C# 高级编程

书栈(BookStack.CN)

目　录

致谢

当前文档 《C# 高级编程》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-07-15。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

内容来源：沈军 https://shenjun.gitbooks.io/c/content/

文档地址：http://www.bookstack.cn/books/shenjun-csharp-advanced

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

C# 高级编程

本教程分为基础部分及高级部分

一、C#、.NET Framework、CLR的关系

很多人没有将C#、.NET Framework（.NET框架）、CLR（Common Language Runtime，公共语言运行库）这三者之间的关系区分清楚，认为其版本号是一一对应的。其实不然，.NET框架是一个独立发布的软件包，其包含了CLR、类库以及相关的语言编辑器等工具。C#代码经过编译之后在CLR环境中运行。

由于.NET框架3.0/3.5其实是.NET2.0的扩展（只是增加了一些新的程序集），所以.NET3.0/3.5的CLR版本还是2.0。而且.NET3.0其实只扩展了WF、WPF、WCF、WCS等组件，并没有提供新的C#编译器，直到.NET3.5中才打包了C#3.0的编译器。.NET4.0在3.0上针对WF、WCF进行了一些新功能增加，所以.NET框架、CLR和C#的版本之间的对应关系如下表所示：

.NET版本	1.0	1.1	2.0	3.0	3.5	4.0	4.5	4.6
CLR版本	1.0	1.1	2.0	2.0	2.0	4.0	4.0	4.0
C#版本	1.0	1.2	2.0	3.0	3.0	4.0	5.0	6.0
Visual Studio	VS.NET 2002	VS.NET 2003	VS 2005	VS 2008	VS 2010	VS 2010	VS2012	VS 2015

也就是说，对于那些不涉及新程序集的C#3.0新特性（比如自动属性、匿名属性等）在.NET2.0的环境中也可以运行，CLR对这些特性是一无所知的。

1、C#是一种面向对象编程语言，是为开发.NET框架上的程序而设计的。

（1）C#是由C和C++衍生出来的，所以其可调用由 C/C++ 编写的本机原生函数，同时不损失C/C++原有的强大的功能。

（2）C#所开发的程序源代码并不是编译成能够直接在操作系统上执行的二进制本地代码。它是被编译成为中间代码，然后通过.NET框架的虚拟机（即CLR）来执行。所以如果计算机上没有安装.Net框架，那么程序将不能够被执行。在执行的过程中，.Net框架会将中间代码翻译成为二进制机器码，从而使它得到正确的运行。最终翻译的二进制代码将被存储在一个缓冲区中。所以一旦程序使用了相同的代码，那么将会调用缓冲区中的版本。这样如果一个.Net程序第二次被运行，那么这种翻译不需要进行第二次，速度会明显加快。

2、.NET框架有三部分组成（如下图）：

1）CLR的介绍参考3

2）编程工具：涵盖了编码和调试需要的一切:包含：VisualStudio集成开发环境、.NET兼容的编译器（例如：C#、VB、JScript和托管的C++）、调试器、服务器端改进（比如ASP.NET）

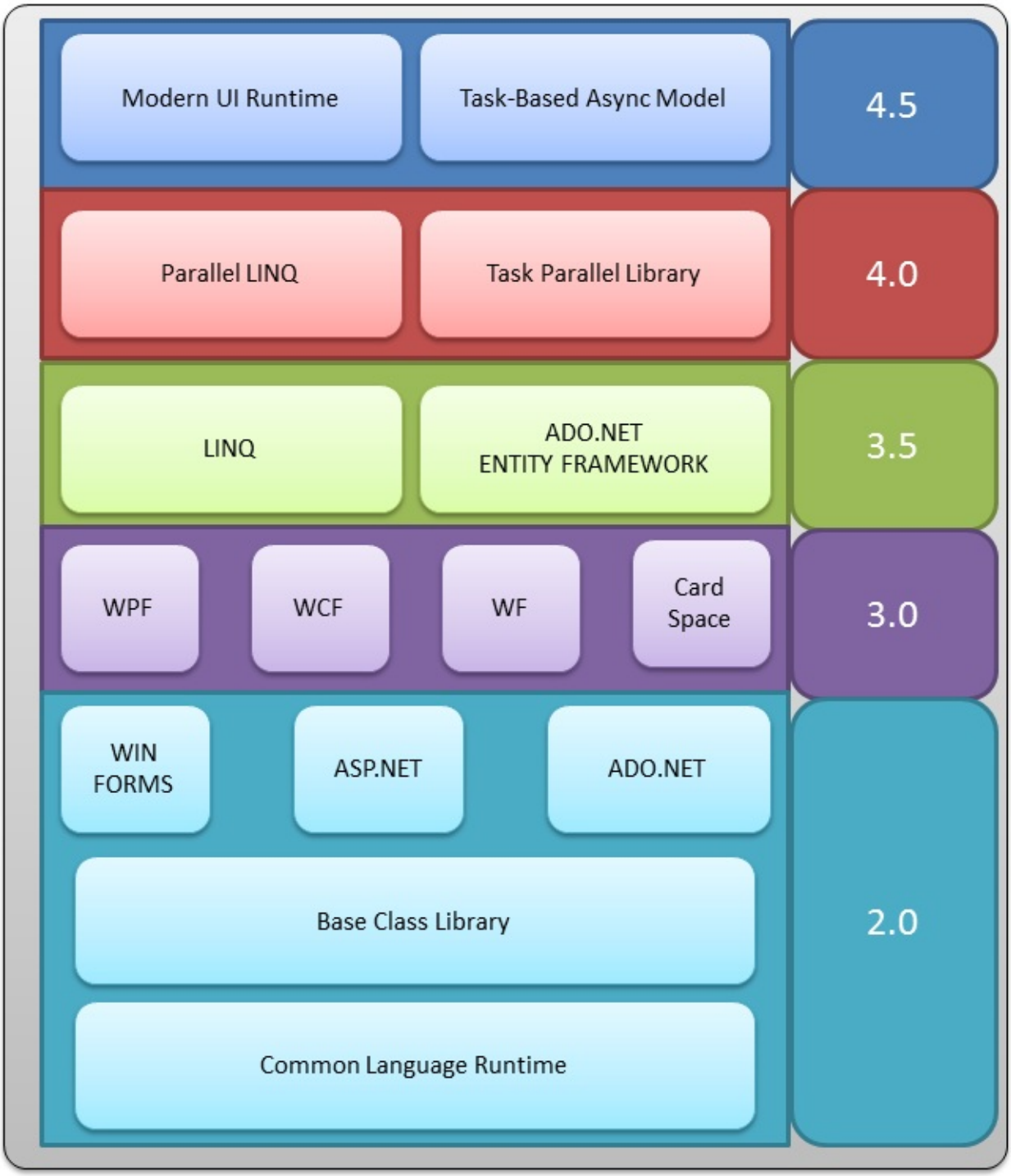
3）BCL（Base Class Library，基类库）：是.NET框架使用的一个大的类库，而且也可以在你的程序中使用。包括以下一些类。

1>通用基础类：这些类提供了一组极为强大的工具，可以应用带广泛的编程任务中，比如字符串操作、安全和加密。

2>集合类：这些类实现了列表、字典、散列表以及位数组。

3>线程和同步类：这些类用于创建多线程程序。

4>XML类，这些类用于创建、读取以及操作XML文档。



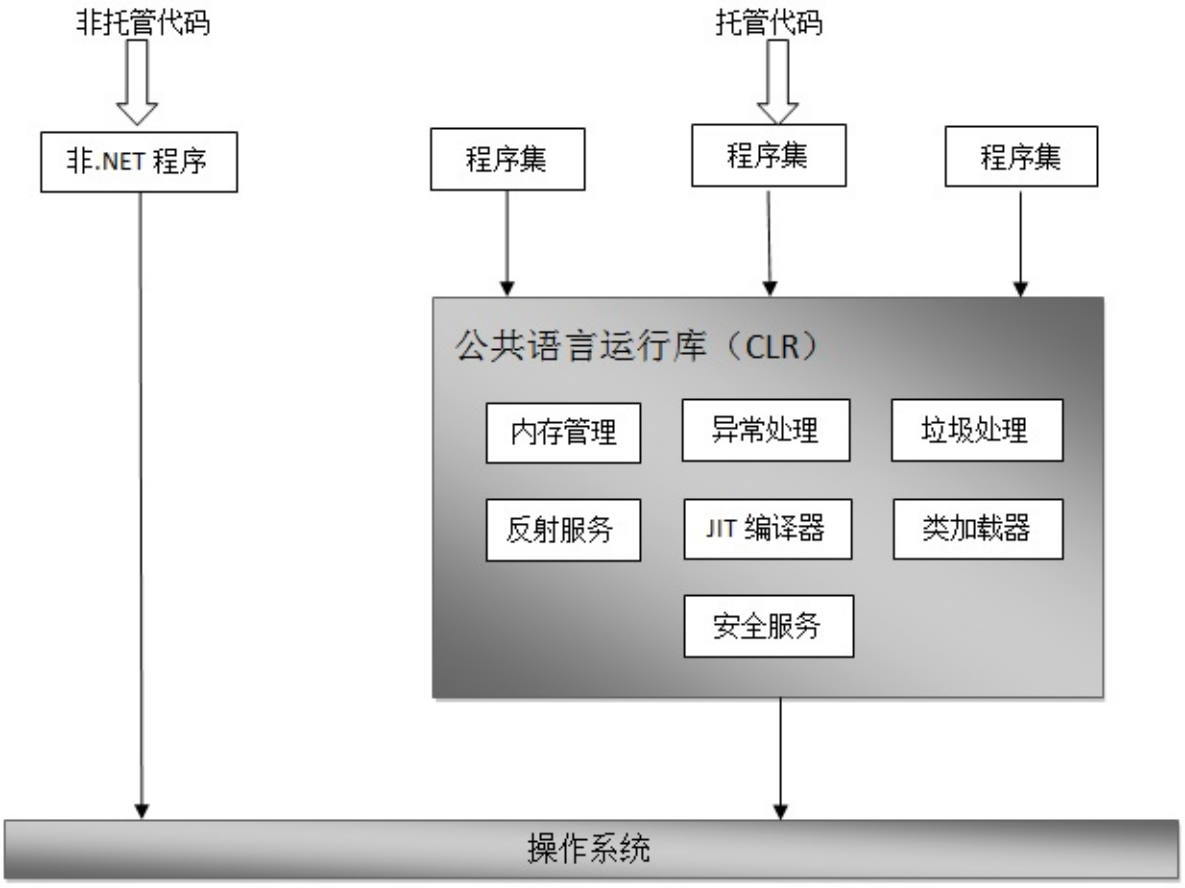
3、CLR（公共语言运行库）在运行期管理程序的执行：主要包含：内存管理、代码安全验证、代码执行、垃圾收集。如下图

（1）自动垃圾收集：CLR有一项服务称为GC（Garbage Collector，垃圾收集），它能为你自动管理内存。

1）GC自动从内存中删除程序不再访问的对象

2）GC是程序员不再操心许多以前必须执行的任务，比如释放内存和检查内存泄漏。这可不是

小特性，因为检查内存泄漏可能非常困难而且耗时。

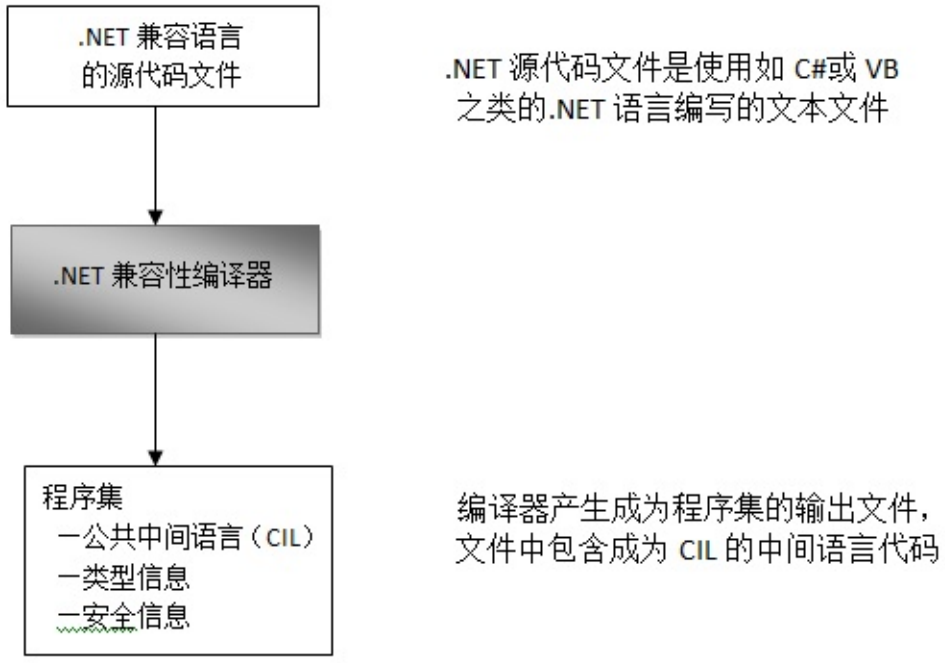


4、代码的编译过程：

（1）编译成**CIL**：.NET语言的编译器接受源代码文件，并生成名为程序集的输出文件。程序集可以是可执行文件或DLL（如下图所示）

1>程序集里的代码并不是本机代码，而是一种名称为CIL（Common Intermediate Language，公共中间语言）的中间代码。

2>程序集包含的信息中，包含下列项目：程序的CIL、程序中使用的类型的元数据、对其他程序集引用的元数据



（**2**）编译成本机代码并执行：程序的CIL直到它被调用运行时才会被编译成本机代码。在运行时，CLR执行下面的步骤（如下图）

1>检查程序集的安全特性

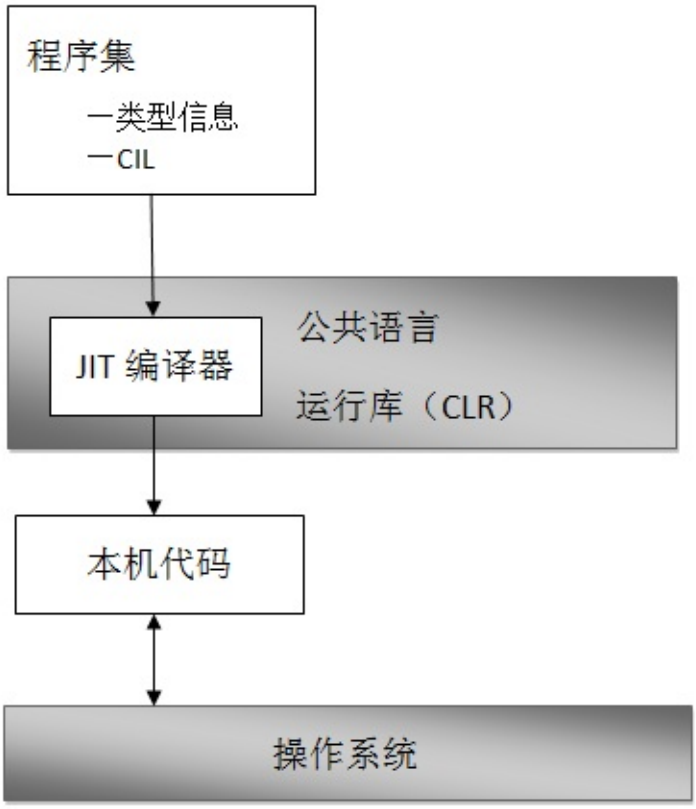
2>在内存中分配空间

3>把程序集中的可执行代码发送给实时（Just-in-Time）编译器，把其中的一部分编译成本机代码。

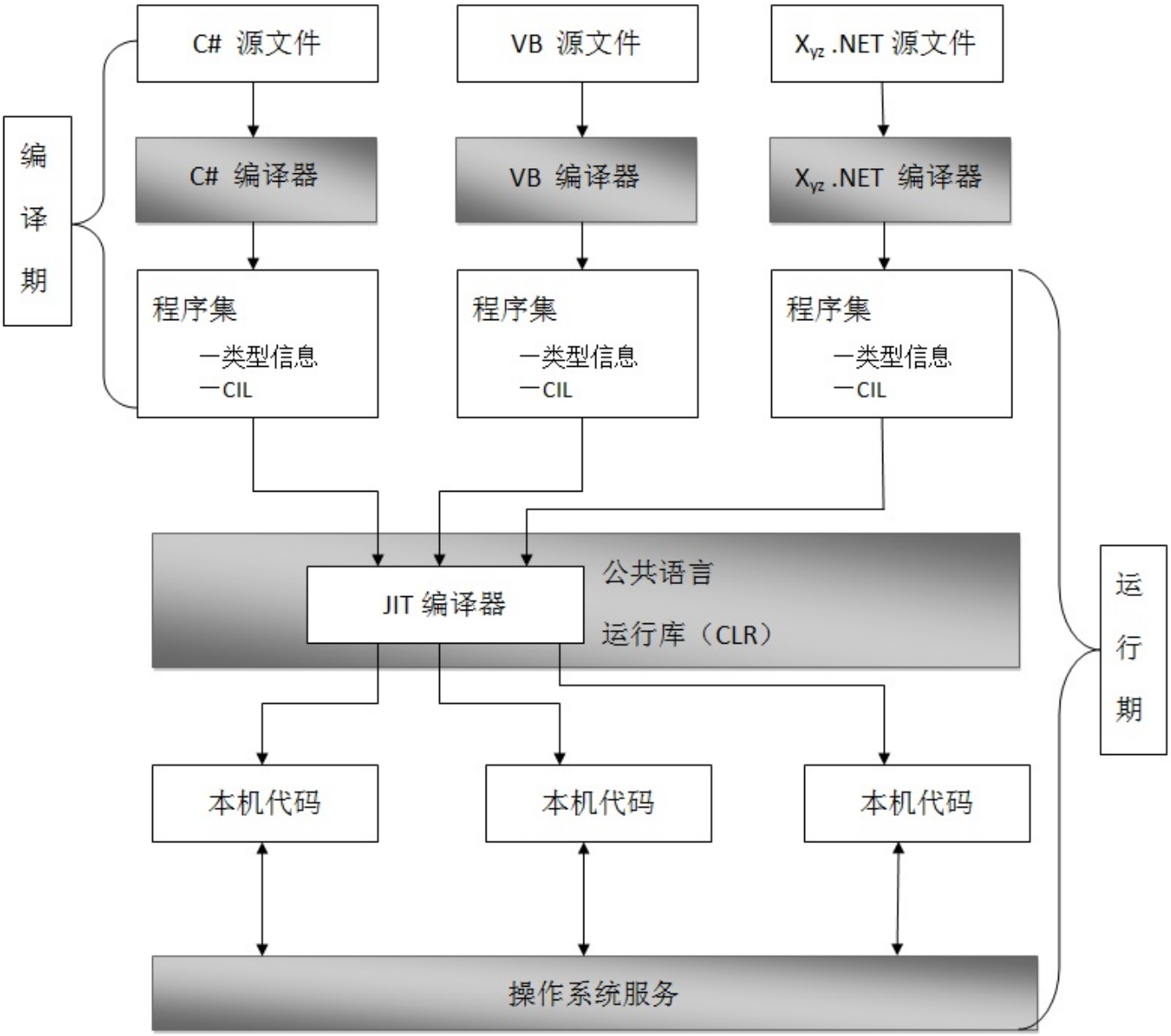
程序集中的可执行代码在需要的时候由实时编译器编译，然后它就被缓存以备在后来的程序中执行，使用这个方法意味着不被调用的代码不会被编译成本机代码，而且被调用到的代码只被编译一次。

一旦CIL被编译成本机代码，CLR就在它运行时管理它，执行像释放无主内存、检查数组边界、检查参数类型和管理异常之类的任务。这里产生了两个重要的术语：

```
1.    托管代码：为.NET框架编写的代码称为托管代码，需要CLR。
2.
3.    非托管代码：不在CLR控制之下运行的代码，比如Win32C/C++ DLL，成为非托管代码。
```



（3）编译和执行综述：无论原始源文件的语言是什么，懂遵循同样的编译和执行过程。


C# 源文件
VB 源文件
Xyz .NET 源文件
编译期
C# 编译器
VB 编译器
Xyz .NET 编译器
程序集
—类型信息
—CIL
程序集
—类型信息
—CIL
程序集
—类型信息
—CIL
JIT 编译器
公共语言
运行库（CLR）
运行期
本机代码
本机代码
本机代码
操作系统服务

C# 简介

C# 是一个现代的、通用的、面向对象的编程语言，它是由微软（Microsoft）开发的，由 Ecma 和 ISO 核准认可的。

C# 是由 Anders Hejlsberg 和他的团队在 .Net 框架开发期间开发的。

C# 是专为公共语言基础结构（CLI）设计的。CLI 由可执行代码和运行时环境组成，允许在不同的计算机平台和体系结构上使用各种高级语言。

下面列出了 C# 成为一种广泛应用的专业语言的原因：

1. • 现代的、通用的编程语言。
2. • 面向对象。
3. • 面向组件。
4. • 容易学习。
5. • 结构化语言。
6. • 它产生高效率的程序。
7. • 它可以在多种计算机平台上编译。
8. • .Net 框架的一部分。

C# 强大的编程功能

虽然 C# 的构想十分接近于传统高级语言 C 和 C++，是一门面向对象的编程语言，但是它与 Java 非常相似，有许多强大的编程功能，因此得到广大程序员的亲睐。下面列出 C# 一些重要的功能：

1. • 布尔条件（Boolean Conditions）
2. • 自动垃圾回收（Automatic Garbage Collection）
3. • 标准库（Standard Library）
4. • 组件版本（Assembly Versioning）
5. • 属性（Properties）和事件（Events）
6. • 委托（Delegates）和事件管理（Events Management）
7. • 易于使用的泛型（Generics）
8. • 索引器（Indexers）
9. • 条件编译（Conditional Compilation）
10. • 简单的多线程（Multithreading）
11. • LINQ 和 Lambda 表达式
12. • 集成 Windows

C# 数据类型

在 C# 中，变量分为以下几种类型：

```
1.  • 值类型（Value types）
2.  • 引用类型（Reference types）
3.  • 指针类型（Pointer types）
```

一、值类型（Value types）

值类型变量可以直接分配给一个值。它们是从类 `System.ValueType` 中派生的。值类型直接包含数据。比如 `int` 、 `char` 、 `float` ，它们分别存储数字、字母、浮点数。当您声明一个 `int` 类型时，系统分配内存来存储值。

下表列出了 C# 2010 中可用的值类型：

类型	描述	范围	默认值
bool	布尔值	True 或 False	False
byte	8 位无符号整数	0 到 255	0
char	16 位 Unicode 字符	U +0000 到 U +ffff	'\0'
decimal	128 位精确的十进制值，28-29 有效位数	(-7.9 x 1028 到 7.9 x 1028) / 100 到 28	0.0M
double	64 位双精度浮点型	(+/-)5.0 x 10-324 到 (+/-)1.7 x 10308	0.0D
float	32 位单精度浮点型	-3.4 x 1038 到 + 3.4 x 1038	0.0F
int	32 位有符号整数类型	-2,147,483,648 到 2,147,483,647	0
long	64 位有符号整数类型	-923,372,036,854,775,808 到 9,223,372,036,854,775,807	0L
sbyte	8 位有符号整数类型	-128 到 127	0
short	16 位有符号整数类型	-32,768 到 32,767	0
uint	32 位无符号整数类型	0 到 4,294,967,295	0
ulong	64 位无符号整数类型	0 到 18,446,744,073,709,551,615	0
ushort	16 位无符号整数类型	0 到 65,535	0

如需得到一个类型或一个变量在特定平台上的准确尺寸，可以使用 `sizeof` 方法。表达式 `sizeof(type)` 产生以字节为单位存储对象或类型的存储尺寸。下面举例获取任何机器上 `int` 类型的存储尺寸：

```
1.      namespace DataTypeApplication
2.      {
```

```
        class Program
        {
            static void Main(string[] args)
            {
                Console.WriteLine("Size of int: {0}", sizeof(int));
                Console.ReadLine();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    Size of int: 4
```

二、引用类型（Reference types）

引用类型不包含存储在变量中的实际数据，但它们包含对变量的引用。换句话说，它们指的是一个内存位置。使用多个变量时，引用类型可以指向一个内存位置。如果内存位置的数据是由一个变量改变的，其他变量会自动反映这种值的变化。内置的 引用类型有：`object`、`dynamic` 和 `string`。

（1）、对象（Object）类型

对象（`Object`）类型 是 C# 通用类型系统（Common Type System - CTS）中所有数据类型的终极基类。`Object` 是 `System.Object` 类的别名。所以对象（`Object`）类型可以被分配任何其他类型（值类型、引用类型、预定义类型或用户自定义类型）的值。但是，在分配值之前，需要先进行类型转换。当一个值类型转换为对象类型时，则被称为 装箱；另一方面，当一个对象类型转换为值类型时，则被称为 拆箱。

```
    object obj;
    obj = 100; // 这是装箱
```

（2）、动态（Dynamic）类型

您可以存储任何类型的值在动态数据类型变量中。这些变量的类型检查是在运行时发生的。声明动态类型的语法：

```
    dynamic <variable_name> = value;
```

例如：

```
    dynamic d = 20;
```

动态类型与对象类型相似，但是对象类型变量的类型检查是在编译时发生的，而动态类型变量的类型检查是在运行时发生的。

（3）、字符串（String）类型

字符串（ `String` ）类型 允许您给变量分配任何字符串值。字符串（ `String` ）类型是 `System.String` 类的别名。它是从对象（ `Object` ）类型派生的。字符串（ `String` ）类型的值可以通过两种形式进行分配：引号和 @引号。

例如：

```
1.      String str = "runoob.com";
```

一个 @引号字符串：

```
1.      @"runoob.com";
```

C# `string` 字符串的前面可以加 @（称作"逐字字符串"）将转义字符（\）当作普通字符对待，比如：

```
1.      string str = @"C:\Windows";
```

等价于：

```
1.      string str = "C:\\Windows";
```

@ 字符串中可以任意换行，换行符及缩进空格都计算在字符串长度之内。

```
1.          string str = @"<script type=""text/javascript"">
2.              <!--
3.              -->
4.          </script>";
```

用户自定义引用类型有： `class` 、 `interface` 或 `delegate` 。我们将在以后的章节中讨论这些类型。

三、指针类型（Pointer types）

指针类型变量存储另一种类型的内存地址。C# 中的指针与 C 或 C++ 中的指针有相同的功能。

声明指针类型的语法：

```
type* identifier;
```

例如：

```
char* cptr;
int* iptr;
```

我们将在章节"不安全的代码"中讨论指针类型。

C# 变量

一个变量只不过是一个供程序操作的存储区的名字。在 C# 中，每个变量都有一个特定的类型，类型决定了变量的内存大小和布局。范围内的值可以存储在内存中，可以对变量进行一系列操作。我们已经讨论了各种数据类型。C# 中提供的基本的值类型大致可以分为以下几类：

类型	举例
整数类型	sbyte、byte、short、ushort、int、uint、long、ulong 和 char
浮点型	float 和 double
十进制类型	decimal
布尔类型	true 或 false 值，指定的值
空类型	可为空值的数据类型

C# 允许定义其他值类型的变量，比如 `enum` ，也允许定义引用类型变量，比如 `class` 。这些我们将在以后的章节中进行讨论。在本章节中，我们只研究基本变量类型。

一、C# 中的变量定义

C# 中变量定义的语法：

```
1.      <data_type> <variable_list>;
```

在这里， `data_type` 必须是一个有效的 C# 数据类型，可以是 `char` 、 `int` 、 `float` 、 `double` 或其他用户自定义的数据类型。 `variable_list` 可以由一个或多个用逗号分隔的标识符名称组成。

一些有效的变量定义如下所示：

```
1.      int i, j, k;
2.      char c, ch;
3.      float f, salary;
4.      double d;
```

您可以在变量定义时进行初始化：

```
1.      int i = 100;
```

二、C# 中的变量初始化

变量通过在等号后跟一个常量表达式进行初始化（赋值）。初始化的一般形式为：

```
variable_name = value;
```

变量可以在声明时被初始化（指定一个初始值）。初始化由一个等号后跟一个常量表达式组成，如下所示：

```
<data_type> <variable_name> = value;
```

一些实例：

```
int d = 3, f = 5;    /* 初始化 d 和 f. */
byte z = 22;         /* 初始化 z. */
double pi = 3.14159; /* 声明 pi 的近似值 */
char x = 'x';        /* 变量 x 的值为 'x' */
```

正确地初始化变量是一个良好的编程习惯，否则有时程序会产生意想不到的结果。

请看下面的实例，使用了各种类型的变量：

```
namespace VariableDefinition
{
    class Program
    {
        static void Main(string[] args)
        {
            short a;
            int b ;
            double c;

            /* 实际初始化 */
            a = 10;
            b = 20;
            c = a + b;
            Console.WriteLine("a = {0}, b = {1}, c = {2}", a, b, c);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.      a = 10, b = 20, c = 30
```

三、接受来自用户的值

`System` 命名空间中的 `Console` 类提供了一个函数 `ReadLine()`，用于接收来自用户的输入，并把它存储到一个变量中。

例如：

```
1.      int num;
2.      num = Convert.ToInt32(Console.ReadLine());
```

函数 `Convert.ToInt32()` 把用户输入的数据转换为 `int` 数据类型，因为 `Console.ReadLine()` 只接受字符串格式的数据。

四、C# 中的 Lvalues 和 Rvalues

C# 中的两种表达式： **lvalue:** lvalue 表达式可以出现在赋值语句的左边或右边。 **rvalue:** rvalue 表达式可以出现在赋值语句的右边，不能出现在赋值语句的左边。

变量是 `lvalue` 的，所以可以出现在赋值语句的左边，也可以出现在右边。数值是 `rvalue` 的，因此不能被赋值，不能出现在赋值语句的左边。下面是一个有效的语句：

```
1.      int g = 20;
2.      int a = g;
```

下面是一个无效的语句，会产生编译时错误：

```
1.      10 = 20;
```

C# 运算符

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。C# 有丰富的内置运算符，分类如下：

1. • 算术运算符
2. • 关系运算符
3. • 逻辑运算符
4. • 位运算符
5. • 赋值运算符
6. • 其他运算符

本教程将逐一讲解算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符及其他运算符。

一、算术运算符

下表显示了 C# 支持的所有算术运算符。假设变量 A 的值为 10，变量 B 的值为 20，则：

运算符	描述	实例
+	把两个操作数相加	A + B 将得到 30
-	从第一个操作数中减去第二个操作数	A - B 将得到 -10
	把两个操作数相乘	A B 将得到 200
/	分子除以分母	B / A 将得到 2
%	取模运算符，整除后的余数	B % A 将得到 0
++	自增运算符，整数值增加 1	A++ 将得到 11
—	自减运算符，整数值减少 1	A— 将得到 9

请看下面的实例，了解 C# 中所有可用的算术运算符：

```
1.      using System;
2.
3.      namespace OperatorsAppl
4.      {
5.          class Program
6.          {
7.              static void Main(string[] args)
8.              {
9.                  int a = 21;
10.                 int b = 10;
```

```
            int c;

            c = a + b;
            Console.WriteLine("Line 1 - c 的值是 {0}", c);
            c = a - b;
            Console.WriteLine("Line 2 - c 的值是 {0}", c);
            c = a * b;
            Console.WriteLine("Line 3 - c 的值是 {0}", c);
            c = a / b;
            Console.WriteLine("Line 4 - c 的值是 {0}", c);
            c = a % b;
            Console.WriteLine("Line 5 - c 的值是 {0}", c);

            // ++a 先进行自增运算再赋值
            c = ++a;
            Console.WriteLine("Line 6 - c 的值是 {0}", c);

            // 此时 a 的值为 22
            // --a 先进行自减运算再赋值
            c = --a;
            Console.WriteLine("Line 7 - c 的值是 {0}", c);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 31
Line 2 - c 的值是 11
Line 3 - c 的值是 210
Line 4 - c 的值是 2
Line 5 - c 的值是 1
Line 6 - c 的值是 22
Line 7 - c 的值是 21
```

```
• c = a++: 先将 a 赋值给 c，再对 a 进行自增运算。
• c = ++a: 先将 a 进行自增运算，再将 a 赋值给 c 。
• c = a--: 先将 a 赋值给 c，再对 a 进行自减运算。
• c = --a: 先将 a 进行自减运算，再将 a 赋值给 c 。
```

```
1.  using System;
2.
3.  namespace OperatorsAppl
4.  {
5.      class Program
6.      {
7.          static void Main(string[] args)
8.          {
9.              int a = 1;
10.             int b;
11.
12.             // a++ 先赋值再进行自增运算
13.             b = a++;
14.             Console.WriteLine("a = {0}", a);
15.             Console.WriteLine("b = {0}", b);
16.             Console.ReadLine();
17.
18.             // ++a 先进行自增运算再赋值
19.             a = 1; // 重新初始化 a
20.             b = ++a;
21.             Console.WriteLine("a = {0}", a);
22.             Console.WriteLine("b = {0}", b);
23.             Console.ReadLine();
24.
25.             // a-- 先赋值再进行自减运算
26.             a = 1;  // 重新初始化 a
27.             b= a--;
28.             Console.WriteLine("a = {0}", a);
29.             Console.WriteLine("b = {0}", b);
30.             Console.ReadLine();
31.
32.             // --a 先进行自减运算再赋值
33.             a = 1;  // 重新初始化 a
34.             b= --a;
35.             Console.WriteLine("a = {0}", a);
36.             Console.WriteLine("b = {0}", b);
37.             Console.ReadLine();
38.         }
39.     }
40. }
```

执行以上程序，输出结果为：

```
a = 2
b = 1
a = 2
b = 2
a = 0
b = 1
a = 0
b = 0
```

二、关系运算符

下表显示了 C# 支持的所有关系运算符。假设变量 A 的值为 10，变量 B 的值为 20，则：

运算符	描述	实例
==	检查两个操作数的值是否相等，如果相等则条件为真。	(A == B) 不为真。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。	(A != B) 为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。	(A > B) 不为真。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。	(A < B) 为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。	(A >= B) 不为真。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。	(A <= B) 为真。

请看下面的实例，了解 C# 中所有可用的关系运算符：

```
using System;

class Program
{
  static void Main(string[] args)
  {
      int a = 21;
      int b = 10;

      if (a == b)
      {
          Console.WriteLine("Line 1 - a 等于 b");
      }
      else
      {
          Console.WriteLine("Line 1 - a 不等于 b");
```

```
            }
            if (a < b)
            {
                Console.WriteLine("Line 2 - a 小于 b");
            }
            else
            {
                Console.WriteLine("Line 2 - a 不小于 b");
            }
            if (a > b)
            {
                Console.WriteLine("Line 3 - a 大于 b");
            }
            else
            {
                Console.WriteLine("Line 3 - a 不大于 b");
            }
            /* 改变 a 和 b 的值 */
            a = 5;
            b = 20;
            if (a <= b)
            {
               Console.WriteLine("Line 4 - a 小于或等于 b");
            }
            if (b >= a)
            {
               Console.WriteLine("Line 5 - b 大于或等于 a");
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    Line 1 - a 不等于 b
    Line 2 - a 不小于 b
    Line 3 - a 大于 b
    Line 4 - a 小于或等于 b
    Line 5 - b 大于或等于 a
```

三、逻辑运算符

下表显示了 C# 支持的所有逻辑运算符。假设变量 A 为布尔值 `true`，变量 B 为布尔值

`false`，则：

运算符	描述	实例
`&&`	称为逻辑与运算符。如果两个操作数都非零，则条件为真。	`(A && B)` 为假。
II	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(A II B) 为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	`!(A && B)` 为真。

请看下面的实例，了解 C# 中所有可用的逻辑运算符：

```
using System;

namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            bool a = true;
            bool b = true;

            if (a && b)
            {
               Console.WriteLine("Line 1 - 条件为真");
            }
            if (a || b)
            {
                Console.WriteLine("Line 2 - 条件为真");
            }
            /* 改变 a 和 b 的值 */
            a = false;
            b = true;
            if (a && b)
            {
                Console.WriteLine("Line 3 - 条件为真");
            }
            else
            {
                Console.WriteLine("Line 3 - 条件不为真");
```

```
                }
                if (!(a && b))
                {
                    Console.WriteLine("Line 4 - 条件为真");
                }
                Console.ReadLine();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    Line 1 - 条件为真
    Line 2 - 条件为真
    Line 3 - 条件不为真
    Line 4 - 条件为真
```

四、位运算符

位运算符作用于位，并逐位执行操作。 `&` 、 `|` 和 `^` 的真值表如下所示：

p	q	p & q	p I q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

假设如果 A = 60，且 B = 13，现在以二进制格式表示，它们如下所示：

```
    A = 0011 1100
    B = 0000 1101
    -----------------
    A&B = 0000 1100
    A|B = 0011 1101
    A^B = 0011 0001
    ~A  = 1100 0011
```

下表列出了 C# 支持的位运算符。假设变量 A 的值为 60，变量 B 的值为 13，则：

运算符	描述	实例
	如果同时存在于两个操作数中，二进制 AND 运算符复	

	制一位到结果中。	
I	如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。	(A I B) 将得到 61，即为 0011 1101
^	如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。	(A ^ B) 将得到 49，即为 0011 0001
~	按位取反运算符是一元运算符，具有"翻转"位效果，即0变成1，1变成0，包括符号位。	(~A) 将得到 -61，即为 1100 0011，一个有符号二进制数的补码形式。
<<	二进制左移运算符。左操作数的值向左移动右操作数指定的位数。	A << 2 将得到 240，即为 1111 0000
>>	二进制右移运算符。左操作数的值向右移动右操作数指定的位数。	A >> 2 将得到 15，即为 0000 1111

请看下面的实例，了解 C# 中所有可用的位运算符：

```
using System;
namespace OperatorsAppl
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 60;              /* 60 = 0011 1100 */
            int b = 13;              /* 13 = 0000 1101 */
            int c = 0;

            c = a & b;           /* 12 = 0000 1100 */
            Console.WriteLine("Line 1 - c 的值是 {0}", c );

            c = a | b;           /* 61 = 0011 1101 */
            Console.WriteLine("Line 2 - c 的值是 {0}", c);

            c = a ^ b;           /* 49 = 0011 0001 */
            Console.WriteLine("Line 3 - c 的值是 {0}", c);

            c = ~a;               /*-61 = 1100 0011 */
            Console.WriteLine("Line 4 - c 的值是 {0}", c);

            c = a << 2;     /* 240 = 1111 0000 */
            Console.WriteLine("Line 5 - c 的值是 {0}", c);

            c = a >> 2;     /* 15 = 0000 1111 */
            Console.WriteLine("Line 6 - c 的值是 {0}", c);
```

```
                Console.WriteLine("Line 6 - c 的值是 {0}", c);
                Console.ReadLine();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Line 1 - c 的值是 12
Line 2 - c 的值是 61
Line 3 - c 的值是 49
Line 4 - c 的值是 -61
Line 5 - c 的值是 240
Line 6 - c 的值是 15
```

