C#系列教程 基础篇

邵发 2020/2/4

shaofa

官网 http://afanihao.cn QQ群 1034784769

目 录

第 1 章　关于本篇

如果你没有编程基础，请先学习 Java 学习指南系列教程。

在学完 Java 入门与进阶、Swing 入门篇、Swing 高级篇之后，再转到 C#系列教程将会十分轻松。

第 2 章　开发环境

2.1　开发环境

本教程使用 Visual Studio 2019 作为演示环境。

略。具体安装和使用请参考视频讲解。

2.2　第一个程序

使用 VS 创建第一个程序。

按视频操作。

```
namespace Basic0202
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("阿发你好");
        }
    }
}
```

第 3 章　类型和变量

3.1　类型和变量

C#里的变量类型，也是分为两类：

1 值类型 Value Type

例如，sbyte、short、int、long、bool、float、double、char 等类型都值类型

2 引用类型 Reference Type:

例如，object, string , …，和所有的创建的 class 都是引用类型

简单做个练习，示例如下。

```
class Program
{
    static void Main(string[] args)
    {
        int a = 12;
        int b = a + 123;
        Console.WriteLine("结果为" + b);
    }
}
```

需要注意的是，C#的语法可以说是非常庞杂，泥沙俱下，在学习的时候建议先按照教程把主要的内容练习一遍。

就基本的值类型来说，它不仅有 sbyte, short, int, long，还有 byte, ushort, uint, ulong。但常用的类型其实就是 int , long , bool, string, double 这些类型。

3.2　控制台输出

使用 Console.WriteLine()可以实现控制台的输出，在入门语法阶段，此方法会

经常调用。示例如下。

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("numbers: " + 30);
        Console.WriteLine("中招:{0},痊愈:{1}", 1932, 39);
    }
}
```

WriteLine 有两种常用的写法。

一是拼接输出，例如：

```
Console.WriteLine("numbers: " + 30);
```

二是参数化的输出，例如：

```
Console.WriteLine("中招:{0},痊愈:{1}", 1932, 39);
```

其中，{0}就是后面第一个参数的值，{1}就是后面第二个参数的值。

第 4 章　类和对象

4.1　添加类

按照视频演示，添加一个类 Student。当创建一个 Student 类时，默认的源码文件为 Student.cs。其中，后缀 cs 为 CSharp 的简称。

在 C#里，一个*.cs 源码文件里是可以定义多个类的。例如。

```
namespace Basic0401
{
    public class Student
    {
        public int id;
        public string name;
        public bool sex;
        public string phone;
    }
    public class Teacher
    {
        public string name;
        public string course;
    }
}
```

一个类可以用 public 修饰，也可以不加访问修饰符(默认为 interal，表示在当前 DLL/EXE 中可见)。对于初学者来说，这个不重要，置空或者用 public 修饰均可。

4.2　字段和方法

在一个类里，添加字符和方法。示例如下。

```
class Example
{
    public int number = 10;
    public void Output()
    {
        Console.WriteLine("number: " + number);
    }
}
```

字符和方法前面都可以加访问修饰符，如 public、protected 或 private。

需要注意的是，在 C#中，建议方法名应该以大写字母开头。

4.3 构造方法

一个类可以重载多个构造方法。示例代码如下。

```
class Student
{
    public int id;
    public string name;
    public bool sex;
    public string phone;

    // 写几个构造方法
    public Student()
    {
    }
    public Student(int id)
    {
        this.id = id;
    }
```

```
    public Student(int id, string name)
    {
        this.id = id;
        this.name = name;
    }
}
```

在创建 Student 对象时，可以选择不同的构造方法。示例代码如下。

```
class Program
{
    static void Main(string[] args)
    {
        Student s1 = new Student();
        Student s2 = new Student(2010002);
        Student s3 = new Student(2010003, "shaofa");
    }
}
```

4.4 类的拆分

一个类的定义，可以分拆多个*.cs 文件中。在分拆定义时，需使用 partial 关键词来告诉编译器：当前文件中只是类的一部分定义。

在以下的例子中，MyForm 类的一部分定义放在 MyForm.cs 中，另一部分定义放在 MyFormDesign.cs 文件中。

首先，在 MyForm.cs 中有一部分定义，示例如下。

```
public partial class MyForm    // 添加 partial 修饰
{
```

```
        public void ButtonClicked()
        {
            Console.WriteLine("Button '{0}' is clicked" ,
this.buttonText);
        }
    }
```

然后，再 MyFormDegisn.cs 中有另一部分定义，示例如下。

```
    public partial class MyForm // 添加 partial 修饰
    {
        public String buttonText = "提交";
        public void UserPressButton()
        {
            this.ButtonClicked();
        }
    }
```

在使用 MyForm 类时没有什么特别之处，就和普通的类型是一样的。例如，

```
        static void Main(string[] args)
        {
            MyForm form = new MyForm();
            form.buttonText = "保存";
            form.UserPressButton();
        }
```

什么时候需要使用类的 partial class 语法呢？这个语法在下一篇教程（Winform 窗口应用开发）中，会经常见到。

第 5 章　属性

属性（Property），是 C#里一个特有的语法。它既像是一个字段，也像是一个方法，是两者的综合体。

5.1　Getter 和 Setter

先来回顾一下 Getter 与 Setter。在面向对象的编程语言里，一般都支持 Getter 和 Setter 术语。

Getter 与 Setter 用于实现字段读写的封装。示例如下。

```
class Student
{
    private string name;

    // Getter
    public string GetName()
    {
        return this.name;
    }
    // Setter
    public void SetName(string name)
    {
        this.name = name;
    }
}
```

其中，name 是一个私有的字段。外界如果想读取这个字符的值，需调用 GetName()方法，称为 Getter。同理，SetName()方法用修改这个字段，称为 Setter。

由于可见，使用 Getter/Setter 可以实现对字段读写的控制。如果同时有

Getter/Setter，则该字段是可读可写的。如果只有 Getter 没有 Setter，则该字段就是只读的。

5.2 添加属性

属性(Property)，是对 字段 + Getter/Setter 的另一种写法。

下面，在 Student 类里定义一个属性，示例如下。

```
class Student
{
    private string name;
    public String Name
    {
        get
        {
            return this.name;
        }
        set
        {
            this.name = value;
        }
    }
}
```

其中，小写的 name 叫字段，大写的 Name 叫做属性。不难理解，属性就是 Getter/Setter 的一种变形的写法。

属性的调用示例如下所示。

```
Student s= new Student();
s.Name = "shaofa"; // 相当于 s.SetName("shaofa")
Console.WriteLine("姓名:" + s.Name); // 相当于 s.GetName()
```

5.3 自动属性

自动属性（Auto Property)，是形式上最简单的一种属性定义。

例如，

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

这是很常见的写法，是一种简化版的属性定义。

它相当于编译器为我们自己定义了一个字段，并生成 get{}set{}。脑补一下，就是编译器为我们写成以下的形式。

```
class Student
{
    public string xxx_auto_field;
    public string Name
     {
        get
        {
           return xxx_auto_field;
        }
        set
        {
            xxx_auto_field = value;
        }
    }
}
```

也就是说，编译会自动定义一个字段比如叫 xxx_auto_field，再自动添加 get 和 set 的实现代码。

第 6 章　继承

C#中的继承语法和 Java 基本一致：单根继承，可以重写，支持多态调用，所有的类有一个共同的父类 object 类。

6.1　继承

作为面向对象的语言，C#自然也是支持继承的。继承的一般写法如下。

```
class MyExample : Example
{
}
```

其中，Example 为父类，MyExample 为子类。父类 Example 中所有的 public/protected 成员被子类继承。

例如，父类的定义如下。

```
class Example
{
    public int number;
    public void Output()
    {
        Console.WriteLine("number:" + number);
    }
}
```

则子类 MyExample 会自动继承 number 字段，和 Output 方法。例如，

```
class Program
{
    static void Main(string[] args)
    {
        MyExample e = new MyExample();
```

