	文 档 编 号	产品版本	密级
	WS-080211	3.0	内部
	项目名称：C#3.0 本质论		共 70 页

C#3.0 本质论

技 术 参 考 书

业务名称	C#3.0 本质论
总体规划	王豫翔
方案编制	王豫翔　江海
教学监察	王豫翔
质量控制	傅晓霞
审校和存档	傅晓霞
上级监管	国家劳动和社会保障部中国就业促进会
教学机构	华大锐志 IT 实训中心（华师大古北校区）
编制日期	2008 年 2 月 25 日

写在扉页

华大锐志的技术专家告诉您：

学习计算机语言和学习人类语言一样，刚开始我们鼓励首先通过大量的示例开始学习，因为向别人学习是一种很自然的方式，可以使学习变得轻松有趣。

但是模仿也只能带你走这么远。示例的方法既有优点，也存在缺点。通过示例来学习，只能学到一些表面的东西，不能达到彻底全面掌握的境界。要想做到真正的“熟练”，必须要学会如何在各种不同的情况下使用语言，这就需要学习理论知识来进行辅助，同时通过大量的练习和实战来巩固学习的效果。

另外还要记住，计算机的语言比人类语言精确的多，必须正确使用语法，否则它就不会起到应有的作用。在已经能熟练地使用语言后，如果还想要成为该语言的真正专家，就必须形成自己的风格：这就意味着不仅要知道什么是合适的，更要知道什么是有效的，同时还必须要考虑他人的阅读、维护和效率等问题。我们编写这一系列教程的目的，正是为了帮助你熟练掌握计算机语言，精通语法、语义、主流工具、经验技巧以及其他各方面能使你成为“专家”的一切。

我们采用最自然的学习模式，也就是通过“全程案例”来进行教学。通过案例推导原理，从简单到复杂、从基础到高级、从实践到理论，我们将陪伴你走过这个美妙的体验过程。我们将告诉你哪些是可行，哪些是不可行；我们也知道哪些能让你兴奋莫名，哪些将让你迷惑苦恼；我们将为你高兴，给你鼓励！

为了让你更快、更快乐地学习编程，刚开始我们也许会忽略一些概念或你应该知道的技巧，但随着示例的深入，你将会学到所有你应该知道的一切——这个就是涡轮式的教学法的妙不可言之处。

华大锐志将帮助你实现你的梦想 —— 对这一点，我们非常坚信！

目 录

一、C#和.NET Framework 概览

Microsoft Visual C# 2008（读作 C sharp）是一种编程语言，它是为生成在 .NET Framework 上运行的多种应用程序而设计的。C# 简单、功能强大、类型安全，而且是面向对象的。C# 凭借它的许多创新，在保持 C 样式语言的表示形式和优美的同时，实现了应用程序的快速开发。

Visual Studio 支持 Visual C#，这是通过功能齐全的代码编辑器、项目模板、设计器、代码向导、功能强大且易于使用的调试器以及其他工具实现的。通过 .NET Framework 类库，可以访问多种操作系统服务和其他有用的精心设计的类，这些类可显著加快开发周期。

C# 是一种简洁、类型安全的面向对象的语言，开发人员可以使用它来构建在 .NET Framework 上运行的各种安全、可靠的应用程序。使用 C#，您可以创建传统的 Windows 客户端应用程序、XML Web services、分布式组件、客户端/服务器应用程序、数据库应用程序等等。Visual C# 2008 提供了高级代码编辑器、方便的用户界面设计器、集成调试器和许多其他工具，使您可以更容易在 C# 语言 3.0 版和 .NET Framework 3.5 版的基础上开发应用程序。

1.1 为什么要学习 C#

C# 语法表现力强，而且简单易学。C# 的大括号语法使任何熟悉 C、C++ 或 Java 的人都可以立即上手。了解上述任何一种语言的开发人员通常在很短的时间内就可以开始使用 C# 高效地进行工作。C# 语法简化了 C++ 的诸多复杂性，并提供了很多强大的功能。还提供了 Java 所不具备的很多优秀的语言能力，例如可为 null 的值类型、枚举、委托、lambda 表达式和直接内存访问。并且 C# 支持泛型方法和类型，从而提供了更出色的类型安全和性能。C# 还提供了迭代器，允许集合类的实施者定义自定义的迭代行为，以便容易被客户端代码使用。在 C# 3.0 中，语言集成查询 (LINQ) 表达式使强类型查询成为了一流的语言构造。

1.2C#的创新特色

作为一种面向对象的语言，C# 支持封装、继承和多态性的概念。所有的变量和方法，包括 Main 方法（应用程序的入口点），都封装在类定义中。类可能直接从一个父类继承，但它可以实现任意数量的接口。重写父类中的虚方法的各种方法要求 override 关键字作为一种避免意外重定义的方式。

在 C# 中，提供了一个 java 不支持的代码体“结构”，其类似于一个轻量类；它是一种堆栈分配的类型，可以实现接口，但不支持继承。

除了这些基本的面向对象的原理之外，C# 还通过几种创新的语言构造简化了软件组件

的开发，这些结构包括：

封装的方法签名（称为“委托”），它实现了类型安全的事件通知。

属性 (Property)，充当私有成员变量的访问器。

属性 (Attribute)，提供关于运行时类型的声明性元数据。

内联 XML 文档注释。

语言集成查询 (LINQ)，提供了跨各种数据源的内置查询功能。

在 C# 中，如果必须与其他 Windows 软件（如 COM 对象或本机 Win32 DLL）交互，则可以通过一个称为“互操作”的过程来实现。互操作使 C# 程序能够完成本机 C++ 应用程序可以完成的几乎任何任务。在直接内存访问必不可少的情况下，C# 甚至支持指针和“不安全”代码的概念。

C# 的生成过程比 C 和 C++ 简单，比 Java 更为灵活。没有单独的头文件，也不要求按照特定顺序声明方法和类型。C# 源文件可以定义任意数量的类、结构、接口和事件。

1.2 什么是.NET Framework 平台体系

.NET Framework 是支持生成和运行下一代应用程序和 XML Web Services 的内部 Windows 组件。.NET Framework 旨在实现下列目标：

提供一个一致的面向对象的编程环境，而无论对象代码是在本地存储和执行，还是在本地执行但在 Internet 上分布，或者是在远程执行的。

提供一个将软件部署和版本控制冲突最小化的代码执行环境。

提供一个可提高代码（包括由未知的或不完全受信任的第三方创建的代码）执行安全性的代码执行环境。

提供一个可消除脚本环境或解释环境的性能问题的代码执行环境。

使开发人员的经验在面对类型大不相同的应用程序（如基于 Windows 的应用程序和基于 Web 的应用程序）时保持一致。

按照工业标准生成所有通信，以确保基于 .NET Framework 的代码可与任何其他代码集成。

.NET Framework 具有两个主要组件：公共语言运行库和 .NET Framework 类库。公共语言运行库是 .NET Framework 的基础。您可以将运行库看作一个在执行时管理代码的代理，它提供内存管理、线程管理和远程处理等核心服务，并且还强制实施严格的类型安全以及可提高安全性和可靠性的其他形式的代码准确性。事实上，代码管理的概念是运行库的基本原则。以运行库为目标的代码称为托管代码，而不以运行库为目标的代码称为非托管代码。.NET Framework 的另一个主要组件是类库，它是一个综合性的面向对象的可重用类型集合，您可以使用它开发多种应用程序，这些应用程序包括传统的命令行或图形用户界面 (GUI) 应用程序，也包括基于 ASP.NET 所提供的最新创新的应用程序（如 Web 窗体和 XML Web Services）。

.NET Framework 可由非托管组件承载，这些组件将公共语言运行库加载到它们的进程中并启动托管代码的执行，从而创建一个可以同时利用托管和非托管功能的软件环境。.NET

Framework 不但提供若干个运行库宿主，而且还支持第三方运行库宿主的开发。

例如，ASP.NET 承载运行库以为托管代码提供可伸缩的服务器端环境。ASP.NET 直接使用运行库以启用 ASP.NET 应用程序和 XML Web Services（本主题稍后将对这两者进行讨论）。

Internet Explorer 是承载运行库（以 MIME 类型扩展的形式）的非托管应用程序的一个示例。使用 Internet Explorer 承载运行库使您能够在 HTML 文档中嵌入托管组件或 Windows 窗体控件。以这种方式承载运行库使得托管移动代码（类似于 Microsoft® ActiveX® 控件）成为可能，不过它需要进行重大改进（如不完全受信任的执行和独立的文件存储），而这种改进只有托管代码才能提供。

1.3 关于 C#我们应该学习哪些内容

C#是一个非常广泛应用的语言，虽然很难说明你必须要学哪些，但华大锐志通过长期的考察，为中国学员提供了一套优秀可行的 C#学习模式。

依据涡轮式的学习理念，华大锐志把 C#的学习内容分为两大部分：核心开发内容和高级开发技术。

1.3.1 核心开发内容

在核心开发内容中，主要是对.NET Framework C#中标准应用程序开发任务的和.NET Framework C#的深入体验和熟练。在这个层次中，华大锐志将告诉你以下知识。

内容提要	注释
开发基础	有关应用程序开发中使用的基本概念和技巧
配置应用程序	对各种类型的配置文件应用设置
数据访问	用 ADO.NET、SQL Server 和 XML 使用.NET 信息
调试和分析应用程序	测试、优化和配置 .NET 应用程序和应用程序环境
部署应用程序	创建自我描述、自我包含的应用程序
编码和本地化	为开发国际性应用程序所提供的全面的支持
基于服务器的组建技术	使用文件系统、安装、消息和系统监视组件
并行执行	运行应用程序、组件或整个运行库的多个副本

1.3.2 高级开发技术

内容提要	注释
管理	使用目录服务和 WMI
组建创作	创建自定义组件以及如何扩展设计时支持
异步编程设计模式	.NET Framework 中的异步编程功能
网络编程	实现基于 Web 的应用和基于 Internet 的应用
反射	通过使用反射在运行时访问类型信息
可靠性编程	面向可靠性的属性，编写可靠代码的最佳做法
序列化	XML 序列化和二进制序列化
线程处理	使用各种同步技术编程

