

关于作者：Aisha Ikram

我现在在英国一家软件公司任技术带头人。我是计算机科学的硕士。我主要使用 .NET 1.1/2.0, C#, VB.NET, ASP.NET, VC++ 6, MFC, ATL, COM/DCOM, SQL Server 2000/2005 等。最近我在学习 .NET 3.x 的全部内容。我的免费源代码和文章网站是：

<http://aishai.netfirms.com/>

本文通过一系列例程以简短但全面的方式讨论了 C# 语言构造和特性，所以你仅需略览代码片刻，即可了解其概念。（注意：本文不是为 C# 宗师而写。有很多初学者的 C# 文章，这只是其中之一。）

接下来关于 C# 的讨论主题：编程结构、命名空间、数据类型、变量、运算符与表达式、枚举、语句、类与结构、修饰符、属性、接口、函数参数、数组、索引器、装箱与拆箱、委托、继承与多态。

以下主题不会进行讨论：C++ 与 C# 的共同点、诸如垃圾回收、线程、文件处理等概念、数据类型转换、异常处理、.NET 库。

编程结构

和 C++ 一样，C# 是大小写敏感的。半角分号(;)是语句分隔符。和 C++ 有所区别的是，C# 中没有单独的声明（头）和实现（CPP）文件。所有代码（类声明和实现）都放在扩展名为.cs 的单一文件中。

看看 C# 中的 Hello World 程序。

```
using System;

namespace MyNameSpace
{
    class HelloWorld
    {
        static void Main(string[] args)
        {
            Console.WriteLine ("Hello World");
        }
    }
}
```

C# 中所有内容都打包在类中，而所有的类又打包在命名空间中（正如文件存与文件夹中）。和 C++ 一样，有一个主函数作为你程序的入口点。C++ 的主函数名为 `main`，而 C# 中是大写 M 打头的 `Main`。

类块或结构定义之后没有必要再加一个半角分号。C++ 中是这样，但 C# 不要求。

命名空间

每个类都打包于一个命名空间。命名空间的概念和 C++ 完全一样，但我们在 C# 中比在 C++ 中更加频繁的使用命名空间。你可以用点（.）定界符访问命名空间中的类。上面的 Hello World 程序中，`MyNameSpace` 是其命名空间。

现在思考当你要从其他命名空间的类中访问 `HelloWorld` 类。

```
using System;
namespace AnotherNameSpace
{
    class AnotherClass
    {
        public void Func()
        {
            Console.WriteLine ("Hello World");
        }
    }
}
```

现在在你的 `HelloWorld` 类中你可以这样访问：

```
using System;
using AnotherNameSpace; // 你可以增加这条语句
namespace MyNameSpace
{
    class HelloWorld
    {
        static void Main(string[] args)
        {
            AnotherClass obj = new AnotherClass();
            obj.Func();
        }
    }
}
```

在 .NET 库中，System 是包含其他命名空间的顶层命名空间。默认情况下存在一个全局命名空间，所以在命名空间外定义的类直接进到此全局命名空间中，因而你可以不用定界符访问此类。你同样可以定义嵌套命名空间。

Using

#include 指示符被后跟命名空间名的 using 关键字代替了。正如上面的 using System。System 是最基层的命名空间，所有其他命名空间和类都包含于其中。System 命名空间中所有对象的基类是 Object。

变量

除了以下差异，C# 中的变量几乎和 C++ 中一样：

1. C# 中（不同于 C++）的变量，总是需要你在访问它们前先进行初始化，否则你将遇到编译时错误。故而，不可能访问未初始化的变量。
2. 你不能在 C# 中访问一个“挂起”指针。
3. 超出数组边界的表达式索引值同样不可访问。
4. C# 中没有全局变量或全局函数，取而代之的是通过静态函数和静态变量完成的。