```
        e.number = 123;
        e.Output();
    }
}
```

6.1.1 重写

如果子类想对父类的方法实现进行修改，可以重写该方法。在重写时，要求父类将此方法声明为 virtual（虚拟方法，表示被重写的）。子类将方法声明为 override（表示重写）。

在父类 Example 中，代码如下。

```
class Example
{
    。。。略
    public virtual void Translate()
    {
        Console.WriteLine("number:" + number);
    }
}
```

其中，父类将 Translate()方法声明为 virtual，表示此方法可以被子类重写。

在子类 MyExample 中，代码如下。

```
class MyExample : Example
{
    public override void Translate()
    {
        Console.WriteLine("号码为:" + number);
    }
}
```

其中，子类将 Translate()方法声明为 override，表示这是在重写父类的方法。

6.2 多态

在 C# （和 Java）语言中，多态是指的是以下几种语法：

1 方法名的重载 Overload

2 继承时的方法重写 Override

3 泛型 Generic Type

本节讨论的是第 2 种情况。

考虑以下的代码。

```
Example e = new MyExample();
e.number = 123;
e.Translate();
```

其中，创建了一个 MyExample 对象，却用一个 Example 类型的引用。也就是说，e 在名义上是一个 Example，实际上却是一个 MyExample 类型的对象。

那么，在执行 e.Translate()的时候，是运行的父类的 Example.Translate()，还是子类的 MyExample.Translate()呢？这种现象便称为一种多态实现，可以验证，在程序运行的时候将执行子类的 Translate()方法。

下面给出一个技巧：在单步调试状态下，可以在 VS 窗口里轻易的分辨出一个引用的实际对象类型。如下图所示。

图中，显示了引用 e 的值和类型。在类型中，可以清楚地看到对象的类型为 Basic0601.Example{Baisc0601.MyExample} ，前者为名义类型，后者为实际类型。

6.3 构造方法的继承

在继承时，默认继承父类的构造方法。也就是说，在创建子类对象时，总是会先行调用父类的构造方法。子类可以用 base 关键词显式的调用父类的某个构造方法。

定义父类 Point。示例代码如下。

```
public class Point
{
    public int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

定义子类 Point3D,示例代码如下。

```
public class Point3D : Point
{
    public int z;
    public Point3D(int x, int y, int z) : base(x, y)
    {
        this.z = z;
    }
}
```

其中，在子类的构造方法之后，显式地调用了 base(x, y) ，表示对父类构造方法的显式调用。

6.4 ToString

C#里，一个子类只能一个父类，称为单根继承。

当一个类没有指明父类时，则相当于继承于 object 类。也就是说， object 类是最有类的最终父类。

例如，

```
class Student
{
}
```

其中，类型 Student 没有指定父类。则默认地，相当于继承 object 类。示例代码如下。

```
class Student : object
{
}
```

由于 Student 继承于 object 类，所以 object 里的一些方法被自然而然的继承过来。比如，object.ToString()方法，就是一个常见的方法。

例如，可以在 Student 类里重写 ToString()方法。示例如下。

```
class Student
{
    public int id;
    public string name;
    public override string ToString()
    {
        return "学号: " + id + ", 姓名:" + name;
    }
}
```

重写 ToString()方法的作用：

1 调试的时候方便查看

2 字符串拼接的时候方便

例如，

```
Console.WriteLine ( stu );
```

相当于，

```
Console.WriteLine ( stu.ToString() );
```

此两种写法是等效的。

第 7 章　抽象类与接口

7.1　抽象类

以 abstract 修饰的类，称为抽象类。

以下定义了一个抽象类，名为 FutureVehicle，里面定义了 2 个抽象方法。示例如下。

```
public abstract class FutureVehicle
{
    public abstract void Run();
    public abstract void Fly();
}
```

其中，FutureVehicle 对“未来交通工具”作为抽象，它应该既能跑(Run)，也能够飞(Fly)。

抽象类无法直接使用，应该定义一个子类，实现它所规定的方法。例如，定义一个 AfVehicle，它实现了 Fly 和 Run 方法。

```
class AfVehicle : FutureVehicle
{
    public override void Fly()
    {
        Console.WriteLine("嗖嗖嗖 .....");
    }
    public override void Run()
    {
        Console.WriteLine("嘀嘀嘀 .....");
    }
}
```

7.2 接口

使用 interface 关键字可以定义一个接口。

例如，定义一个接口 IAudipPlayer，表示音频播放功能。注意，接口的名字应该以大写的 I 字母开头。

```
// 定义一个接口，接口名要以 I 开头
public interface IAudioPlayer
{
    void Play(int[] data);
}
```

同样地，接口也无法直接使用，应该定义它的子类，并实现相应的方法。例如，定义一个 XiaoMiPlayer 类，示例代码如下。

```
class XiaoMiPlayer : IAudioPlayer
{
    public void Play(int[] data)
    {
        Console.WriteLine("da da da ...");
    }
}
```

对 XiaoMiPlayer 的调用示例如下。

```
    static void Main(string[] args)
    {
        int[] data = { 0x12, 0x2c, 0x34, 0xd3 };
        IAudioPlayer player = new XiaoMiPlayer();
        player.Play(data);
    }
```

7.3 内部类型

把一个类型定义在另一个类的内部，便称之为内部类型。

例如，在 Example 类中定义一个内部类 Item，和一个内部接口 IPlayer。示例代码如下。

```
public class Example
{
    // 内部类
    public class Item
    {
        public int id;
        public string name;
    }
    // 内部接口
    public interface IPlayer
    {
        void play(byte[] data);
    }
}
```

在使用的时候，内部类型的名字全称应该加上外部类的名字。例如，Item 类的全称为 Af.Example.Item ，其中 Af 为 namespace 的名称，Example 为外部类的名称。(关于 namespace 的语法，后面有专门介绍)。

以下代码了展示了 Item 类的使用。

```
Af.Example.Item item = new Af.Example.Item();
item.id = 222;
item.name = "xyz";
```

内部类型并没特别之处，和写在外部单独定义相比，基本上是一样的。内部类和外部类之间，没有什么特别的关系。

第 8 章　重载操作符

重载操作符，其实是从 C++里沿袭过来的一个概念。这个语法不太重要，本章稍作了解即可。

8.1　索引器

默认的，数组类型可以使用索引器，来获取一个位置的数据。比如，

```
int[] a = new int[10];
a[0] = 33;
```

其中，a[0]这种中括号传索引的方式称为索引器。

对于自定义 class 类型来说，也有办法使其支持索引器。形如，

```
MyArray a = new MyArray(4);
a[0] = 12;
a[1] = 29;
a[2] = 54;
a[3] = 80;
for(int i=0; i<a.Size(); i++)
{
    Console.Write(a[i] + " ");
}
```

其中，MyArray 是一个自定义的类。要使它支持按索引器，需要在类中添加索引器支持。形如，

```
class MyArray
{
    private int[] items;
    public MyArray(int capacity)
    {
        items = new int[capacity];
```

```
    }
    // 索引器
    public int this[int index]
    {
        get
        {
            return items[index];
        }
        set
        {
            items[index] = value;
        }
    }
}
```

可以看到，所谓的索引器就是一种固定形式的写法。大家看一下，有个印象即可，在实际的项目中基本不会自己来写的。

实际上，索引器的语法相当于添加一个 Get 和一个 Set 方法。例如，

```
class MyArray
{
    private int[] items;
    public void SetItem(int i, int val)
    {
        items[i] = val;
    }
    public int GetItem(int i)
    {
        return items[i];
    }
}
```

其中，SetItem(i, val)用于设置元素的值，GetItem(i)用于取元素的值。这样写更容易读懂一些。

8.2 重载操作符

重载操作符的语法，也是看一下有个印象就好，不太重要。

默认地，基本的值类型支持直接运算的。形如，

```
int a = 10;
int b = 12;
int result = a * b;
```

但是，对于自定义的 class 类型，也是可以支持运算的。形如，

```
MyFraction f1 = new MyFraction(1, 3);
MyFraction f2 = new MyFraction(2, 5);
// 相乘 (重载操作符语法)
MyFraction result = f1 * f2;
Console.WriteLine("结果: " + result);
```