五、赋值运算符

下表列出了 C# 支持的赋值运算符：

运算符	描述	实例
`=`	简单的赋值运算符，把右边操作数的值赋给左边操作数	C = A + B 将把 A + B 的值赋给 C
`+=`	加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数	C += A 相当于 C = C + A
`-=`	减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数	C -= A 相当于 C = C - A
`*=`	乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数	C = *A* 相当于 *C* = *C* A
`/=`	除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数	C /= A 相当于 C = C / A
`%=`	求模且赋值运算符，求两个操作数的模赋值给左边操作数	C %= A 相当于 C = C % A
`<<=`	左移且赋值运算符	`C <<= 2` 等同于 `C = C << 2`
`>>=`	右移且赋值运算符	`C >>= 2` 等同于 `C = C >> 2`
`&=`	按位与且赋值运算符	`C &= 2` 等同于 `C = C & 2`
`^=`	按位异或且赋值运算符	`C ^= 2` 等同于 `C = C ^ 2`
I=	按位或且赋值运算符	C I= 2 等同于 C = C I 2

请看下面的实例，了解 C# 中所有可用的赋值运算符：

```
using System;
```

```
3.      namespace OperatorsAppl
4.      {
5.          class Program
6.          {
7.              static void Main(string[] args)
8.              {
9.                  int a = 21;
10.                 int c;
11.
12.                 c = a;
13.                 Console.WriteLine("Line 1 - =  c 的值 = {0}", c);
14.
15.                 c += a;
16.                 Console.WriteLine("Line 2 - += c 的值 = {0}", c);
17.
18.                 c -= a;
19.                 Console.WriteLine("Line 3 - -=  c 的值 = {0}", c);
20.
21.                 c *= a;
22.                 Console.WriteLine("Line 4 - *=  c 的值 = {0}", c);
23.
24.                 c /= a;
25.                 Console.WriteLine("Line 5 - /=  c 的值 = {0}", c);
26.
27.                 c = 200;
28.                 c %= a;
29.                 Console.WriteLine("Line 6 - %=  c 的值 = {0}", c);
30.
31.                 c <<= 2;
32.                 Console.WriteLine("Line 7 - <<=  c 的值 = {0}", c);
33.
34.                 c >>= 2;
35.                 Console.WriteLine("Line 8 - >>=  c 的值 = {0}", c);
36.
37.                 c &= 2;
38.                 Console.WriteLine("Line 9 - &=  c 的值 = {0}", c);
39.
40.                 c ^= 2;
41.                 Console.WriteLine("Line 10 - ^=  c 的值 = {0}", c);
42.
43.                 c |= 2;
44.                 Console.WriteLine("Line 11 - |=  c 的值 = {0}", c);
```

```
                Console.WriteLine("Line 11 - |=  c 的值 = {0}", c);
                Console.ReadLine();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    Line 1 - =     c 的值 = 21
    Line 2 - +=    c 的值 = 42
    Line 3 - -=    c 的值 = 21
    Line 4 - *=    c 的值 = 441
    Line 5 - /=    c 的值 = 21
    Line 6 - %=    c 的值 = 11
    Line 7 - <<=    c 的值 = 44
    Line 8 - >>=    c 的值 = 11
    Line 9 - &=    c 的值 = 2
    Line 10 - ^=    c 的值 = 0
    Line 11 - |=    c 的值 = 2
```

六、其他运算符

下表列出了 C# 支持的其他一些重要的运算符，包括 `sizeof` 、 `typeof` 和 `? :` 。

运算符	描述	实例
sizeof()	返回数据类型的大小。	`sizeof(int)` ，将返回 4.
typeof()	返回 class 的类型。	`typeof(StreamReader);`
&	返回变量的地址。	&a; 将得到变量的实际地址。
	变量的指针。	a; 将指向一个变量。
? :	条件表达式	如果条件为真 ? 则为 X : 否则为 Y
is	判断对象是否为某一类型。	`If( Ford is Car) // 检查 Ford 是否是 Car 类的一个对象。`
as	强制转换，即使转换失败也不会抛出异常。	`Object obj = new StringReader("Hello");` `StringReader r = obj as StringReader;`

实例

```
    using System;

    namespace OperatorsAppl
    {
```

```
        class Program
        {
           static void Main(string[] args)
           {

              /* sizeof 运算符的实例 */
              Console.WriteLine("int 的大小是 {0}", sizeof(int));
              Console.WriteLine("short 的大小是 {0}", sizeof(short));
              Console.WriteLine("double 的大小是 {0}", sizeof(double));

              /* 三元运算符符的实例 */
              int a, b;
              a = 10;
              b = (a == 1) ? 20 : 30;
              Console.WriteLine("b 的值是 {0}", b);

              b = (a == 10) ? 20 : 30;
              Console.WriteLine("b 的值是 {0}", b);
              Console.ReadLine();
           }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    int 的大小是 4
    short 的大小是 2
    double 的大小是 8
    b 的值是 30
    b 的值是 20
```

七、C# 中的运算符优先级

运算符的优先级确定表达式中项的组合。这会影响到一个表达式如何计算。某些运算符比其他运算符有更高的优先级，例如，乘除运算符具有比加减运算符更高的优先级。

例如 `x = 7 + 3 2` ，在这里，*x* 被赋值为 *13*，而不是 *20*，因为运算符 具有比 + 更高的优先级，所以首先计算乘法 3*2，然后再加上 7。

下表将按运算符优先级从高到低列出各个运算符，具有较高优先级的运算符出现在表格的上面，具有较低优先级的运算符出现在表格的下面。在表达式中，较高优先级的运算符会优先被计算。

类别	运算符	结合性
后缀	`() [] -> . ++ - -`	从左到右
一元	`+ - ! ~ ++ - - (type) & sizeof`	从右到左
乘除	`/ %`	从左到右
加减	`+ -`	从左到右
移位	`<< >>`	从左到右
关系	`< <= > >=`	从左到右
相等	`== !=`	从左到右
位与 AND	&	从左到右
位异或 XOR	^	从左到右
位或 OR	I	从左到右
逻辑与 AND	&&	从左到右
逻辑或 OR	II	从左到右
条件	`?:`	从右到左
赋值	`= += -= *= /= %=>>= <<= &= ^=` I=	从右到左
逗号	,	从左到右

实例

```
using System;

namespace OperatorsAppl
{

   class Program
   {
      static void Main(string[] args)
      {
         int a = 20;
         int b = 10;
         int c = 15;
         int d = 5;
         int e;
         e = (a + b) * c / d;      // ( 30 * 15 ) / 5
         Console.WriteLine("(a + b) * c / d 的值是 {0}", e);

         e = ((a + b) * c) / d;   // (30 * 15 ) / 5
         Console.WriteLine("((a + b) * c) / d 的值是 {0}", e);
```

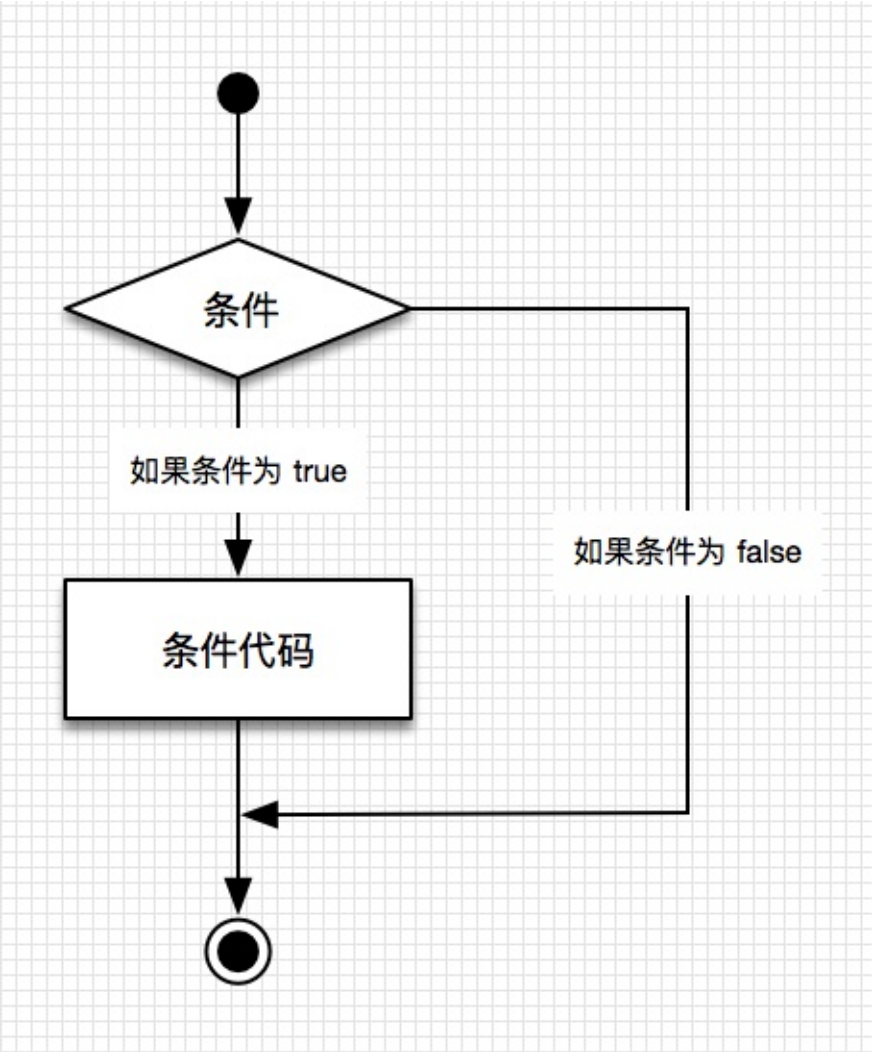
```
20.
21.             e = (a + b) * (c / d);   // (30) * (15/5)
22.             Console.WriteLine("(a + b) * (c / d) 的值是 {0}", e);
23.
24.             e = a + (b * c) / d;    //  20 + (150/5)
25.             Console.WriteLine("a + (b * c) / d 的值是 {0}", e);
26.             Console.ReadLine();
27.         }
28.     }
29. }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.     (a + b) * c / d 的值是 90
2.     ((a + b) * c) / d 的值是 90
3.     (a + b) * (c / d) 的值是 90
4.     a + (b * c) / d 的值是 50
```

C# 分支语句

分支结构要求程序员指定一个或多个要评估或测试的条件，以及条件为真时要执行的语句（必需的）和条件为假时要执行的语句（可选的）。下面是大多数编程语言中典型的分支结构的一般形式：



一、if 语句

一个 if 语句 由一个布尔表达式后跟一个或多个语句组成。

C# 中 if 语句的语法：

```
1.      if(boolean_expression)
2.      {
3.         /* 如果布尔表达式为真将执行的语句 */
4.      }
```

如果布尔表达式为 `true`，则 `if` 语句内的代码块将被执行。如果布尔表达式为 `false`，则 `if` 语句结束后的第一组代码（闭括号后）将被执行。

实例

```
using System;

namespace DecisionMaking
{

    class Program
    {
        static void Main(string[] args)
        {
            /* 局部变量定义 */
            int a = 10;

            /* 使用 if 语句检查布尔条件 */
            if (a < 20)
            {
                /* 如果条件为真，则输出下面的语句 */
                Console.WriteLine("a 小于 20");
            }
            Console.WriteLine("a 的值是 {0}", a);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 小于 20
a 的值是 10
```

二、if…else 语句

一个 `if` 语句 后可跟一个可选的 `else` 语句，`else` 语句在布尔表达式为假时执行。

C# 中 if…else 语句的语法：

```
if(boolean_expression)
{
   /* 如果布尔表达式为真将执行的语句 */
}
else
{
  /* 如果布尔表达式为假将执行的语句 */
}
```

如果布尔表达式为 `true`，则执行 `if` 块内的代码。如果布尔表达式为 `false`，则执行 `else` 块内的代码。

实例

```
using System;

namespace DecisionMaking
{

    class Program
    {
        static void Main(string[] args)
        {

            /* 局部变量定义 */
            int a = 100;

            /* 检查布尔条件 */
            if (a < 20)
            {
                /* 如果条件为真，则输出下面的语句 */
                Console.WriteLine("a 小于 20");
            }
            else
            {
                /* 如果条件为假，则输出下面的语句 */
                Console.WriteLine("a 大于 20");
            }
            Console.WriteLine("a 的值是 {0}", a);
            Console.ReadLine();
```

```
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    a 大于 20
    a 的值是 100
```

三、if…else if…else 语句

一个 `if` 语句后可跟一个可选的 `else if…else` 语句，这可用于测试多种条件。

当使用 `if…else if…else` 语句时，以下几点需要注意：

- 一个 if 后可跟零个或一个 else，它必须在任何一个 else if 之后。
- 一个 if 后可跟零个或多个 else if，它们必须在 else 之前。
- 一旦某个 else if 匹配成功，其他的 else if 或 else 将不会被测试。

C# 中的 if…else if…else 语句的语法：

```
    if(boolean_expression 1)
    {
       /* 当布尔表达式 1 为真时执行 */
    }
    else if( boolean_expression 2)
    {
       /* 当布尔表达式 2 为真时执行 */
    }
    else if( boolean_expression 3)
    {
       /* 当布尔表达式 3 为真时执行 */
    }
    else
    {
       /* 当上面条件都不为真时执行 */
    }
```

实例

```
using System;

namespace DecisionMaking
{

    class Program
    {
        static void Main(string[] args)
        {

            /* 局部变量定义 */
            int a = 100;

            /* 检查布尔条件 */
            if (a == 10)
            {
                /* 如果 if 条件为真，则输出下面的语句 */
                Console.WriteLine("a 的值是 10");
            }
            else if (a == 20)
            {
                /* 如果 else if 条件为真，则输出下面的语句 */
                Console.WriteLine("a 的值是 20");
            }
            else if (a == 30)
            {
                /* 如果 else if 条件为真，则输出下面的语句 */
                Console.WriteLine("a 的值是 30");
            }
            else
            {
                /* 如果上面条件都不为真，则输出下面的语句 */
                Console.WriteLine("没有匹配的值");
            }
            Console.WriteLine("a 的准确值是 {0}", a);
            Console.ReadLine();
        }
    }
```

```
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    没有匹配的值
    a 的准确值是 100
```

四、嵌套if语句

在 C# 中，嵌套 `if-else` 语句是合法的，这意味着您可以在一个 `if` 或 `else if` 语句内使用另一个 `if` 或 `else if` 语句。

C# 中 嵌套 if 语句的语法：

```
    if( boolean_expression 1)
    {
       /* 当布尔表达式 1 为真时执行 */
       if(boolean_expression 2)
       {
          /* 当布尔表达式 2 为真时执行 */
       }
    }
```

您可以嵌套 `else if…else` ，方式与嵌套 `if` 语句相似。

实例

```
    using System;

    namespace DecisionMaking
    {

        class Program
        {
            static void Main(string[] args)
            {

```

```
                //* 局部变量定义 */
                int a = 100;
                int b = 200;

                /* 检查布尔条件 */
                if (a == 100)
                {
                    /* 如果条件为真，则检查下面的条件 */
                    if (b == 200)
                    {
                        /* 如果条件为真，则输出下面的语句 */
                        Console.WriteLine("a 的值是 100，且 b 的值是 200");
                    }
                }
                Console.WriteLine("a 的准确值是 {0}", a);
                Console.WriteLine("b 的准确值是 {0}", b);
                Console.ReadLine();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    a 的值是 100，且 b 的值是 200
    a 的准确值是 100
    b 的准确值是 200
```

五、switch 语句

一个 `switch` 语句允许测试一个变量等于多个值时的情况。每个值称为一个 `case`，且被测试的变量会对每个 `switch case` 进行检查。

C# 中 switch 语句的语法：

```
switch(expression){
    case constant-expression  :
       statement(s);
       break;
    case constant-expression  :
       statement(s);
```

```
        break;

    /* 您可以有任意数量的 case 语句 */
    default : /* 可选的 */
        statement(s);
        break;
}
```

switch 语句必须遵循下面的规则：

- `switch` 语句中的 `expression` 必须是一个整型或枚举类型，或者是一个 `class` 类型，其中 `class` 有一个单一的转换函数将其转换为整型或枚举类型。
- 在一个 `switch` 中可以有任意数量的 `case` 语句。每个 `case` 后跟一个要比较的值和一个冒号。
- `case` 的 `constant-expression` 必须与 `switch` 中的变量具有相同的数据类型，且必须是一个常量。
- 当被测试的变量等于 `case` 中的常量时，`case` 后跟的语句将被执行，直到遇到 `break` 语句为止。
- 当遇到 `break` 语句时，`switch` 终止，控制流将跳转到 `switch` 语句后的下一行。
- 不是每一个 `case` 都需要包含 `break`。如果 `case` 语句为空，则可以不包含 `break`，控制流将会 继续 后续的 `case`，直到遇到 `break` 为止。
- C# 不允许从一个开关部分继续执行到下一个开关部分。如果 `case` 语句中有处理语句，则必须包含 `break` 或其他跳转语句。
- 一个 `switch` 语句可以有一个可选的 `default case`，出现在 `switch` 的结尾。`default case` 可用于在上面所有 `case` 都不为真时执行一个任务。`default case` 中的 `break` 语句不是必需的。
- C# 不支持从一个 `case` 标签显式贯穿到另一个 `case` 标签。如果要使 C# 支持从一个 `case` 标签显式贯穿到另一个 `case` 标签，可以使用 `goto` 一个 `switch-case` 或 `goto default`。

实例

```
using System;

```

```
namespace DecisionMaking
{

    class Program
    {
        static void Main(string[] args)
        {
            /* 局部变量定义 */
            char grade = 'B';

            switch (grade)
            {
                case 'A':
                    Console.WriteLine("很棒！");
                    break;
                case 'B':
                case 'C':
                    Console.WriteLine("做得好");
                    break;
                case 'D':
                    Console.WriteLine("您通过了");
                    break;
                case 'F':
                    Console.WriteLine("最好再试一下");
                    break;
                default:
                    Console.WriteLine("无效的成绩");
                    break;
            }
            Console.WriteLine("您的成绩是 {0}", grade);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
做得好
您的成绩是 B
```

六、嵌套 switch 语句

您可以把一个 `switch` 作为一个外部 `switch` 的语句序列的一部分，即可以在一个 `switch` 语句内使用另一个 `switch` 语句。即使内部和外部 `switch` 的 `case` 常量包含共同的值，也没有矛盾。

C# 中 嵌套 switch 语句的语法：

```
switch(ch1)
{
   case 'A':
      printf("这个 A 是外部 switch 的一部分" );
      switch(ch2)
      {
         case 'A':
            printf("这个 A 是内部 switch 的一部分" );
            break;
         case 'B': /* 内部 B case 代码 */
      }
      break;
   case 'B': /* 外部 B case 代码 */
}
```

实例

```
using System;

namespace DecisionMaking
{

    class Program
    {
        static void Main(string[] args)
        {
            int a = 100;
            int b = 200;

            switch (a)
            {
```

```
                    case 100:
                        Console.WriteLine("这是外部 switch 的一部分");
                        switch (b)
                        {
                            case 200:
                            Console.WriteLine("这是内部 switch 的一部分");
                            break;
                        }
                        break;
                }
                Console.WriteLine("a 的准确值是 {0}", a);
                Console.WriteLine("b 的准确值是 {0}", b);
                Console.ReadLine();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    这是外部 switch 的一部分
    这是内部 switch 的一部分
    a 的准确值是 100
    b 的准确值是 200
```

七、 ?: 运算符

我们已经在前面的章节中讲解了 条件运算符 `? :` ，可以用来替代 `if…else` 语句。它的一般形式如下：

```
    Exp1 ? Exp2 : Exp3;
```

其中， `Exp1` 、 `Exp2` 和 `Exp3` 是表达式。请注意，冒号的使用和位置。

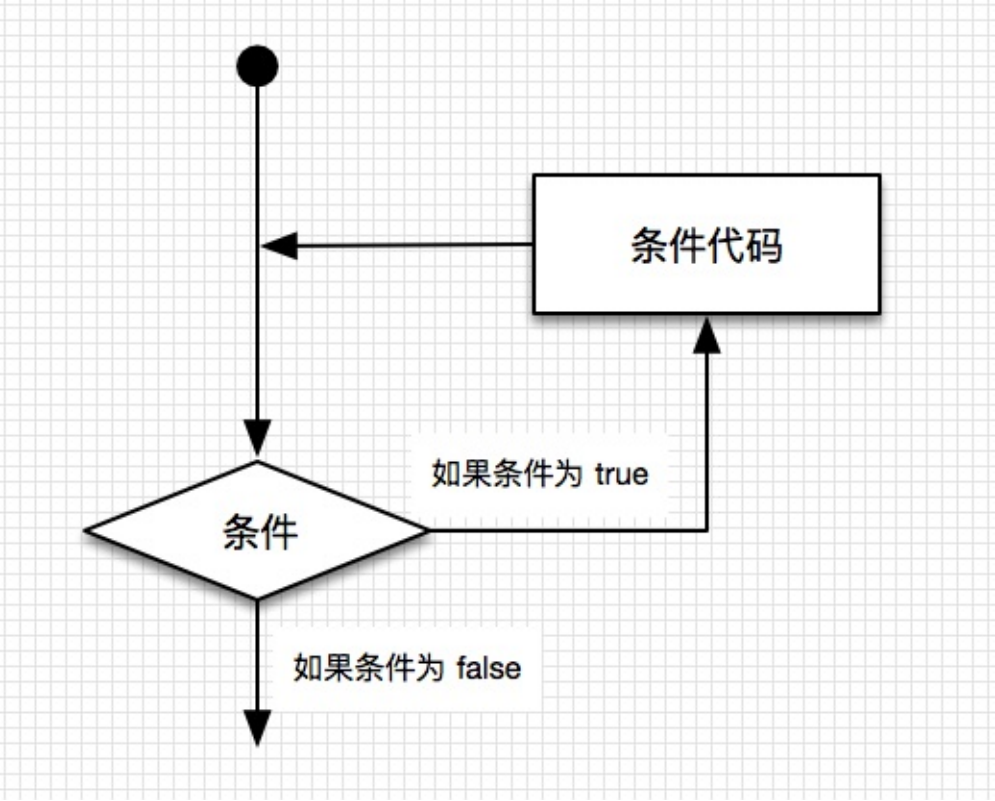
`?` 表达式的值是由 `Exp1` 决定的。如果 `Exp1` 为真，则计算 `Exp2` 的值，结果即为整个 `?` 表达式的值。如果 `Exp1` 为假，则计算 `Exp3` 的值，结果即为整个 `?` 表达式的值。

C# 循环语句

有的时候，可能需要多次执行同一块代码。一般情况下，语句是顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。

编程语言提供了允许更为复杂的执行路径的多种控制结构。

循环语句允许我们多次执行一个语句或语句组，下面是大多数编程语言中循环语句的一般形式：



一、while 循环

只要给定的条件为真，C# 中的 `while` 循环语句会重复执行一个目标语句。

C# 中 while 循环的语法：

```
1.        while(condition)
2.        {
3.           statement(s);
4.        }
```

在这里， `statement(s)` 可以是一个单独的语句，也可以是几个语句组成的代码块。 `condition` 可以是任意的表达式，当为任意非零值时都为真。当条件为真时执行循环。

当条件为假时，程序流将继续执行紧接着循环的下一条语句。

在这里， `while` 循环的关键点是循环可能一次都不会执行。当条件被测试且结果为假时，会跳过循环主体，直接执行紧接着 `while` 循环的下一条语句。

实例

```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            /* 局部变量定义 */
            int a = 10;

            /* while 循环执行 */
            while (a < 20)
            {
                Console.WriteLine("a 的值： {0}", a);
                a++;
            }
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值： 10
a 的值： 11
a 的值： 12
a 的值： 13
a 的值： 14
a 的值： 15
a 的值： 16
a 的值： 17
a 的值： 18
```

```
10.      a 的值： 19
```

二、for/foreach 循环

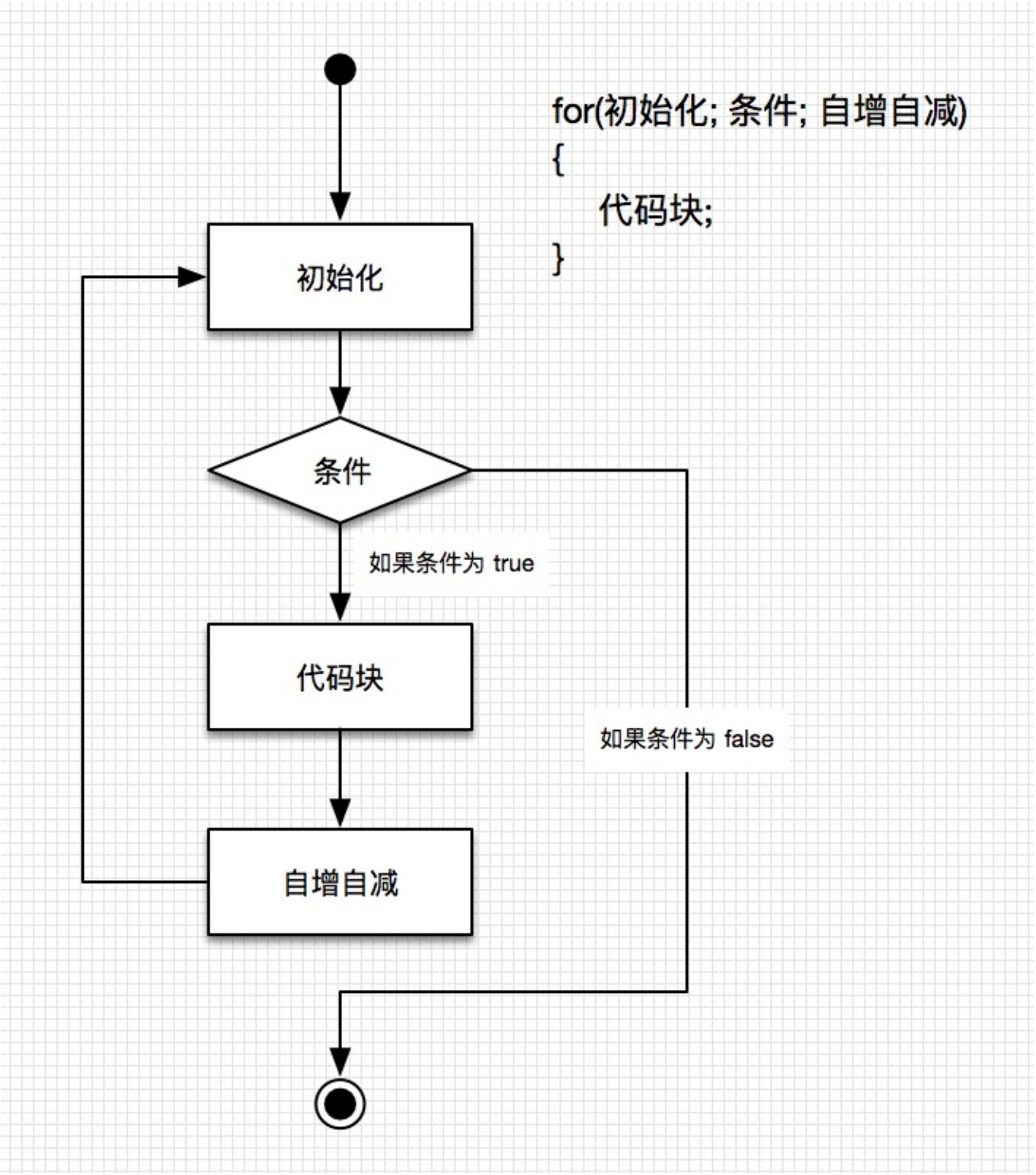
一个 `for` 循环是一个允许您编写一个执行特定次数的循环的重复控制结构。

C# 中 for 循环的语法：

```
1.      for ( init; condition; increment )
2.      {
3.         statement(s);
4.      }
```

下面是 **for** 循环的控制流：

- init 会首先被执行，且只会执行一次。这一步允许您声明并初始化任何循环控制变量。您也可以不在这里写任何语句，只要有一个分号出现即可。
- 接下来，会判断 condition。如果为真，则执行循环主体。如果为假，则不执行循环主体，且控制流会跳转到紧接着 for 循环的下一条语句。
- 在执行完 for 循环主体后，控制流会跳回上面的 increment 语句。该语句允许您更新循环控制变量。该语句可以留空，只要在条件后有一个分号出现即可。
- 条件再次被判断。如果为真，则执行循环，这个过程会不断重复（循环主体，然后增加步值，再然后重新判断条件）。在条件变为假时，for 循环终止。



实例

```
1.      using System;
2.
3.      namespace Loops
4.      {
5.
```

```
        class Program
        {
            static void Main(string[] args)
            {
                /* for 循环执行 */
                for (int a = 10; a < 20; a = a + 1)
                {
                    Console.WriteLine("a 的值： {0}", a);
                }
                Console.ReadLine();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    a 的值： 10
    a 的值： 11
    a 的值： 12
    a 的值： 13
    a 的值： 14
    a 的值： 15
    a 的值： 16
    a 的值： 17
    a 的值： 18
    a 的值： 19
```

foreach

C# 也支持 `foreach` 循环，使用 `foreach` 可以迭代数组或者一个集合对象。

以下实例有三个部分：

- 通过 `foreach` 循环输出整型数组中的元素。
- 通过 `for` 循环输出整型数组中的元素。
- `foreach` 循环设置数组元素的计算器。

```
    class ForEachTest
    {
```

```
        static void Main(string[] args)
        {
            int[] fibarray = new int[] { 0, 1, 1, 2, 3, 5, 8, 13 };
            foreach (int element in fibarray)
            {
                System.Console.WriteLine(element);
            }
            System.Console.WriteLine();


            // 类似 foreach 循环
            for (int i = 0; i < fibarray.Length; i++)
            {
                System.Console.WriteLine(fibarray[i]);
            }
            System.Console.WriteLine();


            // 设置集合中元素的计算器
            int count = 0;
            foreach (int element in fibarray)
            {
                count += 1;
                System.Console.WriteLine("Element #{0}: {1}", count, element);
            }
            System.Console.WriteLine("Number of elements in the array: {0}",
count);
        }
    }
```

输出结果为：

```
    0
    1
    1
    2
    3
    5
    8
    13

    0
```

```
    1
    1
    2
    3
    5
    8
    13

    Element #1: 0
    Element #2: 1
    Element #3: 1
    Element #4: 2
    Element #5: 3
    Element #6: 5
    Element #7: 8
    Element #8: 13
    Number of elements in the array: 8
```

三、do…while 循环

不像 `for` 和 `while` 循环，它们是在循环头部测试循环条件。`do…while` 循环是在循环的尾部检查它的条件。

`do…whil` e 循环与 `while` 循环类似，但是 `do…while` 循环会确保至少执行一次循环。

C# 中 do…while 循环的语法：

```
    do
    {
       statement(s);

    }while( condition );
```

请注意，条件表达式出现在循环的尾部，所以循环中的 `statement(s)` 会在条件被测试之前至少执行一次。

如果条件为真，控制流会跳转回上面的 `do` ，然后重新执行循环中的 `statement(s)` 。这个过程会不断重复，直到给定条件变为假为止。

实例

```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            /* 局部变量定义 */
            int a = 10;

            /* do 循环执行 */
            do
            {
               Console.WriteLine("a 的值： {0}", a);
                a = a + 1;
            } while (a < 20);

            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值： 10
a 的值： 11
a 的值： 12
a 的值： 13
a 的值： 14
a 的值： 15
a 的值： 16
a 的值： 17
a 的值： 18
a 的值： 19
```

四、嵌套循环

C# 允许在一个循环内使用另一个循环，下面演示几个实例来说明这个概念。

C# 中 嵌套 for 循环 语句的语法:

```
for ( init; condition; increment )
{
   for ( init; condition; increment )
   {
      statement(s);
   }
   statement(s);
}
```

C# 中 嵌套 while 循环 语句的语法:

```
while(condition)
{
   while(condition)
   {
      statement(s);
   }
   statement(s);
}
```

C# 中 嵌套 do…while 循环 语句的语法:

```
do
{
   statement(s);
   do
   {
      statement(s);
   }while( condition );

}while( condition );
```

关于嵌套循环有一点值得注意，您可以在任何类型的循环内嵌套其他任何类型的循环。比如，一个 `for` 循环可以嵌套在一个 `while` 循环内，反之亦然。

下面的程序使用了一个嵌套的 **for** 循环来查找 **2** 到 **100** 中的质数：

```
1.      using System;
2.
3.      namespace Loops
4.      {
5.
6.         class Program
7.         {
8.            static void Main(string[] args)
9.            {
10.              /* 局部变量定义 */
11.              int i, j;
12.
13.              for (i = 2; i < 100; i++)
14.              {
15.                 for (j = 2; j <= (i / j); j++)
16.                    if ((i % j) == 0) break; // 如果找到，则不是质数
17.                 if (j > (i / j))
18.                    Console.WriteLine("{0} 是质数", i);
19.              }
20.
21.              Console.ReadLine();
22.           }
23.        }
24.     }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.      2 是质数
2.      3 是质数
3.      5 是质数
4.      7 是质数
5.      11 是质数
6.      13 是质数
7.      17 是质数
8.      19 是质数
```

```
 9.      23 是质数
10.      29 是质数
11.      31 是质数
12.      37 是质数
13.      41 是质数
14.      43 是质数
15.      47 是质数
16.      53 是质数
17.      59 是质数
18.      61 是质数
19.      67 是质数
20.      71 是质数
21.      73 是质数
22.      79 是质数
23.      83 是质数
24.      89 是质数
25.      97 是质数
```

五、循环控制语句

循环控制语句更改执行的正常序列。当执行离开一个范围时，所有在该范围中创建的自动对象都会被销毁。

C# 提供了下列的控制语句：

- `break` 语句。终止 `loop` 或 `switch` 语句，程序流将继续执行紧接着 `loop` 或 `switch` 的下一条语句。
- `continue` 语句。引起循环跳过主体的剩余部分，立即重新开始测试条件。

5.1 break 语句

C# 中 break 语句有以下两种用法：

- 当 `break` 语句出现在一个循环内时，循环会立即终止，且程序流将继续执行紧接着循环的下一条语句。
- 它可用于终止 `switch` 语句中的一个 `case` 。

如果您使用的是嵌套循环（即一个循环内嵌套另一个循环），break 语句会停止执行最内层的循环，然后开始执行该块之后的下一行代码。

实例

```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            /* 局部变量定义 */
            int a = 10;

            /* while 循环执行 */
            while (a < 20)
            {
                Console.WriteLine("a 的值： {0}", a);
                a++;
                if (a > 15)
                {
                    /* 使用 break 语句终止 loop */
                    break;
                }
            }
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值： 10
a 的值： 11
a 的值： 12
a 的值： 13
a 的值： 14
```

```
6.      a 的值： 15
```

5.2 continue 语句

C# 中的 `continue` 语句有点像 `break` 语句。但它不是强迫终止，`continue` 会跳过当前循环中的代码，强迫开始下一次循环。对于 `for` 循环，`continue` 语句会导致执行条件测试和循环增量部分。对于 `while` 和 `do…while` 循环，`continue` 语句会导致程序控制回到条件测试上。

实例

```
1.      using System;
2.
3.      namespace Loops
4.      {
5.
6.          class Program
7.          {
8.              static void Main(string[] args)
9.              {
10.                 /* 局部变量定义 */
11.                 int a = 10;
12.
13.                 /* do 循环执行 */
14.                 do
15.                 {
16.                     if (a == 15)
17.                     {
18.                         /* 跳过迭代 */
19.                         a = a + 1;
20.                         continue;
21.                     }
22.                     Console.WriteLine("a 的值： {0}", a);
23.                     a++;
24.
25.                 } while (a < 20);
26.
27.                 Console.ReadLine();
28.             }
```

```
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a 的值： 10
a 的值： 11
a 的值： 12
a 的值： 13
a 的值： 14
a 的值： 16
a 的值： 17
a 的值： 18
a 的值： 19
```

六、无限循环

如果条件永远不为假，则循环将变成无限循环。 `for` 循环在传统意义上可用于实现无限循环。由于构成循环的三个表达式中任何一个都不是必需的，您可以将某些条件表达式留空来构成一个无限循环。

```
using System;

namespace Loops
{

    class Program
    {
        static void Main(string[] args)
        {
            for (; ; )
            {
                Console.WriteLine("Hey! I am Trapped");
            }

        }
    }
}
```

当条件表达式不存在时，它被假设为真。您也可以设置一个初始值和增量表达式，但是一般情况下，程序员偏向于使用 `for(;;)` 结构来表示一个无限循环。

C# 访问修饰符

封装 被定义为"把一个或多个项目封闭在一个物理的或者逻辑的包中"。在面向对象程序设计方法论中，封装是为了防止对实现细节的访问。

抽象和封装是面向对象程序设计的相关特性。抽象允许相关信息可视化，封装则使开发者实现所需级别的抽象。

C# 封装根据具体的需要，设置使用者的访问权限，并通过 访问修饰符 来实现。

一个 访问修饰符 定义了一个类成员的范围和可见性。C# 支持的访问修饰符如下所示：

- Public: 所有对象都可以访问;
- Private: 对象本身在对象内部可以访问;
- Protected: 只有该类对象及其子类对象可以访问
- Internal: 同一个程序集的对象可以访问;
- Protected internal: 该程序集内的派生类访问，是protected和internal的交集;