数据类型

所有 C# 的类型都是从 object 类继承的。有两种数据类型：

1. 基本/内建类型
2. 用户定义类型

以下是 C# 内建类型的列表：

类型	字节	描述
byte	1	unsigned byte
sbyte	1	signed byte
short	2	signed short
ushort	2	unsigned short
int	4	signed integer
uint	4	unsigned integer
long	8	signed long
ulong	8	unsigned long
float	4	floating point number
double	8	double precision number

decimal	8	fixed precision number
string	-	Unicode string
char	-	Unicode char
bool	true, false	boolean

注意：C# 的类型范围和 C++ 不同。例如：long 在 C++ 中是 4 字节而在 C# 中是 8 字节。bool 和 string 类型均和 C++ 不同。bool 仅接受真、假而非任意整数。

用户定义类型文件包含：

1. 类 (class)
2. 结构 (struct)
3. 接口 (interface)

以下类型继承时均分配内存：

1. 值类型
2. 参考类型

值类型

值类型是在堆栈中分配的数据类型。它们包括了：

1. 除字符串，所有基本和内建类型
2. 结构
3. 枚举类型

引用类型

引用类型在堆 (heap) 中分配内存且当其不再使用时，将自动进行垃圾清理。和 C++ 要求用户显示创建 delete 运算符不一样，它们使用新运算符创建，且没有 delete 运算符。在 C# 中它们自动由垃圾回收系统回收。

引用类型包括：

1. 类
2. 接口
3. 集合类型如数组
4. 字符串

枚举

C# 中的枚举和 C++ 完全一样。通过关键字 `enum` 定义。例子：

```
enum Weekdays
{
    Saturday, Sunday, Monday, Tuesday, Wednesday, Thursday, Friday
}
```

类与结构

除了内存分配的不同外，类和结构就和 C++ 中的情况一样。类的对象在堆中分配，并使用 `new` 关键字创建。而结构是在栈（`stack`）中进行分配。C# 中的结构属于轻量级快速数据类型。当需要大型数据类型时，你应该创建类。

```
struct Date
{
    int day;
    int month;
    int year;
}

class Date
{
    int day;
    int month;
    int year;
    string weekday;
    string monthName;
    public int GetDay()
    {
        return day;
    }
    public int GetMonth()
    {
        return month;
    }
    public int GetYear()
    {
        return year;
    }
    public void SetDay(int Day)
    {

```

```

        day = Day ;
    }
    public void SetMonth(int Month)
    {
        month = Month;
    }
    public void SetYear(int Year)
    {
        year = Year;
    }
    public bool IsLeapYear()
    {
        return (year/4 == 0);
    }
    public void SetDate (int day, int month, int year)
    {
    }
    ...
}

```

属性

如果你熟悉 C++ 面向对象的方法,你一定对属性有自己的认识。对 C++ 来说,前面例子中 Date 类的属性就是 day、month 和 year,而你添加了 Get 和 Set 方法。C# 提供了一种更加便捷、简单而又直接的属性访问方式。

所以上面的类应该写成这样:

```

using System;
class Date
{
    public int Day{
        get {
            return day;
        }
        set {
            day = value;
        }
    }
    int day;

    public int Month{
        get {
            return month;
        }
    }
    int month;
}

```

```

        }
        set {
            month = value;
        }
    }
    int month;

    public int Year{
        get {
            return year;
        }
        set {
            year = value;
        }
    }
    int year;

    public bool IsLeapYear(int year)
    {
        return year%4== 0 ? true: false;
    }
    public void SetDate (int day, int month, int year)
    {
        this.day = day;
        this.month = month;
        this.year = year;
    }
}

```

这里是你的 get 和 set 属性的方法:

```

class User
{
    public static void Main()
    {
        Date date = new Date();
        date.Day = 27;
        date.Month = 6;
        date.Year = 2003;
        Console.WriteLine
            ("Date: {0}/{1}/{2}", date.Day, date.Month, date.Year);
    }
}