1.3.3 如何学习.NET C#

计算机技术的学习就象我们学习其他技能一样，比如英语、数学、物理化学甚至语文，我们很难说清楚，某些内容一定是基础内容，某些内容一定是高级内容。之所以将技术分类为核心开发内容和高级开发技术，只是为学习提供一个阶段性的提示。

另外一个很重要的概念是：每一项的内容并不是独立静止的存在它所出现的教学阶段上。教学阶段只是对这个技术内容进行提炼性的描述。而往往你会看到这个技术点在其教学阶段之前就曾经出现过。更令你惊奇的是，这个技术内容在后期或将更加频繁的出现，并且该技术还总是在试图的表现其不同的运用方式和实现能力。而你在这个反复的接触过程中，常常会有恍然大悟的感觉，这就是优秀的涡轮式学习所能带给你的惊奇体验。

大量的练习是必不可少的，对每一个知识点都必须要通过海量的代码来完成你和它之间的心灵感应。但写代码不要把他作为一个很痛苦无奈的工作，你要挖掘其中的乐趣，至少你要知道，计算机是非常听你的指挥，你想让它作什么它就作什么，只要你的命令是有效的。

那么我们开始进入 C#的世界吧。

1.4 善用你身边的宝典：MSDN

很多初学者，对计算机市场最大的贡献不是购买设备和软件，而是买书。凡是能吸引他眼球的书比如：宝典，21 天从入门到精通等等。

从涡轮式培训的角度来看，我们不赞赏这样的购买行为，微软为开发人员提供了真正的宝典：MSDN。MSDN 的全称是 Microsoft Developer Network。这是微软公司面向软件开发

者的一种信息服务。在这个文档集合中，所有关于类和方法的详细信息你都可以被找到。所以我们这份教案不再和其他的市场上培训机构的教案那样，充满着属性和方法的详细表格。我们更多的专注在案例的实训。

1.5 本阶段的目标

二、一切从控制台应用程序开始

学习过 VB 的程序员，总是很喜欢从窗口开始编程。学习过 WEB 网页设计的程序员也喜欢充一个页面开始。从现在开始，请放弃原先那些不好的习惯，让我们从最有效的训练：控制台应用程序开始。

2.1 未能免俗：Hello, World

虽然说写一个 Hello, World 是一个很俗的代码开始，但不得不承认，要写一个最简单短小程序来表示一个编程语言的基本风格，还是 hello, World 比较适合。那我们来看下 C#版的 Hello, World。

Hello, World

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

namespace ConsoleApplication1
{
    /*
     描述了一个简单的C#应用
     */
    class Program
    {
```

```
        /// <summary>
        /// 类的入口函数
        /// </summary>
        /// <param name="args">用于激活程序的命令行参数</param>
        static void Main(string[] args)
        {
            //向控制台输出字符串：Hello,World
            System.Console.WriteLine("Hello,World");//使用了命名空间的全引用
            Console.WriteLine("C#");//省略了命名空间的输入
        }
    }
}
```

上面这段 C#代码尽可能的向我们描述了 C#程序的一些特征。

//和/**/是注释。其中//是单行的注释，/**/是允许跨行的多行注释。

```
    /*
     描述了一个简单的C#应用
     */
//向控制台输出字符串：Hello,World
```

using 在首行的含义是引入相关的命名空间，这样在代码的正文中可以省略对引入的命名空间的输入。

```
using System;
System.Console.WriteLine("Hello,World");//使用了命名空间的全引用
Console.WriteLine("C#");//省略了命名空间的输入
```

namespace 是声明你自己的命名空间，在不同的命名空间下可以有同名的类。命名空间的{}中可以包含多个类或其他声明。

```
namespace ConsoleApplication1
{
}
```

class 是声明类，在 C#中，所有的函数、变量都是写在类中。

```
class Program
{
}
```

///文档注释标记，传统上来说编译器是不理会注释的内容的，但 C#的编译器会把约定的文档注释标记编译到与代码相关的 XML 资源中。但文档标注对命名空间无效。

```
/// <summary>
/// 类的入口函数
/// </summary>
```

```
/// <param name="args">用于激活程序的命令行参数</param>
```

函数是 C#编写代码执行的地方，类的函数有几个部分组成：访问修饰（默认为私有）、修饰、返回值、函数名称、参数列表

```
static void Main(string[] args)
{
}
```

语句是 C#的执行，必须要符合 C#的语法规范

```
//向控制台输出字符串：Hello,World
System.Console.WriteLine("Hello,World");//使用了命名空间的全引用
Console.WriteLine("C#");//省略了命名空间的输入
```

2.2 程序的入口点：Main

Main 方法是程序的入口点，您将在那里创建对象和调用其他方法。一个 C# 程序中只能有一个入口点。程序控制在该方法中开始和结束。该方法在类或结构的内部声明。它必须为静态方法，而不应为公共方法。（在上面的示例中，它接受默认访问级别 private。）Main 方法有几种不同的表现模式。

没有入口参数

```
static void Main()
{
}
```

有入口参数

```
static void Main(string[] args)
{
}
```

返回 int 类型（必须有 return 返回一个整数数值）

```
static int Main(string[] args)
{
    return 0;
}
```

返回 void 类型

```
static void Main(string[] args)
{
}
```

以上四种模式：有参无参、int 或 void 可以组合使用。

2.3 概览 C#的程序结构

在了解了 C#的最基本的概念后，我们概览下 C#程序的程序结构。C# 程序可由一个或多个文件组成。每个文件都可以包含零个或零个以上的命名空间。一个命名空间除了可包含其他命名空间外，还可包含类、结构、接口、枚举、委托等类型。以下是 C# 程序的主干，它包含所有这些元素。

C#程序结构概览

```
//这个命名空间包含了对生物基类的描述
namespace OrganismBase
{

    /// <summary>
    /// 动作的抽象类
    /// </summary>
    public abstract class Action
    {
    }
    /// <summary>
    /// 攻击行为，从Action继承
    /// </summary>
    public class AttackAction : Action
    {
    }

    /// <summary>
    /// 生物的抽象类
    /// </summary>
    public abstract class Organism
    {
    }
    /// <summary>
    /// 动物的抽象类，从Organism继承
    /// </summary>
    public abstract class Animal : Organism
    {
    }
    /// <summary>
    /// 动物外观的枚举
    /// </summary>
    public enum AnimalSkinFamily
```

```
    {
    }
    /// <summary>
    /// 生物的状态
    /// </summary>
    public abstract class OrganismState : IComparable
    {
    }
    /// <summary>
    /// 动物的状态，从OrganismState继承，密封类，不可以再继承
    /// </summary>
    public sealed class AnimalState : OrganismState
    {
    }
    /// <summary>
    /// 向量
    /// </summary>
    public struct Vector
    {
    }
    /// <summary>
    /// 攻击事件发生时
    /// </summary>
    /// <param name="sender">事件引发的对象</param>
    /// <param name="e">攻击信息参数</param>
    public delegate void AttackedEventHandler(object sender, AttackedEventArgs e);

}
```

上面的代码演示了 C#中所有的结构，我们来理解下 C#中关于结构的关键字。

关键字	含义
namespace	命名空间：隔离和帮助组织类的名称或方法名称的范围
class	类：定义了数据类型的数据和行为
Struct	结构：轻量级的类
Interface	接口：定义类或结构的一组相关功能
Delegate	委托：一种引用方法类型
enum	枚举：常量集

2.4 写入标准输出流：WriteLine 和 Write

我们要向控制台输出文字，需要使用 Console 的 WriteLine 和 Write 方法。这两个方法有非常灵活的模式。

WriteLine 和 Write 方法体验

```
class Program
{
    /// <summary>
    /// 类的入口函数
    /// </summary>
    /// <param name="args">用于激活程序的命令行参数</param>
    static void Main(string[] args)
    {
        System.Console.WriteLine("华大锐志");//输出后跟当前行终止符
        System.Console.Write("华大锐志");//输出后无当前行终止符
        System.Console.Write("华大锐志\n");//通过转义符实现回车
        System.Console.Write("华大锐志--{0}\n","专为中国学生设计");
        System.Console.Write("华大锐志--{1}{0}{2}", "\"涡轮式项目实训\"","推进国家人才战略
","\n");
    }
}
```

*参数从 0 开始计位

C#的 WriteLine 和 Write 方法非常的灵活，特别是参数占位的模式能为程序员的开发带来很多方便。

2.5 结构化语句

面向对象和面向过程的基础程序控制都是使用结构化语法进行控制。结构化语法一共三种：赋值运算、循环迭代、条件选择。除了这三种，C#还提供了：跳转、异常、检查以及锁语句。

2.5.1 赋值运算

在 C#中对所有的变量都必须要求先声明，再使用。且必须保证在使用前，该变量有安全的赋值。

变量表示数值或字符串值或类的对象。变量存储的值可能会发生更改，但名称保持不变。变量是字段的一种类型。

常数是另一种类型的字段。它保存在编译程序时赋予的值，并且从那之后在任何情况下都不会发生更改。

只读变量类似于常数，但其值是在程序启动时赋予的。这允许您根据某些在程序运行后才知道的其他条件设置值。但在第一次赋值之后，在程序运行的过程中不能再次更改该值。

赋值运算体验

