C#

C# 一些重要的功能：

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

Hello World第一个实例

```
using System;
//  命名空间引用
namespace HelloWorldApplication     //  当前程序所处于的命名空间
{
   class HelloWorld                 //  类名
   {
      static void Main(string[] args)   //  方法名称或称为函数
      {
         /* 我的第一个 C# 程序*/
         Console.WriteLine("Hello World");  //  方法体
         Console.ReadKey();
      }
   }
}
```

C#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	-9,223,372,036,854,775,808 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

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。

字符串类型

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：

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

等价于：

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

类型转换

隐式类型转换

隐式转换是不需要编写代码来指定的转换，编译器会自动进行。

隐式转换是指将一个较小范围的数据类型转换为较大范围的数据类型时，编译器会自动完成类型转换，这些转换是 C# 默认的以安全方式进行的转换, 不会导致数据丢失。

例如，从 int 到 long，从 float 到 double 等。

从小的整数类型转换为大的整数类型，从派生类转换为基类。将一个 byte 类型的变量赋值给 int 类型的变量，编译器会自动将 byte 类型转换为 int 类型，不需要显示转换。

实例

```
byte b = 10;
int i = b; // 隐式转换，不需要显式转换
```

将一个整数赋值给一个长整数，或者将一个浮点数赋值给一个双精度浮点数，这种转换不会导致数据丢失:

实例

```
int intValue = 42;
long longValue = intValue; // 隐式转换，从 int 到 long
```

显式转换

显式类型转换，即强制类型转换，需要程序员在代码中明确指定。

显式转换是指将一个较大范围的数据类型转换为较小范围的数据类型时，或者将一个对象类型转换为另一个对象类型时，需要使用强制类型转换符号进行显示转换，强制转换会造成数据丢失。

int 类型的变量赋值给 byte 类型的变量实例

```
int i = 10;
byte b = (byte)i; // 显式转换，需要使用强制类型转换符号
```

强制转换为整数类型实例:

```
double doubleValue = 3.14;
int intValue = (int)doubleValue; // 强制从 double 到 int，数据可能损失小数部分
```

强制转换为浮点数类型实例

```
int intValue = 42;
float floatValue = (float)intValue; // 强制从 int 到 float，数据可能损失精度
```

强制转换为字符串类型实例

```
int intValue = 123;
string stringValue = intValue.ToString(); // 将 int 转换为字符串
```

下面的实例显示了一个显式的类型转换：

实例

```
using System;

namespace TypeConversionApplication
{
  class ExplicitConversion
  {
    static void Main(string[] args)
    {
      double d = 5673.74;
      int i;

      // 强制转换 double 为 int
      i = (int)d;
      Console.WriteLine(i);
      Console.ReadKey();

    }
  }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
5673
```

C# 类型转换方法

序号	方法 & 描述
1	**ToBoolean** 如果可能的话，把类型转换为布尔型。
2	**ToByte** 把类型转换为字节类型。
3	**ToChar** 如果可能的话，把类型转换为单个 Unicode 字符类型。
4	**ToDateTime** 把类型（整数或字符串类型）转换为 日期-时间 结构。
5	**ToDecimal** 把浮点型或整数类型转换为十进制类型。
6	**ToDouble** 把类型转换为双精度浮点型。
7	**ToInt16** 把类型转换为 16 位整数类型。

序号	方法 & 描述
8	**ToInt32** 把类型转换为 32 位整数类型。
9	**ToInt64** 把类型转换为 64 位整数类型。
10	**ToSbyte** 把类型转换为有符号字节类型。
11	**ToSingle** 把类型转换为小浮点数类型。
12	**ToString** 把类型转换为字符串类型。
13	**ToType** 把类型转换为指定类型。
14	**ToUInt16** 把类型转换为 16 位无符号整数类型。
15	**ToUInt32** 把类型转换为 32 位无符号整数类型。
16	**ToUInt64** 把类型转换为 64 位无符号整数类型。

这些方法都定义在 System.Convert 类中，使用时需要包含 System 命名空间。它们提供了一种安全的方式来执行类型转换，因为它们可以处理 null值，并且会抛出异常，如果转换不可能进行。

例如，使用 Convert.ToInt32 方法将字符串转换为整数：

```
string str = "123";
int number = Convert.ToInt32(str); // 转换成功，number为123
```

总结：C# 内置类型转换方法的表格：

方法类别	方法	描述
隐式转换	自动进行的转换	无需显式指定，通常用于安全的类型转换，如从较小类型到较大类型
显式转换（强制转换）	`(type)value`	需要显式指定，通常用于可能导致数据丢失或转换失败的情况
`Convert` 类方法	`Convert.ToBoolean(value)`	将指定类型转换为 `Boolean`
	`Convert.ToByte(value)`	将指定类型转换为 `Byte`
	`Convert.ToChar(value)`	将指定类型转换为 `Char`
	`Convert.ToDateTime(value)`	将指定类型转换为 `DateTime`
	`Convert.ToDecimal(value)`	将指定类型转换为 `Decimal`
	`Convert.ToDouble(value)`	将指定类型转换为 `Double`
	`Convert.ToInt16(value)`	将指定类型转换为 `Int16`（短整型）
	`Convert.ToInt32(value)`	将指定类型转换为 `Int32`（整型）

