Hello, world!]]]");
        }
    }
}
```

7. Run your MVC application and browse to the route corresponding to your controller action, such as http://localhost:[yourportnumber]/default.
   Observe that the URL is changed to reflect your default culture, such as http://localhost:[yourportnumber]/en/default.
8. Replace `/en/` in the URL with `/fr/` (or whatever culture you've selected.) The page should now display the translated version of your text.
9. Change your browser's language setting to prefer your alternate culture and browse to `/default` again. Observe that the URL is changed to reflect your alternate culture and the translated text appears.

10. 在web.config中，添加处理程序以防止用户浏览到您的locale文件夹。

```
<!-- IIS 6 -->
<system.web>
  <httpHandlers>
    <add path="*" verb="*" type="System.Web.HttpNotFoundHandler"/>
  </httpHandlers>
</system.web>

<!-- IIS 7 -->
<system.webServer>
  <handlers>
    <remove name="BlockViewHandler"/>
   <add name="BlockViewHandler" path="*" verb="*" preCondition="integratedMode"
type="System.Web.HttpNotFoundHandler"/>
  </handlers>
</system.webServer>
```


10. In web.config, add handlers so that users cannot browse to your `locale` folder.

```
<!-- IIS 6 -->
<system.web>
  <httpHandlers>
    <add path="*" verb="*" type="System.Web.HttpNotFoundHandler"/>
  </httpHandlers>
</system.web>

<!-- IIS 7 -->
<system.webServer>
  <handlers>
    <remove name="BlockViewHandler"/>
   <add name="BlockViewHandler" path="*" verb="*" preCondition="integratedMode"
type="System.Web.HttpNotFoundHandler"/>
  </handlers>
</system.webServer>
```

第30章：System.Net.Mail

第30.1节：MailMessage

以下是创建带附件邮件消息的示例。创建后，我们使用SmtpClient类发送此消息。这里使用默认的25端口。

```
public class clsMail
{
    private static bool SendMail(string mailfrom, List<string>replytos, List<string> mailtos,
List<string> mailccs, List<string> mailbccs, string body, string subject, List<string> Attachment)
    {
        try
        {
            using(MailMessage MyMail = new MailMessage())
            {
MyMail.From = new MailAddress(mailfrom);
                foreach (string mailto in mailtos)
MyMail.To.Add(mailto);

                if (replytos != null && replytos.Any())
                {
                    foreach (string replyto in replytos)
                        MyMail.ReplyToList.Add(replyto);
                }

                if (mailccs != null && mailccs.Any())
                {
                    foreach (string mailcc in mailccs)
                        MyMail.CC.Add(mailcc);
                }

                if (mailbccs != null && mailbccs.Any())
                {
                    foreach (string mailbcc in mailbccs)
                        MyMail.Bcc.Add(mailbcc);
                }

MyMail.Subject = subject;
                MyMail.IsBodyHtml = true;
                MyMail.Body = body;
MyMail.Priority = MailPriority.Normal;

                if (Attachment != null && Attachment.Any())
                {
                    System.Net.Mail.Attachment attachment;
                    foreach (var item in Attachment)
                    {
attachment = new System.Net.Mail.Attachment(item);
                        MyMail.Attachments.Add(attachment);
                    }
                }

SmtpClient smtpMailObj = new SmtpClient();
                smtpMailObj.Host = "your host";
smtpMailObj.Port = 25;
smtpMailObj.Credentials = new System.Net.NetworkCredential("uid", "pwd");

                smtpMailObj.Send(MyMail);
                return true;
```


Chapter 30: System.Net.Mail

Section 30.1: MailMessage

Here is the example of creating of mail message with attachments. After creating we send this message with the help of `SmtpClient` class. Default 25 port is used here.

```
public class clsMail
{
    private static bool SendMail(string mailfrom, List<string>replytos, List<string> mailtos,
List<string> mailccs, List<string> mailbccs, string body, string subject, List<string> Attachment)
    {
        try
        {
            using(MailMessage MyMail = new MailMessage())
            {
                MyMail.From = new MailAddress(mailfrom);
                foreach (string mailto in mailtos)
                    MyMail.To.Add(mailto);

                if (replytos != null && replytos.Any())
                {
                    foreach (string replyto in replytos)
                        MyMail.ReplyToList.Add(replyto);
                }

                if (mailccs != null && mailccs.Any())
                {
                    foreach (string mailcc in mailccs)
                        MyMail.CC.Add(mailcc);
                }

                if (mailbccs != null && mailbccs.Any())
                {
                    foreach (string mailbcc in mailbccs)
                        MyMail.Bcc.Add(mailbcc);
                }

                MyMail.Subject = subject;
                MyMail.IsBodyHtml = true;
                MyMail.Body = body;
                MyMail.Priority = MailPriority.Normal;

                if (Attachment != null && Attachment.Any())
                {
                    System.Net.Mail.Attachment attachment;
                    foreach (var item in Attachment)
                    {
                        attachment = new System.Net.Mail.Attachment(item);
                        MyMail.Attachments.Add(attachment);
                    }
                }

                SmtpClient smtpMailObj = new SmtpClient();
                smtpMailObj.Host = "your host";
                smtpMailObj.Port = 25;
                smtpMailObj.Credentials = new System.Net.NetworkCredential("uid", "pwd");

                smtpMailObj.Send(MyMail);
                return true;
```

```
            }
        }
        catch
        {
            return false;
        }
    }
}
```

第30.2节：带附件的邮件

MailMessage 表示可以使用 SmtpClient 类进一步发送的邮件消息。可以向邮件消息添加多个附件（文件）。

```
using System.Net.Mail;

using(MailMessage myMail = new MailMessage())
{
Attachment attachment = new Attachment(path);
     myMail.Attachments.Add(attachment);

     // 进一步处理以发送邮件消息

}
```



```
            }
        }
        catch
        {
            return false;
        }
    }
}
```

Section 30.2: Mail with Attachment

`MailMessage` represents mail message which can be sent further using `SmtpClient` class. Several attachments (files) can be added to mail message.

```
using System.Net.Mail;

using(MailMessage myMail = new MailMessage())
{
     Attachment attachment = new Attachment(path);
     myMail.Attachments.Add(attachment);

     // further processing to send the mail message

}
```

第31章：使用 Progress<T> 和 IProgress<T>

第31.1节：简单的进度报告

IProgress<T> 可用于向另一个过程报告某个过程的进度。此示例展示了如何创建一个报告其进度的基本方法。

```
void Main()
{
IProgress<int> p = new Progress<int>(progress =>
    {
Console.WriteLine("运行步骤: {0}", progress);
    });
LongJob(p);
}

public void LongJob(IProgress<int> progress)
{
    var max = 10;
    for (int i = 0; i < max; i++)
    {
progress.Report(i);
    }
}
```

输出：

```
运行步骤: 0
运行步骤: 3
运行步骤: 4
运行步骤: 5
运行步骤: 6
运行步骤: 7
运行步骤: 8
运行步骤: 9
运行步骤: 2
运行步骤: 1
```

请注意，当此代码运行时，您可能会看到数字输出顺序混乱。这是因为 `IProgress<T>.Report()` 方法是异步运行的，因此不太适合需要按顺序报告进度的情况。

第31.2节：使用 IProgress<T>

需要注意的是，System.Progress<T> 类没有可用的 Report() 方法。该方法是从 IProgress<T> 接口显式实现的，因此必须在一个
Progress<T> 当它被转换为 IProgress<T> 时。

```
var p1 = new Progress<int>();
p1.Report(1); //编译错误，Progress 不包含方法 'Report'

IProgress<int> p2 = new Progress<int>();
p2.Report(2); //有效
```


Chapter 31: Using Progress<T> and IProgress<T>

Section 31.1: Simple Progress reporting

`IProgress<T>` can be used to report progress of some procedure to another procedure. This example shows how you can create a basic method that reports its progress.

```
void Main()
{
    IProgress<int> p = new Progress<int>(progress =>
    {
        Console.WriteLine("Running Step: {0}", progress);
    });
    LongJob(p);
}

public void LongJob(IProgress<int> progress)
{
    var max = 10;
    for (int i = 0; i < max; i++)
    {
        progress.Report(i);
    }
}
```

Output:

```
Running Step: 0
Running Step: 3
Running Step: 4
Running Step: 5
Running Step: 6
Running Step: 7
Running Step: 8
Running Step: 9
Running Step: 2
Running Step: 1
```

Note that when you this code runs, you may see numbers be output out of order. This is because the `IProgress<T>.Report()` method is run asynchronously, and is therefore not as suitable for situations where the progress must be reported in order.

Section 31.2: Using IProgress<T>

It's important to note that the `System.Progress<T>` class does not have the `Report()` method available on it. This method was implemented explicitly from the `IProgress<T>` interface, and therefore must be called on a `Progress<T>` when it's cast to an `IProgress<T>`.

```
var p1 = new Progress<int>();
p1.Report(1); //compiler error, Progress does not contain method 'Report'

IProgress<int> p2 = new Progress<int>();
p2.Report(2); //works
```

```
var p3 = new Progress<int>();
((IProgress<int>)p3).Report(3); //有效
```



```
var p3 = new Progress<int>();
((IProgress<int>)p3).Report(3); //works
```

第32章：JSON序列化

第32.1节：使用 System.Web.Script.Serialization.JavaScriptSerializer进行反序列化

JavaScriptSerializer.Deserialize<T>(input) 方法尝试将一段有效的JSON字符串反序列化为指定类型 <T> 的对象，使用 JavaScriptSerializer 原生支持的默认映射。

```
using System.Collections;
using System.Web.Script.Serialization;

// ...

string rawJSON = "{\"Name\":\"斐波那契数列\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";

JavaScriptSerializer JSS = new JavaScriptSerializer();
Dictionary<string, object> parsedObj = JSS.Deserialize<Dictionary<string, object>>(rawJSON);

string name = parsedObj["Name"].toString();
ArrayList numbers = (ArrayList)parsedObj["Numbers"]
```

注意：JavaScriptSerializer 对象是在 .NET 3.5 版本中引入的

第32.2节：使用 Json.NET 进行序列化

```
[JsonObject("person")]
public class Person
{
    [JsonProperty("name")]
    public string PersonName { get; set; }
    [JsonProperty("age")]
    public int PersonAge { get; set; }
    [JsonIgnore]
    public string Address { get; set; }
}

Person person = new Person { PersonName = "Andrius", PersonAge = 99, Address = "某地址" };
string rawJson = JsonConvert.SerializeObject(person);

Console.WriteLine(rawJson); // {"name":"Andrius","age":99}
```

注意属性（和类）如何通过特性进行装饰，以改变它们在生成的json字符串中的表现，或者完全将它们从json字符串中移除（JsonIgnore）。

关于Json.NET序列化特性的更多信息可以在这里找到。

在C#中，公共标识符按惯例使用PascalCase书写。在JSON中，惯例是对所有名称使用camelCase。你可以使用契约解析器在两者之间转换。

```
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    [JsonIgnore]
```


Chapter 32: JSON Serialization

Section 32.1: Deserialization using System.Web.Script.Serialization.JavaScriptSerializer

The `JavaScriptSerializer.Deserialize<T>(input)` method attempts to deserialize a string of valid JSON into an object of the specified type **`<T>`**, using the default mappings natively supported by `JavaScriptSerializer`.

```
using System.Collections;
using System.Web.Script.Serialization;

// ...

string rawJSON = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";

JavaScriptSerializer JSS = new JavaScriptSerializer();
Dictionary<string, object> parsedObj = JSS.Deserialize<Dictionary<string, object>>(rawJSON);

string name = parsedObj["Name"].toString();
ArrayList numbers = (ArrayList)parsedObj["Numbers"]
```

Note: The `JavaScriptSerializer` object was introduced in .NET version 3.5

Section 32.2: Serialization using Json.NET

```
[JsonObject("person")]
public class Person
{
    [JsonProperty("name")]
    public string PersonName { get; set; }
    [JsonProperty("age")]
    public int PersonAge { get; set; }
    [JsonIgnore]
    public string Address { get; set; }
}

Person person = new Person { PersonName = "Andrius", PersonAge = 99, Address = "Some address" };
string rawJson = JsonConvert.SerializeObject(person);

Console.WriteLine(rawJson); // {"name":"Andrius","age":99}
```

Notice how properties (and classes) can be decorated with attributes to change their appearance in resulting json string or to remove them from json string at all (JsonIgnore).

More information about Json.NET serialization attributes can be found here.

In C#, public identifiers are written in *PascalCase* by convention. In JSON, the convention is to use *camelCase* for all names. You can use a contract resolver to convert between the two.

```
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    [JsonIgnore]
```

```
    public string Address { get; set; }
}

public void ToJson() {
Person person = new Person { Name = "Andrius", Age = 99, Address = "Some address" };
    var resolver = new CamelCasePropertyNamesContractResolver();
    var settings = new JsonSerializerSettings { ContractResolver = resolver };
    string json = JsonConvert.SerializeObject(person, settings);

Console.WriteLine(json); // {"name":"Andrius","age":99}
}
```

第32.3节：使用

Newtonsoft.Json进行序列化-反序列化

与其他辅助工具不同，这个使用静态类辅助工具进行序列化和反序列化，因此比其他工具更容易使用。

```
using Newtonsoft.Json;

var rawJSON      = "{\"Name\":\"斐波那契数列\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";
var fibo         = JsonConvert.DeserializeObject<Dictionary<string, object>>(rawJSON);
var rawJSON2     = JsonConvert.SerializeObject(fibo);
```

第32.4节：使用Json.NET进行反序列化

```
internal class Sequence{
    public string Name;
    public List<int> Numbers;
}

// ...

string rawJSON = "{\"Name\":\"斐波那契数列\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";

Sequence sequence = JsonConvert.DeserializeObject<Sequence>(rawJSON);
```

更多信息，请参阅Json.NET官方网站。

注意：Json.NET支持.NET 2及更高版本。

第32.5节：动态绑定

Newtonsoft 的 Json.NET 允许您动态绑定 JSON（使用 ExpandoObject / 动态对象），无需显式创建类型。

序列化

```
dynamic jsonObject = new ExpandoObject();
jsonObject.Title    = "威尼斯商人";
jsonObject.Author   = "威廉·莎士比亚";
Console.WriteLine(JsonConvert.SerializeObject(jsonObject));
```

反序列化

```
var rawJson = "{\"Name\":\"斐波那契数列\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";
dynamic parsedJson = JObject.Parse(rawJson);
```



```
    public string Address { get; set; }
}

public void ToJson() {
    Person person = new Person { Name = "Andrius", Age = 99, Address = "Some address" };
    var resolver = new CamelCasePropertyNamesContractResolver();
    var settings = new JsonSerializerSettings { ContractResolver = resolver };
    string json = JsonConvert.SerializeObject(person, settings);

    Console.WriteLine(json); // {"name":"Andrius","age":99}
}
```

Section 32.3: Serialization-Deserialization using Newtonsoft.Json

Unlike the other helpers, this one uses static class helpers to serialize and deserialize, hence it is a little bit easier than the others to use.

```
using Newtonsoft.Json;

var rawJSON      = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";
var fibo         = JsonConvert.DeserializeObject<Dictionary<string, object>>(rawJSON);
var rawJSON2     = JsonConvert.SerializeObject(fibo);
```

Section 32.4: Deserialization using Json.NET

```
internal class Sequence{
    public string Name;
    public List<int> Numbers;
}

// ...

string rawJSON = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";

Sequence sequence = JsonConvert.DeserializeObject<Sequence>(rawJSON);
```

For more information, refer to the Json.NET official site.

Note: Json.NET supports .NET version 2 and higher.

Section 32.5: Dynamic binding

Newtonsoft's Json.NET allows you to bind json dynamically (using ExpandoObject / Dynamic objects) without the need to create the type explicitly.

Serialization

```
dynamic jsonObject = new ExpandoObject();
jsonObject.Title    = "Merchent of Venice";
jsonObject.Author   = "William Shakespeare";
Console.WriteLine(JsonConvert.SerializeObject(jsonObject));
```

De-serialization

```
var rawJson = "{\"Name\":\"Fibonacci Sequence\",\"Numbers\":[0, 1, 1, 2, 3, 5, 8, 13]}";
dynamic parsedJson = JObject.Parse(rawJson);
```

```
Console.WriteLine("名称: " + parsedJson.Name);
Console.WriteLine("数量: " + parsedJson.Numbers.Length);
```

注意，rawJson 对象中的键已被转换为动态对象中的成员变量。

这在应用程序可以接受/生成不同格式的 JSON 时非常有用。不过建议对 JSON 字符串或序列化/反序列化生成的动态对象使用额外的验证层。

第32.6节：使用 Json.NET 和 JsonSerializerSettings 进行序列化

该序列化器具有一些默认 .NET JSON 序列化器没有的优点，比如对空值的处理，你只需创建 JsonSerializerSettings ：

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings { NullValueHandling
= NullValueHandling.Ignore});
   return result;
}
```

.net 中另一个严重的序列化问题是自引用循环。以一个注册了课程的学生为例，其实例有一个课程属性，而课程有一个学生集合，即一个 List<Student>，这将创建一个引用循环。你可以通过 JsonSerializerSettings 来处理这个问题：

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
ReferenceLoopHandling = ReferenceLoopHandling.Ignore});
   return result;
}
```

你可以像这样设置各种序列化选项：

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings { NullValueHandling
= NullValueHandling.Ignore, ReferenceLoopHandling = ReferenceLoopHandling.Ignore});
   return result;
}
```



```
Console.WriteLine("Name: " + parsedJson.Name);
Console.WriteLine("Name: " + parsedJson.Numbers.Length);
```

Notice that the keys in the rawJson object have been turned into member variables in the dynamic object.

This is useful in cases where an application can accept/ produce varying formats of JSON. It is however suggested to use an extra level of validation for the Json string or to the dynamic object generated as a result of serialization/ de-serialization.

Section 32.6: Serialization using Json.NET with JsonSerializerSettings

This serializer has some nice features that the default .net json serializer doesn't have, like Null value handling, you just need to create the `JsonSerializerSettings` :

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings { NullValueHandling
= NullValueHandling.Ignore});
   return result;
}
```

Another serious serializer issue in .net is the self referencing loop. In the case of a student that is enrolled in a course, its instance has a course property and a course has a collection of students that means a `List<Student>` which will create a reference loop. You can handle this with `JsonSerializerSettings` :

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings {
ReferenceLoopHandling = ReferenceLoopHandling.Ignore});
   return result;
}
```

You can put various serializations option like this:

```
public static string Serialize(T obj)
{
   string result = JsonConvert.SerializeObject(obj, new JsonSerializerSettings { NullValueHandling
= NullValueHandling.Ignore, ReferenceLoopHandling = ReferenceLoopHandling.Ignore});
   return result;
}
```

第33章：.NET 中使用 Newtonsoft.Json 处理 JSON

NuGet 包 Newtonsoft.Json 已成为在 .NET 中使用和操作 JSON 格式文本和对象的事实标准。它是一个快速且易于使用的强大工具。

第33.1节：从 JSON 文本反序列化对象

```
var json = "{\"Name\":\"Joe Smith\",\"Age\":21}";
var person = JsonConvert.DeserializeObject<Person>(json);
```

这将生成一个 Name 为 "Joe Smith"、Age 为 21 的 Person 对象。

第33.2节：将对象序列化为JSON

```
using Newtonsoft.Json;

var obj = new Person
{
Name = "Joe Smith",
    Age = 21
};
var serializedJson = JsonConvert.SerializeObject(obj);
```

这将生成如下 JSON: {"Name":"Joe Smith","Age":21}

Chapter 33: JSON in .NET with Newtonsoft.Json

The NuGet package Newtonsoft.Json has become the defacto standard for using and manipulating JSON formatted text and objects in .NET. It is a robust tool that is fast, and easy to use.

Section 33.1: Deserialize an object from JSON text

```
var json = "{\"Name\":\"Joe Smith\",\"Age\":21}";
var person = JsonConvert.DeserializeObject<Person>(json);
```

This yields a Person object with Name "Joe Smith" and Age 21.

Section 33.2: Serialize object into JSON

```
using Newtonsoft.Json;

var obj = new Person
{
    Name = "Joe Smith",
    Age = 21
};
var serializedJson = JsonConvert.SerializeObject(obj);
```

This results in this JSON: {"Name":"Joe Smith","Age":21}

第34章：XmlSerializer

第34.1节：格式化：自定义日期时间格式

```
public class Dog
{
    private const string _birthStringFormat = "yyyy-MM-dd";

    [XmlIgnore]
    public DateTime Birth {get; set;}

    [XmlElement(ElementName="Birth")]
    public string BirthString
    {
        get { return Birth.ToString(_birthStringFormat); }
        set { Birth = DateTime.ParseExact(value, _birthStringFormat, CultureInfo.InvariantCulture);
}
    }
}
```

第34.2节：序列化对象

```
public void SerializeFoo(string fileName, Foo foo)
{
    var serializer = new XmlSerializer(typeof(Foo));
    using (var stream = File.Open(fileName, FileMode.Create))
    {
serializer.Serialize(stream, foo);
    }
}
```

第34.3节：反序列化对象

```
public Foo DeserializeFoo(string fileName)
{
    var serializer = new XmlSerializer(typeof(Foo));
    using (var stream = File.OpenRead(fileName))
    {
        return (Foo)serializer.Deserialize(stream);
    }
}
```

第34.4节：行为：将数组名称映射到属性 (XmlArray)

```
<Store>
    <Articles>
        <Product/>
        <Product/>
    </Articles>
</Store>
```

```
public class Store
{
    [XmlArray("Articles")]
```


Chapter 34: XmlSerializer

Section 34.1: Formatting: Custom DateTime format

```
public class Dog
{
    private const string _birthStringFormat = "yyyy-MM-dd";

    [XmlIgnore]
    public DateTime Birth {get; set;}

    [XmlElement(ElementName="Birth")]
    public string BirthString
    {
        get { return Birth.ToString(_birthStringFormat); }
        set { Birth = DateTime.ParseExact(value, _birthStringFormat, CultureInfo.InvariantCulture);
}
    }
}
```

Section 34.2: Serialize object

```
public void SerializeFoo(string fileName, Foo foo)
{
    var serializer = new XmlSerializer(typeof(Foo));
    using (var stream = File.Open(fileName, FileMode.Create))
    {
        serializer.Serialize(stream, foo);
    }
}
```

Section 34.3: Deserialize object

```
public Foo DeserializeFoo(string fileName)
{
    var serializer = new XmlSerializer(typeof(Foo));
    using (var stream = File.OpenRead(fileName))
    {
        return (Foo)serializer.Deserialize(stream);
    }
}
```

Section 34.4: Behaviour: Map array name to property (XmlArray)

```
<Store>
    <Articles>
        <Product/>
        <Product/>
    </Articles>
</Store>
```

```
public class Store
{
    [XmlArray("Articles")]
```

```
    public List<Product> Products {get; set; }
}
```

第34.5节：行为：将元素名称映射到属性

```
<Foo>
    <Dog/>
</Foo>
```

```
public class Foo
{
    // 使用 XmlElement
    [XmlElement(Name="Dog")]
    public Animal Cat { get; set; }
}
```

第34.6节：高效构建多个序列化器，动态指定派生类型

我们的起源

有时我们无法在属性中提供 XmlSerializer 框架所需的所有元数据。
假设我们有一个序列化对象的基类，而一些派生类对基类来说是未知的。
我们无法为所有在基类设计时未知的类添加属性。可能还有另一个团队在开发部分派生类。

我们该怎么办

我们可以使用newXmlSerializer(type, knownTypes)，但对于 N 个序列化器来说，这至少是一个 O(N^2) 的操作，用于发现参数中提供的所有类型：

```
// 注意类型数量方面的 N^2 问题。
var allSerializers = allTypes.Select(t => new XmlSerializer(t, allTypes));
var serializerDictionary = Enumerable.Range(0, allTypes.Length)
    .ToDictionary (i => allTypes[i], i => allSerializers[i])
```

在此示例中，基类不知道其派生类，这在面向对象编程中是正常的。

高效实现方法

幸运的是，有一种方法可以有效地为多个序列化器提供已知类型，从而解决这个特定问题：

System.Xml.Serialization.XmlSerializer.FromTypes 方法 (Type[])

> FromTypes 方法允许您高效地为一组 Type 对象创建 XmlSerializer 对象数组以进行处理。

```
var allSerializers = XmlSerializer.FromTypes(allTypes);
var serializerDictionary = Enumerable.Range(0, allTypes.Length)
    .ToDictionary(i => allTypes[i], i => allSerializers[i]);
```

以下是完整的代码示例：


```
    public List<Product> Products {get; set; }
}
```


Section 34.5: Behaviour: Map Element name to Property


```
<Foo>
    <Dog/>
</Foo>
```

```
public class Foo
{
    // Using XmlElement
    [XmlElement(Name="Dog")]
    public Animal Cat { get; set; }
}
```


Section 34.6: Efficiently building multiple serializers with derived types specified dynamically

Where we came from

Sometimes we can't provide all of the required metadata needed for the XmlSerializer framework in attribute. Suppose we have a base class of serialized objects, and some of the derived classes are unknown to the base class. We can't place an attribute for all of the classes which are not know at the design time of the base type. We could have another team developing some of the derived classes.

What can we do

We can use `new XmlSerializer(type, knownTypes)`, but that would be a O(N^2) operation for N serializers, at least to discover all of the types supplied in arguments:

```
// Beware of the N^2 in terms of the number of types.
var allSerializers = allTypes.Select(t => new XmlSerializer(t, allTypes));
var serializerDictionary = Enumerable.Range(0, allTypes.Length)
    .ToDictionary (i => allTypes[i], i => allSerializers[i])
```

In this example, the the Base type is not aware of it's derived types, which is normal in OOP.