```
class Program
{
    /// <summary>
    /// 类的入口函数
    /// </summary>
    /// <param name="args">用于激活程序的命令行参数</param>
    static void Main(string[] args)
    {
        long l;//声明长整形l
        int i = 100;//声明整形i，并赋值其100
        l = i;//将i的值赋给l
        const int SIZE = 99;//声明一个常数SIZE
        i += 10;//i=i=10;
        System.Console.WriteLine(i++);//输出的时候i的值没有变化，输出后i被＋1
        System.Console.WriteLine(i);
        System.Console.WriteLine(++i);//i先被＋1；然后输出
        System.Console.WriteLine(10 % 3);//取模
    }

    readonly string name = "华大锐志";//声明一个只读变量，在被赋值后不可以被更改
    static readonly DateTime now;//声明一个只读变量，该变量还没有赋值
}
```

＊赋值运算符+=

2.5.2 条件选择控制

C#提供两套逻辑判断控制。

语句	控制能力
if-else	单进双出。对于条件的判断真假给予两个选择。
switch-case	处理多个选择和枚举的单进多出
?:	简约的 if-else，表达式？true：false
??	条件赋值，?? 的左非 null，返回左，否则返回右

if-else

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

    if (System.DateTime.Now.Hour < 4 || DateTime.Now.Hour > 23)
    {
        Console.WriteLine("你是夜猫子？工作也太辛苦了吧。现在是{0}点了", DateTime.Now.Hour);
    }
    else
    {
        Console.WriteLine("现在是{0}点，你在干什么？", DateTime.Now.Hour);
    }
}
```

switch

```
static void Main(string[] args)
{
    System.Console.WriteLine("请选择你感兴趣的C#技术方向");

System.Console.WriteLine("1{0}--2--{1}--3{2}--4{3}--5{4}","00","ASp.Net","linq","wwf","wpf")
;

    int choose =int.Parse(System.Console.ReadLine());

    switch(choose)
    {
        case 1:
            System.Console.WriteLine("C#面向对象的实现");
            goto default;
        case 2:
            System.Console.WriteLine("ASP.Net");
            break;
        case 3:
            System.Console.WriteLine("LIQN");
```

```
            break;
        case 4:
            System.Console.WriteLine("Windows Workflow Foundation");
            break;
        case 5:
            System.Console.WriteLine("Windows Presentation Foundation (SilverLight)");
            break;
        default:
            System.Console.WriteLine("设计模式");
            break;
    }
}
```

简约条件

```
static void Main(string[] args)
{
    int? x = 10;//用?修饰int，则该int允许存放null
    int? y = 12;
    if (x < y)
    {
        x = y;
    }
    System.Console.WriteLine(x);//12
    y = x < y ? y : null;
    System.Console.WriteLine(y);//null
    x = y ?? 1000;
    System.Console.WriteLine(x);//1000
}
```

*DateTime 结构
*可以赋 null 的值

switch-case 语句要求每一个 case 块都必须有跳出的控制，但有一种模式可以例外。

```
static void Main(string[] args)
{
    int i = 0;

    switch (i)
    {
        case 1:
        case 2:
```

```
        case 3:
            System.Console.Write(i);
            goto case 5;
        case 5:
            System.Console.Write(i);
            goto default;
        default:
            System.Console.Write("default");
            break;
    }
}
```

2.5.3 迭代循环控制

C#在迭代循环语句中提供了多种模式。

语句	控制能力
while	不确定循环，先判断再执行
do	不确定循环，先执行再判断
for	确定循环
foreach	遍历迭代

while

```
static void Main(string[] args)
{
    const double HOUSEFOUND = 200000;//购房的首付款20W,常数
    double salary = 2000;//第一年的工资/月
    double fund = 0;//住房基金
    int years = 1;//年计数器
    while (fund < HOUSEFOUND)//存的基金不够首付,就继续存
    {
        fund += salary * 0.05 * 12;//工资的5%是交纳住房基金
        years++;//加一年
        salary = salary * 1.1;//每年工资增加10%
    }
    System.Console.WriteLine("哎，钱攒了{0}年了", years);
```

```
    Console.WriteLine("终于攒到了{0},命苦啊", salary.ToString("f0"));
}
```

do

```
static void Main(string[] args)
{
    const double HOUSEFOUND = 200000;//购房的首付款
    double salary = 2000;//第一年的工资/月
    double fund = 0;//住房基金
    int years = 1;//年计数器
    do
    {
        fund += salary * 0.05 * 12;//工资的5%是交纳住房基金
        years++;//加一年
        salary = salary * 1.1;//每年工资增加10%
    } while (fund < HOUSEFOUND);//这里一定要用;

    System.Console.WriteLine("哎, 钱攒了{0}年了", years);
    Console.WriteLine("终于攒到了{0},命苦啊", salary.ToString("f2"));//格式化数字
}
```

for

```
static void Main(string[] args)
{
    int result = 0;
    for (int i = 1; i <= 100; i++)
    {
        //1 + 2 + 3 = 4 + 5 + ... + 97 + 98 + 99 + 100
        result += i;
    }
    System.Console.WriteLine(result);
}
```

for each

```
static void Main(string[] args)
{
    //在项目属性——调试——命令行参数加入 字符
    foreach(string arg in args)
    {
        System.Console.WriteLine(arg);
    }
}
```

*自增与自减

*args 的运用

do 和 while 是不确定循环，要小心出现死循环。for 是确定循环，但最好不要在循环体内对计数变量进行操作。foreace 是用于对集合内部遍历，但不可改变集合的内容。

2.5.4 跳转控制

语句	控制能力
break	终止最近的封闭循环或它所在的 switch 语句
continue	控制权传递给它所在的封闭迭代语句的下一次迭代
default	没有任何 case 匹配，则控制传递给 default 后的语句
goto	将控制传递给特定的 switch-case 标签
return	终止方法（函数），可返回值
yiled	在迭代器块中用于向枚举数提供值或发出结束信号

跳转的运用

```
static void Main(string[] args)
{
    for (int i = 0; i < 1000; i++)
    {
        switch (i)
        {
            case 1:
                System.Console.WriteLine(i);
                break;
            case 2:
                System.Console.WriteLine(i);
                break;
            case 50:
                do
                {
                    if (i > 60)
                    {
                        break;
                    }
                    else
                    {
                        System.Console.WriteLine(i);
```

```
                }
                i++;
            }
            while (i < 70);
            break;
        default:
            while (i < 100)
            {
                i++;
                if (i < 80)
                {
                    continue;
                }
                else
                {
                    System.Console.WriteLine(i);
                }
            }
            break;
    }
}
```

对上述代码仔细分析，思考得到的结果是什么。同时也要理解，代码的清晰和可阅读性是非常重要的工作。

2.5.5 异常

计算机语言是一种精确控制的语言，如果在设计和运行的过程中出现错误，就会导致编译或运行错误。在程序的设计过程中出现的语法等错误，将在编译期间被发觉，从而无法正常编译。优秀的编译器还将友好的提示程序员进行错误修改。

但编译正确的程序，仅仅表示语法的正确，在后期的执行中还将会遇到两种错误：逻辑错误和不可抗拒的意外。

逻辑错误往往是程序员未能理解设计要求所引起的。比如：用户要求验证输入的字符是不是一个合法的财务数值，程序员忘记检查小数点只能有一个这个逻辑了。

不可抗拒的意外往往是程序在执行的过程中和外部数据交互时产生的错误。比如：用户为除数输入了一个 0；在网络的传输中网络意外停止；在磁盘写入的过程中磁盘被写满；被剪切的文件由其他应用程序在使用等等。

这些错误如果要用逻辑方式来处理和避免，将让代码非常的庞大，复杂和不可阅读。

语句	控制能力
try	包含可能导致异常的保护代码
catch	捕获异常
finally	清除 try 块中的资源，即使发生异常时也执行的代码

没有用异常的方式

```
static void Main(string[] args)
{
    if (args.Length < 2)//用户在命令行输入的参数少于2个
    {
        System.Console.WriteLine("对不起，至少需要两个参数");
        return;
    }
    for (int i = 0; i <= 2 - 1; i++)//遍历前两个参数
    {
        for (int j = 0; j <= args[i].Length - 1; j++)//args[i]代表第i个参数
        {//遍历第i个参数中的每一个字符
            if (args[i][j] < '0' || args[i][j] > '9')
            {
                System.Console.WriteLine("对不起，你输入的字符串格式错误，不是有效果的数字格式");
                Console.WriteLine("错误发生在");
                Console.WriteLine("第{0}个参数的，第{1}个字符，该参数的完整形式为：{2}", i + 1, j + 1, args[i]);
                return;
            }
        }
    }//如果可以执行到这里，说明两个参数都是符合整数格式
    int x = int.Parse(args[0]);//把第一个参数从字符串转为整数
    int y = int.Parse(args[1]);//把第二个参数从字符串转为整数
    System.Console.WriteLine("第1个参数是{0}，第2个参数是{1}",x,y);
}
```

＊Parse 方法

使用异常的方式

```
static void Main(string[] args)
{
    int? x = null;//用?修饰int，则该int允许存放null
    int? y = null;
```

```
    try//以下范围可能会发生错误
    {
        x = int.Parse(args[0]);//把第一个参数从字符串转为整数
    }
    catch (System.Exception e1)//如果有错误，则执行这里
    {
        System.Console.WriteLine(e1.Message);
    }
    finally//无论有没有发生错误，这里的代码总是要执行
    {
        try
        {
            y = int.Parse(args[1]);//把第二个参数从字符串转为整数
        }
        catch (System.Exception e1)
        {
            System.Console.WriteLine(e1.Message);
        }

    }
    if (x != null && y != null)
    {
        System.Console.WriteLine("第1个参数是{0}，第2个参数是{1}", x, y);
    }
}
```

*数组的概念

2.5.6 检查

不同的数据能包含的最大数据容量是不一样的。在 C#中可以通过以下代码来展示个数据的最大和最小值的范围。

语句	控制能力
checked	用于对整型算术运算和转换显式启用溢出检查
unchecked	取消整型算术运算和转换的溢出检查

数据范围
static void Main(string[] args)

```
{
    Console.WriteLine("byte:{0}-{1}", byte.MinValue, byte.MaxValue);
    Console.WriteLine("short:{0}-{1}", short.MinValue, short.MaxValue);
    Console.WriteLine("int:{0}-{1}",int.MinValue,int.MaxValue);
    Console.WriteLine("long:{0}-{1}", long.MinValue, long.MaxValue);
    Console.WriteLine("float:{0}-{1}", float.MinValue, float.MaxValue);
    Console.WriteLine("double:{0}-{1}", double.MinValue, double.MaxValue);
    Console.WriteLine("char:{0}-{1}", char.MaxValue, char.MaxValue);
}
```