方法类别	方法	描述
	`Convert.ToInt64(value)`	将指定类型转换为 `Int64`（长整型）
	`Convert.ToSByte(value)`	将指定类型转换为 `SByte`
	`Convert.ToSingle(value)`	将指定类型转换为 `Single`（单精度浮点型）
	`Convert.ToString(value)`	将指定类型转换为 `String`
	`Convert.ToUInt16(value)`	将指定类型转换为 `UInt16`（无符号短整型）
	`Convert.ToUInt32(value)`	将指定类型转换为 `UInt32`（无符号整型）
	`Convert.ToUInt64(value)`	将指定类型转换为 `UInt64`（无符号长整型）
`Parse` 方法	`Boolean.Parse(string)`	将字符串解析为 `Boolean`
	`Byte.Parse(string)`	将字符串解析为 `Byte`
	`Char.Parse(string)`	将字符串解析为 `Char`
	`DateTime.Parse(string)`	将字符串解析为 `DateTime`
	`Decimal.Parse(string)`	将字符串解析为 `Decimal`
	`Double.Parse(string)`	将字符串解析为 `Double`
	`Int16.Parse(string)`	将字符串解析为 `Int16`
	`Int32.Parse(string)`	将字符串解析为 `Int32`
	`Int64.Parse(string)`	将字符串解析为 `Int64`
	`SByte.Parse(string)`	将字符串解析为 `SByte`
	`Single.Parse(string)`	将字符串解析为 `Single`
	`UInt16.Parse(string)`	将字符串解析为 `UInt16`
	`UInt32.Parse(string)`	将字符串解析为 `UInt32`
	`UInt64.Parse(string)`	将字符串解析为 `UInt64`
`TryParse` 方法	`Boolean.TryParse(string, out bool)`	尝试将字符串解析为 `Boolean`，返回布尔值表示是否成功
	`Byte.TryParse(string, out byte)`	尝试将字符串解析为 `Byte`，返回布尔值表示是否成功
	`Char.TryParse(string, out char)`	尝试将字符串解析为 `Char`，返回布尔值表示是否成功

方法类别	方法	描述
	`DateTime.TryParse(string, out DateTime)`	尝试将字符串解析为 `DateTime`，返回布尔值表示是否成功
	`Decimal.TryParse(string, out decimal)`	尝试将字符串解析为 `Decimal`，返回布尔值表示是否成功
	`Double.TryParse(string, out double)`	尝试将字符串解析为 `Double`，返回布尔值表示是否成功
	`Int16.TryParse(string, out short)`	尝试将字符串解析为 `Int16`，返回布尔值表示是否成功
	`Int32.TryParse(string, out int)`	尝试将字符串解析为 `Int32`，返回布尔值表示是否成功
	`Int64.TryParse(string, out long)`	尝试将字符串解析为 `Int64`，返回布尔值表示是否成功
	`SByte.TryParse(string, out sbyte)`	尝试将字符串解析为 `SByte`，返回布尔值表示是否成功
	`Single.TryParse(string, out float)`	尝试将字符串解析为 `Single`，返回布尔值表示是否成功
	`UInt16.TryParse(string, out ushort)`	尝试将字符串解析为 `UInt16`，返回布尔值表示是否成功
	`UInt32.TryParse(string, out uint)`	尝试将字符串解析为 `UInt32`，返回布尔值表示是否成功
	`UInt64.TryParse(string, out ulong)`	尝试将字符串解析为 `UInt64`，返回布尔值表示是否成功

C# 变量

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

C# 4.0引入了**动态类型 (dynamic)**

```
dynamic dynamicVariable = "This can be any type";   //可以是任意类型
```

变量的命名规则

在 C# 中，变量的命名需要遵循一些规则：

- 变量名可以包含字母、数字和下划线。
- 变量名必须以字母或下划线开头。
- 变量名区分大小写。
- 避免使用 C# 的关键字作为变量名。

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：

```
int num;
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。

字符常量

常量是固定值，程序执行期间不会改变。

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。

转义代码

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）

转义序列	含义
\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

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;
```

运算符

算术运算符

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

关系运算符

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。

运算符	描述	实例
>=	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。

逻辑运算符

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。
||	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 || 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。

位运算符

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

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。	(A & B) 将得到 12，即为 0000 1100
|	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 | 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。

运算符	描述	实例
<<	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

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
|=	按位或且赋值运算符	C |= 2 等同于 C = C | 2

其他运算符

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。

运算符	描述	实例
? :	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;

C# 中的运算符优先级

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	|	从左到右
逻辑与 AND	&&	从左到右
逻辑或 OR	||	从左到右
条件	?:	从右到左
赋值	= += -= *= /= %=>>= <<= &= ^= |=	从右到左
逗号	,	从左到右

C# 判断判断语句

C# 提供了以下类型的判断语句。点击链接查看每个语句的细节。

语句	描述
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。
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。
嵌套 if 语句	您可以在一个 **if** 或 **else if** 语句内使用另一个 **if** 或 **else if** 语句。

语句	描述
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。
嵌套 switch 语句	您可以在一个 **switch** 语句内使用另一个 **switch** 语句。

? : 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;
```

C# 循环

循环类型

C# 提供了以下几种循环类型。点击链接查看每个类型的细节。

循环类型	描述
while 循环	当给定条件为真时，重复语句或语句组。它会在执行循环主体之前测试条件。
for/foreach 循环	多次执行一个语句序列，简化管理循环变量的代码。
do...while 循环	除了它是在循环主体结尾测试条件外，其他与 while 语句类似。
嵌套循环	您可以在 while、for 或 do..while 循环内使用一个或多个循环。

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。点击链接查看每个语句的细节。

控制语句	描述
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	跳过本轮循环，开始下一轮循环。

C# 封装

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；**

  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。

- **private：对象本身在对象内部可以访问；**

  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。

- **protected：只有该类对象及其子类对象可以访问**

  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

- **internal：同一个程序集的对象可以访问；**

Internal 访问修饰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。

- **protected internal：访问限于当前程序集或派生自包含类的类型。**

  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。

总结

比如说：一个人A为父类，他的儿子B，妻子C，私生子D（注：D不在他家里）

如果我们给A的事情增加修饰符：

- public事件，地球人都知道，全公开
- protected事件，A，B，D知道（A和他的所有儿子知道，妻子C不知道）
- private事件，只有A知道（隐私？心事？）
- internal事件，A，B，C知道（A家里人都知道，私生子D不知道）
- protected internal事件，A，B，C，D都知道,其它人不知道

C# 方法

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：

