C#编程语言与面向对象基础精简教程

****

作者声明：

（1）本教程为《ASP.NET 程序设计教程》（高等教育出版社 预计 2009 年 2 月出版）的补充资料，其目的是为学习“ASP.NET 开发技术”的学生打好必要的基础，扫清学习的障碍。

（2）本教程主要内容抽取自金旭亮所著之《.NET 2.0 面向对象编程揭秘》（电子工业出版社，2007 年 6 月出版），此书对 C# 2.0 编程语言与对象模型，.NET CLR 内部运行机理，.NET Framework 基类库中的核心技术，以及 Visual Studio 2005 提供的新特性进行了深入介绍，需要深入了解和把握面向对象理论与技术的读者可以通读此书。

（3）作为本教程的编写者，授权允许读者出于学习目的以各种方式使用、复制和传播本教程的内容及相关示例，但不得将其用于商业目的，作者本人也不对使用这些代码所可能带来的各种问题承担责任。

（4）欢迎读者就计算机技术与教学问题与作者进行交流，作者联系方式为：

电子邮件： JinXuLiang@bit.edu.cn

个人博客：（1）http://blog.csdn.net/bitfan

（2）http://blog.sina.com.cn/bitfan

金旭亮

2008 年 9 月

****

目 录

本教程介绍.NET 平台上主流的编程语言 C#，为读者学习 ASP.NET 技术打下基础。C#是一门新设计的语言，它#吸收了其前辈如 C++、Java 和 Delphi 等长处，拥有相当丰富的语言特性。本章仅介绍使用 C#进行面向对象编程的基础知识，重点在于讲清楚面向对象编程中的一些重要而基础的概念。

1 面向对象技术概论

面向对象理论很早就被提出了，但它真正地渗透到软件开发的各个领域，并且在软件开发实践中大规模应用，却要等到上世纪九十年代。到目前为止，面向对象技术已是软件开发的主流，全面取代了结构化编程技术曾经具有的地位。

面向对象技术与结构化编程技术有着不同的风格，但同时也有着密切的联系。从具体编程角度来看，面向对象技术与结构化编程技术很难截然分开，两者的根本差别在于思维方式。

要了解面向对象技术，得从结构化编程技术入手。

1.1 结构化编程引例

结构化编程在上世纪七十年代兴起，由于它具有很多的优点，出现之后没几年就占据了软件开发的主流，大家熟悉的 C 语言就是一种典型的结构化编程语言。

结构化编程的基本方法是“**功能分解法**”，具体来说，就是将要解决的实际问题进行分解，把一个大问题分成若干个子问题，每个子问题又可以被分解为更小的问题，直到得到的子问题可以用一个函数来实现为止。

我们先从一个简单的编程任务开始，逐步体会结构化编程与面向对象编程的不同风格。请看以下的这道编程题目：

> 请编程计算出1999年5月10日到2006年3月8日期间一共有多少天？

这是一个简单的算术问题，尽管如此，为了作个铺垫，我们还是把计算方法再叙述一下，以帮助读者理清思路。

有以下基本常识：

- 一年有 365 天，但闰年有 366 天；
- 一年有 12 个月，大月 31 天，小月 30 天；
- 2 月最特殊，普通年有 28 天，闰年有 29 天。

根据以上基本常识，计算步骤如下：

（1）计算从 1999 到 2006 期间共有多少个整年：

2000、2001、2002、2003、2004、2005，共有 6 个整年，其中 2000 和 2004 年是闰年，因此，共有 6×365＋2＝2192 天。

（2）计算从 1999 年 5 月 10 日到年底（即 12 月 31 日）共有多少天：

5 月 10 日到 12 月 31 日中共有 4 个整的大月（7 月、8 月、10 月、12 月），3 个整的小月（6 月、9 月、11 月），共记 4×31＋3×30＝214 天。