其中，MyFraction 是一个自定义的类，它竟然支持 f1 * f2 这种乘法运算，是不是很神奇呢？

这种语法称为“重载运算符”，就是把运算符重新定义一下。需要在 MyFraction 类中加点东西才行。形如，

```
   class MyFraction
   {
       public int num = 0; // 分子
       public int den = 1; // 分母
       // 重载乘法操作符
       public static MyFraction operator *(MyFraction a,
MyFraction b)
       {
           MyFraction result = new MyFraction();
           result.num = a.num * b.num;
           result.den = a.den * b.den;
```

```
            return result;
        }
    }
```

其中，重载了 operator* 这个运算符，它是一种特殊的形式，相当于有一个方法，方法名字叫 operator * 。有个印象就行了，实际项目中不会自己定义的。

重载操作符的虽然看起来“神奇”，但是可读性并不好，如果可以选择的话，还是推荐大家用普通的静态方法来实现相同的功能。形如，

```
    class MyFraction
    {
        public int num = 0; // 分子
        public int den = 1; // 分母
        // 普通写法
        public static MyFraction Multiply(MyFraction a,
MyFraction b)
        {
            MyFraction result = new MyFraction();
            result.num = a.num * b.num;
            result.den = a.den * b.den;
            return result;
        }
    }
```

在调用的时候，使用如下的形式，

```
MyFraction result = MyFraction.Multiply(f1, f2);
```

显然，这种形式的可读性可好，不会让读者产生混淆，是推荐的写法。

(注： 重载操作符的语法，是从 C++沿袭过来的陋习)

第 9 章　泛型

本章介绍几个泛型类的用法。

9.1　泛型 List

列表（List），是一个常见的泛型类，可以存储多元素。相当于 Java 里面的 ArrayList。

例如，

```
List<int> aaa = new List<int>();
aaa.Add(12);
aaa.Add(3);
aaa.Add(8);
aaa.Add(19);
```

其中，定义了一个 List 对象，泛型参数为 int 类型。表示此 List 中存储 int 类型的元素。

List 的几个常见的 API:

Add(e) 在末尾添加一个元素

Insert(index, e) 在 index 处插入元素 e

Remove(index) 删除 index 处的元素

Count() 取得元素的个数

Clear() 清空所有元素

IndexOf (e) 查找一个元素的位置（要求元素实现了 Equals 方法）

可以用索引器直接访问 List 中的元素，形如，

```
int val = aaa[2];  // 取值
a[3] = 50;  // 设值
```

可以用普通 for 循环来遍历所有元素，形如，

```
for(int i=0; i<aaa.Count(); i++)
{
    Console.WriteLine(aaa[i] + " ");
}
```

也可以用 foreach 循环来遍历，形如，

```
foreach(int item in aaa)
{
    Console.WriteLine(item + " ");
}
```

其中，foreach 是 C#里的特有语法，所有的集合类型都支持 foreach 遍历。按照官方文档的描述，foreach 遍历是推荐的方式，也许内部执行效率更高一些。

9.2 泛型 Dictionary

字典（Dictionary），是用于存储键值的集合，相当于 Java 里面的 HashMap 类。例如，

```
Dictionary<string, string> dict
     =  new Dictionary<string, string>();
dict.Add("txt", "notepad.exe");
dict.Add("bmp", "paint.exe");
dict.Add("dib", "paint.exe");
dict.Add("docx", "wordpad.exe");
```

其中，在创建 Dictionary 对象时要指定键 Key 和值 Value 的类型，此处 Key 类型为 string，值类型也是 string。

使用 Add 方法，可以向字典中添加一项数据。如果该 Key 不存在，则插入成功一项数据；如果 Key 已经存在，则抛出异常。形如，

```
dict.Add("txt", "notepad.exe");
```

使用索引器，也可以添加一项数据。如果 Key 存在，则插入一条数据；如果 Key 已经存在，则覆盖原有的数据。形如，

```
dict["png"] = "photoshop.exe";
```

使用 Remove 方法，可以删除一项数据。如果存在，则删除该项并返回 true；如果不存在，则返回 false。

```
dict.Remove ( "png" );
```

使用 ContainsKey 可以判断一个 Key 是否存在。形如，

```
if (dict.ContainsKey("xml"))
{
    String exeValue = dict["xml"];
}
```

使用索引器，可以从中取得一项数据。如果存在，则返回该项数据；如果不存在，则抛出异常。形如，

```
String exeValue = dict["xml"];
```

9.3 迭代与枚举遍历

集合类支持两种遍历方式：foreach 遍历和枚举遍历。

9.3.1 迭代遍历

集合类都可以使用 foreach 遍历，示例代码如下。

```
List<int> aaa = new List<int>();
aaa.Add(12);
aaa.Add(3);
aaa.Add(8);
aaa.Add(19);
// 迭代遍历
foreach (int item in aaa)
```

```
{
    Console.WriteLine(item + " ");
}
```

9.3.2 枚举遍历

使用枚举器，也可以实现对集合类的遍历。示例代码如下。

```
List<int> aaa = new List<int>();
aaa.Add(12);
aaa.Add(3);
aaa.Add(8);
aaa.Add(19);
// 枚举遍历
List<int>.Enumerator en = aaa.GetEnumerator();
while(en.MoveNext())
{
    int item = en.Current; // 当前值
    Console.WriteLine(item + " ");
}
```

按照官方文档的说明，推荐使用 foreach 方式遍历。

另外，两种遍历都是一个共同的缺点：它们都是只读遍历，不支持“一边遍历、一边删除”的功能。如果要从一个集合中删除多项，需要另行设计一个方案。

第 10 章　委托

委托(delegate)，在 C#中用于实现回调机制。如果你之前学过其他语言，应该知道在各种语言里都可以实现回调机制。比如：

- C 语言里的 函数指针
- C++/Qt 里的 信号和槽
- Java 里面的 接口/监听器

C#里的委托，其实就是回调方法的意思，不过委托这个名字听起来有点晦涩。

10.1　方法的描述

当我们要描述一个方法时，应该说明它的两个特征：

(1) 参考类型列表

(2) 返回值类型

比如，对于以下的 Test1 方法，可以称它是“一个参数为(int)的、返回值为 void 的方法”。

```
class Example
{
    public void Test1(int a)
    {
        Console.WriteLine("test1: " + a);
    }
}
```

同样地，对以下的 Test5 方法，称它是 “一个参数为(int,int)的、返回值为 int 的方法”。

```
    public int test5(int x, int y)
    {
```

```
        return x + y;
    }
```

试着对下方法进行分类：

public void test1(int a) {}

public void test2(int b){}

public int test3(int m){}

public int test4(int n){}

public int test5(int x, int y){}

可以发现，test1 和 test2 同属一类，test3 和 test4 同属一类。那么，有什么标准术语可以来描述一类方法呢？那就是委托(delegate)这个概念。

10.2 委托

委托，delegate，是一个对方法类型的描述

例如：

public delegate void DogFunction(int a);

则 DogFunction 类型指的是“一种参数为(int)、返回值为 void 的方法”。

针对 10.1 列出来的方法，我们可以定义以下委托类型来描述。比如，

```
// 第一类方法
public delegate void DogFunction(int a);
// 第二类方法
public delegate int CatFunction(int b);
// 第三类方法
public delegate int FishFunction(int a, int b);
```

其中，DogFunction 适合描述 Test1 和 Test2 方法，CatFunction 适合描述 Test3 和 Test4 方法，FishFunction 适合描述 Test5 方法。

委托类型的形式有点难懂，可以和 class 和 interface 对照一下。比如，

public **class** Student{}

public **interface** Player{}

public **delegate** void DogFunction (int a);

其中，delegate 声明它是个委托类型，DogFunction 是类型的名字。

其实委托这个名字本身就有点问题，如果改成 callback 会更容易理解一些。

10.2.1 创建委托的实例

委托是一个类型，自然可以创建实例。

形如，

Example ex = new Example();

DogFunction dog = new DogFunction(ex.test1);

dog(18); // 相当于 ex.test1(18)