```
<Access Specifier> <Return Type> <Method Name>(Parameter List)
{
   Method Body
}
```

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。

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。
引用参数	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。
输出参数	这种方式可以返回多个值。

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：

实例

```
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;

      Console.WriteLine("在交换之前，a 的值： {0}", a);
      Console.WriteLine("在交换之前，b 的值： {0}", b);

      //调用函数来交换值
      n.swap(ref a, ref b);

      Console.WriteLine("在交换之后，a 的值： {0}", a);
      Console.WriteLine("在交换之后，b 的值： {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：

```
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：200
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。

C# 可空类型（Nullable）

C# 单问号 ? 与 双问号 ??

? 单问号用于对 **int、double、bool** 等无法直接赋值为 null 的数据类型进行 null 的赋值，意思是这个数据类型是 Nullable 类型的。

```
int? i = 3;
```

等同于：

```
Nullable<int> i = new Nullable<int>(3);
int i; //默认值0
int? ii; //默认值null
```

?? 双问号用于判断一个变量在为 null 的时候返回一个指定的值。

C# 可空类型（Nullable）

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：

实例

```
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：

实例

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

      static void Main(string[] args)
      {

         double? num1 = null;
         double? num2 = 3.14157;
         double num3;
         num3 = num1 ?? 5.34;      // num1 如果为空值则返回 5.34
         Console.WriteLine("num3 的值： {0}", num3);
         num3 = num2 ?? 5.34;
         Console.WriteLine("num3 的值： {0}", num3);
         Console.ReadLine();

      }
   }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
num3 的值： 5.34
num3 的值： 3.14157
```

C# 数组（Array）

声明数组

在 C# 中声明一个数组，您可以使用下面的语法：

```
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：

```
double[] balance;
```

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：

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

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：

```
double[] balance = new double[10];
balance[0] = 4500.0;
```

您可以在声明数组的同时给数组赋值，比如：

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

您也可以创建并初始化一个数组，比如：

```
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：

```
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：

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

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：

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

```
/* 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);
         }
```

```
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
```

C# 数组细节

在 C# 中，数组是非常重要的，且需要了解更多的细节。下面列出了 C# 程序员必须清楚的一些与数组相关的重要概念：

概念	描述
多维数组	C# 支持多维数组。多维数组最简单的形式是二维数组。
交错数组	C# 支持交错数组，即数组的数组。
传递数组给函数	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。
参数数组	这通常用于传递未知数量的参数给函数。
Array 类	在 System 命名空间中定义，是所有数组的基类，并提供了各种用于数组的属性和方法。

C# 字符串（String）

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v

```
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() ;
        }
    }
}
```

输出

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

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。

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。

序号	方法名称 & 描述
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。

序号	方法名称 & 描述
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。

C# 结构体（Struct）

在 C# 中，结构体（struct）是一种值类型（value type），用于组织和存储相关数据。

在 C# 中，结构体是值类型数据结构，这样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

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;
};
```

结构用法:

```
using System;
using System.Text;

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

      /* 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();

   }
}
```

输出：

```
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
```

C# 结构的特点

结构提供了一种轻量级的数据类型，适用于表示简单的数据结构，具有较好的性能特性和值语义：

- 结构可带有方法、字段、索引、属性、运算符方法和事件，适用于表示轻量级数据的情况，如坐标、范围、日期、时间等。
- 结构可定义构造函数，但不能定义析构函数。但是，您不能为结构定义无参构造函数。无参构造函数(默认)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。
- 结构变量通常分配在栈上，这使得它们的创建和销毁速度更快。但是，如果将结构用作类的字段，且这个类是引用类型，那么结构将存储在堆上。
- 结构默认情况下是可变的，这意味着你可以修改它们的字段。但是，如果结构定义为只读，那么它的字段将是不可变的。

类 vs 结构

类和结构在设计和使用时有不同的考虑因素，类适合表示复杂的对象和行为，支持继承和多态性，而结构则更适合表示轻量级数据和值类型，以提高性能并避免引用的管理开销。

类和结构有以下几个基本的不同点：

值类型 vs 引用类型：

- **结构是值类型（Value Type）：** 结构是值类型，它们在栈上分配内存，而不是在堆上。当将结构实例传递给方法或赋值给另一个变量时，将复制整个结构的内容。
- **类是引用类型（Reference Type）：** 类是引用类型，它们在堆上分配内存。当将类实例传递给方法或赋值给另一个变量时，实际上是传递引用（内存地址）而不是整个对象的副本。

继承和多态性：

- **结构不能继承：** 结构不能继承其他结构或类，也不能作为其他结构或类的基类。
- **类支持继承：** 类支持继承和多态性，可以通过派生新类来扩展现有类的功能。

默认构造函数：

- **结构不能有无参数的构造函数：** 结构不能包含无参数的构造函数。每个结构都必须有至少一个有参数的构造函数。
- **类可以有无参数的构造函数：** 类可以包含无参数的构造函数，如果没有提供构造函数，系统会提供默认的无参数构造函数。

赋值行为：

- 类型为类的变量在赋值时存储的是引用，因此两个变量指向同一个对象。
- 结构变量在赋值时会复制整个结构，因此每个变量都有自己的独立副本。

传递方式：

- 类型为类的对象在方法调用时通过引用传递，这意味着在方法中对对象所做的更改会影响到原始对象。
- 结构对象通常通过值传递，这意味着传递的是结构的副本，而不是原始结构对象本身。因此，在方法中对结构所做的更改不会影响到原始对象。

可空性：

- **结构体是值类型，不能直接设置为 *null*：** 因为 **null** 是引用类型的默认值，而不是值类型的默认值。如果你需要表示结构体变量的缺失或无效状态，可以使用 **Nullable** 或称为 **T?** 的可空类型。
- **类默认可为null：** 类的实例默认可以为 `null`，因为它们是引用类型。

性能和内存分配：

- **结构通常更轻量：** 由于结构是值类型且在栈上分配内存，它们通常比类更轻量，适用于简单的数据表示。
- **类可能有更多开销：** 由于类是引用类型，可能涉及更多的内存开销和管理。

代码区别

注释部分演示了结构不能包含无参数的构造函数、不能继承以及结构的实例复制是复制整个结构的内容。与之相反，类可以包含无参数的构造函数，可以继承，并且实例复制是复制引用。