那如果赋给一个数据的值超过了其可以容纳的最大范围，那会出现什么问题？

溢出赋值

```
static void Main(string[] args)
{
    Console.WriteLine("byte:{0}-{1}", byte.MinValue, byte.MaxValue);//0-255
    byte b = byte.MaxValue;
    b += 2;//不可以写成b=b+2
    System.Console.WriteLine(b);//1
}
```

如果，你不希望编译器如此宽松的检查处理，那你就需要用检查控制来辅助。

检查

```
static void Main(string[] args)
{
    Console.WriteLine("byte:{0}-{1}", byte.MinValue, byte.MaxValue);//0-255
    byte b = byte.MaxValue;

    try
    {
        b = checked(b += 2);//需要检查的时候，建议用try来保护异常
        System.Console.WriteLine(b);
    }
    catch(Exception e1)
    {
        System.Console.WriteLine(e1.Message);
    }

    b = unchecked(b += 2);
    System.Console.WriteLine(b);//1
}
```

2.5.7 锁

C#支持多线程处理，但在多线程中，需要考虑资源的抢占和冲突。

单线程的机票订购

```
class Program
{
    static void Main(string[] args)
    {
        while (Account.airTicket > 0)
        {
            new Account().BuyTicket(3);
            System.Console.WriteLine("－－－剩余机票{0}", Account.airTicket);
        }
    }
}

public class Account
{
    //使用static的含义是这个airTicket是个共享的数据
    public static int airTicket = 10;//初始飞机票的数量

    /// <summary>
    /// 买飞机票
    /// </summary>
    /// <param name="count">需要购买的数量</param>
    public void BuyTicket(int count)
    {
        if (airTicket <= 0)//如果机票没有了
        {
            return;
        }
        if (airTicket >= count)//如果剩余机票够
        {
            airTicket -= count;
            System.Console.Write("{0}成功购买{1}", DateTime.Now, count);
        }
```

```
            return;
        }
    }
```

*类的实例化

上述的机票看上去似乎很稳定，但在实际过程中，不可能有如此线性的购买行为。

模拟真实的多窗口买票

```
class Program
{
    static void Main(string[] args)
    {
        //Thread是多线程
        System.Threading.Thread[] clients = new System.Threading.Thread[100];//有100个窗口
        for (int i = 0; i <= clients.Length - 1;i++ )
        {
            clients[i] = new System.Threading.Thread(new Account().BuyTicket);
        }
        for (int i = 0; i <= clients.Length - 1; i++)
        {
            clients[i].Start(3);
            System.Console.WriteLine("---剩余机票{0}", Account.airTicket);
        }
    }
}

public class Account
{
    //使用static的含义是这个airTicket是个共享的数据
    public static int airTicket = 100;//初始飞机票的数量

    /// <summary>
    /// 买飞机票
    /// </summary>
    /// <param name="count">需要购买的数量</param>
    public void BuyTicket(object count)
    {
        if (airTicket <= 0)//如果机票没有了
        {
            return;
        }
```

```
        if (airTicket >= (int)count)//如果剩余机票够
        {
            //让用户随机的想0-20毫秒，思考是不是真的要买
            System.Threading.Thread.Sleep(new System.Random().Next(0,21));
            airTicket -= (int)count;
            System.Console.Write("{0}成功购买{1}", DateTime.Now, count);
        }
        return;
    }
}
```

＊线程类 Thread

使用锁来管理并发业务现象

```
class Program
{
    static void Main(string[] args)
    {
        //Thread是多线程
        System.Threading.Thread[] clients = new System.Threading.Thread[100];//有100个窗口
        for (int i = 0; i <= clients.Length - 1; i++)
        {
            clients[i] = new System.Threading.Thread(new Account().BuyTicket);//告诉每一个窗口，购票用的方法
        }
        for (int i = 0; i <= clients.Length - 1; i++)
        {
            clients[i].Start(3);//每个窗口买三张
        }
    }
}

public class Account
{
    //使用static的含义是这个airTicket是个共享的数据
    public static int airTicket = 100;//初始飞机票的数量

    /// <summary>
    /// 买飞机票
    /// </summary>
    /// <param name="count">需要购买的数量</param>
    public void BuyTicket(object count)
```

```
    {
        if (airTicket <= 0)//如果机票没有了
        {
            return;
        }
        if (airTicket >= (int)count)//如果剩余机票够
        {
            System.Console.WriteLine("现在看到有剩余[{0}]机票", airTicket);
            //让用户随机的想0-20毫秒，思考是不是真的要买
            System.Threading.Thread.Sleep(new System.Random().Next(0, 100));
            lock (new object())
            {

                if (airTicket >= (int)count)//如果剩余机票够
                {
                    System.Console.Write("剩余机票[{0}]--", Account.airTicket);
                    airTicket -= (int)count;
                    System.Console.WriteLine("[{0}]成功购买[{1}]", DateTime.Now, count,
airTicket);
                }
            }
        }
    }
}
```

2.6 数据，计算机世界的原点

计算机是用数据来描述一切，并对数据进行处理和加工，其得到的数据，反馈的也是数据。

2.6.1 数据类型分类

C#数据类型分为：值类型、引用类型和指针类型

数据类型分类表及关键字

<table>
<tr><td rowspan="7">值类型</td><td>枚举</td><td colspan="2">enum</td></tr>
<tr><td rowspan="6">内置类型</td><td rowspan="5">结构</td><td>整数：sbyte、byte、char、short、ushort、int、uint、long、ulong</td></tr>
<tr><td>浮点：float、double</td></tr>
<tr><td>高精度：decimal</td></tr>
<tr><td>布尔：bool</td></tr>
<tr><td>无类型：void</td></tr>
<tr><td rowspan="5">引用类型</td><td colspan="2">根对象：object</td></tr>
<tr><td colspan="2">字符串：string</td></tr>
<tr><td colspan="3">类：class</td></tr>
<tr><td colspan="3">委托：delegate</td></tr>
<tr><td colspan="3">接口：interface</td></tr>
<tr><td colspan="3">数组：Array</td></tr>
<tr><td>指针类</td><td colspan="3"></td></tr>
</table>

枚举

```
/// <summary>
/// 动物外观
/// </summary>
public enum AnimalSkinFamily
{
    /// <summary>
    /// 蚂蚁
    /// </summary>
    Ant,
    /// <summary>
    /// 甲虫
    /// </summary>
    Beetle,
    /// <summary>
    /// 螨虫
    /// </summary>
    Spider,
    /// <summary>
    /// 蝎子
    /// </summary>
    Inchworm,
    /// <summary>
    /// 蜘蛛
    /// </summary>
    Scorpion
}
```

*枚举定义

2.6.2 数据的大小和表示范围

在计算机中的数据，都有预先约定的大小。这些大小描述了各数据能表示的最大最小值范围。

数据类型	范围	大小
sbyte	-128 到 127	有符号 8 位整数
byte	0 到 255	无符号 8 位整数
char	U+0000 到 U+ffff	16 位 Unicode 字符
short	32,768 到 32,767	有符号 16 位整数
ushort	0 到 65,535	无符号 16 位整数
int	2,147,483,648 到 2,147,483,647	有符号 32 位整数
uint	0 到 4,294,967,295	无符号 32 位整数
long	-9,223,372,036,854,775,808 到 9,223,372,036,854,775,807	有符号 64 位整数
ulong	0 到 18,446,744,073,709,551,615	无符号 64 位整数
float	±1.5e−45 到 ±3.4e38	7 位
double	±5.0e−324 到 ±1.7e308	15 到 16 位
decimal	±1.0 × 10-28 到 ±7.9 × 1028	28 到 29 位
bool	true 和 false	

decimal 数据类型表示 128 位数据类型。同浮点型相比，decimal 类型具有更高的精度和更小的范围，这使它适合于财务和货币计算。我们在开发中常用的都是有符号类型。float 类型往往在图形图像中使用。

使用代码来测试数据类型的大小，我们在 2.5.6 检查中作了演示。

2.6.3 数据类型转换

从小学数学中，我们已经了解到不同的数据单位是不可以相互运行的。比如：一头牛＋一条狗是不能相互运算的，必须把数据单位转成一致才可以计算。

在编程中，我们同样需要将不同的数据类型转为相同的类型，才可以计算。

数据类型的转换分为：隐式（默认）转换和显式（强制）转换。转换的规则是：

- 小数据向大数据转为隐式（默认）转换，大数据向小数据转是显式（强制）转换
- 隐式（默认）转换总会成功，且是安全的。显式（强制）转换是非安全的，有可能是会

丢失精度。在 2.5.6 也描述了这个现象。

- 转换的双方必平等。都是值类型或都是引用类型

数据转换

```
static void Main(string[] args)
{
    //以下的转换是隐式（默认）转换总会成功，且是安全的
    byte b = 10;
    short sh = b;
    int i = sh;
    long l = i;
    Console.WriteLine(l);
    float f = l;
    double d = f;
    Console.WriteLine(d);

    //以下的转换是显式（强制）转换是非安全的，有可能是会丢失精度
    d = double.MaxValue;
    Console.WriteLine(d);
    f = (float)d;
    Console.WriteLine(f);
    l = (long)f;
    Console.WriteLine(l);
    i = (int)l;
    Console.WriteLine(i);
    sh = (short)i;
    Console.WriteLine(sh);
    b = (byte)sh;
    Console.WriteLine(b);
}
```

要描述将字符串格式转为相应的数值数据，有特殊的方法。

字符串转相应数据

```
static void Main(string[] args)
{
    //程序员必须确保，字符可以转换，否则出现异常
    byte b = byte.Parse("168");
    short sh = short.Parse("126");
    int i = int.Parse("1000");
    long l = long.Parse("1995");
    float f = float.Parse("12.24");
    double d = double.Parse("213.343");
```

```
    decimal de = decimal.Parse("123.34534");
    char c = char.Parse("b");
    DateTime date = DateTime.Parse("2008-8-18");
}
```