一、Public 访问修饰符

`Public` 访问修饰符允许一个类将其成员变量和成员函数暴露给其他的函数和对象。任何公有成员可以被外部的类访问。

下面的实例说明了这点：

```
1.      using System;
2.
3.      namespace RectangleApplication
4.      {
5.          class Rectangle
6.          {
7.              //成员变量
8.              public double length;
9.              public double width;
10.
11.             public double GetArea()
```

```
            {
                return length * width;
            }
            public void Display()
            {
                Console.WriteLine("长度： {0}", length);
                Console.WriteLine("宽度： {0}", width);
                Console.WriteLine("面积： {0}", GetArea());
            }
        }// Rectangle 结束

        class ExecuteRectangle
        {
            static void Main(string[] args)
            {
                Rectangle r = new Rectangle();
                r.length = 4.5;
            r.width = 3.5;
                r.Display();
                Console.ReadLine();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    长度： 4.5
    宽度： 3.5
    面积： 15.75
```

在上面的实例中，成员变量 `length` 和 `width` 被声明为 `public`，所以它们可以被函数 `Main()` 使用 `Rectangle` 类的实例 `r` 访问。

成员函数 `Display()` 和 `GetArea()` 可以直接访问这些变量。

成员函数 `Display()` 也被声明为 `public`，所以它也能被 `Main()` 使用 `Rectangle` 类的实例 `r` 访问。

二、Private 访问修饰符

`Private` 访问修饰符允许一个类将其成员变量和成员函数对其他的函数和对象进行隐藏。只有同一

个类中的函数可以访问它的私有成员。即使是类的实例也不能访问它的私有成员。

下面的实例说明了这点：

```
using System;

namespace RectangleApplication
{
    class Rectangle
    {
        //成员变量
        private double length;
        private double width;

        public void Acceptdetails()
        {
            Console.WriteLine("请输入长度：");
            length = Convert.ToDouble(Console.ReadLine());
            Console.WriteLine("请输入宽度：");
            width = Convert.ToDouble(Console.ReadLine());
        }
        public double GetArea()
        {
            return length * width;
        }
        public void Display()
        {
            Console.WriteLine("长度： {0}", length);
            Console.WriteLine("宽度： {0}", width);
            Console.WriteLine("面积： {0}", GetArea());
        }
    }//end class Rectangle
    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle();
            r.Acceptdetails();
            r.Display();
            Console.ReadLine();
        }
```

```
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
请输入长度：
4.4
请输入宽度：
3.3
长度： 4.4
宽度： 3.3
面积： 14.52
```

在上面的实例中，成员变量 `length` 和 `width` 被声明为 `private`，所以它们不能被函数 `Main()` 访问。

成员函数 `AcceptDetails()` 和 `Display()` 可以访问这些变量。

由于成员函数 `AcceptDetails()` 和 `Display()` 被声明为 `public`，所以它们可以被 `Main()` 使用 `Rectangle` 类的实例 `r` 访问。

三、Protected 访问修饰符

`Protected` 访问修饰符允许子类访问它的基类的成员变量和成员函数。这样有助于实现继承。我们将在继承的章节详细讨论这个。更详细地讨论这个。

四、Internal 访问修饰符

`Internal` 访问说明符允许一个类将其成员变量和成员函数暴露给当前程序中的其他函数和对象。换句话说，带有 `internal` 访问修饰符的任何成员可以被定义在该成员所定义的应用程序内的任何类或方法访问。

下面的实例说明了这点：

```
using System;

namespace RectangleApplication
```

```
    {
        class Rectangle
        {
            //成员变量
            internal double length;
            internal double width;

            double GetArea()
            {
                return length * width;
            }
           public void Display()
            {
                Console.WriteLine("长度： {0}", length);
                Console.WriteLine("宽度： {0}", width);
                Console.WriteLine("面积： {0}", GetArea());
            }
        }//end class Rectangle
        class ExecuteRectangle
        {
            static void Main(string[] args)
            {
                Rectangle r = new Rectangle();
                r.length = 4.5;
                r.width = 3.5;
                r.Display();
                Console.ReadLine();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    长度： 4.5
    宽度： 3.5
    面积： 15.75
```

在上面的实例中，请注意成员函数 `GetArea()` 声明的时候不带有任何访问修饰符。如果没有指定访问修饰符，则使用类成员的默认访问修饰符，即为 `private` 。

五、Protected Internal 访问修饰符

`Protected Internal` 访问修饰符允许在本类,派生类或者包含该类的程序集中访问。这也被用于实现继承。

C# 方法

一个方法是把一些相关的语句组织在一起，用来执行一个任务的语句块。每一个 C# 程序至少有一个带有 `Main` 方法的类。

要使用一个方法，您需要：

```
1. 1. 定义方法
2. 2. 调用方法
```

一、C# 中定义方法

当定义一个方法时，从根本上说是在声明它的结构的元素。在 C# 中，定义方法的语法如下：

```
1.        <Access Specifier> <Return Type> <Method Name>(Parameter List)
2.        {
3.           Method Body
4.        }
```

下面是方法的各个元素：

- Access Specifier: 访问修饰符，这个决定了变量或方法对于另一个类的可见性。
- Return type: 返回类型，一个方法可以返回一个值。返回类型是方法返回的值的数据类型。如果方法不返回任何值，则返回类型为 void。
- Method name: 方法名称，是一个唯一的标识符，且是大小写敏感的。它不能与类中声明的其他标识符相同。
- Parameter list: 参数列表，使用圆括号括起来，该参数是用来传递和接收方法的数据。参数列表是指方法的参数类型、顺序和数量。参数是可选的，也就是说，一个方法可能不包含参数。
- Method body: 方法主体，包含了完成任务所需的指令集。

实例

下面的代码片段显示一个函数 `FindMax`，它接受两个整数值，并返回两个中的较大值。它有

`public` 访问修饰符，所以它可以使用类的实例从类的外部进行访问。

```
class NumberManipulator
{
   public int FindMax(int num1, int num2)
   {
      /* 局部变量声明 */
      int result;

      if (num1 > num2)
         result = num1;
      else
         result = num2;

      return result;
   }
   ...
}
```

二、C# 中调用方法

您可以使用方法名调用方法。下面的实例演示了这点：

```
using System;

namespace CalculatorApplication
{
   class NumberManipulator
   {
      public int FindMax(int num1, int num2)
      {
         /* 局部变量声明 */
         int result;

         if (num1 > num2)
            result = num1;
         else
            result = num2;

         return result;
```

```
            }
            static void Main(string[] args)
            {
               /* 局部变量定义 */
               int a = 100;
               int b = 200;
               int ret;
               NumberManipulator n = new NumberManipulator();

               //调用 FindMax 方法
               ret = n.FindMax(a, b);
               Console.WriteLine("最大值是： {0}", ret );
               Console.ReadLine();
            }
         }
      }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    最大值是： 200
```

您也可以使用类的实例从另一个类中调用其他类的公有方法。例如，方法 `FindMax` 属于 `NumberManipulator` 类，您可以从另一个类 `Test` 中调用它。

```
      using System;

      namespace CalculatorApplication
      {
          class NumberManipulator
          {
              public int FindMax(int num1, int num2)
              {
                  /* 局部变量声明 */
                  int result;

                  if (num1 > num2)
                      result = num1;
                  else
                      result = num2;

```

```
                return result;
            }
        }
        class Test
        {
            static void Main(string[] args)
            {
                /* 局部变量定义 */
                int a = 100;
                int b = 200;
                int ret;
                NumberManipulator n = new NumberManipulator();
                //调用 FindMax 方法
                ret = n.FindMax(a, b);
                Console.WriteLine("最大值是： {0}", ret );
                Console.ReadLine();

            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    最大值是： 200
```

三、递归方法调用

一个方法可以自我调用。这就是所谓的 递归。下面的实例使用递归函数计算一个数的阶乘：

```
    using System;

    namespace CalculatorApplication
    {
        class NumberManipulator
        {
            public int factorial(int num)
            {
                /* 局部变量定义 */
                int result;

```

```
                if (num == 1)
                {
                    return 1;
                }
                else
                {
                    result = factorial(num - 1) * num;
                    return result;
                }
            }

            static void Main(string[] args)
            {
                NumberManipulator n = new NumberManipulator();
                //调用 factorial 方法
                Console.WriteLine("6 的阶乘是： {0}", n.factorial(6));
                Console.WriteLine("7 的阶乘是： {0}", n.factorial(7));
                Console.WriteLine("8 的阶乘是： {0}", n.factorial(8));
                Console.ReadLine();

            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    6 的阶乘是： 720
    7 的阶乘是： 5040
    8 的阶乘是： 40320
```

四、参数传递

当调用带有参数的方法时，您需要向方法传递参数。在 C# 中，有三种向方法传递参数的方式：

方式	描述
值参数	这种方式复制参数的实际值给函数的形式参数，实参和形参使用的是两个不同内存中的值。在这种情况下，当形参的值发生改变时，不会影响实参的值，从而保证了实参数据的安全。
引用参	这种方式复制参数的内存位置的引用给形式参数。这意味着，当形参的值发生改变时，同时也改变实参的值。

数	
输出参数	这种方式可以返回多个值。

4.1 按值传递参数

这是参数传递的默认方式。在这种方式下，当调用一个方法时，会为每个值参数创建一个新的存储位置。实际参数的值会复制给形参，实参和形参使用的是两个不同内存中的值。所以，当形参的值发生改变时，不会影响实参的值，从而保证了实参数据的安全。下面的实例演示了这个概念：

```
1.       using System;
2.       namespace CalculatorApplication
3.       {
4.          class NumberManipulator
5.          {
6.             public void swap(int x, int y)
7.             {
8.                int temp;
9.
10.               temp = x; /* 保存 x 的值 */
11.               x = y;    /* 把 y 赋值给 x */
12.               y = temp; /* 把 temp 赋值给 y */
13.            }
14.
15.            static void Main(string[] args)
16.            {
17.               NumberManipulator n = new NumberManipulator();
18.               /* 局部变量定义 */
19.               int a = 100;
20.               int b = 200;
21.
22.               Console.WriteLine("在交换之前, a 的值： {0}", a);
23.               Console.WriteLine("在交换之前, b 的值： {0}", b);
24.
25.               /* 调用函数来交换值 */
26.               n.swap(a, b);
27.
28.               Console.WriteLine("在交换之后, a 的值： {0}", a);
29.               Console.WriteLine("在交换之后, b 的值： {0}", b);
30.
```

```
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
在交换之前，a 的值：100
在交换之前，b 的值：200
在交换之后，a 的值：100
在交换之后，b 的值：200
```

结果表明，即使在函数内改变了值，值也没有发生任何的变化。

4.2 按引用传递参数

引用参数是一个对变量的内存位置的引用。当按引用传递参数时，与值参数不同的是，它不会为这些参数创建一个新的存储位置。引用参数表示与提供给方法的实际参数具有相同的内存位置。

在 C# 中，使用 `ref` 关键字声明引用参数。下面的实例演示了这点：

```
using System;
namespace CalculatorApplication
{
   class NumberManipulator
   {
      public void swap(ref int x, ref int y)
      {
         int temp;

         temp = x; /* 保存 x 的值 */
         x = y;    /* 把 y 赋值给 x */
         y = temp; /* 把 temp 赋值给 y */
       }

      static void Main(string[] args)
      {
         NumberManipulator n = new NumberManipulator();
         /* 局部变量定义 */
         int a = 100;
         int b = 200;
```

```
21.
22.                 Console.WriteLine("在交换之前，a 的值： {0}", a);
23.                 Console.WriteLine("在交换之前，b 的值： {0}", b);
24.
25.                 /* 调用函数来交换值 */
26.                 n.swap(ref a, ref b);
27.
28.                 Console.WriteLine("在交换之后，a 的值： {0}", a);
29.                 Console.WriteLine("在交换之后，b 的值： {0}", b);
30.
31.                 Console.ReadLine();
32.
33.             }
34.          }
35.       }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.        在交换之前，a 的值：100
2.        在交换之前，b 的值：200
3.        在交换之后，a 的值：200
4.        在交换之后，b 的值：100
```

结果表明， `swap` 函数内的值改变了，且这个改变可以在 `Main` 函数中反映出来。

4.3 按输出传递参数

`return` 语句可用于只从函数中返回一个值。但是，可以使用 输出参数 来从函数中返回两个值。输出参数会把方法输出的数据赋给自己，其他方面与引用参数相似。

下面的实例演示了这点：

```
1.        using System;
2.
3.        namespace CalculatorApplication
4.        {
5.           class NumberManipulator
6.           {
7.              public void getValue(out int x )
8.              {
9.                 int temp = 5;
```

```
            x = temp;
        }

        static void Main(string[] args)
        {
            NumberManipulator n = new NumberManipulator();
            /* 局部变量定义 */
            int a = 100;

            Console.WriteLine("在方法调用之前, a 的值： {0}", a);

            /* 调用函数来获取值 */
            n.getValue(out a);

            Console.WriteLine("在方法调用之后, a 的值： {0}", a);
            Console.ReadLine();

        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
在方法调用之前, a 的值： 100
在方法调用之后, a 的值： 5
```

提供给输出参数的变量不需要赋值。当需要从一个参数没有指定初始值的方法中返回值时，输出参数特别有用。请看下面的实例，来理解这一点：

```
using System;

namespace CalculatorApplication
{
    class NumberManipulator
    {
        public void getValues(out int x, out int y )
        {
            Console.WriteLine("请输入第一个值： ");
            x = Convert.ToInt32(Console.ReadLine());
            Console.WriteLine("请输入第二个值： ");
            y = Convert.ToInt32(Console.ReadLine());
        }
```

```
14.
15.             static void Main(string[] args)
16.             {
17.                NumberManipulator n = new NumberManipulator();
18.                /* 局部变量定义 */
19.                int a , b;
20.
21.                /* 调用函数来获取值 */
22.                n.getValues(out a, out b);
23.
24.                Console.WriteLine("在方法调用之后，a 的值： {0}", a);
25.                Console.WriteLine("在方法调用之后，b 的值： {0}", b);
26.                Console.ReadLine();
27.             }
28.          }
29.       }
```

当上面的代码被编译和执行时，它会产生下列结果（取决于用户输入）：

```
1.        请输入第一个值：
2.        7
3.        请输入第二个值：
4.        8
5.        在方法调用之后，a 的值： 7
6.        在方法调用之后，b 的值： 8
```

C# 常量

常量是固定值，程序执行期间不会改变。常量可以是任何基本数据类型，比如整数常量、浮点常量、字符常量或者字符串常量，还有枚举常量。常量可以被当作常规的变量，只是它们的值在定义后不能被修改。

一、整数常量

整数常量可以是十进制、八进制或十六进制的常量。前缀指定基数：0x 或 0X 表示十六进制，0 表示八进制，没有前缀则表示十进制。

整数常量也可以有后缀，可以是 U 和 L 的组合，其中，U 和 L 分别表示 `unsigned` 和 `long` 。后缀可以是大写或者小写，多个后缀以任意顺序进行组合。

这里有一些整数常量的实例：

```
212         /* 合法 */
215u        /* 合法 */
0xFeeL      /* 合法 */
078         /* 非法：8 不是一个八进制数字 */
032UU       /* 非法：不能重复后缀 */
```

以下是各种类型的整数常量的实例：

```
85         /* 十进制 */
0213       /* 八进制 */
0x4b       /* 十六进制 */
30         /* int */
30u        /* 无符号 int */
30l        /* long */
30ul       /* 无符号 long */
```

二、浮点常量

一个浮点常量是由整数部分、小数点、小数部分和指数部分组成。您可以使用小数形式或者指数形式来表示浮点常量。

这里有一些浮点常量的实例：

```
3.14159       /* 合法 */
314159E-5L    /* 合法 */
510E          /* 非法：不完全指数 */
210f          /* 非法：没有小数或指数 */
.e55          /* 非法：缺少整数或小数 */
```

使用小数形式表示时，必须包含小数点、指数或同时包含两者。使用指数形式表示时，必须包含整数部分、小数部分或同时包含两者。有符号的指数是用 e 或 E 表示的。

三、字符常量

字符常量是括在单引号里，例如，'x'，且可存储在一个简单的字符类型变量中。一个字符常量可以是一个普通字符（例如 'x'）、一个转义序列（例如 '\t'）或者一个通用字符（例如 '\u02C0'）。

在 C# 中有一些特定的字符，当它们的前面带有反斜杠时有特殊的意义，可用于表示换行符（\n）或制表符 tab（\t）。在这里，列出一些转义序列码：

转义序列	含义
\	\ 字符
\'	' 字符
\"	" 字符
\?	? 字符
\a	Alert 或 bell
\b	退格键（Backspace）
\f	换页符（Form feed）
\n	换行符（Newline）
\r	回车
\t	水平制表符 tab
\v	垂直制表符 tab
\ooo	一到三位的八进制数
\xhh . . .	一个或多个数字的十六进制数

以下是一些转义序列字符的实例：

```
namespace EscapeChar
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello\tWorld\n\n");
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Hello   World
```

四、字符串常量

字符常量是括在双引号 `""` 里，或者是括在 `@""` 里。字符串常量包含的字符与字符常量相似，可以是：普通字符、转义序列和通用字符

使用字符串常量时，可以把一个很长的行拆成多个行，可以使用空格分隔各个部分。

这里是一些字符串常量的实例。下面所列的各种形式表示相同的字符串。

```
string a = "hello, world";                  // hello, world
string b = @"hello, world";                // hello, world
string c = "hello \t world";                // hello     world
string d = @"hello \t world";                // hello \t world
string e = "Joe said \"Hello\" to me";      // Joe said "Hello" to me
string f = @"Joe said ""Hello"" to me";   // Joe said "Hello" to me
string g = "\\\\server\\share\\file.txt";   // \\server\share\file.txt
string h = @"\\server\share\file.txt";      // \\server\share\file.txt
string i = "one\r\ntwo\r\nthree";
string j = @"one
two
three";
```

五、定义常量

常量是使用 **const** 关键字来定义的 。定义一个常量的语法如下：

```
const <data_type> <constant_name> = value;
```

下面的代码演示了如何在程序中定义和使用常量：

```
using System;

namespace DeclaringConstants
{
    class Program
    {
        static void Main(string[] args)
        {
            const double pi = 3.14159; // 常量声明
            double r;
            Console.WriteLine("Enter Radius: ");
            r = Convert.ToDouble(Console.ReadLine());
            double areaCircle = pi * r * r;
            Console.WriteLine("Radius: {0}, Area: {1}", r, areaCircle);
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Enter Radius:
3
Radius: 3, Area: 28.27431
```

C# 可空类型

C# 提供了一个特殊的数据类型，**nullable** 类型（可空类型），可空类型可以表示其基础值类型正常范围内的值，再加上一个 `null` 值。

例如，`Nullable< Int32 >`，读作"可空的 `Int32`"，可以被赋值为 `-2,147,483,648` 到 `2,147,483,647` 之间的任意值，也可以被赋值为 `null` 值。类似的，`Nullable< bool >` 变量可以被赋值为 `true` 或 `false` 或 `null`。

在处理数据库和其他包含可能未赋值的元素的数据类型时，将 `null` 赋值给数值类型或布尔型的功能特别有用。例如，数据库中的布尔型字段可以存储值 `true` 或 `false`，或者，该字段也可以未定义。

声明一个 nullable 类型（可空类型）的语法如下：

```
< data_type> ? <variable_name> = null;
```

下面的实例演示了可空数据类型的用法：

```
using System;
namespace CalculatorApplication
{
   class NullablesAtShow
   {
      static void Main(string[] args)
      {
         int? num1 = null;
         int? num2 = 45;
         double? num3 = new double?();
         double? num4 = 3.14157;

         bool? boolval = new bool?();

         // 显示值

         Console.WriteLine("显示可空类型的值： {0}, {1}, {2}, {3}",
                           num1, num2, num3, num4);
```

```
            Console.WriteLine("一个可空的布尔值： {0}", boolval);
            Console.ReadLine();

        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
显示可空类型的值： , 45,  , 3.14157
一个可空的布尔值：
```

Null 合并运算符（ ?? ）

`Null` 合并运算符用于定义可空类型和引用类型的默认值。`Null` 合并运算符为类型转换定义了一个预设值，以防可空类型的值为 `Null` 。`Null` 合并运算符把操作数类型隐式转换为另一个可空（或不可空）的值类型的操作数的类型。

如果第一个操作数的值为 `null` ，则运算符返回第二个操作数的值，否则返回第一个操作数的值。下面的实例演示了这点：

```
using System;
namespace CalculatorApplication
{
   class NullablesAtShow
   {

      static void Main(string[] args)
      {

         double? num1 = null;
         double? num2 = 3.14157;
         double num3;
         num3 = num1 ?? 5.34;
         Console.WriteLine("num3 的值： {0}", num3);
         num3 = num2 ?? 5.34;
         Console.WriteLine("num3 的值： {0}", num3);
         Console.ReadLine();

      }
```

```
20.          }
21.      }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.       num3 的值： 5.34
2.       num3 的值： 3.14157
```

C# 字符串

在 C# 中，您可以使用字符数组来表示字符串，但是，更常见的做法是使用 `string` 关键字来声明一个字符串变量。`string` 关键字是 `System.String` 类的别名。

一、创建 String 对象

您可以使用以下方法之一来创建 `string` 对象：

- 通过给 `String` 变量指定一个字符串
- 通过使用 `String` 类构造函数
- 通过使用字符串串联运算符（ `+` ）
- 通过检索属性或调用一个返回字符串的方法
- 通过格式化方法来转换一个值或对象为它的字符串表示形式

下面的实例演示了这点：

```
using System;

namespace StringApplication
{
    class Program
    {
        static void Main(string[] args)
        {
           //字符串，字符串连接
            string fname, lname;
            fname = "Rowan";
            lname = "Atkinson";

            string fullname = fname + lname;
            Console.WriteLine("Full Name: {0}", fullname);

            //通过使用 string 构造函数
            char[] letters = { 'H', 'e', 'l', 'l','o' };
```

```
                string greetings = new string(letters);
                Console.WriteLine("Greetings: {0}", greetings);

                //方法返回字符串
                string[] sarray = { "Hello", "From", "Tutorials", "Point" };
                string message = String.Join(" ", sarray);
                Console.WriteLine("Message: {0}", message);

                //用于转化值的格式化方法
                DateTime waiting = new DateTime(2012, 10, 10, 17, 58, 1);
                string chat = String.Format("Message sent at {0:t} on {0:D}",
                waiting);
                Console.WriteLine("Message: {0}", chat);
                Console.ReadKey() ;
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    Full Name: RowanAtkinson
    Greetings: Hello
    Message: Hello From Tutorials Point
    Message: Message sent at 17:58 on Wednesday, 10 October 2012
```

二、String 类的属性

String 类有以下两个属性：

序号	属性名称 & 描述
1	Chars在当前 String 对象中获取 Char 对象的指定位置。
2	Length在当前的 String 对象中获取字符数。

三、String 类的方法

`String` 类有许多方法用于 `string` 对象的操作。下面的表格提供了一些最常用的方法：

序号	方法名称 & 描述

1	public static int Compare(string strA, string strB) 比较两个指定的 string 对象，并返回一个表示它们在排列顺序中相对位置的整数。该方法区分大小写。
2	public static int Compare(string strA, string strB, bool ignoreCase) 比较两个指定的 string 对象，并返回一个表示它们在排列顺序中相对位置的整数。但是，如果布尔参数为真时，该方法不区分大小写。
3	public static string Concat(string str0, string str1) 连接两个 string 对象。
4	public static string Concat(string str0, string str1, string str2) 连接三个 string 对象。
5	public static string Concat(string str0, string str1, string str2, string str3) 连接四个 string 对象。
6	public bool Contains(string value) 返回一个表示指定 string 对象是否出现在字符串中的值。
7	public static string Copy(string str) 创建一个与指定字符串具有相同值的新的 String 对象。
8	public void CopyTo(int sourceIndex, char[] destination, int destinationIndex, int count) 从 string 对象的指定位置开始复制指定数量的字符到 Unicode 字符数组中的指定位置。
9	public bool EndsWith(string value) 判断 string 对象的结尾是否匹配指定的字符串。
10	public bool Equals(string value) 判断当前的 string 对象是否与指定的 string 对象具有相同的值。
11	public static bool Equals(string a, string b) 判断两个指定的 string 对象是否具有相同的值。
12	public static string Format(string format, Object arg0) 把指定字符串中一个或多个格式项替换为指定对象的字符串表示形式。
13	public int IndexOf(char value) 返回指定 Unicode 字符在当前字符串中第一次出现的索引，索引从 0 开始。
14	public int IndexOf(string value) 返回指定字符串在该实例中第一次出现的索引，索引从 0 开始。
15	public int IndexOf(char value, int startIndex) 返回指定 Unicode 字符从该字符串中指定字符位置开始搜索第一次出现的索引，索引从 0 开始。
16	public int IndexOf(string value, int startIndex) 返回指定字符串从该实例中指定字符位置开始搜索第一次出现的索引，索引从 0 开始。
17	public int IndexOfAny(char[] anyOf) 返回某一个指定的 Unicode 字符数组中任意字符在该实例中第一次出现的索引，索引从 0 开始。
18	public int IndexOfAny(char[] anyOf, int startIndex) 返回某一个指定的 Unicode 字符数组中任意字符从该实例中指定字符位置开始搜索第一次出现的索引，索引从 0 开始。
19	public string Insert(int startIndex, string value) 返回一个新的字符串，其中，指定的字符串被插入在当前 string 对象的指定索引位置。
20	public static bool IsNullOrEmpty(string value) 指示指定的字符串是否为 null 或者是否为一个空的字符串。
21	public static string Join(string separator, string[] value) 连接一个字符串数组中的所有元素，使用指定的分隔符分隔每个元素。
22	public static string Join(string separator, string[] value, int startIndex, int count) 连接接一个字符串数组中的指定位置开始的指定元素，使用指定的分隔符分隔每个元素。
23	public int LastIndexOf(char value) 返回指定 Unicode 字符在当前 string 对象中最后一次出现的索引位置，索引从 0 开始。
24	public int LastIndexOf(string value) 返回指定字符串在当前 string 对象中最后一次出现的索引位置，索引从 0 开始。

25	public string Remove(int startIndex) 移除当前实例中的所有字符，从指定位置开始，一直到最后一个位置为止，并返回字符串。
26	public string Remove(int startIndex, int count) 从当前字符串的指定位置开始移除指定数量的字符，并返回字符串。
27	public string Replace(char oldChar, char newChar) 把当前 string 对象中，所有指定的 Unicode 字符替换为另一个指定的 Unicode 字符，并返回新的字符串。
28	public string Replace(string oldValue, string newValue) 把当前 string 对象中，所有指定的字符串替换为另一个指定的字符串，并返回新的字符串。
29	public string[] Split(params char[] separator) 返回一个字符串数组，包含当前的 string 对象中的子字符串，子字符串是使用指定的 Unicode 字符数组中的元素进行分隔的。
30	public string[] Split(char[] separator, int count) 返回一个字符串数组，包含当前的 string 对象中的子字符串，子字符串是使用指定的 Unicode 字符数组中的元素进行分隔的。int 参数指定要返回的子字符串的最大数目。
31	public bool StartsWith(string value) 判断字符串实例的开头是否匹配指定的字符串。
32	public char[] ToCharArray()返回一个带有当前 string 对象中所有字符的 Unicode 字符数组。
33	public char[] ToCharArray(int startIndex, int length) 返回一个带有当前 string 对象中所有字符的 Unicode 字符数组，从指定的索引开始，直到指定的长度为止。
34	public string ToLower()把字符串转换为小写并返回。
35	public string ToUpper()把字符串转换为大写并返回。
36	public string Trim()移除当前 String 对象中的所有前导空白字符和后置空白字符。

上面的方法列表并不详尽，请访问 MSDN 库，查看完整的方法列表和 `String` 类构造函数。

实例

下面的实例演示了上面提到的一些方法：

比较字符串

```
1.      using System;
2.
3.      namespace StringApplication
4.      {
5.          class StringProg
6.          {
7.              static void Main(string[] args)
8.              {
9.                  string str1 = "This is test";
10.                 string str2 = "This is text";
11.
12.                 if (String.Compare(str1, str2) == 0)
13.                 {
```

```
            Console.WriteLine(str1 + " and " + str2 +  " are equal.");
         }
         else
         {
            Console.WriteLine(str1 + " and " + str2 + " are not equal.");
         }
         Console.ReadKey() ;
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
This is test and This is text are not equal.
```

字符串包含字符串：

```
using System;

namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string str = "This is test";
         if (str.Contains("test"))
         {
            Console.WriteLine("The sequence 'test' was found.");
         }
         Console.ReadKey() ;
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
The sequence 'test' was found.
```

获取子字符串：

```
using System;
namespace StringApplication
{
    class StringProg
    {
        static void Main(string[] args)
        {
            string str = "Last night I dreamt of San Pedro";
            Console.WriteLine(str);
            string substr = str.Substring(23);
            Console.WriteLine(substr);
            Console.ReadKey() ;
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Last night I dreamt of San Pedro
San Pedro
```

连接字符串：

```
using System;

namespace StringApplication
{
   class StringProg
   {
      static void Main(string[] args)
      {
         string[] starray = new string[]{"Down the way nights are dark",
         "And the sun shines daily on the mountain top",
         "I took a trip on a sailing ship",
         "And when I reached Jamaica",
         "I made a stop"};

         string str = String.Join("\n", starray);
         Console.WriteLine(str);
         Console.ReadKey() ;
      }
```

```
19.          }
20.      }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.       Down the way nights are dark
2.       And the sun shines daily on the mountain top
3.       I took a trip on a sailing ship
4.       And when I reached Jamaica
5.       I made a stop
```

C# 数组

数组是一个存储相同类型元素的固定大小的顺序集合。数组是用来存储数据的集合，通常认为数组是一个同一类型变量的集合。

声明数组变量并不是声明 `number0`、`number1`、...、`number99` 一个个单独的变量，而是声明一个就像 `numbers` 这样的变量，然后使用 `numbers[0]`、`numbers[1]`、...、`numbers[99]` 来表示一个个单独的变量。数组中某个指定的元素是通过索引来访问的。

所有的数组都是由连续的内存位置组成的。最低的地址对应第一个元素，最高的地址对应最后一个元素。

一、声明数组

在 C# 中声明一个数组，您可以使用下面的语法：

```
1.      datatype[] arrayName;
```

其中，

- `datatype` 用于指定被存储在数组中的元素的类型。
- `[ ]` 指定数组的秩（维度）。秩指定数组的大小。
- `arrayName` 指定数组的名称。

例如：

```
1.      double[] balance;
```

二、初始化数组

声明一个数组不会在内存中初始化数组。当初始化数组变量时，您可以赋值给数组。

数组是一个引用类型，所以您需要使用 `new` 关键字来创建数组的实例。

例如：

```
1.      double[] balance = new double[10];
```

三、赋值给数组

您可以通过使用索引号赋值给一个单独的数组元素，比如：

```
1.      double[] balance = new double[10];
2.      balance[0] = 4500.0;
```

您可以在声明数组的同时给数组赋值，比如：

```
1.      double[] balance = { 2340.0, 4523.69, 3421.0};
```

您也可以创建并初始化一个数组，比如：

```
1.      int [] marks = new int[5]  { 99,  98, 92, 97, 95};
```

在上述情况下，你也可以省略数组的大小，比如：

```
1.      int [] marks = new int[]  { 99,  98, 92, 97, 95};
```

您也可以赋值一个数组变量到另一个目标数组变量中。在这种情况下，目标和源会指向相同的内存位置：

```
1.      int [] marks = new int[]  { 99,  98, 92, 97, 95};
2.      int[] score = marks;
```

当您创建一个数组时，C# 编译器会根据数组类型隐式初始化每个数组元素为一个默认值。例如，`int` 数组的所有元素都会被初始化为 0。

四、访问数组元素

元素是通过带索引的数组名称来访问的。这是通过把元素的索引放置在数组名称后的方括号中来实现的。例如：

```
1.      double salary = balance[9];
```

下面是一个实例，使用上面提到的三个概念，即声明、赋值、访问数组：

```
using System;
namespace ArrayApplication
{
   class MyArray
   {
      static void Main(string[] args)
      {
         int []  n = new int[10]; /* n 是一个带有 10 个整数的数组 */
         int i,j;


         /* 初始化数组 n 中的元素 */
         for ( i = 0; i < 10; i++ )
         {
            n[ i ] = i + 100;
         }

         /* 输出每个数组元素的值 */
         for (j = 0; j < 10; j++ )
         {
            Console.WriteLine("Element[{0}] = {1}", j, n[j]);
         }
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

五、使用 foreach 循环

在前面的实例中，我们使用一个 for 循环来访问每个数组元素。您也可以使用一个 foreach 语句来遍历数组。

```
using System;

namespace ArrayApplication
{
   class MyArray
   {
      static void Main(string[] args)
      {
         int []  n = new int[10]; /* n 是一个带有 10 个整数的数组 */


         /* 初始化数组 n 中的元素 */
         for ( int i = 0; i < 10; i++ )
         {
            n[i] = i + 100;
         }

         /* 输出每个数组元素的值 */
         foreach (int j in n )
         {
            int i = j-100;
            Console.WriteLine("Element[{0}] = {1}", i, j);
         }
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
```

```
6.        Element[5] = 105
7.        Element[6] = 106
8.        Element[7] = 107
9.        Element[8] = 108
10.       Element[9] = 109
```

六、多维数组

C# 支持多维数组。多维数组又称为矩形数组。

您可以声明一个 string 变量的二维数组，如下：

```
1.      string [,] names;
```

或者，您可以声明一个 `int` 变量的三维数组，如下：

```
1.      int [ , , ] m;
```

6.1 二维数组

多维数组最简单的形式是二维数组。一个二维数组，在本质上，是一个一维数组的列表。

一个二维数组可以被认为是一个带有 x 行和 y 列的表格。下面是一个二维数组，包含 3 行和 4 列：

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

因此，数组中的每个元素是使用形式为 a[i , j] 的元素名称来标识的，其中 a 是数组名称，i 和 j 是唯一标识 a 中每个元素的下标。

6.11 初始化二维数组

多维数组可以通过在括号内为每行指定值来进行初始化。下面是一个带有 3 行 4 列的数组。

```
       int [,] a = new int [3,4] {
        {0, 1, 2, 3} ,   /*  初始化索引号为 0 的行 */
        {4, 5, 6, 7} ,   /*  初始化索引号为 1 的行 */
        {8, 9, 10, 11}   /*  初始化索引号为 2 的行 */
       };
```

6.12 访问二维数组元素

二维数组中的元素是通过使用下标（即数组的行索引和列索引）来访问的。例如：

```
     int val = a[2,3];
```

上面的语句将获取数组中第 3 行第 4 个元素。您可以通过上面的示意图来进行验证。让我们来看看下面的程序，我们将使用嵌套循环来处理二维数组：

```
       using System;

       namespace ArrayApplication
       {
           class MyArray
           {
               static void Main(string[] args)
               {
                   /* 一个带有 5 行 2 列的数组 */
                   int[,] a = new int[5, 2] {{0,0}, {1,2}, {2,4}, {3,6}, {4,8}
};

                   int i, j;

                   /* 输出数组中每个元素的值 */
                   for (i = 0; i < 5; i++)
                   {
                       for (j = 0; j < 2; j++)
                       {
                           Console.WriteLine("a[{0},{1}] = {2}", i, j, a[i,j]);
                       }
                   }
                  Console.ReadKey();
               }
           }
```

```
25.         }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.          a[0,0]: 0
2.          a[0,1]: 0
3.          a[1,0]: 1
4.          a[1,1]: 2
5.          a[2,0]: 2
6.          a[2,1]: 4
7.          a[3,0]: 3
8.          a[3,1]: 6
9.          a[4,0]: 4
10.         a[4,1]: 8
```

七、交错数组

交错数组是数组的数组。您可以声明一个带有 int 值的交错数组 scores，如下所示：

```
1.      int [][] scores;
```

声明一个数组不会在内存中创建数组。创建上面的数组：

```
1.          int[][] scores = new int[5][];
2.          for (int i = 0; i < scores.Length; i++)
3.          {
4.             scores[i] = new int[4];
5.          }
```

您可以初始化一个交错数组，如下所示：

```
1.          int[][] scores = new int[2][]{new int[]{92,93,94},new int[]
  {85,66,87,88}};
```

其中，scores 是一个由两个整型数组组成的数组 — `scores[0]` 是一个带有 3 个整数的数组，`scores[1]` 是一个带有 4 个整数的数组。

下面的实例演示了如何使用交错数组：

```
using System;

namespace ArrayApplication
{
    class MyArray
    {
        static void Main(string[] args)
        {
            /* 一个由 5 个整型数组组成的交错数组 */
            int[][] a = new int[][]{new int[]{0,0},new int[]{1,2},
            new int[]{2,4},new int[]{ 3, 6 }, new int[]{ 4, 8 } };

            int i, j;

            /* 输出数组中每个元素的值 */
            for (i = 0; i < 5; i++)
            {
                for (j = 0; j < 2; j++)
                {
                    Console.WriteLine("a[{0}][{1}] = {2}", i, j, a[i]
[j]);
                }
            }
           Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
a[0][0] = 0
a[0][1] = 0
a[1][0] = 1
a[1][1] = 2
a[2][0] = 2
a[2][1] = 4
a[3][0] = 3
a[3][1] = 6
a[4][0] = 4
a[4][1] = 8
```

八、传递数组给函数

在 C# 中，您可以传递数组作为函数的参数。您可以通过指定不带索引的数组名称来给函数传递一个指向数组的指针。

下面的实例演示了如何传递数组给函数：

```
1.       using System;
2.
3.       namespace ArrayApplication
4.       {
5.          class MyArray
6.          {
7.             double getAverage(int[] arr, int size)
8.             {
9.                int i;
10.               double avg;
11.               int sum = 0;
12.
13.               for (i = 0; i < size; ++i)
14.               {
15.                  sum += arr[i];
16.               }
17.
18.               avg = (double)sum / size;
19.               return avg;
20.            }
21.            static void Main(string[] args)
22.            {
23.               MyArray app = new MyArray();
24.               /* 一个带有 5 个元素的 int 数组 */
25.               int [] balance = new int[]{1000, 2, 3, 17, 50};
26.               double avg;
27.
28.               /* 传递数组的指针作为参数 */
29.               avg = app.getAverage(balance, 5 ) ;
30.
31.               /* 输出返回值 */
32.               Console.WriteLine( "平均值是： {0} ", avg );
33.               Console.ReadKey();
34.            }
35.         }
```

```
36.       }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.      平均值是： 214.4
```

九、参数数组

有时，当声明一个方法时，您不能确定要传递给函数作为参数的参数数目。C# 参数数组解决了这个问题，参数数组通常用于传递未知数量的参数给函数。

params 关键字

在使用数组作为形参时，C# 提供了 `params` 关键字，使调用数组为形参的方法时，既可以传递数组实参，也可以只传递一组数组。 `params` 的使用格式为：