```
using System;

// 结构声明
struct MyStruct
{
    public int X;
    public int Y;

    // 结构不能有无参数的构造函数
    // public MyStruct()
    // {
    // }
```

```
    // 有参数的构造函数
    public MyStruct(int x, int y)
    {
        X = x;
        Y = y;
    }

    // 结构不能继承
    // struct MyDerivedStruct : MyBaseStruct
    // {
    // }
}

// 类声明
class MyClass
{
    public int X;
    public int Y;

    // 类可以有无参数的构造函数
    public MyClass()
    {
    }

    // 有参数的构造函数
    public MyClass(int x, int y)
    {
        X = x;
        Y = y;
    }

    // 类支持继承
    // class MyDerivedClass : MyBaseClass
    // {
    // }
}

class Program
{
    static void Main()
    {
        // 结构是值类型，分配在栈上
        MyStruct structInstance1 = new MyStruct(1, 2);
        MyStruct structInstance2 = structInstance1; // 复制整个结构

        // 类是引用类型，分配在堆上
        MyClass classInstance1 = new MyClass(3, 4);
        MyClass classInstance2 = classInstance1; // 复制引用，指向同一个对象

        // 修改结构实例不影响其他实例
        structInstance1.X = 5;
        Console.WriteLine($"Struct: {structInstance1.X}, {structInstance2.X}");

        // 修改类实例会影响其他实例
        classInstance1.X = 6;
```

```
        Console.WriteLine($"Class: {classInstance1.X}, {classInstance2.X}");
    }
}
```

针对上述讨论，让我们重写前面的实例：

```
using System;
using System.Text;

//  结构
struct Books
{
   private string title;
   private string author;
   private string subject;
   private int book_id;
   public void setValues(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 Books */
      Books Book2 = new Books(); /* 声明 Book2，类型为 Books */

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

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

      /* 打印 Book1 信息 */
      Book1.display();

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

```
      Console.ReadKey();

   }
}
```

执行后：

```
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（Enum）

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# 枚举是值类型。换句话说，枚举包含自己的值，且不能继承或传递继承。

声明 *enum* 变量

声明枚举的一般语法：

```
enum <enum_name>
{
    enumeration list
};
```

其中，

- *enum_name* 指定枚举的类型名称。
- *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：

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

C# 类（Class）

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。

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;
      }

      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());
      }
   }
}
```

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。

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。

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

   // 基类 PaintCost
   public interface PaintCost       //   接口
   {
      int getCost(int area);
```

```
    }
    // 派生类
    class Rectangle : Shape, PaintCost
    {
       public int getArea()
       {
          return (width * height);
       }
       public int getCost(int area)
       {
          return area * 70;
       }
    }
    class RectangleTester
    {
       static void Main(string[] args)
       {
          Rectangle Rect = new Rectangle();
          int area;
          Rect.setWidth(5);
          Rect.setHeight(7);
          area = Rect.getArea();
          // 打印对象的面积
          Console.WriteLine("总面积:  {0}",  Rect.getArea());
          Console.WriteLine("油漆总成本:  ${0}" , Rect.getCost(area));
          Console.ReadKey();
       }
    }
}
```

输出

```
总面积:  35
油漆总成本:  $2450
```

C# 多态性

多态是同一个行为具有多个不同表现形式或形态的能力。

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。

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：

- 函数重载
- 运算符重载

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。不同重载只有返回类型不同的函数声明。

```
using System;
namespace PolymorphismApplication
{
    public class TestData
    {
        public int Add(int a, int b, int c)
        {
            return a + b + c;
        }
        public int Add(int a, int b)
        {
            return a + b;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            TestData dataClass = new TestData();
            int add1 = dataClass.Add(1, 2);
            int add2 = dataClass.Add(1, 2, 3);

            Console.WriteLine("add1 :" + add1);
            Console.WriteLine("add2 :" + add2);
        }
    }
}
```

C# 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：

```
public static Box operator+ (Box b, Box c)
{
   Box box = new Box();
   box.length = b.length + c.length;
   box.breadth = b.breadth + c.breadth;
   box.height = b.height + c.height;
   return box;
}
```

operator关键字

operator 关键字用于在类或结构声明中声明运算符。运算符声明可以采用下列四种形式之一：

```
public static result-type operator unary-operator ( op-type operand )
public static result-type operator binary-operator ( op-type operand, op-type2
operand2 )
public static implicit operator conv-type-out ( conv-type-in operand )
public static explicit operator conv-type-out ( conv-type-in operand )
```

参数：

- result-type 运算符的结果类型。
- unary-operator 下列运算符之一：+ - ! ~ ++ — true false
- op-type 第一个（或唯一一个）参数的类型。
- operand 第一个（或唯一一个）参数的名称。
- binary-operator 其中一个：+ - * / % & | ^ << >> == != > < >= <=
- op-type2 第二个参数的类型。
- operand2 第二个参数的名称。
- conv-type-out 类型转换运算符的目标类型。
- conv-type-in 类型转换运算符的输入类型。

注意：

前两种形式声明了用户定义的重载内置运算符的运算符。并非所有内置运算符都可以被重载（请参见可重载的运算符）。op-type 和 op-type2 中至少有一个必须是封闭类型（即运算符所属的类型，或理解为自定义的类型）。例如，这将防止重定义整数加法运算符。

后两种形式声明了转换运算符。conv-type-in 和 conv-type-out 中正好有一个必须是封闭类型（即，转换运算符只能从它的封闭类型转换为其他某个类型，或从其他某个类型转换为它的封闭类型）。

运算符只能采用值参数，不能采用 ref 或 out 参数。

C# 要求成对重载比较运算符。如果重载了==，则也必须重载!=，否则产生编译错误。同时，比较运算符必须返回bool类型的值，这是与其他算术运算符的根本区别。

C# 不允许重载=运算符，但如果重载例如+运算符，编译器会自动使用+运算符的重载来执行+=运算符的操作。

运算符重载的其实就是函数重载。首先通过指定的运算表达式调用对应的运算符函数，然后再将运算对象转化为运算符函数的实参，接着根据实参的类型来确定需要调用的函数的重载，这个过程是由编译器完成。

任何运算符声明的前面都可以有一个可选的属性（C# 编程指南）列表。

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

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