Doing it efficiently

Luckily, there is a method which addresses this particular problem - supplying known types for multiple serializers efficiently:

System.Xml.Serialization.XmlSerializer.FromTypes Method (Type[])

> The FromTypes method allows you to efficiently create an array of XmlSerializer objects for processing an array of Type objects.

```
var allSerializers = XmlSerializer.FromTypes(allTypes);
var serializerDictionary = Enumerable.Range(0, allTypes.Length)
    .ToDictionary(i => allTypes[i], i => allSerializers[i]);
```

Here is a complete code sample:

```
using System;
using System.Collections.Generic;
using System.Xml.Serialization;
using System.Linq;
using System.Linq;

public class Program
{
    public class Container
    {
        public Base Base { get; set; }
    }

    public class Base
    {
        public int JustSomePropInBase { get; set; }
    }

    public class Derived : Base
    {
        public int JustSomePropInDerived { get; set; }
    }

    public void Main()
    {
        var sampleObject = new Container { Base = new Derived() };
        var allTypes = new[] { typeof(Container), typeof(Base), typeof(Derived) };

Console.WriteLine("尝试在没有派生类元数据的情况下序列化:");
        SetupSerializers(allTypes.Except(new[] { typeof(Derived) }).ToArray());
         try
         {
Serialize(sampleObject);
         }
         catch (InvalidOperationException e)
         {
Console.WriteLine();
Console.WriteLine("这个错误是预料之中的,");
             Console.WriteLine("我们没有提供派生类。");
           Console.WriteLine(e);
         }
Console.WriteLine("现在尝试使用所有类型信息进行序列化:");
        SetupSerializers(allTypes);
序列化(sampleObject);
         Console.WriteLine();
Console.WriteLine("这次滑动得很顺畅！");
    }

    静态 void 序列化<T>(T o)
    {
serializerDictionary[typeof(T)].Serialize(Console.Out, o);
    }

    私有静态 Dictionary<Type, XmlSerializer> serializerDictionary;

    静态 void 设置序列化器(Type[] allTypes)
    {
        var allSerializers = XmlSerializer.FromTypes(allTypes);
        serializerDictionary = Enumerable.Range(0, allTypes.Length)
            .ToDictionary(i => allTypes[i], i => allSerializers[\⊠ti]);
    }
```



```
using System;
using System.Collections.Generic;
using System.Xml.Serialization;
using System.Linq;
using System.Linq;

public class Program
{
    public class Container
    {
        public Base Base { get; set; }
    }

    public class Base
    {
        public int JustSomePropInBase { get; set; }
    }

    public class Derived : Base
    {
        public int JustSomePropInDerived { get; set; }
    }

    public void Main()
    {
        var sampleObject = new Container { Base = new Derived() };
        var allTypes = new[] { typeof(Container), typeof(Base), typeof(Derived) };

        Console.WriteLine("Trying to serialize without a derived class metadata:");
        SetupSerializers(allTypes.Except(new[] { typeof(Derived) }).ToArray());
        try
        {
            Serialize(sampleObject);
        }
        catch (InvalidOperationException e)
        {
            Console.WriteLine();
            Console.WriteLine("This error was anticipated,");
            Console.WriteLine("we have not supplied a derived class.");
            Console.WriteLine(e);
        }
        Console.WriteLine("Now trying to serialize with all of the type information:");
        SetupSerializers(allTypes);
        Serialize(sampleObject);
        Console.WriteLine();
        Console.WriteLine("Slides down well this time!");
    }

    static void Serialize<T>(T o)
    {
        serializerDictionary[typeof(T)].Serialize(Console.Out, o);
    }

    private static Dictionary<Type, XmlSerializer> serializerDictionary;

    static void SetupSerializers(Type[] allTypes)
    {
        var allSerializers = XmlSerializer.FromTypes(allTypes);
        serializerDictionary = Enumerable.Range(0, allTypes.Length)
            .ToDictionary(i => allTypes[i], i => allSerializers[i]);
    }
```

```
}
```

输出：

```
尝试在没有派生类元数据的情况下序列化：
<?xml version="1.0" encoding="utf-16"?>
System.InvalidOperationException: 未预期类型 Program+Derived。请使用 XmlInclude
或 SoapInclude 属性来指定静态未知的类型。
在 Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write2_Base(String n,
String ns, Base o, Boolean isNullable, Boolean needType) 处
在 Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write3_Container(String
n, String ns, Container o, Boolean isNullable, Boolean needType) 处
在 Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write4_Container(Object
o) 处
在 System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle, String id) 处
--- 内部异常堆栈跟踪结束 ---
在 System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle, String id) 处
在 System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle) 处
在 System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces) 处
在 Program.Serialize[T](T o)
在 Program.Main() 处
现在尝试使用所有类型信息进行序列化：
<?xml version="1.0" encoding="utf-16"?>
```



```
}
```

Output:

```
Trying to serialize without a derived class metadata:
<?xml version="1.0" encoding="utf-16"?>
 System.InvalidOperationException: The type Program+Derived was not expected. Use the XmlInclude
or SoapInclude attribute to specify types that are not known statically.
at Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write2_Base(String n,
String ns, Base o, Boolean isNullable, Boolean needType)
at Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write3_Container(String
n, String ns, Container o, Boolean isNullable, Boolean needType)
at Microsoft.Xml.Serialization.GeneratedAssembly.XmlSerializationWriter1.Write4_Container(Object
o)
at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle, String id)
--- End of inner exception stack trace ---
at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle, String id)
at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces, String encodingStyle)
at System.Xml.Serialization.XmlSerializer.Serialize(XmlWriter xmlWriter, Object o,
XmlSerializerNamespaces namespaces)
at Program.Serialize[T](T o)
at Program.Main()
Now trying to serialize with all of the type information:
<?xml version="1.0" encoding="utf-16"?>


0
0


Slides down well this time!
```

What's in the output

This error message recommends what we tried to avoid (or what we can not do in some scenarios) - referencing derived types from base class:

`Use the XmlInclude or SoapInclude attribute to specify types that are not known statically.`

This is how we get our derived class in the XML:

```
<Base xsi:type="Derived">
```

`Base` corresponds to the property type declared in the `Container` type, and `Derived` being the type of the instance actually supplied.

Here is a working example fiddle

第35章：VB窗体

第35.1节：VB.NET窗体中的Hello World

在窗体显示后显示消息框：

```
公共类 Form1
    私有子程序 Form1_Shown(sender 作为 对象, e 作为 事件参数) 处理 MyBase.已显示
        MessageBox.Show("Hello, World!")
    结束子程序
结束类
在窗体显示之前显示消息框：

公共类 Form1
    私有子程序 Form1_Load(sender 作为 对象, e 作为 事件参数) 处理 MyBase.加载
        MessageBox.Show("Hello, World!")
    结束子程序
结束类
```

Load() 会在窗体首次加载时首先且仅调用一次。Show() 会在用户每次启动窗体时调用。Activate() 会在用户每次激活窗体时调用。

Load() 会在调用 Show() 之前执行，但请注意：在 Show 中调用 msgBox() 可能会导致该 msgBox() 在 Load() 完成之前执行。通常不建议依赖 Load()、Show() 及类似事件的顺序。

第35.2节：初学者指南

所有初学者都应该知道/做的一些事情，这将帮助他们顺利开始使用VB .Net：

设置以下选项：

```
可以永久设置
工具 / 选项 / 项目和解决方案 / VB 默认值
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

End Class
```

使用 &，而不是 + 进行字符串连接。字符串 应该详细学习，因为它们被广泛使用。

花些时间理解值类型和值引用类型。

切勿使用Application.DoEvents。注意“注意事项”。当你遇到似乎必须使用它的情况时，请先询问。

文档是你的朋友。

第35.3节：定时器表单

Windows.Forms.Timer 组件可用于向用户提供非时间关键的信息。创建一个包含一个按钮、一个标签和一个 Timer 组件的窗体。

Chapter 35: VB Forms

Section 35.1: Hello World in VB.NET Forms

To show a message box when the form has been shown:

```
Public Class Form1
    Private Sub Form1_Shown(sender As Object, e As EventArgs) Handles MyBase.Shown
        MessageBox.Show("Hello, World!")
    End Sub
End Class
To show a message box before the form has been shown:

Public Class Form1
    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles MyBase.Load
        MessageBox.Show("Hello, World!")
    End Sub
End Class
```

Load() will be called first, and only once, when the form first loads. Show() will be called every time the user launches the form. Activate() will be called every time the user makes the form active.

Load() will execute before Show() is called, but be warned: calling msgBox() in show can cause that msgBox() to execute before Load() is finished. **It is generally a bad idea to depend on event ordering between Load(), Show(), and similar.**

Section 35.2: For Beginners

Some things all beginners should know / do that will help them have a good start with VB .Net:

Set the following Options:

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

End Class
```

Use &, not + for string concatenation. Strings should be studied in some detail as they are widely used.

Spend some time understanding Value and Reference Types.

Never use Application.DoEvents. Pay attention to the 'Caution'. When you reach a point where this seems like something you must use, ask.

The documentation is your friend.

Section 35.3: Forms Timer

The Windows.Forms.Timer component can be used to provide the user information that is **not** time critical. Create a form with one button, one label, and a Timer component.

例如，它可以用来定期向用户显示当前时间。

```
可以永久设置
工具 / 选项 / 项目和解决方案 / VB 默认值
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
Timer1.Interval = 60 * 1000 '一分钟间隔
        '启动计时器
        Timer1.Start()
        Label1.Text = DateTime.Now.ToLongTimeString
    结束子程序

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
       Label1.Text = DateTime.Now.ToLongTimeString
    End Sub
结束类
```

但这个计时器不适合用于计时。一个例子是用它来做倒计时。在这个例子中，我们将模拟一个三分钟的倒计时。这可能是这里最无聊但又非常重要的例子之一。

```
可以永久设置
工具 / 选项 / 项目和解决方案 / VB 默认值
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
ctSecs = 0 '清零计数
Timer1.Interval = 1000 '一秒钟，单位毫秒
        '启动计时器
stpw.Reset()
        stpw.Start()
        Timer1.Start()
    结束子程序

    Dim stpw As New Stopwatch
    Dim ctSecs As Integer

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        ctSecs += 1
        If ctSecs = 180 Then '我的电脑大约慢了2.5秒！
            '停止计时
stpw.Stop()
            Timer1.Stop()
            '显示实际经过时间
            '是否接近180？
Label1.Text = stpw.Elapsed.TotalSeconds.ToString("n1")
        End If
    End Sub
End Class
```


For example it could be used to show the user the time of day periodically.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        Timer1.Interval = 60 * 1000 'one minute intervals
        'start timer
        Timer1.Start()
        Label1.Text = DateTime.Now.ToLongTimeString
    End Sub

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        Label1.Text = DateTime.Now.ToLongTimeString
    End Sub
End Class
```

But this timer is not suited for timing. An example would be using it for a countdown. In this example we will simulate a countdown to three minutes. This may very well be one of the most boringly important examples here.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        ctSecs = 0 'clear count
        Timer1.Interval = 1000 'one second in ms.
        'start timers
        stpw.Reset()
        stpw.Start()
        Timer1.Start()
    End Sub

    Dim stpw As New Stopwatch
    Dim ctSecs As Integer

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        ctSecs += 1
        If ctSecs = 180 Then 'about 2.5 seconds off on my PC!
            'stop timing
            stpw.Stop()
            Timer1.Stop()
            'show actual elapsed time
            'Is it near 180?
            Label1.Text = stpw.Elapsed.TotalSeconds.ToString("n1")
        End If
    End Sub
End Class
```

点击 button1 后，大约三分钟过去，label1 显示结果。label1 会显示 180 吗？大概不会。在我的机器上显示的是 182.5！

差异的原因在文档中，“Windows 窗体计时器组件是单线程的，
且精度限制为 55 毫秒。”这就是为什么它不应该用于计时的原因。

通过稍微不同地使用计时器和秒表，我们可以获得更好的结果。

```
可以永久设置
工具 / 选项 / 项目和解决方案 / VB 默认值
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
Timer1.Interval = 100 '一秒的十分之一，单位为毫秒。
        '启动计时器
stpw.Reset()
        stpw.Start()
        Timer1.Start()
    结束子程序

    Dim stpw As New Stopwatch
    Dim threeMinutes As TimeSpan = TimeSpan.FromMinutes(3)

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        If stpw.Elapsed >= threeMinutes Then '我的电脑误差为0.1！
            '停止计时
stpw.Stop()
            Timer1.Stop()
            '显示实际经过时间
            '误差有多大？
Label1.Text = stpw.Elapsed.TotalSeconds.ToString("n1")
        End If
    End Sub
End Class
```

还有其他计时器可以根据需要使用。这个搜索应该对此有所帮助。

After button1 is clicked, about three minutes pass and label1 shows the results. Does label1 show 180? Probably not. On my machine it showed 182.5!

The reason for the discrepancy is in the documentation, "The Windows Forms Timer component is single-threaded, and is limited to an accuracy of 55 milliseconds." This is why it shouldn't be used for timing.

By using the timer and stopwatch a little differently we can obtain better results.

```
'can be permanently set
' Tools / Options / Projects and Soluntions / VB Defaults
Option Strict On
Option Explicit On
Option Infer Off

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
        Button1.Enabled = False
        Timer1.Interval = 100 'one tenth of a second in ms.
        'start timers
        stpw.Reset()
        stpw.Start()
        Timer1.Start()
    End Sub

    Dim stpw As New Stopwatch
    Dim threeMinutes As TimeSpan = TimeSpan.FromMinutes(3)

    Private Sub Timer1_Tick(sender As Object, e As EventArgs) Handles Timer1.Tick
        If stpw.Elapsed >= threeMinutes Then '0.1 off on my PC!
            'stop timing
            stpw.Stop()
            Timer1.Stop()
            'show actual elapsed time
            'how close?
            Label1.Text = stpw.Elapsed.TotalSeconds.ToString("n1")
        End If
    End Sub
End Class
```

There are other timers that can be used as needed. This search should help in that regard.

第36章：JIT 编译器

JIT 编译，或称即时编译，是代码解释或提前编译的另一种方法。JIT 编译用于 .NET 框架。CLR 代码（C#、F#、Visual Basic 等）首先被编译成称为中间语言（IL）的代码。这是一种较低级别的代码，更接近机器码，但不是特定平台的代码。相反，在运行时，这段代码会被编译成对应系统的机器码。

第36.1节：IL 编译示例

简单的 Hello World 应用程序：

```
using System;

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

等效的 IL 代码（将被 JIT 编译）

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 4.6.1055.0
//  Copyright (c) Microsoft Corporation.  All rights reserved.



// Metadata version: v4.0.30319
.assembly extern mscorlib
{
.publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .z\V.4..
.ver 4:0:0:0
}
.assembly HelloWorld
{
.custom instance void
[mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.ctor(int32) = ( 01 00
08 00 00 00 00 00 )
.custom instance void
[mscorlib]System.Runtime.CompilerServices.RuntimeCompatibilityAttribute::.ctor() = ( 01 00 01 00
54 02 16 57 72 61 70 4E 6F 6E 45 78   // ....T..WrapNonEx
63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 )       // ceptionThrows.

// --- 以下自定义属性自动添加，请勿取消注释 -------
//  .custom instance void [mscorlib]System.Diagnostics.DebuggableAttribute::.ctor(valuetype
[mscorlib]System.Diagnostics.DebuggableAttribute/DebuggingModes) = ( 01 00 07 01 00 00 00 00 )

.custom instance void [mscorlib]System.Reflection.AssemblyTitleAttribute::.ctor(string) = ( 01 00
0A 48 65 6C 6C 6F 57 6F 72 6C 64 00 00 )    // ...HelloWorld..
.custom instance void [mscorlib]System.Reflection.AssemblyDescriptionAttribute::.ctor(string) = (
01 00 00 00 00 )
.custom instance void [mscorlib]System.Reflection.AssemblyConfigurationAttribute::.ctor(string) =
( 01 00 00 00 00 )
.custom instance void [mscorlib]System.Reflection.AssemblyCompanyAttribute::.ctor(string) = ( 01
00 00 00 00 )
```


Chapter 36: JIT compiler

JIT compilation, or just-in-time compilation, is an alternative approach to interpretation of code or ahead-of-time compilation. JIT compilation is used in the .NET framework. The CLR code (C#, F#, Visual Basic, etc.) is first compiled into something called Interpreted Language, or IL. This is lower level code that is closer to machine code, but is not platform specific. Rather, at runtime, this code is compiled into machine code for the relevant system.

Section 36.1: IL compilation sample

Simple Hello World Application:

```
using System;

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

Equivalent IL Code (which will be JIT compiled)

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 4.6.1055.0
//  Copyright (c) Microsoft Corporation.  All rights reserved.



// Metadata version: v4.0.30319
.assembly extern mscorlib
{
.publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .z\V.4..
.ver 4:0:0:0
}
.assembly HelloWorld
{
.custom instance void
[mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.ctor(int32) = ( 01 00
08 00 00 00 00 00 )
.custom instance void
[mscorlib]System.Runtime.CompilerServices.RuntimeCompatibilityAttribute::.ctor() = ( 01 00 01 00
54 02 16 57 72 61 70 4E 6F 6E 45 78   // ....T..WrapNonEx
63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 )       // ceptionThrows.

// --- The following custom attribute is added automatically, do not uncomment -------
//  .custom instance void [mscorlib]System.Diagnostics.DebuggableAttribute::.ctor(valuetype
[mscorlib]System.Diagnostics.DebuggableAttribute/DebuggingModes) = ( 01 00 07 01 00 00 00 00 )

.custom instance void [mscorlib]System.Reflection.AssemblyTitleAttribute::.ctor(string) = ( 01 00
0A 48 65 6C 6C 6F 57 6F 72 6C 64 00 00 )    // ...HelloWorld..
.custom instance void [mscorlib]System.Reflection.AssemblyDescriptionAttribute::.ctor(string) = (
01 00 00 00 00 )
.custom instance void [mscorlib]System.Reflection.AssemblyConfigurationAttribute::.ctor(string) =
( 01 00 00 00 00 )
.custom instance void [mscorlib]System.Reflection.AssemblyCompanyAttribute::.ctor(string) = ( 01
00 00 00 00 )
```

```
.custom instance void [mscorlib]System.Reflection.AssemblyProductAttribute::.ctor(string) = ( 01
00 0A 48 65 6C 6C 6F 57 6F 72 6C 64 00 00 )     // ...HelloWorld..
.custom instance void [mscorlib]System.Reflection.AssemblyCopyrightAttribute::.ctor(string) = ( 01
00 12 43 6F 70 79 72 69 67 68 74 20 C2 A9 20    // ...版权 ..
20 32 30 31 37 00 00 )                               //  2017..
.custom instance void [mscorlib]System.Reflection.AssemblyTrademarkAttribute::.ctor(string) = ( 01
00 00 00 00 )
.custom instance void [mscorlib]System.Runtime.InteropServices.ComVisibleAttribute::.ctor(bool) =
( 01 00 00 00 00 )
.custom instance void [mscorlib]System.Runtime.InteropServices.GuidAttribute::.ctor(string) = ( 01
00 24 33 30 38 62 33 64 38 36 2D 34 31 37 32    // ..$308b3d86-4172
2D 34 30 32 32 2D 61 66 63 63 2D 33 66 38 65 33    // -4022-afcc-3f8e3
32 33 33 63 35 62 30 00 00 )                        // 233c5b0..
.custom instance void [mscorlib]System.Reflection.AssemblyFileVersionAttribute::.ctor(string) = (
01 00 07 31 2E 30 2E 30 2E 30 00 00 )               // ...1.0.0.0..
.custom instance void [mscorlib]System.Runtime.Versioning.TargetFrameworkAttribute::.ctor(string)
= ( 01 00 1C 2E 4E 45 54 46 72 61 6D 65 77 6F 72 6B    // ....NETFramework
2C 56 65 72 73 69 6F 6E 3D 76 34 2E 35 2E 32 01    // ,Version=v4.5.2.
00 54 0E 14 46 72 61 6D 65 77 6F 72 6B 44 69 73    // .T..FrameworkDis
70 6C 61 79 4E 61 6D 65 14 2E 4E 45 54 20 46 72    // playName..NET Fr
61 6D 65 77 6F 72 6B 20 34 2E 35 2E 32 )           // amework 4.5.2
.hash algorithm 0x00008004
.ver 1:0:0:0
}
.module HelloWorld.exe
// MVID: {2A7E1D59-1272-4B47-85F6-D7E1ED057831}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00020003    //  ILONLY 32BITPREFERRED
// Image base: 0x0000021C70230000


// =============== 类成员声明 ===================

.class private auto ansi beforefieldinit HelloWorld.Program
extends [mscorlib]System.Object
{
.method private hidebysig static void  Main(string[] args) cil managed
{
.entrypoint
// 代码大小       13 (0xd)
.maxstack  8
IL_0000:  nop
IL_0001:  ldstr      "Hello World"
IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
IL_000b:  nop
IL_000c:  ret
} // Program::Main 方法结束

.method public hidebysig specialname rtspecialname
instance void  .ctor() cil managed
{
// 代码大小       8 (0x8)
.maxstack  8
IL_0000:  ldarg.0
IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
IL_0006:  nop
IL_0007:  ret
} // Program::.ctor 方法结束

} // HelloWorld.Program 类结束
```



```
.custom instance void [mscorlib]System.Reflection.AssemblyProductAttribute::.ctor(string) = ( 01
00 0A 48 65 6C 6C 6F 57 6F 72 6C 64 00 00 )    // ...HelloWorld..
.custom instance void [mscorlib]System.Reflection.AssemblyCopyrightAttribute::.ctor(string) = ( 01
00 12 43 6F 70 79 72 69 67 68 74 20 C2 A9 20   // ...Copyright ..
20 32 30 31 37 00 00 )                              //  2017..
.custom instance void [mscorlib]System.Reflection.AssemblyTrademarkAttribute::.ctor(string) = ( 01
00 00 00 00 )
.custom instance void [mscorlib]System.Runtime.InteropServices.ComVisibleAttribute::.ctor(bool) =
( 01 00 00 00 00 )
.custom instance void [mscorlib]System.Runtime.InteropServices.GuidAttribute::.ctor(string) = ( 01
00 24 33 30 38 62 33 64 38 36 2D 34 31 37 32   // ..$308b3d86-4172
2D 34 30 32 32 2D 61 66 63 63 2D 33 66 38 65 33   // -4022-afcc-3f8e3
32 33 33 63 35 62 30 00 00 )                       // 233c5b0..
.custom instance void [mscorlib]System.Reflection.AssemblyFileVersionAttribute::.ctor(string) = (
01 00 07 31 2E 30 2E 30 2E 30 00 00 )              // ...1.0.0.0..
.custom instance void [mscorlib]System.Runtime.Versioning.TargetFrameworkAttribute::.ctor(string)
= ( 01 00 1C 2E 4E 45 54 46 72 61 6D 65 77 6F 72 6B   // ....NETFramework
2C 56 65 72 73 69 6F 6E 3D 76 34 2E 35 2E 32 01   // ,Version=v4.5.2.
00 54 0E 14 46 72 61 6D 65 77 6F 72 6B 44 69 73   // .T..FrameworkDis
70 6C 61 79 4E 61 6D 65 14 2E 4E 45 54 20 46 72   // playName..NET Fr
61 6D 65 77 6F 72 6B 20 34 2E 35 2E 32 )          // amework 4.5.2
.hash algorithm 0x00008004
.ver 1:0:0:0
}
.module HelloWorld.exe
// MVID: {2A7E1D59-1272-4B47-85F6-D7E1ED057831}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00020003    //  ILONLY 32BITPREFERRED
// Image base: 0x0000021C70230000


// =============== CLASS MEMBERS DECLARATION ===================

.class private auto ansi beforefieldinit HelloWorld.Program
extends [mscorlib]System.Object
{
.method private hidebysig static void  Main(string[] args) cil managed
{
.entrypoint
// Code size       13 (0xd)
.maxstack  8
IL_0000:  nop
IL_0001:  ldstr      "Hello World"
IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
IL_000b:  nop
IL_000c:  ret
} // end of method Program::Main

.method public hidebysig specialname rtspecialname
instance void  .ctor() cil managed
{
// Code size       8 (0x8)
.maxstack  8
IL_0000:  ldarg.0
IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
IL_0006:  nop
IL_0007:  ret
} // end of method Program::.ctor

} // end of class HelloWorld.Program
```

使用 MS ILDASM 工具（IL 反汇编器）生成

Generated with MS ILDASM tool (IL disassembler)

第37章：CLR

第37.1节：公共语言运行时简介

公共语言运行时（CLR）是一个虚拟机环境，是.NET框架的一部分。它包含：

- 一种可移植的字节码语言，称为公共中间语言（简称CIL或IL）
- 一个即时编译器，用于生成机器码
- 一个跟踪垃圾回收器，提供自动内存管理
- 支持称为应用域（AppDomains）的轻量级子进程
- 通过可验证代码和信任级别的概念实现的安全机制

在CLR中运行的代码称为托管代码，以区别于在CLR外运行的代码（通常是本机代码），后者称为非托管代码。存在多种机制促进托管代码与非托管代码之间的互操作性。

Chapter 37: CLR

Section 37.1: An introduction to Common Language Runtime

The **Common Language Runtime (CLR)** is a virtual machine environment and part of the .NET Framework. It contains:

- A portable bytecode language called **Common Intermediate Language** (abbreviated CIL, or IL)
- A Just-In-Time compiler that generates machine code
- A tracing garbage collector that provides automatic memory management
- Support for lightweight sub-processes called AppDomains
- Security mechanisms through the concepts of verifiable code and trust levels

Code that runs in the CLR is referred to as *managed code* to distinguish it from code running outside the CLR (usually native code) which is referred to as *unmanaged code*. There are various mechanisms that facilitate interoperability between managed and unmanaged code.

第38章：TPL数据流

第38.1节：带有有界缓冲区的异步生产者消费者

```
var bufferBlock = new BufferBlock<int>(new DataflowBlockOptions
{
BoundedCapacity = 1000
});

var cancellationToken = new CancellationTokenSource(TimeSpan.FromSeconds(10)).Token;

var producerTask = Task.Run(async () =>
{
    var random = new Random();

    while (!cancellationToken.IsCancellationRequested)
    {
        var value = random.Next();
        await bufferBlock.SendAsync(value, cancellationToken);
    }
});

var consumerTask = Task.Run(async () =>
{
    while (await bufferBlock.OutputAvailableAsync())
    {
        var value = bufferBlock.Receive();
        Console.WriteLine(value);
    }
});

await Task.WhenAll(producerTask, consumerTask);
```

第38.2节：向ActionBlock发布并等待完成

```
// 创建一个带有异步操作的块
var block = new ActionBlock<string>(async hostName =>
{
IPAddress[] ipAddresses = await Dns.GetHostAddressesAsync(hostName);
    Console.WriteLine(ipAddresses[0]);
});

block.Post("google.com"); // 将项目发布到块的InputQueue以进行处理
block.Post("reddit.com");
block.Post("stackoverflow.com");

block.Complete(); // 告诉块完成并停止接受新项目
await block.Completion; // 异步等待直到所有项目处理完成
```

第38.3节：链接块以创建管道

```
var httpClient = new HttpClient();

// 创建一个接受URI并返回其内容字符串的块
var downloaderBlock = new TransformBlock<string, string>(
```


Chapter 38: TPL Dataflow

Section 38.1: Asynchronous Producer Consumer With A Bounded BufferBlock

```
var bufferBlock = new BufferBlock<int>(new DataflowBlockOptions
{
    BoundedCapacity = 1000
});

var cancellationToken = new CancellationTokenSource(TimeSpan.FromSeconds(10)).Token;

var producerTask = Task.Run(async () =>
{
    var random = new Random();

    while (!cancellationToken.IsCancellationRequested)
    {
        var value = random.Next();
        await bufferBlock.SendAsync(value, cancellationToken);
    }
});

var consumerTask = Task.Run(async () =>
{
    while (await bufferBlock.OutputAvailableAsync())
    {
        var value = bufferBlock.Receive();
        Console.WriteLine(value);
    }
});

await Task.WhenAll(producerTask, consumerTask);
```

Section 38.2: Posting to an ActionBlock and waiting for completion

```
// Create a block with an asynchronous action
var block = new ActionBlock<string>(async hostName =>
{
    IPAddress[] ipAddresses = await Dns.GetHostAddressesAsync(hostName);
    Console.WriteLine(ipAddresses[0]);
});

block.Post("google.com"); // Post items to the block's InputQueue for processing
block.Post("reddit.com");
block.Post("stackoverflow.com");

block.Complete(); // Tell the block to complete and stop accepting new items
await block.Completion; // Asynchronously wait until all items completed processingu
```

Section 38.3: Linking blocks to create a pipeline

```
var httpClient = new HttpClient();

// Create a block the accepts a uri and returns its contents as a string
var downloaderBlock = new TransformBlock<string, string>(
```

```
    async uri => await httpClient.GetStringAsync(uri));

// 创建一个接受内容并将其打印到控制台的块
var printerBlock = new ActionBlock<string>(
    contents => Console.WriteLine(contents));

// 当downloaderBlock完成时，使其完成printerBlock
var dataflowLinkOptions = new DataflowLinkOptions {PropagateCompletion = true};

// 链接块以创建一个管道
downloaderBlock.LinkTo(printerBlock, dataflowLinkOptions);

// 向第一个块发布网址，该块将其内容传递给第二个块
downloaderBlock.Post("http://youtube.com");
downloaderBlock.Post("http://github.com");
downloaderBlock.Post("http://twitter.com");

downloaderBlock.Complete(); // 完成状态将传递到printerBlock
await printerBlock.Completion; // 只需等待管道中的最后一个块
```

第38.4节：带有缓冲区块（BufferBlock<T>）的同步生产者/消费者