```
1.      public 返回类型 方法名称( params 类型名称[] 数组名称 )
```

下面的实例演示了如何使用参数数组：

```
1.        using System;
2.
3.        namespace ArrayApplication
4.        {
5.           class ParamArray
6.           {
7.              public int AddElements(params int[] arr)
8.              {
9.                 int sum = 0;
10.                foreach (int i in arr)
11.                {
12.                   sum += i;
13.                }
14.                return sum;
15.             }
16.          }
17.
18.          class TestClass
19.          {
20.             static void Main(string[] args)
```

```
            {
                ParamArray app = new ParamArray();
                int sum = app.AddElements(512, 720, 250, 567, 889);
                Console.WriteLine("总和是： {0}", sum);
                Console.ReadKey();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    总和是： 2938
```

十、Array 类

Array 类是 C# 中所有数组的基类，它是在 System 命名空间中定义。Array 类提供了各种用于数组的属性和方法。

10.1 Array 类的属性

下表列出了 Array 类中一些最常用的属性：

序号	属性 & 描述
1	IsFixedSize获取一个值，该值指示数组是否带有固定大小。
2	IsReadOnly获取一个值，该值指示数组是否只读。
3	Length获取一个 32 位整数，该值表示所有维度的数组中的元素总数。
4	LongLength获取一个 64 位整数，该值表示所有维度的数组中的元素总数。
5	Rank获取数组的秩（维度）。

如需了解 Array 类的完整的属性列表，请参阅微软的 C# 文档。

10.2 Array 类的方法

下表列出了 Array 类中一些最常用的方法：

序号	方法 & 描述
1	Clear根据元素的类型，设置数组中某个范围的元素为零、为 false 或者为 null。

2	Copy(Array, Array, Int32)从数组的第一个元素开始复制某个范围的元素到另一个数组的第一个元素位置。长度由一个 32 位整数指定。
3	CopyTo(Array, Int32)从当前的一维数组中复制所有的元素到一个指定的一维数组的指定索引位置。索引由一个 32 位整数指定。
4	GetLength 获取一个 32 位整数，该值表示指定维度的数组中的元素总数。
5	GetLongLength获取一个 64 位整数，该值表示指定维度的数组中的元素总数。
6	GetLowerBound获取数组中指定维度的下界。
7	GetType获取当前实例的类型。从对象（Object）继承。
8	GetUpperBound获取数组中指定维度的上界。
9	GetValue(Int32)获取一维数组中指定位置的值。索引由一个 32 位整数指定。
10	IndexOf(Array, Object)搜索指定的对象，返回整个一维数组中第一次出现的索引。
11	Reverse(Array)逆转整个一维数组中元素的顺序。
12	SetValue(Object, Int32)给一维数组中指定位置的元素设置值。索引由一个 32 位整数指定。
13	Sort(Array)使用数组的每个元素的 IComparable 实现来排序整个一维数组中的元素。
14	ToString返回一个表示当前对象的字符串。从对象（Object）继承。

如需了解 Array 类的完整的方法列表，请参阅微软的 C# 文档。

下面的程序演示了 Array 类的一些方法的用法：

```
using System;
namespace ArrayApplication
{
    class MyArray
    {

        static void Main(string[] args)
        {
            int[] list = { 34, 72, 13, 44, 25, 30, 10 };
            int[] temp = list;

            Console.Write("原始数组： ");
            foreach (int i in list)
            {
                Console.Write(i + " ");
            }
            Console.WriteLine();

            // 逆转数组
```

```
                    Array.Reverse(temp);
                    Console.Write("逆转数组： ");
                    foreach (int i in temp)
                    {
                        Console.Write(i + " ");
                    }
                    Console.WriteLine();

                    // 排序数组
                    Array.Sort(list);
                    Console.Write("排序数组： ");
                    foreach (int i in list)
                    {
                        Console.Write(i + " ");
                    }
                    Console.WriteLine();

                   Console.ReadKey();
               }
           }
       }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
        原始数组： 34 72 13 44 25 30 10
        逆转数组： 10 30 25 44 13 72 34
        排序数组： 10 13 25 30 34 44 72
```

C# 结构体

在 C# 中，结构体是值类型数据结构。它使得一个单一变量可以存储各种数据类型的相关数据。**struct** 关键字用于创建结构体。

结构体是用来代表一个记录。假设您想跟踪图书馆中书的动态。您可能想跟踪每本书的以下属性：

- Title
- Author
- Subject
- Book ID

一、定义结构体

为了定义一个结构体，您必须使用 `struct` 语句。`struct` 语句为程序定义了一个带有多个成员的新的数据类型。

例如，您可以按照如下的方式声明 Book 结构：

```
struct Books
{
   public string title;
   public string author;
   public string subject;
   public int book_id;
};
```

下面的程序演示了结构的用法：

```
using System;

struct Books
{
   public string title;
   public string author;
   public string subject;
   public int book_id;
```

```
        };

        public class testStructure
        {
           public static void Main(string[] args)
           {

              Books Book1;        /* 声明 Book1, 类型为 Book */
              Books Book2;        /* 声明 Book2, 类型为 Book */

              /* book 1 详述 */
              Book1.title = "C Programming";
              Book1.author = "Nuha Ali";
              Book1.subject = "C Programming Tutorial";
              Book1.book_id = 6495407;

              /* book 2 详述 */
              Book2.title = "Telecom Billing";
              Book2.author = "Zara Ali";
              Book2.subject =  "Telecom Billing Tutorial";
              Book2.book_id = 6495700;

              /* 打印 Book1 信息 */
              Console.WriteLine( "Book 1 title : {0}", Book1.title);
              Console.WriteLine("Book 1 author : {0}", Book1.author);
              Console.WriteLine("Book 1 subject : {0}", Book1.subject);
              Console.WriteLine("Book 1 book_id :{0}", Book1.book_id);

              /* 打印 Book2 信息 */
              Console.WriteLine("Book 2 title : {0}", Book2.title);
              Console.WriteLine("Book 2 author : {0}", Book2.author);
              Console.WriteLine("Book 2 subject : {0}", Book2.subject);
              Console.WriteLine("Book 2 book_id : {0}", Book2.book_id);

              Console.ReadKey();

           }
        }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
        Book 1 title : C Programming
```

```
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

二、C# 结构的特点

您已经用了一个简单的名为 Books 的结构。在 C# 中的结构与传统的 C 或 C++ 中的结构不同。C# 中的结构有以下特点：

- 结构可带有方法、字段、索引、属性、运算符方法和事件。
- 结构可定义构造函数，但不能定义析构函数。但是，您不能为结构定义默认的构造函数。默认的构造函数是自动定义的，且不能被改变。
- 与类不同，结构不能继承其他的结构或类。
- 结构不能作为其他结构或类的基础结构。
- 结构可实现一个或多个接口。
- 结构成员不能指定为 abstract、virtual 或 protected。
- 当您使用 New 操作符创建一个结构对象时，会调用适当的构造函数来创建结构。与类不同，结构可以不使用 New 操作符即可被实例化。
- 如果不使用 New 操作符，只有在所有的字段都被初始化之后，字段才被赋值，对象才被使用。

三、类 vs 结构

类和结构有以下几个基本的不同点：

```
    • 类是引用类型，结构是值类型。
    • 结构不支持继承。
    • 结构不能声明默认的构造函数。
```

针对上述讨论，让我们重写前面的实例：

```
using System;

struct Books
{
   private string title;
   private string author;
   private string subject;
   private int book_id;
   public void getValues(string t, string a, string s, int id)
   {
      title = t;
      author = a;
      subject = s;
      book_id =id;
   }
   public void display()
   {
      Console.WriteLine("Title : {0}", title);
      Console.WriteLine("Author : {0}", author);
      Console.WriteLine("Subject : {0}", subject);
      Console.WriteLine("Book_id :{0}", book_id);
   }

};

public class testStructure
{
   public static void Main(string[] args)
   {

      Books Book1 = new Books(); /* 声明 Book1, 类型为 Book */
      Books Book2 = new Books(); /* 声明 Book2, 类型为 Book */

      /* book 1 详述 */
      Book1.getValues("C Programming",
      "Nuha Ali", "C Programming Tutorial",6495407);

      /* book 2 详述 */
      Book2.getValues("Telecom Billing",
      "Zara Ali", "Telecom Billing Tutorial", 6495700);

      /* 打印 Book1 信息 */
```

```
            Book1.display();

            /* 打印 Book2 信息 */
            Book2.display();

            Console.ReadKey();

        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Title : C Programming
Author : Nuha Ali
Subject : C Programming Tutorial
Book_id : 6495407
Title : Telecom Billing
Author : Zara Ali
Subject : Telecom Billing Tutorial
Book_id : 6495700
```

C# 枚举

枚举是一组命名整型常量。枚举类型是使用 **enum** 关键字声明的。

C# 枚举是值数据类型。换句话说，枚举包含自己的值，且不能继承或传递继承。

一、声明 enum 变量

声明枚举的一般语法：

```
1.        enum <enum_name>
2.        {
3.            enumeration list
4.        };
```

其中，

1. • enum_name 指定枚举的类型名称。
2. • enumeration list 是一个用逗号分隔的标识符列表。

枚举列表中的每个符号代表一个整数值，一个比它前面的符号大的整数值。默认情况下，第一个枚举符号的值是 0.例如：

```
1.      enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };
```

实例

下面的实例演示了枚举变量的用法：

```
1.        using System;
2.        namespace EnumApplication
3.        {
4.           class EnumProgram
5.           {
6.              enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };
7.
8.              static void Main(string[] args)
9.              {
10.                int WeekdayStart = (int)Days.Mon;
```

```
            int WeekdayEnd = (int)Days.Fri;
            Console.WriteLine("Monday: {0}", WeekdayStart);
            Console.WriteLine("Friday: {0}", WeekdayEnd);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Monday: 1
Friday: 5
```

C# 类

当你定义一个类时，你定义了一个数据类型的蓝图。这实际上并没有定义任何的数据，但它定义了类的名称意味着什么，也就是说，类的对象由什么组成及在这个对象上可执行什么操作。对象是类的实例。构成类的方法和变量成为类的成员。

一、类的定义

类的定义是以关键字 `class` 开始，后跟类的名称。类的主体，包含在一对花括号内。下面是类定义的一般形式：

```
    <access specifier> class  class_name
    {
        // member variables
        <access specifier> <data type> variable1;
        <access specifier> <data type> variable2;
        ...
        <access specifier> <data type> variableN;
        // member methods
        <access specifier> <return type> method1(parameter_list)
        {
            // method body
        }
        <access specifier> <return type> method2(parameter_list)
        {
            // method body
        }
        ...
        <access specifier> <return type> methodN(parameter_list)
        {
            // method body
        }
    }
```

请注意：

- 访问标识符 指定了对类及其成员的访问规则。如果没有指定，则使用默认的访问标识符。类的默认访问标识符是 `internal`，成员的默认访问标识符是 `private`。

- 数据类型 指定了变量的类型，返回类型 指定了返回的方法返回的数据类型。
- 如果要访问类的成员，你要使用点（.）运算符。
- 点运算符链接了对象的名称和成员的名称。

下面的实例说明了目前为止所讨论的概念：

```
using System;
namespace BoxApplication
{
    class Box
    {
       public double length;   // 长度
       public double breadth;  // 宽度
       public double height;   // 高度
    }
    class Boxtester
    {
        static void Main(string[] args)
        {
            Box Box1 = new Box();        // 声明 Box1, 类型为 Box
            Box Box2 = new Box();        // 声明 Box2, 类型为 Box
            double volume = 0.0;         // 体积

            // Box1 详述
            Box1.height = 5.0;
            Box1.length = 6.0;
            Box1.breadth = 7.0;

            // Box2 详述
            Box2.height = 10.0;
            Box2.length = 12.0;
            Box2.breadth = 13.0;

            // Box1 的体积
            volume = Box1.height * Box1.length * Box1.breadth;
            Console.WriteLine("Box1 的体积： {0}",  volume);

            // Box2 的体积
            volume = Box2.height * Box2.length * Box2.breadth;
```

```
                Console.WriteLine("Box2 的体积： {0}", volume);
                Console.ReadKey();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    Box1 的体积： 210
    Box2 的体积： 1560
```

二、成员函数和封装

类的成员函数是一个在类定义中有它的定义或原型的函数，就像其他变量一样。作为类的一个成员，它能在类的任何对象上操作，且能访问该对象的类的所有成员。

成员变量是对象的属性（从设计角度），且它们保持私有来实现封装。这些变量只能使用公共成员函数来访问。

让我们使用上面的概念来设置和获取一个类中不同的类成员的值：

```
    using System;
    namespace BoxApplication
    {
        class Box
        {
           private double length;   // 长度
           private double breadth;  // 宽度
           private double height;   // 高度
           public void setLength( double len )
           {
                length = len;
           }

           public void setBreadth( double bre )
           {
                breadth = bre;
           }

           public void setHeight( double hei )
```

```
            {
                 height = hei;
            }
            public double getVolume()
            {
                return length * breadth * height;
            }
        }
        class Boxtester
        {
            static void Main(string[] args)
            {
                Box Box1 = new Box();        // 声明 Box1, 类型为 Box
                Box Box2 = new Box();         // 声明 Box2, 类型为 Box
                double volume;                 // 体积


                // Box1 详述
                Box1.setLength(6.0);
                Box1.setBreadth(7.0);
                Box1.setHeight(5.0);

                // Box2 详述
                Box2.setLength(12.0);
                Box2.setBreadth(13.0);
                Box2.setHeight(10.0);

                // Box1 的体积
                volume = Box1.getVolume();
                Console.WriteLine("Box1 的体积： {0}" ,volume);

                // Box2 的体积
                volume = Box2.getVolume();
                Console.WriteLine("Box2 的体积： {0}", volume);

                Console.ReadKey();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Box1 的体积： 210
Box2 的体积： 1560
```

三、C# 中的构造函数

类的 构造函数 是类的一个特殊的成员函数，当创建类的新对象时执行。

构造函数的名称与类的名称完全相同，它没有任何返回类型。

下面的实例说明了构造函数的概念：

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // 线条的长度
      public Line()
      {
         Console.WriteLine("对象已创建");
      }

      public void setLength( double len )
      {
         length = len;
      }
      public double getLength()
      {
         return length;
      }

      static void Main(string[] args)
      {
         Line line = new Line();
         // 设置线条长度
         line.setLength(6.0);
         Console.WriteLine("线条的长度： {0}", line.getLength());
         Console.ReadKey();
      }
   }
```

```
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
对象已创建
线条的长度： 6
```

默认的构造函数没有任何参数。但是如果你需要一个带有参数的构造函数可以有参数，这种构造函数叫做参数化构造函数。这种技术可以帮助你在创建对象的同时给对象赋初始值，具体请看下面实例：

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // 线条的长度
      public Line(double len)  // 参数化构造函数
      {
         Console.WriteLine("对象已创建, length = {0}", len);
         length = len;
      }

      public void setLength( double len )
      {
         length = len;
      }
      public double getLength()
      {
         return length;
      }

      static void Main(string[] args)
      {
         Line line = new Line(10.0);
         Console.WriteLine("线条的长度： {0}", line.getLength());
         // 设置线条长度
         line.setLength(6.0);
         Console.WriteLine("线条的长度： {0}", line.getLength());
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
对象已创建, length = 10
线条的长度： 10
线条的长度： 6
```

四、C# 中的析构函数

类的 析构函数 是类的一个特殊的成员函数，当类的对象超出范围时执行。

析构函数的名称是在类的名称前加上一个波浪形（~）作为前缀，它不返回值，也不带任何参数。

析构函数用于在结束程序（比如关闭文件、释放内存等）之前释放资源。析构函数不能继承或重载。

下面的实例说明了析构函数的概念：

```
using System;
namespace LineApplication
{
   class Line
   {
      private double length;   // 线条的长度
      public Line()  // 构造函数
      {
         Console.WriteLine("对象已创建");
      }
      ~Line() //析构函数
      {
         Console.WriteLine("对象已删除");
      }

      public void setLength( double len )
      {
         length = len;
      }
      public double getLength()
      {
         return length;
      }

```

```
25.          static void Main(string[] args)
26.          {
27.             Line line = new Line();
28.             // 设置线条长度
29.             line.setLength(6.0);
30.             Console.WriteLine("线条的长度： {0}", line.getLength());
31.          }
32.       }
33.    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.    对象已创建
2.    线条的长度： 6
3.    对象已删除
```

五、C# 类的静态成员

我们可以使用 **static** 关键字把类成员定义为静态的。当我们声明一个类成员为静态时，意味着无论有多少个类的对象被创建，只会有一个该静态成员的副本。

关键字 **static** 意味着类中只有一个该成员的实例。静态变量用于定义常量，因为它们的值可以通过直接调用类而不需要创建类的实例来获取。静态变量可在成员函数或类的定义外部进行初始化。你也可以在类的定义内部初始化静态变量。

下面的实例演示了静态变量的用法：

```
1.    using System;
2.    namespace StaticVarApplication
3.    {
4.        class StaticVar
5.        {
6.           public static int num;
7.            public void count()
8.            {
9.                num++;
10.           }
11.           public int getNum()
12.           {
13.               return num;
```

```
            }
        }
        class StaticTester
        {
            static void Main(string[] args)
            {
                StaticVar s1 = new StaticVar();
                StaticVar s2 = new StaticVar();
                s1.count();
                s1.count();
                s1.count();
                s2.count();
                s2.count();
                s2.count();
                Console.WriteLine("s1 的变量 num： {0}", s1.getNum());
                Console.WriteLine("s2 的变量 num： {0}", s2.getNum());
                Console.ReadKey();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    s1 的变量 num： 6
    s2 的变量 num： 6
```

你也可以把一个成员函数声明为 **static**。这样的函数只能访问静态变量。静态函数在对象被创建之前就已经存在。下面的实例演示了静态函数的用法：

```
    using System;
    namespace StaticVarApplication
    {
        class StaticVar
        {
           public static int num;
            public void count()
            {
                num++;
            }
            public static int getNum()
            {
                return num;
```

```
            }
        }
        class StaticTester
        {
            static void Main(string[] args)
            {
                StaticVar s = new StaticVar();
                s.count();
                s.count();
                s.count();
                Console.WriteLine("变量 num： {0}", StaticVar.getNum());
                Console.ReadKey();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    变量 num： 3
```

C# 继承

继承是面向对象程序设计中最重要的概念之一。继承允许我们根据一个类来定义另一个类，这使得创建和维护应用程序变得更容易。同时也有利于重用代码和节省开发时间。

当创建一个类时，程序员不需要完全重新编写新的数据成员和成员函数，只需要设计一个新的类，继承了已有的类的成员即可。这个已有的类被称为的基类，这个新的类被称为派生类。

继承的思想实现了 属于（**IS-A**） 关系。例如，哺乳动物 属于（**IS-A**） 动物，狗 属于（**IS-A**） 哺乳动物，因此狗 属于（**IS-A**） 动物。

一、基类和派生类

一个类可以派生自多个类或接口，这意味着它可以从多个基类或接口继承数据和函数。

C# 中创建派生类的语法如下：

```
1.       <acess-specifier> class <base_class>
2.       {
3.        ...
4.       }
5.       class <derived_class> : <base_class>
6.       {
7.        ...
8.       }
```

假设，有一个基类 `Shape` ，它的派生类是 `Rectangle` ：

```
1.       using System;
2.       namespace InheritanceApplication
3.       {
4.          class Shape
5.          {
6.             public void setWidth(int w)
7.             {
8.                width = w;
9.             }
10.            public void setHeight(int h)
11.            {
```

```
            height = h;
         }
         protected int width;
         protected int height;
      }

      // 派生类
      class Rectangle: Shape
      {
         public int getArea()
         {
            return (width * height);
         }
      }

      class RectangleTester
      {
         static void Main(string[] args)
         {
            Rectangle Rect = new Rectangle();

            Rect.setWidth(5);
            Rect.setHeight(7);

            // 打印对象的面积
            Console.WriteLine("总面积： {0}",  Rect.getArea());
            Console.ReadKey();
         }
      }
   }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    总面积： 35
```

二、基类的初始化

派生类继承了基类的成员变量和成员方法。因此父类对象应在子类对象创建之前被创建。您可以在成员初始化列表中进行父类的初始化。

下面的程序演示了这点：

```
using System;
namespace RectangleApplication
{
   class Rectangle
   {
      // 成员变量
      protected double length;
      protected double width;
      public Rectangle(double l, double w)
      {
         length = l;
         width = w;
      }
      public double GetArea()
      {
         return length * width;
      }
      public void Display()
      {
         Console.WriteLine("长度： {0}", length);
         Console.WriteLine("宽度： {0}", width);
         Console.WriteLine("面积： {0}", GetArea());
      }
   }//end class Rectangle
   class Tabletop : Rectangle
   {
      private double cost;
      public Tabletop(double l, double w) : base(l, w)
      { }
      public double GetCost()
      {
         double cost;
         cost = GetArea() * 70;
         return cost;
      }
      public void Display()
      {
         base.Display();
         Console.WriteLine("成本： {0}", GetCost());
      }
```

```
        }
        class ExecuteRectangle
        {
            static void Main(string[] args)
            {
                Tabletop t = new Tabletop(4.5, 7.5);
                t.Display();
                Console.ReadLine();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    长度： 4.5
    宽度： 7.5
    面积： 33.75
    成本： 2362.5
```

三、C# 多重继承

多重继承指的是一个类别可以同时从多于一个父类继承行为与特征的功能。与单一继承相对，单一继承指一个类别只可以继承自一个父类。

C# 不支持多重继承。但是，您可以使用接口来实现多重继承。下面的程序演示了这点：

```
    using System;
    namespace InheritanceApplication
    {
        class Shape
        {
            public void setWidth(int w)
            {
                width = w;
            }
            public void setHeight(int h)
            {
                height = h;
            }
            protected int width;
```

```
15.             protected int height;
16.          }
17.
18.          // 基类 PaintCost
19.          public interface PaintCost
20.          {
21.             int getCost(int area);
22.
23.          }
24.          // 派生类
25.          class Rectangle : Shape, PaintCost
26.          {
27.             public int getArea()
28.             {
29.                return (width * height);
30.             }
31.             public int getCost(int area)
32.             {
33.                return area * 70;
34.             }
35.          }
36.          class RectangleTester
37.          {
38.             static void Main(string[] args)
39.             {
40.                Rectangle Rect = new Rectangle();
41.                int area;
42.                Rect.setWidth(5);
43.                Rect.setHeight(7);
44.                area = Rect.getArea();
45.                // 打印对象的面积
46.                Console.WriteLine("总面积： {0}",  Rect.getArea());
47.                Console.WriteLine("油漆总成本： ${0}" , Rect.getCost(area));
48.                Console.ReadKey();
49.             }
50.          }
51.       }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.      总面积： 35
2.      油漆总成本： $2450
```

C# 多态性

多态性意味着有多重形式。在面向对象编程范式中，多态性往往表现为"一个接口，多个功能"。多态性可以是静态的或动态的。在静态多态性中，函数的响应是在编译时发生的。在动态多态性中，函数的响应是在运行时发生的。

一、静态多态性

在编译时，函数和对象的连接机制被称为早期绑定，也被称为静态绑定。C# 提供了两种技术来实现静态多态性。分别为：

1. • 函数重载
2. • 运算符重载

运算符重载将在下一章节讨论，接下来我们将讨论函数重载。

二、函数重载

您可以在同一个范围内对相同的函数名有多个定义。函数的定义必须彼此不同，可以是参数列表中的参数类型不同，也可以是参数个数不同。不能重载只有返回类型不同的函数声明。

下面的实例演示了几个相同的函数 `print()` ，用于打印不同的数据类型：

```
1.       using System;
2.       namespace PolymorphismApplication
3.       {
4.          class Printdata
5.          {
6.             void print(int i)
7.             {
8.                Console.WriteLine("Printing int: {0}", i );
9.             }
10.
11.            void print(double f)
12.            {
13.               Console.WriteLine("Printing float: {0}" , f);
14.            }
```

```
15.
16.             void print(string s)
17.             {
18.                Console.WriteLine("Printing string: {0}", s);
19.             }
20.             static void Main(string[] args)
21.             {
22.                Printdata p = new Printdata();
23.                // 调用 print 来打印整数
24.                p.print(5);
25.                // 调用 print 来打印浮点数
26.                p.print(500.263);
27.                // 调用 print 来打印字符串
28.                p.print("Hello C++");
29.                Console.ReadKey();
30.             }
31.          }
32.       }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.        Printing int: 5
2.        Printing float: 500.263
3.        Printing string: Hello C++
```

三、动态多态性

C# 允许您使用关键字 **abstract** 创建抽象类，用于提供接口的部分类的实现。当一个派生类继承自该抽象类时，实现即完成。抽象类包含抽象方法，抽象方法可被派生类实现。派生类具有更专业的功能。

请注意，下面是有关抽象类的一些规则：

- 您不能创建一个抽象类的实例。
- 您不能在一个抽象类外部声明一个抽象方法。
- 通过在类定义前面放置关键字 **sealed**，可以将类声明为密封类。当一个类被声明为 **sealed** 时，它不能被继承。抽象类不能被声明为 sealed。

下面的程序演示了一个抽象类：

```
using System;
namespace PolymorphismApplication
{
   abstract class Shape
   {
      public abstract int area();
   }
   class Rectangle:  Shape
   {
      private int length;
      private int width;
      public Rectangle( int a=0, int b=0)
      {
         length = a;
         width = b;
      }
      public override int area ()
      {
         Console.WriteLine("Rectangle 类的面积：");
         return (width * length);
      }
   }

   class RectangleTester
   {
      static void Main(string[] args)
      {
         Rectangle r = new Rectangle(10, 7);
         double a = r.area();
         Console.WriteLine("面积： {0}",a);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Rectangle 类的面积：
面积： 70
```

当有一个定义在类中的函数需要在继承类中实现时，可以使用虚方法。虚方法是使用关键字 **virtual**

声明的。虚方法可以在不同的继承类中有不同的实现。对虚方法的调用是在运行时发生的。

动态多态性是通过 抽象类 和 虚方法 实现的。

下面的程序演示了这点：

```
using System;
namespace PolymorphismApplication
{
   class Shape
   {
      protected int width, height;
      public Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      public virtual int area()
      {
         Console.WriteLine("父类的面积：");
         return 0;
      }
   }
   class Rectangle: Shape
   {
      public Rectangle( int a=0, int b=0): base(a, b)
      {

      }
      public override int area ()
      {
         Console.WriteLine("Rectangle 类的面积：");
         return (width * height);
      }
   }
   class Triangle: Shape
   {
      public Triangle(int a = 0, int b = 0): base(a, b)
      {

      }
      public override int area()
      {
```

```
            Console.WriteLine("Triangle 类的面积：");
            return (width * height / 2);
         }
      }
      class Caller
      {
         public void CallArea(Shape sh)
         {
            int a;
            a = sh.area();
            Console.WriteLine("面积： {0}", a);
         }
      }
      class Tester
      {

         static void Main(string[] args)
         {
            Caller c = new Caller();
            Rectangle r = new Rectangle(10, 7);
            Triangle t = new Triangle(10, 5);
            c.CallArea(r);
            c.CallArea(t);
            Console.ReadKey();
         }
      }
   }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
   Rectangle 类的面积：
   面积：70
   Triangle 类的面积：
   面积：25
```

运算符重载

您可以重定义或重载 C# 中内置的运算符。因此，程序员也可以使用用户自定义类型的运算符。重载运算符是具有特殊名称的函数，是通过关键字 **operator** 后跟运算符的符号来定义的。与其他函数一样，重载运算符有返回类型和参数列表。

例如，请看下面的函数：

```
1.      public static Box operator+ (Box b, Box c)
2.      {
3.         Box box = new Box();
4.         box.length = b.length + c.length;
5.         box.breadth = b.breadth + c.breadth;
6.         box.height = b.height + c.height;
7.         return box;
8.      }
```

上面的函数为用户自定义的类 Box 实现了加法运算符（+）。它把两个 Box 对象的属性相加，并返回相加后的 Box 对象。

一、运算符重载的实现

下面的程序演示了完整的实现：

```
1.      using System;
2.
3.      namespace OperatorOvlApplication
4.      {
5.         class Box
6.         {
7.            private double length;      // 长度
8.            private double breadth;     // 宽度
9.            private double height;      // 高度
10.
11.           public double getVolume()
12.           {
13.              return length * breadth * height;
14.           }
15.           public void setLength( double len )
```

```
16.          {
17.             length = len;
18.          }
19.
20.          public void setBreadth( double bre )
21.          {
22.             breadth = bre;
23.          }
24.
25.          public void setHeight( double hei )
26.          {
27.             height = hei;
28.          }
29.          // 重载 + 运算符来把两个 Box 对象相加
30.          public static Box operator+ (Box b, Box c)
31.          {
32.             Box box = new Box();
33.             box.length = b.length + c.length;
34.             box.breadth = b.breadth + c.breadth;
35.             box.height = b.height + c.height;
36.             return box;
37.          }
38.
39.       }
40.
41.       class Tester
42.       {
43.          static void Main(string[] args)
44.          {
45.             Box Box1 = new Box();         // 声明 Box1, 类型为 Box
46.             Box Box2 = new Box();         // 声明 Box2, 类型为 Box
47.             Box Box3 = new Box();         // 声明 Box3, 类型为 Box
48.             double volume = 0.0;          // 体积
49.
50.             // Box1 详述
51.             Box1.setLength(6.0);
52.             Box1.setBreadth(7.0);
53.             Box1.setHeight(5.0);
54.
55.             // Box2 详述
56.             Box2.setLength(12.0);
57.             Box2.setBreadth(13.0);
```

```
            Box2.setHeight(10.0);

            // Box1 的体积
            volume = Box1.getVolume();
            Console.WriteLine("Box1 的体积： {0}", volume);

            // Box2 的体积
            volume = Box2.getVolume();
            Console.WriteLine("Box2 的体积： {0}", volume);

            // 把两个对象相加
            Box3 = Box1 + Box2;

            // Box3 的体积
            volume = Box3.getVolume();
            Console.WriteLine("Box3 的体积： {0}", volume);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Box1 的体积： 210
Box2 的体积： 1560
Box3 的体积： 5400
```

二、可重载和不可重载运算符

下表描述了 C# 中运算符重载的能力：

运算符	描述
+, -, !, ~, ++, –	这些一元运算符只有一个操作数，且可以被重载。
+, -, , /, %	这些二元运算符带有两个操作数，且可以被重载。
==, !=, `<`, `>`, `<=`, `>=`	这些比较运算符可以被重载。
&&, \	\	这些条件逻辑运算符不能被直接重载。
+=, -=, =, /=, %=	这些赋值运算符不能被重载。
=, ., ?:, ->, new, is,	这些运算符不能被重载。

sizeof, typeof	这些运算符不能被重载。

实例

针对上述讨论，让我们扩展上面的实例，重载更多的运算符：

```
1.      using System;
2.
3.      namespace OperatorOvlApplication
4.      {
5.          class Box
6.          {
7.             private double length;      // 长度
8.             private double breadth;     // 宽度
9.             private double height;      // 高度
10.
11.            public double getVolume()
12.            {
13.              return length * breadth * height;
14.            }
15.           public void setLength( double len )
16.           {
17.               length = len;
18.           }
19.
20.           public void setBreadth( double bre )
21.           {
22.               breadth = bre;
23.           }
24.
25.           public void setHeight( double hei )
26.           {
27.               height = hei;
28.           }
29.           // 重载 + 运算符来把两个 Box 对象相加
30.           public static Box operator+ (Box b, Box c)
31.           {
32.               Box box = new Box();
33.               box.length = b.length + c.length;
34.               box.breadth = b.breadth + c.breadth;
35.               box.height = b.height + c.height;
```

```
36.             return box;
37.         }
38.
39.         public static bool operator == (Box lhs, Box rhs)
40.         {
41.             bool status = false;
42.             if (lhs.length == rhs.length && lhs.height == rhs.height
43.                && lhs.breadth == rhs.breadth)
44.             {
45.                 status = true;
46.             }
47.             return status;
48.         }
49.         public static bool operator !=(Box lhs, Box rhs)
50.         {
51.             bool status = false;
52.             if (lhs.length != rhs.length || lhs.height != rhs.height
53.                 || lhs.breadth != rhs.breadth)
54.             {
55.                 status = true;
56.             }
57.             return status;
58.         }
59.         public static bool operator <(Box lhs, Box rhs)
60.         {
61.             bool status = false;
62.             if (lhs.length < rhs.length && lhs.height
63.                 < rhs.height && lhs.breadth < rhs.breadth)
64.             {
65.                 status = true;
66.             }
67.             return status;
68.         }
69.
70.         public static bool operator >(Box lhs, Box rhs)
71.         {
72.             bool status = false;
73.             if (lhs.length > rhs.length && lhs.height
74.                 > rhs.height && lhs.breadth > rhs.breadth)
75.             {
76.                 status = true;
77.             }
```

```
78.             return status;
79.         }
80.
81.         public static bool operator <=(Box lhs, Box rhs)
82.         {
83.             bool status = false;
84.             if (lhs.length <= rhs.length && lhs.height
85.                 <= rhs.height && lhs.breadth <= rhs.breadth)
86.             {
87.                 status = true;
88.             }
89.             return status;
90.         }
91.
92.         public static bool operator >=(Box lhs, Box rhs)
93.         {
94.             bool status = false;
95.             if (lhs.length >= rhs.length && lhs.height
96.                >= rhs.height && lhs.breadth >= rhs.breadth)
97.             {
98.                 status = true;
99.             }
100.            return status;
101.        }
102.        public override string ToString()
103.        {
104.            return String.Format("({0}, {1}, {2})", length, breadth, height);
105.        }
106.
107.    }
108.
109.    class Tester
110.    {
111.        static void Main(string[] args)
112.        {
113.          Box Box1 = new Box();          // 声明 Box1, 类型为 Box
114.          Box Box2 = new Box();          // 声明 Box2, 类型为 Box
115.          Box Box3 = new Box();          // 声明 Box3, 类型为 Box
116.          Box Box4 = new Box();
117.          double volume = 0.0;   // 体积
118.
119.          // Box1 详述
```

```
            Box1.setLength(6.0);
            Box1.setBreadth(7.0);
            Box1.setHeight(5.0);

            // Box2 详述
            Box2.setLength(12.0);
            Box2.setBreadth(13.0);
            Box2.setHeight(10.0);

           // 使用重载的 ToString() 显示两个盒子
            Console.WriteLine("Box1： {0}", Box1.ToString());
            Console.WriteLine("Box2： {0}", Box2.ToString());

            // Box1 的体积
            volume = Box1.getVolume();
            Console.WriteLine("Box1 的体积： {0}", volume);

            // Box2 的体积
            volume = Box2.getVolume();
            Console.WriteLine("Box2 的体积： {0}", volume);

            // 把两个对象相加
            Box3 = Box1 + Box2;
            Console.WriteLine("Box3： {0}", Box3.ToString());
            // Box3 的体积
            volume = Box3.getVolume();
            Console.WriteLine("Box3 的体积： {0}", volume);

            //comparing the boxes
            if (Box1 > Box2)
              Console.WriteLine("Box1 大于 Box2");
            else
              Console.WriteLine("Box1 不大于 Box2");
            if (Box1 < Box2)
              Console.WriteLine("Box1 小于 Box2");
            else
              Console.WriteLine("Box1 不小于 Box2");
            if (Box1 >= Box2)
              Console.WriteLine("Box1 大于等于 Box2");
            else
              Console.WriteLine("Box1 不大于等于 Box2");
            if (Box1 <= Box2)
```

```
            Console.WriteLine("Box1 小于等于 Box2");
          else
            Console.WriteLine("Box1 不小于等于 Box2");
          if (Box1 != Box2)
            Console.WriteLine("Box1 不等于 Box2");
          else
            Console.WriteLine("Box1 等于 Box2");
          Box4 = Box3;
          if (Box3 == Box4)
            Console.WriteLine("Box3 等于 Box4");
          else
            Console.WriteLine("Box3 不等于 Box4");

          Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    Box1： (6, 7, 5)
    Box2： (12, 13, 10)
    Box1 的体积： 210
    Box2 的体积： 1560
    Box3： (18, 20, 15)
    Box3 的体积： 5400
    Box1 不大于 Box2
    Box1 小于 Box2
    Box1 不大于等于 Box2
    Box1 小于等于 Box2
    Box1 不等于 Box2
    Box3 等于 Box4
```

C# 接口

接口定义了所有类继承接口时应遵循的语法合同。接口定义了语法合同 "是什么" 部分，派生类定义了语法合同 "怎么做" 部分。

接口定义了属性、方法和事件，这些都是接口的成员。接口只包含了成员的声明。成员的定义是派生类的责任。接口提供了派生类应遵循的标准结构。

接口使得实现接口的类或结构在形式上保持一致。

抽象类在某种程度上与接口类似，但是，它们大多只是用在当只有少数方法由基类声明由派生类实现时。

一、定义接口：MyInterface.cs

接口使用 **interface** 关键字声明，它与类的声明类似。接口声明默认是 `public` 的。下面是一个接口声明的实例：

```
1.      interface IMyInterface
2.      {
3.          void MethodToImplement();
4.      }
```

以上代码定义了接口 `IMyInterface` 。通常接口命令以 **I** 字母开头，这个接口只有一个方法 `MethodToImplement()` ，没有参数和返回值，当然我们可以按照需求设置参数和返回值。

值得注意的是，该方法并没有具体的实现。

接下来我们来实现以上接口：**InterfaceImplementer.cs**

```
1.      using System;
2.
3.      interface IMyInterface
4.      {
5.          // 接口成员
6.          void MethodToImplement();
7.      }
8.
```

```
 9.      class InterfaceImplementer : IMyInterface
10.      {
11.          static void Main()
12.          {
13.              InterfaceImplementer iImp = new InterfaceImplementer();
14.              iImp.MethodToImplement();
15.          }
16.
17.          public void MethodToImplement()
18.          {
19.              Console.WriteLine("MethodToImplement() called.");
20.          }
21.      }
```

`InterfaceImplementer` 类实现了 `IMyInterface` 接口，接口的实现与类的继承语法格式类似：

```
1.      class InterfaceImplementer : IMyInterface
```