```
        {
            Student s1 = new Student(20, "Tom");
            Student s2 = new Student(18, "Jack");
            Student s3 = s1 + s2;

            s3.sayPlus();
            (s1 - s2).sayMinus();
            Console.ReadKey();
        }
    }
    public class Student
    {
        public Student() { }
        public Student(int age, string name)
        {
            this.name = name;
            this.age = age;

        }
        private string name;
        private int age;

        public void sayPlus()
        {
            System.Console.WriteLine("{0} 年龄之和为：{1}", this.name, this.age);

        }
        public void sayMinus() {
            System.Console.WriteLine("{0} 年龄之差为：{1}", this.name, this.age);
        }
        //覆盖“+”操作符
        public static Student operator +(Student s1, Student s2)
        {
            return new Student(s1.age + s2.age, s1.name + " And " + s2.name);

        }
        //覆盖“-”操作符
        public static Student operator -(Student s1, Student s2) {
            return new Student(Math.Abs(s1.age - s2.age), s1.name + "And" +
s2.name);
        }
   }
}
```

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。

运算符	描述
+, -, *, /, %	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, sizeof, typeof	这些运算符不能被重载。

实例：

```
using System;

namespace OperatorOvlApplication
{
    class Box
    {
       private double length;      // 长度
       private double breadth;     // 宽度
       private double height;      // 高度

       public double getVolume()
       {
         return length * breadth * height;
       }
      public void setLength( double len )
      {
          length = len;
      }

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

      public void setHeight( double hei )
      {
          height = hei;
      }
      // 重载 + 运算符来把两个 Box 对象相加
      public static Box operator+ (Box b, Box c)
      {
          Box box = new Box();
          box.length = b.length + c.length;
          box.breadth = b.breadth + c.breadth;
          box.height = b.height + c.height;
          return box;
      }

      public static bool operator == (Box lhs, Box rhs)
      {
          bool status = false;
```

```
        if (lhs.length == rhs.length && lhs.height == rhs.height
           && lhs.breadth == rhs.breadth)
        {
            status = true;
        }
        return status;
    }
    public static bool operator !=(Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length != rhs.length || lhs.height != rhs.height
            || lhs.breadth != rhs.breadth)
        {
            status = true;
        }
        return status;
    }
    public static bool operator <(Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length < rhs.length && lhs.height
            < rhs.height && lhs.breadth < rhs.breadth)
        {
            status = true;
        }
        return status;
    }

    public static bool operator >(Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length > rhs.length && lhs.height
            > rhs.height && lhs.breadth > rhs.breadth)
        {
            status = true;
        }
        return status;
    }

    public static bool operator <=(Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length <= rhs.length && lhs.height
            <= rhs.height && lhs.breadth <= rhs.breadth)
        {
            status = true;
        }
        return status;
    }

    public static bool operator >=(Box lhs, Box rhs)
    {
        bool status = false;
        if (lhs.length >= rhs.length && lhs.height
           >= rhs.height && lhs.breadth >= rhs.breadth)
        {
```

```
            status = true;
         }
         return status;
      }
      public override string ToString()
      {
         return String.Format("({0}, {1}, {2})", length, breadth, height);
      }

   }

   class Tester
   {
      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
        Box Box3 = new Box();          // 声明 Box3，类型为 Box
        Box Box4 = new Box();
        double volume = 0.0;   // 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);

       // 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();
      }
    }
}
```

输出

```
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
```

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。
- ```
  using System;
  namespace PolymorphismApplication
  {
     abstract class Shape
     {
         abstract public 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();
        }
     }
  }
  ```

  输出:

  ```
  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。

以下实例创建了 Shape 基类，并创建派生类 Circle、 Rectangle、Triangle， Shape 类提供一个名为 Draw 的虚拟方法，在每个派生类中重写该方法以绘制该类的指定形状。

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

public class Shape
{
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // 虚方法
    public virtual void Draw()
    {
        Console.WriteLine("执行基类的画图任务");
    }
}

class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("画一个圆形");
        base.Draw();
    }
}
class Rectangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("画一个长方形");
        base.Draw();
    }
}
class Triangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("画一个三角形");
        base.Draw();
    }
}

class Program
{
```

```
    static void Main(string[] args)
    {
        // 创建一个 List<Shape> 对象，并向该对象添加 Circle、Triangle 和 Rectangle
        var shapes = new List<Shape>
        {
            new Rectangle(),
            new Triangle(),
            new Circle()
        };

        // 使用 foreach 循环对该列表的派生类进行循环访问，并对其中的每个 Shape 对象调用
Draw 方法
        foreach (var shape in shapes)
        {
            shape.Draw();
        }

        Console.WriteLine("按下任意键退出。");
        Console.ReadKey();
    }

}
```

输出

```
画一个长方形
执行基类的画图任务
画一个三角形
执行基类的画图任务
画一个圆形
执行基类的画图任务
按下任意键退出。
```

C# 接口（Interface）

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。

接口本身并不实现任何功能，它只是和声明实现该接口的对象订立一个必须实现哪些行为的契约。

抽象类不能直接实例化，但允许派生出具体的，具有实际功能的类。

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：

```
interface IMyInterface
{
  void MethodToImplement();
}
```

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：