```
public class 生产者
{
    private static Random random = new Random((int)DateTime.UtcNow.Ticks);
    // 生成将发布到缓冲区块的值
    public double 生成 ( )
    {
        var value = random.NextDouble();
       Console.WriteLine($"生成的值: {value}");
        return value;
    }
}

public class 消费者
{
    // 消费将从缓冲区块接收的值
    public void 消费 (double value) => Console.WriteLine($"消费的值: {value}");
}

class 程序
{
    private static BufferBlock<double> buffer = new BufferBlock<double>();
    static void Main (string[] args)
    {
                                //启动一个任务，每隔1秒从生产者向缓冲区块发布一个值
        var producerTask = Task.Run(async () =>
        {
            var producer = new Producer();
            while(true)
            {
buffer.Post(producer.Produce());
                await Task.Delay(1000);
            }
        });
        //启动一个任务，从缓冲区块接收值并进行消费
        var consumerTask = Task.Run(() =>
        {
            var consumer = new Consumer();
            while(true)
```



```
    async uri => await httpClient.GetStringAsync(uri));

// Create a block that accepts the content and prints it to the console
var printerBlock = new ActionBlock<string>(
    contents => Console.WriteLine(contents));

// Make the downloaderBlock complete the printerBlock when its completed.
var dataflowLinkOptions = new DataflowLinkOptions {PropagateCompletion = true};

// Link the block to create a pipeline
downloaderBlock.LinkTo(printerBlock, dataflowLinkOptions);

// Post urls to the first block which will pass their contents to the second one.
downloaderBlock.Post("http://youtube.com");
downloaderBlock.Post("http://github.com");
downloaderBlock.Post("http://twitter.com");

downloaderBlock.Complete(); // Completion will propagate to printerBlock
await printerBlock.Completion; // Only need to wait for the last block in the pipeline
```

Section 38.4: Synchronous Producer/Consumer with BufferBlock<T>

```
public class Producer
{
    private static Random random = new Random((int)DateTime.UtcNow.Ticks);
    //produce the value that will be posted to buffer block
    public double Produce ( )
    {
        var value = random.NextDouble();
        Console.WriteLine($"Producing value: {value}");
        return value;
    }
}

public class Consumer
{
    //consume the value that will be received from buffer block
    public void Consume (double value) => Console.WriteLine($"Consuming value: {value}");
}

class Program
{
    private static BufferBlock<double> buffer = new BufferBlock<double>();
    static void Main (string[] args)
    {
        //start a task that will every 1 second post a value from the producer to buffer block
        var producerTask = Task.Run(async () =>
        {
            var producer = new Producer();
            while(true)
            {
                buffer.Post(producer.Produce());
                await Task.Delay(1000);
            }
        });
        //start a task that will recieve values from bufferblock and consume it
        var consumerTask = Task.Run(() =>
        {
            var consumer = new Consumer();
            while(true)
```

```
            {
consumer.Consume(buffer.Receive());
            }
        });

Task.WaitAll(new[] { producerTask, consumerTask });
    }
}
```



```
            {
                consumer.Consume(buffer.Receive());
            }
        });

        Task.WaitAll(new[] { producerTask, consumerTask });
    }
}
```

第39章：线程

第39.1节：从其他线程访问窗体控件

如果你想从创建控件的GUI线程以外的其他线程更改控件（如文本框或标签）的属性，
你必须调用它，否则可能会收到错误信息，内容如下：

> “跨线程操作无效：控件 'control_name' 被非创建它的线程访问。”

在 system.windows.forms 窗体上使用此示例代码将抛出带有该消息的异常：

```
private void button4_Click(object sender, EventArgs e)
{
Thread thread = new Thread(updatetextbox);
    thread.Start();
}

private void updatetextbox()
{
textBox1.Text = "updated"; // 抛出异常
}
```

当你想从不拥有该控件的线程中更改文本框的文本时，应使用 Control.Invoke 或
Control.BeginInvoke。你也可以使用 Control.InvokeRequired 来检查是否需要调用控件。

```
private void updatetextbox()
{
    if (textBox1.InvokeRequired)
textBox1.BeginInvoke((Action)(() => textBox1.Text = "updated"));
    else
textBox1.Text = "updated";
}
```

如果你需要经常这样做，可以为可调用对象编写一个扩展方法，以减少进行此检查所需的代码量：

```
public static class Extensions
{
    public static void BeginInvokeIfRequired(this ISynchronizeInvoke obj, Action action)
    {
        if (obj.InvokeRequired)
obj.BeginInvoke(action, new object[0]);
        else
action();
    }
}
```

从任何线程更新文本框变得更简单了一些：

```
private void updatetextbox()
{
textBox1.BeginInvokeIfRequired(() => textBox1.Text = "updated");
}
```


Chapter 39: Threading

Section 39.1: Accessing form controls from other threads

If you want to change an attribute of a control such as a textbox or label from another thread than the GUI thread that created the control, you will have to invoke it or else you might get an error message stating:

> "Cross-thread operation not valid: Control 'control_name' accessed from a thread other than the thread it was created on."

Using this example code on a system.windows.forms form will cast an exception with that message:

```
private void button4_Click(object sender, EventArgs e)
{
    Thread thread = new Thread(updatetextbox);
    thread.Start();
}

private void updatetextbox()
{
    textBox1.Text = "updated"; // Throws exception
}
```

Instead when you want to change a textbox's text from within a thread that doesn't own it use Control.Invoke or Control.BeginInvoke. You can also use Control.InvokeRequired to check if invoking the control is necessary.

```
private void updatetextbox()
{
    if (textBox1.InvokeRequired)
        textBox1.BeginInvoke((Action)(() => textBox1.Text = "updated"));
    else
        textBox1.Text = "updated";
}
```

If you need to do this often, you can write an extension for invokeable objects to reduce the amount of code necessary to make this check:

```
public static class Extensions
{
    public static void BeginInvokeIfRequired(this ISynchronizeInvoke obj, Action action)
    {
        if (obj.InvokeRequired)
            obj.BeginInvoke(action, new object[0]);
        else
            action();
    }
}
```

And updating the textbox from any thread becomes a bit simpler:

```
private void updatetextbox()
{
    textBox1.BeginInvokeIfRequired(() => textBox1.Text = "updated");
}
```

请注意，本示例中使用的 Control.BeginInvoke 是异步的，这意味着在调用 Control.BeginInvoke 之后的代码可以立即执行，无论传入的委托是否已经执行完毕。

如果你需要确保在继续执行之前 textBox1 已经更新，请改用 Control.Invoke，它会阻塞调用线程，直到你的委托执行完毕。需要注意的是，如果你频繁调用 Invoke，这种方式会显著降低代码性能；并且如果你的 GUI 线程正在等待调用线程完成或释放某个资源，这会导致应用程序死锁。

Be aware that Control.BeginInvoke as used in this example is asynchronous, meaning that code coming after a call to Control.BeginInvoke can be run immedeately after, whether or not the passed delegate has been executed yet.

If you need to be sure that textBox1 is updated before continuing, use Control.Invoke instead, which will block the calling thread until your delegate has been executed. Do note that this approach can slow your code down significantly if you make many invoke calls and note that it will deadlock your application if your GUI thread is waiting for the calling thread to complete or release a held resource.

第40章：进程和线程亲和性设置

参数	详情
亲和性	描述进程允许运行的处理器集合的整数。例如，在一个8处理器系统上，如果你希望进程只在处理器3和4上运行，那么你可以选择如下亲和性：00001100，等于12

第40.1节：获取进程亲和性掩码

```
    public static int GetProcessAffinityMask(string processName = null)
    {
Process myProcess = GetProcessByName(ref processName);

         int processorAffinity = (int)myProcess.ProcessorAffinity;
        Console.WriteLine("进程 {0} 的亲和掩码是 : {1}", processName,
FormatAffinity(processorAffinity));

         return processorAffinity;
    }

    public static Process GetProcessByName(ref string processName)
    {
Process myProcess;
        if (string.IsNullOrEmpty(processName))
        {
myProcess = Process.GetCurrentProcess();
            processName = myProcess.ProcessName;
        }
        否则
        {
Process[] processList = Process.GetProcessesByName(processName);
            myProcess = processList[0];
        }
        return myProcess;
    }

    private static string FormatAffinity(int affinity)
    {
        return Convert.ToString(affinity, 2).PadLeft(Environment.ProcessorCount, '0');
    }
}
```

使用示例：

```
private static void Main(string[] args)
{
GetProcessAffinityMask();

    Console.ReadKey();
}
// 输出：
// 进程 Test.vshost 亲和掩码是 : 11111111
```

第40.2节：设置进程亲和掩码

```
    public static void SetProcessAffinityMask(int affinity, string processName = null)
```


Chapter 40: Process and Thread affinity setting

Parameter	Details
affinity	integer that describes the set of processors on which the process is allowed to run. For example, on a 8 processor system if you want your process to be executed only on processors 3 and 4 than you choose affinity like this : 00001100 which equals 12

Section 40.1: Get process affinity mask

```
    public static int GetProcessAffinityMask(string processName = null)
    {
        Process myProcess = GetProcessByName(ref processName);

        int processorAffinity = (int)myProcess.ProcessorAffinity;
        Console.WriteLine("Process {0} Affinity Mask is : {1}", processName,
FormatAffinity(processorAffinity));

        return processorAffinity;
    }

    public static Process GetProcessByName(ref string processName)
    {
        Process myProcess;
        if (string.IsNullOrEmpty(processName))
        {
            myProcess = Process.GetCurrentProcess();
            processName = myProcess.ProcessName;
        }
        else
        {
            Process[] processList = Process.GetProcessesByName(processName);
            myProcess = processList[0];
        }
        return myProcess;
    }

    private static string FormatAffinity(int affinity)
    {
        return Convert.ToString(affinity, 2).PadLeft(Environment.ProcessorCount, '0');
    }
}
```

Example of usage :

```
private static void Main(string[] args)
{
    GetProcessAffinityMask();

    Console.ReadKey();
}
// Output:
// Process Test.vshost Affinity Mask is : 11111111
```

Section 40.2: Set process affinity mask

```
    public static void SetProcessAffinityMask(int affinity, string processName = null)
```

```
    {
Process myProcess = GetProcessByName(ref processName);

        Console.WriteLine("进程 {0} 旧的亲和掩码是：{1}", processName,
FormatAffinity((int)myProcess.ProcessorAffinity));

myProcess.ProcessorAffinity = new IntPtr(affinity);
        Console.WriteLine("进程 {0} 新的亲和掩码是：{1}", processName,
FormatAffinity((int)myProcess.ProcessorAffinity));
    }
```

使用示例：

```
private static void Main(string[] args)
{
    int newAffinity = Convert.ToInt32("10101010", 2);
    SetProcessAffinityMask(newAffinity);

    Console.ReadKey();
}
// 输出：
// 进程 Test.vshost 旧的亲和掩码是：11111111
// 进程 Test.vshost 新的亲和掩码是：10101010
```



```
    {
        Process myProcess = GetProcessByName(ref processName);

        Console.WriteLine("Process {0} Old Affinity Mask is : {1}", processName,
FormatAffinity((int)myProcess.ProcessorAffinity));

        myProcess.ProcessorAffinity = new IntPtr(affinity);
        Console.WriteLine("Process {0} New Affinity Mask is : {1}", processName,
FormatAffinity((int)myProcess.ProcessorAffinity));
    }
```

Example of usage :

```
private static void Main(string[] args)
{
    int newAffinity = Convert.ToInt32("10101010", 2);
    SetProcessAffinityMask(newAffinity);

    Console.ReadKey();
}
// Output :
// Process Test.vshost Old Affinity Mask is : 11111111
// Process Test.vshost New Affinity Mask is : 10101010
```

第41章：使用 .Net 框架的并行处理

本主题关于使用 .NET 框架的任务并行库进行多核编程。任务并行库允许你编写可读性强的代码，并根据可用核心数自动调整。因此，你可以确保你的软件会随着环境的升级自动升级。

第41.1节：并行扩展

并行扩展与任务并行库一起引入，以实现数据并行。数据并行指的是在源集合或数组中的元素上同时（即并行）执行相同操作的场景。 .NET 提供了新的结构，通过使用 Parallel.For 和 Parallel.Foreach 结构来实现数据并行。

```
// 顺序版本

foreach (var item in sourcecollection){

Process(item);

}

// 并行等价版本

Parallel.foreach(sourcecollection, item => Process(item));
```

上述的 Parallel.ForEach 结构利用了多核处理器，从而以相同的方式提升了性能。

Chapter 41: Parallel processing using .Net framework

This Topic is about Multi core programming using Task Parallel Library with .NET framework. The task parallel library allows you to write code which is human readable and adjusts itself with the number of Cores available. So you can be sure that your software would auto-upgrade itself with the upgrading environment.

Section 41.1: Parallel Extensions

Parallel extensions have been introduced along with the Task Parallel Library to achieve data Parallelism. Data parallelism refers to scenarios in which the same operation is performed concurrently (that is, in parallel) on elements in a source collection or array. The .NET provides new constructs to achieve data parallelism by using Parallel.For and Parallel.Foreach constructs.

```
//Sequential version

foreach (var item in sourcecollection){

Process(item);

}

// Parallel equivalent

Parallel.foreach(sourcecollection, item => Process(item));
```

The above mentioned Parallel.ForEach construct utilizes the multiple cores and thus enhances the performance in the same fashion.

第42章：任务并行库（TPL）

第42.1节：基本的生产者-消费者循环（BlockingCollection）

```
var collection = new BlockingCollection<int>(5);
var random = new Random();

var producerTask = Task.Run(() => {
    for(int item=1; item<=10; item++)
    {
collection.Add(item);
        Console.WriteLine("Produced: " + item);
        Thread.Sleep(random.Next(10,1000));
    }
collection.CompleteAdding();
    Console.WriteLine("Producer completed!");
});
```

值得注意的是，如果不调用collection.CompleteAdding();，即使消费者任务正在运行，你仍然可以继续向集合中添加元素。当你确定不再有新的添加时，只需调用collection.CompleteAdding();。此功能可用于实现多生产者单消费者模式，即多个来源向BlockingCollection中添加元素，单个消费者从中取出元素并进行处理。如果在调用CompleteAdding之前BlockingCollection为空，

从collection.GetConsumingEnumerable()获取的枚举将会阻塞，直到有新元素被添加到集合中，或者调用BlockingCollection.CompleteAdding();且队列为空。

```
var consumerTask = Task.Run(() => {
    foreach(var item in collection.GetConsumingEnumerable())
    {
Console.WriteLine("Consumed: " + item);
        Thread.Sleep(random.Next(10,1000));
    }
Console.WriteLine("消费者完成！");
});

Task.WaitAll(producerTask, consumerTask);

Console.WriteLine("全部完成！");
```

第42.2节：Parallel.Invoke

```
var actions = Enumerable.Range(1, 10).Select(n => new Action(() =>
{
Console.WriteLine("我是任务 " + n);
    if((n & 1) == 0)
        throw new Exception("来自任务 " + n的异常");
})).ToArray();

try
{
Parallel.Invoke(actions);
}
catch(AggregateException ex)
{
    foreach(var inner in ex.InnerExceptions)
        Console.WriteLine("任务失败： " + inner.Message);
```


Chapter 42: Task Parallel Library (TPL)

Section 42.1: Basic producer-consumer loop (BlockingCollection)

```
var collection = new BlockingCollection<int>(5);
var random = new Random();

var producerTask = Task.Run(() => {
    for(int item=1; item<=10; item++)
    {
        collection.Add(item);
        Console.WriteLine("Produced: " + item);
        Thread.Sleep(random.Next(10,1000));
    }
    collection.CompleteAdding();
    Console.WriteLine("Producer completed!");
});
```

It is worth noting that if you do not call `collection.CompleteAdding();`, you are able to keep adding to the collection even if your consumer task is running. Just call `collection.CompleteAdding();` when you are sure there are no more additions. This functionality can be used to make a Multiple Producer to a Single Consumer pattern where you have multiple sources feeding items into the BlockingCollection and a single consumer pulling items out and doing something with them. If your BlockingCollection is empty before you call complete adding, the Enumerable from `collection.GetConsumingEnumerable()` will block until a new item is added to the collection or BlockingCollection.CompleteAdding(); is called and the queue is empty.

```
var consumerTask = Task.Run(() => {
    foreach(var item in collection.GetConsumingEnumerable())
    {
        Console.WriteLine("Consumed: " + item);
        Thread.Sleep(random.Next(10,1000));
    }
    Console.WriteLine("Consumer completed!");
});

Task.WaitAll(producerTask, consumerTask);

Console.WriteLine("Everything completed!");
```

Section 42.2: Parallel.Invoke

```
var actions = Enumerable.Range(1, 10).Select(n => new Action(() =>
{
    Console.WriteLine("I'm task " + n);
    if((n & 1) == 0)
        throw new Exception("Exception from task " + n);
})).ToArray();

try
{
    Parallel.Invoke(actions);
}
catch(AggregateException ex)
{
    foreach(var inner in ex.InnerExceptions)
        Console.WriteLine("Task failed: " + inner.Message);
```

```
}
```

第42.3节：任务：返回值

返回值的任务具有返回类型Task< TResult >，其中 TResult 是需要返回的值的类型。你可以通过其 Result 属性查询任务的结果。

```
Task<int> t = Task.Run(() =>
    {
        int sum = 0;

        for(int i = 0; i < 500; i++)
            sum += i;

        return sum;
    });

Console.WriteLine(t.Result); // Outuput 124750
```

如果任务异步执行，则等待任务会返回其结果。

```
public async Task DoSomeWork()
{
WebClient client = new WebClient();
    // 因为任务被等待，任务的结果被赋值给 response
    string response = await client.DownloadStringTaskAsync("http://somedomain.com");
}
```

第42.4节：Parallel.ForEach

此示例使用Parallel.ForEach通过多个线程计算1到10000之间数字的总和。为了实现线程安全，使用了Interlocked.Add来累加数字。

```
using System.Threading;

int Foo()
{
    int total = 0;
    var numbers = Enumerable.Range(1, 10000).ToList();
    Parallel.ForEach(numbers,
        () => 0, // 初始值,
        (num, state, localSum) => num + localSum,
        localSum => Interlocked.Add(ref total, localSum));
    return total; // total = 50005000
}
```

第42.5节：Parallel.For

此示例使用Parallel.For通过多个线程计算1到10000之间数字的总和。为了实现线程安全，使用了Interlocked.Add来累加数字。

```
using System.Threading;

int Foo()
{
    int total = 0;
    Parallel.For(1, 10001,
```



```
}
```

Section 42.3: Task: Returning a value

Task that return a value has return type of `Task< TResult >` where TResult is the type of value that needs to be returned. You can query the outcome of a Task by its Result property.

```
Task<int> t = Task.Run(() =>
    {
        int sum = 0;

        for(int i = 0; i < 500; i++)
            sum += i;

        return sum;
    });

Console.WriteLine(t.Result); // Outuput 124750
```

If the Task execute asynchronously than awaiting the Task returns it's result.

```
public async Task DoSomeWork()
{
    WebClient client = new WebClient();
    // Because the task is awaited, result of the task is assigned to response
    string response = await client.DownloadStringTaskAsync("http://somedomain.com");
}
```

Section 42.4: Parallel.ForEach

This example uses `Parallel.ForEach` to calculate the sum of the numbers between 1 and 10000 by using multiple threads. To achieve thread-safety, `Interlocked.Add` is used to sum the numbers.

```
using System.Threading;

int Foo()
{
    int total = 0;
    var numbers = Enumerable.Range(1, 10000).ToList();
    Parallel.ForEach(numbers,
        () => 0, // initial value,
        (num, state, localSum) => num + localSum,
        localSum => Interlocked.Add(ref total, localSum));
    return total; // total = 50005000
}
```

Section 42.5: Parallel.For

This example uses `Parallel.For` to calculate the sum of the numbers between 1 and 10000 by using multiple threads. To achieve thread-safety, `Interlocked.Add` is used to sum the numbers.

```
using System.Threading;

int Foo()
{
    int total = 0;
    Parallel.For(1, 10001,
```

```
        () => 0, // 初始值,
        (num, state, localSum) => num + localSum,
        localSum => Interlocked.Add(ref total, localSum));
    return total; // total = 50005000
}
```

第42.6节：任务：基本实例化和等待

任务可以通过直接实例化Task类来创建...

```
var task = new Task(() =>
{
Console.WriteLine("任务代码开始..." );
    Thread.Sleep(2000);
Console.WriteLine("...任务代码结束！");
});

Console.WriteLine("开始任务..." );
task.Start();
task.Wait();
Console.WriteLine("任务完成！");
```

...或者使用静态的Task.Run方法：

```
Console.WriteLine("开始任务..." );
var task = Task.Run(() =>
{
Console.WriteLine("任务代码开始..." );
    Thread.Sleep(2000);
Console.WriteLine("...任务代码结束！");
});
task.Wait();
Console.WriteLine("任务完成！");
```

注意，只有在第一种情况下才需要显式调用Start。

第42.7节：Task.WhenAll

```
var random = new Random();
IEnumerable<Task<int>> 任务 = Enumerable.Range(1, 5).Select(n => Task.Run(() =>
{
Console.WriteLine("我是任务 " + n);
    return n;
}));

Task<int[]> 任务数组 = Task.WhenAll(任务);
int[] 结果 = await 任务数组;

Console.WriteLine(string.Join(",", 结果.Select(n => n.ToString())));
// 输出：1,2,3,4,5
```

第42.8节：使用AsyncLocal传递执行上下文

当你需要将一些数据从父任务传递给其子任务，使其逻辑上随执行流动时，
使用AsyncLocal类：

```
void Main()
{
```



```
        () => 0, // initial value,
        (num, state, localSum) => num + localSum,
        localSum => Interlocked.Add(ref total, localSum));
    return total; // total = 50005000
}
```

Section 42.6: Task: basic instantiation and Wait

A task can be created by directly instantiating the `Task` class...

```
var task = new Task(() =>
{
    Console.WriteLine("Task code starting...");
    Thread.Sleep(2000);
    Console.WriteLine("...task code ending!");
});

Console.WriteLine("Starting task...");
task.Start();
task.Wait();
Console.WriteLine("Task completed!");
```

...or by using the static `Task.Run` method:

```
Console.WriteLine("Starting task...");
var task = Task.Run(() =>
{
    Console.WriteLine("Task code starting...");
    Thread.Sleep(2000);
    Console.WriteLine("...task code ending!");
});
task.Wait();
Console.WriteLine("Task completed!");
```

Note that only in the first case it is necessary to explicitly invoke `Start`.

Section 42.7: Task.WhenAll

```
var random = new Random();
IEnumerable<Task<int>> tasks = Enumerable.Range(1, 5).Select(n => Task.Run(() =>
{
    Console.WriteLine("I'm task " + n);
    return n;
}));

Task<int[]> task = Task.WhenAll(tasks);
int[] results = await task;

Console.WriteLine(string.Join(",", results.Select(n => n.ToString())));
// Output: 1,2,3,4,5
```

Section 42.8: Flowing execution context with AsyncLocal

When you need to pass some data from the parent task to its children tasks, so it logically flows with the execution, use `AsyncLocal` class:

```
void Main()
{
```

```
AsyncLocal<string> 用户 = new AsyncLocal<string>();
    用户.Value = "初始用户";

    // 这不会影响其他任务——数值是相对于执行流程分支的局部值
Task.Run(() => user.Value = "user from another task");

    var task1 = Task.Run(() =>
    {
Console.WriteLine(user.Value); // 输出 "initial user"
        Task.Run(() =>
          {
              // 输出 "initial user" - 值已从主方法传递到此任务且未被更改

Console.WriteLine(user.Value);
          }).Wait();

user.Value = "user from task1";

          Task.Run(() =>
          {
              // 输出 "user from task1" - 值已从主方法传递到 task1
              // 然后值被更改并传递到此任务。
Console.WriteLine(user.Value);
          }).Wait();
    });

task1.Wait();

    // 输出 "initial user" - 更改不会向上游执行流传播
Console.WriteLine(user.Value);
}
```

注意： 如上例所示， AsyncLocal.Value 具有 读时复制（copy on read） 语义，但如果你传递某个引用类型并修改其属性，将会影响其他任务。因此，使用 AsyncLocal 的最佳实践是使用值类型或不可变类型。

第42.9节：VB.NET中的Parallel.ForEach

```
For Each row As DataRow In FooDataTable.Rows
    Me.RowsToProcess.Add(row)
Next

Dim myOptions As ParallelOptions = New ParallelOptions()
myOptions.MaxDegreeOfParallelism = environment.processorcount

Parallel.ForEach(RowsToProcess, myOptions, Sub(currentRow, state)
                                       ProcessRowParallel(currentRow, state)
                                   End Sub)
```

第42.10节：任务：WaitAll和变量捕获

```
var tasks = Enumerable.Range(1, 5).Select(n => new Task<int>(() =>
{
Console.WriteLine("我是任务 " + n);
    return n;
})).ToArray();

foreach(var task in tasks) task.Start();
Task.WaitAll(tasks);
```



```
    AsyncLocal<string> user = new AsyncLocal<string>();
    user.Value = "initial user";

    // this does not affect other tasks - values are local relative to the branches of execution flow
    Task.Run(() => user.Value = "user from another task");

    var task1 = Task.Run(() =>
    {
        Console.WriteLine(user.Value); // outputs "initial user"
        Task.Run(() =>
        {
            // outputs "initial user" - value has flown from main method to this task without being
changed
            Console.WriteLine(user.Value);
        }).Wait();

        user.Value = "user from task1";

        Task.Run(() =>
        {
            // outputs "user from task1" - value has flown from main method to task1
            // than value was changed and flown to this task.
            Console.WriteLine(user.Value);
        }).Wait();
    });

    task1.Wait();

    // outputs "initial user" - changes do not propagate back upstream the execution flow
    Console.WriteLine(user.Value);
}
```

Note: As can be seen from the example above AsynLocal.Value has copy on read semantic, but if you flow some reference type and change its properties you will affect other tasks. Hence, best practice with AsyncLocal is to use value types or immutable types.

Section 42.9: Parallel.ForEach in VB.NET

```
For Each row As DataRow In FooDataTable.Rows
    Me.RowsToProcess.Add(row)
Next

Dim myOptions As ParallelOptions = New ParallelOptions()
myOptions.MaxDegreeOfParallelism = environment.processorcount

Parallel.ForEach(RowsToProcess, myOptions, Sub(currentRow, state)
                                       ProcessRowParallel(currentRow, state)
                                   End Sub)
```

Section 42.10: Task: WaitAll and variable capturing

```
var tasks = Enumerable.Range(1, 5).Select(n => new Task<int>(() =>
{
    Console.WriteLine("I'm task " + n);
    return n;
})).ToArray();

foreach(var task in tasks) task.Start();
Task.WaitAll(tasks);
```

```
foreach(var task in tasks)
    Console.WriteLine(task.Result);
```

第42.11节：任务：WaitAny

```
var allTasks = Enumerable.Range(1, 5).Select(n => new Task<int>(() => n)).ToArray();
var pendingTasks = allTasks.ToArray();

foreach(var task in allTasks) task.Start();

while(pendingTasks.Length > 0)
{
    var finishedTask = pendingTasks[Task.WaitAny(pendingTasks)];
    Console.WriteLine("任务 {0} 完成", finishedTask.Result);
    pendingTasks = pendingTasks.Except(new[] {finishedTask}).ToArray();
}

Task.WaitAll(allTasks);
```

注意： 最后的 WaitAll 是必要的，因为 WaitAny 不会导致异常被观察到。

第42.12节：任务：处理异常（使用 Wait）

```
var task1 = Task.Run(() =>
{
Console.WriteLine("任务 1 代码开始..." );
    throw new Exception("哦不，任务 1 抛出异常！！");
});

var task2 = Task.Run(() =>
{
Console.WriteLine("任务 2 代码开始..." );
    throw new Exception("哦不，任务 2 抛出异常！！");
});