继承接口后，我们需要实现接口的方法 MethodToImplement() ， 方法名必须与接口定义的方法名一致。

二、接口继承: InterfaceInheritance.cs

以下实例定义了两个接口 `IMyInterface` 和 `IParentInterface` 。

如果一个接口继承其他接口，那么实现类或结构就需要实现所有接口的成员。

以下实例 `IMyInterface` 继承了 `IParentInterface` 接口，因此接口实现类必须实现 `MethodToImplement()` 和 `ParentInterfaceMethod()` 方法：

```
1.      using System;
2.
3.      interface IParentInterface
4.      {
5.          void ParentInterfaceMethod();
6.      }
7.
8.      interface IMyInterface : IParentInterface
9.      {
10.         void MethodToImplement();
```

```
    }

    class InterfaceImplementer : IMyInterface
    {
        static void Main()
        {
            InterfaceImplementer iImp = new InterfaceImplementer();
            iImp.MethodToImplement();
            iImp.ParentInterfaceMethod();
        }

        public void MethodToImplement()
        {
            Console.WriteLine("MethodToImplement() called.");
        }

        public void ParentInterfaceMethod()
        {
            Console.WriteLine("ParentInterfaceMethod() called.");
        }
    }
```

实例输出结果为：

```
    MethodToImplement() called.
    ParentInterfaceMethod() called.
```

C# 命名空间

命名空间的设计目的是提供一种让一组名称与其他名称分隔开的方式。在一个命名空间中声明的类的名称与另一个命名空间中声明的相同的类的名称不冲突。

一、定义命名空间

命名空间的定义是以关键字 **namespace** 开始，后跟命名空间的名称，如下所示：

```
namespace namespace_name
{
   // 代码声明
}
```

为了调用支持命名空间版本的函数或变量，会把命名空间的名称置于前面，如下所示：

```
namespace_name.item_name;
```

下面的程序演示了命名空间的用法：

```
using System;
namespace first_space
{
   class namespace_cl
   {
      public void func()
      {
         Console.WriteLine("Inside first_space");
      }
   }
}
namespace second_space
{
   class namespace_cl
   {
      public void func()
      {
         Console.WriteLine("Inside second_space");
      }
```

```
20.          }
21.       }
22.       class TestClass
23.       {
24.          static void Main(string[] args)
25.          {
26.             first_space.namespace_cl fc = new first_space.namespace_cl();
27.             second_space.namespace_cl sc = new second_space.namespace_cl();
28.             fc.func();
29.             sc.func();
30.             Console.ReadKey();
31.          }
32.       }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.      Inside first_space
2.      Inside second_space
```

二、using 关键字

using 关键字表明程序使用的是给定命名空间中的名称。例如，我们在程序中使用 **System** 命名空间，其中定义了类 Console。我们可以只写：

```
1.      Console.WriteLine ("Hello there");
```

我们可以写完全限定名称，如下：

```
1.      System.Console.WriteLine("Hello there");
```

您也可以使用 **using** 命名空间指令，这样在使用的时候就不用在前面加上命名空间名称。该指令告诉编译器随后的代码使用了指定命名空间中的名称。下面的代码演示了命名空间的应用。

让我们使用 using 指定重写上面的实例：

```
1.       using System;
2.       using first_space;
3.       using second_space;
4.
5.       namespace first_space
```

```
    {
        class abc
        {
            public void func()
            {
                Console.WriteLine("Inside first_space");
            }
        }
    }
    namespace second_space
    {
        class efg
        {
            public void func()
            {
                Console.WriteLine("Inside second_space");
            }
        }
    }
    class TestClass
    {
        static void Main(string[] args)
        {
            abc fc = new abc();
            efg sc = new efg();
            fc.func();
            sc.func();
            Console.ReadKey();
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    Inside first_space
    Inside second_space
```

三、嵌套命名空间

命名空间可以被嵌套，即您可以在一个命名空间内定义另一个命名空间，如下所示：

```
namespace namespace_name1
{
   // 代码声明
   namespace namespace_name2
   {
     // 代码声明
   }
}
```

您可以使用点（.）运算符访问嵌套的命名空间的成员，如下所示：

```
using System;
using SomeNameSpace;
using SomeNameSpace.Nested;

namespace SomeNameSpace
{
    public class MyClass
    {
        static void Main()
        {
            Console.WriteLine("In SomeNameSpace");
            Nested.NestedNameSpaceClass.SayHello();
        }
    }

    // 内嵌命名空间
    namespace Nested
    {
        public class NestedNameSpaceClass
        {
            public static void SayHello()
            {
                Console.WriteLine("In Nested");
            }
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.        In SomeNameSpace
2.        In Nested
```

C# 异常处理

异常是在程序执行期间出现的问题。C# 中的异常是对程序运行时出现的特殊情况的一种响应，比如尝试除以零。

异常提供了一种把程序控制权从某个部分转移到另一个部分的方式。C# 异常处理时建立在四个关键词之上的：**try**、**catch**、**finally** 和 **throw**。

- try：一个 try 块标识了一个将被激活的特定的异常的代码块。后跟一个或多个 catch 块。
- catch：程序通过异常处理程序捕获异常。catch 关键字表示异常的捕获。
- finally：finally 块用于执行给定的语句，不管异常是否被抛出都会执行。例如，如果您打开一个文件，不管是否出现异常文件都要被关闭。
- throw：当问题出现时，程序抛出一个异常。使用 throw 关键字来完成。

语法

假设一个块将出现异常，一个方法使用 try 和 catch 关键字捕获异常。try/catch 块内的代码为受保护的代码，使用 try/catch 语法如下所示：

```
1.      try
2.      {
3.          // 引起异常的语句
4.      }
5.      catch( ExceptionName e1 )
6.      {
7.          // 错误处理代码
8.      }
9.      catch( ExceptionName e2 )
10.     {
11.         // 错误处理代码
12.     }
13.     catch( ExceptionName eN )
14.     {
15.         // 错误处理代码
16.     }
17.     finally
18.     {
```

```
19.             // 要执行的语句
20.         }
```

您可以列出多个 catch 语句捕获不同类型的异常，以防 try 块在不同的情况下生成多个异常。

一、C# 中的异常类

C# 异常是使用类来表示的。C# 中的异常类主要是直接或间接地派生于 **System.Exception** 类。**System.ApplicationException** 和 **System.SystemException** 类是派生于 System.Exception 类的异常类。

System.ApplicationException 类支持由应用程序生成的异常。所以程序员定义的异常都应派生自该类。

System.SystemException 类是所有预定义的系统异常的基类。

下表列出了一些派生自 Sytem.SystemException 类的预定义的异常类:

异常类	描述
System.IO.IOException	处理 I/O 错误。
System.IndexOutOfRangeException	处理当方法指向超出范围的数组索引时生成的错误。
System.ArrayTypeMismatchException	处理当数组类型不匹配时生成的错误。
System.NullReferenceException	处理当依从一个空对象时生成的错误。
System.DivideByZeroException	处理当除以零时生成的错误。
System.InvalidCastException	处理在类型转换期间生成的错误。
System.OutOfMemoryException	处理空闲内存不足生成的错误。
System.StackOverflowException	处理栈溢出生成的错误。

二、异常处理

C# 以 try 和 catch 块的形式提供了一种结构化的异常处理方案。使用这些块，把核心程序语句与错误处理语句分离开。

这些错误处理块是使用 **try**、**catch** 和 **finally** 关键字实现的。下面是一个当除以零时抛出异常的实例:

```
1.      using System;
2.      namespace ErrorHandlingApplication
```

```
    {
        class DivNumbers
        {
            int result;
            DivNumbers()
            {
                result = 0;
            }
            public void division(int num1, int num2)
            {
                try
                {
                    result = num1 / num2;
                }
                catch (DivideByZeroException e)
                {
                    Console.WriteLine("Exception caught: {0}", e);
                }
                finally
                {
                    Console.WriteLine("Result: {0}", result);
                }

            }
            static void Main(string[] args)
            {
                DivNumbers d = new DivNumbers();
                d.division(25, 0);
                Console.ReadKey();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    Exception caught: System.DivideByZeroException: Attempted to divide by
zero.
    at ...
    Result: 0
```

三、创建用户自定义异常

您也可以定义自己的异常。用户自定义的异常类是派生自 **ApplicationException** 类。下面的实例演示了这点：

```
using System;
namespace UserDefinedException
{
   class TestTemperature
   {
      static void Main(string[] args)
      {
         Temperature temp = new Temperature();
         try
         {
            temp.showTemp();
         }
         catch(TempIsZeroException e)
         {
            Console.WriteLine("TempIsZeroException: {0}", e.Message);
         }
         Console.ReadKey();
      }
   }
}
public class TempIsZeroException: ApplicationException
{
   public TempIsZeroException(string message): base(message)
   {
   }
}
public class Temperature
{
   int temperature = 0;
   public void showTemp()
   {
      if(temperature == 0)
      {
         throw (new TempIsZeroException("Zero Temperature found"));
      }
      else
      {
```

```
                Console.WriteLine("Temperature: {0}", temperature);
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    TempIsZeroException: Zero Temperature found
```

四、抛出对象

如果异常是直接或间接派生自 **System.Exception** 类，您可以抛出一个对象。您可以在 catch 块中使用 throw 语句来抛出当前的对象，如下所示：

```
    Catch(Exception e)
    {
        ...
        Throw e
    }
```

C# 预处理器指令

预处理器指令指导编译器在实际编译开始之前对信息进行预处理。

所有的预处理器指令都是以 # 开始。且在一行上，只有空白字符可以出现在预处理器指令之前。预处理器指令不是语句，所以它们不以分号（;）结束。

C# 编译器没有一个单独的预处理器，但是，指令被处理时就像是有一个单独的预处理器一样。在 C# 中，预处理器指令用于在条件编译中起作用。与 C 和 C++ 不同的是，它们不是用来创建宏。一个预处理器指令必须是该行上的唯一指令。

一、C# 预处理器指令列表

下表列出了 C# 中可用的预处理器指令：

预处理器指令	描述
#define	它用于定义一系列成为符号的字符。
#undef	它用于取消定义符号。
#if	它用于测试符号是否为真。
#else	它用于创建复合条件指令，与 #if 一起使用。
#elif	它用于创建复合条件指令。
#endif	指定一个条件指令的结束。
#line	它可以让您修改编译器的行数以及（可选地）输出错误和警告的文件名。
#error	它允许从代码的指定位置生成一个错误。
#warning	它允许从代码的指定位置生成一级警告。
#region	它可以让您在使用 Visual Studio Code Editor 的大纲特性时，指定一个可展开或折叠的代码块。
#endregion	它标识着 #region 块的结束。

二、#define 预处理器

#define 预处理器指令创建符号常量。**#define** 允许您定义一个符号，这样，通过使用符号作为传递给 #if 指令的表达式，表达式将返回 true。它的语法如下：

```
1.      #define symbol
```

下面的程序说明了这点：

```
#define PI
using System;
namespace PreprocessorDAppl
{
   class Program
   {
      static void Main(string[] args)
      {
         #if (PI)
            Console.WriteLine("PI is defined");
         #else
            Console.WriteLine("PI is not defined");
         #endif
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
PI is defined
```

三、条件指令

您可以使用 #if 指令来创建一个条件指令。条件指令用于测试符号是否为真。如果为真，编译器会执行 #if 和下一个指令之间的代码。条件指令的语法：

```
#if symbol [operator symbol]...
```

其中，symbol 是要测试的符号名称。您也可以使用 true 和 false，或在符号前放置否定运算符。

常见运算符有：

- == (等于)
- != (不等于)
- && (与)
- || (或)

您也可以用括号把符号和运算符进行分组。条件指令用于在调试版本或编译指定配置时编译代码。一个以 **#if** 指令开始的条件指令，必须显示地以一个 **#endif** 指令终止。

下面的程序演示了条件指令的用法：

```
#define DEBUG
#define VC_V10
using System;
public class TestClass
{
   public static void Main()
   {

      #if (DEBUG && !VC_V10)
         Console.WriteLine("DEBUG is defined");
      #elif (!DEBUG && VC_V10)
         Console.WriteLine("VC_V10 is defined");
      #elif (DEBUG && VC_V10)
         Console.WriteLine("DEBUG and VC_V10 are defined");
      #else
         Console.WriteLine("DEBUG and VC_V10 are not defined");
      #endif
      Console.ReadKey();
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
DEBUG and VC_V10 are defined
```

C# 正则表达式

正则表达式 是一种匹配输入文本的模式。 .Net 框架提供了允许这种匹配的正则表达式引擎。模式由一个或多个字符、运算符和结构组成。

定义正则表达式

下面列出了用于定义正则表达式的各种类别的字符、运算符和结构。

1. • 字符转义
2. • 字符类
3. • 定位点
4. • 分组构造
5. • 限定符
6. • 反向引用构造
7. • 备用构造
8. • 替换
9. • 杂项构造

一、字符转义

正则表达式中的反斜杠字符（\）指示其后跟的字符是特殊字符，或应按原义解释该字符。

下表列出了转义字符：

转义字符	描述	模式	匹配
\a	与报警 (bell) 符 \u0007 匹配。	\a	"Warning!" + '\u0007' 中的 "\u0007"
\b	在字符类中，与退格键 \u0008 匹配。	[\b]{3,}	"\b\b\b\b" 中的 "\b\b\b\b"
\t	与制表符 \u0009 匹配。	(\w+)\t	"Name\tAddr\t" 中的 "Name\t" 和 "Addr\t"
\r	与回车符 \u000D 匹配。（\r 与换行符 \n 不是等效的。）	\r\n(\w+)	"\r\Hello\nWorld." 中的 "\r\nHello"
\v	与垂直制表符 \u000B 匹配。	[\v]{2,}	"\v\v\v" 中的 "\v\v\v"
\f	与换页符 \u000C 匹配。	[\f]{2,}	"\f\f\f" 中的 "\f\f\f"
\n	与换行符 \u000A 匹配。	\r\n(\w+)	"\r\Hello\nWorld." 中的 "\r\nHello"

\e	与转义符 \u001B 匹配。	\e "\x001B" 中的 "\x001B"	
\nnn	使用八进制表示形式指定一个字符（nnn 由二到三位数字组成）。	\w\040\w	"a bc d" 中的 "a b" 和 "c d"
\xnn	使用十六进制表示形式指定字符（nn 恰好由两位数字组成）。	\w\x20\w	"a bc d" 中的 "a b" 和 "c d"
\c X \c x	匹配 X 或 x 指定的 ASCII 控件字符，其中 X 或 x 是控件字符的字母。	\cC	"\x0003" 中的 "\x0003" (Ctrl-C)
\unnnn	使用十六进制表示形式匹配一个 Unicode 字符（由 nnnn 表示的四位数）。	\w\u0020\w	"a bc d" 中的 "a b" 和 "c d"
\	在后面带有不识别的转义字符时，与该字符匹配。	\d+[+-x]\d+\d+[+-x\d+	"(2+2) _ 3_9" 中的 "2+2" 和 "3*9"

二、字符类

字符类与一组字符中的任何一个字符匹配。

下表列出了字符类：

字符类	描述	模式	匹配
[character*group*]	匹配 character_group 中的任何单个字符。默认情况下，匹配区分大小写。	[mn]	"mat" 中的 "m"，"moon" 中的 "m" 和 "n"
character_group	非：与不在 character_group 中的任何单个字符匹配。 默认情况下，character_group 中的字符区分大小写。	aei	"avail" 中的 "v" 和 "l"
[first - last]	字符范围：与从 first 到 last 的范围中的任何单个字符匹配。	(\w+)\t	"Name\tAddr\t" 中的 "Name\t" 和 "Addr\t"
.	通配符：与除 \n 之外的任何单个字符匹配。若要匹配原意句点字符（. 或 \u002E），您必须在该字符前面加上转义符 (.)。	a.e	"have" 中的 "ave"， "mate" 中的 "ate"
\p{ name }	与 name 指定的 Unicode 通用类别或命名块中的任何单个字符匹配。	\p{Lu}	"City Lights" 中的 "C" 和 "L"
\P{ name }	与不在 name 指定的 Unicode 通用类别或命名块中的任何单个字符匹配。	\P{Lu}	"City" 中的 "i"、 "t" 和 "y"
\w	与任何单词字符匹配。	\w	"Room#1" 中的 "R"、 "o"、 "m" 和 "1"
\W	与任何非单词字符匹配。	\W	"Room#1" 中的 "#"
\s	与任何空白字符匹配。	\w\s	"ID A1.3" 中的 "D "
\S	与任何非空白字符匹配。	\s\S	"int __ctr" 中的 " "

\d	与任何十进制数字匹配。	\d	"4 = IV" 中的 "4"
\D	匹配不是十进制数的任意字符。	\D	"4 = IV" 中的 " "、 "="、 " "、 "I" 和 "V"

三、定位点

定位点或原子零宽度断言会使匹配成功或失败，具体取决于字符串中的当前位置，但它们不会使引擎在字符串中前进或使用字符。

下表列出了定位点：

断言	描述	模式	匹配
^	匹配必须从字符串或一行的开头开始。	^\d{3}	"567-777-" 中的 "567"
$	匹配必须出现在字符串的末尾或出现在行或字符串末尾的 \n 之前。	-\d{4}$	"8-12-2012" 中的 "-2012"
\A	匹配必须出现在字符串的开头。	\A\w{4}	"Code-007-" 中的 "Code"
\Z	匹配必须出现在字符串的末尾或出现在字符串末尾的 \n 之前。	-\d{3}\Z	"Bond-901-007" 中的 "-007"
\z	匹配必须出现在字符串的末尾。	-\d{3}\z	"-901-333" 中的 "-333"
\G	匹配必须出现在上一个匹配结束的地方。	\G(\d)	"(1)(3)(5)7" 中的 "(1)"、 "(3)" 和 "(5)"
\b	匹配一个单词边界，也就是指单词和空格间的位置。	er\b	匹配"never"中的"er"，但不能匹配"verb"中的"er"。
\B	匹配非单词边界。	er\B	匹配"verb"中的"er"，但不能匹配"never"中的"er"。

四、分组构造

分组构造描述了正则表达式的子表达式，通常用于捕获输入字符串的子字符串。

下表列出了分组构造：

分组构造	描述	模式	
(subexpression)	捕获匹配的子表达式并将其分配到一个从零开始的序号中。	(\w)\1	"de
(?< name >subexpression)	将匹配的子表达式捕获到一个命名组中。	(?< double>\w)\k< double>	"de

(?< name1 -name2 >subexpression)	定义平衡组定义。	(((?'Open'()())+((?'Close-Open'))[()](https://shenjun.gitbooks.io/c/content/一、CSharp 基础教程/#fn%5C(%5C)))+)*(?(Open)(?!))$	"3+ 的 '
(?: subexpression)	定义非捕获组。	Write(?:Line)?	"Co 中的
(?imnsx-imnsx:subexpression)	应用或禁用 subexpression 中指定的选项。	A\d{2}(?i:\w+)\b	"A1 中的 "A1
(?= subexpression)	零宽度正预测先行断言。	\w+(?=.)	"He The "is
(?! subexpression)	零宽度负预测先行断言。	\b(?!un)\w+\b	"un use "us
(?<=subexpression)	零宽度正回顾后发断言。	(?<=19)\d{2}\b	"18 200 "99
(?<! subexpression)	零宽度负回顾后发断言。	(?	"Hi 的 '
(?> subexpression)	非回溯（也称为"贪婪"）子表达式。	13579	"1A 中的 和 '

实例

```
1.      using System;
2.      using System.Text.RegularExpressions;
3.
4.      public class Example
5.      {
6.         public static void Main()
7.         {
8.            string input = "1851 1999 1950 1905 2003";
9.            string pattern = @"(?<=19)\d{2}\b";
10.
11.           foreach (Match match in Regex.Matches(input, pattern))
12.              Console.WriteLine(match.Value);
13.        }
14.     }
```

五、限定符

限定符指定在输入字符串中必须存在上一个元素（可以是字符、组或字符类）的多少个实例才能出现匹配项。 限定符包括下表中列出的语言元素。

下表列出了限定符：

限定符	描述	模式	匹配
	匹配上一个元素零次或多次。	\d.\d	".0"、 "19.9"、 "219.9"
+	匹配上一个元素一次或多次。	"be+"	"been" 中的 "bee"， "bent" 中的 "be"
?	匹配上一个元素零次或一次。	"rai?n"	"ran"、 "rain"
{ n }	匹配上一个元素恰好 n 次。	",\d{3}"	"1,043.6" 中的 ",043"，"9,876,543,210" 中的 ",876"、",543" 和 ",210"
{ n ,}	匹配上一个元素至少 n 次。	"\d{2,}"	"166"、 "29"、 "1930"
{ n , m }	匹配上一个元素至少 n 次，但不多于 m 次。	"\d{3,5}"	"166"， "17668"， "193024" 中的 "19302"
?	匹配上一个元素零次或多次，但次数尽可能少。	\d?.\d	".0"、 "19.9"、 "219.9"
+?	匹配上一个元素一次或多次，但次数尽可能少。	"be+?"	"been" 中的 "be"， "bent" 中的 "be"
??	匹配上一个元素零次或一次，但次数尽可能少。	"rai??n"	"ran"、 "rain"
{ n }?	匹配前导元素恰好 n 次。	",\d{3}?"	"1,043.6" 中的 ",043"，"9,876,543,210" 中的 ",876"、",543" 和 ",210"
{ n ,}?	匹配上一个元素至少 n 次，但次数尽可能少。	"\d{2,}?"	"166"、 "29" 和 "1930"
{ n , m }?	匹配上一个元素的次数介于 n 和 m 之间，但次数尽可能少。	"\d{3,5}?"	"166"， "17668"， "193024" 中的 "193" 和 "024"

六、反向引用构造

反向引用允许在同一正则表达式中随后标识以前匹配的子表达式。

下表列出了反向引用构造：

反向引用构造	描述	模式	匹配
\ number	反向引用。 匹配编号子表达式的值。	(\w)\1	"seek" 中的 "ee"
\k< name >	命名反向引用。 匹配命名表达式的值。	(?< char>\w)\k< char>	"seek" 中的 "ee"

七、备用构造

备用构造用于修改正则表达式以启用 either/or 匹配。

下表列出了备用构造：

备用构造	描述	模式	匹配			
|	匹配以竖线 (|) 字符分隔的任何一个元素。	th(e|is|at)	"this is the" 中的 "th" 和 "th"			
(?(expression)yes|no)	如果正则表达式模式由 expression 匹配指定，则匹配 yes；否则匹配可选的 no 部分。expression 被解释为零宽度断言。	(?(A)A\d{2}\b|\b\d{3}\b)	"A10 C103 910" 中的 "A10" 和 "910"			
(?(name)yes|no)	如果 name 或已命名或已编号的捕获组具有匹配，则匹配 yes；否则匹配可选的 no。	(?<quoted>")?(?(quoted).+?"|\S+\s)	"Dogs.jpg "Yiska playing.jpg"" 中的 Dogs.jpg 和 "Yiska playing.jpg"			

八、替换

替换是替换模式中使用的正则表达式。

下表列出了用于替换的字符：

字符	描述	模式	替换模式	输入字符串	结果字符串
$number	替换按组 number 匹配的子字符串。	\b(\w+)(\s)(\w+)\b	$3$2$1	"one two"	"two one"
${name}	替换按命名组 name 匹配的子字符串。	\b(?<word1>\w+)(\s)(?<word2>\w+)\b	${word2} ${word1}	"one two"	"two one"

$$	替换字符"$"。	\b(\d+)\s?USD	$$$1	"103 USD"	"$103"
$&	替换整个匹配项的一个副本。	(\$(\d(.+\d+)?){1})	$&	"$1.30"	"$1.30**"
$`	替换匹配前的输入字符串的所有文本。	"AABBCC"	"AAAACC"		
$'	替换匹配后的输入字符串的所有文本。	B+	$'	"AABBCC"	"AACCCC"
$+	替换最后捕获的组。	B+(C+)	$+	"AABBCCDD"	AACCDD
$_	替换整个输入字符串。	B+	$_	"AABBCC"	"AAAABBCCCC"

九、杂项构造

下表列出了各种杂项构造：

构造	描述	实例
(?imnsx-imnsx)	在模式中间对诸如不区分大小写这样的选项进行设置或禁用。	\bA(?i)b\w+\b 匹配 "ABA Able Act" 中的 "ABA" 和 "Able"
(?#注释)	内联注释。该注释在第一个右括号处终止。	\bA(?#匹配以A开头的单词)\w+\b
# [行尾]	该注释以非转义的 # 开头，并继续到行的结尾。	(?x)\bA\w+\b#匹配以 A 开头的单词

十、Regex 类

Regex 类用于表示一个正则表达式。

下表列出了 Regex 类中一些常用的方法：

序号	方法 & 描述
1	public bool IsMatch(string input) 指示 Regex 构造函数中指定的正则表达式是否在指定的输入字符串中找到匹配项。
2	public bool IsMatch(string input, int startat) 指示 Regex 构造函数中指定的正则表达式是否在指定的输入字符串中找到匹配项，从字符串中指定的开始位置开始。
3	public static bool IsMatch(string input, string pattern) 指示指定的正则表达式是否在指定的输入字符串中找到匹配项。

4	public MatchCollection Matches(string input) 在指定的输入字符串中搜索正则表达式的所有匹配项。
5	public string Replace(string input, string replacement) 在指定的输入字符串中，把所有匹配正则表达式模式的所有匹配的字符串替换为指定的替换字符串。
6	public string[] Split(string input) 把输入字符串分割为子字符串数组，根据在 Regex 构造函数中指定的正则表达式模式定义的位置进行分割。

如需了解 Regex 类的完整的属性列表，请参阅微软的 C# 文档。

实例 1

下面的实例匹配了以 'S' 开头的单词：

```
1.       using System;
2.       using System.Text.RegularExpressions;
3.
4.       namespace RegExApplication
5.       {
6.          class Program
7.          {
8.             private static void showMatch(string text, string expr)
9.             {
10.               Console.WriteLine("The Expression: " + expr);
11.               MatchCollection mc = Regex.Matches(text, expr);
12.               foreach (Match m in mc)
13.               {
14.                  Console.WriteLine(m);
15.               }
16.            }
17.            static void Main(string[] args)
18.            {
19.               string str = "A Thousand Splendid Suns";
20.
21.               Console.WriteLine("Matching words that start with 'S': ");
22.               showMatch(str, @"\bS\S*");
23.               Console.ReadKey();
24.            }
25.         }
26.      }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.       Matching words that start with 'S':
```

```
2.       The Expression: \bS\S*
3.       Splendid
4.       Suns
```

实例 2

下面的实例匹配了以 'm' 开头以 'e' 结尾的单词：

```
1.       using System;
2.       using System.Text.RegularExpressions;
3.
4.       namespace RegExApplication
5.       {
6.          class Program
7.          {
8.             private static void showMatch(string text, string expr)
9.             {
10.               Console.WriteLine("The Expression: " + expr);
11.               MatchCollection mc = Regex.Matches(text, expr);
12.               foreach (Match m in mc)
13.               {
14.                  Console.WriteLine(m);
15.               }
16.            }
17.            static void Main(string[] args)
18.            {
19.               string str = "make maze and manage to measure it";
20.
                  Console.WriteLine("Matching words start with 'm' and ends with
21. 'e':");
22.               showMatch(str, @"\bm\S*e\b");
23.               Console.ReadKey();
24.            }
25.         }
26.      }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.       Matching words start with 'm' and ends with 'e':
2.       The Expression: \bm\S*e\b
```

```
make
maze
manage
measure
```

实例 3

下面的实例替换掉多余的空格：

```
using System;
using System.Text.RegularExpressions;

namespace RegExApplication
{
   class Program
   {
      static void Main(string[] args)
      {
         string input = "Hello   World   ";
         string pattern = "\\s+";
         string replacement = " ";
         Regex rgx = new Regex(pattern);
         string result = rgx.Replace(input, replacement);

         Console.WriteLine("Original String: {0}", input);
         Console.WriteLine("Replacement String: {0}", result);
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Original String: Hello   World
Replacement String: Hello World
```

C# 文件处理

一个 文件 是一个存储在磁盘中带有指定名称和目录路径的数据集合。当打开文件进行读写时，它变成一个 流。

从根本上说，流是通过通信路径传递的字节序列。有两个主要的流：输入流 和 输出流。输入流用于从文件读取数据（读操作），输出流用于向文件写入数据（写操作）。

一、C# I/O 类

System.IO 命名空间有各种不同的类，用于执行各种文件操作，如创建和删除文件、读取或写入文件，关闭文件等。

下表列出了一些 System.IO 命名空间中常用的非抽象类：

I/O 类	描述
BinaryReader	从二进制流读取原始数据。
BinaryWriter	以二进制格式写入原始数据。
BufferedStream	字节流的临时存储。
Directory	有助于操作目录结构。
DirectoryInfo	用于对目录执行操作。
DriveInfo	提供驱动器的信息。
File	有助于处理文件。
FileInfo	用于对文件执行操作。
FileStream	用于文件中任何位置的读写。
MemoryStream	用于随机访问存储在内存中的数据流。
Path	对路径信息执行操作。
StreamReader	用于从字节流中读取字符。
StreamWriter	用于向一个流中写入字符。
StringReader	用于读取字符串缓冲区。
StringWriter	用于写入字符串缓冲区。

二、FileStream 类

System.IO 命名空间中的 **FileStream** 类有助于文件的读写与关闭。该类派生自抽象类 Stream。

您需要创建一个 **FileStream** 对象来创建一个新的文件，或打开一个已有的文件。创建 **FileStream** 对象的语法如下：

```
FileStream <object_name> = new FileStream( <file_name>,
<FileMode Enumerator>, <FileAccess Enumerator>, <FileShare Enumerator>);
```

例如，创建一个 FileStream 对象 F 来读取名为 sample.txt 的文件：

```
FileStream F = new FileStream("sample.txt", FileMode.Open, FileAccess.Read,
FileShare.Read);
```

参数	描述
FileMode	FileMode 枚举定义了各种打开文件的方法。FileMode 枚举的成员有：• Append：打开一个已有的文件，并将光标放置在文件的末尾。如果文件不存在，则创建文件。• Create：创建一个新的文件。如果文件已存在，则删除旧文件，然后创建新文件。• CreateNew：指定操作系统应创建一个新的文件。如果文件已存在，则抛出异常。• Open：打开一个已有的文件。如果文件不存在，则抛出异常。• OpenOrCreate：指定操作系统应打开一个已有的文件。如果文件不存在，则用指定的名称创建一个新的文件打开。• Truncate：打开一个已有的文件，文件一旦打开，就将被截断为零字节大小。然后我们可以向文件写入全新的数据，但是保留文件的初始创建日期。如果文件不存在，则抛出异常。
FileAccess	FileAccess 枚举的成员有：Read、ReadWrite 和 Write。
FileShare	FileShare 枚举的成员有：• Inheritable：允许文件句柄可由子进程继承。Win32 不直接支持此功能。• None：谢绝共享当前文件。文件关闭前，打开该文件的任何请求（由此进程或另一进程发出的请求）都将失败。• Read：允许随后打开文件读取。如果未指定此标志，则文件关闭前，任何打开该文件以进行读取的请求（由此进程或另一进程发出的请求）都将失败。但是，即使指定了此标志，仍可能需要附加权限才能够访问该文件。• ReadWrite：允许随后打开文件读取或写入。如果未指定此标志，则文件关闭前，任何打开该文件以进行读取或写入的请求（由此进程或另一进程发出）都将失败。但是，即使指定了此标志，仍可能需要附加权限才能够访问该文件。• Write：允许随后打开文件写入。如果未指定此标志，则文件关闭前，任何打开该文件以进行写入的请求（由此进程或另一进过程发出的请求）都将失败。但是，即使指定了此标志，仍可能需要附加权限才能够访问该文件。• Delete：允许随后删除文件。

实例

下面的程序演示了 FileStream 类的用法：

```
using System;
using System.IO;

namespace FileIOApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            FileStream F = new FileStream("test.dat",
```

```
                FileMode.OpenOrCreate, FileAccess.ReadWrite);

                for (int i = 1; i <= 20; i++)
                {
                    F.WriteByte((byte)i);
                }

                F.Position = 0;

                for (int i = 0; i <= 20; i++)
                {
                    Console.Write(F.ReadByte() + " ");
                }
                F.Close();
                Console.ReadKey();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 -1
```

三、C# 高级文件操作

上面的实例演示了 C# 中简单的文件操作。但是，要充分利用 C# System.IO 类的强大功能，您需要知道这些类常用的属性和方法。在下面的章节中，我们将讨论这些类和它们执行的操作。

四、文本文件的读写

StreamReader 和 **StreamWriter** 类用于文本文件的数据读写。这些类从抽象基类 Stream 继承，Stream 支持文件流的字节读写。

4.1 StreamReader 类

StreamReader 类继承自抽象基类 TextReader，表示阅读器读取一系列字符。

下表列出了 **StreamReader** 类中一些常用的方法：

序号	方法 & 描述
1	public override void Close()关闭 StreamReader 对象和基础流，并释放任何与读者相关的系统资源。
2	public override int Peek()返回下一个可用的字符，但不使用它。
3	public override int Read()从输入流中读取下一个字符，并把字符位置往前移一个字符。

如需查看完整的方法列表，请访问微软的 C# 文档。实例

下面的实例演示了读取名为 Jamaica.txt 的文件。文件如下：

```
    Down the way where the nights are gay
    And the sun shines daily on the mountain top
    I took a trip on a sailing ship
    And when I reached Jamaica
    I made a stop
```

```
    using System;
    using System.IO;

    namespace FileApplication
    {
        class Program
        {
            static void Main(string[] args)
            {
                try
                {
                    // 创建一个 StreamReader 的实例来读取文件
                    // using 语句也能关闭 StreamReader
                    using (StreamReader sr = new
StreamReader("c:/jamaica.txt"))
                    {
                        string line;

                        // 从文件读取并显示行，直到文件的末尾
                        while ((line = sr.ReadLine()) != null)
                        {
                            Console.WriteLine(line);
```

```
                    }
                }
            }
            catch (Exception e)
            {
                // 向用户显示出错消息
                Console.WriteLine("The file could not be read:");
                Console.WriteLine(e.Message);
            }
            Console.ReadKey();
        }
    }
}
```

当您编译和执行上面的程序时，它会显示文件的内容。

4.2 StreamWriter 类

StreamWriter 类继承自抽象类 TextWriter，表示编写器写入一系列字符。

下表列出了 **StreamWriter** 类中一些常用的方法：

序号	方法 & 描述
1	public override void Close()关闭当前的 StreamWriter 对象和基础流。
2	public override void Flush()清理当前编写器的所有缓冲区，使得所有缓冲数据写入基础流。
3	public virtual void Write(bool value)把一个布尔值的文本表示形式写入到文本字符串或流。（继承自 TextWriter。）
4	public override void Write(char value) 把一个字符写入到流。
5	public virtual void Write(decimal value) 把一个十进制值的文本表示形式写入到文本字符串或流。
6	public virtual void Write(double value) 把一个 8 字节浮点值的文本表示形式写入到文本字符串或流。
7	public virtual void Write(int value) 把一个 4 字节有符号整数的文本表示形式写入到文本字符串或流。
8	public override void Write(string value) 把一个字符串写入到流。
9	public virtual void WriteLine()把行结束符写入到文本字符串或流。

如需查看完整的方法列表，请访问微软的 C# 文档。

实例

下面的实例演示了使用 StreamWriter 类向文件写入文本数据：

```
using System;
using System.IO;

namespace FileApplication
{
    class Program
    {
        static void Main(string[] args)
        {

            string[] names = new string[] {"Zara Ali", "Nuha Ali"};
            using (StreamWriter sw = new StreamWriter("names.txt"))
            {
                foreach (string s in names)
                {
                    sw.WriteLine(s);

                }
            }

            // 从文件中读取并显示每行
            string line = "";
            using (StreamReader sr = new StreamReader("names.txt"))
            {
                while ((line = sr.ReadLine()) != null)
                {
                    Console.WriteLine(line);
                }
            }
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Zara Ali
Nuha Ali
```

五、二进制文件的读写

BinaryReader 和 **BinaryWriter** 类用于二进制文件的读写。

5.1 BinaryReader 类

BinaryReader 类用于从文件读取二进制数据。一个 **BinaryReader** 对象通过向它的构造函数传递 **FileStream** 对象而被创建。

下表列出了 BinaryReader 类中一些常用的方法：

序号	方法 & 描述
1	public override void Close()关闭 BinaryReader 对象和基础流。
2	public virtual int Read()从基础流中读取字符，并把流的当前位置往前移。
3	public virtual bool ReadBoolean()从当前流中读取一个布尔值，并把流的当前位置往前移一个字节。
4	public virtual byte ReadByte()从当前流中读取下一个字节，并把流的当前位置往前移一个字节。
5	public virtual byte[] ReadBytes(int count) 从当前流中读取指定数目的字节到一个字节数组中，并把流的当前位置往前移指定数目的字节。
6	public virtual char ReadChar()从当前流中读取下一个字节，并把流的当前位置按照所使用的编码和从流中读取的指定的字符往前移。
7	public virtual char[] ReadChars(int count)从当前流中读取指定数目的字节，在一个字符数组中返回数组，并把流的当前位置按照所使用的编码和从流中读取的指定的字符往前移。
8	public virtual double ReadDouble()从当前流中读取一个 8 字节浮点值，并把流的当前位置往前移八个字节。
9	public virtual int ReadInt32()从当前流中读取一个 4 字节有符号整数，并把流的当前位置往前移四个字节。
10	public virtual string ReadString() 从当前流中读取一个字符串。字符串以长度作为前缀，同时编码为一个七位的整数。

如需查看完整的方法列表，请访问微软的 C# 文档。

5.2 BinaryWriter 类

BinaryWriter 类用于向文件写入二进制数据。一个 **BinaryWriter** 对象通过向它的构造函数传递 **FileStream** 对象而被创建。