```
using System;

interface IParentInterface
{
    void ParentInterfaceMethod();
}

interface IMyInterface : IParentInterface
{
    void MethodToImplement();
}

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

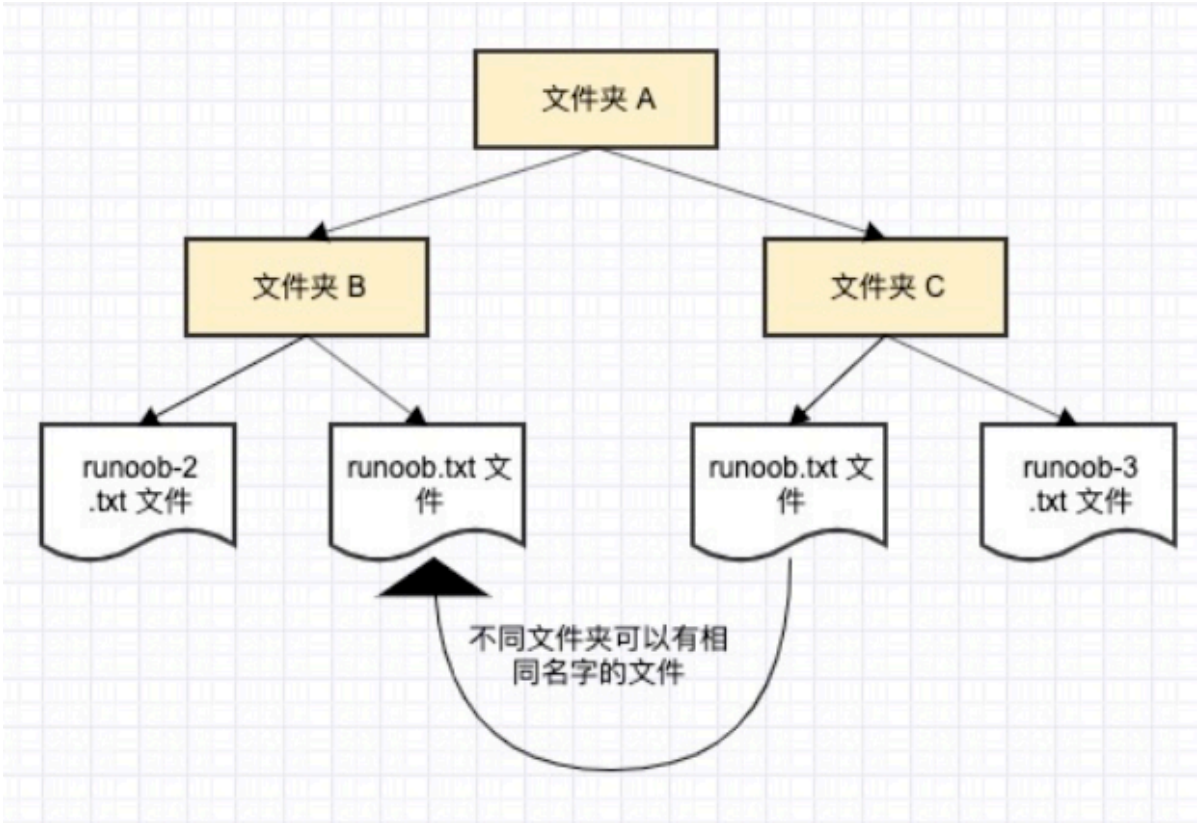
    public void MethodToImplement()
    {
        Console.WriteLine("MethodToImplement() called.");
    }

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

C# 命名空间（Namespace）

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。

我们举一个计算机系统中的例子，一个文件夹(目录)中可以包含多个文件夹，每个文件夹中不能有相同的文件名，但不同文件夹中的文件可以重名。



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;
```

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：

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

我们可以写完全限定名称，如下：

```
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。

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
   }
}
```

C# 预处理器指令

预处理器指令（Preprocessor Directives）指导编译器在实际编译开始之前对信息进行预处理。

通过这些指令，可以控制编译器如何编译文件或编译哪些部分。常见的预处理器指令包括条件编译、宏定义等。

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。

C# 预处理器指令列表

指令	描述
`#define`	定义一个符号，可以用于条件编译。
`#undef`	取消定义一个符号。
`#if`	开始一个条件编译块，如果符号被定义则包含代码块。
`#elif`	如果前面的 `#if` 或 `#elif` 条件不满足，且当前条件满足，则包含代码块。
`#else`	如果前面的 `#if` 或 `#elif` 条件不满足，则包含代码块。
`#endif`	结束一个条件编译块。
`#warning`	生成编译器警告信息。
`#error`	生成编译器错误信息。

指令	描述
`#region`	标记一段代码区域，可以在IDE中折叠和展开这段代码，便于代码的组织和阅读。
`#endregion`	结束一个代码区域。
`#line`	更改编译器输出中的行号和文件名，可以用于调试或生成工具的代码。
`#pragma`	用于给编译器发送特殊指令，例如禁用或恢复特定的警告。
`#nullable`	控制可空性上下文和注释，允许启用或禁用对可空引用类型的编译器检查。

#define 和 #undef 预处理器

#define 用于定义符号（通常用于条件编译），**#undef** 用于取消定义符号。

```
#define DEBUG

#undef RELEASE
```

#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：

```
#define symbol
```

条件指令：#if, #elif, #else 和 #endif

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。

```
#define DEBUG

#if DEBUG
    Console.WriteLine("Debug mode");
#elif RELEASE
    Console.WriteLine("Release mode");
#else
    Console.WriteLine("Other mode");
#endif
```

#warning 和 #error

#warning 用于生成编译器警告，#error 用于生成编译器错误。

```
#warning This is a warning message
#error This is an error message
```

#region 和 #endregion

用于代码折叠，使代码更加可读。

```
#region MyRegion
    // Your code here
#endregion
```

#line

用于更改文件行号和文件名的编译器输出。