2.6.4 格式化数据

.NET Framework 提供了可自定义的、适于常规用途的格式设置机制，可将值转换为适合显示的字符串。使用 ToString 方法可以实现各种数据格式的呈现。

数据格式化

```
static void Main(string[] args)
{
    double dvalue = 1234.4567890;
    //货币格式
    Console.WriteLine(dvalue.ToString("C",
System.Globalization.CultureInfo.InvariantCulture));
    Console.WriteLine(dvalue.ToString("C4",
System.Globalization.CultureInfo.InvariantCulture));//3位小数

Console.WriteLine(dvalue.ToString("C3",System.Globalization.CultureInfo.CreateSpecificCultur
e("en-US")));
    Console.WriteLine(dvalue.ToString("C3",
System.Globalization.CultureInfo.CreateSpecificCulture("zh-CN")));
    int ivalue = 12312;
    //十进制数,只有整型才支持此格式
    Console.WriteLine(ivalue.ToString("D"));
    Console.WriteLine(ivalue.ToString("D8"));
    //科学记数法
    Console.WriteLine(ivalue.ToString("E",
System.Globalization.CultureInfo.InvariantCulture));
    Console.WriteLine(ivalue.ToString("E",
System.Globalization.CultureInfo.InvariantCulture));
    //定点,数字转换为“-ddd.ddd…”形式的字符串，其中每个“d”表示一个数字 (0-9)。如果该数字
为负，则该字符串以减号开头。
    Console.WriteLine(dvalue.ToString("F",
System.Globalization.CultureInfo.InvariantCulture));
    Console.WriteLine(dvalue.ToString("F3",
System.Globalization.CultureInfo.InvariantCulture));
```

```
    //百分比
    Console.WriteLine(dvalue.ToString("P",
System.Globalization.CultureInfo.InvariantCulture));
    //十六进制数,只有整型才支持此格式。
    Console.WriteLine(ivalue.ToString("x"));
    Console.WriteLine(ivalue.ToString("X8"));
}
```

＊区域信息 CultureInfo

2.6.5 数据运算

除了上面所述的数据类型转换或检查方式，C#还给予了一些数据运算的方式进行数据类型检查、获取类型大小等。

运算符	功能
as	将对象转换为可兼容类型，必须对引用类型或能 null 类型使用
is	检查对象的运行时类型
sizeof	获取值类型的大小
typeof	获取类型

数据运算

```
static void Main(string[] args)
{
    int i = 10;
    long l = 100;
    string s = "Hello";
    DateTime d = DateTime.Now;
    //as 运算符类似于强制转换操作。但是，如果无法进行转换，则 as 返回 null 而非引发异常
    Console.WriteLine(s as string);

    //is检查对象是否与给定类型兼容
    Console.WriteLine(i is string);
    Console.WriteLine(l is string);
    Console.WriteLine(s is string);

    //用于获取 值类型 的字节大小
    Console.WriteLine(sizeof(byte));
    Console.WriteLine(sizeof(short));
    Console.WriteLine(sizeof(int));
```

```
    Console.WriteLine(sizeof(short));
    Console.WriteLine(sizeof(long));
    Console.WriteLine(sizeof(float));
    Console.WriteLine(sizeof(double));

    //用于获取类型的 System.Type 对象
    Console.WriteLine(i.GetType());
    Console.WriteLine(l.GetType());
    Console.WriteLine(s.GetType());
    Console.WriteLine(d.GetType());
    Console.WriteLine(typeof(int));
}
```

＊类型类 Type

三、开始进入面向对象

现在说开始进入面向对象是非常的不严谨的描述，其实用 Hello,World 开始我们就一直在编写面向对象的程序和代码。

C# 是面向对象的编程语言，它使用类和结构来实现类型定义和使用。C# 提供了许多功能强大的类定义方式，提供不同的访问级别，从其他类继承功能，允许程序员指定实例化或销毁类型时的操作。C#3.0 还可以通过使用类型参数将类定义为泛型，通过类型参数，客户端代码可以类型安全的有效方式来自定义类。

C#对类和结构的定义

- 对象是给定数据类型的实例。
- 新数据类型是使用类和结构定义的。
- 类和结构组成了 C# 应用程序的生成块 C# 应用程序始终包含至少一个类。
- 结构可视为轻量类，是创建用于存储少量数据的数据类型的理想选择。
- C# 类支持继承；类可以从先前定义的类中派生。

编写面向对象的代码时，你要记得你有两个完全不同的身份：对象的创造者和对象的使用者。

3.1 无序的函数世界和有序的对象世界

为什么要使用面向对象开发呢？在早先的面向过程的开发中，我们关注是函数。大量的

函数独立分散在代码中，这个是一个离散的函数世界。所以很多工具书在整理的时候，都要罗列下函数，按函数服务的数据对象进行分类。这个现象说明，人类学习和使用函数的习惯。因此业界开始提出面向对象的模式，将数据的功能和数据的行为绑定起来，这就是有序的面向对象世界。

那么，到底什么是类，什么是对象呢？

3.2 类就是代码，对象是内存

不必麻烦的说什么类是对世界实体的抽象描述，对象是类的实例。我们用另一个角度来更自然的了解什么是类，什么是对象。

假设我们要设计一部手机，当这个手机的所有构造，功能仅仅是设计图上描述的时候，我们称这个可以查看的设计图为：**类**。

当我们设计完毕，开始生成的时候，生产线上流出的每一个手机，我们就称为：**对象**。

好，我们来看下，这个手机设计图和手机实物的关系。每一个手机实物在功能上严格的依赖于设计图，其大小，重量，外观和功能在这个生产线上和其他所有的手机保持一致。

但每个手机又有自己的特殊数据：ID 号，也许外观颜色还可以更有特色等。

手机和设计图还有一个区别就是，设计图没有占用实际资源，而生产出的手机肯定暂用了实际上的社会资源（消耗了材料、空间）。当然对这个话，不要太过于执着资源的概念。

在面向对象的代码中，用 class 来写类（设计图），用 new 来实例化对象（生产消耗资源）。

类和对象

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

        Phone phone1 = new Phone("356374000106267");//第一台手机
        Phone phone2 = new Phone("356374000106157");//第二台手机
        Phone phone3 = new Phone("356374000106482");//第三台手机
        Phone phone4 = new Phone("356374000106582");//第四台手机

    }
}

/// <summary>
/// 模拟手机
/// </summary>
class Phone
{
```

```
    public Phone(string imei)
    {
        IMEI = imei;
    }


    /// <summary>
    /// 拨号
    /// </summary>
    /// <param name="telephoneNumber">对方电话号码</param>
    /// <returns>如果接通则返回true，否则是false</returns>
    public bool Dial(string telephoneNumber)
    {
        return true;
    }

    public void hangUp()
    {

    }

    /// <summary>
    /// 一个集合，描述了电话号码本
    /// </summary>
    public System.Collections.ArrayList PhoneBook = new System.Collections.ArrayList();
    /// <summary>
    /// 短信收件箱
    /// </summary>
    public System.Collections.ArrayList InBox = new System.Collections.ArrayList();
    /// <summary>
    /// International Mobile Equipment Identity,手机串号
    /// </summary>
    public readonly string IMEI;
    /// <summary>
    /// 进网许可证
    /// </summary>
    public const string Licence = "02-5707-050850";
    /// <summary>
    /// 制作商
    /// </summary>
    public const string Manufacturer = "Lenovo";

}
```

＊ArrayList

看上面的代码，你可以有个基本了解。class 定义了一个 Phone 的基本能力（拨号，挂断，电话号码本已经收件箱）。每个被生产出的新手机（new 这个关键字如此的合乎常理）都拥有上面的功能，但对具体的数值（IEMI，电话号码簿，收件箱）各有表达了。

所以我们又得到一个推论：

3.2 行为是公共的，数据是私有的

上面的 Phone 描述了对类的一个定义：类是两大成员的集合，即数据成员和函数成员。数据成员我们又称为字段。函数成员我们称为方法。

手机的功能（函数）是每台手机（同 class 实例对象）都是完全相同的，他们严格的依照 class 的定义进行描述，但数据方面允许每个实例有自己的数据值。

个人所得税计算

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

        PersonalTaxCalculator p1 = new PersonalTaxCalculator();
        p1.Name = "张三";
        p1.Salary = 12340;
        PersonalTaxCalculator p2 = new PersonalTaxCalculator();
        p2.Name = "李四";
        p2.Salary = 3000;

        System.Console.WriteLine("{0}的收入是{1},个税为{2}", p1.Name, p1.Salary,
p1.TaxAllowance());
        System.Console.WriteLine("{0}的收入是{1},个税为{2}", p2.Name, p2.Salary,
p2.TaxAllowance());
    }
}

/// <summary>
/// 个税计算器
/// </summary>
class PersonalTaxCalculator
```

```
{
    public string Name = "";

    public double Salary = 0;

    public double TaxAllowance()
    {
        double tax = 0;

        if (Salary < 500)
        {
            tax = Salary * 0.05;
        }
        if (Salary > 500 && Salary < 2000)
        {
            tax = Salary * 0.1 - 25;
        }
        if (Salary > 2000 && Salary < 5000)
        {
            tax = Salary * 0.15 - 125;
        }
        if (Salary > 5000 && Salary < 20000)
        {
            tax = Salary * 0.20 - 375;
        }
        if (Salary > 20000 && Salary < 40000)
        {
            tax = Salary * 0.25 - 1375;
        }
        return tax;
    }
}
```

上面的例子我们可以清晰的了解到，类的方法（函数）对类的所有实例都是共性的，无论哪个实例，都使用同样的逻辑进行处理。类的字段（数据）保持每个类的私有性。

在这个例子中，只要我们给予类的实例其 Salary 字段的值，就可以通过 TaxAllowance 方法得到这个实例需要交纳的个税。个税的计算是非常复杂的（在各国都是这样），但现在我们写了一个类，让使用类的人，非常简单的完成了个税得的计算。在术语中，我们说：PersonalTaxCalculator 类封装了个税的计算。那么封装是什么意思呢？