下表列出了 **BinaryWriter** 类中一些常用的方法：

序号	方法 & 描述
1	public override void Close()关闭 BinaryWriter 对象和基础流。
2	public virtual void Flush()清理当前编写器的所有缓冲区，使得所有缓冲数据写入基础设备。
3	public virtual long Seek(int offset, SeekOrigin origin) 设置当前流内的位置。
4	public virtual void Write(bool value) 把一个单字节的布尔值写入到当前流中，0 表示 false，1 表示 true。
5	public virtual void Write(byte value) 把一个无符号字节写入到当前流中，并把流的位置往前移一个字节。
6	public virtual void Write(byte[] buffer) 把一个字节数组写入到基础流中。
7	public virtual void Write(char ch) 把一个 Unicode 字符写入到当前流中，并把流的当前位置按照所使用的编码和要写入到流中的指定的字符往前移。
8	public virtual void Write(char[] chars) 把一个字符数组写入到当前流中，并把流的当前位置按照所使用的编码和要写入到流中的指定的字符往前移。
9	public virtual void Write(double value) 把一个 8 字节浮点值写入到当前流中，并把流位置往前移八个字节。
10	public virtual void Write(int value) 把一个 4 字节有符号整数写入到当前流中，并把流位置往前移四个字节。
11	public virtual void Write(string value) 把一个以长度为前缀的字符串写入到 BinaryWriter 的当前编码的流中，并把流的当前位置按照所使用的编码和要写入到流中的指定的字符往前移。

如需查看完整的方法列表，请访问微软的 C# 文档。

实例

下面的实例演示了读取和写入二进制数据：

```
using System;
using System.IO;

namespace BinaryFileApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            BinaryWriter bw;
            BinaryReader br;
            int i = 25;
            double d = 3.14157;
```

```
14.                 bool b = true;
15.                 string s = "I am happy";
16.                 // 创建文件
17.                 try
18.                 {
19.                     bw = new BinaryWriter(new FileStream("mydata",
20.                     FileMode.Create));
21.                 }
22.                 catch (IOException e)
23.                 {
24.                     Console.WriteLine(e.Message + "\n Cannot create file.");
25.                     return;
26.                 }
27.                 // 写入文件
28.                 try
29.                 {
30.                     bw.Write(i);
31.                     bw.Write(d);
32.                     bw.Write(b);
33.                     bw.Write(s);
34.                 }
35.                 catch (IOException e)
36.                 {
37.                     Console.WriteLine(e.Message + "\n Cannot write to file.");
38.                     return;
39.                 }
40.
41.                 bw.Close();
42.                 // 读取文件
43.                 try
44.                 {
45.                     br = new BinaryReader(new FileStream("mydata",
46.                     FileMode.Open));
47.                 }
48.                 catch (IOException e)
49.                 {
50.                     Console.WriteLine(e.Message + "\n Cannot open file.");
51.                     return;
52.                 }
53.                 try
54.                 {
55.                     i = br.ReadInt32();
```

```
                    Console.WriteLine("Integer data: {0}", i);
                    d = br.ReadDouble();
                    Console.WriteLine("Double data: {0}", d);
                    b = br.ReadBoolean();
                    Console.WriteLine("Boolean data: {0}", b);
                    s = br.ReadString();
                    Console.WriteLine("String data: {0}", s);
                }
                catch (IOException e)
                {
                    Console.WriteLine(e.Message + "\n Cannot read from file.");
                    return;
                }
                br.Close();
                Console.ReadKey();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    Integer data: 25
    Double data: 3.14157
    Boolean data: True
    String data: I am happy
```

六、Windows 文件系统的操作

C# 允许您使用各种目录和文件相关的类来操作目录和文件，比如 **DirectoryInfo** 类和 **FileInfo** 类。

6.1 DirectoryInfo 类

DirectoryInfo 类派生自 **FileSystemInfo** 类。它提供了各种用于创建、移动、浏览目录和子目录的方法。该类不能被继承。

下表列出了 **DirectoryInfo** 类中一些常用的属性：

序号	属性 & 描述

1	Attributes获取当前文件或目录的属性。
2	CreationTime获取当前文件或目录的创建时间。
3	Exists获取一个表示目录是否存在的布尔值。
4	Extension获取表示文件存在的字符串。
5	FullName获取目录或文件的完整路径。
6	LastAccessTime获取当前文件或目录最后被访问的时间。
7	Name获取该 DirectoryInfo 实例的名称。

下表列出了 **DirectoryInfo** 类中一些常用的方法:

序号	方法 & 描述
1	public void Create()创建一个目录。
2	public DirectoryInfo CreateSubdirectory(string path) 在指定的路径上创建子目录。指定的路径可以是相对于 DirectoryInfo 类的实例的路径。
3	public override void Delete()如果为空的，则删除该 DirectoryInfo。
4	public DirectoryInfo[] GetDirectories()返回当前目录的子目录。
5	public FileInfo[] GetFiles()从当前目录返回文件列表。

如需查看完整的属性和方法列表，请访问微软的 C# 文档。

6.2 FileInfo 类

FileInfo 类派生自 **FileSystemInfo** 类。它提供了用于创建、复制、删除、移动、打开文件的属性和方法，且有助于 FileStream 对象的创建。该类不能被继承。

下表列出了 **FileInfo** 类中一些常用的属性:

序号	属性 & 描述
1	Attributes获取当前文件的属性。
2	CreationTime获取当前文件的创建时间。
3	Directory获取文件所属目录的一个实例。
4	Exists获取一个表示文件是否存在的布尔值。
5	Extension获取表示文件存在的字符串。
6	FullName获取文件的完整路径。
7	LastAccessTime获取当前文件最后被访问的时间。
8	LastWriteTime获取文件最后被写入的时间。

9	Length获取当前文件的大小，以字节为单位。
10	Name获取文件的名称。

下表列出了 **FileInfo** 类中一些常用的方法：

序号	方法 & 描述
1	public StreamWriter AppendText()创建一个 StreamWriter，追加文本到由 FileInfo 的实例表示的文件中。
2	public FileStream Create()创建一个文件。
3	public override void Delete()永久删除一个文件。
4	public void MoveTo(string destFileName) 移动一个指定的文件到一个新的位置，提供选项来指定新的文件名。
5	public FileStream Open(FileMode mode) 以指定的模式打开一个文件。
6	public FileStream Open(FileMode mode, FileAccess access) 以指定的模式，使用 read、write 或 read/write 访问，来打开一个文件。
7	public FileStream Open(FileMode mode, FileAccess access, FileShare share) 以指定的模式，使用 read、write 或 read/write 访问，以及指定的分享选项，来打开一个文件。
8	public FileStream OpenRead()创建一个只读的 FileStream。
9	public FileStream OpenWrite()创建一个只写的 FileStream。

如需查看完整的属性和方法列表，请访问微软的 C# 文档。

实例

下面的实例演示了上面提到的类的用法：

```
1.      using System;
2.      using System.IO;
3.
4.      namespace WindowsFileApplication
5.      {
6.          class Program
7.          {
8.              static void Main(string[] args)
9.              {
10.                 // 创建一个 DirectoryInfo 对象
11.                 DirectoryInfo mydir = new DirectoryInfo(@"c:\Windows");
```

```
12.
13.                 // 获取目录中的文件以及它们的名称和大小
14.                 FileInfo [] f = mydir.GetFiles();
15.                 foreach (FileInfo file in f)
16.                 {
17.                     Console.WriteLine("File Name: {0} Size: {1}",
18.                         file.Name, file.Length);
19.                 }
20.                 Console.ReadKey();
21.             }
22.         }
23.     }
```

当您编译和执行上面的程序时，它会显示文件的名称及它们在 Windows 目录中的大小。

C# 属性

属性（**Property**） 是类（class）、结构（structure）和接口（interface）的命名（named）成员。类或结构中的成员变量或方法称为 域（Field）。属性（Property）是域（Field）的扩展，且可使用相同的语法来访问。它们使用 访问器（**accessors**） 让私有域的值可被读写或操作。

属性（Property）不会确定存储位置。相反，它们具有可读写或计算它们值的 访问器（**accessors**）。

例如，有一个名为 Student 的类，带有 age、name 和 code 的私有域。我们不能在类的范围以外直接访问这些域，但是我们可以拥有访问这些私有域的属性。

一、访问器（Accessors）

属性（Property）的访问器（**accessor**）包含有助于获取（读取或计算）或设置（写入）属性的可执行语句。访问器（accessor）声明可包含一个 get 访问器、一个 set 访问器，或者同时包含二者。例如:

```
1.        // 声明类型为 string 的 Code 属性
2.        public string Code
3.        {
4.           get
5.           {
6.              return code;
7.           }
8.           set
9.           {
10.             code = value;
11.          }
12.       }
13.
14.       // 声明类型为 string 的 Name 属性
15.       public string Name
16.       {
17.          get
18.          {
19.            return name;
20.          }
21.          set
```

```
      {
         name = value;
      }
   }

   // 声明类型为 int 的 Age 属性
   public int Age
   {
      get
      {
         return age;
      }
      set
      {
         age = value;
      }
   }
```

实例

下面的实例演示了属性（Property）的用法：

```
using System;
namespace tutorialspoint
{
   class Student
   {

      private string code = "N.A";
      private string name = "not known";
      private int age = 0;

      // 声明类型为 string 的 Code 属性
      public string Code
      {
         get
         {
            return code;
         }
         set
```

```
19.                 {
20.                     code = value;
21.                 }
22.             }
23.
24.             // 声明类型为 string 的 Name 属性
25.             public string Name
26.             {
27.                 get
28.                 {
29.                     return name;
30.                 }
31.                 set
32.                 {
33.                     name = value;
34.                 }
35.             }
36.
37.             // 声明类型为 int 的 Age 属性
38.             public int Age
39.             {
40.                 get
41.                 {
42.                     return age;
43.                 }
44.                 set
45.                 {
46.                     age = value;
47.                 }
48.             }
49.             public override string ToString()
50.             {
51.                 return "Code = " + Code +", Name = " + Name + ", Age = " + Age;
52.             }
53.         }
54.         class ExampleDemo
55.         {
56.             public static void Main()
57.             {
58.                 // 创建一个新的 Student 对象
59.                 Student s = new Student();
60.
```

```
                // 设置 student 的 code、name 和 age
                s.Code = "001";
                s.Name = "Zara";
                s.Age = 9;
                Console.WriteLine("Student Info: {0}", s);
                // 增加年龄
                s.Age += 1;
                Console.WriteLine("Student Info: {0}", s);
                Console.ReadKey();
             }
          }
       }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
       Student Info: Code = 001, Name = Zara, Age = 9
       Student Info: Code = 001, Name = Zara, Age = 10
```

二、抽象属性（Abstract Properties）

抽象类可拥有抽象属性，这些属性应在派生类中被实现。下面的程序说明了这点：

```
       using System;
       namespace tutorialspoint
       {
          public abstract class Person
          {
             public abstract string Name
             {
                get;
                set;
             }
             public abstract int Age
             {
                get;
                set;
             }
          }
          class Student : Person
          {
```

```
19.
20.             private string code = "N.A";
21.             private string name = "N.A";
22.             private int age = 0;
23.
24.             // 声明类型为 string 的 Code 属性
25.             public string Code
26.             {
27.                get
28.                {
29.                   return code;
30.                }
31.                set
32.                {
33.                   code = value;
34.                }
35.             }
36.
37.             // 声明类型为 string 的 Name 属性
38.             public override string Name
39.             {
40.                get
41.                {
42.                   return name;
43.                }
44.                set
45.                {
46.                   name = value;
47.                }
48.             }
49.
50.             // 声明类型为 int 的 Age 属性
51.             public override int Age
52.             {
53.                get
54.                {
55.                   return age;
56.                }
57.                set
58.                {
59.                   age = value;
60.                }
```

```
            }
            public override string ToString()
            {
               return "Code = " + Code +", Name = " + Name + ", Age = " + Age;
            }
         }
         class ExampleDemo
         {
            public static void Main()
            {
               // 创建一个新的 Student 对象
               Student s = new Student();

               // 设置 student 的 code、name 和 age
               s.Code = "001";
               s.Name = "Zara";
               s.Age = 9;
               Console.WriteLine("Student Info:- {0}", s);
               // 增加年龄
               s.Age += 1;
               Console.WriteLine("Student Info:- {0}", s);
               Console.ReadKey();
             }
         }
      }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
      Student Info: Code = 001, Name = Zara, Age = 9
      Student Info: Code = 001, Name = Zara, Age = 10
```

C# 索引器

索引器（**Indexer**） 允许一个对象可以像数组一样被索引。当您为类定义一个索引器时，该类的行为就会像一个 虚拟数组（**virtual array**） 一样。您可以使用数组访问运算符（[ ]）来访问该类的实例。

一、语法

一维索引器的语法如下：

```
element-type this[int index]
{
   // get 访问器
   get
   {
      // 返回 index 指定的值
   }

   // set 访问器
   set
   {
      // 设置 index 指定的值
   }
}
```

二、索引器（Indexer）的用途

索引器的行为的声明在某种程度上类似于属性（ `property` ）。就像属性（ `property` ），您可使用 **get** 和 **set** 访问器来定义索引器。但是，属性返回或设置一个特定的数据成员，而索引器返回或设置对象实例的一个特定值。换句话说，它把实例数据分为更小的部分，并索引每个部分，获取或设置每个部分。

定义一个属性（ `property` ）包括提供属性名称。索引器定义的时候不带有名称，但带有 `this` 关键字，它指向对象实例。下面的实例演示了这个概念：

```
using System;
namespace IndexerApplication
```

```
3.      {
4.         class IndexedNames
5.         {
6.            private string[] namelist = new string[size];
7.            static public int size = 10;
8.            public IndexedNames()
9.            {
10.              for (int i = 0; i < size; i++)
11.              namelist[i] = "N. A.";
12.           }
13.           public string this[int index]
14.           {
15.              get
16.              {
17.                 string tmp;
18.
19.                 if( index >= 0 && index <= size-1 )
20.                 {
21.                    tmp = namelist[index];
22.                 }
23.                 else
24.                 {
25.                    tmp = "";
26.                 }
27.
28.                 return ( tmp );
29.              }
30.              set
31.              {
32.                 if( index >= 0 && index <= size-1 )
33.                 {
34.                    namelist[index] = value;
35.                 }
36.              }
37.           }
38.
39.           static void Main(string[] args)
40.           {
41.              IndexedNames names = new IndexedNames();
42.              names[0] = "Zara";
43.              names[1] = "Riz";
44.              names[2] = "Nuha";
```

```
            names[3] = "Asif";
            names[4] = "Davinder";
            names[5] = "Sunil";
            names[6] = "Rubic";
            for ( int i = 0; i < IndexedNames.size; i++ )
            {
               Console.WriteLine(names[i]);
            }
            Console.ReadKey();
         }
      }
   }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
   Zara
   Riz
   Nuha
   Asif
   Davinder
   Sunil
   Rubic
   N. A.
   N. A.
   N. A.
```

三、重载索引器（Indexer）

索引器（ `Indexer` ）可被重载。索引器声明的时候也可带有多个参数，且每个参数可以是不同的类型。没有必要让索引器必须是整型的。C# 允许索引器可以是其他类型，例如，字符串类型。

下面的实例演示了重载索引器：

```
   using System;
   namespace IndexerApplication
   {
      class IndexedNames
      {
         private string[] namelist = new string[size];
         static public int size = 10;
```

```
8.              public IndexedNames()
9.              {
10.                for (int i = 0; i < size; i++)
11.                {
12.                 namelist[i] = "N. A.";
13.                }
14.             }
15.             public string this[int index]
16.             {
17.                get
18.                {
19.                   string tmp;
20.
21.                   if( index >= 0 && index <= size-1 )
22.                   {
23.                      tmp = namelist[index];
24.                   }
25.                   else
26.                   {
27.                      tmp = "";
28.                   }
29.
30.                   return ( tmp );
31.                }
32.                set
33.                {
34.                   if( index >= 0 && index <= size-1 )
35.                   {
36.                      namelist[index] = value;
37.                   }
38.                }
39.             }
40.             public int this[string name]
41.             {
42.                get
43.                {
44.                   int index = 0;
45.                   while(index < size)
46.                   {
47.                      if (namelist[index] == name)
48.                      {
49.                       return index;
```

```
                    }
                    index++;
                }
                return index;
            }

        }

        static void Main(string[] args)
        {
            IndexedNames names = new IndexedNames();
            names[0] = "Zara";
            names[1] = "Riz";
            names[2] = "Nuha";
            names[3] = "Asif";
            names[4] = "Davinder";
            names[5] = "Sunil";
            names[6] = "Rubic";
            // 使用带有 int 参数的第一个索引器
            for (int i = 0; i < IndexedNames.size; i++)
            {
                Console.WriteLine(names[i]);
            }
            // 使用带有 string 参数的第二个索引器
            Console.WriteLine(names["Nuha"]);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Zara
Riz
Nuha
Asif
Davinder
Sunil
Rubic
N. A.
N. A.
N. A.
```

```
11.         2
```

C# 委托

C# 中的委托（Delegate）类似于 C 或 C++ 中函数的指针。委托（**Delegate**） 是存有对某个方法的引用的一种引用类型变量。引用可在运行时被改变。

委托（Delegate）特别用于实现事件和回调方法。所有的委托（Delegate）都派生自 **System.Delegate** 类。

一、声明委托（Delegate）

委托声明决定了可由该委托引用的方法。委托可指向一个与其具有相同标签的方法。

例如，假设有一个委托：

```
1.        public delegate int MyDelegate (string s);
```

上面的委托可被用于引用任何一个带有一个单一的 string 参数的方法，并返回一个 int 类型变量。

声明委托的语法如下：

```
1.        delegate <return type> <delegate-name> <parameter list>
```

二、实例化委托（Delegate）

一旦声明了委托类型，委托对象必须使用 **new** 关键字来创建，且与一个特定的方法有关。当创建委托时，传递到 **new** 语句的参数就像方法调用一样书写，但是不带有参数。例如：

```
1.        public delegate void printString(string s);
2.        ...
3.        printString ps1 = new printString(WriteToScreen);
4.        printString ps2 = new printString(WriteToFile);
```

下面的实例演示了委托的声明、实例化和使用，该委托可用于引用带有一个整型参数的方法，并返回一个整型值。

```
1.        using System;
```

```
2.
3.      delegate int NumberChanger(int n);
4.      namespace DelegateAppl
5.      {
6.         class TestDelegate
7.         {
8.            static int num = 10;
9.            public static int AddNum(int p)
10.           {
11.              num += p;
12.              return num;
13.           }
14.
15.           public static int MultNum(int q)
16.           {
17.              num *= q;
18.              return num;
19.           }
20.           public static int getNum()
21.           {
22.              return num;
23.           }
24.
25.           static void Main(string[] args)
26.           {
27.              // 创建委托实例
28.              NumberChanger nc1 = new NumberChanger(AddNum);
29.              NumberChanger nc2 = new NumberChanger(MultNum);
30.              // 使用委托对象调用方法
31.              nc1(25);
32.              Console.WriteLine("Value of Num: {0}", getNum());
33.              nc2(5);
34.              Console.WriteLine("Value of Num: {0}", getNum());
35.              Console.ReadKey();
36.           }
37.        }
38.     }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.      Value of Num: 35
2.      Value of Num: 175
```

三、委托的多播（Multicasting of a Delegate）

委托对象可使用 "+" 运算符进行合并。一个合并委托调用它所合并的两个委托。只有相同类型的委托可被合并。"-" 运算符可用于从合并的委托中移除组件委托。

使用委托的这个有用的特点，您可以创建一个委托被调用时要调用的方法的调用列表。这被称为委托的多播（**multicasting**），也叫组播。下面的程序演示了委托的多播：

```
1.        using System;
2.
3.        delegate int NumberChanger(int n);
4.        namespace DelegateAppl
5.        {
6.           class TestDelegate
7.           {
8.              static int num = 10;
9.              public static int AddNum(int p)
10.             {
11.                num += p;
12.                return num;
13.             }
14.
15.             public static int MultNum(int q)
16.             {
17.                num *= q;
18.                return num;
19.             }
20.             public static int getNum()
21.             {
22.                return num;
23.             }
24.
25.             static void Main(string[] args)
26.             {
27.                // 创建委托实例
28.                NumberChanger nc;
29.                NumberChanger nc1 = new NumberChanger(AddNum);
30.                NumberChanger nc2 = new NumberChanger(MultNum);
31.                nc = nc1;
32.                nc += nc2;
```

```
            // 调用多播
            nc(5);
            Console.WriteLine("Value of Num: {0}", getNum());
            Console.ReadKey();
         }
      }
   }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
   Value of Num: 75
```

四、委托（Delegate）的用途

下面的实例演示了委托的用法。委托 printString 可用于引用带有一个字符串作为输入的方法，并不返回任何东西。

我们使用这个委托来调用两个方法，第一个把字符串打印到控制台，第二个把字符串打印到文件：

```
   using System;
   using System.IO;

   namespace DelegateAppl
   {
      class PrintString
      {
         static FileStream fs;
         static StreamWriter sw;
         // 委托声明
         public delegate void printString(string s);

         // 该方法打印到控制台
         public static void WriteToScreen(string str)
         {
            Console.WriteLine("The String is: {0}", str);
         }
         // 该方法打印到文件
         public static void WriteToFile(string s)
         {
            fs = new FileStream("c:\\message.txt",
```

```
                FileMode.Append, FileAccess.Write);
                sw = new StreamWriter(fs);
                sw.WriteLine(s);
                sw.Flush();
                sw.Close();
                fs.Close();
            }
            // 该方法把委托作为参数，并使用它调用方法
            public static void sendString(printString ps)
            {
                ps("Hello World");
            }
            static void Main(string[] args)
            {
                printString ps1 = new printString(WriteToScreen);
                printString ps2 = new printString(WriteToFile);
                sendString(ps1);
                sendString(ps2);
                Console.ReadKey();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
The String is: Hello World
```

C# 事件

事件（**Event**） 基本上说是一个用户操作，如按键、点击、鼠标移动等等，或者是一些出现，如系统生成的通知。应用程序需要在事件发生时响应事件。例如，中断。事件是用于进程间通信。

一、通过事件使用委托

事件在类中声明且生成，且通过使用同一个类或其他类中的委托与事件处理程序关联。包含事件的类用于发布事件。这被称为 发布器（**publisher**） 类。其他接受该事件的类被称为 订阅器（**subscriber**） 类。事件使用 发布-订阅（**publisher-subscriber**） 模型。发布器（**publisher**） 是一个包含事件和委托定义的对象。事件和委托之间的联系也定义在这个对象中。发布器（publisher）类的对象调用这个事件，并通知其他的对象。

订阅器（**subscriber**） 是一个接受事件并提供事件处理程序的对象。在发布器（publisher）类中的委托调用订阅器（subscriber）类中的方法（事件处理程序）。

二、声明事件（Event）

在类的内部声明事件，首先必须声明该事件的委托类型。例如：

```
1.      public delegate void BoilerLogHandler(string status);
```

然后，声明事件本身，使用 **event** 关键字：

```
1.      // 基于上面的委托定义事件
2.      public event BoilerLogHandler BoilerEventLog;
```

上面的代码定义了一个名为 BoilerLogHandler 的委托和一个名为 BoilerEventLog 的事件，该事件在生成的时候会调用委托。

实例 1

```
1.        using System;
2.        namespace SimpleEvent
3.        {
```

```
4.          using System;
5.          /***********发布器类***********/
6.          public class EventTest
7.          {
8.            private int value;
9.
10.           public delegate void NumManipulationHandler();
11.
12.
13.           public event NumManipulationHandler ChangeNum;
14.           protected virtual void OnNumChanged()
15.           {
16.             if ( ChangeNum != null )
17.             {
18.               ChangeNum(); /* 事件被触发 */
19.             }else {
20.               Console.WriteLine( "event not fire" );
21.               Console.ReadKey(); /* 回车继续 */
22.             }
23.           }
24.
25.
26.           public EventTest()
27.           {
28.             int n = 5;
29.             SetValue( n );
30.           }
31.
32.
33.           public void SetValue( int n )
34.           {
35.             if ( value != n )
36.             {
37.               value = n;
38.               OnNumChanged();
39.             }
40.           }
41.         }
42.
43.
44.         /***********订阅器类***********/
45.
```

```
        public class subscribEvent
        {
          public void printf()
          {
            Console.WriteLine( "event fire" );
            Console.ReadKey(); /* 回车继续 */
          }
        }

        /***********触发***********/
        public class MainClass
        {
          public static void Main()
          {
            EventTest e = new EventTest(); /* 实例化对象,第一次没有触发事件 */
            subscribEvent v = new subscribEvent(); /* 实例化对象 */
            e.ChangeNum += new EventTest.NumManipulationHandler( v.printf ); /*
注册 */
            e.SetValue( 7 );
            e.SetValue( 11 );
          }
        }
      }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
      event not fire
      event fire
      event fire
```

实例 2

本实例提供一个简单的用于热水锅炉系统故障排除的应用程序。当维修工程师检查锅炉时，锅炉的温度和压力会随着维修工程师的备注自动记录到日志文件中。

```
      using System;
      using System.IO;

      namespace BoilerEventAppl
      {
```

```
6.
7.          // boiler 类
8.          class Boiler
9.          {
10.            private int temp;
11.            private int pressure;
12.            public Boiler(int t, int p)
13.            {
14.               temp = t;
15.               pressure = p;
16.            }
17.
18.            public int getTemp()
19.            {
20.               return temp;
21.            }
22.            public int getPressure()
23.            {
24.               return pressure;
25.            }
26.         }
27.         // 事件发布器
28.         class DelegateBoilerEvent
29.         {
30.            public delegate void BoilerLogHandler(string status);
31.
32.            // 基于上面的委托定义事件
33.            public event BoilerLogHandler BoilerEventLog;
34.
35.            public void LogProcess()
36.            {
37.               string remarks = "O. K";
38.               Boiler b = new Boiler(100, 12);
39.               int t = b.getTemp();
40.               int p = b.getPressure();
41.               if(t > 150 || t < 80 || p < 12 || p > 15)
42.               {
43.                  remarks = "Need Maintenance";
44.               }
45.               OnBoilerEventLog("Logging Info:\n");
46.               OnBoilerEventLog("Temparature " + t + "\nPressure: " + p);
47.               OnBoilerEventLog("\nMessage: " + remarks);
```

```
48.             }
49.
50.             protected void OnBoilerEventLog(string message)
51.             {
52.                if (BoilerEventLog != null)
53.                {
54.                   BoilerEventLog(message);
55.                }
56.             }
57.          }
58.          // 该类保留写入日志文件的条款
59.          class BoilerInfoLogger
60.          {
61.             FileStream fs;
62.             StreamWriter sw;
63.             public BoilerInfoLogger(string filename)
64.             {
                   fs = new FileStream(filename, FileMode.Append,
65. FileAccess.Write);
66.                sw = new StreamWriter(fs);
67.             }
68.             public void Logger(string info)
69.             {
70.                sw.WriteLine(info);
71.             }
72.             public void Close()
73.             {
74.                sw.Close();
75.                fs.Close();
76.             }
77.          }
78.          // 事件订阅器
79.          public class RecordBoilerInfo
80.          {
81.             static void Logger(string info)
82.             {
83.                Console.WriteLine(info);
84.             }//end of Logger
85.
86.             static void Main(string[] args)
87.             {
                   BoilerInfoLogger filelog = new
88. BoilerInfoLogger("e:\\boiler.txt");
```

```
            DelegateBoilerEvent boilerEvent = new DelegateBoilerEvent();
            boilerEvent.BoilerEventLog += new
            DelegateBoilerEvent.BoilerLogHandler(Logger);
            boilerEvent.BoilerEventLog += new
            DelegateBoilerEvent.BoilerLogHandler(filelog.Logger);
            boilerEvent.LogProcess();
            Console.ReadLine();
            filelog.Close();
        }//end of main

    }//end of RecordBoilerInfo
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    Logging info:

    Temperature 100
    Pressure 12

    Message: O. K
```

C# 泛型（Generic）

泛型（**Generic**） 允许您延迟编写类或方法中的编程元素的数据类型的规范，直到实际在程序中使用它的时候。换句话说，泛型允许您编写一个可以与任何数据类型一起工作的类或方法。

您可以通过数据类型的替代参数编写类或方法的规范。当编译器遇到类的构造函数或方法的函数调用时，它会生成代码来处理指定的数据类型。下面这个简单的实例将有助于您理解这个概念：

```
1.      using System;
2.      using System.Collections.Generic;
3.
4.      namespace GenericApplication
5.      {
6.          public class MyGenericArray<T>
7.          {
8.              private T[] array;
9.              public MyGenericArray(int size)
10.             {
11.                 array = new T[size + 1];
12.             }
13.             public T getItem(int index)
14.             {
15.                 return array[index];
16.             }
17.             public void setItem(int index, T value)
18.             {
19.                 array[index] = value;
20.             }
21.         }
22.
23.         class Tester
24.         {
25.             static void Main(string[] args)
26.             {
27.                 // 声明一个整型数组
28.                 MyGenericArray<int> intArray = new MyGenericArray<int>(5);
29.                 // 设置值
30.                 for (int c = 0; c < 5; c++)
31.                 {
32.                     intArray.setItem(c, c*5);
33.                 }
```

```
                // 获取值
                for (int c = 0; c < 5; c++)
                {
                    Console.Write(intArray.getItem(c) + " ");
                }
                Console.WriteLine();
                // 声明一个字符数组
                MyGenericArray<char> charArray = new MyGenericArray<char>(5);
                // 设置值
                for (int c = 0; c < 5; c++)
                {
                    charArray.setItem(c, (char)(c+97));
                }
                // 获取值
                for (int c = 0; c < 5; c++)
                {
                    Console.Write(charArray.getItem(c) + " ");
                }
                Console.WriteLine();
                Console.ReadKey();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    0 5 10 15 20
    a b c d e
```

一、泛型（Generic）的特性

使用泛型是一种增强程序功能的技术，具体表现在以下几个方面：

- 它有助于您最大限度地重用代码、保护类型的安全以及提高性能。
- 您可以创建泛型集合类。.NET 框架类库在 System.Collections.Generic 命名空间中包含了一些新的泛型集合类。您可以使用这些泛型集合类来替代 System.Collections 中的集合类。
- 您可以创建自己的泛型接口、泛型类、泛型方法、泛型事件和泛型委托。

- 您可以对泛型类进行约束以访问特定数据类型的方法。
- 关于泛型数据类型中使用的类型的信息可在运行时通过使用反射获取。

二、泛型（Generic）方法

在上面的实例中，我们已经使用了泛型类，我们可以通过类型参数声明泛型方法。下面的程序说明了这个概念：

```
1.      using System;
2.      using System.Collections.Generic;
3.
4.      namespace GenericMethodAppl
5.      {
6.          class Program
7.          {
8.              static void Swap<T>(ref T lhs, ref T rhs)
9.              {
10.                 T temp;
11.                 temp = lhs;
12.                 lhs = rhs;
13.                 rhs = temp;
14.             }
15.             static void Main(string[] args)
16.             {
17.                 int a, b;
18.                 char c, d;
19.                 a = 10;
20.                 b = 20;
21.                 c = 'I';
22.                 d = 'V';
23.
24.                 // 在交换之前显示值
25.                 Console.WriteLine("Int values before calling swap:");
26.                 Console.WriteLine("a = {0}, b = {1}", a, b);
27.                 Console.WriteLine("Char values before calling swap:");
28.                 Console.WriteLine("c = {0}, d = {1}", c, d);
29.
30.                 // 调用 swap
31.                 Swap<int>(ref a, ref b);
32.                 Swap<char>(ref c, ref d);
```

```
33.
34.                 // 在交换之后显示值
35.                 Console.WriteLine("Int values after calling swap:");
36.                 Console.WriteLine("a = {0}, b = {1}", a, b);
37.                 Console.WriteLine("Char values after calling swap:");
38.                 Console.WriteLine("c = {0}, d = {1}", c, d);
39.                 Console.ReadKey();
40.             }
41.         }
42.     }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.     Int values before calling swap:
2.     a = 10, b = 20
3.     Char values before calling swap:
4.     c = I, d = V
5.     Int values after calling swap:
6.     a = 20, b = 10
7.     Char values after calling swap:
8.     c = V, d = I
```

三、泛型（Generic）委托

您可以通过类型参数定义泛型委托。例如：

```
1.     delegate T NumberChanger<T>(T n);
```

下面的实例演示了委托的使用：

```
1.     using System;
2.     using System.Collections.Generic;
3.
4.     delegate T NumberChanger<T>(T n);
5.     namespace GenericDelegateAppl
6.     {
7.         class TestDelegate
8.         {
9.             static int num = 10;
10.            public static int AddNum(int p)
```

```
            {
                num += p;
                return num;
            }

            public static int MultNum(int q)
            {
                num *= q;
                return num;
            }
            public static int getNum()
            {
                return num;
            }

            static void Main(string[] args)
            {
                // 创建委托实例
                NumberChanger<int> nc1 = new NumberChanger<int>(AddNum);
                NumberChanger<int> nc2 = new NumberChanger<int>(MultNum);
                // 使用委托对象调用方法
                nc1(25);
                Console.WriteLine("Value of Num: {0}", getNum());
                nc2(5);
                Console.WriteLine("Value of Num: {0}", getNum());
                Console.ReadKey();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    Value of Num: 35
    Value of Num: 175
```

C# 集合

集合（Collection）类是专门用于数据存储和检索的类。这些类提供了对栈（stack）、队列（queue）、列表（list）和哈希表（hash table）的支持。大多数集合类实现了相同的接口。集合（Collection）类服务于不同的目的，如为元素动态分配内存，基于索引访问列表项等等。这些类创建 Object 类的对象的集合。在 C# 中，Object 类是所有数据类型的基类。

一、各种集合类和它们的用法

下面是各种常用的 System.Collection 命名空间的类。点击下面的链接查看细节。

类	描述和用法
动态数组（ArrayList）	它代表了可被单独索引的对象的有序集合。它基本上可以替代一个数组。但是，与数组不同的是，您可以使用索引在指定的位置添加和移除项目，动态数组会自动重新调整它的大小。它也允许在列表中进行动态内存分配、增加、搜索、排序各项。
哈希表（Hashtable）	它使用键来访问集合中的元素。当您使用键访问元素时，则使用哈希表，而且您可以识别一个有用的键值。哈希表中的每一项都有一个键/值对。键用于访问集合中的项目。
排序列表（SortedList）	它可以使用键和索引来访问列表中的项。排序列表是数组和哈希表的组合。它包含一个可使用键或索引访问各项的列表。如果您使用索引访问各项，则它是一个动态数组（ArrayList），如果您使用键访问各项，则它是一个哈希表（Hashtable）。集合中的各项总是按键值排序。
堆栈（Stack）	它代表了一个后进先出的对象集合。当您需要对各项进行后进先出的访问时，则使用堆栈。当您在列表中添加一项，称为推入元素，当您从列表中移除一项时，称为弹出元素。
队列（Queue）	它代表了一个先进先出的对象集合。当您需要对各项进行先进先出的访问时，则使用队列。当您在列表中添加一项，称为入队，当您从列表中移除一项时，称为出队。
点阵列（BitArray）	它代表了一个使用值 1 和 0 来表示的二进制数组。当您需要存储位，但是事先不知道位数时，则使用点阵列。您可以使用整型索引从点阵列集合中访问各项，索引从零开始。

C# 匿名方法

我们已经提到过，委托是用于引用与其具有相同标签的方法。换句话说，您可以使用委托对象调用可由委托引用的方法。

匿名方法（**Anonymous methods**） 提供了一种传递代码块作为委托参数的技术。匿名方法是没有名称只有主体的方法。

在匿名方法中您不需要指定返回类型，它是从方法主体内的 `return` 语句推断的。

一、编写匿名方法的语法

匿名方法是通过使用 **delegate** 关键字创建委托实例来声明的。例如：

```
1.      delegate void NumberChanger(int n);
2.      ...
3.      NumberChanger nc = delegate(int x)
4.      {
5.          Console.WriteLine("Anonymous Method: {0}", x);
6.      };
```

代码块 `Console.WriteLine("Anonymous Method: {0}", x);` 是匿名方法的主体。委托可以通过匿名方法调用，也可以通过命名方法调用，即，通过向委托对象传递方法参数。例如：

```
1.      nc(10);
```

实例

下面的实例演示了匿名方法的概念：

```
1.      using System;
2.
3.      delegate void NumberChanger(int n);
4.      namespace DelegateAppl
5.      {
6.          class TestDelegate
7.          {
```

```
8.              static int num = 10;
9.              public static void AddNum(int p)
10.             {
11.                 num += p;
12.                 Console.WriteLine("Named Method: {0}", num);
13.             }
14.
15.             public static void MultNum(int q)
16.             {
17.                 num *= q;
18.                 Console.WriteLine("Named Method: {0}", num);
19.             }
20.             public static int getNum()
21.             {
22.                 return num;
23.             }
24.
25.             static void Main(string[] args)
26.             {
27.                 // 使用匿名方法创建委托实例
28.                 NumberChanger nc = delegate(int x)
29.                 {
30.                    Console.WriteLine("Anonymous Method: {0}", x);
31.                 };
32.
33.                 // 使用匿名方法调用委托
34.                 nc(10);
35.
36.                 // 使用命名方法实例化委托
37.                 nc =  new NumberChanger(AddNum);
38.
39.                 // 使用命名方法调用委托
40.                 nc(5);
41.
42.                 // 使用另一个命名方法实例化委托
43.                 nc =  new NumberChanger(MultNum);
44.
45.                 // 使用命名方法调用委托
46.                 nc(2);
47.                 Console.ReadKey();
48.             }
49.         }
```

```
50.     }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.      Anonymous Method: 10
2.      Named Method: 15
3.      Named Method: 30
```

Lambda 表达式

Lambda 表达式是一种可用于创建 委托 或 表达式目录树 类型的 匿名函数 。

如下面的示例所示，你可以将此表达式分配给委托类型：

```
1.      delegate int del(int i);
2.      static void Main(string[] args)
3.      {
4.          del myDelegate = x => x * x;
5.          int j = myDelegate(5); //j = 25
6.      }
```

如下面的示例所示，你可以将此表达式分配给委托类型：

```
1.      using System.Linq.Expressions;
2.
3.      namespace ConsoleApplication1
4.      {
5.          class Program
6.          {
7.              static void Main(string[] args)
8.              {
9.                  Expression<del> myET = x => x * x;
10.             }
11.         }
12.     }
```

在 `is` 或 `as` 运算符的左侧不允许使用 Lambda。

表达式 lambda

表达式位于 => 运算符右侧的 Lambda 表达式称为“表达式 lambda”。 表达式 lambda 广泛用于表达式树的构造。 表达式 lambda 会返回表达式的结果，并采用以下基本形式：

```
1.      (input-parameters) => expression
```