```
#line 100 "MyFile.cs"
    // The next line will be reported as line 100 in MyFile.cs
    Console.WriteLine("This is line 100");
#line default
    // Line numbering returns to normal
```

#pragma

用于向编译器发送特殊指令。最常见的用法是禁用特定的警告。

```
#pragma warning disable 414
    private int unusedVariable;
#pragma warning restore 414
```

使用预处理器指令的注意事项

- **提高代码可读性**：使用`#region`可以帮助分隔代码块，提高代码的组织性。
- **条件编译**：通过`#if`等指令可以在开发和生产环境中编译不同的代码，方便调试和发布。
- **警告和错误**：通过`#warning`和`#error`可以在编译时提示开发人员注意特定问题。

通过正确使用这些预处理器指令，可以更好地控制代码的编译过程，提高代码的灵活性和可维护性。

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

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。

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\nHello\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\nHello\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"
\x nn	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)
\u nnnn	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"

转义字符	描述	模式	匹配
**	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) * 39" *中的* "2+2" *和* "39"

字符类

字符类与一组字符中的任何一个字符匹配。

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。	[b-d]	[b-d]irds 可以匹配 Birds、 Cirds、 Dirds
.	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"

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。

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"。

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。

分组构造	描述	模式	匹配
(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	"deep" 中的 "ee"
(?< name >subexpression)	将匹配的子表达式捕获到一个命名组中。	(?<double>\w)\k<double>	"deep" 中的 "ee"
(?< name1 -name2 >subexpression)	定义平衡组定义。	(((?'Open'()[^\(\)])+((?'Close-Open'))[^\(\)])+)*(?(Open)(?!))$	"3+2^((1-3)*(3-1))" 中的 "((1-3)(3-1))"
(?: subexpression)	定义非捕获组。	Write(?:Line)?	"Console.WriteLine()" 中的 "WriteLine"
(?imnsx-imnsx:subexpression)	应用或禁用 *subexpression* 中指定的选项。	A\d{2}(?i:\w+)\b	"A12xl A12XL a12xl" 中的 "A12xl" 和 "A12XL"

分组构造	描述	模式	匹配
(?= subexpression)	零宽度正预测先行断言。	\w+(?=.)	"He is. The dog ran. The sun is out." 中的 "is"、 "ran" 和 "out"
(?! subexpression)	零宽度负预测先行断言。	\b(?!un)\w+\b	"unsure sure unity used" 中的 "sure" 和 "used"
(?<=subexpression)	零宽度正回顾后发断言。	(?<=19)\d{2}\b	"1851 1999 1950 1905 2003" 中的 "99"、"50"和 "05"
(?<! subexpression)	零宽度负回顾后发断言。	(?<!wo)man\b	"Hi woman Hi man" 中的 "man"
(?> subexpression)	非回溯（也称为"贪婪"）子表达式。	13579	"1ABB 3ABBC 5AB 5AC" 中的 "1ABB"、 "3ABB" 和 "5AB"

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。

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"

限定符	描述	模式	匹配
{ 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"

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。

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"

备用构造

备用构造用于修改正则表达式以启用 either/or 匹配。

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 day. " 中的 "the" 和 "this"
(?(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"

替换

替换是替换模式中使用的正则表达式。

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。	B+	$`	"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"

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

Regex 类

Regex 类用于表示一个正则表达式。

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。

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

```
try
{
   // 引起异常的语句
}
catch( ExceptionName e1 )
{
   // 错误处理代码
}
catch( ExceptionName e2 )
{
   // 错误处理代码
}
catch( ExceptionName eN )
```

```
{
   // 错误处理代码
}
finally
{
   // 要执行的语句
}
```

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。

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。

C#文件的输入与输出

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）。

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。

I/O 类	描述
BinaryReader	从二进制流读取原始数据。

I/O 类	描述
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。

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。

```
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();
        }
```

```
    }
}
```

输出：

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

文件属性操作

静态方法和实例化方法的区别

```
//use File class
Console.WriteLine(File.GetAttributes(filePath));
File.SetAttributes(filePath,FileAttributes.Hidden | FileAttributes.ReadOnly);
Console.WriteLine(File.GetAttributes(filePath));

//user FilInfo class
FileInfo fi = new FileInfo(filePath);
Console.WriteLine(fi.Attributes.ToString());
fi.Attributes = FileAttributes.Hidden | FileAttributes.ReadOnly; //隐藏与只读
Console.WriteLine(fi.Attributes.ToString());

//只读与系统属性，删除时会提示拒绝访问
fi.Attributes = FileAttributes.Archive;
Console.WriteLine(fi.Attributes.ToString());
```

文件路径：

文件和文件夹的路径操作都在Path类中。另外还可以用Environment类，里面包含环境和程序的信息。

```
string dirPath = @"D:\TestDir";
string filePath = @"D:\TestDir\TestFile.txt";
Console.WriteLine("<<<<<<<<<<{0}>>>>>>>>>>", "文件路径");
//获得当前路径
Console.WriteLine(Environment.CurrentDirectory);
//文件或文件夹所在目录
Console.WriteLine(Path.GetDirectoryName(filePath));     //D:\TestDir
Console.WriteLine(Path.GetDirectoryName(dirPath));      //D:\
//文件扩展名
Console.WriteLine(Path.GetExtension(filePath));         //.txt
//文件名
Console.WriteLine(Path.GetFileName(filePath));          //TestFile.txt
Console.WriteLine(Path.GetFileName(dirPath));           //TestDir
Console.WriteLine(Path.GetFileNameWithoutExtension(filePath)); //TestFile
//绝对路径
Console.WriteLine(Path.GetFullPath(filePath));          //D:\TestDir\TestFile.txt
Console.WriteLine(Path.GetFullPath(dirPath));           //D:\TestDir
//更改扩展名
Console.WriteLine(Path.ChangeExtension(filePath,
".jpg"));//D:\TestDir\TestFile.jpg
//根目录
Console.WriteLine(Path.GetPathRoot(dirPath));           //D:\
```