则 dog 就是一个实例，它指向了 ex.test1()方法。由于 dog 指向一个方法，所以可以调用它。其中，dog(18)，就相当于调用 ex.test1(18)。

10.3 委托的更多用法

以下代码，定义了一个实例方法 Test 和一个静态方法 Print。

```
class Example
{
    public void Test(int a)
    {
        Console.WriteLine("Test: " + a);
    }
    public static void Print(int a)
    {
        Console.WriteLine("Print: " + a);
    }
};
```

然后，定义一个委托类型，

public delegate void DogFunction(int a);

以此为例，再介绍委托的几个用法细节。

10.3.1 委托的调用

委托的调用有两个形式，都是可以的。示例如下。

```
Example ex = new Example();


DogFunction dog = new DogFunction(ex.Test);
// dog(18);
dog.Invoke(18);
```

其中，使用 dog(10)或者 dog.Invoke(18)的效果是相同的。都相当于直接调用 ex.Test(18)。

10.3.2 静态方法的调用

委托也可以指向一个静态方法。

DogFunction dog2 = new DogFunction(Example.Print);

dog2.Invoke(19);

其中，Example.Print()为静态方法。

10.3.3 作为内部类型

委托是一种类型，可以和 class、interface 同级，也可以作为 class 的内部类型。简单地讲，就像内部类一样，写在另一类的内部。形如，

```
public class YourClass
{
    public delegate void DogFunction (int a);
```

}

则此委托类型的全称为： YourNameSpace.YourClass.DogFunction 。这是比较常见的一种定义方式。

在创建时，可以指定其全称，例如。

YourClass.DogFunction func = new YourClass.DogFunction(Example.Print);

第 11 章　事件

事件(event)，是 C#里实现回调的一种方式。事件和委托，是 C#为回调机制专门定义的概念，相当于 C++/Qt 界面开发技术中信号和槽的技术。

11.1　回调

事件也是一个很晦涩的概念。为了方便理解，先不管什么事件，先看看怎么用委托来实现回调机制。

假设以下场景：有一个颜色选择控件 ColorBox，用户可以选择颜色。在用户选择事件发生后，希望能执行一个事件处理方法。类似如下的调用。

```
class Program
{
    static void OnColorChanged(string color)
    {
        Console.WriteLine("使用颜色:" + color);
    }
    static void Main(string[] args)
    {
        ColorBox box = new ColorBox();
        // 注册回调
        box.handler
    = new ColorBox.ChangeHandler(Program.OnColorChanged);

        // 用户点击
        box.UserSelect(0);
    }
}
```

当用户选择 UserSelect()事件发生后，执行回调方法 OnColorChanged()处理。那么，ColorBox 内部如何实现呢？示例代码如下。

```
class ColorBox
{
    private string[] options =  {
        "white", "red", "green", "blue", "black"
    };
    // 定义委拖类型
    public delegate void ChangeHandler(string color);
    // 定义回调
    public ChangeHandler handler;

    // 模拟用户选择
    public void UserSelect(int index)
    {
        string color = options[index];

        // 调用回调方法
        if(handler != null)
            handler.Invoke(color);
    }
}
```

这其实是自定义事件的典型方式。在 ColorBox 内，保存一个回调方法的引用 handler。当用户选择时，执行 handler.Invoke()方法。这便是一个回调机制的典型实现了。

11.2 事件

下面，改用 event 的语法实现上面的回调机制。

```
    // 定义委拖类型
    public delegate void ChangeHandler(string color);
    // 定义事件 ( 其实就是一个处理器 )
    public event ChangeHandler Handlers;
    // 模拟用户选择
    public void UserSelect(int index)
    {
        string color = options[index];
```

```
            // 调用回调方法
            if(Handlers != null)
                Handlers.Invoke(color);
        }
```

可以看到，事件这个名字起得也有问题，所谓的事件，其实是定义一个处理器（回调）。其中这一句，

public event ChangeHandler Handlers;

这一句要说理解很难，这个语法形式只能强行记住。可以认为是定义了一个类型为 event ChangeHandler 的对象 Handlers，在 Handlers 对象内可以存储多个回调方法。

向 Handlers 里注册回调方法，示例代码如下。

```
ColorBox box = new ColorBox();
box.Handlers += new ColorBox.ChangeHandler(Program.OnColorChanged);
box.UserSelect(0);
```

直接用 += 就能够添加一个回调，这也够奇特的。而且，看起来 Handers 好像也没有实例化就能直接使用的样子。

总之，事件 event 是一个奇葩的语法，不好解释也不用解释，强行记住其形式即可。

第 12 章　命名空间

命名空(namespace) ，对应于 Java 里的 package，用 namespace 来对功能模块分包管理。

12.1　命名空间

当一个项目中的代码规模比较庞大时，就需要分包管理。在 C#里，使用 namespace 对代码进行分包管理。一般形式为：

```
namespace YourNS
{
    public class YourClass
   {
   }
   public class YourInterface
   {
   }
   public class delegate ..... ;
}
```

其中，定义了一个命名空间叫 YourNS。在 namespace 里，可以定义 class、interface、delegate 等其他类型。

当创建一个项目时，默认的 namespace 即为当前的项目名称。例如，

```
namespace Basic1201
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

如果两个类在同一个命名空间下，则类名是互相可见的，可以直接使用。

如果两个类分属不同的命名空间，则在使用的时候需要使用 using 引入。例

如，另外定义一个类 Student，放在命名空间 My.Good.School 下面。

```
namespace My.Good.School
{
    class Student
    {
        public int id;
        public string name;
    }
}
```

那么在 Program 类中使用 Student 类里，就需要先 using 引入它的包名。例如，

```
using My.Good.School;
using System;
namespace Basic1201
{
    class Program
    {
        static void Main(string[] args)
        {
            Student stu = new Student();
            stu.id = 20200001;
            stu.name = "shaofa";
        }
    }
}
```

也可以使用别名 Alias 的语法，定义一个别名。比如，

using SSS= using My.Good.School.Student;

则 SSS 就是一个类型的别名，可以直接使用。比如，

SSS stu = new SSS();

第 13 章　异常

异常(Exception)，用于实现出错处理。C#里支持完善的异常机制，和 Java 语言相同。如果不了解异常原理，可以参考《Java 学习指南》。

13.1　异常机制

C#里也支持 try...catch...finally... 这一套机制，和 Java 里的用法相同。例如，

```
try
{
   int a = Convert.ToInt32("123a5");
   Console.WriteLine("结果: " + a);
 }catch(Exception e)
 {
    Console.WriteLine("出错: " + e.Message);
    Console.WriteLine(e.StackTrace);
 }
```

其中，try{} 用于监视一段可能出错的代码，当有异常抛出时，可能被 catch{} 捕获处理。

13.2　自定义异常

当应用程序中需要自定义异常时，可以从 Exception 类派生。例如，

```
class NotFoundException : Exception
{
    public NotFoundException()
    {
    }
    public NotFoundException(string message)
        : base(message)
```

```
    {
    }
}
```

其中，定义了自定义的异常类 NotFoundException。一般应定义一个带参的构造方法，message 参数为出错原因。

```
class Program
{
    public static void find(int a)
    {
        if (a < 0)
            throw new NotFoundException(a + "不存在");

        // ok
    }
    static void Main(string[] args)
    {
        find(-10);