仅当 lambda 只有一个输入参数时，括号才是可选的；否则括号是必需的。 括号内的两个或更多输入参数使用逗号加以分隔：

```
(x, y) => x == y
```

有时，编译器难以或无法推断输入类型。 如果出现这种情况，你可以按以下示例中所示方式显式指定类型：

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

使用空括号指定零个输入参数：

```
() => SomeMethod()
```

语句 lambda

语句 lambda 与表达式 lambda 表达式类似，只是语句括在大括号中：

```
(input-parameters) => { statement; }
```

语句 lambda 的主体可以包含任意数量的语句；但是，实际上通常不会多于两个或三个。

```
delegate void TestDelegate(string s);
```

```
TestDelegate del = n => { string s = n + " World";
                          Console.WriteLine(s); };
```

C# 特性

特性（**Attribute**）是用于在运行时传递程序中各种元素（比如类、方法、结构、枚举、组件等）的行为信息的声明性标签。您可以通过使用特性向程序添加声明性信息。一个声明性标签是通过放置在它所应用的元素前面的方括号（[ ]）来描述的。

特性（Attribute）用于添加元数据，如编译器指令和注释、描述、方法、类等其他信息。.Net 框架提供了两种类型的特性：预定义特性和自定义特性。

一、规定特性（Attribute）

规定特性（Attribute）的语法如下：

```
1.      [attribute(positional_parameters, name_parameter = value, ...)]
2.      element
```

特性（Attribute）的名称和值是在方括号内规定的，放置在它所应用的元素之前。positional_parameters 规定必需的信息，name_parameter 规定可选的信息。

二、预定义特性（Attribute）

.Net 框架提供了三种预定义特性：

```
1.      • AttributeUsage
2.      • Conditional
3.      • Obsolete
```

2.1 AttributeUsage

预定义特性 **AttributeUsage** 描述了如何使用一个自定义特性类。它规定了特性可应用到的项目的类型。

规定该特性的语法如下：

```
1.      [AttributeUsage(
2.         validon,
3.         AllowMultiple=allowmultiple,
```

```
4.         Inherited=inherited
5.     )]
```

其中：

- 参数 `validon` 规定特性可被放置的语言元素（规定当前特性适用的范围，比如适用于类、构造函数、字段、方法、属性）。它是枚举器 `AttributeTargets` 的值的组合。默认值是 `AttributeTargets.All` 。
- 参数 `allowmultiple` （可选的）为该特性的 `AllowMultiple` 属性（ `property` ）提供一个布尔值。如果为 `true` ，则该特性是多用的（在同一个类上多次调用）。默认值是 `false` （单用的）。
- 参数 `inherited` （可选的）为该特性的 `Inherited` 属性（ `property` ）提供一个布尔值。如果为 `true` ，则该特性可被派生类继承（当此特性的类被继承以后，该特性也被继承）。默认值是 `false` （不被继承）。

例如：

```
1.     [AttributeUsage(AttributeTargets.Class |
2.     AttributeTargets.Constructor |
3.     AttributeTargets.Field |
4.     AttributeTargets.Method |
5.     AttributeTargets.Property,
6.     AllowMultiple = true),
7.     Inherited = false]
```

2.2 Conditional

这个预定义特性标记了一个条件方法，其执行依赖于指定的预处理标识符。

它会引起方法调用的条件编译，取决于指定的值，比如 **Debug** 或 **Trace**。例如，当调试代码时显示变量的值。

规定该特性的语法如下：

```
1.     [Conditional(
2.         conditionalSymbol
3.     )]
```

例如：

```
[Conditional("DEBUG")]
```

下面的实例演示了该特性：

```
#define DEBUG
using System;
using System.Diagnostics;
public class Myclass
{
    [Conditional("DEBUG")]
    public static void Message(string msg)
    {
        Console.WriteLine(msg);
    }
}
class Test
{
    static void function1()
    {
        Myclass.Message("In Function 1.");
        function2();
    }
    static void function2()
    {
        Myclass.Message("In Function 2.");
    }
    public static void Main()
    {
        Myclass.Message("In Main function.");
        function1();
        Console.ReadKey();
    }
}
```

当上面的代码被编译和执行时，在Debug模式下它会产生下列结果，在Release模式下则不会有任何结果：

```
In Main function
In Function 1
In Function 2
```

此时，可以通过预处理指令在Release模式下也可以实现DEBUG的输出：

```
#define DEBUG
```

2.3 Obsolete

这个预定义特性标记了不应被使用的程序实体。它可以让您通知编译器丢弃某个特定的目标元素。例如，当一个新方法被用在一个类中，但是您仍然想要保持类中的旧方法，您可以通过显示一个应该使用新方法，而不是旧方法的消息，来把它标记为 `obsolete` （过时的）。

规定该特性的语法如下：

```
[Obsolete( message )]
[Obsolete( message, iserror )]
```

其中：

- 参数 `message` ，是一个字符串，描述项目为什么过时的原因以及该替代使用什么。
- 参数 `iserror` ，是一个布尔值。如果该值为 `true` ，编译器应把该项目的使用当作一个错误。默认值是 `false` （编译器生成一个警告）。

下面的实例演示了该特性：

```
using System;
public class MyClass
{
   [Obsolete("Don't use OldMethod, use NewMethod instead", true)]
   static void OldMethod()
   {
      Console.WriteLine("It is the old method");
   }
   static void NewMethod()
   {
      Console.WriteLine("It is the new method");
   }
   public static void Main()
   {
      OldMethod();
   }
```

```
    }
```

当您尝试编译该程序时，编译器会给出一个错误消息说明：

```
Don't use OldMethod, use NewMethod instead
```

三、创建自定义特性（Attribute）

.Net 框架允许创建自定义特性，用于存储声明性的信息，且可在运行时被检索。该信息根据设计标准和应用程序需要，可与任何目标元素相关。

创建并使用自定义特性包含四个步骤：

- 声明自定义特性
- 构建自定义特性
- 在目标程序元素上应用自定义特性
- 通过反射访问特性

最后一个步骤包含编写一个简单的程序来读取元数据以便查找各种符号。元数据是用于描述其他数据的数据和信息。该程序应使用反射来在运行时访问特性。我们将在下一章详细讨论这点。

3.1 声明自定义特性

一个新的自定义特性应派生自 **System.Attribute** 类。例如：

```
// 一个自定义特性 BugFix 被赋给类及其成员
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]

public class DeBugInfo : System.Attribute
```

在上面的代码中，我们已经声明了一个名为 DeBugInfo 的自定义特性。

3.2 构建自定义特性

让我们构建一个名为 `DeBugInfo` 的自定义特性，该特性将存储调试程序获得的信息。它存储下面的信息：

```
    • bug 的代码编号
    • 辨认该 bug 的开发人员名字
    • 最后一次审查该代码的日期
    • 一个存储了开发人员标记的字符串消息
```

我们的 `DeBugInfo` 类将带有三个用于存储前三个信息的私有属性（ `property` ）和一个用于存储消息的公有属性（ `property` ）。所以 `bug` 编号、开发人员名字和审查日期将是 `DeBugInfo` 类的必需的定位（ `positional` ）参数，消息将是一个可选的命名（ `named` ）参数。

每个特性必须至少有一个构造函数。必需的定位（ `positional` ）参数应通过构造函数传递。下面的代码演示了 `DeBugInfo` 类：

```
// 一个自定义特性 BugFix 被赋给类及其成员
[AttributeUsage(AttributeTargets.Class |
AttributeTargets.Constructor |
AttributeTargets.Field |
AttributeTargets.Method |
AttributeTargets.Property,
AllowMultiple = true)]

public class DeBugInfo : System.Attribute
{
  private int bugNo;
  private string developer;
  private string lastReview;
  private string message;

  public DeBugInfo(int bg, string dev, string d)
  {
      this.bugNo = bg;
      this.developer = dev;
      this.lastReview = d;
  }

  public int BugNo
  {
```

```
            get
            {
                return bugNo;
            }
        }
        public string Developer
        {
            get
            {
                return developer;
            }
        }
        public string LastReview
        {
            get
            {
                return lastReview;
            }
        }
        public string Message
        {
            get
            {
                return message;
            }
            set
            {
                message = value;
            }
        }
    }
```

3.3 应用自定义特性

通过把特性放置在紧接着它的目标之前，来应用该特性：

```
    [DeBugInfo(45, "Zara Ali", "12/8/2012", Message = "Return type mismatch")]
    [DeBugInfo(49, "Nuha Ali", "10/10/2012", Message = "Unused variable")]
    class Rectangle
    {
```

```
        // 成员变量
        protected double length;
        protected double width;
        public Rectangle(double l, double w)
        {
            length = l;
            width = w;
        }
        [DeBugInfo(55, "Zara Ali", "19/10/2012",
        Message = "Return type mismatch")]
        public double GetArea()
        {
            return length * width;
        }
        [DeBugInfo(56, "Zara Ali", "19/10/2012")]
        public void Display()
        {
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }
```

在下一章中，我们将使用 Reflection 类对象来检索这些信息。

C# 反射

反射指程序可以访问、检测和修改它本身状态或行为的一种能力。

程序集包含模块，而模块包含类型，类型又包含成员。反射则提供了封装程序集、模块和类型的对象。

您可以使用反射动态地创建类型的实例，将类型绑定到现有对象，或从现有对象中获取类型。然后，可以调用类型的方法或访问其字段和属性。

优缺点

优点：

1．反射提高了程序的灵活性和扩展性。 2．降低耦合性，提高自适应能力。 3．它允许程序创建和控制任何类的对象，无需提前硬编码目标类。

缺点：

1．性能问题：使用反射基本上是一种解释操作，用于字段和方法接入时要远慢于直接代码。因此反射机制主要应用在对灵活性和拓展性要求很高的系统框架上，普通程序不建议使用。

2．使用反射会模糊程序内部逻辑；程序员希望在源代码中看到程序的逻辑，反射却绕过了源代码的技术，因而会带来维护的问题，反射代码比相应的直接代码更复杂。

一、反射（Reflection）的用途

反射（Reflection）有下列用途：

1．它允许在运行时查看特性（attribute）信息。

2．它允许审查集合中的各种类型，以及实例化这些类型。

3．它允许延迟绑定的方法和属性（property）。

4．它允许在运行时创建新类型，然后使用这些类型执行一些任务。

> （1）使用 `Assembly` 定义和加载程序集，加载在程序集清单中列出模块，以及从此程序集中查找类型并创建该类型的实例。
>
> （2）使用 `Module` 了解包含模块的程序集以及模块中的类等，还可以获取在模块上定义的所有全局方法或其他特定的非全局方法。

> （3）使用 `ConstructorInfo` 了解构造函数的名称、参数、访问修饰符（如 `public` 或 `private`）和实现详细信息（如 `abstract` 或 `virtual`）等。
>
> （4）使用 `MethodInfo` 了解方法的名称、返回类型、参数、访问修饰符（如 `public` 或 `private`）和实现详细信息（如 `abstract` 或 `virtual`）等。
>
> （5）使用 `FieldInfo` 了解字段的名称、访问修饰符（如 `public` 或 `private`）和实现详细信息（如 `static`）等，并获取或设置字段值。
>
> （6）使用 `EventInfo` 了解事件的名称、事件处理程序数据类型、自定义属性、声明类型和反射类型等，添加或移除事件处理程序。
>
> （7）使用 `PropertyInfo` 了解属性的名称、数据类型、声明类型、反射类型和只读或可写状态等，获取或设置属性值。
>
> （8）使用 `ParameterInfo` 了解参数的名称、数据类型、是输入参数还是输出参数，以及参数在方法签名中的位置等。

二、查看元数据

我们已经在上面的章节中提到过，使用反射（Reflection）可以查看特性（attribute）信息。

System.Reflection 类的 **MemberInfo** 对象需要被初始化，用于发现与类相关的特性（attribute）。为了做到这点，您可以定义目标类的一个对象，如下：

```
1.      System.Reflection.MemberInfo info = typeof(MyClass);
```

下面的程序演示了这点：

```
1.        using System;
2.
3.        [AttributeUsage(AttributeTargets.All)]
4.        public class HelpAttribute : System.Attribute
5.        {
6.           public readonly string Url;
7.
8.           public string Topic  // Topic 是一个命名（named）参数
9.           {
10.             get
11.             {
12.                return topic;
13.             }
14.             set
```

```
            {
               topic = value;
            }
         }

         public HelpAttribute(string url)  // url 是一个定位（positional）参数
         {
            this.Url = url;
         }

         private string topic;
      }

      [HelpAttribute("Information on the class MyClass")]
      class MyClass
      {
      }

      namespace AttributeAppl
      {
         class Program
         {
            static void Main(string[] args)
            {
               System.Reflection.MemberInfo info = typeof(MyClass);
               object[] attributes = info.GetCustomAttributes(true);
               for (int i = 0; i < attributes.Length; i++)
               {
                  System.Console.WriteLine(attributes[i]);
               }
               Console.ReadKey();
            }
         }
      }
```

当上面的代码被编译和执行时，它会显示附加到类 MyClass 上的自定义特性：

```
      HelpAttribute
```

实例

在本实例中，我们将使用在上一章中创建的 `DeBugInfo` 特性，并使用反射（`Reflection`）来读取 `Rectangle` 类中的元数据。

```
using System;
using System.Reflection;
namespace BugFixApplication
{
   // 一个自定义特性 BugFix 被赋给类及其成员
   [AttributeUsage(AttributeTargets.Class |
   AttributeTargets.Constructor |
   AttributeTargets.Field |
   AttributeTargets.Method |
   AttributeTargets.Property,
   AllowMultiple = true)]
   public class DeBugInfo : System.Attribute
   {
      private int bugNo;
      private string developer;
      private string lastReview;
      public string message;

      public DeBugInfo(int bg, string dev, string d)
      {
         this.bugNo = bg;
         this.developer = dev;
         this.lastReview = d;
      }

      public int BugNo
      {
         get
         {
            return bugNo;
         }
      }
      public string Developer
      {
         get
         {
            return developer;
         }
      }
```

```
40.             public string LastReview
41.             {
42.                get
43.                {
44.                   return lastReview;
45.                }
46.             }
47.             public string Message
48.             {
49.                get
50.                {
51.                   return message;
52.                }
53.                set
54.                {
55.                   message = value;
56.                }
57.             }
58.          }
59.          [DeBugInfo(45, "Zara Ali", "12/8/2012",
60.           Message = "Return type mismatch")]
61.          [DeBugInfo(49, "Nuha Ali", "10/10/2012",
62.           Message = "Unused variable")]
63.          class Rectangle
64.          {
65.             // 成员变量
66.             protected double length;
67.             protected double width;
68.             public Rectangle(double l, double w)
69.             {
70.                length = l;
71.                width = w;
72.             }
73.             [DeBugInfo(55, "Zara Ali", "19/10/2012",
74.              Message = "Return type mismatch")]
75.             public double GetArea()
76.             {
77.                return length * width;
78.             }
79.             [DeBugInfo(56, "Zara Ali", "19/10/2012")]
80.             public void Display()
81.             {
```

```
            Console.WriteLine("Length: {0}", length);
            Console.WriteLine("Width: {0}", width);
            Console.WriteLine("Area: {0}", GetArea());
        }
    }//end class Rectangle

    class ExecuteRectangle
    {
        static void Main(string[] args)
        {
            Rectangle r = new Rectangle(4.5, 7.5);
            r.Display();
            Type type = typeof(Rectangle);
            // 遍历 Rectangle 类的特性
            foreach (Object attributes in type.GetCustomAttributes(false))
            {
                DeBugInfo dbi = (DeBugInfo)attributes;
                if (null != dbi)
                {
                    Console.WriteLine("Bug no: {0}", dbi.BugNo);
                    Console.WriteLine("Developer: {0}", dbi.Developer);
                    Console.WriteLine("Last Reviewed: {0}",
                                dbi.LastReview);
                    Console.WriteLine("Remarks: {0}", dbi.Message);
                }
            }

            // 遍历方法特性
            foreach (MethodInfo m in type.GetMethods())
            {
                foreach (Attribute a in m.GetCustomAttributes(true))
                {
                    DeBugInfo dbi = (DeBugInfo)a;
                    if (null != dbi)
                    {
                        Console.WriteLine("Bug no: {0}, for Method: {1}",
                                dbi.BugNo, m.Name);
                        Console.WriteLine("Developer: {0}", dbi.Developer);
                        Console.WriteLine("Last Reviewed: {0}",
                                dbi.LastReview);
                        Console.WriteLine("Remarks: {0}", dbi.Message);
                    }
```

```
                }
            }
            Console.ReadLine();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    Length: 4.5
    Width: 7.5
    Area: 33.75
    Bug No: 49
    Developer: Nuha Ali
    Last Reviewed: 10/10/2012
    Remarks: Unused variable
    Bug No: 45
    Developer: Zara Ali
    Last Reviewed: 12/8/2012
    Remarks: Return type mismatch
    Bug No: 55, for Method: GetArea
    Developer: Zara Ali
    Last Reviewed: 19/10/2012
    Remarks: Return type mismatch
    Bug No: 56, for Method: Display
    Developer: Zara Ali
    Last Reviewed: 19/10/2012
    Remarks:
```

三、反射的用法

反射用到的命名空间：

```
  System.Reflection
  System.Type
  System.Reflection.Assembly
```

反射用到的主要类：

System.Type 类－－通过这个类可以访问任何给定数据类型的信息。 **System.Reflection.Assembly**类－－它可以用于访问给定程序集的信息，或者把这个程序集加载到程序中。

- System.Type类:

System.Type 类对于反射起着核心的作用。但它是一个抽象的基类， `Type` 有与每种数据类型对应的派生类，我们使用这个派生类的对象的方法、字段、属性来查找有关该类型的所有信息。 获取给定类型的 `Type` 引用有3种常用方式:

```
1.      • 使用 C# typeof 运算符。
2.          Type t = typeof(string);
3.      • 使用对象GetType()方法。
4.          string s = "grayworm";
5.          Type t = s.GetType();
6.      • 还可以调用Type类的静态方法GetType()。
7.          Type t = Type.GetType("System.String");
```

上面这三类代码都是获取string类型的Type，在取出string类型的Type引用t后，我们就可以通过t来探测string类型的结构了。

```
1.      string n = "grayworm";
2.      Type t = n.GetType();
3.      foreach (MemberInfo mi in t.GetMembers())
4.      {
5.          Console.WriteLine("{0}\t{1}", mi.MemberType, mi.Name);
6.      }
```

Type类的属性:

```
1.      Name 数据类型名
2.      FullName 数据类型的完全限定名(包括命名空间名)
3.      Namespace 定义数据类型的命名空间名
4.      IsAbstract 指示该类型是否是抽象类型
5.      IsArray    指示该类型是否是数组
6.      IsClass    指示该类型是否是类
7.      IsEnum    指示该类型是否是枚举
8.      IsInterface     指示该类型是否是接口
```

```
    IsPublic 指示该类型是否是公有的
    IsSealed 指示该类型是否是密封类
    IsValueType 指示该类型是否是值类型
```

Type类的方法：

```
    GetConstructor(), GetConstructors()：返回ConstructorInfo类型，用于取得该类的构造函数的信息
    GetEvent(), GetEvents()：返回EventInfo类型，用于取得该类的事件的信息
    GetField(), GetFields()：返回FieldInfo类型，用于取得该类的字段（成员变量）的信息
    GetInterface(), GetInterfaces()：返回InterfaceInfo类型，用于取得该类实现的接口的信息
    GetMember(), GetMembers()：返回MemberInfo类型，用于取得该类的所有成员的信息
    GetMethod(), GetMethods()：返回MethodInfo类型，用于取得该类的方法的信息
    GetProperty(), GetProperties()：返回PropertyInfo类型，用于取得该类的属性的信息
可以调用这些成员，其方式是调用Type的InvokeMember()方法，或者调用MethodInfo,
PropertyInfo和其他类的Invoke()方法。
```

```
    查看类中的构造方法：
        NewClassw nc = new NewClassw();
        Type t = nc.GetType();
        ConstructorInfo[] ci = t.GetConstructors();    //获取类的所有构造函数
        foreach (ConstructorInfo c in ci) //遍历每一个构造函数
        {
            ParameterInfo[] ps = c.GetParameters();    //取出每个构造函数的所有参数
            foreach (ParameterInfo pi in ps)   //遍历并打印所该构造函数的所有参数
            {
                Console.Write(pi.ParameterType.ToString()+" "+pi.Name+",");
            }
            Console.WriteLine();
        }

    用构造函数动态生成对象：
        Type t = typeof(NewClassw);
        Type[] pt = new Type[2];
        pt[0] = typeof(string);
        pt[1] = typeof(string);
        //根据参数类型获取构造函数
        ConstructorInfo ci = t.GetConstructor(pt);
        //构造Object数组，作为构造函数的输入参数
```

```
23.         object[] obj = new object[2]{"grayworm","hi.baidu.com/grayworm"};
24.         //调用构造函数生成对象
25.         object o = ci.Invoke(obj);
26.         //调用生成的对象的方法测试是否对象生成成功
27.         //((NewClassw)o).show();
28.
29.     用Activator生成对象：
30.         Type t = typeof(NewClassw);
31.         //构造函数的参数
32.         object[] obj = new object[2] { "grayworm", "hi.baidu.com/grayworm" };
33.         //用Activator的CreateInstance静态方法，生成新对象
            object o =
34. Activator.CreateInstance(t,"grayworm","hi.baidu.com/grayworm");
35.         //((NewClassw)o).show();
36.
37.     查看类中的属性：
38.         NewClassw nc = new NewClassw();
39.         Type t = nc.GetType();
40.         PropertyInfo[] pis = t.GetProperties();
41.         foreach(PropertyInfo pi in pis)
42.         {
43.             Console.WriteLine(pi.Name);
44.         }
45.
46.     查看类中的public方法：
47.         NewClassw nc = new NewClassw();
48.         Type t = nc.GetType();
49.         MethodInfo[] mis = t.GetMethods();
50.         foreach (MethodInfo mi in mis)
51.         {
52.             Console.WriteLine(mi.ReturnType+" "+mi.Name);
53.         }
54.
55.     查看类中的public字段
56.         NewClassw nc = new NewClassw();
57.         Type t = nc.GetType();
58.         FieldInfo[] fis = t.GetFields();
59.         foreach (FieldInfo fi in fis)
60.         {
61.             Console.WriteLine(fi.Name);
62.         } (http://hi.baidu.com/grayworm)
63.
```

```
64.     用反射生成对象，并调用属性、方法和字段进行操作
65.         NewClassw nc = new NewClassw();
66.         Type t = nc.GetType();
67.         object obj = Activator.CreateInstance(t);
68.         //取得私有字段ID
            FieldInfo fi = t.GetField("ID", BindingFlags.NonPublic |
69. BindingFlags.Instance);
70.         //给ID字段赋值
71.         fi.SetValue(obj, "k001");
72.         //取得MyName属性
73.         PropertyInfo pi1 = t.GetProperty("MyName");
74.         //给MyName属性赋值
75.         pi1.SetValue(obj, "grayworm", null);
76.         PropertyInfo pi2 = t.GetProperty("MyInfo");
77.         pi2.SetValue(obj, "hi.baidu.com/grayworm", null);
78.         //取得show方法
79.         MethodInfo mi = t.GetMethod("show");
80.         //调用show方法
81.         mi.Invoke(obj, null);
```

- System.Reflection.Assembly类

Assembly类可以获得程序集的信息，也可以动态的加载程序集，以及在程序集中查找类型信息，并创建该类型的实例。　　使用**Assembly**类可以降低程序集之间的耦合，有利于软件结构的合理化。

```
1.
2.  通过程序集名称返回Assembly对象
3.         Assembly ass = Assembly.Load("ClassLibrary831");
4.
5.  通过DLL文件名称返回Assembly对象
6.         Assembly ass = Assembly.LoadFrom("ClassLibrary831.dll");
7.
8.  通过类型返回Assembly对象
9.         Assembly editorAssembly = Assembly.GetAssembly(typeof(NewClassw));
10.
11. 通过Assembly获取程序集中类
12.         Type t = ass.GetType("ClassLibrary831.NewClass");   //参数必须是类的全名
13.
14. 通过Assembly获取程序集中所有的类
15.         Type[] t = ass.GetTypes();
```

```
1.
```

```
    //通过程序集的名称反射
    Assembly ass = Assembly.Load("ClassLibrary831");
    Type t = ass.GetType("ClassLibrary831.NewClass");
    object o = Activator.CreateInstance(t, "grayworm",
"http://hi.baidu.com/grayworm");
    MethodInfo mi = t.GetMethod("show");
    mi.Invoke(o, null);

    //通过DLL文件全名反射其中的所有类型
    Assembly assembly = Assembly.LoadFrom("xxx.dll的路径");
    Type[] aa = a.GetTypes();

    foreach(Type t in aa)
    {
        if(t.FullName == "a.b.c")
        {
            object o = Activator.CreateInstance(t);
        }
    }
```

C# 不安全代码

当一个代码块使用 **unsafe** 修饰符标记时，C# 允许在函数中使用指针变量。不安全代码或非托管代码是指使用了指针变量的代码块。

一、指针变量

指针 是值为另一个变量的地址的变量，即，内存位置的直接地址。就像其他变量或常量，您必须在使用指针存储其他变量地址之前声明指针。

指针变量声明的一般形式为：

```
type *var-name;
```

以下是有效的指针声明：

```
int    *ip;    /* 指向一个整数 */
double *dp;    /* 指向一个双精度数 */
float  *fp;    /* 指向一个浮点数 */
char   *ch     /* 指向一个字符 */
```

下面的实例说明了 C# 中使用了 **unsafe** 修饰符时指针的使用：

```
using System;
namespace UnsafeCodeApplication
{
    class Program
    {
        static unsafe void Main(string[] args)
        {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} ",  var);
            Console.WriteLine("Address is: {0}",  (int)p);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Data is: 20
Address is: 99215364
```

您也可以不用声明整个方法作为不安全代码，只需要声明方法的一部分作为不安全代码。下面的实例说明了这点。

二、使用指针检索数据值

您可以使用 **ToString()** 方法检索存储在指针变量所引用位置的数据。下面的实例演示了这点：

```
using System;
namespace UnsafeCodeApplication
{
   class Program
   {
      public static void Main()
      {
         unsafe
         {
            int var = 20;
            int* p = &var;
            Console.WriteLine("Data is: {0} " , var);
            Console.WriteLine("Data is: {0} " , p->ToString());
            Console.WriteLine("Address is: {0} " , (int)p);
         }
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Data is: 20
Data is: 20
Address is: 77128984
```

三、传递指针作为方法的参数

您可以向方法传递指针变量作为方法的参数。下面的实例说明了这点：

```
1.    using System;
2.    namespace UnsafeCodeApplication
3.    {
4.       class TestPointer
5.       {
6.          public unsafe void swap(int* p, int *q)
7.          {
8.             int temp = *p;
9.             *p = *q;
10.            *q = temp;
11.         }
12.
13.         public unsafe static void Main()
14.         {
15.            TestPointer p = new TestPointer();
16.            int var1 = 10;
17.            int var2 = 20;
18.            int* x = &var1;
19.            int* y = &var2;
20.
21.            Console.WriteLine("Before Swap: var1:{0}, var2: {1}", var1, var2);
22.            p.swap(x, y);
23.
24.            Console.WriteLine("After Swap: var1:{0}, var2: {1}", var1, var2);
25.            Console.ReadKey();
26.         }
27.      }
28.   }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1.    Before Swap: var1: 10, var2: 20
2.    After Swap: var1: 20, var2: 10
```

四、使用指针访问数组元素

在 C# 中，数组名称和一个指向与数组数据具有相同数据类型的指针是不同的变量类型。例如，int *p 和 int[] p 是不同的类型。您可以增加指针变量 p，因为它在内存中不是固定的，但是数组地址在内存中是固定的，所以您不能增加数组 p。

因此，如果您需要使用指针变量访问数组数据，可以像我们通常在 C 或 C++ 中所做的那样，使用 fixed 关键字来固定指针。下面的实例演示了这点：

```
using System;
namespace UnsafeCodeApplication
{
   class TestPointer
   {
      public unsafe static void Main()
      {
         int[]  list = {10, 100, 200};
         fixed(int *ptr = list)

         /* 显示指针中数组地址 */
         for ( int i = 0; i < 3; i++)
         {
            Console.WriteLine("Address of list[{0}]={1}",i,(int)(ptr + i));
            Console.WriteLine("Value of list[{0}]={1}", i, *(ptr + i));
         }
         Console.ReadKey();
      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Address of list[0] = 31627168
Value of list[0] = 10
Address of list[1] = 31627172
Value of list[1] = 100
Address of list[2] = 31627176
Value of list[2] = 200
```

五、编译不安全代码

为了编译不安全代码，您必须切换到命令行编译器指定 **/unsafe** 命令行。

例如，为了编译包含不安全代码的名为 prog1.cs 的程序，需在命令行中输入命令：

```
1.     csc /unsafe prog1.cs
```

如果您使用的是 Visual Studio IDE，那么您需要在项目属性中启用不安全代码。步骤如下：

- 通过双击资源管理器（Solution Explorer）中的属性（properties）节点，打开项目属性（project properties）。
- 点击 Build 标签页。
- 选择选项"Allow unsafe code"。

C# 多线程

线程 被定义为程序的执行路径。每个线程都定义了一个独特的控制流。如果您的应用程序涉及到复杂的和耗时的操作，那么设置不同的线程执行路径往往是有益的，每个线程执行特定的工作。

线程是轻量级进程。一个使用线程的常见实例是现代操作系统中并行编程的实现。使用线程节省了 CPU 周期的浪费，同时提高了应用程序的效率。

到目前为止我们编写的程序是一个单线程作为应用程序的运行实例的单一的过程运行的。但是，这样子应用程序同时只能执行一个任务。为了同时执行多个任务，它可以被划分为更小的线程。

一、线程生命周期

线程生命周期开始于 System.Threading.Thread 类的对象被创建时，结束于线程被终止或完成执行时。

下面列出了线程生命周期中的各种状态：

- 未启动状态：当线程实例被创建但 Start 方法未被调用时的状况。
- 就绪状态：当线程准备好运行并等待 CPU 周期时的状况。
- 不可运行状态：下面的几种情况下线程是不可运行的：

```
1.   • 已经调用 Sleep 方法
2.   • 已经调用 Wait 方法
3.   • 通过 I/O 操作阻塞
```

- 死亡状态：当线程已完成执行或已中止时的状况。

二、主线程

在 C# 中，**System.Threading.Thread** 类用于线程的工作。它允许创建并访问多线程应用程序中的单个线程。进程中第一个被执行的线程称为主线程。

当 C# 程序开始执行时，主线程自动创建。使用 **Thread** 类创建的线程被主线程的子线程调用。您可以使用 Thread 类的 **CurrentThread** 属性访问线程。

下面的程序演示了主线程的执行：

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class MainThreadProgram
    {
        static void Main(string[] args)
        {
            Thread th = Thread.CurrentThread;
            th.Name = "MainThread";
            Console.WriteLine("This is {0}", th.Name);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
This is MainThread
```

三、Thread 类常用的属性和方法

下表列出了 **Thread** 类的一些常用的 属性：

属性	描述
CurrentContext	获取线程正在其中执行的当前上下文。
CurrentCulture	获取或设置当前线程的区域性。
CurrentPrinciple	获取或设置线程的当前负责人（对基于角色的安全性而言）。
CurrentThread	获取当前正在运行的线程。
CurrentUICulture	获取或设置资源管理器使用的当前区域性以便在运行时查找区域性特定的资源。
ExecutionContext	获取一个 ExecutionContext 对象，该对象包含有关当前线程的各种上下文的信息。
IsAlive	获取一个值，该值指示当前线程的执行状态。
IsBackground	获取或设置一个值，该值指示某个线程是否为后台线程。
IsThreadPoolThread	获取一个值，该值指示线程是否属于托管线程池。
ManagedThreadId	获取当前托管线程的唯一标识符。
Name	获取或设置线程的名称。

Priority	获取或设置一个值，该值指示线程的调度优先级。
ThreadState	获取一个值，该值包含当前线程的状态。

下表列出了 **Thread** 类的一些常用的 方法：

序号	方法名 & 描述
1	public void Abort()在调用此方法的线程上引发 ThreadAbortException，以开始终止此线程的过程。调用此方法通常会终止线程。
2	public static LocalDataStoreSlot AllocateDataSlot()在所有的线程上分配未命名的数据槽。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
3	public static LocalDataStoreSlot AllocateNamedDataSlot(string name) 在所有线程上分配已命名的数据槽。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
4	public static void BeginCriticalRegion()通知主机执行将要进入一个代码区域，在该代码区域内线程中止或未经处理的异常的影响可能会危害应用程序域中的其他任务。
5	public static void BeginThreadAffinity()通知主机托管代码将要执行依赖于当前物理操作系统线程的标识的指令。
6	public static void EndCriticalRegion()通知主机执行将要进入一个代码区域，在该代码区域内线程中止或未经处理的异常仅影响当前任务。
7	public static void EndThreadAffinity()通知主机托管代码已执行完依赖于当前物理操作系统线程的标识的指令。
8	public static void FreeNamedDataSlot(string name)为进程中的所有线程消除名称与槽之间的关联。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
9	public static Object GetData(LocalDataStoreSlot slot) 在当前线程的当前域中从当前线程上指定的槽中检索值。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
10	public static AppDomain GetDomain()返回当前线程正在其中运行的当前域。
11	public static AppDomain GetDomainID()返回唯一的应用程序域标识符。
12	public static LocalDataStoreSlot GetNamedDataSlot(string name) 查找已命名的数据槽。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
13	public void Interrupt()中断处于 WaitSleepJoin 线程状态的线程。
14	public void Join()在继续执行标准的 COM 和 SendMessage 消息泵处理期间，阻塞调用线程，直到某个线程终止为止。此方法有不同的重载形式。
15	public static void MemoryBarrier()按如下方式同步内存存取：执行当前线程的处理器在对指令重新排序时，不能采用先执行 MemoryBarrier 调用之后的内存存取，再执行 MemoryBarrier 调用之前的内存存取的方式。
16	public static void ResetAbort()取消为当前线程请求的 Abort。
17	public static void SetData(LocalDataStoreSlot slot, Object data) 在当前正在运行的线程上为此线程的当前域在指定槽中设置数据。为了获得更好的性能，请改用以 ThreadStaticAttribute 属性标记的字段。
18	public void Start()开始一个线程。
19	public static void Sleep(int millisecondsTimeout) 让线程暂停一段时间。
20	public static void SpinWait(int iterations) 导致线程等待由 iterations 参数定义的时间量。

21	public static byte VolatileRead(ref byte address)public static double VolatileRead(ref double address)public static int VolatileRead(ref int address)public static Object VolatileRead(ref Object address) 读取字段值。无论处理器的数目或处理器缓存的状态如何，该值都是由计算机的任何处理器写入的最新值。此方法有不同的重载形式。这里只给出了一些形式。
22	public static void VolatileWrite(ref byte address, byte value)public static void VolatileWrite(ref double address, double value)public static void VolatileWrite(ref int address, int value)public static void VolatileWrite(ref Object address, Object value) 立即向字段写入一个值，以使该值对计算机中的所有处理器都可见。此方法有不同的重载形式。这里只给出了一些形式。
23	public static bool Yield()导致调用线程执行准备好在当前处理器上运行的另一个线程。由操作系统选择要执行的线程。

四、创建线程

线程是通过扩展 Thread 类创建的。扩展的 Thread 类调用 **Start()** 方法来开始子线程的执行。

下面的程序演示了这个概念：

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class ThreadCreationProgram
    {
        public static void CallToChildThread()
        {
            Console.WriteLine("Child thread starts");
        }

        static void Main(string[] args)
        {
            ThreadStart childref = new ThreadStart(CallToChildThread);
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
In Main: Creating the Child thread
Child thread starts
```

五、管理线程

Thread 类提供了各种管理线程的方法。

下面的实例演示了 **sleep()** 方法的使用，用于在一个特定的时间暂停线程。

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class ThreadCreationProgram
    {
        public static void CallToChildThread()
        {
            Console.WriteLine("Child thread starts");
            // 线程暂停 5000 毫秒
            int sleepfor = 5000;
            Console.WriteLine("Child Thread Paused for {0} seconds",
                              sleepfor / 1000);
            Thread.Sleep(sleepfor);
            Console.WriteLine("Child thread resumes");
        }

        static void Main(string[] args)
        {
            ThreadStart childref = new ThreadStart(CallToChildThread);
            Console.WriteLine("In Main: Creating the Child thread");
            Thread childThread = new Thread(childref);
            childThread.Start();
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
In Main: Creating the Child thread
Child thread starts
Child Thread Paused for 5 seconds
Child thread resumes
```

六、销毁线程

Abort() 方法用于销毁线程。

通过抛出 **threadabortexception** 在运行时中止线程。这个异常不能被捕获，如果有 finally 块，控制会被送至 finally 块。

下面的程序说明了这点：

```
using System;
using System.Threading;

namespace MultithreadingApplication
{
    class ThreadCreationProgram
    {
        public static void CallToChildThread()
        {
            try
            {

                Console.WriteLine("Child thread starts");
                // 计数到 10
                for (int counter = 0; counter <= 10; counter++)
                {
                    Thread.Sleep(500);
                    Console.WriteLine(counter);
                }
                Console.WriteLine("Child Thread Completed");

            }
            catch (ThreadAbortException e)
            {
                Console.WriteLine("Thread Abort Exception");
            }
```

```
                finally
                {
                    Console.WriteLine("Couldn't catch the Thread Exception");
                }