```

修饰符

你必须知道 C++ 中常用的 public、private 和 protected 修饰符。我将在这里讨论一些 C# 引入的新的修饰符。

readonly

readonly 修饰符仅用于修饰类的数据成员。正如其名字说的，一旦它们已经进行了写操作、直接初始化或在构造函数中对其进行了赋值，readonly 数据成员就只能对其进行读取。readonly 和 const 数据成员不同之处在于 const 要求你在声明时进行直接初始化。看下面的例程：

```
class MyClass
{
    const int constInt = 100; //直接进行
    readonly int myInt = 5; //直接进行
    readonly int myInt2;

    public MyClass()
    {
        myInt2 = 8;           //间接进行
    }
    public Func()
    {
        myInt = 7; //非法
        Console.WriteLine(myInt2.ToString());
    }
}
```

sealed

带有 sealed 修饰符的类不允许你从它继承任何类。所以如果你不想一个类被继承，你可以对该类使用 sealed 关键字。

```
sealed class CanNotbeTheParent
{
    int a = 5;
}
```

unsafe

你可以使用 unsafe 修饰符在 C# 中定义一个不安全上下文。在不安全上下文中，你可以插入不安全代码，如 C++ 的指针等。参见以下代码：

```
public unsafe MyFunction( int * pInt, double* pDouble)
{
    int* pAnotherInt = new int;
    *pAnotherInt = 10;
    pInt = pAnotherInt;
```



```

...
    *pDouble = 8.9;
}

```

接口

如果你有 COM 的思想，你马上就知道我在说什么了。接口是只包含函数签名而在子类中实现的抽象基类。在 C# 中，你可以用 `interface` 关键字声明这样的接口类。.NET 就是基于这样的接口的。C# 中你不能对类进行多重继承——这在 C++ 中是允许的。通过接口，多重继承的精髓得以实现。即你的子类可以实现多重接口。（译注：由此可以实现多重继承）

```

using System;
interface myDrawing
{
    int originx
    {
        get;
        set;
    }
    int originy
    {
        get;
        set;
    }
    void Draw(object shape);
}

class Shape: myDrawing
{
    int OriX;
    int OriY;

    public int originx
    {
        get{
            return OriX;
        }
        set{
            OriX = value;
        }
    }
    public int originy
    {

```

```

        get{
            return OriY;
        }
        set{
            OriY = value;
        }
    }
    public void Draw(object shape)
    {
        ... // 要做的事
    }

    // 类自身的方法
    public void MoveShape(int newX, int newY)
    {
        .....
    }
}

```

数组

数组在 C# 中比 C++ 中要高级很多。数组分配于堆中，所以是引用类型的。你不能访问数组边界外的元素。所以 C# 防止你引发那种 bug。同时也提供了迭代数组元素的帮助函数。foreach 是这样的迭代语句之一。C++ 和 C# 数组的语法差异在于：

1. 方括号在类型后面而不是在变量名后面
2. 创建元素使用 new 运算符
3. C# 支持一维、多维和交错数组（数组的数组）

```

int[] array = new int[10]; // int 型一维数组
for (int i = 0; i < array.Length; i++)
    array[i] = i;

```

```

int[,] array2 = new int[5,10]; // int 型二维数组
array2[1,2] = 5;

```

```

int[,,] array3 = new int[5,10,5]; // int 型三维数组
array3[0,2,4] = 9;

```

```

int[][] arrayOfarray = new int[2]; // int 型交错数组 - 数组的数组
arrayOfarray[0] = new int[4];
arrayOfarray[0] = new int[] {1,2,15};

```

索引器

索引器用于书写一个可以通过使用 [] 像数组一样直接访问集合元素的方法。你所需要的只是指定待访问实例或元素的索引。索引器的语法和类属性语法相同，除了接受作为元素索引的输入参数外。