3.3 封装就是隔离

面向对象三大特征的第一个就是“封装”。封装就是说对使用者而言，不需要去过多的了解被使用者的实现细节（当然，你天生具有好奇心，什么事情都要打破沙锅问到底那是另一个境界）。

我们使用移动电话，只要知道如何简单的使用拨号功能打电话就可以了，也许你还知道你的手机是支持 GSM 的。但到此为止，对一般的用户而言已经足够了。你没有必要了解 GSM 工作规范（工作频段、频道间隔、多址方案、在时域和频域中的间隙、无线接口管理、信道、保密、PING 等让非专业人士足以噩梦的技术细节）。

所以，封装是人类在大规模产业工作中，最自然最高效的互相信任模式。封装隔离了使用者和创造者之间对于细节的关注，封装迫使创造者提供最人性化的使用方案，封装提供了模块化开发的可能。

封装了复杂的所得税计算

```
class TaxCalculator
{
    /// <summary>
    /// 个人所得税计算
    /// </summary>
    /// <param name="salary">收入</param>
    /// <returns>应交纳所得税</returns>
    public double PersonalTax(double salary)
    {
        return 0;
    }

    /// <summary>
    /// 企业所得税计算
    /// </summary>
    /// <param name="taking">营业收入</param>
    /// <returns>应交纳所得税</returns>
    public double EnterpriseTax(double taking)
    {
        return 0;
    }

    /// <summary>
    /// 个体工商所得税计算
    /// </summary>
    /// <param name="taking">营业收入</param>
```

```
    /// <returns>应交纳所得税</returns>
    public double SelfMmployedPeopleTax(double salary)
    {
        return 0;
    }

    /// <summary>
    /// 劳务性收入所得税计算
    /// </summary>
    /// <param name="taking">收入</param>
    /// <returns>应交纳所得税</returns>
    public double LaborTax(double taking)
    {
        return 0;
    }
}
```

上面的代码中，我们可以看到 TaxCalculator 类提供了四种方法，允许使用者通过调用，就可以完成对收入所得税的计算。

封装可以让开发人员只把需要让用户知道的以方法来提供，用户不需要关心如果计算的细节，只要提供方法所需要的参数。

为了让类能确保哪些可以让用户看到，哪些不能让用户看到，我们需要

3.3.1 访问修饰控制

C#提供了访问修饰符，约束了五种访问控制

方法修饰符	可访问范围
public	访问不受限制
protected	访问仅限于包含类或从包含类派生的类型
internal	访问仅限于当前程序集
protected internal	访问仅限于从包含类派生的当前程序集或类型
private	访问仅限于包含类型

*程序集

访问控制

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

        SalesContract Contract = new SalesContract();
        Contract.Seller = "软件供应商";
        Contract.Buyer = "企业";
        Contract.PreferentialPrice = 19230;
        Contract.SalePrice = 20000;
        Console.WriteLine(Contract.CheckPrice());

    }
}

/// <summary>
/// 合同
/// </summary>
public class Contract
{
    /// <summary>
    /// 买方
    /// </summary>
    public string Buyer;//
    /// <summary>
    /// 卖方
    /// </summary>
    public string Seller;
    /// <summary>
    /// 成本价
    /// </summary>
    protected double CostPrice;
}

/// <summary>
/// 销售合同
/// </summary>
public class SalesContract : Contract
{
    /// <summary>
```

```
    /// 最低价
    /// </summary>
    private double MinimumPrice = 18030;
    /// <summary>
    /// 销售价格
    /// </summary>
    public double SalePrice;
    /// <summary>
    /// 价格检查
    /// </summary>
    /// <returns></returns>
    public bool CheckPrice()
    {
        return SalePrice < Math.Min(MinimumPrice, CostPrice);
    }
    /// <summary>
    /// 优惠价
    /// </summary>
    internal double PreferentialPrice;
}
```

＊Math 类

将内部的数据允许外部访问，C#提供了两种方式

3.3.2 数据成员

字段是类或结构中的对象或值。类和结构使用字段可以封装数据。字段通常用 private 封装。如果是常数或只读字段可以用 public 进行公开。

字段

```
public class Account
{
    /// <summary>
    /// 账户编号
    /// </summary>
    private readonly string ID="";
    /// <summary>
    /// 余额
```

```
    /// </summary>
    private double Balance;
    /// <summary>
    /// 上次的交易时间
    /// </summary>
    private DateTime LastTransaction;
    /// <summary>
    /// 上次交易金额
    /// </summary>
    private double LastTransactionBalance;
    /// <summary>
    /// 上次交易的设备好
    /// </summary>
    private int LastTransactionDeviceID;

}
```

3.3.4 函数成员

函数成员的表现形式比较多，有：方法、属性、事件、索引器、运算符、构造函数、析构函数。

属性和方法

```
public class Account
{
    /// <summary>
    /// 账户编号
    /// </summary>
    private readonly string ID = "";
    /// <summary>
    /// 余额
    /// </summary>
    private double balance;
    /// <summary>
    /// 上次的交易时间
    /// </summary>
    private DateTime LastTransaction;
    /// <summary>
    /// 上次交易金额
```

```
    /// </summary>
    private double LastTransactionBalance;
    /// <summary>
    /// 上次交易的设备好
    /// </summary>
    private int LastTransactionDeviceID;

    /// <summary>
    /// 获取余额
    /// </summary>
    public double Balance//属性
    {
        get
        {
            return balance;
        }
    }

    /// <summary>
    /// 存款，余额余额会增加
    /// </summary>
    /// <param name="money">需要交易的金额</param>
    public void Deposit(double money)//方法
    {
        balance += money;
    }
    /// <summary>
    /// 取款，如果成功则会减少余额
    /// </summary>
    /// <param name="money">需要交易的金额</param>
    /// <returns>如果成功，则返回ture否则为false</returns>
    public bool Withdraw(double money)//方法
    {
        if (balance > money)
        {
            balance -= money;
            return true;
        }
        else
        {
            if (balance < money)
            {
                Console.WriteLine("对不起，余额不够");
```

```
            }
            else
            {
                Console.WriteLine("账户内必须有余额");
            }
            return false;
        }
    }
}
```

属性是字段和方法的中间过程，其表现形式在方法和字段之间。

3.3.5 成员修饰

在 C#中成员有两种修饰：实例成员修饰和静态成员修饰。

成员	访问方式
实例成员	类实例为对象后才可以访问的成员
静态成员	无需创建类的实例就可以访问的成员

在我们第一个程序 Hello,World 中接触的 Main 方法就是静态方法。静态成员往往描述共性的行为和数据。

静态和实例成员

```
class Program
{
    static void Main(string[] args)
    {
        new Player();//id——1
        Console.WriteLine(Player.GetOnlineCount());
        new Player();//id——2
        new Player();//id——3
        new Player();//id——4
        Console.WriteLine(new Player().ID);
        Console.WriteLine(Player.GetOnlineCount());
    }
}
```

```
/// <summary>
/// 游戏玩家
/// </summary>
public class Player
{
    private static int count;
    private int id = ++count;
    /// <summary>
    /// 流水号
    /// </summary>
    public int ID
    {
        get
        {
            return id;
        }
    }
    /// <summary>
    /// 姓名
    /// </summary>
    public string Name;
    /// <summary>
    /// 性别
    /// </summary>
    public Sex Sex;
    /// <summary>
    /// 年龄
    /// </summary>
    public int Age;
    /// <summary>
    /// 在线人数
    /// </summary>
    /// <returns></returns>
    public static double GetOnlineCount()
    {
        return count;
    }
}

/// <summary>
/// 性别
/// </summary>
```

```
public enum Sex
{
    /// <summary>
    /// 男
    /// </summary>
    Male,
    /// <summary>
    /// 女
    /// </summary>
    Female,
}
```

在上面的代码中，你可以了解到，静态成员和实例成员各自的用途和数据处理。现在我们知道，需要访问控制一方面是为了封装的需要，但另一方面是要体现面向对象的第二大特性：

3.4 继承

类可以从其他类中继承。这是通过以下方式实现的：在声明类时，在类名称后放置一个冒号，然后在冒号后指定要从中继承的类（即基类）。

在 C#中，System. Object 是所有其他类型的最高类型，C#由单继承机制从该类型（直接或间接）派生出其他所有类型。

System.String	字符串类型
System.ValueType	值类型的基类
System.Enum	枚举类型的基类
System.Array	所有数组类型基类
System.Delegate	所有委托类型基类
System.Exception	所有异常类型基类

3.4.1 抽象类、密封类和静态类

在 C#中关于类的继承有三种情况：有的可以被继承，有的必须被继承，有的不能被继承

关键字	描述

abstract	创建仅用于继承用途的类和类成员，即定义派生的非抽象类的功能
sealed	密封类主要用于防止派生
static	描述一个只能包含静态成员的静态类，不能被实例化。

抽象类表示一个不完整的类：该类只可以作基类，不允许实例化，可以包含抽象成员，也可以不包含抽象成员，但包含了抽象成员的类一定是抽象类。

密封类不允许被继承，不允许包含抽象成员。静态类是仅包含静态成员的密封类。

没有被修饰为抽象或密封的类为一般类，可以继承，也可以被继承。

继承、抽象、密封、静态类

```
/// <summary>
/// 仓库
/// </summary>
public abstract class Warehouse//
{
    private double area;
    /// <summary>
    /// 面积
    /// </summary>
    public double Area
    {
        set
        {
            area = value;
        }
        get
        {
            return area;
        }
    }

    private string address;
    /// <summary>
    /// 地址
    /// </summary>
    public string Address
    {
        set
        {
            address = value;
        }
```

```
        get
        {
            return address;
        }
    }
}

/// <summary>
/// 自动化仓库
/// </summary>
public class AutomaticWarehouse : Warehouse
{
}