            }

            static void Main(string[] args)
            {
                ThreadStart childref = new ThreadStart(CallToChildThread);
                Console.WriteLine("In Main: Creating the Child thread");
                Thread childThread = new Thread(childref);
                childThread.Start();
                // 停止主线程一段时间
                Thread.Sleep(2000);
                // 现在中止子线程
                Console.WriteLine("In Main: Aborting the Child thread");
                childThread.Abort();
                Console.ReadKey();
            }
        }
    }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
    In Main: Creating the Child thread
    Child thread starts
    0
    1
    2
    In Main: Aborting the Child thread
    Thread Abort Exception
    Couldn't catch the Thread Exception
```

LINQ

Language Intergrated Query(Linq) 集成查询语言

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.Collections.Concurrent;
4.  using System.Linq;
5.
6.  namespace LinqTest
7.  {
8.
9.      class MainClass
10.     {
11.         public static void Main(string[] args)
12.         {
13.
14.
15.          /*
16.           * where         筛选操作符定义了返回元素的条件
17.           *
18.           * Select        投射操作符用于把对象转换为另一个类型的新对象。
19.           *
20.           * OrderBy       排序操作符返回的元素的顺序
21.           * ThenBy
22.           * OerderByDescending
23.           * ThenByDescending
24.           * Reverse
25.           *
26.           * Join          连接操作符用于合并不直接相关的集合。
27.           * GroupJoin
28.           *
29.           * GroupBy       组合操作符把数组放在组中。
30.           * ToLookup
31.           *
32.           * Any           如果元素序列满足指定的条件，限定符操作就返回布尔值
33.           * All
34.           * Contains      检查某个元素是否在集合中
35.           *
36.           * Take          分区操作符返回集合的一个子集。指定要从集合中提取的元素个数
37.           * Skip          跳过指定的元素个数
```

```
38.             * TakeWhile
39.             * SkipWhile
40.             *
41.             * Distinct     Set操作符返回一个集合。从集合中删除重复的元素。
42.             * Union        需要两个集合，返回出现在其中一个集合中的唯一元素。
43.             * Intersect    需要两个集合，返回两个集合都有的元素。
44.             * Except       需要两个集合，返回只出现在一个集合中的元素。
45.             * Zip          需要两个集合，把两个集合合并为一个
46.             *
47.             * First        这些元素操作符仅返回一个元素。返回第一个满足条件的元素。
48.             * FirstOrDefault    返回第一个满足条件的元素。如果没有找到，就返回默认值。
49.             * Last
50.             * LastOrDefault
51.             * ElementAt    指定了要返回的元素的位置
52.             * ElementAtOrDefault
                * Single       只返回一个满足条件的元素。如果有多个元素都满足条件，就抛出一个异
53. 常。
54.             * SingleOrDefault
55.             *
                * Count        聚合操作符计算集合的一个值。利用这些聚合操作符，可以计算所有值的
56. 总和、所有元素的个数
57.             * Sum          总和
58.             * Min          值最小的元素
59.             * Max          值最大的元素
60.             * Average      平均值
61.             * Aggregate    聚合      可以传递一个lambda表达式，该表达式对有所的值进行聚合
62.             *
63.             * ToArray      转换为数组
64.             * AsEnumerable 转换为IEnumerable
65.             * ToList       转换为IList
66.             * ToDictionary 转换为IDictionary
67.             * Cast<TResult>
68.             *
69.             * Empty        这些生成操作符返回一个新集合。使用Empty时集合是空的
70.             * Range        返回一系列数字
                * Repeat       返回一个始终重复一个值的集合；返回一个迭代器，该迭代器把同一个值
71. 重复特定的次数。
72.             *
73.             */
74.
75.
76.
77.
```

```
78.
79.             // 1. 查询出 Unity1609 的所有冠军 并按照总分排序
80.
81.             /*
82.              * 查询表达式必须以 from 子句开头，以 select 或 group 子句结束。
83.              * 在这两个子句之间，可以使用 where、oderby、join、let 和其他 from 子句。
84.              */
85.
86.             var query = from n in Formula.GetChampions()
87.                         where n.ClassesName == "Unity1609"
88.                         orderby n.Sum descending
89.                         select n;
90.
91.             // 使用扩展方法实现
92.             var champions = new List<Student>(Formula.GetChampions());
                IEnumerable<Student> result = champions.Where(o => o.ClassesName ==
93. "Unity1609").
94.             OrderByDescending(o => o.Sum).
95.             Select(n => n);
96.
97.
98.             foreach (var item in result)
99.             {
100.                Console.WriteLine("{0:N}", item);
101.            }
102.
103.
104.
105.
106.            // 2. 筛选
107.            // 使用where子句，可以合并多个表达式。
108.
109.
110.            // 3. 用索引筛选
                // 不能使用LINQ查询的一个例子是Where()方法的重载。在Where()方法的重载时，可
111. 以传递第2个参数-索引。
112.            // 索引是筛选器返回的每个结果的计数器。
113.
114.
115.            // 4. 类型筛选
116.            // 为了进行基于类型的筛选，可以使用OfType<>()扩展方法。
117.
118.            object[] data1 = { "one", 2, 3, "four" };
```

```
            var query1 = data1.OfType<string>();    // 从集合中仅返回字符串


            // 5. 复合的from子句
            // 如果需要根据对象的一个成员进行筛选，而该成员本身是一个系列，就可以使用复合的from子句。
            // 查询元素是集合的元素  扩展方法 SelectMany() 用于迭代序列的序列
            //var query2 = from r in Formula.GetChampions()
            //from c in r.Scores
            //where c > 90
            //orderby r.Sum
            //select r.Name + " " + r.Sum;

            var query2 = Formula.GetChampions().
            SelectMany(source => source.Scores, (source, collection) => new { Stu = source, Scores = collection }).
            Where(newObj => newObj.Stu.Sum > 90).
            OrderBy(newObj => newObj.Stu.Sum).
            Select(newObj => newObj.Stu.Name + " " + newObj.Stu.Sum);


            // 6. 排序
            // 要对序列排序，前面使用了orderby子句。


            // 7. 分组
            // 要根据一个关键字值对查询结果分组，可以使用group子句。


            //var query3 = from n in Formula.GetChampions()
            //group n by n.ClassesName into g
            //orderby g.Count(), g.Key
            //where g.Count() >= 2
            //select new
            //{
            //    ClassName = g.Key,
            //    Count = g.Count()
            //};

            // GroupBy()  IGrouping  Key
            var query3 = Formula.GetChampions().
            GroupBy(n => n.ClassesName).
            OrderByDescending(group => group.Count()).
```

```
            ThenBy(group => group.Key).
            Where(g => g.Count() >= 2).
            Select(g => new { ClassName = g.Key, Count = g.Count() });


            // 8. 对嵌套的对象分组
            // 如果分组的对象应包含嵌套的序列，就可以改变select子句创建的匿名类型
            var query4 = from r in Formula.GetChampions()
                         group r by r.ClassesName into g
                         orderby g.Count() descending, g.Key
                         where g.Count() >= 2
                         select new
                         {
                             ClassName = g.Key,
                             Count = g.Count(),
                             Scores = from r1 in g
                                      orderby r1.Sum descending, r1.Name
                                      select r1.Scores
                         };

            foreach (var item in query4)
            {
                Console.WriteLine(item.ClassName + " " + item.Count);
            }


            // 9. 内连接
            // 使用join子句可以根据特定的条件合并两个数据源
            var query5 = from n in Formula.GetChampions()
                         from score in n.Scores
                         select new
                         {
                             Name = n.Name,
                             Score = score,
                             ClassName = n.ClassesName,
                             Years = n.Years
                         };
            var query6 = from n in Formula.GetContructorChampions()
                         from y in n.Years
                         select new
                         {
                             ClassName = n.ClassesName,
```

```
                                Year = y
                            };
                var query7 = (from n in query5
                              join t in query6 on n.ClassName equals t.ClassName
                              select new
                              {
                                  n.ClassName,
                                  n.Name,
                                  n.Score,
                                  t.Year
                              }).Take(10);

                foreach (var item in query7)
                {
                    Console.WriteLine("{0},{1},{2},{3}", item.ClassName, item.Name,
item.Score, item.Year);
                }


                // 10. 左外连接
                // 返回左边序列的全部元素，即时它们在右边的序列中没有匹配的元素
                // 使用 join into、 DefaultIfEmpty 定义
                var query8 = from n in query5
                             join t in query6 on n.ClassName equals t.ClassName
into rt
                             from q in rt.DefaultIfEmpty()
                             orderby n.Score
                             select new
                             {
                                 Name = n.Name,
                                 ClassName = q == null ? "NoClassName" :
n.ClassName,
                                 Score = n.Score,
                                 Year = q == null ? -1 : q.Year
                             };

                foreach (var item in query8)
                {
                    Console.WriteLine($"{item.Name},{item.ClassName},{item.Score},
{item.Year}");
                }


```

```
            // 11. 组连接
            // 左外连接使用了组连接和into子句。它有一部分语法与组连接相同。只不过组连接不使用DefaultIfEmpty方法。
            // 使用组连接时，可以连接两个独立的序列，对于其中一个序列中的某个元素，另一个序列中存在对应的一个项列表。

            var query9 = from r in query5
                         from y in r.Years
                         join r2 in query6 on
                         new { ClassName = r.ClassName, Year = y }
                         equals
                         new { ClassName = r2.ClassName, Year = r2.Year }
                         into g
                         select new { ClassName = r.ClassName, Name = r.Name, Year = y, Score = r.Score };

            foreach (var item in query9)
            {
                Console.WriteLine($"{item.ClassName},{item.Name},{item.Score},{item.Year}");
            }


            // 12. 集合操作
            // 扩展方法 Distinct()、Union()、Intersect()和Except() 都是集合操作。


            // 13. 合并
            // Zip()方法允许用一个谓词函数把两个相关的序列合并为一个。
            var query10 = query5.Zip(query6, (first, second) => first.ClassName + "," + first.Name + "," + second.Year);


            // 14. 分区
            // 扩展方法 Take() 和 Skip() 等的分区操作可用于分页。
            // 使用 TakeWhile() 和 SkipWhile() 扩展方法，还可以传递一个谓词，根据谓词的结果提取或跳过某些项。


            // 15. 聚合操作符
            // 聚合操作符（如Count()、Sum()、Min()、Max()、Average()和Aggregate()）不返回一个序列，而返回一个值。
            var query11 = from r in Formula.GetChampions()
```

```
                            let scoreCount = r.Scores.Count()        // let子句定
277.  义了一个变量
278.                            where scoreCount >= 2
279.                            orderby scoreCount descending
280.                            select new
281.                            {
282.                                Name = r.Name,
283.                                ClassName = r.ClassesName
284.                            };
285.
286.
287.              // 16. 转换操作符
288.              // ToList()
289.              // Lookup<TKey, TElement>   一个键可以对应多个值
290.              var query12 = (from r in Formula.GetChampions()
291.                             from s in r.Scores
292.                             select new
293.                             {
294.                                 ClassName = r.ClassesName,
295.                                 Name = r.Name,
296.                                 Score = s
297.                             }).ToLookup(cr => cr.ClassName, cr => cr.Name);
298.
299.              foreach (var item in query12["Unity1605"])
300.              {
301.                  Console.WriteLine($"{item}");
302.              }
303.
                 // 如果需要在非类型化的集合上（如ArrayList）使用LINQ查询，就可以使用Cast()方
304.  法。
305.
                 var list = new System.Collections.ArrayList(Formula.GetChampions()
306.  as System.Collections.ICollection);
307.              var query13 = from r in list.Cast<Student>()
308.                            where r.ClassesName == "Unity1605"
309.                            select r;
310.              foreach (var item in query13)
311.              {
312.                  Console.WriteLine($"{item.ClassesName}, {item.Name}");
313.              }
314.
315.
316.              // 17. 生成操作符
```

```
            // 生成操作符Range()、Empty()和Repeat()不是扩展方法，而是返回序列的正常静态
317. 方法。
            //var values = Enumerable.Range(1, 20);   // 第一个参数作为起始值，第二
318. 个参数作为要填充的项数
319.             //values = values.Select(n => n * 3);
320.             //foreach (var item in values)
321.             //{
322.             //    Console.WriteLine(item);
323.             //}
            // Range()方法不返回填充了所定义值的集合，这个方法与其他方法一样，也推迟执行查
324. 询，
325.             // 并返回一个RangeEnumerator，其中只有一条yield return 语句，来递增值。
326.
327.
328.             // 18. 并行LINQ
            // System.Linq名称空间中包含的类ParalleEnumerable可以分解查询的工作，使其
329. 分布在多个线程上。
330.
331.             //var res = (from x in SampleData().AsParallel()
332.             //where Math.Log(x) < 4
333.             //select x).Average();
334.
            var res = SampleData().AsParallel().Where(x => Math.Log(x) <
335. 4).Select(x => x).Average();
336.             Console.WriteLine(res);
337.
338.             // 19. 分区器
            // AsParalle() 方法不仅扩展了IEnuerable<T> 接口，还扩展了 Partitioner
339. 类。通过它，可以影响要创建的分区。
            // Partitioner 类用 System.Collection.ConCurrent 名称空间定义，并且有不
340. 同的变体。
341.             // Create() 方法接受实现了 IList<T> 类的数组或对象。
342.
343.             var data = SampleData();
344.             var res1 = (from x in Partitioner.Create(data).AsParallel()
345.                         where Math.Log(x) < 4
346.                         select x).Average();
347.             Console.WriteLine(res1);
348.
            // 也可以调用 WithExecutionMode() 和 WithDegreeOfParallelism() 方法，
349. 来影响并行机制。
350.
351.             // 20. 取消
```

```
352.             // .NET 提供了一种标准方式，来取消长时间运行的任务，这也适用于并行LINQ
                 // 要取消长时间运行的查询，可以给查询添加WithCancellation()方法，并传递一个
353. CancellationToken令牌作为参数。
354.             // CancellationToken令牌从CancellationTokenSource类中创建。
                 // 该查询在单独的线程中运行，在该线程中，捕获一个
355. OperationCanceledException类型的异常。
                 // 如果取消了查询，就触发这个异常。在主线程中，调用CancellationTokenSource
356. 类的Cancel()方法可以取消任务。
357.
358.
359.             // 21. 表达式树 Expression<T>
360.         }
361.
362.
363.         static IEnumerable<int> SampleData()
364.         {
365.             const int arraySize = 100000000;
366.             var r = new Random();
                 return Enumerable.Range(0, arraySize).Select(x =>
367. r.Next(140)).ToList();
368.         }
369.     }
370.
371.
372.
373.
374.     #region 扩展方法
375.     /*
          * 扩展方法 ： 定义为静态方法，其第一个参数定义了它扩展的类型，扩展方法在一个静态类中声
376. 明。
          *           为了区分扩展方法和一般的静态方法，扩展方法还需要对第一个参数使用 this 关
377. 键字。
378.      *
          *           扩展方法不能访问它扩展的类型的私有成员。调用扩展方法只是调用静态方法的一种
379. 新语法。
380.      *           string s = "hello";
381.      *           s.Foo();
382.      *           StringExtension.Foo(s);
383.      */
384.
385.     public static class StringExtension
386.     {
387.         public static void Foo(this string s)
```

```
388.         {
389.             Console.WriteLine("Foo invoked for {0}", s);
390.         }
391.     }
392.     #endregion
393.
394.     // 2016~2018年间，所有获得第一名的名单
395.     static class Formula
396.     {
397.         public static List<Student> nameList;
398.
399.         public static IList<Student> GetChampions()
400.         {
401.             if (nameList == null)
402.             {
403.                 nameList = new List<Student>()
404.                 {
                        new Student("张三", 18, "Unity1605", new int[]{99, 98}, new
405. int[]{2016}),
                        new Student("李四", 20, "Unity1605", new int[]{98, 97}, new
406. int[]{2016}),
                        new Student("李四", 20, "Unity1607", new int[]{98, 97}, new
407. int[]{2016}),
                        new Student("刘五", 21, "Unity1609", new int[]{98, 96}, new
408. int[]{2016}),
                        new Student("王二", 21, "Unity1609", new int[]{99, 96}, new
409. int[]{2016}),
                        new Student("王六", 22, "Unity1609", new int[]{94, 95}, new
410. int[]{2017}),
                        new Student("赵六", 23, "Unity1701", new int[]{96, 96}, new
411. int[]{2017}),
                        new Student("田七", 23, "Unity1711", new int[]{95, 97}, new
412. int[]{2017}),
                        new Student("孙二", 22, "Unity1711", new int[]{94, 98}, new
413. int[]{2018}),
                        new Student("王八", 21, "Unity1801", new int[]{99, 99}, new
414. int[]{2018}),
                        new Student("王八", 21, "Unity1805", new int[]{99, 99}, new
415. int[]{2018}),
                 };
417.             }
418.             return nameList;
419.         }
420.
```

```
421.         private static List<ClassesChampion> teams;
422.         public static IList<ClassesChampion> GetContructorChampions()
423.         {
424.             if (teams == null)
425.             {
426.                 teams = new List<ClassesChampion>()
427.                 {
428.                     new ClassesChampion("Unity1605", 2016),
429.                     new ClassesChampion("Unity1607", 2016),
430.                     new ClassesChampion("Unity1609", 2016, 2017),
431.                     new ClassesChampion("Unity1701", 2017),
432.                     new ClassesChampion("Unity1711", 2017, 2018),
433.                     new ClassesChampion("Unity1801", 2018),
434.                     new ClassesChampion("Unity1803", 2019)
435.                 };
436.             }
437.             return teams;
438.         }
439.     }
440. 
441.     // 排行榜 获得第一名学员的班级名字和年份
442.     class ClassesChampion
443.     {
444.         public ClassesChampion(string name, params int[] years)
445.         {
446.             this.ClassesName = name;
447.             this.Years = new List<int>(years);
448.         }
449. 
450.         public string ClassesName { get; private set; }
451.         public IEnumerable<int> Years { get; private set; }
452.     }
453. 
454. 
455.     class Student : IComparable<Student>, IFormattable
456.     {
457.         public string Name { get; private set; }
458.         public string ClassesName { get; private set; }
459.         public int Age { get; set; }
460.         public int Sum { get; private set; }
461. 
462.         public IEnumerable<int> Scores { get; private set; }
```

```
        public IEnumerable<int> Years { get; private set; }        // 获得第一名
463. 的年份
464.
        public Student(string name, int age, string classesName) : this(name,
465. age, classesName, null, null) { }
466.
        public Student(string name, int age, string classesName,
467. IEnumerable<int> scores, IEnumerable<int> years)
468.         {
469.             this.Name = name;
470.             this.ClassesName = classesName;
471.             this.Age = age;
472.             this.Scores = new List<int>(scores);
473.             this.Years = new List<int>(years);
474.
475.             foreach (var item in Scores)
476.             {
477.                 Sum += item;
478.             }
479.         }
480.
481.         public override string ToString()
482.         {
483.             return string.Format("[Student: Name={0}, Age={1}]", Name, Age);
484.         }
485.
486.         public string ToString(string format)
487.         {
488.             return ToString(format, null);
489.         }
490.
491.         public string ToString(string format, IFormatProvider formatProvider)
492.         {
493.             switch (format)
494.             {
495.                 case "N":
496.                     return this.Name;
497.                 default:
498.                     return ToString();
499.             }
500.         }
501.
502.         public int CompareTo(Student other)
```

```
503.         {
504.             if (other == null) return 1;
505.             return this.Sum - other.Sum;
506.         }
507.     }
508. }
```

```
 1.             // Aggregate 用法
 2.
                // ---------------------------- 一、对第1个和第2个元素执行操作，将执行的
 3. 结果继续携带进行操作 ----------------
 4.             // 求1～100的和
 5.             var list = Enumerable.Range(1, 100);
 6.             var sum = list.Aggregate((a, b) => a + b);
 7.             Console.WriteLine(sum);
 8.
                // ---------------------------- 二、种子重载 -----------------------
 9. --------------------------------
10.             // 先从种子开始作为第一个元素执行操作。
11.             // 求1～5的阶乘
12.             var nums = Enumerable.Range(2, 4);  // 2 3 4 5
13.             var sum1 = nums.Aggregate(1, (a, b) => a * b);
14.             Console.WriteLine(sum1);
15.
16.             // 翻转单词
17.             string content = "i am hxsd";
18.             content = content.Split(' ').Aggregate((a, b)=> b + " " + a );
19.             Console.WriteLine(content);
20.
                // --------------------------- 三、结果选择器 ----------------------
21. --------------------------------
22.             // 最长的字符串 输出大写
23.             string[] fruits = { "apple", "mango", "orange", "passionfruit" };
                string longestName = fruits.Aggregate(fruits[0], (longest, next) =>
24. longest.Length < next.Length ? next : longest, n => n.ToUpper());
25.             Console.WriteLine(longestName);
```

C#4.0 协变 和 抗变（逆变）

关于协变和逆变要从面向对象继承说起。继承关系是指子类和父类之间的关系；子类从父类继承所以子类的实例也就是父类的实例。比如说Animal是父类，Dog是从Animal继承的子类；如果一个对象的类型是Dog，那么他必然是Animal。协变逆变正是利用继承关系 对不同参数类型或返回值类型 的委托或者泛型接口之间做转变。

如果一个方法要接受Dog参数，那么另一个接受Animal参数的方法肯定也可以接受这个方法的参数，这是Animal向Dog方向的转变是逆变。如果一个方法要求的返回值是Animal，那么返回Dog的方法肯定是可以满足其返回值要求的，这是Dog向Animal方向的转变是协变。

由子类向父类方向转变是协变 协变用于返回值类型用out关键字由父类向子类方向转变是逆变 逆变用于方法的参数类型用in关键字

一、定义

一个可变性和子类到父类转换的方向一样，就称作协变；而如果和子类到父类的转换方向相反，就叫抗变！

那到底这个协变或者抗变有什么实际利用价值呢？其价值就在于，在.net 4.0之前可以这么写：

```
1.      Sharp sharp = new Rectange();
```

但是却不能这么写：

```
1.      IEnumerable<Sharp> sharps = new List<Rectange>();
```

4.0之后，可以允许按上面的写法了，因为泛型接口 `IEnumerable<T>` 被声明成如下：

```
1.      public interface IEnumerable<out T> : IEnumerable
```

数组不支持抗变。在.Net 4.0之后，支持协变和抗变的有两种类型：泛型接口和泛型委托。

二、泛型接口中的协变和抗变

接下来定义一个泛型接口：

```
1.      public interface ICovariant<T>  // Covariant 协变的
```

并且让上面的两个类各自继承一下该接口：

```
1.      public class Sharp : ICovariant<Sharp>
2.      {
3.      }
4.
5.      public class Rectange : Sharp,ICovariant<Rectange>
6.      {
7.      }
```

编写测试代码：

```
1.      static void Main(string[] args)
2.      {
3.          ICovariant<Sharp> isharp = new Sharp();
4.          ICovariant<Rectange> irect = new Rectange();
5.
6.          isharp = irect;
7.      }
```

编译并不能通过，原因是无法将 `ICovariant<Rectange>` 隐式转化为 `ICovariant<Sharp>` ！

再将接口修改为：

```
1.      public interface ICovariant<out T>
2.      {
3.      }
```

编译顺利通过。这里我为泛型接口的类型参数增加了一个修饰符 `out` ，它表示这个泛型接口支持对类型T的协变。

即：如果一个泛型接口 `IFoo<T>` ， `IFoo<TSub>` 可以转换为 `IFoo<TParent>` 的话，我们称这个过程为协变，而且说“这个泛型接口支持对T的协变”。

那我如果反过来呢，考虑如下代码：

```
1.      static void Main(string[] args)
2.      {
3.          ICovariant<Sharp> isharp = new Sharp();
4.          ICovariant<Rectange> irect = new Rectange();
5.
```

```
6.          irect = isharp;
7.         // isharp =irect;
8.      }
```

发现编译又不通过了， 原因是无法将 `ICovariant<Sharp>` 隐式转化为 `ICovariant<Rectange>` ！

将接口修改为：

```
1.      public interface ICovariant<in T>
2.      {
3.      }
```

编译顺利通过。这里我将泛型接口的类型参数T修饰符修改成 `in` ，它表示这个泛型接口支持对类型参数T的抗变。即：如果一个泛型接口 `IFoo<T>` ， `IFoo<TParent>` 可以转换为 `IFoo<TSub>` 的话，我们称这个过程为抗变（contravariant），而且说“这个泛型接口支持对T的抗变”！

泛型接口并不单单只有一个参数，所以我们不能简单地说一个接口支持协变还是抗变，只能说一个接口对某个具体的类型参数支持协变或抗变，如 `ICovariant<out T1,in T2>` 说明该接口对类型参数T1支持协变，对T2支持抗变。

举个例子就是： `ICovariant<Rectange,Sharp>` 能够转化成 `ICovariant<Sharp,Rectange>` ，这里既有协变也有抗变。

以上都是接口并没有属性或方法的情形，接下来给接口添加一些方法：

```
1.      //这时候，无论如何修饰T，都不能编译通过
2.      public interface ICovariant<out T>
3.      {
4.          T Method1();
5.          void Method2(T param);
6.      }
```

发现无论用 `out` 还是 `in` 修饰T参数，根本编译不通过。

原因是，我把仅有的一个类型参数T既用作函数的返回值类型，又用作函数的参数类型。

所以：1）当我用out修饰时，即允许接口对类型参数T协变，也就是满足从 `ICovariant<Rectange>` 到 `ICovariant<Sharp>` 转换，Method1返回值Rectange到Sharp转换没有任何问题：

```
1.      ICovariant<Sharp> isharp = new Sharp();
2.      ICovariant<Rectange> irect = new Rectange();
```

```
3.
4.      isharp = irect;
5.      Sharp sharp = isharp.Method1();
```

但是对于把T作为参数类型的方法Method2(Rectange)会去替换Method2(Sharp):

```
1.      ICovariant<Sharp> isharp = new Sharp();
2.      ICovariant<Rectange> irect = new Rectange();
3.
4.      isharp = irect;
5.      isharp.Method2(new Sharp());
```

即如果执行最后一行代码，会发现参数中，Sharp类型并不能安全转化成Rectange类型，因为Method2(Sharp)实际上已经被替换成Method2(Rectange) ！

2)同样，当我用in修饰时， 即允许接口对类型参数T抗变，也就是满足
从 `ICovariant<Sharp>` 到 `ICovariant<Rectange>` 转换:

```
1.      ICovariant<Sharp> isharp = new Sharp();
2.      ICovariant<Rectange> irect = new Rectange();
3.
4.      //isharp = irect;
5.      irect = isharp;
6.      irect.Method2(new Rectange());
```

Method2(Sharp)会去替换Method2(Rectange)，所以上面的最后一句代码无论以Rectange类型还是Sharp类型为参数都没有任何问题；但是Method1返回的将是Sharp类型:

```
1.      ICovariant<Sharp> isharp = new Sharp();
2.      ICovariant<Rectange> irect = new Rectange();
3.
4.      //isharp = irect;
5.      irect = isharp;
6.      Rectange rect = irect.Method1();
```

执行最后一句代码，同样将会是不安全的！

综上：在没有额外机制的限制下，接口进行协变或抗变都是类型不安全的。.NET 4.0有了改进，它允许在类型参数的声明时增加一个额外的描述，以确定这个类型参数的使用范围，这个额外的描述即in,out修饰符，它们俩的用法如下：如果一个类型参数仅仅能用于函数的返回值，那么这个类型参数就对协变相容，用out修饰。而相反，一个类型参数如果仅能用于方法参数，那么这个类型参数就对抗

变相容，用in修饰。

所以，需要将上面的接口拆成两个接口即可：

```
1.      public interface ICovariant<out T>
2.      {
3.          T Method1();
4.
5.      }
6.
7.      public interface IContravariant<in T>
8.      {
9.          void Method2(T param);
10.     }
```

.net中很多接口都仅将参数用于函数返回类型或函数参数类型，如：

```
1.      public interface IComparable<in T>
2.
3.
4.      public interface IEnumerable<out T> : IEnumerable
```

几个重要的注意点：1.仅有泛型接口和泛型委托支持对类型参数的可变性，泛型类或泛型方法是不支持的。2.值类型不参与协变或抗变，`IFoo<int>`永远无法协变成`IFoo<object>`，不管有无声明out。因为.NET泛型，每个值类型会生成专属的封闭构造类型，与引用类型版本不兼容。3.声明属性时要注意，可读写的属性会将类型同时用于参数和返回值。因此只有只读属性才允许使用out类型参数，只写属性能够使用in参数。

接下来将接口代码改成：

```
1.      public interface ICovariant<out T>
2.      {
3.          T Method1();
4.          void Method3(IContravariant<T> param);
5.      }
6.
7.      public interface IContravariant<in T>
8.      {
9.          void Method2(T param);
10.     }
```

同样是可以编译通过的.

我们需要费一些周折来理解这个问题。现在我们考虑 `ICovariant<Rectange>` ，它应该能够协变成 `ICovariant<Sharp>` ，因为Rectange是Sharp的子类。因此Method3(Rectange)也就协变成了Method3(Sharp)。当我们调用这个协变,Method3(Sharp)必须能够安全变成Method3(Rectange)才能满足原函数的需要(具体原因上面已经示例过了）。这里对Method3的参数类型要求是Sharp能够抗变成Rectange！也就是说，如果一个接口需要对类型参数T协变，那么这个接口所有方法的参数类型必须支持对类型参数T的抗变（如果T有作为某些方法的参数类型）。同理我们也可以看出，如果接口要支持对T抗变，那么接口中方法的参数类型都必须支持对T协变才行。这就是方法参数的协变-抗变互换原则。所以，我们并不能简单地说out参数只能用于方法返回类型参数，它确实只能直接用于声明返回值类型，但是只要一个支持抗变的类型协助，out类型参数就也可以用于参数类型！（即上面的例子），换句话说，in除了直接声明方法参数类型支持抗变之外，也仅能借助支持协变的类型才能用于方法参数，仅支持对T抗变的类型作为方法参数类型也是不允许的。

既然方法类型参数协变和抗变有上面的互换影响。那么方法的返回值类型会不会有同样的问题呢？将接口修改为：

```
1.      public interface IContravariant<in T>
2.      {
3.
4.      }
5.      public interface ICovariant<out T>
6.      {
7.
8.      }
9.
10.     public interface ITest<out T1, in T2>
11.     {
12.         ICovariant<T1> test1();
13.         IContravariant<T2> test2();
14.     }
```

我们看到和刚刚正好相反，如果一个接口需要对类型参数T进行协变或抗变，那么这个接口所有方法的返回值类型必须支持对T同样方向的协变或抗变（如果有某些方法的返回值是T类型）。这就是方法返回值的协变-抗变一致原则。也就是说，即使in参数也可以用于方法的返回值类型，只要借助一个可以抗变的类型作为桥梁即可。

三、泛型委托中的协变和抗变

泛型委托的协变抗变，与泛型接口协变抗变类似。继续延用Sharp，Rectange类作为示例：新建一个简单的泛型接口：

```
1.      public delegate void MyDelegate1<T>();
```

测试代码：

```
1.      MyDelegate1<Sharp> sharp1 = new MyDelegate1<Sharp>(MethodForParent1);
2.      MyDelegate1<Rectange> rect1 = new MyDelegate1<Rectange>(MethodForChild1);
3.      sharp1 = rect1;
```

其中两个方法为：

```
1.      public static void MethodForParent1()
2.      {
3.          Console.WriteLine("Test1");
4.      }
5.      public static void MethodForChild1()
6.      {
7.          Console.WriteLine("Test2");
8.      }
```

编译并不能通过，因为无法将 `MyDelegate1<Rectange>` 隐式转化为 `MyDelegate1<Sharp>` ，接下来我将接口修改为支持对类型参数T协变，即加out修饰符：

```
1.      public delegate void MyDelegate1<out T>();
```

编译顺利用过。同样，如果反过来，对类型参数T进行抗变：

```
1.      MyDelegate1<Sharp> sharp1 = new MyDelegate1<Sharp>(MethodForParent1);
2.      MyDelegate1<Rectange> rect1 = new MyDelegate1<Rectange>(MethodForChild1);
3.      //sharp1 = rect1;
4.      rect1 = sharp1;
```

只需将修饰符改为in即可：

```
1.      public delegate void MyDelegate1<in T>();
```

考虑第二个委托：

```
1.      public delegate T MyDelegate2<out T>();
```

测试代码：

```
MyDelegate2<Sharp> sharp2 = new MyDelegate2<Sharp>(MethodForParent2);
MyDelegate2<Rectange> rect2 = new MyDelegate2<Rectange>(MethodForChild2);
sharp2 = rect2;
```

其中两个方法为：

```
public static Sharp MethodForParent2()
{
    return new Sharp();
}
public static Rectange MethodForChild2()
{
    return new Rectange();
}
```

该委托对类型参数T进行协变没有任何问题，编译通过；如果我要对T进行抗变呢？是否只要将修饰符改成in就OK了？测试如下：

```
public delegate T MyDelegate2<in T>();

MyDelegate2<Sharp> sharp2 = new MyDelegate2<Sharp>(MethodForParent2);
MyDelegate2<Rectange> rect2 = new MyDelegate2<Rectange>(MethodForChild2);
//sharp2 = rect2;
rect2 = sharp2;
```

错误如下：变体无效： 类型参数“T”必须为对于“ `MyDelegate2<T>.Invoke()` ”有效的 协变式。“T”为 逆变。意思就是：这里的类型参数T已经被声明成抗变，如果上面的最后一句有效，那么以后rect2()执行结果返回的将是一个Sharp类型的实例，如果再出现这种代码：

```
Rectange rectange = rect2();
```

那么这将是一个从Sharp类到Rectange类的不安全的类型转换！所以如果类型参数T抗变，并且要用于方法返回类型，那么方法的返回类型也必须支持抗变。即上面所说的方法返回类型协变-抗变一致原则。

那么如何对上面的返回类型进行抗变呢？很简单，只要借助一个支持抗变的泛型委托作为方法返回类型即可：

```
public delegate Contra<T> MyDelegate2<in T>();
public delegate void Contra<in T>();
```

具体的方法也需要对应着修改一下：

```
    public static Contra<Sharp> MethodForParent3()
    {
        return new Contra<Sharp>(MethodForParent1);
    }
    public static Contra<Rectange> MethodForChild3()
    {
        return new Contra<Rectange>(MethodForChild1);
    }
```

测试代码：

```
    MyDelegate2<Sharp> sharp2 = new MyDelegate2<Sharp>(MethodForParent3);
    MyDelegate2<Rectange> rect2 = new MyDelegate2<Rectange>(MethodForChild3);
    rect2 = sharp2;
```

编译通过。

接下来考虑第三个委托：

```
    public delegate T MyDelegate3<T>(T param);
```

首先，对类型参数T进行协变：

```
    public delegate T MyDelegate3<out T>(T param);
```

对应的方法及测试代码：

```
    public static Sharp MethodForParent4(Sharp param)
    {
        return new Sharp();
    }
    public static Rectange MethodForChild4(Rectange param)
    {
        return new Rectange();
    }

    MyDelegate3<Sharp> sharp3 = new MyDelegate3<Sharp>(MethodForParent4);
    MyDelegate3<Rectange> rect3 = new MyDelegate3<Rectange>(MethodForChild4);
    sharp3 = rect3;
```

和泛型接口类似，这里的委托类型参数T被同时用作方法返回类型和方法参数类型，不管修饰符改成in或out,编译都无法通过。所以如果用out修饰T，那么方法参数param的参数类型T就需借助一样东西来转换一下：一个对类型参数T能抗变的泛型委托。即：

```
1.      public delegate T MyDelegate3<out T>(Contra<T> param);
```

两个方法也需对应着修改：

```
1.      public static Sharp MethodForParent4(Contra<Sharp> param)
2.      {
3.          return new Sharp();
4.      }
5.      public static Rectange MethodForChild4(Contra<Rectange> param)
6.      {
7.          return new Rectange();
8.      }
```

这就是上面所说的方法参数的协变-抗变互换原则

同理，如果对该委托类型参数T进行抗变，那么根据方法返回类型协变-抗变一致原则，方法返回参数也是要借助一个对类型参数能抗变的泛型委托：

```
1.      public delegate Contra<T> MyDelegate3<in T>(T param);
```

两个方法也需对应着修改为：

```
1.      public static Contra<Sharp> MethodForParent4(Sharp param)
2.      {
3.          return new Contra<Sharp>(MethodForParent1);
4.      }
5.      public static Contra<Rectange> MethodForChild4(Rectange param)
6.      {
7.          return new Contra<Rectange>(MethodForChild1);
8.      }
```

推广到一般的泛型委托：

```
1.      public delegate T1 MyDelegate4<T1,T2,T3>(T2 param1,T3 param2);
```

可能三个参数T1，T2，T3会有各自的抗变和协变，如：

```
1.      public delegate T1 MyDelegate4<out T1,in T2,in T3>(T2 param1,T3 param2);
```

这是一种最理想的情况，T1支持协变，用于方法返回值；T2，T3支持抗变，用于方法参数。

但是如果变成：

```
1.      public delegate T1 MyDelegate4<in T1,out T2,in T3>(T2 param1,T3 param2);
```

那么对应的T1，T2类型参数就会出问题，原因上面都已经分析过了。于是就需要修改T1对应的方法返回类型，T2对应的方法参数类型，如何修改？只要根据上面提到的：1）方法返回类型的协变-抗变一致原则；2）方法参数类型的协变-抗变互换原则！

对应本篇的例子，就可以修改成：

```
1.      public delegate Contra<T1> MyDelegate4<in T1, out T2, in T3>(Contra<T2>
    param1, T3 param2);
```

四、C#中的范型接口与泛型委托

```
1.      interface IEnumerable<out T> : IEnumerable
2.      {
3.          IEnumerator<T> GetEnumerator();
4.      }
5.
6.      public interface IEnumerator<out T> : IEnumerator, IDisposable
7.      {
8.          T Current
9.          {
10.             get;
11.         }
12.     }
13.
        public interface IQueryable<out T> : IEnumerable<T>, IEnumerable,
14.  IQueryable
15.     {
16.     }
17.
18.     public interface IQueryable : IEnumerable
19.     {
20.         //
21.         // Properties
22.         //
```

```
23.         Type ElementType
24.         {
25.             get;
26.         }
27.
28.         Expression Expression
29.         {
30.             get;
31.         }
32.
33.         IQueryProvider Provider
34.         {
35.             get;
36.         }
37.     }
38.
        public interface IGrouping<out TKey, out TElement> : IEnumerable<TElement>,
39. IEnumerable
40.     {
41.         TKey Key
42.         {
43.             get;
44.         }
45.     }
46.
47.     public interface IComparer<in T>
48.     {
49.         int Compare(T x, T y);
50.     }
51.
52.     public interface IEqualityComparer<in T>
53.     {
54.         bool Equals(T x, T y);
55.
56.         int GetHashCode(T obj);
57.     }
58.
59.     public interface IComparable<in T>
60.     {
61.         int CompareTo(T other);
62.     }
```

C#中的范型委托

```
1.      public delegate void Action<in T>(T obj);
2.      public delegate TResult Func<in T, out TResult>(T arg);
3.      public delegate bool Predicate<in T>(T obj);
4.      public delegate int Comparison<in T>(T x, T y);
5.      public delegate TOutput Converter<in TInput, out TOutput>(TInput input);
```