    }
}
```

所有抛出的异常对应都应被捕获，如果未被捕获，最终将导致程序异常终止。

第 14 章　库

C#里以 DLL 的形式来发布库，相当于 Java 里的 JAR 文件。

本章内容以实际操作为主，请对照视频操作一下。

第 15 章　常用工具类

本章介绍几个常用的工具类。

15.1　字符串

string 类型表示一个字符串，实际它是 System.String 的别名。

String 类提供了比较丰富的方法，以实现各种功能。

将主要的功能逻辑如下：

a.EndsWith(x)　是否以 x 结尾

a.Equals(b)　判断 a,b 是否相同

a = String.Format(fmt, args)　格式化

index = a.IndexOf(sub)　判断子串是否存在

a.Insert(index , sub)　插入一个子串

String.IsNullOrEmpty(a)　判断是否 null 或为空串

index = a.LastIndexOf(sub)　从后面查找一个子串

a.Remove(index, count)　删除一部分子串

a.Replace(oldstr, newstr)　查找并替换子串

a.Split(ccc)　分割

a.StartsWith(x)　判断是否以 x 开头

a.Substring(start, length)　获取子串

a.ToLower() / a.ToUpper()　转成小写/大写

a.Trim()　去除空白

15.2 日期与时间

日期与时间相关的处理，主要涉及两个类：

DateTime 一个时间点 ，如 2020-1-29 17:21:45

TimeSpan 一个时间长度，如 3 天零 4 小时 50 分钟

下面演示几个主要的用法：

1 创建一个 DateTime

```
DateTime dt1 = new DateTime(2020, 1, 29);
```

2 取系统的当前时间

```
DateTime dt2 = DateTime.Now;
int year = dt2.Year;
int hour = dt2.Hour;
```

3 转换 DateTime => String

```
String str = dt2.ToString("yyyy-MM-dd HH:mm:ss");
```

4 解析 String => DateTime

```
DateTime dt3 = DateTime.ParseExact("2020/1/29",
                    "yyyy/M/d",
                    CultureInfo.InvariantCulture);
```

5 历法计算

```
DateTime dt4 = dt3.AddDays(-100);
Console.WriteLine("100 天前: " + dt4.ToString("yyyy-MM-dd"));
```

6 时间差 interval = t1.Substract(t2)

例如，计算从 1970-1-1 开始到现在的时间流逝

```
TimeSpan interval = DateTime.Now.Subtract(new DateTime(1970, 1, 1));
double milli = interval.TotalMilliseconds;
```

15.3 文件与目录

文件和目录操作，主要涉及以下几个类：

File , FileInfo ： 文件的创建、复制、删除、移动

Directory, DirectoryInfo： 目录的创建、删除、移动和枚举

Path： 文件目录的路径处理

15.3.1 目录操作

目录操作有两个类: Directory / DirectoryInfo 。它们所实现的功能是类似的，都是目录相关的操作，如创建目录、移动、删除、枚举子文件等。

Directory: 提供一系列 **静态方法** ，例如：

```
Directory.CreateDirectory("c:\\test\\x\\y\\z");
```

DirectoryInfo: 提供一系列 **实例方法**，例如：

```
string dirPath = "c:\\test\\x\\y\\z";
DirectoryInfo dir = new DirectoryInfo(dirPath);
dir.Create();
```

可以发现，DirectoryInfo 必须要先 new 一个实例，才能调用该实例的方法。故称为实例方法。

15.3.1 文件操作

File 和 FileInfo 类，实现与文件相关的操作。比如，文件的移动、拷贝、删除等操作。同样的，File 类提供一系列静态方法，而 FileInfo 提供的是实例方法。用法示例如下。

File: 提供一系列 静态方法 。例如，

```
File.Move( path1,  path2);
```

FileInfo: 提供一系列 实例方法。例如，

```
FileInfo info = new FileInfo(filePath);
long size = info.Length;
DateTime time = info.LastWriteTime;
```

关于它们的更多用法，可以参考官方文档。

需要强调的是，File 和 FileInfo 所提供的方法，只是对文件本身的操作，并不支持对文件内容的读写。在下一章中，才会介绍如何读写文件的内容。

第 16 章　文件 IO

本章介绍怎么样读写文件中的数据。实际上，文件的数据编码是一个比较复杂的课题，在这里不会深入探讨。

与文件读写的类比较多，主要有 FileStream，BinaryReader、BinaryWriter，StreamReader、StreamWriter 等。其中，

FileStream: 文件流，最基础的文件读写 API

BinaryReader / BinaryWriter: 基于二进制的编解码 API

StreamReader / StreamWriter: 基于文本的读写 API，支持按行读写

16.1　写文件

使用 FileStream，可以实现对文件的写入。

示例如下：

```
// 如果目录不存在，则创建目录
FileInfo fi = new FileInfo("c:/example/data");
fi.Directory.Create();
// 1 打开文件流 (写方式)
FileStream stream = fi.OpenWrite();
// 2 写入数据
byte[] data = { 1, 2, 3, 4 , 0xA0, 0xB1, 0xC2, 0xC3};
stream.Write(data, 0, 8);
// 3 关闭文件流
stream.Close();
```

其中，

- 以 FileInfo.OpenWriter()，打开一个文件流
- stream.write() 写入数据
- stream.Close() 关闭文件流

在写入文件之前，须保证目录已经存在。如果目录不存在，则会抛出异常。所以要调用 fi.Directory.Create() 创建所需的目录。

注意：在 Windows10 下，不允许在 C:\根目录下直接创建文件。所以在这里，创建一个测试用的目录 C:\example\ 。

16.2 读文件

使用 FileStream，也可以实现对文件内容的读取。

示例代码如下：

```
// 文件路径
FileInfo fi = new FileInfo("c:/example/data");

// 1 打开文件流 (读方式)
FileStream stream = fi.OpenRead();
// 2 读取数据
byte[] buffer = new byte[4000];
int n = stream.Read(buffer, 0, buffer.Length);
// 返回值 n 是实际读取到的字节数
Console.WriteLine("Got Bytes: " + n);

// 3 关闭文件流
stream.Close();
```

其中，

- fi.OpenRead()，是以只读方式来打开文件
- stream.Read()，来读取数据，读到数据存储在 buffer 数组中。返回值 n 表示实际读到的字节的个数。
- stream.Close()，关闭文件流

可以发现，文件的写入和读取过程类似。区别是，在写入的时候调用OpenWrite()方法，表示以“写方式”来打开文件流。而在读取的时候，调用OpenRead()方法，表示以“读方式”来打开文件流。

在此情况下，文件长度为 8。所以 stream.Read()的返回值 n 为 8，并且 buffer 的前 8 个字节就是文件内容。

16.3 文件句柄的关闭

文件句柄，是程序开发时需注意的一个问题。所有打开的句柄，都应该及时关闭。此时的 FileSteam 对象，就对应了一个文件句柄。在程序时，需保证调用FileSteam.Close()方法来关闭文件流（及句柄）。

由于在文件读写过程中可能出现异常，所以我们可以用 try...finally...来确保Close()方法的调用。示例代码如下，

```
FileStream stream = fi.OpenWrite();
try
{
   byte[] data = { 1, 2, 3, 4, 0xA0, 0xB1, 0xC2, 0xC3 };
   stream.Write(data, 0, 8);
 }
 finally
 {
    stream.Close();
 }
```

其中，根据异常的语法，无论 try{} 中的语句有无异常发生，都会保证 finally{}中的语法执行。所以，stream.Close()方法总是执行，从而保证了无论何种情况文件句柄总是被关闭。

在 C#里，对资源的关闭也有专门的语法。例如，使用以下的代码也可以实现

相同的功能。

```
using (FileStream stream = fi.OpenWrite())
{
   byte[] data = { 1, 2, 3, 4, 0xA0, 0xB1, 0xC2, 0xC3 };
   stream.Write(data, 0, 8);
}
```

其中，using(stream) {... } 是一种异常的语法，小括号里为保护的资源。当{}中执行后，内部会保证在结束后会自动调用 stream.Close()方法。

提示，这个 using 语法是异常的语法，跟导包什么没有关系。这个语法对 Java 语言里的 try-with-resources 语法表达的是同一个意思。

16.4 托管和非托管

托管和非托管，是.NET Framework 中经常看到的术语。

- 托管（Managed）：由框架自动管理的资源
- 非托管 （Unmanaged）：由我们手工管理的资源

16.4.1 托管资源

什么资源是托管的呢？最常见的，当我们用 new 创建一个对象，该对象使用完毕后，就不需要管了。例如，