注意：CollectionBase 是用于建立集合的库类。List 是 CollectionBase 中用于存放集合列表的受保护成员。

```
class Shapes: CollectionBase
{
    public void add(Shape shp)
    {
        List.Add(shp);
    }

    //indexer
    public Shape this[int index]
    {
        get {
            return (Shape) List[index];
        }
        set {
            List[index] = value ;
        }
    }
}
```

装箱/拆箱

装箱的思想在 C# 中是创新的。正如前面提到的，所有的数据类型，无论是内建的还是用户定义的，都是从 System 命名空间的基类 object 继承的。所以基础的或是原始的类型打包为一个对象称为装箱，相反的处理称为拆箱。

```
class Test
{
    static void Main()
    {
        int myInt = 12;
        object obj = myInt ;           // 装箱
        int myInt2 = (int) obj;        // 拆箱
    }
}
```

```
    }  
}
```

例程展示了装箱和拆箱两个过程。一个 `int` 值可以被转换为对象，并且能够再次转换回 `int`。当某种值类型的变量需要被转换为一个引用类型时，便会产生一个对象箱保存该值。拆箱则完全相反。当某个对象箱被转换回其原值类型时，该值从箱中拷贝至适当的存储空间。

函数参数

C# 中的参数有三种类型：

1. 按值传递/输入参数
2. 按引用传递/输入-输出参数
3. 输出参数

如果你有 COM 接口的思想，而且还是参数类型的，你会很容易理解 C# 的参数类型。

按值传递/输入参数

值参数的概念和 C++ 中一样。传递的值复制到了新的地方并传递给函数。

```
SetDay(5);  
...  
void SetDay(int day)  
{  
    ....  
}
```

按引用传递/输入-输出参数

C++ 中的引用参数是通过指针或引用运算符 `&` 传递的。C# 中的引用参数更不易出错。你可以传递一个引用地址，你传递一个输入的值并通过函数得到一个输出的值。因此引用参数也被称为输入-输出参数。

你不能将未初始化的引用参数传递给函数。C# 使用关键字 `ref` 指定引用参数。你同时还必须在传递参数给要求引用参数的函数时使用关键字 `ref`。

```
int a= 5;  
FunctionA(ref a); // 使用 ref, 否则将引发编译时错误  
Console.WriteLine(a); // 打印 20  
  
void FunctionA(ref int Val)  
{  
    int x= Val;
```

```

        Val = x* 4;
    }

```

输出参数

输出参数是只从函数返回值的参数。输入值不要求。C# 使用关键字 `out` 表示输出参数。

```

int Val;
    GetNodeValue(Val);

bool GetNodeValue(out int Val)
{
    Val = value;
    return true;
}

```

参数和数组的数量变化

C# 中的数组使用关键字 `params` 进行传递。一个数组类型的参数必须总是函数最右边的参数。只有一个参数可以是数组类型。你可以传送任意数量的元素作为数组类型的参数。看了下面的例子你可以更好的理解：

注意：使用数组是 C# 提供用于可选或可变数量参数的唯一途径。

```

void Func(params int[] array)
{
    Console.WriteLine("number of elements {0}", array.Length);
}

Func(); // 打印 0
Func(5); // 打印 1
Func(7,9); // 打印 2
Func(new int[] {3,8,10}); // 打印 3
int[] array = new int[8] {1,3,4,5,5,6,7,5};
Func(array); // 打印 8