5 月 10 日到本月底（31 日）还有 31-10＝21 天。

所以，1999 年 5 月 10 日到年底共有 214＋21＝235 天。

（3）计算从 2006 年元旦到 2006 年 3 月 8 日期间一共有多少天：

1 月有 31 天，2006 年不是闰年，2 月有 28 天，所以，总共有 31＋28＋8＝67 天。

综上所述，1999 年 5 月 10 日到 2006 年 3 月 8 日期间一共有 2192＋235＋67＝2494 天。

事实上，上述计算过程其实就是一个计算机**算法**（algorithm），由于步骤很明确，可以很容易地将这一过程转为程序。

编程之前，先将实际问题抽象为以下模型（图 1）：

图 1 程序的最高层抽象

如图 1 所示，我们要完成的工作就是开发出这样一个程序：程序接收两个日期值，经过计算之后，输出这两个日期之间的天数。

（1）结构化分析过程

为了清晰地表达出程序需要处理的信息，先定义一个结构体类型：

```
//封装日期信息
public struct MyDate
{
    public int Year;    //年
    public int Month;   //月
    public int Day;     //日
}
```

结构体类型 MyDate 其实是定义了一种**数据结构（data structure）**。我们正是在这个数据结构之上构建出整个程序的。

对图 1 所示的模型进行结构化分析的第一步，是将“程序”方框完成的功能转化为由一个函数 CalculateDaysOfTwoDate()实现：

```
//计算两个日期中的整天数
static int CalculateDaysOfTwoDate(MyDate beginDate, MyDate endDate)
{
    //……
}
```

余下的开发工作体现为给 CalculateDaysOfTwoDate()编写代码实现日期计算的功能。

在结构化编程中，有这样一个重要的公式：

程序＝数据结构＋算法

数据结构代表了要处理的信息，而算法则表明要对这些信息进行哪些处理工作。

只要确定了数据结构和算法，一个程序就成形了。因此，

将程序中要处理的数据抽象为某种数据结构是结构化编程的基础。

在本例中，算法非常简单，可以直接将人计算过程中的每一个步骤转为一个函数，由此

得到以下两个函数：

```
//计算两年之间的整年天数，不足一年的去掉
static int CalculateDaysOfTwoYear(int beginYear, int endYear)
{
    //……
}
//根据两个日期，计算出这两个日期之间的天数，不理会中间的整年
static int CalculateDaysOfTwoMonth(MyDate beginDate, MyDate endDate)
{
    //……
}
```

第一个函数根据两个年份之间的整年数计算出天数，第二个函数根据月和日计算出两个日期之间的天数（不理会中间的整年）。

在深入地考虑这两个函数的具体实现算法时，会发现它们都需要判断一年是否是闰年，于是，设计另一个函数 IsLeapYear()完成此功能：

```
//根据年数判断其是否为闰年
static bool IsLeapYear(int year)
{
    //……
}
```

这样，函数 CalculateDaysOfTwoYear()和 CalculateDaysOfTwoMonth()在需要的时候即可调用 IsLeapYear()函数来判断是否某年为闰年。

至此设计工作完成，得到了以下结果（图 2）：

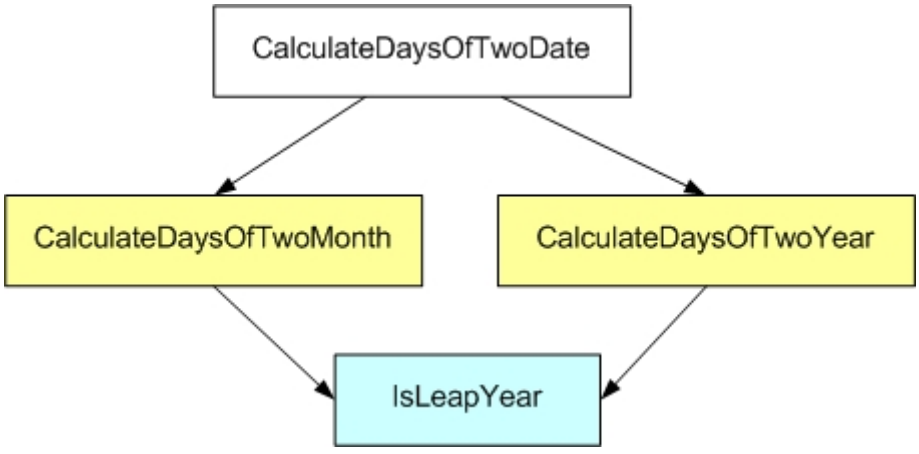

图 2 结构化程序设计结果

图 2 展示了结构化分析得到的设计方案，图中的箭头表示函数调用关系。

在整个结构化分析过程中，我们采用的是先设计出最顶层的 CalculateDaysOfTwoDate()函数的接口，再设计第二层的两个函数 CalculateDaysOfTwoYear()和