```
Student stu = new Student();
```

我们不需要关心 stu 对象的销毁，当对象失去引用时，会由系统框架会自动地回收销毁它（GC）。在 C#中的，这个.NET 的运行环境称为 CLR (Common Language Runtime，公共语言运行时)。它的作用相当于 Java 里的 JVM（Java 虚拟机）。

CLR 会自动管理所有托管的资源，在合适的时候将资源回收销毁。这也是 C#语言的优势，使得程序员可以少操点心。

16.4.2 非托管资源

那什么资源是非托管的呢？最常见的，文件句柄、窗口句柄或网络连接。

非托管的资源，CLR 是不负责管理的，需要程序员在用完之后自己把它销毁（Dispose）。例如，

```
// 如果目录不存在，则创建目录
FileInfo fi = new FileInfo("c:/example/data");
fi.Directory.Create();
// 1 打开文件流 (写方式)
FileStream stream = fi.OpenWrite();
 // 2 写入数据
byte[] data = { 1, 2, 3, 4, 0xA0, 0xB1, 0xC2, 0xC3 };
stream.Write(data, 0, 8);
// 3 销毁文件句柄
stream.Dispose();
```

文件流 FileStream 关联了一个文件句柄资源，它是非托管资源，所以我们需要在文件流用完之后，自行调用 Dispose()方法将文件句柄释放。

在.NET Framework 的 API 中，如果一个类实现了 IDisposable 接口，则标识了这个类中关联了非托管资源。对于这样的对象，我们在用完之后要确认手工调用 Dispose()方法来释放相关的资源（否则称为“资源泄露”）。

一句说总结：一个对象如果实现了 IDisposable 接口，那么它就包含了非托管资源，就得调用 Dispose()方法来手工释放资源。

第 17 章　XML

本章介绍 XML 的操作，在.NET Framework API 里已经包含了对 XML 的支持。相关的 API 在 System.Xml 这个命名空间下面。

17.1　创建 XML

创建 XML 时，主要使用以下几个相关的类。

XmlDocument ，表示一个 XML 文档

XmlDeclaration ，表示 XML 第一行

XmlElement ， 表示一个 XML 元素

XmlAttribute ， 表示一个元素的属性

下面看一个例子。

```
XmlDocument doc = new XmlDocument();
XmlDeclaration dec
     = doc.CreateXmlDeclaration("1.0", "utf-8", null);
doc.AppendChild(dec);

// <root>
XmlElement root = doc.CreateElement("root");
doc.AppendChild(root);;
```

这样就创建一个 XML 文档对象 doc，并添加了一个根元素。一个 XmlElement 表示一个元素节点，下面还可以添加更多的元素。

```
// <root>/<student>
XmlElement stu = doc.CreateElement("student");
root.AppendChild(stu);
```

```
// 节点属性
stu.SetAttribute("id", "20200001");

// <root>/<student>/<name>
XmlElement name = doc.CreateElement("name");
name.InnerText = "邵发";
stu.AppendChild(name);

// <root>/<student>/<sex>
XmlElement sex = doc.CreateElement("sex");
sex.InnerText = "1";
stu.AppendChild(sex);

// <root>/<student>/<author>
XmlElement author = doc.CreateElement("phone");
author.InnerText = "13810012345";
stu.AppendChild(author);

// 特殊字符转义（同 HTML）
XmlElement book = doc.CreateElement("book");
book.InnerText = "<C#基础篇>";
root.AppendChild(book);
```

最后，可以把 XML 文档输出到一个文件中。

```
// Save 方法有多个重载版本
// 1 输出到文件
doc.Save("student.xml");
// 2 输出为字符串
string xmlstr = doc.InnerXml;
Console.WriteLine("输出 XML: " + xmlstr);;
```

其中，如果要输出到一个文件，可以调用 doc.Save (filePath) 。如果要转成字符串，可以调用 doc.InnerXml 属性，会转成 XML 格式的字符串。

运行上述程序，得到的 XML 内容如下。

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <student id="20200001">
    <name>邵发</name>
    <sex>1</sex>
    <phone>13810012345</phone>
  </student>
</root>
```

需要注意的是，如果元素的值文本中包含 HTML 转义字符，API 会自动进行转义。比如，< 会转义为 < 。

17.2 解析 XML

下面介绍一下，怎么从一个 XML 里提取出信息。解析的过程仍然是使用 System.Xml 下的 API 来完成。

首先，给定一个 String 或者文件，可以调用 doc.Load(filePath) 或 doc.LoadXml(xmlstr) 方法加载进来。示例如下。

```
    XmlDocument doc = new XmlDocument();
    doc.Load("student.xml");
```

加载文件时，可以使用文件路径，也可以使用 FileStream。

然后，便可以从 doc 中得到提定的元素。比如，可以从 doc 中获取根元素。示例如下。

```
    XmlElement root = doc.DocumentElement;
```

也可以根据元素的层次路径，来获取指定的元素节点。调用 SelectSingleNode(xpath) 可以获得单个节点，使用 SelectNodes(xpath)可以获取多个同名的节点。例如，

```
XmlElement student
    = (XmlElement )root.SelectSingleNode("student");
```

使用索引器，可以快速地获取一个子元素的文本。例如，

```
string name = student["name"].InnerText;
string sex = student["sex"].InnerText;
string phone = student["phone"].InnerText;
```

使用 GetAttribute()方法可以获取元素的属性的值。例如，

```
int id = Convert.ToInt32(student.GetAttribute("id"));
```

其中，GetAttribute()的返回值是 string，应该视情况转成自己所需要的类型。

除此之外，还可以直接遍历一个元素的所有的子元素。

```
foreach(XmlNode node in node.ChildNodes)
{
}
```

其中，node.ChildNodes 属性可以获取 node 的所有子节点。

17.2.1 XPath

在此 API 中，把元素 XmlElement 和 属性 XmlAttribute 统称为节点 XmlNode。而一个节点的位置可以用 XPath 描述。比如，以下 XML 中的 name 元素的位置，可以用 "root/student/name" 来表示。

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <student id="20200001">
    <name>邵发</name>
    <sex>1</sex>
```

```
    <phone>13810012345</phone>
  </student>
</root>
```

可以使用 doc.SelectSingleNode (xpath) 来直接定位到一个节点。示例如下。

```
    XmlNode node
        = doc.SelectSingleNode( "root/student/name" )
```

关于 XPath 的格式描述，可以参考官方文档。

一个 XPath 可以描述一个元素，也可以指定一个元素的属性。如果这个路径是一个元素，则可以将结果转转为 XmlElement。如果这个位置指示的是一个属性，则可以转成 XmlAttribute。例如，

```
    XmlElement node = (XmlElement)
         doc.SelectSingleNode( "root/student/name" )
```

第 18 章　JSON

JSON，是一种比 XML 更常见的文本数据格式。.NET Framework 不带对 JSON 的支持，需要使用第三方库来实现。支持 JSON 的库有多种，本课程演示所用的库的名字是 Newtonsoft.Json。

18.1　JSON 的构造

按照第 14 章中的演示办法，把 Newtonsoft.Json.dll 加入到项目，便可以使用其 API 来生成和解析 JSON 文本了。

在 JSON API 中，用 JObject 表示一个 JSON 对象，JArray 表示一个 JSON 数组。JToken 是 JObject 和 JArray 的共同父类。

以下代码演示了怎么来生成一个 JSON 文本。

```
JObject j = new JObject();
j["id"] = 2020001;
j["name"] = "邵发";
j.Add("sex", true);
j.Add("phone", "13810012345");

JArray colors = new JArray();
colors.Add("red");
colors.Add("green");
colors.Add("blue");
j.Add("colors", colors);
```

其中，向 JObject 中添加字段有两种方式。可以用索引器方式，如 j["id"] = 20200001。也可以调用 Add 方法，如 j.Add("id", 2020001)。

最后，可以把构造出来的 JObject 的内容输出为文本。示例如下。

```
string jsonstr = j.ToString();
Console.WriteLine(jsonstr);
```

运行程序，输出如下。

```
{
  "id": 2020001,
  "name": "邵发",
  "sex": true,
  "phone": "13810012345",
  "colors": [
    "red",
    "green",
    "blue"
  ]
}
```

18.2 JSON 的解析

反过来，亦可以从一个 JSON 字符串中提取出数据。

例如，