Console.WriteLine("开始任务...");
try
{
Task.WaitAll(task1, task2);
}
catch(AggregateException ex)
{
Console.WriteLine("任务失败！");
    foreach(var inner in ex.InnerExceptions)
        Console.WriteLine(inner.Message);
}

Console.WriteLine("任务1状态是: " + task1.Status); //Faulted
Console.WriteLine("任务2状态是: " + task2.Status); //Faulted
```

第42.13节：任务：处理异常（不使用Wait）

```
var task1 = Task.Run(() =>
{
Console.WriteLine("任务 1 代码开始..." );
    throw new Exception("哦不，任务 1 抛出异常！！");
});

var task2 = Task.Run(() =>
```



```
foreach(var task in tasks)
    Console.WriteLine(task.Result);
```

Section 42.11: Task: WaitAny

```
var allTasks = Enumerable.Range(1, 5).Select(n => new Task<int>(() => n)).ToArray();
var pendingTasks = allTasks.ToArray();

foreach(var task in allTasks) task.Start();

while(pendingTasks.Length > 0)
{
    var finishedTask = pendingTasks[Task.WaitAny(pendingTasks)];
    Console.WriteLine("Task {0} finished", finishedTask.Result);
    pendingTasks = pendingTasks.Except(new[] {finishedTask}).ToArray();
}

Task.WaitAll(allTasks);
```

Note: The final `WaitAll` is necessary becasue `WaitAny` does not cause exceptions to be observed.

Section 42.12: Task: handling exceptions (using Wait)

```
var task1 = Task.Run(() =>
{
    Console.WriteLine("Task 1 code starting...");
    throw new Exception("Oh no, exception from task 1!!");
});

var task2 = Task.Run(() =>
{
    Console.WriteLine("Task 2 code starting...");
    throw new Exception("Oh no, exception from task 2!!");
});

Console.WriteLine("Starting tasks...");
try
{
    Task.WaitAll(task1, task2);
}
catch(AggregateException ex)
{
    Console.WriteLine("Task(s) failed!");
    foreach(var inner in ex.InnerExceptions)
        Console.WriteLine(inner.Message);
}

Console.WriteLine("Task 1 status is: " + task1.Status); //Faulted
Console.WriteLine("Task 2 status is: " + task2.Status); //Faulted
```

Section 42.13: Task: handling exceptions (without using Wait)

```
var task1 = Task.Run(() =>
{
    Console.WriteLine("Task 1 code starting...");
    throw new Exception("Oh no, exception from task 1!!");
});

var task2 = Task.Run(() =>
```

```
{
Console.WriteLine("任务 2 代码开始..." );
    throw new Exception("哦不，任务 2 抛出异常！！");
});

var tasks = new[] {task1, task2};

Console.WriteLine("开始任务...");
while(tasks.All(task => !task.IsCompleted));

foreach(var task in tasks)
{
    if(task.IsFaulted)
Console.WriteLine("任务失败: " +
            task.Exception.InnerExceptions.First().Message);
}

Console.WriteLine("任务1状态是: " + task1.Status); //Faulted
Console.WriteLine("任务2状态是: " + task2.Status); //Faulted
```

第42.14节：任务：使用CancellationToken取消

```
var cancellationTokenSource = new CancellationTokenSource();
var cancellationToken = cancellationTokenSource.Token;

var task = new Task((state) =>
    {
        int i = 1;
        var myCancellationToken = (CancellationToken)state;
        while(true)
        {
Console.Write("{0} ", i++);
            Thread.Sleep(1000);
myCancellationToken.ThrowIfCancellationRequested();
        }
    },
cancellationToken: cancellationToken,
    state: cancellationToken);

Console.WriteLine("Counting to infinity. Press any key to cancel!");
task.Start();
Console.ReadKey();

cancellationTokenSource.Cancel();
try
{
task.Wait();
}
catch(AggregateException ex)
{
ex.Handle(inner => inner is OperationCanceledException);
}

Console.WriteLine($"{Environment.NewLine}You have cancelled! Task status is: {task.Status}");
//已取消
```

作为ThrowIfCancellationRequested的替代方案，可以通过IsCancellationRequested检测取消请求，并手动抛出OperationCanceledException：

```
//新任务委托
```



```
{
    Console.WriteLine("Task 2 code starting...");
    throw new Exception("Oh no, exception from task 2!!");
});

var tasks = new[] {task1, task2};

Console.WriteLine("Starting tasks...");
while(tasks.All(task => !task.IsCompleted));

foreach(var task in tasks)
{
    if(task.IsFaulted)
        Console.WriteLine("Task failed: " +
            task.Exception.InnerExceptions.First().Message);
}

Console.WriteLine("Task 1 status is: " + task1.Status); //Faulted
Console.WriteLine("Task 2 status is: " + task2.Status); //Faulted
```

Section 42.14: Task: cancelling using CancellationToken

```
var cancellationTokenSource = new CancellationTokenSource();
var cancellationToken = cancellationTokenSource.Token;

var task = new Task((state) =>
    {
        int i = 1;
        var myCancellationToken = (CancellationToken)state;
        while(true)
        {
            Console.Write("{0} ", i++);
            Thread.Sleep(1000);
            myCancellationToken.ThrowIfCancellationRequested();
        }
    },
    cancellationToken: cancellationToken,
    state: cancellationToken);

Console.WriteLine("Counting to infinity. Press any key to cancel!");
task.Start();
Console.ReadKey();

cancellationTokenSource.Cancel();
try
{
    task.Wait();
}
catch(AggregateException ex)
{
    ex.Handle(inner => inner is OperationCanceledException);
}

Console.WriteLine($"{Environment.NewLine}You have cancelled! Task status is: {task.Status}");
//Canceled
```

As an alternative to ThrowIfCancellationRequested, the cancellation request can be detected with IsCancellationRequested and a OperationCanceledException can be thrown manually:

```
//New task delegate
```

```
int i = 1;
var myCancellationToken = (CancellationToken)state;
while(!myCancellationToken.IsCancellationRequested)
{
Console.Write("{0} ", i++);
    Thread.Sleep(1000);
}
Console.WriteLine($"{Environment.NewLine}哎呀，我已被取消！！");
throw new OperationCanceledException(myCancellationToken);
```

注意取消令牌是如何通过cancellationToken参数传递给任务构造函数的。这是
必要的，以便当调用ThrowIfCancellationRequested时，任务状态转变为Canceled状态，而不是Faulted状态。
同样出于这个原因，在第二种情况下，取消令牌也被显式地传递给OperationCanceledException的构造函数。

第42.15节：Task.WhenAny

```
var random = new Random();
IEnumerable<Task<int>> tasks = Enumerable.Range(1, 5).Select(n => Task.Run(async() =>
{
Console.WriteLine("我是任务 " + n);
    await Task.Delay(random.Next(10,1000));
    return n;
}));

Task<Task<int>> whenAnyTask = Task.WhenAny(tasks);
Task<int> completedTask = await whenAnyTask;
Console.WriteLine("获胜者是：任务 " + await completedTask);

await Task.WhenAll(tasks);
Console.WriteLine("所有任务已完成！");
```



```
int i = 1;
var myCancellationToken = (CancellationToken)state;
while(!myCancellationToken.IsCancellationRequested)
{
    Console.Write("{0} ", i++);
    Thread.Sleep(1000);
}
Console.WriteLine($"{Environment.NewLine}Ouch, I have been cancelled!!");
throw new OperationCanceledException(myCancellationToken);
```

Note how the cancellation token is passed to the task constructor in the `cancellationToken` parameter. This is needed so that the task transitions to the `Canceled` state, not to the `Faulted` state, when `ThrowIfCancellationRequested` is invoked. Also, for the same reason, the cancellation token is explicitly supplied in the constructor of `OperationCanceledException` in the second case.

Section 42.15: Task.WhenAny

```
var random = new Random();
IEnumerable<Task<int>> tasks = Enumerable.Range(1, 5).Select(n => Task.Run(async() =>
{
    Console.WriteLine("I'm task " + n);
    await Task.Delay(random.Next(10,1000));
    return n;
}));

Task<Task<int>> whenAnyTask = Task.WhenAny(tasks);
Task<int> completedTask = await whenAnyTask;
Console.WriteLine("The winner is: task " + await completedTask);

await Task.WhenAll(tasks);
Console.WriteLine("All tasks finished!");
```

第43章：任务并行库（TPL）API概述

第43.1节：响应按钮点击执行工作并更新用户界面

此示例演示如何通过在工作线程上执行一些工作来响应按钮点击，
然后更新用户界面以指示完成情况

```
void MyButton_OnClick(object sender, EventArgs args)
{
Task.Run(() => // 使用线程池调度工作
        {
                        System.Threading.Thread.Sleep(5000); // 睡眠5秒以模拟工作。
        })
    .ContinueWith(p => // 这个续续包含将在UI线程上运行的“更新”代码
    {
        this.TextBlock_ResultText.Text = "工作完成于 " + DateTime.Now.ToString()
    },
TaskScheduler.FromCurrentSynchronizationContext()); // 确保更新在UI线程上运行。


}
```


Chapter 43: Task Parallel Library (TPL) API Overviews

Section 43.1: Perform work in response to a button click and update the UI

This example demonstrates how you can respond to a button click by performing some work on a worker thread and then update the user interface to indicate completion

```
void MyButton_OnClick(object sender, EventArgs args)
{
    Task.Run(() => // Schedule work using the thread pool
        {
            System.Threading.Thread.Sleep(5000); // Sleep for 5 seconds to simulate work.
        })
    .ContinueWith(p => // this continuation contains the 'update' code to run on the UI thread
    {
        this.TextBlock_ResultText.Text = "The work completed at " + DateTime.Now.ToString()
    },
    TaskScheduler.FromCurrentSynchronizationContext()); // make sure the update is run on the UI
thread.

}
```

第44章：同步上下文

第44.1节：在执行后台工作后在UI线程上执行代码

此示例展示了如何使用SynchronizationContext从后台线程更新UI组件

```
void Button_Click(object sender, EventArgs args)
{
SynchronizationContext context = SynchronizationContext.Current;
    Task.Run(() =>
    {
        for(int i = 0; i < 10; i++)
        {
Thread.Sleep(500); // 模拟正在进行的工作
            context.Post(ShowProgress, "项目 " + i " 工作完成");
        }
    }
}

void UpdateCallback(object state)
{
    // 由于此方法仅从UI线程调用，因此可以安全地更新UI
    this.MyTextBox.Text = state as string;
}
```

在此示例中，如果你尝试在for循环内直接更新MyTextBox.Text，将会遇到线程错误。
通过将UpdateCallback操作发布到SynchronizationContext，文本框会在与其余用户界面相同的线程上更新。

在实际操作中，进度更新应使用System.IProgress<T>的实例来执行。默认的
实现System.Progress<T>会自动捕获其创建时的同步上下文。

Chapter 44: Synchronization Contexts

Section 44.1: Execute code on the UI thread after performing background work

This example shows how to update a UI component from a background thread by using a SynchronizationContext

```
void Button_Click(object sender, EventArgs args)
{
    SynchronizationContext context = SynchronizationContext.Current;
    Task.Run(() =>
    {
        for(int i = 0; i < 10; i++)
        {
            Thread.Sleep(500); //simulate work being done
            context.Post(ShowProgress, "Work complete on item " + i);
        }
    }
}

void UpdateCallback(object state)
{
    // UI can be safely updated as this method is only called from the UI thread
    this.MyTextBox.Text = state as string;
}
```

In this example, if you tried to directly update MyTextBox.Text inside the for loop, you would get a threading error. By posting the UpdateCallback action to the SynchronizationContext, the text box is updated on the same thread as the rest of the UI.

In practice, progress updates should be performed using an instance of System.IProgress<T>. The default implementation System.Progress<T> automatically captures the synchronisation context it is created on.

第45章：内存管理

第45.1节：在封装非托管资源时使用SafeHandle

在为非托管资源编写包装器时，您应该继承SafeHandle，而不是尝试自行实现自己实现 IDisposable 和终结器。你的 SafeHandle 子类应尽可能小且简单，以最大限度地减少句柄泄漏的可能性。这很可能意味着你的 SafeHandle 实现将是一个类的内部实现细节，该类封装它以提供可用的 API。该类确保即使程序泄漏了你的 SafeHandle 实例，你的非托管句柄也会被释放。

```
using System.Runtime.InteropServices;

class MyHandle : SafeHandle
{
    public override bool IsInvalid => handle == IntPtr.Zero;
    public MyHandle() : base(IntPtr.Zero, true)
    { }

    public MyHandle(int length) : this()
    {
SetHandle(Marshal.AllocHGlobal(length));
    }

    protected override bool ReleaseHandle()
    {
Marshal.FreeHGlobal(handle);
        return true;
    }
}
```

免责声明：此示例旨在展示如何使用SafeHandle来保护托管资源，SafeHandle为你实现了IDisposable接口并适当配置了终结器。此示例非常牵强，以这种方式分配一块内存很可能毫无意义。

第45.2节：非托管资源

当我们谈论GC和“堆”时，实际上指的是所谓的托管堆。托管堆上的对象可以访问托管堆之外的资源，例如在写入或读取文件时。当文件被打开用于读取后发生异常，导致文件句柄未能像正常情况那样关闭时，可能会出现意外行为。因此，.NET要求非托管资源实现IDisposable接口。该接口只有一个无参数的方法，称为Dispose：

```
public interface IDisposable
{
Dispose();
}
```

在处理非托管资源时，应确保它们被正确释放。你可以通过在finally块中显式调用Dispose()，或者使用using语句来实现。

```
StreamReader sr;
string textFromFile;
string filename = "SomeFile.txt";
try
```


Chapter 45: Memory management

Section 45.1: Use SafeHandle when wrapping unmanaged resources

When writing wrappers for unmanaged resources, you should subclass `SafeHandle` rather than trying to implement `IDisposable` and a finalizer yourself. Your `SafeHandle` subclass should be as small and simple as possible to minimize the chance of a handle leak. This likely means that your SafeHandle implementation would an internal implementation detail of a class which wraps it to provide a usable API. This class ensures that, even if a program leaks your `SafeHandle` instance, your unmanaged handle is released.

```
using System.Runtime.InteropServices;

class MyHandle : SafeHandle
{
    public override bool IsInvalid => handle == IntPtr.Zero;
    public MyHandle() : base(IntPtr.Zero, true)
    { }

    public MyHandle(int length) : this()
    {
        SetHandle(Marshal.AllocHGlobal(length));
    }

    protected override bool ReleaseHandle()
    {
        Marshal.FreeHGlobal(handle);
        return true;
    }
}
```

Disclaimer: This example is an attempt to show how to guard a managed resource with `SafeHandle` which implements `IDisposable` for you and configures finalizers appropriately. It is very contrived and likely pointless to allocate a chunk of memory in this manner.

Section 45.2: Unmanaged Resources

When we talk about the GC and the "heap", we're really talking about what's called the *managed heap*. Objects on the *managed heap* can access resources not on the managed heap, for example, when writing to or reading from a file. Unexpected behavior can occur when, a file is opened for reading and then an exception occurs, preventing the file handle from closing as it normally would. For this reason, .NET requires that unmanaged resources implement the `IDisposable` interface. This interface has a single method called `Dispose` with no parameters:

```
public interface IDisposable
{
    Dispose();
}
```

When handling unmanaged resources, you should make sure that they are properly disposed. You can do this by explicitly calling `Dispose()` in a `finally` block, or with a `using` statement.

```
StreamReader sr;
string textFromFile;
string filename = "SomeFile.txt";
try
```

```
{
sr = new StreamReader(filename);
    textFromFile = sr.ReadToEnd();
}
finally
{
    if (sr != null) sr.Dispose();
}
```

或者

```
string textFromFile;
string filename = "SomeFile.txt";

using (StreamReader sr = new Streamreader(filename))
{
textFromFile = sr.ReadToEnd();
}
```

后者是首选方法，并且在编译期间会自动扩展为前者。


```
{
    sr = new StreamReader(filename);
    textFromFile = sr.ReadToEnd();
}
finally
{
    if (sr != null) sr.Dispose();
}
```

or

```
string textFromFile;
string filename = "SomeFile.txt";

using (StreamReader sr = new Streamreader(filename))
{
    textFromFile = sr.ReadToEnd();
}
```

The latter is the preferred method, and is automatically expanded to the former during compilation.

第46章：垃圾回收

在 .Net 中，使用 new() 创建的对象分配在托管堆上。这些对象从不由使用它们的程序显式终结；相反，这一过程由 .Net 垃圾回收器控制。

下面的一些示例是“实验案例”，用于展示垃圾回收器的工作及其行为的一些重要细节，而其他示例则侧重于如何准备类以便垃圾回收器正确处理。

第46.1节：（垃圾）回收的基本示例

给定以下类：

```
public class FinalizableObject
{
    public FinalizableObject()
    {
Console.WriteLine("实例已初始化");
    }

~FinalizableObject()
    {
Console.WriteLine("实例已终结");
    }
}
```

一个创建实例的程序，即使不使用它：

```
new FinalizableObject(); // 对象已实例化，准备使用
```

产生以下输出：

```
<namespace>.FinalizableObject 已初始化
```

如果没有其他操作，对象直到程序结束时才会被终结（此时会释放托管堆上的所有对象，并在此过程中终结它们）。

可以在指定点强制垃圾回收器运行，方法如下：

```
new FinalizableObject(); // 对象已实例化，准备使用
GC.Collect();
```

这会产生以下结果：

```
<namespace>.FinalizableObject 已初始化
<namespace>.FinalizableObject 已终结
```

这次，一旦垃圾回收器被调用，未使用的（也称为“死”）对象就会被终结并从托管堆中释放。

第46.2节：存活对象和死对象——基础知识

经验法则：当发生垃圾回收时，“存活对象”是指仍在使用中的对象，而“死对象”是指不再使用的对象（任何引用它们的变量或字段，如果有的话，在回收发生前已经超出作用域）。

Chapter 46: Garbage Collection

In .Net, objects created with new() are allocated on the managed heap. These objects are never explicitly finalized by the program that uses them; instead, this process is controlled by the .Net Garbage Collector.

Some of the examples below are "lab cases" to show the Garbage Collector at work and some significant details of its behavior, while other focus on how to prepare classes for proper handling by the Garbage Collector.

Section 46.1: A basic example of (garbage) collection

Given the following class:

```
public class FinalizableObject
{
    public FinalizableObject()
    {
        Console.WriteLine("Instance initialized");
    }

    ~FinalizableObject()
    {
        Console.WriteLine("Instance finalized");
    }
}
```

A program that creates an instance, even without using it:

```
new FinalizableObject(); // Object instantiated, ready to be used
```

Produces the following output:

```
<namespace>.FinalizableObject initialized
```

If nothing else happens, the object is not finalized until the program ends (which frees all objects on the managed heap, finalizing these in the process).

It is possible to force the Garbage Collector to run at a given point, as follows:

```
new FinalizableObject(); // Object instantiated, ready to be used
GC.Collect();
```

Which produces the following result:

```
<namespace>.FinalizableObject initialized
<namespace>.FinalizableObject finalized
```

This time, as soon as the Garbage Collector was invoked, the unused (aka "dead") object was finalized and freed from the managed heap.

Section 46.2: Live objects and dead objects - the basics

Rule of thumb: when garbage collection occurs, "live objects" are those still in use, while "dead objects" are those no longer used (any variable or field referencing them, if any, has gone out of scope before the collection occurs).

在下面的示例中（为了方便，FinalizableObject1和FinalizableObject2是上面示例中FinalizableObject的子类，因此继承了初始化/终结消息的行为）：

```
var obj1 = new FinalizableObject1(); // 这里分配了Finalizable1实例
var obj2 = new FinalizableObject2(); // 这里分配了Finalizable2实例
obj1 = null; // 不再有对Finalizable1实例的引用
GC.Collect();
```

输出将是：

```
<namespace>.FinalizableObject1 初始化
<namespace>.FinalizableObject2 初始化
<namespace>.FinalizableObject1 终结
```

当垃圾回收器被调用时，FinalizableObject1是一个死对象并被终结，而FinalizableObject2是一个存活对象，保留在托管堆上。

第46.3节：多个死对象

如果两个（或多个）本应被回收的对象相互引用，会怎样？下面的例子展示了这种情况，假设 OtherObject 是 FinalizableObject 的一个公共属性：

```
var obj1 = new FinalizableObject1();
var obj2 = new FinalizableObject2();
obj1.OtherObject = obj2;
obj2.OtherObject = obj1;
obj1 = null; // 程序不再引用 Finalizable1 实例
obj2 = null; // 程序不再引用 Finalizable2 实例
// 但这两个对象仍然相互引用
GC.Collect();
```

这会产生以下输出：

```
<namespace>.FinalizedObject1 初始化完成
<namespace>.FinalizedObject2 初始化完成
<namespace>.FinalizedObject1 已终结
<namespace>.FinalizedObject2 已终结
```

尽管这两个对象相互引用，但它们仍被终结并从托管堆中释放（因为没有其他实际存活的对象引用它们中的任何一个）。

第46.4节：弱引用

弱引用是指对其他对象（也称为“目标”）的引用，但“弱”是指它们不会阻止这些对象被垃圾回收。换句话说，弱引用在垃圾回收器评估对象为“存活”或“死亡”时不计入其中。

以下代码：

```
var weak = new WeakReference<FinalizableObject>(new FinalizableObject());
GC.Collect();
```

产生输出：

```
<namespace>.FinalizableObject 已初始化
```


In the following example (for convenience, FinalizableObject1 and FinalizableObject2 are subclasses of FinalizableObject from the example above and thus inherit the initialization / finalization message behavior):

```
var obj1 = new FinalizableObject1(); // Finalizable1 instance allocated here
var obj2 = new FinalizableObject2(); // Finalizable2 instance allocated here
obj1 = null; // No more references to the Finalizable1 instance
GC.Collect();
```

The output will be:

```
<namespace>.FinalizableObject1 initialized
<namespace>.FinalizableObject2 initialized
<namespace>.FinalizableObject1 finalized
```

At the time when the Garbage Collector is invoked, FinalizableObject1 is a dead object and gets finalized, while FinalizableObject2 is a live object and it is kept on the managed heap.

Section 46.3: Multiple dead objects

What if two (or several) otherwise dead objects reference one another? This is shown in the example below, supposing that OtherObject is a public property of FinalizableObject:

```
var obj1 = new FinalizableObject1();
var obj2 = new FinalizableObject2();
obj1.OtherObject = obj2;
obj2.OtherObject = obj1;
obj1 = null; // Program no longer references Finalizable1 instance
obj2 = null; // Program no longer references Finalizable2 instance
// But the two objects still reference each other
GC.Collect();
```

This produces the following output:

```
<namespace>.FinalizedObject1 initialized
<namespace>.FinalizedObject2 initialized
<namespace>.FinalizedObject1 finalized
<namespace>.FinalizedObject2 finalized
```

The two objects are finalized and freed from the managed heap despite referencing each other (because no other reference exists to any of them from an actually live object).

Section 46.4: Weak References

Weak references are... references, to other objects (aka "targets"), but "weak" as they do not prevent those objects from being garbage-collected. In other words, weak references do not count when the Garbage Collector evaluates objects as "live" or "dead".

The following code:

```
var weak = new WeakReference<FinalizableObject>(new FinalizableObject());
GC.Collect();
```

Produces the output:

```
<namespace>.FinalizableObject initialized
```

```
<namespace>.FinalizableObject 已终结
```

尽管被 WeakReference 变量引用（在调用垃圾回收器时仍在作用域内），该对象仍然从托管堆中释放。

后果 #1：在任何时候，都不能安全地假设 WeakReference 的目标是否仍然分配在托管堆上。

后果 #2：每当程序需要访问 WeakReference 的目标时，代码应针对目标仍然分配或已被回收两种情况进行处理。访问目标的方法是 TryGetTarget：

```
var target = new object(); // 任何对象都可以作为目标
var weak = new WeakReference<object>(target); // 创建弱引用
target = null; // 取消对目标的强引用

// … 期间可能发生许多事情

// 检查目标是否仍然可用
if(weak.TryGetTarget(out target))
{
    // 使用重新初始化的 target 变量
    // 执行需要目标的操作
}
else
{
    // 当没有更多目标对象时执行某些操作
    // 此处不应使用目标变量的值
}
```

自 .Net 4.5 起，泛型版本的 WeakReference 可用。所有框架版本都提供了一个非泛型的、无类型的版本，其构建方式相同，检查方式如下：

```
var target = new object(); // 任何对象都可以作为目标
var weak = new WeakReference(target); // 创建弱引用
target = null; // 释放对目标的强引用

// … 期间可能发生许多事情

// 检查目标是否仍然可用
if (weak.IsAlive)
{
target = weak.Target;

    // 使用重新初始化的 target 变量
    // 执行需要目标的操作
}
else
{
    // 当没有更多目标对象时执行某些操作
    // 此处不应使用目标变量的值
}
```

第46.5节：Dispose() 与终结器

实现 Dispose() 方法（并将包含类声明为 IDisposable），以确保任何占用大量内存的资源在对象不再使用时尽快释放。问题在于，无法强保证 Dispose() 方法一定会被调用（与终结器不同，终结器总会在对象生命周期结束时被调用）。


```
<namespace>.FinalizableObject finalized
```

The object is freed from the managed heap despite being referenced by the WeakReference variable (still in scope when the Garbage collector was invoked).

Consequence #1: at any time, it is unsafe to assume whether a WeakReference target is still allocated on the managed heap or not.

Consequence #2: whenever a program needs to access the target of a Weakreference, code should be provided for both cases, of the target being still allocated or not. The method to access the target is TryGetTarget:

```
var target = new object(); // Any object will do as target
var weak = new WeakReference<object>(target); // Create weak reference
target = null; // Drop strong reference to the target

// ... Many things may happen in-between

// Check whether the target is still available
if(weak.TryGetTarget(out target))
{
    // Use re-initialized target variable
    // To do whatever the target is needed for
}
else
{
    // Do something when there is no more target object
    // The target variable value should not be used here
}
```

The generic version of WeakReference is available since .Net 4.5. All framework versions provide a non-generic, untyped version that is built in the same way and checked as follows:

```
var target = new object(); // Any object will do as target
var weak = new WeakReference(target); // Create weak reference
target = null; // Drop strong reference to the target

// ... Many things may happen in-between

// Check whether the target is still available
if (weak.IsAlive)
{
    target = weak.Target;

    // Use re-initialized target variable
    // To do whatever the target is needed for
}
else
{
    // Do something when there is no more target object
    // The target variable value should not be used here
}
```

Section 46.5: Dispose() vs. finalizers

Implement Dispose() method (and declare the containing class as IDisposable) as a means to ensure any memory-heavy resources are freed as soon as the object is no longer used. The "catch" is that there is no strong guarantee the the Dispose() method would ever be invoked (unlike finalizers that always get invoked at the end of the life of the object).

一种场景是程序对其显式创建的对象调用 Dispose()：

```
private void SomeFunction()
{
    // 初始化一个使用大量外部资源的对象
    var disposableObject = new ClassThatImplementsIDisposable();

    // ... 使用该对象

    // 不再使用时立即释放
disposableObject.Dispose();