/// <summary>
/// 虚拟仓库
/// </summary>
public sealed class VirtualWarehouse : Warehouse
{
}
/// <summary>
/// 立体仓库
/// </summary>
public class StereoscopicWarehouse : BonedWarehouse
{
}
/// <summary>
/// 保税仓库
/// </summary>
public class BonedWarehouse : Warehouse
{
}
/// <summary>
/// 仓库检查
/// </summary>
public static class WarehouseCheck
{
    /// <summary>
    /// 是否还有空间
    /// </summary>
    /// <param name="w">需要检查的仓库</param>
    /// <returns>如果有空间，则返回true，否则返回false</returns>
    public static bool HasSpace(Warehouse w)
```

```
    {
        return true;
    }

    /// <summary>
    /// 仓库目前状态是启用还是关闭
    /// </summary>
    /// <param name="w">需要检查的仓库</param>
    /// <returns>如果被关闭则返回true，否则返回flase</returns>
    public static bool IsClose(Warehouse w)
    {

    }
}
```

静态类往往作为工具类使用。

3.4.2 构造函数

一种特殊的函数，该函数在类的实例化时由 new 运算符调用。构造函数的名字必须和类的名字一样，并且该函数没有任何返回值。构造函数也同样可以有实例构造函数和静态构造函数。静态构造函数只能有一个，且不能有参数，不能有访问修饰，在实例构造函数之前先构造。

构造函数

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

        PlayerOnlineManage.Join(new Player());
        PlayerOnlineManage.Join(new Player());
        Console.WriteLine(PlayerOnlineManage.GetOnlineCount());//2
        PlayerOnlineManage.Join(new Player());
        PlayerOnlineManage.Join(new Player());
        Console.WriteLine(PlayerOnlineManage.FindPlayer(3).ID);//3
        Console.WriteLine(PlayerOnlineManage.GetOnlineCount());//4
        PlayerOnlineManage.KillPlayer(2);
        Console.WriteLine(PlayerOnlineManage.GetOnlineCount());//3
```

```
        Console.WriteLine(PlayerOnlineManage.FindPlayer(3).ID);//3

    }
}


/// <summary>
/// 玩家在线管理
/// </summary>
public static class PlayerOnlineManage
{

    private static System.Collections.ArrayList OnlineList = null;
    static PlayerOnlineManage()
    {
        OnlineList = new System.Collections.ArrayList();
    }

    /// <summary>
    /// 加入新的玩家
    /// </summary>
    /// <param name="player"></param>
    /// <returns></returns>
    public static Player Join(Player player)
    {
        OnlineList.Add(player);
        return player;
    }

    /// <summary>
    /// 在线人数
    /// </summary>
    /// <returns></returns>
    public static double GetOnlineCount()
    {
        return OnlineList.Count;
    }

    /// <summary>
    /// 根据ID寻找一个玩家
    /// </summary>
    /// <param name="id">玩家的ID</param>
    /// <returns>如果找到返回该玩家的实例，否则返回null</returns>
```

```
    public static Player FindPlayer(int id)
    {
        for (int i = 0; i <= OnlineList.Count - 1; i++)
        {
            if (((Player)OnlineList[i]).ID == id)
            {
                return (Player)OnlineList[i];
            }
        }
        return null;
    }

    /// <summary>
    /// 将一个玩家出局
    /// </summary>
    /// <param name="id">该玩家的id</param>
    public static void KillPlayer(int id)
    {
        for (int i = 0; i <= OnlineList.Count - 1; i++)
        {
            if (((Player)OnlineList[i]).ID == id)
            {
                OnlineList.RemoveAt(i);
            }
        }
    }
}

/// <summary>
/// 游戏玩家
/// </summary>
public class Player
{
    private static int count;

    /// <summary>
    /// 流水号
    /// </summary>
    public readonly int ID;


    public Player()
```

```
    {
        ID = ++count;
    }

    /// <summary>
    /// 姓名
    /// </summary>
    public string Name;
    /// <summary>
    /// 性别
    /// </summary>
    public Sex Sex;
    /// <summary>
    /// 年龄
    /// </summary>
    public int Age;
}

/// <summary>
/// 性别
/// </summary>
public enum Sex
{
    /// <summary>
    /// 男
    /// </summary>
    Male,
    /// <summary>
    /// 女
    /// </summary>
    Female,
}
```

关注这个案例中的 Player 类的 ID 字段。在 3.3.5 成员修饰时，该 ID 是以属性表现出来。但在在这个案例中以只读字段表现。

对上述代码的优化

优化后的 PlayerOnlineManage

```
/// <summary>
/// 玩家在线管理
/// </summary>
public static class PlayerOnlineManage
```

```
{

    private static System.Collections.ArrayList OnlineList = null;
    static PlayerOnlineManage()
    {
        OnlineList = new System.Collections.ArrayList();
    }

    /// <summary>
    /// 加入新的玩家
    /// </summary>
    /// <param name="player"></param>
    /// <returns></returns>
    public static Player Join(Player player)
    {
        OnlineList.Add(player);
        return player;
    }

    /// <summary>
    /// 在线人数
    /// </summary>
    /// <returns></returns>
    public static double GetOnlineCount()
    {
        return OnlineList.Count;
    }

    /// <summary>
    /// 根据ID寻找一个玩家
    /// </summary>
    /// <param name="id">玩家的ID</param>
    /// <returns>如果找到返回该玩家的实例，否则返回null</returns>
    public static Player FindPlayer(int id)
    {
        int index = GetPlayerIndexByID(id);
        return index > -1 ? (Player)OnlineList[index] : null;
    }

    /// <summary>
    /// 将一个玩家出局
    /// </summary>
    /// <param name="id">该玩家的id</param>
```

```
    public static void KillPlayer(int id)
    {
        int index = GetPlayerIndexByID(id);
        if (index > -1)
        {
            OnlineList.RemoveAt(index);
        }
    }

    /// <summary>
    /// 根据玩家的id返回其在集合中的位置
    /// </summary>
    /// <param name="id">该玩家的id</param>
    /// <returns>如果找到则返回其位置索引，否则返回－1</returns>
    private static int GetPlayerIndexByID(int id)
    {
        for (int i = 0; i <= OnlineList.Count - 1; i++)
        {
            if (((Player)OnlineList[i]).ID == id)
            {
                return i;
            }
        }
        return -1;
    }
}
```

但却对用户的代码调用没有丝毫影响，这在面向对象中我们称为接口不变。

3.4.2 接口

接口是只对类或结构其公开（pubic）方法、属性、事件和索引器这四种函数成员的定义。类和结构可以用继承的模式从接口获得对成员的定义，并且类和结构可以实现对多个接口的继承。

接口其实就是纯虚的抽象类，但不需要声明接口为抽象或虚拟，接口不允许有构造函数，也允许有析构函数，其所有成员都是抽象的。因此接口只可以从接口继承，成员不允许有任何修饰，默认全部是 public。

将接口成员以类或结构的成员方式实现是需要注意

- 必须是 public 访问修饰
- 必须和接口成员的名称一致
- 必须和接口成员的参数列表一致
- 必须和接口成员的返回参数一致
- 必须实现接口的所有成员
- 可以修饰为虚拟或抽象

接口的定义和实现

```
/// <summary>
/// 职员
/// </summary>
public class employee
{
    /// <summary>
    /// 姓名
    /// </summary>
    public string name;
    /// <summary>
    /// 性别
    /// </summary>
    public bool sex;
    /// <summary>
    /// 部门
    /// </summary>
    public string department;
}

/// <summary>
/// 注册会计师
/// </summary>
public interface ICPA//
{
    /// <summary>
    /// 成本核算
    /// </summary>
    void CostKeeping();
    /// <summary>
    /// 签署
    /// </summary>
    void subscription();
}
```

```
/// <summary>
/// 销售
/// </summary>
public interface ISalesman
{
    /// <summary>
    /// 推销
    /// </summary>
    void SalesmanShip();
    /// <summary>
    /// 收款
    /// </summary>
    void Gathering();
    /// <summary>
    /// 喝酒
    /// </summary>
    void Drink();
    /// <summary>
    /// 跳舞
    /// </summary>
    void Dance();
    /// <summary>
    /// 唱歌
    /// </summary>
    void Sing();
}

public class CPA : employee, ICPA
{ }

public class Salesman : employee, ISalesman
{ }
```

接口有两种实现模式：标准实现和显式接口。

使用接口的完全限定名称在类或结构中实现

- 必须完全限定接口名称
- 不允许访问修饰
- 不允许为虚拟或抽象
- 不允许类的实例访问
- 只允许接口的实例访问

接口实现

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(new PersonalTax().TaxType);
        Console.WriteLine(new EnterpriseTax().TaxType);
        //Console.WriteLine(new SelfMmployedPeopleTax().TaxType);//显示实现不可以直接调用
        ITaxCalculator tax = new PersonalTax();
        Console.WriteLine(tax.TaxType);
        tax = new SelfMmployedPeopleTax();
        Console.WriteLine(tax.TaxType);
    }
}



/// <summary>
/// 关于所得税计算的接口定义
/// </summary>
public interface ITaxCalculator
{
    double Payable(double Income);
    string TaxType { get; }

}



public class PersonalTax : ITaxCalculator
{
    public double Payable(double Income)
    {
        return 0;
    }

    public string TaxType
    {
        get
        {
            return "个人所得税";
        }
    }
}
```

```
public class EnterpriseTax : ITaxCalculator
{
    public double Payable(double Income)
    {
        return 0;
    }

    public string TaxType
    {
        get
        {
            return "企业所得税";
        }
    }
}

public class SelfMmployedPeopleTax : ITaxCalculator
{
    double ITaxCalculator.Payable(double Income)
    {
        return 0;
    }

    string ITaxCalculator.TaxType
    {
        get
        {
            return "个体经营所得税";
        }
    }
}
```

在上面的代码，我们看到一个现象，TaxType 属性和 Payable 方法由接口定义，但不同的类对这属性和方法有不同的实现，这体现了面向对象的第三个特征

3.5 多态

3.5.1 在继承中使用重写

子类能对基类的虚拟方法和抽象方法进行重写，以实现自己的特殊实现模式。

关键字	概念
abstract	虚拟方法：基类有方法体，子类可以重写，也可以不重写
virtual	抽象方法，基类没有方法体，子类必须重写
override	重写基类方法

方法重写