```
JObject j2 = JObject.Parse(jsonstr);
```

其中，jsonstr 是一个字符串，调用 Parse 方法将其转成 JObject 对象。然后，就可以从 JObject 对象中提取各个字段的值了。

例如，使用索引器可以取得一个字段的值。

```
int id = (int)j2["id"];
bool sex = (bool)j2["sex"];
```

也可以使用 GetValue 方法来取得一个字段。

```
string name = (string)j2.GetValue("name");
```

使用 ContainsKey 方法可以判断一个字段是否存在。

```
    if(j2.ContainsKey("name"))
    {
        string name = (string)j2.GetValue("name");
        Console.WriteLine("姓名:" + name);
    }
```

如果一个字段是数组类型，也可以提取出来。示例如下。

```
JArray colors = (JArray)j2["colors"];
for (int i = 0; i < colors.Count; i++)
{
   string c = (string)colors[i];
   Console.WriteLine("color:" + c);
}
```

18.3 对象的序列化

在 C#里，把一个纯粹表示数据的类称为实体类，相当于 Java 里的 POJO 类。那么，怎么把一个实体类对象直接转成 JSON 文本呢？

18.3.1 实体类的定义

以下，定义了一个实体类 Student。

```
   class Student   {
      public int Id { get; set; }
      public string Name { get; set; }
      public bool Sex { get; set;  }
      public string Phone { get; set; }
   }
```

其中，在 Student 类中添加了 4 个属性，分别为 Id、Name、Sex、Phone。

也可以使用字段来定义实体类，并配上 Getter/Setter 方法。例如。

```
class Student
{
    // 属性
    public int Id { get; set; }
    public string Name { get; set; }
    public bool Sex { get; set;  }
    public string Phone { get; set; }

    // 字段 + Getter / Setter
    public string address;
    public string GetAddress()
    {
        return address;
    }
    public void SetAddress(string address)
    {
        this.address = address;
    }
}
```

其中，定义了 4 个属性，和 1 个字段: address。

可以发现，属性的定义在形式上要简洁一些，它是对 Getter/Setter 的简化写法。

18.3.2 对象与 JSON 的转换

使用 Newtonsoft.JSON，可以实现实体类对象和 JSON 的直接转换，称为序列化和反序列化。

序列化 (Serialize) ： Student -> JSON

反序列化(Deserilize) ： JSON -> Student

```
Student stu = new Student();
stu.Id = 20200001;
stu.Name = "邵发";
stu.Sex = true;
stu.Phone = "13810012345";
stu.SetAddress("蚌埠市");

// POJO <=> JSON String
string jsonstr = JsonConvert.SerializeObject(stu);
Student pojo
    = JsonConvert.DeserializeObject<Student>(jsonstr);

// POJO <=> JObject
JObject j2 = JObject.FromObject(stu);
Student s2 = j2.ToObject<Student>();
```

显然，Newtonsoft API 内部使用了反射技术。通过反射，取得一个对象的属性，并据此自动取值设值，完成和 JSON 的自动转换。

C#中的反射机制，和 Java 里的基本相同。关于反射技术，可以参考 Java 学习指南系列的《反射与框架篇》，里面有详细的讲解。

第 19 章　线程

本章介绍线程和互斥的实现。

19.1　线程的创建

要创建线程，首先定义线程的主方法。比如，

```
class MyCounter
{
    public void Run()
    {
        int i = 10;
        while(i> 0)
        {
            Console.WriteLine(i);
            i -= 1;

            Thread.Sleep(1000);
        }

        Console.WriteLine("倒计时线程退出");
    }
}
```

其中，在 MyCounter 类中定义一个方法 Run，此方法实现了一个倒计数的功能。每秒倒计数减 1，一共需要约 10 秒种完成。

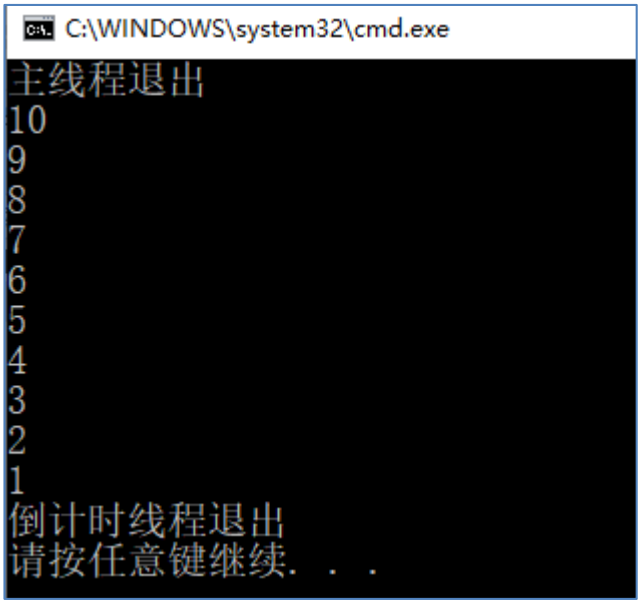
然后创建一个线程，以 MyCounter.Run 为主方法来运行。示例如下。

```
MyCounter counter = new MyCounter();
```

```
Thread t = new Thread(new ThreadStart(counter.Run));
t.Start();
```

其中，Thread 表示一个线程，ThreadStart 是一个委托方法。new ThreadStart(counter.Run) 个表示把 counter.Run 为主方法，交给线程来运行。

运行以上程序，则创建了一个线程，实现了倒计数的打印显示。效果如图所示。



19.2 线程的终止

在上节课中，调用 t.Start() 可以启动一个线程。

什么时候线程算是终止呢？很简单，当线程主方法运行结束，线程即告终止。在上例中，MyCounter.Run 方法开始运行，10 秒钟之后，Run 方法退出，线程也就自然终止了。

进一步地考虑：如果一个线程运行要很长时间，有没有办法中途强行终止呢？可以采用“杀死线程”、“取消线程”之类的 API 吗？例如，以下的线程主方法，

需要一万秒才能执行结束。

```
public void Run()
{
    for(int i=1; i <= 10000; i++) // 循环 10000 次
    {
        Console.WriteLine(i);

        Thread.Sleep(1000);
    }
    Console.WriteLine("计时线程退出");
}
```

注意，在多线程编程中，万不可以有“杀死”线程这样的想法。因为一个线程都是有它的工作任务的，任务正在进行中的时候，强行杀死它，则可能使整个系统进入不稳定的状态。甚至业务逻辑紊乱，数据损坏。

多线程设计的正确思路是，总是让线程“寿终正寝”，让它自然地退出。对于以上的线程任务，完全可能通过精巧的设计，控制它正常地退出。示例代码如下。

```
class MyCounter
{
    bool quitFlag = false; // 添加一个字段
    public void Run()
    {
        for(int i=1; i <= 10; i++)
        {
            if (quitFlag) break; // 中断循环，退出
            Console.WriteLine(i);
            Thread.Sleep(1000);
        }
```

```
            Console.WriteLine("计时线程退出");
        }
    }
```

其中，在 MyCounter 中添加一个字段 quitFlag，来控制 Run 方法的运行。如果要中止线程，只需要把 counter.quitFlag 设置为 true，则 Run 方法就会从 for 循环中退出。而 Run 方法一旦退出，线程也就自然退出了。

19.3 更多用法

19.3.1 Join() 等待线程退出

使用 Join()方法，可以等待一个线程的退出。示例如下。

要创建线程，首先定义线程的主方法。比如，

```
MyCounter counter = new MyCounter();


Thread t = new Thread(new ThreadStart(counter.Run));
t.Start();
t.Join(); // 等待线程退出
```

其中，t.Start()启动一个线程，t.Join()等待线程的终止。

Join()方法是阻塞，如果目标线程需要 10 秒钟结束，则程序会“卡”在这里 10 秒值。当目标线程结束之后，才会继续往下运行。

19.3.2 CurrentThread 当前线程

使用 Thread.CurrentThread 属性，可以获取当前线程的信息。具体参考视频演示。

19.3.3 IsBackground 后台线程

当 IsBackground 属性的值为 true 时，设置为后台线程。否则，默认为前台线程。示例代码如下。