    // ... 执行其他操作

    // disposableObject 变量在此处超出作用域
    // 该对象稍后将被终结（具体时间不确定）
    // 但在调用 Dispose 后，它不再持有沉重的外部资源
}
```

另一种情况是声明一个由框架实例化的类。在这种情况下，新类通常继承一个基类，例如在 MVC 中，创建一个控制器类作为 System.Web.Mvc.ControllerBase 的子类。
当基类实现了 IDisposable 接口时，这通常表明框架会正确调用 Dispose() 方法——但这仍然没有强制保证。

因此，Dispose() 不能替代终结器；相反，两者应当用于不同的目的：

- 终结器最终释放资源，以避免内存泄漏的发生Dispose() 则在资源不再需要时立即释放资源（
- 可能是相同的资源），以减轻整体内存分配的压力。

第46.6节：对象的正确释放和终结

由于 Dispose() 和终结器的目的不同，管理外部大内存资源的类应同时实现这两者。其结果是编写类时需妥善处理两种可能的场景：

- 仅当调用终结器时
- 当先调用 Dispose()，随后终结器也被调用时

一种解决方案是编写清理代码，使其运行一次或两次的结果与只运行一次相同。
可行性取决于清理的性质，例如：

- 关闭一个已经关闭的数据库连接可能不会有任何影响，因此这样做是可行的。更新某些“使用
- 计数”则很危险，调用两次而非一次会产生错误的结果。

更安全的解决方案是通过设计确保无论外部环境如何，清理代码只被调用一次。
这可以通过“经典方法”使用专用标志来实现：

```
public class DisposableFinalizable1: IDisposable
{
    private bool disposed = false;

~DisposableFinalizable1() { Cleanup(); }

    public void Dispose() { Cleanup(); }

    private void Cleanup()
```


One scenario is a program calling Dispose() on objects it explicitly creates:

```
private void SomeFunction()
{
    // Initialize an object that uses heavy external resources
    var disposableObject = new ClassThatImplementsIDisposable();

    // ... Use that object

    // Dispose as soon as no longer used
    disposableObject.Dispose();

    // ... Do other stuff

    // The disposableObject variable gets out of scope here
    // The object will be finalized later on (no guarantee when)
    // But it no longer holds to the heavy external resource after it was disposed
}
```

Another scenario is declaring a class to be instantiated by the framework. In this case the new class usually inherits a base class, for instance in MVC one creates a controller class as a subclass of System.Web.Mvc.ControllerBase. When the base class implements IDisposable interface, this is a good hint that Dispose() would be invoked properly by the framework - but again there is no strong guarantee.

Thus Dispose() is not a substitute for a finalizer; instead, the two should be used for different purposes:

- A finalizer eventually frees resources to avoid memory leaks that would occur otherwise
- Dispose() frees resources (possibly the same ones) as soon as these are no longer needed, to ease pressure on overall memory allocation.

Section 46.6: Proper disposal and finalization of objects

As Dispose() and finalizers are aimed to different purposes, a class managing external memory-heavy resources should implement both of them. The consequence is writing the class so that it handles well two possible scenarios:

- When only the finalizer is invoked
- When Dispose() is invoked first and later the finalizer is invoked as well

One solution is writing the cleanup code in such a way that running it once or twice would produce the same result as running it only once. Feasibility depends on the nature of the cleanup, for instance:

- Closing an already closed database connection would probably have no effect so it works
- Updating some "usage count" is dangerous and would produce a wrong result when called twice instead of once.

A safer solution is ensuring by design that the cleanup code is called once and only once whatever the external context. This can be achieved the "classic way" using a dedicated flag:

```
public class DisposableFinalizable1: IDisposable
{
    private bool disposed = false;

    ~DisposableFinalizable1() { Cleanup(); }

    public void Dispose() { Cleanup(); }

    private void Cleanup()
```

```
    {
        if(!disposed)
        {
            // 实际释放资源的代码写在这里，然后
disposed = true;
        }
    }
}
```

或者，垃圾回收器提供了一个特定的方法 SuppressFinalize()，允许在调用 Dispose 之后跳过终结器：

```
public class DisposableFinalizable2 : IDisposable
{
~DisposableFinalizable2() { Cleanup(); }

    public void Dispose()
    {
Cleanup();
        GC.SuppressFinalize(this);
    }

    private void Cleanup()
    {
        // 实际释放资源的代码写在这里
    }
}
```



```
    {
        if(!disposed)
        {
            // Actual code to release resources gets here, then
            disposed = true;
        }
    }
}
```

Alternately, the Garbage Collector provides a specific method SuppressFinalize() that allows skipping the finalizer after Dispose has been invoked:

```
public class DisposableFinalizable2 : IDisposable
{
    ~DisposableFinalizable2() { Cleanup(); }

    public void Dispose()
    {
        Cleanup();
        GC.SuppressFinalize(this);
    }

    private void Cleanup()
    {
        // Actual code to release resources gets here
    }
}
```

第47章：异常

第47.1节：捕获并重新抛出捕获的异常

当你想捕获异常并执行某些操作，但由于异常无法继续执行当前代码块时，你可能希望将异常重新抛出给调用栈中的下一个异常处理程序。对此，有好的做法也有不好的做法。

```
private static void AskTheUltimateQuestion()
{
    try
    {
        var x = 42;
        var y = x / (x - x); // 将抛出 DivideByZeroException

        // 重要提示：以下字符串格式中的错误是故意的
        // 目的是触发下面的 FormatException 捕获异常
Console.WriteLine("生命、宇宙及一切的秘密是 {1}", y);
    }
    catch (DivideByZeroException)
    {
        // 我们不需要异常的引用
Console.WriteLine("除以零将摧毁宇宙。");

        // 这样做是为了保留堆栈跟踪：
        throw;
    }
    catch (FormatException ex)
    {
        // 仅当需要更改要抛出的异常类型并包装内部异常时才执行此操作


        // 请记住，外部异常的堆栈跟踪将指向
        // 下一行

        // 你需要检查 InnerException 属性以获取实际引发问题的代码行的堆栈跟踪


        throw new InvalidOperationException("请注意格式字符串索引。", ex);
    }
    catch (Exception ex)
    {
       Console.WriteLine("发生了其他严重错误。异常信息: " + ex.Message);

        // 不要这样做，因为堆栈跟踪将被更改为指向
        // 此位置，而不是异常发生的位置
        // 最初抛出的位置：
        throw ex;
    }
}

static void Main()
{
    try
    {
AskTheUltimateQuestion();
    }
    catch
    {
        // 如果你不需要关于捕获的异常的任何信息，可以选择这种catch
```


Chapter 47: Exceptions

Section 47.1: Catching and rethrowing caught exceptions

When you want to catch an exception and do something, but you can't continue execution of the current block of code because of the exception, you may want to rethrow the exception to the next exception handler in the call stack. There are good ways and bad ways to do this.

```
private static void AskTheUltimateQuestion()
{
    try
    {
        var x = 42;
        var y = x / (x - x); // will throw a DivideByZeroException

        // IMPORTANT NOTE: the error in following string format IS intentional
        // and exists to throw an exception to the FormatException catch, below
        Console.WriteLine("The secret to life, the universe, and everything is {1}", y);
    }
    catch (DivideByZeroException)
    {
        // we do not need a reference to the exception
        Console.WriteLine("Dividing by zero would destroy the universe.");

        // do this to preserve the stack trace:
        throw;
    }
    catch (FormatException ex)
    {
        // only do this if you need to change the type of the Exception to be thrown
        // and wrap the inner Exception

        // remember that the stack trace of the outer Exception will point to the
        // next line

        // you'll need to examine the InnerException property to get the stack trace
        // to the line that actually started the problem

        throw new InvalidOperationException("Watch your format string indexes.", ex);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Something else horrible happened. The exception: " + ex.Message);

        // do not do this, because the stack trace will be changed to point to
        // this location instead of the location where the exception
        // was originally thrown:
        throw ex;
    }
}

static void Main()
{
    try
    {
        AskTheUltimateQuestion();
    }
    catch
    {
        // choose this kind of catch if you don't need any information about
```

```
        // 这个代码块会"吞掉"所有异常，而不是

        重新抛出它们
    }
}
```

你可以按异常类型甚至异常属性进行过滤（C# 6.0中新增加的功能，VB.NET中已存在一段时间（需要引用））：

文档/C#/新特性

第47.2节：使用finally块

finally { ... } 块在 try-finally 或 try-catch-finally 中总会执行，无论是否发生异常（除非抛出了 StackOverflowException 或调用了Environment.FailFast()）。

它可以用来安全地释放或清理在 try { ... } 块中获取的资源。

```
Console.Write("请输入文件名: ");
string filename = Console.ReadLine();

Stream fileStream = null;

try
{
fileStream = File.Open(filename);
}
catch (FileNotFoundException)
{
Console.WriteLine("文件 '{0}' 未找到。", filename);
}
finally
{
    if (fileStream != null)
    {
fileStream.Dispose();
    }
}
```

第47.3节：异常过滤器

从C# 6.0开始，可以使用when操作符来过滤异常。

这类似于使用简单的if语句，但如果when中的条件不满足，则不会展开调用栈。

示例

```
尝试
{
  // ...
}
catch (Exception e) when (e.InnerException != null) // 这里可以放任何条件。
{
  // ...
}
```

相同的信息可以在C# 6.0特性中找到：异常过滤器


```
        // the exception that was caught

        // this block "eats" all exceptions instead of rethrowing them
    }
}
```

You can filter by exception type and even by exception properties (new in C# 6.0, a bit longer available in VB.NET (citation needed)):

Documentation/C#/new features

Section 47.2: Using a finally block

The `finally { ... }` block of a `try-finally` or `try-catch-finally` will always execute, regardless of whether an exception occurred or not (except when a `StackOverflowException` has been thrown or call has been made to `Environment.FailFast()`).

It can be utilized to free or clean up resources acquired in the `try { ... }` block safely.

```
Console.Write("Please enter a filename: ");
string filename = Console.ReadLine();

Stream fileStream = null;

try
{
    fileStream = File.Open(filename);
}
catch (FileNotFoundException)
{
    Console.WriteLine("File '{0}' could not be found.", filename);
}
finally
{
    if (fileStream != null)
    {
        fileStream.Dispose();
    }
}
```

Section 47.3: Exception Filters

Since C# 6.0 exceptions can be filtered using the `when` operator.

This is similar to using a simple `if` but does not unwind the stack if the condition inside the `when` is not met.

Example

```
try
{
  // ...
}
catch (Exception e) when (e.InnerException != null) // Any condition can go in here.
{
  // ...
}
```

The same info can be found in the C# 6.0 Features here: Exception filters

第47.4节：在catch块中重新抛出异常

在catch块中，throw关键字可以单独使用，而不指定异常值，以重新抛出\刚刚捕获的异常。重新抛出异常允许原始异常继续沿着异常处理链向上传递，保留其调用堆栈或相关数据：

```
try {...}
catch (Exception ex) {
  // 注意：ex变量*未*被使用
  throw;
}
```

一个常见的反模式是改为throw ex，这会限制下一个异常处理程序对堆栈跟踪的查看范围：

```
try {...}
catch (Exception ex) {
  // 注意：ex变量被抛出
  // 该异常未来的堆栈跟踪将看不到之前的调用
  throw ex;
}
```

一般来说，不建议使用throw ex，因为未来检查堆栈跟踪的异常处理程序只能看到从throw ex开始的调用。通过省略ex变量，仅使用throw关键字，原始异常将会“冒泡”。

第47.5节：从不同方法抛出异常时保留其信息

有时你可能想捕获一个异常，并从不同的线程或方法中抛出它，同时保留原始异常堆栈。这可以通过ExceptionDispatchInfo来实现：

```
using System.Runtime.ExceptionServices;

void Main()
{
ExceptionDispatchInfo capturedException = null;
    try
    {
        throw new Exception();
    }
    catch (Exception ex)
    {
capturedException = ExceptionDispatchInfo.Capture(ex);
    }

Foo(capturedException);
}

void Foo(ExceptionDispatchInfo exceptionDispatchInfo)
{
    // 执行操作

    if (capturedException != null)
    {
        // 异常堆栈跟踪将显示它是从 Main() 抛出的，而不是从 Foo() 抛出的
exceptionDispatchInfo.Throw();
    }
```


Section 47.4: Rethrowing an exception within a catch block

Within a `catch` block the `throw` keyword can be used on its own, without specifying an exception value, to *rethrow* the exception which was just caught. Rethrowing an exception allows the original exception to continue up the exception handling chain, preserving its call stack or associated data:

```
try {...}
catch (Exception ex) {
  // Note: the ex variable is *not* used
  throw;
}
```

A common anti-pattern is to instead `throw ex`, which has the effect of limiting the next exception handler's view of the stack trace:

```
try {...}
catch (Exception ex) {
  // Note: the ex variable is thrown
  //  future stack traces of the exception will not see prior calls
  throw ex;
}
```

In general using `throw ex` isn't desirable, as future exception handlers which inspect the stack trace will only be able to see calls as far back as `throw ex`. By omitting the `ex` variable, and using the `throw` keyword alone the original exception will "bubble-up".

Section 47.5: Throwing an exception from a different method while preserving its information

Occasionally you'd want to catch an exception and throw it from a different thread or method while preserving the original exception stack. This can be done with `ExceptionDispatchInfo`:

```
using System.Runtime.ExceptionServices;

void Main()
{
    ExceptionDispatchInfo capturedException = null;
    try
    {
        throw new Exception();
    }
    catch (Exception ex)
    {
        capturedException = ExceptionDispatchInfo.Capture(ex);
    }

    Foo(capturedException);
}

void Foo(ExceptionDispatchInfo exceptionDispatchInfo)
{
    // Do stuff

    if (capturedException != null)
    {
        // Exception stack trace will show it was thrown from Main() and not from Foo()
        exceptionDispatchInfo.Throw();
    }
```

```
}
```

第47.6节：捕获异常

代码可以且应该在异常情况下抛出异常。示例包括：

- 尝试读取超出流末尾
- 没有必要的权限访问文件
- 尝试执行无效操作，例如除以零
- 从互联网下载文件时发生超时

调用者可以通过“捕获”这些异常来处理它们，并且只有在以下情况下才应这样做：

- 它实际上可以解决异常情况或适当恢复，或者；
- 它可以为异常提供额外的上下文信息，如果需要重新抛出异常，这些信息将很有用（重新抛出的异常会被调用堆栈上层的异常处理程序捕获）

需要注意的是，如果打算让异常在更高层处理，选择不捕获异常是完全有效的。

捕获异常是通过将可能抛出异常的代码包裹在`try { ... }`块中实现的，并在`catch (ExceptionType) { ... }`块中捕获它能够处理的异常，如下所示：

```
Console.Write("请输入文件名: ");
string filename = Console.ReadLine();

Stream fileStream;

try
{
fileStream = File.Open(filename);
}
catch (FileNotFoundException)
{
Console.WriteLine("文件 '{0}' 未找到。", filename);
}
```



```
}
```

Section 47.6: Catching an exception

Code can and should throw exceptions in exceptional circumstances. Examples of this include:

- Attempting to read past the end of a stream
- Not having necessary permissions to access a file
- Attempting to perform an invalid operation, such as dividing by zero
- A timeout occurring when downloading a file from the internet

The caller can handle these exceptions by "catching" them, and should only do so when:

- It can actually resolve the exceptional circumstance or recover appropriately, or;
- It can provide additional context to the exception that would be useful if the exception needs to be re-thrown (re-thrown exceptions are caught by exception handlers further up the call stack)

It should be noted that choosing *not* to catch an exception is perfectly valid if the intention is for it to be handled at a higher level.

Catching an exception is done by wrapping the potentially-throwing code in a `try { ... }` block as follows, and catching the exceptions it's able to handle in a `catch (ExceptionType) { ... }` block:

```
Console.Write("Please enter a filename: ");
string filename = Console.ReadLine();

Stream fileStream;

try
{
    fileStream = File.Open(filename);
}
catch (FileNotFoundException)
{
    Console.WriteLine("File '{0}' could not be found.", filename);
}
```

第48章：System.Diagnostics

第48.1节：运行Shell命令

```
string strCmdText = "/C copy /b Image1.jpg + Archive.rar Image2.jpg";
System.Diagnostics.Process.Start("CMD.exe",strCmdText);
```

这是为了隐藏cmd窗口。

```
System.Diagnostics.Process process = new System.Diagnostics.Process();
System.Diagnostics.ProcessStartInfo startInfo = new System.Diagnostics.ProcessStartInfo();
startInfo.WindowStyle = System.Diagnostics.ProcessWindowStyle.Hidden;
startInfo.FileName = "cmd.exe";
startInfo.Arguments = "/C copy /b Image1.jpg + Archive.rar Image2.jpg";
process.StartInfo = startInfo;
process.Start();
```

第48.2节：向CMD发送命令并接收输出

此方法允许向Cmd.exe发送command，并返回标准输出（包括标准错误）作为字符串：

```
private static string SendCommand(string command)
{
    var cmdOut = string.Empty;

    var startInfo = new ProcessStartInfo("cmd", command)
    {
WorkingDirectory = @"C:\Windows\System32", // 调用时的工作目录
        WindowStyle = ProcessWindowStyle.Hidden,    // 隐藏窗口
UseShellExecute = false,                    // 不使用操作系统外壳启动进程
        CreateNoWindow = true,                      // 在新窗口中启动进程
RedirectStandardOutput = true,              // 需要获取标准输出
        RedirectStandardError = true                // 需要获取标准错误输出
    };

    var p = new Process {StartInfo = startInfo};

p.Start();

p.OutputDataReceived += (x, y) => cmdOut += y.Data;
    p.ErrorDataReceived += (x, y) => cmdOut += y.Data;
    p.BeginOutputReadLine();
p.BeginErrorReadLine();
    p.WaitForExit();
    return cmdOut;
}
```

用法

```
var servername = "SVR-01.domain.co.za";
var currentUsers = SendCommand($"/C QUERY USER /SERVER:{servername}")
```

输出

> string currentUsers = "USERNAME SESSIONNAME ID STATE IDLE TIME LOGON TIME Joe.Bloggs ica-cgp#0 2

Chapter 48: System.Diagnostics

Section 48.1: Run shell commands

```
string strCmdText = "/C copy /b Image1.jpg + Archive.rar Image2.jpg";
System.Diagnostics.Process.Start("CMD.exe",strCmdText);
```

This is to hide the cmd window.

```
System.Diagnostics.Process process = new System.Diagnostics.Process();
System.Diagnostics.ProcessStartInfo startInfo = new System.Diagnostics.ProcessStartInfo();
startInfo.WindowStyle = System.Diagnostics.ProcessWindowStyle.Hidden;
startInfo.FileName = "cmd.exe";
startInfo.Arguments = "/C copy /b Image1.jpg + Archive.rar Image2.jpg";
process.StartInfo = startInfo;
process.Start();
```

Section 48.2: Send Command to CMD and Receive Output

This method allows a `command` to be sent to `Cmd.exe`, and returns the standard output (including standard error) as a string:

```
private static string SendCommand(string command)
{
    var cmdOut = string.Empty;

    var startInfo = new ProcessStartInfo("cmd", command)
    {
        WorkingDirectory = @"C:\Windows\System32", // Directory to make the call from
        WindowStyle = ProcessWindowStyle.Hidden,   // Hide the window
        UseShellExecute = false,                   // Do not use the OS shell to start the process
        CreateNoWindow = true,                     // Start the process in a new window
        RedirectStandardOutput = true,             // This is required to get STDOUT
        RedirectStandardError = true               // This is required to get STDERR
    };

    var p = new Process {StartInfo = startInfo};

    p.Start();

    p.OutputDataReceived += (x, y) => cmdOut += y.Data;
    p.ErrorDataReceived += (x, y) => cmdOut += y.Data;
    p.BeginOutputReadLine();
    p.BeginErrorReadLine();
    p.WaitForExit();
    return cmdOut;
}
```

Usage

```
var servername = "SVR-01.domain.co.za";
var currentUsers = SendCommand($"/C QUERY USER /SERVER:{servername}")
```

Output

> string currentUsers = "USERNAME SESSIONNAME ID STATE IDLE TIME LOGON TIME Joe.Bloggs ica-cgp#0 2

> Active 24692+13:29 25/07/2016 07:50 Jim.McFlannegan ica-cgp#1 3 Active . 25/07/2016 08:33 Andy.McAnderson ica-cgp#2 4 Active . 25/07/2016 08:54 John.Smith ica-cgp#4 5 Active 14 25/07/2016 08:57 Bob.Bobbington ica-cgp#5 6 Active 24692+13:29 25/07/2016 09:05 Tim.Tom ica-cgp#6 7 Active . 25/07/2016 09:08 Bob.Joges ica-cgp#7 8 Active 24692+13:29 25/07/2016 09:13"

在某些情况下，对相关服务器的访问可能仅限于特定用户。如果您拥有该用户的登录凭据，则可以通过此方法发送查询：

```
private static string SendCommand(string command)
{
    var cmdOut = string.Empty;

    var startInfo = new ProcessStartInfo("cmd", command)
    {
WorkingDirectory = @"C:\Windows\System32",
        WindowStyle = ProcessWindowStyle.Hidden,    // 这实际上无法与
"runas" 一起使用——控制台窗口仍然会出现！
UseShellExecute = false,
        CreateNoWindow = true,
        RedirectStandardOutput = true,
        RedirectStandardError = true,

        Verb = "runas",
Domain = "doman1.co.za",
        UserName = "administrator",
        Password = GetPassword()
    };

    var p = new Process {StartInfo = startInfo};

p.Start();

p.OutputDataReceived += (x, y) => cmdOut += y.Data;
    p.ErrorDataReceived += (x, y) => cmdOut += y.Data;
    p.BeginOutputReadLine();
p.BeginErrorReadLine();
    p.WaitForExit();
    return cmdOut;
}
```

获取密码：

```
static SecureString 获取密码()
{
    var plainText = "password123";
    var ss = new SecureString();
    foreach (char c in plainText)
    {
ss.AppendChar(c);
    }

    return ss;
}
```

注意事项

上述两种方法都会返回 STDOUT 和 STDERR 的连接结果，因为 `OutputDataReceived` 和 `ErrorDataReceived` 都附加到同一个变量 - `cmdOut`。

> Active 24692+13:29 25/07/2016 07:50 Jim.McFlannegan ica-cgp#1 3 Active . 25/07/2016 08:33 Andy.McAnderson ica-cgp#2 4 Active . 25/07/2016 08:54 John.Smith ica-cgp#4 5 Active 14 25/07/2016 08:57 Bob.Bobbington ica-cgp#5 6 Active 24692+13:29 25/07/2016 09:05 Tim.Tom ica-cgp#6 7 Active . 25/07/2016 09:08 Bob.Joges ica-cgp#7 8 Active 24692+13:29 25/07/2016 09:13"

On some occasions, access to the server in question may be restricted to certain users. If you have the login credentials for this user, then it is possible to send queries with this method:

```
private static string SendCommand(string command)
{
    var cmdOut = string.Empty;

    var startInfo = new ProcessStartInfo("cmd", command)
    {
        WorkingDirectory = @"C:\Windows\System32",
        WindowStyle = ProcessWindowStyle.Hidden,    // This does not actually work in conjunction
with "runas" - the console window will still appear!
        UseShellExecute = false,
        CreateNoWindow = true,
        RedirectStandardOutput = true,
        RedirectStandardError = true,

        Verb = "runas",
        Domain = "doman1.co.za",
        UserName = "administrator",
        Password = GetPassword()
    };

    var p = new Process {StartInfo = startInfo};

    p.Start();

    p.OutputDataReceived += (x, y) => cmdOut += y.Data;
    p.ErrorDataReceived += (x, y) => cmdOut += y.Data;
    p.BeginOutputReadLine();
    p.BeginErrorReadLine();
    p.WaitForExit();
    return cmdOut;
}
```

Getting the password:

```
static SecureString GetPassword()
{
    var plainText = "password123";
    var ss = new SecureString();
    foreach (char c in plainText)
    {
        ss.AppendChar(c);
    }

    return ss;
}
```

Notes

Both of the above methods will return a concatenation of STDOUT and STDERR, as `OutputDataReceived` and `ErrorDataReceived` are both appending to the same variable - `cmdOut`.

第48.3节：秒表

此示例展示了如何使用 Stopwatch 来对一段代码进行基准测试。

```
using System;
using System.Diagnostics;

public class Benchmark : IDisposable
{
    private Stopwatch sw;

    public Benchmark()
    {
sw = Stopwatch.StartNew();
    }

    public void Dispose()
    {
sw.Stop();
Console.WriteLine(sw.Elapsed);
    }
}

public class Program
{
    public static void Main()
    {
        using (var bench = new Benchmark())
        {
Console.WriteLine("Hello World");
        }
    }
}
```


Section 48.3: Stopwatch

This example shows how `Stopwatch` can be used to benchmark a block of code.