```

运算符与表达式

运算符和表达式跟 C++ 中完全一致。然而同时也添加了一些新的有用的运算符。有些在这里进行了讨论。

is 运算符

is 运算符是用于检查操作数类型是否相等或可以转换。is 运算符特别适合用于多态的情形。is 运算符使用两个操作数，其结果是布尔值。参考例子：

```
void function(object param)
{
    if(param is ClassA)
        //做要做的事
    else if(param is MyStruct)
        //做要做的事
    }
}
```

as 运算符

as 运算符检查操作数的类型是否可转换或是相等（as 是由 is 运算符完成的），如果是，则处理结果是已转换或已装箱的对象（如果操作数可以装箱为目标类型，参考 装箱/拆箱）。如果对象不是可转换的或可装箱的，返回值为 null。看看下面的例子以更好的理解这个概念。

```
Shape shp = new Shape();
Vehicle veh = shp as Vehicle; // 返回 null，类型不可转换
```

```
Circle cir = new Circle();
Shape shp = cir;
Circle cir2 = shp as Circle; //将进行转换
```

```
object[] objects = new object[2];
objects[0] = "Aisha";
object[1] = new Shape();
```

```
string str;
for(int i="0"; i< objects.Length; i++)
{
    str = objects as string;
    if(str == null)
        Console.WriteLine("can not be converted");
    else
        Console.WriteLine("{0}",str);
}
```

Output:

```
Aisha
can not be converted
```

语句

除了些许附加的新语句和修改外，C# 的语句和 C++ 的基本一致。以下是新的语句：

foreach-用于迭代数组等集合。

```
foreach (string s in array)
    Console.WriteLine(s);
```

lock

在线程中使代码块称为重点部分。（译注：lock 关键字将语句块标记为临界区，方法是获取给定对象的互斥锁，执行语句，然后释放该锁。lock 确保当一个线程位于代码的临界区时，另一个线程不进入临界区。如果其他线程试图进入锁定的代码，则它将一直等待（即被阻止），直到该对象被释放。）

checked/unchecked

用于数字操作中的溢出检查。

```
int x = Int32.MaxValue; x++; // 溢出检查
{
    x++; // 异常
}
unchecked
{
    x++; // 溢出
}
```

Switch

Switch 语句在 C# 中已进行了修改：

1. 现在在执行一条 case 语句后，程序流不能跳至下一 case 语句。之前在 C++ 中这是可以的。

```
int var = 100;
switch (var)
{
    case 100: Console.WriteLine("<Value is 100>"); // 这里没有 break
    case 200: Console.WriteLine("<Value is 200>"); break;
}
```

C++ 的输出:

<Value is 100><Value is 200>

而在 C# 中你将得到一个编译时错误:

```
error CS0163: Control cannot fall through
        from one case label ('case 100:') to another
```

2. 然而你可以像在 C++ 中一样这么用:

```
switch (var)
{
    case 100:
    case 200: Console.WriteLine("100 or 200<VALUE is 200>");
break;
}
```

3. 你还可以用常数变量作为 case 值:

```
const string WeekEnd = "Sunday";
const string WeekDay1 = "Monday";

....

string WeekDay = Console.ReadLine();
switch (WeekDay )
{
case WeekEnd: Console.WriteLine("It's weekend!!"); break;
case WeekDay1: Console.WriteLine("It's Monday"); break;
}
```

委托

委托让我们可以把函数引用保存在变量中。这就像在 C++ 中使用 typedef 保存函数指针一样。委托使用关键字 delegate 声明。看看这个例子,你就能理解什么是委托:

```
delegate int Operation(int val1, int val2);
public int Add(int val1, int val2)
{
    return val1 + val2;
}
public int Subtract (int val1, int val2)
{
    return val1- val2;
}
```



```

public void Perform()
{
    Operation Oper;
    Console.WriteLine("Enter + or - ");
    string optor = Console.ReadLine();
    Console.WriteLine("Enter 2 operands");

    string opnd1 = Console.ReadLine();
    string opnd2 = Console.ReadLine();

    int val1 = Convert.ToInt32 (opnd1);
    int val2 = Convert.ToInt32 (opnd2);

    if (optor == "+")
        Oper = new Operation(Add);
    else
        Oper = new Operation(Subtract);

    Console.WriteLine(" Result = {0}", Oper(val1, val2));
}