```
MyCounter counter = new MyCounter();


Thread t = new Thread(new ThreadStart(counter.Run));
t.IsBackground = true;  // 设置为后台线程
t.Start();
```

当一个线程设置为后台线程时，此线程的“地位”变弱。

前台线程：当所有的线程结束时，整个进程才算退出。

后台线程：当进程退出时，后台线程会直接中断运行（也许会产生损坏数据）

所以，在决定使用后台线程前，一定得清楚后台线程的这个弱点，确定它不会损害业务逻辑的完整性。

19.4 线程句柄泄露

线程 Thread 对象中有没有非托管资源呢？根据 16.4 节的讲解，如果一个类有 Dispose 方法，则该类需要手工 Dispose。检查一下 API 文档，可以发现 Thread 类没有 Dispose 方法，所以创建的对象不含非托管资源。

也就是说，当一个线程终止之后，并不需要更多的操作。此线程对象交给 CLR 自动托管了，将由 CLR 来销毁此线程对象所关联的系统资源。

不过，在任务管理器中监视进程，可以发现它似乎有句柄泄露的问题。每次创建线程，当线程结束后，有 5 个句柄的遗留。（观看演示讲解）。

按 C#的官方文档，似乎这不是个问题。CLR (即.NET 框架的运行时)会在合适的时候触发 GC，在 GC 的时候会自动回收线程的句柄资源。简单地说，官方认

为我们不需要操心线程的句柄泄露问题。

如果一定纠结这个问题的话，可以创建一个定时器，定期调用 GC.Collect()方法。当 GC.Collect()执行时，会清理释放线程的残留句柄。

19.5 互斥锁

当多个线程同时访问一个资源时，则有必要使用互斥，以保证线程对资源的完整访问、不被打断。

在 C#中，可以用 lock{}语句，或者用 Mutex API 来实现互斥。

lock 语法和 Java 中 synchronized 语法类似。形如，

```
lock ( resourceLock )
{
    Access to the resource ...
}
```

其中，resource 为要保护的资源，resourceLock 为控制访问的资源锁。可以单独创建一个对象作为锁，也可以直接使用资源对象作为锁。示例如下。

```
    class MyFactory
    {
        List<int> repo = new List<int>();
        // 向 repo 仓库中添加产品
        public void Produce()
        {
            Random rand = new Random();
            while( true )
            {
                lock(repo)
                {
                    int number = rand.Next(10000);
```

```
                repo.Add(number);
                Console.WriteLine("添加: " + number);
            }
            // 暂歇一定时间
            Thread.Sleep(1000);
        }
    }
    public void Consume()
    {
        Random rand = new Random();
        while (true)
        {
            lock (repo)
            {
                if (repo.Count > 0)
                {
                    int number = repo[0];
                    repo.RemoveAt(0);
                    Console.WriteLine("取走: " + number);
                }
            }

            // 暂歇一定时间
            Thread.Sleep(500);
        }
    }
```

下面创建两个线程。一个线程来运行 MyFactory.Produce()，负责定时“生产”数据并保存到 repo 对象。另一个线程运行 MyFactory.Consume()，负责定时从 repo

中取出数据。示例代码如下。

```
MyFactory factory = new MyFactory();
// 生产者线程
Thread t1
   = new Thread(new ThreadStart(factory.Produce));
t1.Start();
// 消费者线程
Thread t2
  = new Thread(new ThreadStart(factory.Consume));
t2.Start();
```

则同时运行了 2 个线程 t1 和 t2，它们都要访问 factory.repo 这个对象。这是一个典型的互斥访问场景，必须采用互斥保护。

所以，在 t1 线程中，使用 lock 语法实现互斥保护。示例如下。

```
lock(repo)
{
  int number = rand.Next(10000);
  repo.Add(number);
  Console.WriteLine("添加: " + number);
}
```

在 t2 线程，也使用 lock 语法实现互斥保护。示例如下。

```
  lock (repo)
  {
      if (repo.Count > 0) {
          int number = repo[0];
          repo.RemoveAt(0);
          Console.WriteLine("取走: " + number);
      }
  }
```

第 20 章　线程池与定时器

本章介绍线程池和定时器的用法。线程池，可用于执行较短的任务。定时器，用于执行定时任务。

20.1　线程池

线程池(Thread Pool)，是一种池的设计。在池子中预备一些线程，当需要执行任务时，取出一个线程并让此线程执行某个任务。

ThreadPool 类提供线程池的支持。使用时，需要先定义一个任务，也就是一个回调方法。例如，

```
public static void SimpleJob(object state)
{
    for(int i=0; i<3; i++)
    {
        Console.WriteLine(i + 1);
        Thread.Sleep(1000);
    }
    Console.WriteLine("任务完毕");
}
```

有两种办法，一是使用前一章的技术，创建一个 Thread 来执行此任务。二是使用 ThreadPool，由线程池来调度执行此任务。示例代码如下。

```
ThreadPool.QueueUserWorkItem(new WaitCallback(SimpleJob));
```

其中，创建一个 WaitCallback 回调，交给线程池来执行。线程池内有一个队列，任务首先放以队列里排列。线程池内部会安排调度，当空闲时执行这个任务。

在线程池内部，会安排一个线程来执行这个任务。任务完成时，线程并不会

结束（里面有特殊的设计），会从队列里取出下一个任务继续执行。线程池一共有多少个工作线程呢，这个数量是可以设定的。

需要注意的是，线程池内创建的线程是后台线程。具体请回顾 19 章，明白后台线程的性质。将 ThreadPool 和 Thread 的区别罗列如下。

ThreadPool	Thread
适合较短小的任务	任务长任务和短任务
后台线程	默认为前台线程
自动管理，无法控制	手动管理，可以灵活控制

20.2 定时器

使用定时器，可以定时执行一些任务。比如，每分钟执行一次任务，每天执行一次任务等等。

在 .NET Framework API 里有 4 个 Timer 类，容易让人产生混淆。

System.Timers.Timer

System.Threading.Timer

System.Windows.Forms.Timer

System.Web.UI.Timer

其中，前两个的功能和用法是类似的。下面介绍一下 System.Threading.Timer 的用法。

首先，还是要定义一个回调方法，作为要执行的任务。示例如下。

```
public static void SimpleJob(object state)
{
    for (int i = 1; i <= 3; i++)
    {
```

```
            Console.WriteLine( i );
            Thread.Sleep(350);
        }
        Console.WriteLine("... OK ");
    }
```

然后，创建一个定时器，每隔 1 秒钟执行一次 SimpleJob()。示例如下。

```
Timer timer = new Timer(new TimerCallback(SimpleJob)
            , null
            , 1000
            , 1000);
```

其中，创建 Timer 的时候要指定 4 个参数。

new Timer(callback, state, dueTime, period)

分别为：

callback : 任务回调

state : 任务参数

dueTime : 首次执行的时间延迟

0 , 立即开始执行

N , N 毫秒之后执行

Timeout.Infinite　禁用

period : 周期运行的时间间隔

0 , 禁用

N, N 毫秒周期间隔

Timeout.Infinite, 禁用

需要注意的是，时间间隔应大于操作系统的最小时间精度(约这 15ms)。如果太小，则没什么意义。比如，把间隔设为 3ms 和 15ms 的实际效果是等同的。

20.2.1 定时器的要点

(1) 定时器的精度限制

时间间隔最小 15ms 左右，更小的值没有意义。

(2) 实际使用的是 ThreadPool 来执行的任务

(3) 任务的叠加

如果一个任务执行时间太长，可能造成任务的叠加。每个任务是在不同的线程中运行的。定时器回调方法的的设计要求：可重入、可叠加。

(4) 定义器含有非托管资源，用完后需要手工 Dispose() 。

后续课程

基础篇的语法可以快速过一遍，进入下一篇课程的练习。

下一篇课程为桌面应用的开发教程，请从官网进入。

http://afanihao.cn/cs