```
using System;
using System.Diagnostics;

public class Benchmark : IDisposable
{
    private Stopwatch sw;

    public Benchmark()
    {
        sw = Stopwatch.StartNew();
    }

    public void Dispose()
    {
        sw.Stop();
        Console.WriteLine(sw.Elapsed);
    }
}

public class Program
{
    public static void Main()
    {
        using (var bench = new Benchmark())
        {
            Console.WriteLine("Hello World");
        }
    }
}
```

第49章：加密 / 密码学

第49.1节：使用密码学进行加密和解密（AES）

解密代码

```
    public static string Decrypt(string cipherText)
    {
        if (cipherText == null)
            return null;

        byte[] cipherBytes = Convert.FromBase64String(cipherText);
        using (Aes encryptor = Aes.Create())
        {
Rfc2898DeriveBytes pdb = new Rfc2898DeriveBytes(CryptKey, new byte[] { 0x49, 0x76,
0x61, 0x6e, 0x20, 0x4d, 0x65, 0x64, 0x76, 0x65, 0x64, 0x65, 0x76 });
encryptor.Key = pdb.GetBytes(32);
            encryptor.IV = pdb.GetBytes(16);

            using (MemoryStream ms = new MemoryStream())
            {
                using (CryptoStream cs = new CryptoStream(ms, encryptor.CreateDecryptor(),
CryptoStreamMode.Write))
                {
cs.Write(cipherBytes, 0, cipherBytes.Length);
                    cs.Close();
                }

cipherText = Encoding.Unicode.GetString(ms.ToArray());
            }
        }

        return cipherText;
    }
```

加密代码

```
    public static string Encrypt(string cipherText)
    {
        if (cipherText == null)
            return null;

        byte[] clearBytes = Encoding.Unicode.GetBytes(cipherText);
        using (Aes encryptor = Aes.Create())
        {
Rfc2898DeriveBytes pdb = new Rfc2898DeriveBytes(CryptKey, new byte[] { 0x49, 0x76,
0x61, 0x6e, 0x20, 0x4d, 0x65, 0x64, 0x76, 0x65, 0x64, 0x65, 0x76 });
encryptor.Key = pdb.GetBytes(32);
            encryptor.IV = pdb.GetBytes(16);

            using (MemoryStream ms = new MemoryStream())
            {
                using (CryptoStream cs = new CryptoStream(ms, encryptor.CreateEncryptor(),
CryptoStreamMode.Write))
                {
cs.Write(clearBytes, 0, clearBytes.Length);
                    cs.Close();
                }
```


Chapter 49: Encryption / Cryptography

Section 49.1: Encryption and Decryption using Cryptography (AES)

Decryption Code

```
    public static string Decrypt(string cipherText)
    {
        if (cipherText == null)
            return null;

        byte[] cipherBytes = Convert.FromBase64String(cipherText);
        using (Aes encryptor = Aes.Create())
        {
            Rfc2898DeriveBytes pdb = new Rfc2898DeriveBytes(CryptKey, new byte[] { 0x49, 0x76,
0x61, 0x6e, 0x20, 0x4d, 0x65, 0x64, 0x76, 0x65, 0x64, 0x65, 0x76 });
            encryptor.Key = pdb.GetBytes(32);
            encryptor.IV = pdb.GetBytes(16);

            using (MemoryStream ms = new MemoryStream())
            {
                using (CryptoStream cs = new CryptoStream(ms, encryptor.CreateDecryptor(),
CryptoStreamMode.Write))
                {
                    cs.Write(cipherBytes, 0, cipherBytes.Length);
                    cs.Close();
                }

                cipherText = Encoding.Unicode.GetString(ms.ToArray());
            }
        }

        return cipherText;
    }
```

Encryption Code

```
    public static string Encrypt(string cipherText)
    {
        if (cipherText == null)
            return null;

        byte[] clearBytes = Encoding.Unicode.GetBytes(cipherText);
        using (Aes encryptor = Aes.Create())
        {
            Rfc2898DeriveBytes pdb = new Rfc2898DeriveBytes(CryptKey, new byte[] { 0x49, 0x76,
0x61, 0x6e, 0x20, 0x4d, 0x65, 0x64, 0x76, 0x65, 0x64, 0x65, 0x76 });
            encryptor.Key = pdb.GetBytes(32);
            encryptor.IV = pdb.GetBytes(16);

            using (MemoryStream ms = new MemoryStream())
            {
                using (CryptoStream cs = new CryptoStream(ms, encryptor.CreateEncryptor(),
CryptoStreamMode.Write))
                {
                    cs.Write(clearBytes, 0, clearBytes.Length);
                    cs.Close();
                }
```

```
cipherText = Convert.ToBase64String(ms.ToArray());
            }
        }
        return cipherText;
    }
```

用法

```
var textToEncrypt = "TestEncrypt";

var encrypted = Encrypt(textToEncrypt);

var decrypted = Decrypt(encrypted);
```

第49.2节：RijndaelManaged

所需命名空间：System.Security.Cryptography

```
private class Encryption {

          private const string SecretKey = "topSecretKeyusedforEncryptions";

          private const string SecretIv = "secretVectorHere";

          public string Encrypt(string data) {
            return string.IsNullOrEmpty(data) ? data :
Convert.ToBase64String(this.EncryptStringToBytesAes(data, this.GetCryptographyKey(),
this.GetCryptographyIv()));
          }

          public string Decrypt(string data) {
            return string.IsNullOrEmpty(data) ? data :
this.DecryptStringFromBytesAes(Convert.FromBase64String(data), this.GetCryptographyKey(),
this.GetCryptographyIv());
          }

          private byte[] GetCryptographyKey() {
            return Encoding.ASCII.GetBytes(SecretKey.Replace('e', '!'));
          }

          private byte[] GetCryptographyIv() {
            return Encoding.ASCII.GetBytes(SecretIv.Replace('r', '!'));
          }

          private byte[] EncryptStringToBytesAes(string plainText, byte[] key, byte[] iv) {
            MemoryStream encrypt;
RijndaelManaged aesAlg = null;
            try {
aesAlg = new RijndaelManaged {
                Key = key,
IV = iv
              };
              var encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
              encrypt = new MemoryStream();
              using (var csEncrypt = new CryptoStream(encrypt, encryptor, CryptoStreamMode.Write))
{
                using (var swEncrypt = new StreamWriter(csEncrypt)) {
                  swEncrypt.Write(plainText);
                }
              }
```



```
                cipherText = Convert.ToBase64String(ms.ToArray());
            }
        }
        return cipherText;
    }
```

Usage

```
var textToEncrypt = "TestEncrypt";

var encrypted = Encrypt(textToEncrypt);

var decrypted = Decrypt(encrypted);
```

Section 49.2: RijndaelManaged

Required Namespace: `System.Security.Cryptography`

```
private class Encryption {

          private const string SecretKey = "topSecretKeyusedforEncryptions";

          private const string SecretIv = "secretVectorHere";

          public string Encrypt(string data) {
            return string.IsNullOrEmpty(data) ? data :
Convert.ToBase64String(this.EncryptStringToBytesAes(data, this.GetCryptographyKey(),
this.GetCryptographyIv()));
          }

          public string Decrypt(string data) {
            return string.IsNullOrEmpty(data) ? data :
this.DecryptStringFromBytesAes(Convert.FromBase64String(data), this.GetCryptographyKey(),
this.GetCryptographyIv());
          }

          private byte[] GetCryptographyKey() {
            return Encoding.ASCII.GetBytes(SecretKey.Replace('e', '!'));
          }

          private byte[] GetCryptographyIv() {
            return Encoding.ASCII.GetBytes(SecretIv.Replace('r', '!'));
          }

          private byte[] EncryptStringToBytesAes(string plainText, byte[] key, byte[] iv) {
            MemoryStream encrypt;
            RijndaelManaged aesAlg = null;
            try {
              aesAlg = new RijndaelManaged {
                Key = key,
                IV = iv
              };
              var encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
              encrypt = new MemoryStream();
              using (var csEncrypt = new CryptoStream(encrypt, encryptor, CryptoStreamMode.Write))
{
                using (var swEncrypt = new StreamWriter(csEncrypt)) {
                  swEncrypt.Write(plainText);
                }
              }
```

```
            } finally {
aesAlg?.Clear();
            }
            return encrypt.ToArray();
          }

          private string DecryptStringFromBytesAes(byte[] cipherText, byte[] key, byte[] iv) {
            RijndaelManaged aesAlg = null;
            string plaintext;
            try {
aesAlg = new RijndaelManaged {
                Key = key,
IV = iv
              };
              var decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);
              using (var msDecrypt = new MemoryStream(cipherText)) {
                using (var csDecrypt = new CryptoStream(msDecrypt, decryptor,
CryptoStreamMode.Read)) {
                  using (var srDecrypt = new StreamReader(csDecrypt))
                    plaintext = srDecrypt.ReadToEnd();
                }
              }
            } finally {
aesAlg?.Clear();
            }
            return plaintext;
          }
        }
```

用法

```
var textToEncrypt = "hello World";

 var encrypted = new Encryption().Encrypt(textToEncrypt); //-> zBmW+FUxOvdbpOGm9Ss/vQ==

 var decrypted = new Encryption().Decrypt(encrypted); //-> hello World
```

注意：

- Rijndael 是标准对称加密算法 AES 的前身。

第49.3节：使用 AES 加密和解密数据（C#）

```
using System;
using System.IO;
using System.Security.Cryptography;

namespace Aes_Example
{
    class AesExample
    {
        public static void Main()
        {
            try
            {
                string original = "这里有一些要加密的数据！";

                // 创建 Aes 类的新实例。
                // 这将生成新的密钥和初始化向量（IV）。
                using (Aes myAes = Aes.Create())
```



```
            } finally {
              aesAlg?.Clear();
            }
            return encrypt.ToArray();
          }

          private string DecryptStringFromBytesAes(byte[] cipherText, byte[] key, byte[] iv) {
            RijndaelManaged aesAlg = null;
            string plaintext;
            try {
              aesAlg = new RijndaelManaged {
                Key = key,
                IV = iv
              };
              var decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);
              using (var msDecrypt = new MemoryStream(cipherText)) {
                using (var csDecrypt = new CryptoStream(msDecrypt, decryptor,
CryptoStreamMode.Read)) {
                  using (var srDecrypt = new StreamReader(csDecrypt))
                    plaintext = srDecrypt.ReadToEnd();
                }
              }
            } finally {
              aesAlg?.Clear();
            }
            return plaintext;
          }
        }
```

Usage

```
var textToEncrypt = "hello World";

 var encrypted = new Encryption().Encrypt(textToEncrypt); //-> zBmW+FUxOvdbpOGm9Ss/vQ==

 var decrypted = new Encryption().Decrypt(encrypted); //-> hello World
```

Note:

- Rijndael is the predecessor of the standard symmetric cryptographic algorithm AES.

Section 49.3: Encrypt and decrypt data using AES (in C#)

```
using System;
using System.IO;
using System.Security.Cryptography;

namespace Aes_Example
{
    class AesExample
    {
        public static void Main()
        {
            try
            {
                string original = "Here is some data to encrypt!";

                // Create a new instance of the Aes class.
                // This generates a new key and initialization vector (IV).
                using (Aes myAes = Aes.Create())
```

```
                {
                    // 将字符串加密为字节数组。
                    byte[] encrypted = EncryptStringToBytes_Aes(original,
                                                                myAes.Key,
                                                                myAes.IV);

                    // 将字节解密为字符串。
                    string roundtrip = DecryptStringFromBytes_Aes(encrypted,
                                                                  myAes.Key,
                                                                  myAes.IV);

                    // 显示原始数据和解密后的数据。
Console.WriteLine("Original:   {0}", original);
                    Console.WriteLine("Round Trip: {0}", roundtrip);
                }
            }
            catch (Exception e)
            {
Console.WriteLine("Error: {0}", e.Message);
            }
        }

        static byte[] EncryptStringToBytes_Aes(string plainText, byte[] Key, byte[] IV)
        {
            // 检查参数。
            if (plainText == null || plainText.Length <= 0)
                throw new ArgumentNullException("plainText");
            if (Key == null || Key.Length <= 0)
                throw new ArgumentNullException("Key");
            if (IV == null || IV.Length <= 0)
                throw new ArgumentNullException("IV");

            byte[] encrypted;

            // 使用指定的密钥和初始化向量创建一个Aes对象。
            using (Aes aesAlg = Aes.Create())
            {
aesAlg.Key = Key;
                aesAlg.IV = IV;

                // 创建一个解密器以执行流转换。
ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key,
                                                                    aesAlg.IV);

                // 创建用于加密的流。
                using (MemoryStream msEncrypt = new MemoryStream())
                {
                    using (CryptoStream csEncrypt = new CryptoStream(msEncrypt,
                                                    encryptor,
                                                                     CryptoStreamMode.Write))
                    {
                        using (StreamWriter swEncrypt = new StreamWriter(csEncrypt))
                        {
                            // 将所有数据写入流。
swEncrypt.Write(plainText);
                        }

encrypted = msEncrypt.ToArray();
                    }
                }
            }
```



```
                {
                    // Encrypt the string to an array of bytes.
                    byte[] encrypted = EncryptStringToBytes_Aes(original,
                                                                myAes.Key,
                                                                myAes.IV);

                    // Decrypt the bytes to a string.
                    string roundtrip = DecryptStringFromBytes_Aes(encrypted,
                                                                  myAes.Key,
                                                                  myAes.IV);

                    //Display the original data and the decrypted data.
                    Console.WriteLine("Original:   {0}", original);
                    Console.WriteLine("Round Trip: {0}", roundtrip);
                }
            }
            catch (Exception e)
            {
                Console.WriteLine("Error: {0}", e.Message);
            }
        }

        static byte[] EncryptStringToBytes_Aes(string plainText, byte[] Key, byte[] IV)
        {
            // Check arguments.
            if (plainText == null || plainText.Length <= 0)
                throw new ArgumentNullException("plainText");
            if (Key == null || Key.Length <= 0)
                throw new ArgumentNullException("Key");
            if (IV == null || IV.Length <= 0)
                throw new ArgumentNullException("IV");

            byte[] encrypted;

            // Create an Aes object with the specified key and IV.
            using (Aes aesAlg = Aes.Create())
            {
                aesAlg.Key = Key;
                aesAlg.IV = IV;

                // Create a decrytor to perform the stream transform.
                ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key,
                                                                    aesAlg.IV);

                // Create the streams used for encryption.
                using (MemoryStream msEncrypt = new MemoryStream())
                {
                    using (CryptoStream csEncrypt = new CryptoStream(msEncrypt,
                                                                     encryptor,
                                                                     CryptoStreamMode.Write))
                    {
                        using (StreamWriter swEncrypt = new StreamWriter(csEncrypt))
                        {
                            //Write all data to the stream.
                            swEncrypt.Write(plainText);
                        }

                        encrypted = msEncrypt.ToArray();
                    }
                }
            }
```

```
            // 从内存流返回加密的字节。
            return encrypted;
        }

        static string DecryptStringFromBytes_Aes(byte[] cipherText, byte[] Key, byte[] IV)
        {
            // 检查参数。
            if (cipherText == null || cipherText.Length <= 0)
                throw new ArgumentNullException("cipherText");
            if (Key == null || Key.Length <= 0)
                throw new ArgumentNullException("Key");
            if (IV == null || IV.Length <= 0)
                throw new ArgumentNullException("IV");

            // 声明用于保存解密文本的字符串。
            string plaintext = null;

            // 使用指定的密钥和初始化向量创建一个Aes对象。
            using (Aes aesAlg = Aes.Create())
            {
aesAlg.Key = Key;
                aesAlg.IV = IV;

                // 创建一个解密器以执行流转换。
ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key,
                                                                    aesAlg.IV);

                // 创建用于解密的流。
                使用(MemoryStream msDecrypt = new MemoryStream(cipherText))
                {
                    使用 (CryptoStream csDecrypt = new CryptoStream(msDecrypt,
                                                                    decryptor,
                                                                CryptoStreamMode.Read))
                    {
                        使用 (StreamReader srDecrypt = new StreamReader(csDecrypt))
                        {

                            // 从解密流中读取解密后的字节
                            // 并将它们放入字符串中。
                        明文 = srDecrypt.ReadToEnd();
                        }
                    }
                }
            }

            return plaintext;
        }
    }
}
```

此示例来自MSDN。

这是一个控制台演示应用程序，展示了如何使用标准AES加密字符串，以及如何随后解密它。

(AES = 高级加密标准，由美国制定的电子数据加密规范
国家标准与技术研究院（NIST）于2001年发布，至今仍是对称加密的事实标准)

备注：


```
            // Return the encrypted bytes from the memory stream.
            return encrypted;
        }

        static string DecryptStringFromBytes_Aes(byte[] cipherText, byte[] Key, byte[] IV)
        {
            // Check arguments.
            if (cipherText == null || cipherText.Length <= 0)
                throw new ArgumentNullException("cipherText");
            if (Key == null || Key.Length <= 0)
                throw new ArgumentNullException("Key");
            if (IV == null || IV.Length <= 0)
                throw new ArgumentNullException("IV");

            // Declare the string used to hold the decrypted text.
            string plaintext = null;

            // Create an Aes object with the specified key and IV.
            using (Aes aesAlg = Aes.Create())
            {
                aesAlg.Key = Key;
                aesAlg.IV = IV;

                // Create a decrytor to perform the stream transform.
                ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key,
                                                                    aesAlg.IV);

                // Create the streams used for decryption.
                using (MemoryStream msDecrypt = new MemoryStream(cipherText))
                {
                    using (CryptoStream csDecrypt = new CryptoStream(msDecrypt,
                                                                     decryptor,
                                                                     CryptoStreamMode.Read))
                    {
                        using (StreamReader srDecrypt = new StreamReader(csDecrypt))
                        {

                            // Read the decrypted bytes from the decrypting stream
                            // and place them in a string.
                            plaintext = srDecrypt.ReadToEnd();
                        }
                    }
                }
            }

            return plaintext;
        }
    }
}
```

This example is from MSDN.

It is a console demo application, showing how to encrypt a string by using the standard **AES** encryption, and how to decrypt it afterwards.

(**AES = Advanced Encryption Standard**, a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001 which is still the de-facto standard for symmetric encryption)

Notes:

- 在实际的加密场景中，您需要选择合适的密码模式（可以通过从CipherMode枚举中选择一个值赋给Mode属性）。切勿使用CipherMode.ECB（电子密码本模式），因为这会产生一个弱的密码流。
- 要创建一个好的（而非弱的）Key，可以使用加密随机生成器，或者使用上面的示例（从密码创建密钥）。推荐的KeySize是256位。支持的密钥大小可通过LegalKeySizes属性获得。
- 要初始化初始化向量IV，可以使用示例中所示的盐值（随机盐值）。支持的块大小可通过SupportedBlockSizes
- 属性获得，块大小可以通过BlockSize属性设置。

用法：参见Main()方法。

第49.4节：从密码/随机盐值创建密钥（C#示例）

```
using System;
using System.Security.Cryptography;
using System.Text;

public class PasswordDerivedBytesExample
{
    public static void Main(String[] args)
    {
        // 从用户获取密码。
Console.WriteLine("请输入密码以生成密钥：" );

        byte[] pwd = Encoding.Unicode.GetBytes(Console.ReadLine());

        byte[] salt = CreateRandomSalt(7);

        // 创建一个 TripleDESCryptoServiceProvider 对象。
TripleDESCryptoServiceProvider tdes = new TripleDESCryptoServiceProvider();

        try
        {
Console.WriteLine("正在使用 PasswordDeriveBytes 创建密钥..." );

            // 创建一个 PasswordDeriveBytes 对象，然后从密码和盐创建
            // 一个 TripleDES 密钥。
PasswordDeriveBytes pdb = new PasswordDeriveBytes(pwd, salt);

            // 创建密钥并将其设置到 TripleDESCryptoServiceProvider
            对象的 Key 属性中。
tdes.Key = pdb.CryptDeriveKey("TripleDES", "SHA1", 192, tdes.IV);

            Console.WriteLine("操作完成。" );
        }
        catch (Exception e)
        {
Console.WriteLine(e.Message);
        }
        finally
        {
            // 清除缓冲区
ClearBytes(pwd);
            ClearBytes(salt);
```


- In a real encryption scenario, you need to choose a proper cipher mode (can be assigned to the `Mode` property by selecting a value from the `CipherMode` enumeration). **Never** use the `CipherMode.ECB` (electronic codebook mode), since this procuces a weak cypher stream
- To create a good (and not a weak) `Key`, either use a cryptographic random generator or use the example above (**Create a Key from a Password**). The recommended **KeySize** is 256 bit. Supported key sizes are available via the `LegalKeySizes` property.
- To initialize the initialization vector `IV`, you can use a SALT as shown in the example above (**Random SALT**)
- Supported block sizes are available via the `SupportedBlockSizes` property, the block size can be assigned via the `BlockSize` property

Usage: see Main() method.

Section 49.4: Create a Key from a Password / Random SALT (in C#)

```
using System;
using System.Security.Cryptography;
using System.Text;

public class PasswordDerivedBytesExample
{
    public static void Main(String[] args)
    {
        // Get a password from the user.
        Console.WriteLine("Enter a password to produce a key:");

        byte[] pwd = Encoding.Unicode.GetBytes(Console.ReadLine());

        byte[] salt = CreateRandomSalt(7);

        // Create a TripleDESCryptoServiceProvider object.
        TripleDESCryptoServiceProvider tdes = new TripleDESCryptoServiceProvider();

        try
        {
            Console.WriteLine("Creating a key with PasswordDeriveBytes...");

            // Create a PasswordDeriveBytes object and then create
            // a TripleDES key from the password and salt.
            PasswordDeriveBytes pdb = new PasswordDeriveBytes(pwd, salt);

            // Create the key and set it to the Key property
            // of the TripleDESCryptoServiceProvider object.
            tdes.Key = pdb.CryptDeriveKey("TripleDES", "SHA1", 192, tdes.IV);

            Console.WriteLine("Operation complete.");
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
        finally
        {
            // Clear the buffers
            ClearBytes(pwd);
            ClearBytes(salt);
```

```
            // 清除密钥。
tdes.Clear();
        }

Console.ReadLine();
    }

    #region 辅助方法

    /// <summary>
    /// 生成指定长度的随机盐值。
    /// </summary>
    public static byte[] CreateRandomSalt(int length)
    {
        // 创建缓冲区
        byte[] randBytes;

        if (length >= 1)
        {
randBytes = new byte[length];
        }
        否则
        {
randBytes = new byte[1];
        }

        // 创建一个新的 RNGCryptoServiceProvider。
RNGCryptoServiceProvider rand = new RNGCryptoServiceProvider();

        // 用随机字节填充缓冲区。
rand.GetBytes(randBytes);

        // 返回字节数组。
        return randBytes;
    }

    /// <summary>
    /// 清除缓冲区中的字节，以防止它们以后被从内存中读取。
    /// </summary>
    public static void ClearBytes(byte[] buffer)
    {
        // 检查参数。
        if (buffer == null)
        {
            throw new ArgumentNullException("buffer");
        }

        // 将缓冲区中的每个字节设置为0。
        for (int x = 0; x < buffer.Length; x++)
        {
buffer[x] = 0;
        }
    }

    #endregion
}
```

此示例摘自MSDN。

这是一个控制台演示，展示了如何基于用户定义的密码创建安全密钥，以及如何基于加密随机生成器创建随机盐值（SALT）。


```
            // Clear the key.
            tdes.Clear();
        }

        Console.ReadLine();
    }

    #region Helper methods

    /// <summary>
    /// Generates a random salt value of the specified length.
    /// </summary>
    public static byte[] CreateRandomSalt(int length)
    {
        // Create a buffer
        byte[] randBytes;

        if (length >= 1)
        {
            randBytes = new byte[length];
        }
        else
        {
            randBytes = new byte[1];
        }

        // Create a new RNGCryptoServiceProvider.
        RNGCryptoServiceProvider rand = new RNGCryptoServiceProvider();

        // Fill the buffer with random bytes.
        rand.GetBytes(randBytes);

        // return the bytes.
        return randBytes;
    }

    /// <summary>
    /// Clear the bytes in a buffer so they can't later be read from memory.
    /// </summary>
    public static void ClearBytes(byte[] buffer)
    {
        // Check arguments.
        if (buffer == null)
        {
            throw new ArgumentNullException("buffer");
        }

        // Set each byte in the buffer to 0.
        for (int x = 0; x < buffer.Length; x++)
        {
            buffer[x] = 0;
        }
    }

    #endregion
}
```

This example is taken from MSDN.

It is a console demo, and it shows how to create a secure key based on a user-defined password, and how to create a random SALT based on the cryptographic random generator.

备注：

- 内置函数PasswordDeriveBytes使用标准的PBKDF1算法从密码生成密钥。默认情况下，它使用100次迭代来生成密钥，以减缓暴力破解攻击。随机生成的盐值进一步增强了密钥的安全性。

- 函数CryptDeriveKey通过使用指定的哈希算法（此处为“SHA1”），将PasswordDeriveBytes生成的密钥转换为与指定加密算法（此处为“三重DES”）兼容的密钥。此示例中的密钥大小为192字节，初始化向量IV取自三重DES加密提供程序。

- 通常，这种机制用于通过密码保护一个更强的随机生成密钥，该密钥用于加密大量数据。你也可以用它为不同用户提供多个密码，以便访问相同的数据（由不同的随机密钥保护）。

- 不幸的是，CryptDeriveKey 目前不支持 AES。详见 here。
  注意： 作为一种变通方法，你可以为要保护的数据创建一个随机的 AES 密钥进行加密，并将 AES 密钥存储在使用 CryptDeriveKey 生成的密钥的 TripleDES 容器中。但这将安全性限制在 TripleDES，无法利用 AES 的更大密钥长度，并且产生对 TripleDES 的依赖。

用法： 参见 Main() 方法。

Notes:

- The built-in function `PasswordDeriveBytes` uses the standard PBKDF1 algorithm to generate a key from the password. Per default, it uses 100 iterations to generate the key to slow down brute force attacks. The SALT generated randomly further strenghens the key.

- The function `CryptDeriveKey` converts the key generated by `PasswordDeriveBytes` into a key compatible with the specified encryption algorithm (here "TripleDES") by using the specified hash algorithm (here "SHA1"). The keysize in this example is 192 bytes, and the initialization vector IV is taken from the triple-DES crypto provider

- Usually, this mechanism is used to protect a stronger random generated key by a password, which encrypts large amount of data. You can also use it to provide multiple passwords of different users to give access to the same data (being protected by a different random key).

- Unfortunately, `CryptDeriveKey` does currently not support AES. See here.
  NOTE: As a workaround, you can create a random AES key for encryption of the data to be protected with AES and store the AES key in a TripleDES-Container which uses the key generated by `CryptDeriveKey`. But that limits the security to TripleDES, does not take advantage of the larger keysizes of AES and creates a dependency to TripleDES.

Usage: See Main() method.

第50章：在 C# 中使用 SHA1

在本项目中，你将学习如何使用 SHA1 加密哈希函数。例如，如何从字符串获取哈希值以及如何破解 SHA1 哈希。

完整源码在 GitHub：https://github.com/mahdiabasi/SHA1Tool

第50.1节：#生成文件的 SHA1 校验和

> 首先，你需要将 System.Security.Cryptography 命名空间添加到你的项目中

```
public string GetSha1Hash(string filePath)
    {
        using (FileStream fs = File.OpenRead(filePath))
        {
SHA1 sha = new SHA1Managed();
            return BitConverter.ToString(sha.ComputeHash(fs));
        }
    }
```

第50.2节：#生成文本的哈希值

```
  public static string TextToHash(string text)
    {
        var sh = SHA1.Create();
        var hash = new StringBuilder();
        byte[] bytes = Encoding.UTF8.GetBytes(text);
        byte[] b = sh.ComputeHash(bytes);
        foreach (byte a in b)
        {
            var h = a.ToString("x2");
            hash.Append(h);
        }
        return hash.ToString();
    }
```


Chapter 50: Work with SHA1 in C#

in this project you see how to work with SHA1 cryptographic hash function. for example get hash from string and how to crack SHA1 hash.

source compelete on github: https://github.com/mahdiabasi/SHA1Tool

Section 50.1: #Generate SHA1 checksum of a file

> first you add System.Security.Cryptography namespace to your project

```
public string GetSha1Hash(string filePath)
    {
        using (FileStream fs = File.OpenRead(filePath))
        {
            SHA1 sha = new SHA1Managed();
            return BitConverter.ToString(sha.ComputeHash(fs));
        }
    }
```

Section 50.2: #Generate hash of a text

```
  public static string TextToHash(string text)
    {
        var sh = SHA1.Create();
        var hash = new StringBuilder();
        byte[] bytes = Encoding.UTF8.GetBytes(text);
        byte[] b = sh.ComputeHash(bytes);
        foreach (byte a in b)
        {
            var h = a.ToString("x2");
            hash.Append(h);
        }
        return hash.ToString();
    }
```

第51章：单元测试

第51.1节：向现有解决方案添加MSTest单元测试项目

- 右键点击解决方案，添加新项目
- 从测试部分，选择一个单元测试项目
- 为程序集选择一个名称——如果你正在测试项目Foo，名称可以是Foo.Tests
- 在单元测试项目的引用中添加对被测试项目的引用

第51.2节：创建示例测试方法

MSTest（默认的测试框架）要求你的测试类使用`[TestClass]`特性进行装饰，
测试方法使用`[TestMethod]`特性，并且必须是公共的。

```
[TestClass]
public class FizzBuzzFixture
{
    [测试方法]
    public void 测试1()
    {
        //排列
        var solver = new FizzBuzzSolver();
        //act
        var result = solver.FizzBuzz(1);
        //assert
Assert.AreEqual("1",result);
    }
}
```


Chapter 51: Unit testing

Section 51.1: Adding MSTest unit testing project to an existing solution

- Right click on the solution, Add new project
- From the Test section, select an Unit Test Project
- Pick a name for the assembly - if you are testing project `Foo`, the name can be `Foo.Tests`
- Add a reference to the tested project in the unit test project references

Section 51.2: Creating a sample test method

MSTest (the default testing framework) requires you to have your test classes decorated by a `[TestClass]` attribute, and the test methods with a `[TestMethod]` attribute, and to be public.