```
//生成路径
Console.WriteLine(Path.Combine(new string[] { @"D:\", "BaseDir", "SubDir",
"TestFile.txt" })); //D:\BaseDir\SubDir\TestFile.txt
//生成随即文件夹名或文件名
Console.WriteLine(Path.GetRandomFileName());
//创建磁盘上唯一命名的零字节的临时文件并返回该文件的完整路径
Console.WriteLine(Path.GetTempFileName());
//返回当前系统的临时文件夹的路径
Console.WriteLine(Path.GetTempPath());
//文件名中无效字符
Console.WriteLine(Path.GetInvalidFileNameChars());
//路径中无效字符
Console.WriteLine(Path.GetInvalidPathChars());
```

C# 高级文件操作

C# 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。

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。

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",
Encoding.UTF8))
                {
                    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();
        }
    }
}
```

输出：文件内所有的内容

StreamWriter 类

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

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。

序号	方法 & 描述
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。

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();
        }
    }
}
```

输出：

```
Zara Ali
Nuha Ali
```

读取中文显示乱码 使用：

读取中文的时候会显示乱码，在读取文件内容的时候使用：

```
using (StreamReader sr = new StreamReader("C:/a.txt",
Encoding.GetEncoding("GB2312")))
```

然后编译的时候会报错，无法编译。

报错的解决办法如下：

第一：

```
using System.Text;
```

第二：

在 .csproj 文件中应添加如下代码：

```
<ItemGroup>
    <PackageReference Include="System.Text.Encoding.CodePages" Version="4.4.0" />
</ItemGroup
```

第三步：

在使用 **System.Text.Encoding.GetEncoding ("GB2312")** 之前，在代码中执行：

```
System.Text.Encoding.RegisterProvider
(System.Text.CodePagesEncodingProvider.Instance);
```

注册完之后，获取 GB2312 编码对象就不会报错了，并且可以正常使用其中的函数。

C# 二进制文件的读写

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。

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。

序号	方法 & 描述
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。

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。

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。

序号	方法 & 描述
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。

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;
            bool b = true;
            string s = "I am happy";
            // 创建文件
            try
            {
                bw = new BinaryWriter(new FileStream("mydata",
                                FileMode.Create));
            }
            catch (IOException e)
            {
                Console.WriteLine(e.Message + "\n Cannot create file.");
                return;
            }
            // 写入文件
            try
            {
                bw.Write(i);
                bw.Write(d);
                bw.Write(b);
                bw.Write(s);
            }
            catch (IOException e)
            {
                Console.WriteLine(e.Message + "\n Cannot write to file.");
                return;
            }

            bw.Close();
            // 读取文件
            try
```

```
            {
                br = new BinaryReader(new FileStream("mydata",
                                FileMode.Open));
            }
            catch (IOException e)
            {
                Console.WriteLine(e.Message + "\n Cannot open file.");
                return;
            }
            try
            {
                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();
        }
    }
}
```

输出

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

C# 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。

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**。

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。

序号	属性 & 描述
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

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。**

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。

实例

```
using System;
using System.IO;

namespace WindowsFileApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            // 创建一个 DirectoryInfo 对象
            DirectoryInfo mydir = new DirectoryInfo(@"c:\Windows");

            // 获取目录中的文件以及它们的名称和大小
            FileInfo [] f = mydir.GetFiles();
            foreach (FileInfo file in f)
            {
                Console.WriteLine("File Name: {0} Size: {1}",
                                  file.Name, file.Length);
            }
            Console.ReadKey();
        }
    }
}
```

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。

例子

匹配以 'S' 开头的单词

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

namespace RegExApplication
{
   class Program
   {
      private static void showMatch(string text, string expr)
      {
         Console.WriteLine("The Expression: " + expr);
```

```
            MatchCollection mc = Regex.Matches(text, expr);
            foreach (Match m in mc)
            {
               Console.WriteLine(m);
            }
         }
         static void Main(string[] args)
         {
            string str = "A Thousand Splendid Suns";

            Console.WriteLine("Matching words that start with 'S': ");
            showMatch(str, @"\bS\S*");
            Console.ReadKey();
         }
      }
}
```

匹配以 'm' 开头以 'e' 结尾的单词：

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

namespace RegExApplication
{
   class Program
   {
      private static void showMatch(string text, string expr)
      {
         Console.WriteLine("The Expression: " + expr);
         MatchCollection mc = Regex.Matches(text, expr);
         foreach (Match m in mc)
         {
            Console.WriteLine(m);
         }
      }
      static void Main(string[] args)
      {
         string str = "make maze and manage to measure it";

         Console.WriteLine("Matching words start with 'm' and ends with 'e':");
         showMatch(str, @"\bm\S*e\b");
         Console.ReadKey();
      }
   }
}
```

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();
      }
   }
}
```

除零异常

```
using System;
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();
        }
    }
}
```

遍历指定目录下所有的文件

```
namespace Test1
{
    internal class Program
    {
        static void Main(string[] Args)
        {
            Console.WriteLine("请输入路径：");
            string path = Console.ReadLine();
            if (!Directory.Exists(path))
            {
                Console.WriteLine("路径不存在！");
                return;
            }
            Console.WriteLine();
            Console.WriteLine();
            Console.WriteLine("文件详情列表：");
            TraverseFile.SearchFile(path);
        }
    }

    class TraverseFile
    {
        public static void SearchFile(string path)
        {
            DirectoryInfo di = new DirectoryInfo(path);
            FileInfo[] fi = di.GetFiles();
            foreach (FileInfo f in fi)
            {
                Console.WriteLine($"文件名：{f.Name}，文件大小：{f.Length}");
            }
            DirectoryInfo[] dList = di.GetDirectories();
            foreach (DirectoryInfo d in dList)
            {
                SearchFile(d.FullName);
            }
        }
    }
}
```