```
/// <summary>
/// 动物
/// </summary>
public abstract class Animal
{
    public abstract void Attack(Animal animal);

}

/// <summary>
/// 蚂蚁
/// </summary>
public class Ant:Animal
{
    /// <summary>
    /// 战斗
    /// </summary>
    /// <param name="animal">需要攻击的对方</param>
    public override void Attack(Animal animal)
    {
        Console.WriteLine("我不打架");
    }

    /// <summary>
    /// 通讯
    /// </summary>
    /// <returns>返回需要传递的消息</returns>
    public virtual string Antenna()
    {
        return "一般蚂蚁我还不告诉。。。";
```

```
    }
}
/// <summary>
/// 兵蚁
/// </summary>
public class EnlistedAnt : Ant
{
    public override void Attack(Animal animal)
    {
        Console.WriteLine("打架，是我的职责");
    }
}
```

3.5.2 方法重载

在 C#中允许对不同参数列表的函数使用相同的函数名称类优化代码的使用。

重载

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(new Price().GetGain(1000, 850));
        Console.WriteLine(new Price().GetGain(1000, 850, 0.17));
        Console.WriteLine(new Price().GetGain(1000, 850, 0.2, 72));
    }
}

/// <summary>
/// 价格计算
/// </summary>
public class Price
{
    /// <summary>
    /// 利润计算，默认无其他成本，税为0.33
    /// </summary>
    /// <param name="retailPrice">实际价格</param>
    /// <param name="costPrice">成本</param>
    /// <returns>利润</returns>
```

```
    public double GetGain(double retailPrice, double costPrice)//利润
    {
        return this.GetGain(retailPrice, costPrice, 0.33);
    }

    /// <summary>
    /// 利润计算，默认无其他成本
    /// </summary>
    /// <param name="retailPrice">实际价格</param>
    /// <param name="costPrice">成本</param>
    /// <param name="tax">税</param>
    /// <returns></returns>
    public double GetGain(double retailPrice, double costPrice, double tax)
    {
        return this.GetGain(retailPrice, costPrice, tax, 0);
    }
    /// <summary>
    /// 利润计算
    /// </summary>
    /// <param name="retailPrice">实际价格</param>
    /// <param name="costPrice">成本</param>
    /// <param name="tax">税</param>
    /// <param name="otherCost">其他成本</param>
    /// <returns></returns>
    public double GetGain(double retailPrice, double costPrice, double tax, double otherCost)
    {
        //(零售价-成本价)*(1-税率)-其他成本
        return (retailPrice - costPrice) * (1 - tax) - otherCost;
    }
}
```

3.5.3 参数多重修饰

重载对返回值无效，对 params 参数修饰的方法无效，对 ref 或 out 修饰的参数方法有效。

参数修饰	控制能力
params	参数数目可变，且在方法声明中只允许一个
ref	参数引用传递，要求变量必须在传递之前进行初始化
out	参数引用传递，当控制权传递回调用方法时进行初始化

参数修饰

```
class Program
{
    static void Main(string[] args)
    {
        SimpleDS ds = new SimpleDS();
        int i = 12;
        ds.Max(i, new int[] { 2, 4, 7, 12, 5, 11, 21, 9 });
        Console.WriteLine(i);
        ds.Max(ref i, new int[] { 2, 4, 7, 12, 5, 11, 21, 9 });
        Console.WriteLine(i);
        ds.Max((double)i, 2, 4, 7, 12, 5, 11, 21, 9);
        Console.WriteLine(i);
    }
}


/// <summary>
/// 简单的数字服务
/// </summary>
public class SimpleDS
{
    public bool Max(int value, int[] values)
    {
        for (int i = 0; i <= values.Length - 1; i++)
        {
            if (values[i] > value)
            {
                return false;
            }
        }
        return true;
    }

    public bool Max(ref int value, int[] values)
    {
        bool result = false;
        for (int i = 0; i <= values.Length - 1; i++)
        {
            if (values[i] > value)
            {
                value = values[i];
```

```
                result = false;
            }
        }
        return result;
    }

    public bool Max(double value, params double[] values)
    {
        bool result = false;
        for (int i = 0; i <= values.Length - 1; i++)
        {
            if (values[i] > value)
            {
                value = values[i];
                result = false;
            }
        }
        return result;
    }
}
```

四、调试和测试代码

4.1 调试

4.2 单元测试

对于开发人员来讲，单元测试是非常重要的技术手段。单元测试是一种编程测试，单元

测试的代码通过直接调用某个类的方法（传递相应参数）来执行其结果。使用单元测试不仅能够测试公共方法，还可以测试私有方法。

单元测试是在软件开发过程中要进行的最低级别的测试活动，在单元测试活动中，软件的独立单元将在与程序的其他部分相隔离的情况下进行测试。单元测试不仅仅是作为无错编码的一种辅助手段在一次性的开发过程中使用，单元测试必须是可重复的，无论是在软件修改，或是移植到新的运行环境的过程中。因此，所有的测试都必须在整个软件系统的生命周期中进行维护。

Visual Studio 2008 的单元测试将测试代码和被测试代码独立分为两个独立项目进行实施。

4.2.1 单元测试项目环境

由于单元测试也是编程，并且需要和别测试项目隔离，所以需要给单元测试建立独立的项目。作为单元测试项目需要至少引入以下的组件

组件名称	用途
Microsoft.VisualStudio.QualityTools.UnitTestFramework	为单元测试提供类
System	Framework 的支持

然后需要引入需要测试的程序集。包含一个[TestClass]的类，该类有一个[TestMethod]方法。

单元测试类

```
/// <summary>
/// 一个测试类型
/// </summary>
[TestClass]
public class UnitTest1
{
    public UnitTest1()
    {

    }

    private TestContext testContextInstance;

    /// <summary>
    ///获取或设置测试上下文，该上下文提供
    ///有关当前测试运行及其功能的信息。
    ///</summary>
```

```
    public TestContext TestContext
    {
        get
        {
            return testContextInstance;
        }
        set
        {
            testContextInstance = value;
        }
    }

    [TestMethod]//测试方法必须由[TestMethod]修饰
    public void TestMethod1()
    {

    }
}
```

4.2.2 测试公有方法

由于计算机代码一般都执行精确的操作（随机除外），又应为我们一般要求计算机对处理的结果有一个返回，所以我们可以说，代码是对一个数据进行逻辑加工并返回的过程。因此单元测试就利用这个原则，对代码进行测试。通过 Assert.AreEqual 方法比较比较计算机处理的结果和我们预期的结果是否一致。

对 3.2 PersonalTaxCalculator 代码的测试

```
using ConsoleApplication1;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace TestProject1
{
    [TestClass()]
    public class PersonalTaxCalculatorTest
    {
        [TestMethod()]
        public void TaxAllowanceTest1000()
        {
            PersonalTaxCalculator tax = new PersonalTaxCalculator();
            tax.Salary = 1000;
```

```
            //以下测试成功
            Assert.AreEqual(tax.TaxAllowance(), (1000 - 500) * 0.1 + 500 * 0.05);
        }

        [TestMethod()]
        public void TaxAllowanceTest2000()
        {
            PersonalTaxCalculator tax = new PersonalTaxCalculator();
            tax.Salary = 2000;
            //以下测试失败
            Assert.AreEqual(tax.TaxAllowance(), (2000 - 1000) * 0.15 + (2000 - 1000 - 500) * 0.1
+ 500 * 0.05);
        }
    }
}
```

4.2.3 测试私有方法

使用单元测试可以对公共方法和私有方法进行测试。与公共方法类似，在从要测试的代码生成测试时，将自动创建私有方法的单元测试。

尽管可以手动编写单元测试代码以测试任何方法，但这需要对反射的复杂性有较好的理解，因此为私有方法编写测试代码比为公共方法编写测试代码的难度更大。因此，生成私有方法的测试比手动编写其测试代码更为方便。

生成私有方法的单元测试时，Visual Studio 会创建一个专用访问器。专用访问器是一个程序集，通过该程序集，测试可以从该方法的类的外部访问私有方法。

对 3.4.2 改进版 PlayerOnlineManage 的测试

```
[TestClass()]
public class PlayerOnlineManageTest
{
    System.Collections.Generic.List<Player> PlsyerList = new
System.Collections.Generic.List<Player>();

    public PlayerOnlineManageTest()
    {
        for (int i = 0; i <= 10;i++ )
        {
            PlsyerList.Add(new Player());
        }
```

```
    }

    [TestMethod()]
    public void KillPlayerTest()
    {
        PlayerOnlineManage.KillPlayer(3);
        Assert.AreEqual(PlayerOnlineManage.FindPlayer(3), null);

    }

    [TestMethod()]
    public void JoinTest()
    {
        for (int i = 0; i <= PlsyerList.Count - 1;i++ )
        {
            PlayerOnlineManage.Join(PlsyerList[i]);
        }
        Assert.AreEqual(PlayerOnlineManage.FindPlayer(5), PlsyerList[5]);
    }

    /// <summary>
    ///对private方法测试
    ///</summary>
    [TestMethod()]
    [DeploymentItem("ConsoleApplication1.exe")]
    public void GetPlayerIndexByIDTest()
    {
        //如果是private方法，则要使用这个代理
        Assert.AreEqual(PlayerOnlineManage_Accessor.GetPlayerIndexByID(5),null);
    }

    [TestMethod()]
    public void GetOnlineCountTest()
    {
        for (int i = 0; i <= PlsyerList.Count - 1; i++)
        {
            PlayerOnlineManage.Join(PlsyerList[i]);
        }
        Assert.AreEqual(PlayerOnlineManage.GetOnlineCount(), PlsyerList.Count);
    }

    [TestMethod()]
    public void FindPlayerTest()
```

```
    {
        Assert.AreEqual(PlayerOnlineManage.FindPlayer(3), null);
    }
}
```

（以下内容略去）

4.2.4 继承测试单元类

五、面向对象的深入体验

5.1 一张订单的进化

5.2 二维虚拟表格

六、C#语言的其他高级特征

6.1 泛型

6.2 委托

6.3 事件

6.4 匿名