```
[TestClass]
public class FizzBuzzFixture
{
    [TestMethod]
    public void Test1()
    {
        //arrange
        var solver = new FizzBuzzSolver();
        //act
        var result = solver.FizzBuzz(1);
        //assert
        Assert.AreEqual("1",result);
    }
}
```

第52章：写入和读取标准错误流

第52.1节：使用Console写入标准错误输出

```
var sourceFileName = "NonExistingFile";
try
{
    System.IO.File.Copy(sourceFileName, "DestinationFile");
}
catch (Exception e)
{
    var stdErr = Console.Error;
stdErr.WriteLine($"无法复制 '{sourceFileName}': {e.Message}");
}
```

第52.2节：从子进程的标准错误读取

```
var errors = new System.Text.StringBuilder();
var process = new Process
{
StartInfo = new ProcessStartInfo
    {
RedirectStandardError = true,
        FileName = "xcopy.exe",
Arguments = "\"NonExistingFile\" \"DestinationFile\"",
        UseShellExecute = false
    },

};
process.ErrorDataReceived += (s, e) => errors.AppendLine(e.Data);
process.Start();
process.BeginErrorReadLine();
process.WaitForExit();

if (errors.Length > 0) // something went wrongSystem.
     Console.Error.WriteLine($"Child process error: \r {errors}");
```


Chapter 52: Write to and read from StdErr stream

Section 52.1: Write to standard error output using Console

```
var sourceFileName = "NonExistingFile";
try
{
    System.IO.File.Copy(sourceFileName, "DestinationFile");
}
catch (Exception e)
{
    var stdErr = Console.Error;
    stdErr.WriteLine($"Failed to copy '{sourceFileName}': {e.Message}");
}
```

Section 52.2: Read from standard error of child process

```
var errors = new System.Text.StringBuilder();
var process = new Process
{
    StartInfo = new ProcessStartInfo
    {
        RedirectStandardError = true,
        FileName = "xcopy.exe",
        Arguments = "\"NonExistingFile\" \"DestinationFile\"",
        UseShellExecute = false
    },

};
process.ErrorDataReceived += (s, e) => errors.AppendLine(e.Data);
process.Start();
process.BeginErrorReadLine();
process.WaitForExit();

if (errors.Length > 0) // something went wrong
    System.Console.Error.WriteLine($"Child process error: \r\n {errors}");
```

第53章：上传文件和向

网络服务器POST数据

第53.1节：使用WebRequest上传文件

要在单个请求中发送文件和表单数据，内容应为multipart/form-data类型。

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
using System.Threading.Tasks;

public async Task<string> UploadFile(string url, string filename,
    Dictionary<string, object> postData)
{
    var request = WebRequest.CreateHttp(url);
    var boundary = $"{Guid.NewGuid():N}"; // boundary 将分隔每个参数
    request.ContentType = $"multipart/form-data; {nameof(boundary)}={boundary}";
    request.Method = "POST";

    using (var requestStream = request.GetRequestStream())
    using (var writer = new StreamWriter(requestStream))
    {
        foreach (var data in postData)
            await writer.WriteAsync( // 将所有 POST 数据写入请求
$"\r--{boundary}\rContent-Disposition: " +                $"form-data; nam
e=\"{data.Key}\"\r\r{data.Value}");

        await writer.WriteAsync( // 文件头
$"\r--{boundary}\rContent-Disposition: " +                $"form-data; na
me=\"File\"; filename=\"{Path.GetFileName(filename)}\"\r" +"Content-Type: application/octet-stream\r\r");


        await writer.FlushAsync();using
        (var fileStream = File.OpenRead(filename))await fileStream.
            CopyToAsync(requestStream);await writer.WriteAsync($

        "\r--{boundary}--\r");
    }

    using (var response = (HttpWebResponse) await request.GetResponseAsync())
    using (var responseStream = response.GetResponseStream())
    {
        if (responseStream == null)
            return string.Empty;
        using (var reader = new StreamReader(responseStream))
            return await reader.ReadToEndAsync();
    }
}
```

用法：

```
var response = await uploader.UploadFile("< YOUR URL >", "< PATH TO YOUR FILE >",
    new Dictionary<string, object>
    {
        {"Comment", "test"},
        {"Modified", DateTime.Now }
```


Chapter 53: Upload file and POST data to webserver

Section 53.1: Upload file with WebRequest

To send a file and form data in single request, content should have multipart/form-data type.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
using System.Threading.Tasks;

public async Task<string> UploadFile(string url, string filename,
    Dictionary<string, object> postData)
{
    var request = WebRequest.CreateHttp(url);
    var boundary = $"{Guid.NewGuid():N}"; // boundary will separate each parameter
    request.ContentType = $"multipart/form-data; {nameof(boundary)}={boundary}";
    request.Method = "POST";

    using (var requestStream = request.GetRequestStream())
    using (var writer = new StreamWriter(requestStream))
    {
        foreach (var data in postData)
            await writer.WriteAsync( // put all POST data into request
                $"\r\n--{boundary}\r\nContent-Disposition: " +
                $"form-data; name=\"{data.Key}\"\r\n\r\n{data.Value}");

        await writer.WriteAsync( // file header
            $"\r\n--{boundary}\r\nContent-Disposition: " +
            $"form-data; name=\"File\"; filename=\"{Path.GetFileName(filename)}\"\r\n" +
            "Content-Type: application/octet-stream\r\n\r\n");

        await writer.FlushAsync();
        using (var fileStream = File.OpenRead(filename))
            await fileStream.CopyToAsync(requestStream);

        await writer.WriteAsync($"\r\n--{boundary}--\r\n");
    }

    using (var response = (HttpWebResponse) await request.GetResponseAsync())
    using (var responseStream = response.GetResponseStream())
    {
        if (responseStream == null)
            return string.Empty;
        using (var reader = new StreamReader(responseStream))
            return await reader.ReadToEndAsync();
    }
}
```

Usage:

```
var response = await uploader.UploadFile("< YOUR URL >", "< PATH TO YOUR FILE >",
    new Dictionary<string, object>
    {
        {"Comment", "test"},
        {"Modified", DateTime.Now }
```

```
});
```


});

第54章：网络

第54.1节：基础TCP聊天（TcpListener，TcpClient，NetworkStream）

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpChat
{
    static void Main(string[] args)
    {
        if(args.Length == 0)
        {
Console.WriteLine("基础 TCP 聊天");
            Console.WriteLine();
Console.WriteLine("用法:");
            Console.WriteLine("tcpchat server <端口>");
            Console.WriteLine("tcpchat client <网址> <端口>");
            return;
        }

        try
        {
Run(args);
        }
        catch(IOException)
        {
Console.WriteLine("--- 连接丢失");
        }
        catch(SocketException ex)
        {
Console.WriteLine("--- 无法连接: " + ex.Message);
        }
    }

    static void Run(string[] args)
    {
TcpClient client;
        NetworkStream stream;
        byte[] buffer = new byte[256];
        var encoding = Encoding.ASCII;

        if(args[0].StartsWith("s", StringComparison.InvariantCultureIgnoreCase))
        {
            var port = int.Parse(args[1]);
            var listener = new TcpListener(IPAddress.Any, port);
            listener.Start();
Console.WriteLine("--- 等待连接中...");
           client = listener.AcceptTcpClient();
        }
        否则
        {
            var hostName = args[1];
            var port = int.Parse(args[2]);
            client = new TcpClient();
            client.Connect(hostName, port);
```


Chapter 54: Networking

Section 54.1: Basic TCP chat (TcpListener, TcpClient, NetworkStream)

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;
using System.Text;

class TcpChat
{
    static void Main(string[] args)
    {
        if(args.Length == 0)
        {
            Console.WriteLine("Basic TCP chat");
            Console.WriteLine();
            Console.WriteLine("Usage:");
            Console.WriteLine("tcpchat server <port>");
            Console.WriteLine("tcpchat client <url> <port>");
            return;
        }

        try
        {
            Run(args);
        }
        catch(IOException)
        {
            Console.WriteLine("--- Connection lost");
        }
        catch(SocketException ex)
        {
            Console.WriteLine("--- Can't connect: " + ex.Message);
        }
    }

    static void Run(string[] args)
    {
        TcpClient client;
        NetworkStream stream;
        byte[] buffer = new byte[256];
        var encoding = Encoding.ASCII;

        if(args[0].StartsWith("s", StringComparison.InvariantCultureIgnoreCase))
        {
            var port = int.Parse(args[1]);
            var listener = new TcpListener(IPAddress.Any, port);
            listener.Start();
            Console.WriteLine("--- Waiting for a connection...");
            client = listener.AcceptTcpClient();
        }
        else
        {
            var hostName = args[1];
            var port = int.Parse(args[2]);
            client = new TcpClient();
            client.Connect(hostName, port);
```

```
        }

stream = client.GetStream();
Console.WriteLine("--- 已连接。开始输入！(按 Ctrl-C 退出)");

        while(true)
        {
            if(Console.KeyAvailable)
            {
                var lineToSend = Console.ReadLine();
                var bytesToSend = encoding.GetBytes(lineToSend + "\r");
    stream.Write(bytesToSend, 0, bytesToSend.Length);                stream.F
lush();
            }

            if (stream.DataAvailable)
            {
                var receivedBytesCount = stream.Read(buffer, 0, buffer.Length);
                var receivedString = encoding.GetString(buffer, 0, receivedBytesCount);
                Console.Write(receivedString);
            }
        }
    }
}
```

第54.2节：基础SNTP客户端（UdpClient）

有关SNTP协议的详细信息，请参见RFC 2030。

```
using System;
using System.Globalization;
using System.Linq;
using System.Net;
using System.Net.Sockets;

class SntpClient
{
    const int SntpPort = 123;
    static DateTime BaseDate = new DateTime(1900, 1, 1);

    static void Main(string[] args)
    {
        if(args.Length == 0) {
Console.WriteLine("简单的SNTP客户端");
            Console.WriteLine();
Console.WriteLine("用法：sntpclient <sntp 服务器网址> [<本地时区>]" );
            Console.WriteLine();
Console.WriteLine("本地时区：一个介于-12到12之间的数字，表示相对于UTC的小时数");
              Console.WriteLine("(附加.5表示额外的半小时)");
              return;
        }

        double localTimeZoneInHours = 0;
        if(args.Length > 1)
localTimeZoneInHours = double.Parse(args[1], CultureInfo.InvariantCulture);

        var udpClient = new UdpClient();
        udpClient.Client.ReceiveTimeout = 5000;

          var sntpRequest = new byte[48];
sntpRequest[0] = 0x23; //LI=0（无警告），VN=4，Mode=3（客户端）
```



```
        }

        stream = client.GetStream();
        Console.WriteLine("--- Connected. Start typing! (exit with Ctrl-C)");

        while(true)
        {
            if(Console.KeyAvailable)
            {
                var lineToSend = Console.ReadLine();
                var bytesToSend = encoding.GetBytes(lineToSend + "\r\n");
                stream.Write(bytesToSend, 0, bytesToSend.Length);
                stream.Flush();
            }

            if (stream.DataAvailable)
            {
                var receivedBytesCount = stream.Read(buffer, 0, buffer.Length);
                var receivedString = encoding.GetString(buffer, 0, receivedBytesCount);
                Console.Write(receivedString);
            }
        }
    }
}
```

Section 54.2: Basic SNTP client (UdpClient)

See RFC 2030 for details on the SNTP protocol.

```
using System;
using System.Globalization;
using System.Linq;
using System.Net;
using System.Net.Sockets;

class SntpClient
{
    const int SntpPort = 123;
    static DateTime BaseDate = new DateTime(1900, 1, 1);

    static void Main(string[] args)
    {
        if(args.Length == 0) {
            Console.WriteLine("Simple SNTP client");
            Console.WriteLine();
            Console.WriteLine("Usage: sntpclient <sntp server url> [<local timezone>]");
            Console.WriteLine();
            Console.WriteLine("<local timezone>: a number between -12 and 12 as hours from UTC");
            Console.WriteLine("(append .5 for an extra half an hour)");
            return;
        }

        double localTimeZoneInHours = 0;
        if(args.Length > 1)
            localTimeZoneInHours = double.Parse(args[1], CultureInfo.InvariantCulture);

        var udpClient = new UdpClient();
        udpClient.Client.ReceiveTimeout = 5000;

        var sntpRequest = new byte[48];
        sntpRequest[0] = 0x23; //LI=0 (no warning), VN=4, Mode=3 (client)
```

```
udpClient.Send(
            dgram: sntpRequest,
            bytes: sntpRequest.Length,
            hostname: args[0],
port: SntpPort);

        byte[] sntpResponse;
        try
        {
IPEndPoint 远程端点 = null;
          sntp响应 = udp客户端.接收(ref 远程端点);
        }
        catch(SocketException)
        {
Console.写行("*** 未收到服务器响应");
            return;
        }

        uint 秒数;
        if(BitConverter.IsLittleEndian)
秒数 = BitConverter.ToUInt32(
               sntp响应.跳过(40).取(4).反转().转数组()
               ,0);
        否则
秒数 = BitConverter.ToUInt32(sntp响应, 40);

        var 日期 = 基准日期.加秒(秒数).加小时(本地时区小时数);

Console.写行(
$"服务器当前日期: {日期:yyyy-MM-dd HH:mm:ss}
UTC{本地时区小时数:+0.#;-0.#;.}");
    }
}
```



```
        udpClient.Send(
            dgram: sntpRequest,
            bytes: sntpRequest.Length,
            hostname: args[0],
            port: SntpPort);

        byte[] sntpResponse;
        try
        {
            IPEndPoint remoteEndpoint = null;
            sntpResponse = udpClient.Receive(ref remoteEndpoint);
        }
        catch(SocketException)
        {
            Console.WriteLine("*** No response received from the server");
            return;
        }

        uint numberOfSeconds;
        if(BitConverter.IsLittleEndian)
            numberOfSeconds = BitConverter.ToUInt32(
                sntpResponse.Skip(40).Take(4).Reverse().ToArray()
                ,0);
        else
            numberOfSeconds = BitConverter.ToUInt32(sntpResponse, 40);

        var date = BaseDate.AddSeconds(numberOfSeconds).AddHours(localTimeZoneInHours);

        Console.WriteLine(
            $"Current date in server: {date:yyyy-MM-dd HH:mm:ss}
UTC{localTimeZoneInHours:+0.#;-0.#;.}");
    }
}
```

第55章：HTTP服务器

第55.1节：基本只读HTTP文件服务器（ASP.NET Core）

1 - 创建一个空文件夹，后续步骤中创建的文件将保存在此文件夹中。

2 - 创建一个名为project.json的文件，内容如下（根据需要调整端口号和rootDirectory）：

```
{
  "dependencies": {
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
    "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final"
  },

  "commands": {
    "web": "Microsoft.AspNet.Server.Kestrel --server.urls http://localhost:60000"
  },

  "frameworks": {
    "dnxcore50": { }
  },

  "fileServer": {
    "rootDirectory": "c:\\users\\username\\Documents"
  }
}
```

3 - 创建一个名为Startup.cs的文件，内容如下代码：

```
using System;
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.FileProviders;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.StaticFiles;
using Microsoft.Extensions.Configuration;

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        var builder = new ConfigurationBuilder();
        builder.AddJsonFile("project.json");
        var config = builder.Build();
        var rootDirectory = config["fileServer:rootDirectory"];
       Console.WriteLine("文件服务器根目录: " + rootDirectory);

        var fileProvider = new PhysicalFileProvider(rootDirectory);

        var options = new StaticFileOptions();
options.ServeUnknownFileTypes = true;
        options.FileProvider = fileProvider;
        options.OnPrepareResponse = context =>
        {
context.Context.Response.ContentType = "application/octet-stream";
            context.Context.Response.Headers.Add(
                "Content-Disposition",
$"Attachment; filename=\"{context.File.Name}\"");
        };
```


Chapter 55: HTTP servers

Section 55.1: Basic read-only HTTP file server (ASP.NET Core)

1 - Create an empty folder, it will contain the files created in the next steps.

2 - Create a file named `project.json` with the following content (adjust the port number and `rootDirectory` as appropriate):

```
{
  "dependencies": {
    "Microsoft.AspNet.Server.Kestrel": "1.0.0-rc1-final",
    "Microsoft.AspNet.StaticFiles": "1.0.0-rc1-final"
  },

  "commands": {
    "web": "Microsoft.AspNet.Server.Kestrel --server.urls http://localhost:60000"
  },

  "frameworks": {
    "dnxcore50": { }
  },

  "fileServer": {
    "rootDirectory": "c:\\users\\username\\Documents"
  }
}
```

3 - Create a file named `Startup.cs` with the following code:

```
using System;
using Microsoft.AspNet.Builder;
using Microsoft.AspNet.FileProviders;
using Microsoft.AspNet.Hosting;
using Microsoft.AspNet.StaticFiles;
using Microsoft.Extensions.Configuration;

public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        var builder = new ConfigurationBuilder();
        builder.AddJsonFile("project.json");
        var config = builder.Build();
        var rootDirectory = config["fileServer:rootDirectory"];
        Console.WriteLine("File server root directory: " + rootDirectory);

        var fileProvider = new PhysicalFileProvider(rootDirectory);

        var options = new StaticFileOptions();
        options.ServeUnknownFileTypes = true;
        options.FileProvider = fileProvider;
        options.OnPrepareResponse = context =>
        {
            context.Context.Response.ContentType = "application/octet-stream";
            context.Context.Response.Headers.Add(
                "Content-Disposition",
                $"Attachment; filename=\"{context.File.Name}\"");
        };
```

```
app.UseStaticFiles(options);
    }
}
```

4 - 打开命令提示符，导航到该文件夹并执行：

```
dnvm use 1.0.0-rc1-final -r coreclr -p
dnu restore
```

注意： 这些命令只需运行一次。使用 dnvm list 检查已安装的最新 core CLR 版本的实际编号。

5 - 使用 dnx web 启动服务器。现在可以通过 http://localhost:60000/path/to/file.ext 请求文件。

为简化起见，假设文件名均为 ASCII（用于 Content-Disposition 头中的文件名部分），且不处理文件访问错误。

第 55.2 节：基本只读 HTTP 文件服务器（HttpListener）

备注：

此示例必须以管理员模式运行。

仅支持一个同时客户端连接。

为简化起见，假设文件名均为 ASCII（用于 Content-Disposition 头中的 filename 部分），且不处理文件访问错误。

```
using System;
using System.IO;
using System.Net;

class HttpFileServer
{
    private static HttpListenerResponse response;
    private static HttpListener listener;
    private static string baseFilesystemPath;

    static void Main(string[] args)
    {
        if (!HttpListener.IsSupported)
        {
Console.WriteLine(
                "*** HttpListener requires at least Windows XP SP2 or Windows Server 2003.");
            return;
        }

        if(args.Length < 2)
        {
Console.WriteLine("Basic read-only HTTP file server");
            Console.WriteLine();
Console.WriteLine("Usage: httpfileserver <base filesystem path> <port>");
            Console.WriteLine("Request format: http://url:port/path/to/file.ext");
            return;
        }

baseFilesystemPath = Path.GetFullPath(args[0]);
        var port = int.Parse(args[1]);
```



```
        app.UseStaticFiles(options);
    }
}
```

4 - Open a command prompt, navigate to the folder and execute:

```
dnvm use 1.0.0-rc1-final -r coreclr -p
dnu restore
```

Note: These commands need to be run only once. Use `dnvm list` to check the actual number of the latest installed version of the core CLR.

5 - Start the server with: `dnx web`. Files can now be requested at `http://localhost:60000/path/to/file.ext`.

For simplicity, filenames are assumed to be all ASCII (for the filename part in the Content-Disposition header) and file access errors are not handled.

Section 55.2: Basic read-only HTTP file server (HttpListener)

Notes:

This example must be run in administrative mode.

Only one simultaneous client is supported.

For simplicity, filenames are assumed to be all ASCII (for the *filename* part in the *Content-Disposition* header) and file access errors are not handled.

```
using System;
using System.IO;
using System.Net;

class HttpFileServer
{
    private static HttpListenerResponse response;
    private static HttpListener listener;
    private static string baseFilesystemPath;

    static void Main(string[] args)
    {
        if (!HttpListener.IsSupported)
        {
            Console.WriteLine(
                "*** HttpListener requires at least Windows XP SP2 or Windows Server 2003.");
            return;
        }

        if(args.Length < 2)
        {
            Console.WriteLine("Basic read-only HTTP file server");
            Console.WriteLine();
            Console.WriteLine("Usage: httpfileserver <base filesystem path> <port>");
            Console.WriteLine("Request format: http://url:port/path/to/file.ext");
            return;
        }

        baseFilesystemPath = Path.GetFullPath(args[0]);
        var port = int.Parse(args[1]);
```

```
listener = new HttpListener();
        listener.Prefixes.Add("http://*:" + port + "/");
        listener.Start();

Console.WriteLine("--- 服务器已启动，基础路径为: " + baseFilesystemPath);
        Console.WriteLine("--- 正在监听，按 Ctrl-C 退出");
        try
        {
ServerLoop();
        }
        catch(Exception ex)
        {
Console.WriteLine(ex);
            if(response != null)
            {
SendErrorResponse(500, "内部服务器错误");
            }
        }
    }

    static void ServerLoop()
    {
        while(true)
        {
            var context = listener.GetContext();

            var request = context.Request;
response = context.Response;
            var fileName = request.RawUrl.Substring(1);
          Console.WriteLine(
                "--- 收到 {0} 请求，路径为: {1}",
              request.HttpMethod, fileName);

            if (request.HttpMethod.ToUpper() != "GET")
            {
SendErrorResponse(405, "方法必须是GET");
                continue;
            }

            var fullFilePath = Path.Combine(baseFilesystemPath, fileName);
            if(!File.Exists(fullFilePath))
            {
SendErrorResponse(404, "文件未找到");
                continue;
            }

Console.Write("     正在发送文件...");
            using (var fileStream = File.OpenRead(fullFilePath))
            {
response.ContentType = "application/octet-stream";
                response.ContentLength64 = (new FileInfo(fullFilePath)).Length;
                response.AddHeader(
                    "Content-Disposition",
                    "Attachment; filename=\"" + Path.GetFileName(fullFilePath) + "\"");
                fileStream.CopyTo(response.OutputStream);
            }

response.OutputStream.Close();
            response = null;
Console.WriteLine(" 完成!");
        }
    }
```



```
        listener = new HttpListener();
        listener.Prefixes.Add("http://*:" + port + "/");
        listener.Start();

        Console.WriteLine("--- Server stated, base path is: " + baseFilesystemPath);
        Console.WriteLine("--- Listening, exit with Ctrl-C");
        try
        {
            ServerLoop();
        }
        catch(Exception ex)
        {
            Console.WriteLine(ex);
            if(response != null)
            {
                SendErrorResponse(500, "Internal server error");
            }
        }
    }

    static void ServerLoop()
    {
        while(true)
        {
            var context = listener.GetContext();

            var request = context.Request;
            response = context.Response;
            var fileName = request.RawUrl.Substring(1);
            Console.WriteLine(
                "--- Got {0} request for: {1}",
                request.HttpMethod, fileName);

            if (request.HttpMethod.ToUpper() != "GET")
            {
                SendErrorResponse(405, "Method must be GET");
                continue;
            }

            var fullFilePath = Path.Combine(baseFilesystemPath, fileName);
            if(!File.Exists(fullFilePath))
            {
                SendErrorResponse(404, "File not found");
                continue;
            }

            Console.Write("    Sending file...");
            using (var fileStream = File.OpenRead(fullFilePath))
            {
                response.ContentType = "application/octet-stream";
                response.ContentLength64 = (new FileInfo(fullFilePath)).Length;
                response.AddHeader(
                    "Content-Disposition",
                    "Attachment; filename=\"" + Path.GetFileName(fullFilePath) + "\"");
                fileStream.CopyTo(response.OutputStream);
            }

            response.OutputStream.Close();
            response = null;
            Console.WriteLine(" Ok!");
        }
    }
```

```
    static void SendErrorResponse(int statusCode, string statusResponse)
    {
response.ContentLength64 = 0;
        response.StatusCode = statusCode;
        response.StatusDescription = statusResponse;
        response.OutputStream.Close();
Console.WriteLine("*** 发送错误: {0} {1}", statusCode, statusResponse);
    }
}
```



```
    static void SendErrorResponse(int statusCode, string statusResponse)
    {
        response.ContentLength64 = 0;
        response.StatusCode = statusCode;
        response.StatusDescription = statusResponse;
        response.OutputStream.Close();
        Console.WriteLine("*** Sent error: {0} {1}", statusCode, statusResponse);
    }
}
```

第56章：HTTP客户端

第56.1节：使用

System.Net.HttpClient读取GET响应为字符串

HttpClient 可通过 NuGet: Microsoft HTTP Client Libraries 获得。

```
string requestUri = "http://www.example.com";
string responseData;

using (var client = new HttpClient())
{
    using(var response = client.GetAsync(requestUri).Result)
    {
response.EnsureSuccessStatusCode();
        responseData = response.Content.ReadAsStringAsync().Result;
    }
}
```

第56.2节：使用

System.Net.Http.HttpClient的基本HTTP下载器

```
using System;
using System.IO;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;

class HttpGet
{
    private static async Task DownloadAsync(string fromUrl, string toFile)
    {
        using (var fileStream = File.OpenWrite(toFile))
        {
            using (var httpClient = new HttpClient())
            {
Console.WriteLine("Connecting...");
                using (var networkStream = await httpClient.GetStreamAsync(fromUrl))
                {
Console.WriteLine("Downloading...");
                    await networkStream.CopyToAsync(fileStream);
                    await fileStream.FlushAsync();
                }
            }
        }
    }

    static void Main(string[] args)
    {
        try
        {
Run(args).Wait();
        }
        catch (Exception ex)
        {
            if (ex is AggregateException)
ex = ((AggregateException)ex).Flatten().InnerExceptions.First();
```


Chapter 56: HTTP clients

Section 56.1: Reading GET response as string using System.Net.HttpClient

`HttpClient` is available through NuGet: Microsoft HTTP Client Libraries.