```

继承与多态

C# 只允许单一继承。多重继承可以通过接口达到。

```

class Parent{
}

```

```

class Child : Parent

```

虚函数

虚函数在 C# 中同样是用于实现多态的概念的，除了你要使用 `override` 关键字在子类中实现虚函数外。父类使用同样的 `virtual` 关键字。每个重写虚函数的类都使用 `override` 关键字。（译注：作者所说的“同样”，“除……外”都是针对 C# 和 C++ 而言的）

```

class Shape
{
    public virtual void Draw()
    {

```

```

        Console.WriteLine("Shape.Draw")
    }
}

class Rectangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Rectangle.Draw");
    }
}

class Square : Rectangle
{
    public override void Draw()
    {
        Console.WriteLine("Square.Draw");
    }
}

class MainClass
{
    static void Main(string[] args)
    {
        Shape[] shp = new Shape[3];
        Rectangle rect = new Rectangle();

        shp[0] = new Shape();
        shp[1] = rect;
        shp[2] = new Square();

        shp[0].Draw();
        shp[1].Draw();
        shp[2].Draw();
    }
}

```

Output:

Shape.Draw

Rectangle.Draw

Square.Draw

使用“new”隐藏父类函数

你可以隐藏基类中的函数而在子类中定义其新版本。关键字 `new` 用于声明新的版本。思考下面的例子，该例是上一例子的修改版本。注意输出，我用 关键字 `new` 替换了 `Rectangle` 类中的关键字 `override`。

```
class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Shape.Draw");
    }
}

class Rectangle : Shape
{
    public new void Draw()
    {
        Console.WriteLine("Rectangle.Draw");
    }
}

class Square : Rectangle
{
    //这里不用 override
    public new void Draw()
    {
        Console.WriteLine("Square.Draw");
    }
}

class MainClass
{
    static void Main(string[] args)
    {
        Console.WriteLine("Using Polymorphism:");
        Shape[] shp = new Shape[3];
        Rectangle rect = new Rectangle();

        shp[0] = new Shape();
        shp[1] = rect;
        shp[2] = new Square();

        shp[0].Draw();
        shp[1].Draw();
        shp[2].Draw();

        Console.WriteLine("Using without
```

```

Polymorphism:"");
        rect.Draw();
        Square sqr = new Square();
        sqr.Draw();
    }
}

```

Output:

Using Polymorphism

Shape.Draw

Shape.Draw

Shape.Draw

Using without Polymorphism:

Rectangle.Draw

Square.Draw

多态性认为 Rectangle 类的 Draw 方法是和 Shape 类的 Draw 方法不同的另一个方法，而不是认为是其多态实现。所以为了防止父类和子类间的命名冲突，我们只有使用 new 修饰符。

注意：你不能在一个类中使用一个方法的两个版本，一个用 new 修饰符，另一个用 override 或 virtual。就像在上面的例子中，我不能在 Rectangle 类中增加另一个名为 Draw 的方法，因为它是一个 virtual 或 override 的方法。同样在 Square 类中，我也不能重写 Shape 类的虚方法 Draw。

调用基类成员

如果子类的数据成员和基类中的有同样的名字，为了避免命名冲突，基类成员和函数使用 base 关键字进行访问。看看下面的例子，基类构造函数是如何调用的，而数据成员又是如何使用的。

```

public Child(int val) :base(val)
{
    myVar = 5;
    base.myVar;
}

```

OR

```

public Child(int val)
{
    base(val);
    myVar = 5 ;
    base.myVar;
}

```

```
}
```

前景展望

本文仅仅是作为 C# 语言的一个快速浏览，以便你可以熟悉该语言的一些特性。尽管我尝试用实例以一种简短而全面的方式讨论了 C# 几乎所有的主要概念，但我认为还是有很多内容需要增加和讨论的。

以后，我会增加更多的没有讨论过的命令和概念，包括事件等。我还想给初学者写一下怎么用 C# 进行 Windows 编程。