```
string requestUri = "http://www.example.com";
string responseData;

using (var client = new HttpClient())
{
    using(var response = client.GetAsync(requestUri).Result)
    {
        response.EnsureSuccessStatusCode();
        responseData = response.Content.ReadAsStringAsync().Result;
    }
}
```

Section 56.2: Basic HTTP downloader using System.Net.Http.HttpClient

```
using System;
using System.IO;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;

class HttpGet
{
    private static async Task DownloadAsync(string fromUrl, string toFile)
    {
        using (var fileStream = File.OpenWrite(toFile))
        {
            using (var httpClient = new HttpClient())
            {
                Console.WriteLine("Connecting...");
                using (var networkStream = await httpClient.GetStreamAsync(fromUrl))
                {
                    Console.WriteLine("Downloading...");
                    await networkStream.CopyToAsync(fileStream);
                    await fileStream.FlushAsync();
                }
            }
        }
    }

    static void Main(string[] args)
    {
        try
        {
            Run(args).Wait();
        }
        catch (Exception ex)
        {
            if (ex is AggregateException)
                ex = ((AggregateException)ex).Flatten().InnerExceptions.First();
```

```
Console.WriteLine("--- Error: " +
                  (ex.InnerException?.Message ?? ex.Message));
        }
    }
    static async Task Run(string[] args)
    {
        if (args.Length < 2)
        {
Console.WriteLine("基础 HTTP 下载器");
            Console.WriteLine();
Console.WriteLine("用法: httpget <url>[<:port>] <文件>");
            return;
        }

        await DownloadAsync(fromUrl: args[0], toFile: args[1]);

Console.WriteLine("完成！");
    }
}
```

第56.3节：使用

System.Net.HttpWebRequest读取GET响应为字符串

```
string requestUri = "http://www.example.com";
string responseData;

HttpWebRequest request = (HttpWebRequest)WebRequest.Create(parameters.Uri);
WebResponse response = request.GetResponse();

using (StreamReader responseReader = new StreamReader(response.GetResponseStream()))
{
responseData = responseReader.ReadToEnd();
}
```

第56.4节：使用

System.Net.WebClient读取GET响应为字符串

```
string requestUri = "http://www.example.com";
string responseData;

using (var client = new WebClient())
{
responseData = client.DownloadString(requestUri);
}
```

第56.5节：使用 System.Net.HttpWebRequest 发送带字符串负载的 POST 请求

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";

HttpWebRequest request = (HttpWebRequest)WebRequest.Create(requestUri)
{
Method = requestMethod,
  ContentType = contentType,
};
```



```
            Console.WriteLine("--- Error: " +
                (ex.InnerException?.Message ?? ex.Message));
        }
    }
    static async Task Run(string[] args)
    {
        if (args.Length < 2)
        {
            Console.WriteLine("Basic HTTP downloader");
            Console.WriteLine();
            Console.WriteLine("Usage: httpget <url>[<:port>] <file>");
            return;
        }

        await DownloadAsync(fromUrl: args[0], toFile: args[1]);

        Console.WriteLine("Done!");
    }
}
```

Section 56.3: Reading GET response as string using System.Net.HttpWebRequest

```
string requestUri = "http://www.example.com";
string responseData;

HttpWebRequest request = (HttpWebRequest)WebRequest.Create(parameters.Uri);
WebResponse response = request.GetResponse();

using (StreamReader responseReader = new StreamReader(response.GetResponseStream()))
{
    responseData = responseReader.ReadToEnd();
}
```

Section 56.4: Reading GET response as string using System.Net.WebClient

```
string requestUri = "http://www.example.com";
string responseData;

using (var client = new WebClient())
{
    responseData = client.DownloadString(requestUri);
}
```

Section 56.5: Sending a POST request with a string payload using System.Net.HttpWebRequest

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";

HttpWebRequest request = (HttpWebRequest)WebRequest.Create(requestUri)
{
  Method = requestMethod,
  ContentType = contentType,
};
```

```
byte[] bytes = Encoding.UTF8.GetBytes(requestBodyString);
Stream stream = request.GetRequestStream();
stream.Write(bytes, 0, bytes.Length);
stream.Close();

HttpWebResponse response = (HttpWebResponse)request.GetResponse();
```

第56.6节：使用 System.Net.WebClient 发送带字符串负载的 POST 请求

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";

byte[] responseBody;
byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestBodyString);

using (var client = new WebClient())
{
client.Headers[HttpRequestHeader.ContentType] = contentType;
    responseBody = client.UploadData(requestUri, requestMethod, requestBodyBytes);
}
```

第56.7节：使用字符串负载发送POST请求 使用 System.Net.HttpClient

HttpClient 可通过 NuGet: Microsoft HTTP Client Libraries 获得。

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";

var request = new HttpRequestMessage
{
RequestUri = requestUri,
    Method = requestMethod,
};

byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestBodyString);
request.Content = new ByteArrayContent(requestBodyBytes);

request.Content.Headers.ContentType = new MediaTypeHeaderValue(contentType);

HttpResponseMessage result = client.SendAsync(request).Result;
result.EnsureSuccessStatusCode();
```



```
byte[] bytes = Encoding.UTF8.GetBytes(requestBodyString);
Stream stream = request.GetRequestStream();
stream.Write(bytes, 0, bytes.Length);
stream.Close();

HttpWebResponse response = (HttpWebResponse)request.GetResponse();
```

Section 56.6: Sending a POST request with a string payload using System.Net.WebClient

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";

byte[] responseBody;
byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestBodyString);

using (var client = new WebClient())
{
    client.Headers[HttpRequestHeader.ContentType] = contentType;
    responseBody = client.UploadData(requestUri, requestMethod, requestBodyBytes);
}
```

Section 56.7: Sending a POST request with a string payload using System.Net.HttpClient

`HttpClient` is available through NuGet: Microsoft HTTP Client Libraries.

```
string requestUri = "http://www.example.com";
string requestBodyString = "Request body string.";
string contentType = "text/plain";
string requestMethod = "POST";

var request = new HttpRequestMessage
{
    RequestUri = requestUri,
    Method = requestMethod,
};

byte[] requestBodyBytes = Encoding.UTF8.GetBytes(requestBodyString);
request.Content = new ByteArrayContent(requestBodyBytes);

request.Content.Headers.ContentType = new MediaTypeHeaderValue(contentType);

HttpResponseMessage result = client.SendAsync(request).Result;
result.EnsureSuccessStatusCode();
```

第57章：串口

第57.1节：基本操作

```
var serialPort = new SerialPort("COM1", 9600, Parity.Even, 8, StopBits.One);
serialPort.Open();
serialPort.WriteLine("Test data");
string response = serialPort.ReadLine();
Console.WriteLine(response);
serialPort.Close();
```

第57.2节：列出可用端口名称

```
string[] portNames = SerialPort.GetPortNames();
```

第57.3节：异步读取

```
void SetupAsyncRead(SerialPort serialPort)
{
serialPort.DataReceived += (sender, e) => {
        byte[] buffer = new byte[4096];
        switch (e.EventType)
        {
             case SerialData.Chars:
                 var port = (SerialPort)sender;
                 int bytesToRead = port.BytesToRead;
                 if (bytesToRead > buffer.Length)
Array.Resize(ref buffer, bytesToRead);
                 int bytesRead = port.Read(buffer, 0, bytesToRead);
                 // 在此处理读取的缓冲区
                 // ...
                 break;
            case SerialData.Eof:
                 // 在此终止服务
                 // ...
                 break;
        }
    };
```

第57.4节：同步文本回显服务

```
using System.IO.Ports;

namespace TextEchoService
{
    class Program
    {
        static void Main(string[] args)
        {
            var serialPort = new SerialPort("COM1", 9600, Parity.Even, 8, StopBits.One);
            serialPort.Open();
            string message = "";
            while (message != "quit")
            {
message = serialPort.ReadLine();
                serialPort.WriteLine(message);
            }
```


Chapter 57: Serial Ports

Section 57.1: Basic operation

```
var serialPort = new SerialPort("COM1", 9600, Parity.Even, 8, StopBits.One);
serialPort.Open();
serialPort.WriteLine("Test data");
string response = serialPort.ReadLine();
Console.WriteLine(response);
serialPort.Close();
```

Section 57.2: List available port names

```
string[] portNames = SerialPort.GetPortNames();
```

Section 57.3: Asynchronous read

```
void SetupAsyncRead(SerialPort serialPort)
{
    serialPort.DataReceived += (sender, e) => {
        byte[] buffer = new byte[4096];
        switch (e.EventType)
        {
            case SerialData.Chars:
                var port = (SerialPort)sender;
                int bytesToRead = port.BytesToRead;
                if (bytesToRead > buffer.Length)
                    Array.Resize(ref buffer, bytesToRead);
                int bytesRead = port.Read(buffer, 0, bytesToRead);
                // Process the read buffer here
                // ...
                break;
            case SerialData.Eof:
                // Terminate the service here
                // ...
                break;
        }
    };
```

Section 57.4: Synchronous text echo service

```
using System.IO.Ports;

namespace TextEchoService
{
    class Program
    {
        static void Main(string[] args)
        {
            var serialPort = new SerialPort("COM1", 9600, Parity.Even, 8, StopBits.One);
            serialPort.Open();
            string message = "";
            while (message != "quit")
            {
                message = serialPort.ReadLine();
                serialPort.WriteLine(message);
            }
```

```
serialPort.Close();
        }
    }
}
```

第57.5节：异步消息接收器

```
using System;
using System.Collections.Generic;
using System.IO.Ports;
using System.Text;
using System.Threading;

namespace AsyncReceiver
{
    class Program
    {
        const byte STX = 0x02;
        const byte ETX = 0x03;
        const byte ACK = 0x06;
        const byte NAK = 0x15;
        static ManualResetEvent terminateService = new ManualResetEvent(false);
        static readonly object eventLock = new object();
        static List<byte> unprocessedBuffer = null;

        static void Main(string[] args)
        {
            尝试
            {
                var serialPort = new SerialPort("COM11", 9600, Parity.Even, 8, StopBits.One);
                serialPort.DataReceived += DataReceivedHandler;
                serialPort.ErrorReceived += ErrorReceivedHandler;
                serialPort.Open();
terminateService.WaitOne();
                serialPort.Close();
            }
            catch (Exception e)
            {
Console.WriteLine("Exception occurred: {0}", e.Message);
            }
Console.ReadKey();
        }

        static void DataReceivedHandler(object sender, SerialDataReceivedEventArgs e)
        {
            lock (eventLock)
            {
                byte[] buffer = new byte[4096];
                switch (e.EventType)
                {
                    case SerialData.Chars:
                        var port = (SerialPort)sender;
                        int bytesToRead = port.BytesToRead;
                        if (bytesToRead > buffer.Length)
Array.Resize(ref buffer, bytesToRead);
                        int bytesRead = port.Read(buffer, 0, bytesToRead);
                        ProcessBuffer(buffer, bytesRead);
                        break;
                    case SerialData.Eof:
terminateService.Set();
                        break;
```



```
            serialPort.Close();
        }
    }
}
```

Section 57.5: Asynchronous message receiver

```
using System;
using System.Collections.Generic;
using System.IO.Ports;
using System.Text;
using System.Threading;

namespace AsyncReceiver
{
    class Program
    {
        const byte STX = 0x02;
        const byte ETX = 0x03;
        const byte ACK = 0x06;
        const byte NAK = 0x15;
        static ManualResetEvent terminateService = new ManualResetEvent(false);
        static readonly object eventLock = new object();
        static List<byte> unprocessedBuffer = null;

        static void Main(string[] args)
        {
            try
            {
                var serialPort = new SerialPort("COM11", 9600, Parity.Even, 8, StopBits.One);
                serialPort.DataReceived += DataReceivedHandler;
                serialPort.ErrorReceived += ErrorReceivedHandler;
                serialPort.Open();
                terminateService.WaitOne();
                serialPort.Close();
            }
            catch (Exception e)
            {
                Console.WriteLine("Exception occurred: {0}", e.Message);
            }
            Console.ReadKey();
        }

        static void DataReceivedHandler(object sender, SerialDataReceivedEventArgs e)
        {
            lock (eventLock)
            {
                byte[] buffer = new byte[4096];
                switch (e.EventType)
                {
                    case SerialData.Chars:
                        var port = (SerialPort)sender;
                        int bytesToRead = port.BytesToRead;
                        if (bytesToRead > buffer.Length)
                            Array.Resize(ref buffer, bytesToRead);
                        int bytesRead = port.Read(buffer, 0, bytesToRead);
                        ProcessBuffer(buffer, bytesRead);
                        break;
                    case SerialData.Eof:
                        terminateService.Set();
                        break;
```

```
                }
            }
        }
        static void ErrorReceivedHandler(object sender, SerialErrorReceivedEventArgs e)
        {
            lock (eventLock)
                if (e.EventType == SerialError.TXFull)
                {
Console.WriteLine("错误：TXFull。无法处理！");
                    terminateService.Set();
                }
                否则
                {
Console.WriteLine("错误：{0}。正在重置所有内容", e.EventType);
                    var port = (SerialPort)sender;
port.DiscardInBuffer();
                    port.DiscardOutBuffer();
                    unprocessedBuffer = null;
                    port.Write(new byte[] { NAK }, 0, 1);
                }
        }

        static void ProcessBuffer(byte[] buffer, int length)
        {
List<byte> message = unprocessedBuffer;
            for (int i = 0; i < length; i++)
                if (buffer[i] == ETX)
                {
                    if (message != null)
                    {
Console.WriteLine("MessageReceived: {0}",
                            Encoding.ASCII.GetString(message.ToArray()));
                        message = null;
                    }
                }
                else if (buffer[i] == STX)
                    message = null;
                else if (message != null)
                    message.Add(buffer[i]);
            unprocessedBuffer = message;
        }
    }
}
```

该程序等待被 STX 和 ETX 字节包围的消息，并输出它们之间的文本内容。
其他内容均被丢弃。写缓冲区溢出时程序停止。遇到其他错误时，重置输入和输出缓冲区并等待后续消息。

代码演示了：

- 异步串口读取（参见 SerialPort.DataReceived 的用法）。
- 串口错误处理（参见 SerialPort.ErrorReceived 的用法）。
- 基于非文本消息的协议实现。
- 部分消息读取。
    - SerialPort.DataReceived 事件可能会在整个消息（直到 ETX）到达之前发生。整个消息也可能不在输入缓冲区中（SerialPort.Read(..., ..., port.BytesToRead) 只读取消息的一部分）。在这种情况下，我们会暂存接收到的部分（unprocessedBuffer），并继续等待更多数据。
- 处理一次性接收多个消息的情况。


```
                }
            }
        }
        static void ErrorReceivedHandler(object sender, SerialErrorReceivedEventArgs e)
        {
            lock (eventLock)
                if (e.EventType == SerialError.TXFull)
                {
                    Console.WriteLine("Error: TXFull. Can't handle this!");
                    terminateService.Set();
                }
                else
                {
                    Console.WriteLine("Error: {0}. Resetting everything", e.EventType);
                    var port = (SerialPort)sender;
                    port.DiscardInBuffer();
                    port.DiscardOutBuffer();
                    unprocessedBuffer = null;
                    port.Write(new byte[] { NAK }, 0, 1);
                }
        }

        static void ProcessBuffer(byte[] buffer, int length)
        {
            List<byte> message = unprocessedBuffer;
            for (int i = 0; i < length; i++)
                if (buffer[i] == ETX)
                {
                    if (message != null)
                    {
                        Console.WriteLine("MessageReceived: {0}",
                            Encoding.ASCII.GetString(message.ToArray()));
                        message = null;
                    }
                }
                else if (buffer[i] == STX)
                    message = null;
                else if (message != null)
                    message.Add(buffer[i]);
            unprocessedBuffer = message;
        }
    }
}
```

This program waits for messages enclosed in STX and ETX bytes and outputs the text coming between them. Everything else is discarded. On write buffer overflow it stops. On other errors it reset input and output buffers and waits for further messages.

The code illustrates:

- Asynchronous serial port reading (see `SerialPort.DataReceived` usage).
- Serial port error processing (see `SerialPort.ErrorReceived` usage).
- Non-text message-based protocol implementation.
- Partial message reading.
    - The `SerialPort.DataReceived` event may happen earlier than entire message (up to ETX) comes. The entire message may also not be available in the input buffer (SerialPort.Read(..., ..., port.BytesToRead) reads only a part of the message). In this case we stash the received part (`unprocessedBuffer`) and carry on waiting for further data.
- Dealing with several messages coming in one go.

- SerialPort.DataReceived 事件可能只在对端发送了多个消息之后才发生。

- The `SerialPort.DataReceived` event may happen only after several messages have been sent by the other end.

附录 A：缩略语词汇表

A.1 节：.Net 相关缩略语

请注意，像 JIT 和 GC 这样的术语足够通用，适用于许多编程语言环境和运行时。

CLR：公共语言运行时
IL：中间语言
EE：执行引擎
JIT：即时编译器
GC：垃圾回收器
OOM：内存溢出
STA：单线程单元
MTA：多线程单元

Appendix A: Acronym Glossary

Section A.1: .Net Related Acronyms

Please note that some terms like JIT and GC are generic enough to apply to many programming language environments and runtimes.

CLR: Common Language Runtime
IL: Intermediate Language
EE: Execution Engine
JIT: Just-in-time compiler
GC: Garbage Collector
OOM: Out of memory
STA: Single-threaded apartment
MTA: Multi-threaded apartment

致谢

非常感谢所有来自Stack Overflow Documentation的人员提供此内容，
更多更改可发送至web@petercv.com以发布或更新新内容

阿迪·莱斯特	第42章和第47章
阿迪尔·马马多夫	第8章
阿德里亚诺·雷佩蒂	第1章、第2章、第4章和第18章
阿克谢·阿南德	第25章
艾伦·麦克比	第12章、第14章和第47章
ale10ander	第16章和第35章
亚历克斯·安德烈耶夫	第10章、第52章和第53章
亚历山大·曼特	第49章
亚历山大·V.	第8章
阿尔弗雷德·迈尔斯	第47章
阿曼·沙尔马	第5章和第42章
安德鲁·詹斯	第1章
安德鲁·莫顿	第25章
安德烈·谢金	第28章
安德里乌斯	第32章
阿尼克·萨哈	第28章
远日点	第34章
阿尔文·巴凯	第47章
阿尔克斯	第19章
阿什托尼亚人	第28章
阿塔里	第1章
avat	第46章
阿克萨里达克斯	第51章
BananaSft	第47章
Bassie	第48章
贝赫扎德	第39章
本杰明·霍奇森	第8章
binki	第45章
比约恩	第4章和第10章
布拉德利·格兰杰	第8章
布鲁诺·加西亚	第8章
BrunoLM	第15章
卡洛斯·穆尼奥斯	第8章
CodeCaster	第8、17、28、47和56章
丹尼尔·A·怀特	第1和25章
达雷尔·李	第4章
戴夫·R.	第47章
dbasnett	第35章
删除我	第18章
demonplus	第30章和第49章
Denuath	第15章
DLeh	第31章和第33章
德米特里·叶戈罗夫	第27章和第57章
DoNot	第8章
罗布·兰博士	第9章
德鲁	第25章
德鲁乔丹	第12章和第45章

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

Adi Lester	Chapters 42 and 47
Adil Mammadov	Chapter 8
Adriano Repetti	Chapters 1, 2, 4 and 18
Akshay Anand	Chapter 25
Alan McBee	Chapters 12, 14 and 47
ale10ander	Chapters 16 and 35
Aleks Andreev	Chapters 10, 52 and 53
Alexander Mandt	Chapter 49
Alexander V.	Chapter 8
Alfred Myers	Chapter 47
Aman Sharma	Chapters 5 and 42
Andrew Jens	Chapter 1
Andrew Morton	Chapter 25
Andrey Shchekin	Chapter 28
Andrius	Chapter 32
Anik Saha	Chapter 28
Aphelion	Chapter 34
Arvin Baccay	Chapter 47
Arxae	Chapter 19
Ashtonian	Chapter 28
Athari	Chapter 1
avat	Chapter 46
Axarydax	Chapter 51
BananaSft	Chapter 47
Bassie	Chapter 48
Behzad	Chapter 39
Benjamin Hodgson	Chapter 8
binki	Chapter 45
Bjørn	Chapters 4 and 10
Bradley Grainger	Chapter 8
Bruno Garcia	Chapter 8
BrunoLM	Chapter 15
Carlos Muñoz	Chapter 8
CodeCaster	Chapters 8, 17, 28, 47 and 56
Daniel A. White	Chapters 1 and 25
Darrel Lee	Chapter 4
Dave R.	Chapter 47
dbasnett	Chapter 35
delete me	Chapter 18
demonplus	Chapters 30 and 49
Denuath	Chapter 15
DLeh	Chapters 31 and 33
Dmitry Egorov	Chapters 27 and 57
DoNot	Chapter 8
Dr Rob Lang	Chapter 9
Drew	Chapter 25
DrewJordan	Chapters 12 and 45

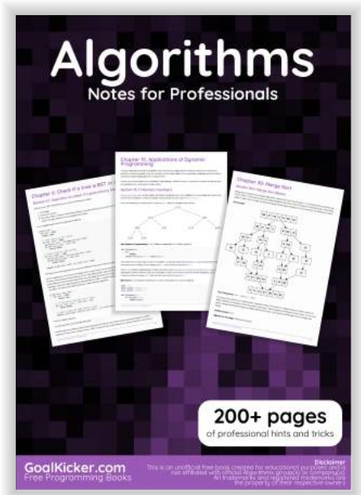
爱德华多·莫尔特尼	第8章
艾赫桑·萨贾德	第8章
埃里克	第32章
费利佩·奥里亚尼	第4章和第5章
菲利普·弗a茨	第17章
弗雷杜	第48章
加杰德拉	第37章
银河牛仔	第3章和第8章
乔治·波列沃伊	第4章、第11章和第34章
关系	第22章
古斯多尔	第43章和第44章
哈尼	第8章
哈里奥特	第5章
斧头	第4章
海因茨	第25章
霍根	第4章
海威尔·里斯	第7章
i3arnon	第38章
伊恩	第4章
伊戈尔	第25章
Ingenioushax	第16章
Jacobr365	第38、42和43章
贾加迪沙·B·S	第49章
JamyRyals	第42章
jbtule	第8章
吉加尔	第10章
吉姆	第11章
jnovo	第8章
乔·阿门塔	第8章和第20章
约翰	第3章
凯文·蒙特罗斯	第1章
Konamiman	第8、42、54、55和56章
克里科尔·艾兰吉安	第36章
克里特纳	第47章
lokusking	第49章
洛伦佐·德马特	第10章
Luaan	第23章
卢卡斯·特热斯涅夫斯基	第9章
M22an	第32章
Mafii	第47章
马赫迪·阿巴西	第50章
马尔钦·尤拉谢克	第1章和第8章
马克·C.	第5章
马塔斯·瓦伊特凯维丘斯	第10章和第28章
马蒂亚斯·穆勒	第42章
马特	第49章
马特 dc	第15章
马蒂亚	第1章
马修·怀特德	第13章
麦凯	第8章
梅洛	第39章
米哈伊尔·斯坦切斯库	第24章
明多尔先生	第8章

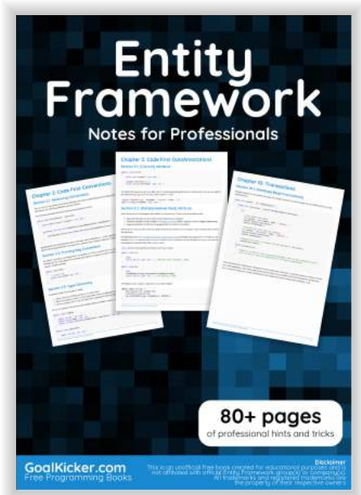
Eduardo Molteni	Chapter 8
Ehsan Sajjad	Chapter 8
Eric	Chapter 32
Felipe Oriani	Chapters 4 and 5
Filip Frącz	Chapter 17
Fredou	Chapter 48
Gajendra	Chapter 37
GalacticCowboy	Chapters 3 and 8
George Polevoy	Chapters 4, 11 and 34
Guanxi	Chapter 22
Gusdor	Chapters 43 and 44
Haney	Chapter 8
harriyott	Chapter 5
hatchet	Chapter 4
Heinzi	Chapter 25
Hogan	Chapter 4
Hywel Rees	Chapter 7
i3arnon	Chapter 38
Ian	Chapter 4
Igor	Chapter 25
Ingenioushax	Chapter 16
Jacobr365	Chapters 38, 42 and 43
Jagadisha B S	Chapter 49
JamyRyals	Chapter 42
jbtule	Chapter 8
Jigar	Chapter 10
Jim	Chapter 11
jnovo	Chapter 8
Joe Amenta	Chapters 8 and 20
John	Chapter 3
Kevin Montrose	Chapter 1
Konamiman	Chapters 8, 42, 54, 55 and 56
Krikor Ailanjian	Chapter 36
Kritner	Chapter 47
lokusking	Chapter 49
Lorenzo Dematté	Chapter 10
Luaan	Chapter 23
Lucas Trzesniewski	Chapter 9
M22an	Chapter 32
Mafii	Chapter 47
mahdi abasi	Chapter 50
MarcinJuraszek	Chapters 1 and 8
Mark C.	Chapter 5
Matas Vaitkevicius	Chapters 10 and 28
Mathias Müller	Chapter 42
Matt	Chapter 49
Matt dc	Chapter 15
matteeyah	Chapter 1
Matthew Whited	Chapter 13
McKay	Chapter 8
Mellow	Chapter 39
Mihail Stancescu	Chapter 24
Mr.Mindor	Chapter 8

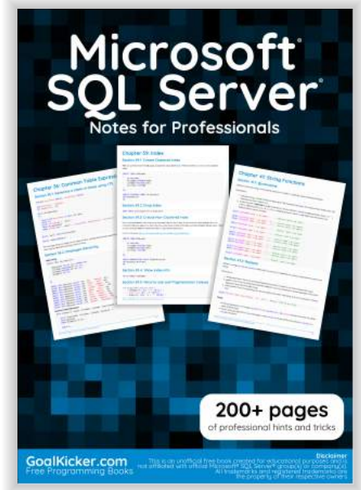
MSE	第40章
n.podbielski	第11章
内特·巴贝蒂尼	第8章
尼古拉·卢科维奇	第38章
尼古拉·康德拉季耶夫	第23章
奥格拉斯	第48章
Ondřej Štorc	第48章
奥扎尔·卡弗雷	第28章
帕维尔·马约罗夫	第42章
帕维尔·沃罗宁	第2章和第42章
佩德罗·索基	第32章
ProgramFOX	第21章
林吉尔	第4章
里昂·威廉姆斯	第1章
罗伯特·哥伦比亚	第4章
皇家土豆	第17章
鲁本·斯坦斯	第8章
萨尔瓦多·鲁比奥·马丁内斯	第8章
萨米	第8章
斯科特·汉嫩	第26章和第29章
SeeuD1	第1章
塞尔吉奥·多明格斯	第8章
Sidewinder94	第8章
smdrager	第10章
starbeamrainbowlabs	第37章和第47章
史蒂夫	第30章
史蒂文·多加特	第1章
斯蒂尔加尔	第11章
坦维尔·巴达尔	第58章
tehDorf	第6章和第15章
狂战士	第4章
西奥多罗斯 哈齐吉安纳基斯	第37章
托马斯·布莱德索	第42章
斯里格尔	第32章
托德莫	第23章
托尔加·埃夫西门	第32章
托马š 休贝尔鲍尔	第4章
user2321864	第25章
维基	第30章
王恩正	第11章
亚夫菲	第41章
ʇoleəz əʊʇ bqo	第1章

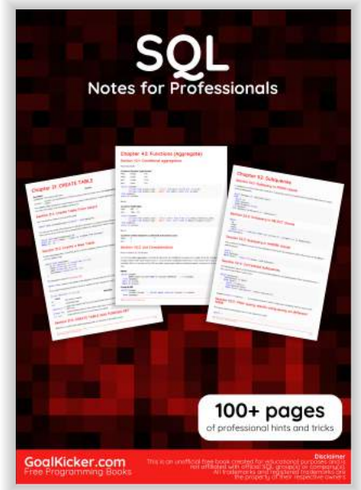
MSE	Chapter 40
n.podbielski	Chapter 11
Nate Barbettini	Chapter 8
Nikola.Lukovic	Chapter 38
NikolayKondratyev	Chapter 23
Ogglas	Chapter 48
Ondřej Štorc	Chapter 48
Ozair Kafray	Chapter 28
Pavel Mayorov	Chapter 42
Pavel Voronin	Chapters 2 and 42
PedroSouki	Chapter 32
ProgramFOX	Chapter 21
Ringil	Chapter 4
Rion Williams	Chapter 1
Robert Columbia	Chapter 4
RoyalPotato	Chapter 17
Ruben Steins	Chapter 8
Salvador Rubio Martinez	Chapter 8
Sammi	Chapter 8
Scott Hannen	Chapters 26 and 29
SeeuD1	Chapter 1
Sergio Domínguez	Chapter 8
Sidewinder94	Chapter 8
smdrager	Chapter 10
starbeamrainbowlabs	Chapters 37 and 47
Steve	Chapter 30
Steven Doggart	Chapter 1
Stilgar	Chapter 11
Tanveer Badar	Chapter 58
tehDorf	Chapters 6 and 15
the berserker	Chapter 4
Theodoros Chatzigiannakis	Chapter 37
Thomas Bledsoe	Chapter 42
Thriggle	Chapter 32
toddmo	Chapter 23
Tolga Evcimen	Chapter 32
Tomáš Hübelbauer	Chapter 4
user2321864	Chapter 25
vicky	Chapter 30
wangengzheng	Chapter 11
Yahfoufi	Chapter 41
ʇoleəz əʊʇ qoq	Chapter 1

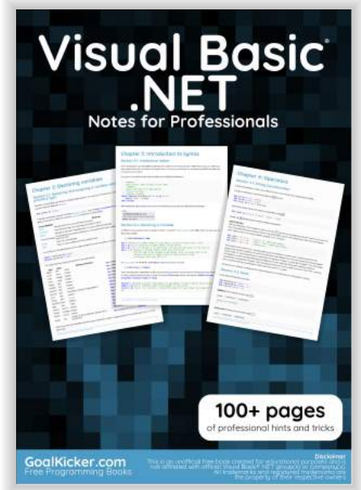
你可能也喜欢











You may also like