CalculateDaysOfTwoMonth()，最后抽象出第三层的函数 IsLeapYear()。

有了设计图，即可动手写代码。现在有四个函数需要开发，如何确定开发顺序？

很明显，必须先开发 IsLeapYear()函数，因为此函数被其他函数调用，但它不调用其他的函数。

接着可以开发 CalculateDaysOfTwoMonth()和 CalculateDaysOfTwoYear()两个函数，因为 CalculateDaysOfTwoYear()函数比较简单，所以先开发它。

最后开发 CalculateDaysOfTwoDate()函数。

上述开发次序的确定可以用两句话来表达：

（1）**盖楼先打地基**：先开发最底层的函数，因为不完成开发这些函数，调用它们的上层函数就无法运行。

（2）**柿子捡软的捏**：在同一层次的函数中，先开发简单的，再开发复杂的。

函数开发完成之后，以下代码调用最顶层函数 CalculateDaysOfTwoDate()完成计算两日期之间天数的工作：

```
MyDate d1, d2;     //起始日期和结束日期
//1999年5月10日
d1.Year = 1999;
d1.Month = 5;
d1.Day = 10;
//2006年3月8日
d2.Year = 2006;
d2.Month = 3;
d2.Day = 8;
//计算结果
int  days  = CalculateDaysOfTwoDate(d1, d2);
```

现在可以对结构化编程方法作个小结。

- 结构化软件系统的基本编程单位是函数。
- 整个系统按功能划分为若干个模块，每个模块都由逻辑上或功能上相关的若干个函数构成，各模块在功能上相对独立。
- 公用的函数存放在公用模块中，各模块间可以相互调用，拥有调用关系的模块形成一个树型结构，这种调用关系应尽可能做到是单向的。

结构化软件系统的架构如图 3 所示：

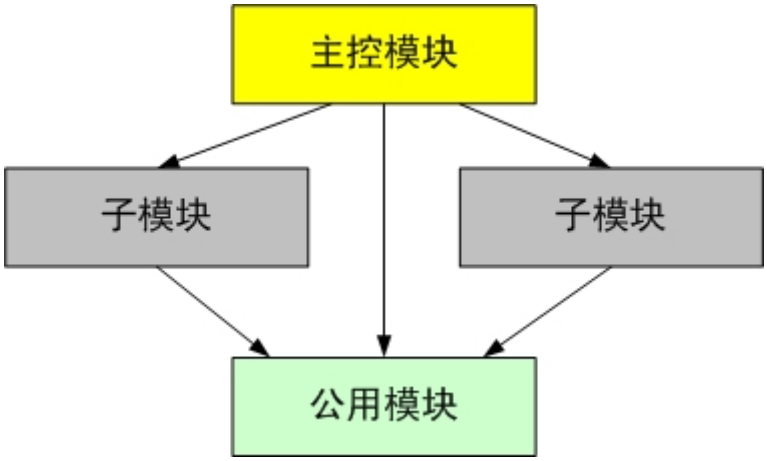

图 3 结构化软件的架构

结构化编程的开发过程可以分为以下三个阶段：

（1）**分析阶段**：在编程之前，需要仔细分析要解决的问题，确定好数据结构与算法。

（2）**设计阶段**：结构化编程的基本单元是函数，每个函数都完成整个程序的一个功能，整个设计过程就是函数接口的设计过程，这是一个“**自顶向下，逐步求精**”的过程，将一个大函数不断分解为多个小函数，直至可以很容易用某种程序设计语言实现时为止。

（3）**编码阶段**：在开发时，根据在设计阶段得到的函数调用图，先开发最底层的函数，再开发上层函数。这是一个“**自底向上，逐层盖楼**”的方法。

结构化编程中“自顶向下，逐步求精”的“功能分解法”，是一种重要的软件开发方法，其本质是一种“分而治之”的思维方式，在面向对象的程序中也有广泛的应用。掌握这种分析方法，对软件工程师而言是非常重要的。

（2）面向对象分析过程

有了结构化分析的基础，可以很容易的将原先结构化的程序转为面向对象的程序。

创建一个 CalculateDate 类，作为上面结构化分析得到的四个函数的“新家”，如图 4 所示：

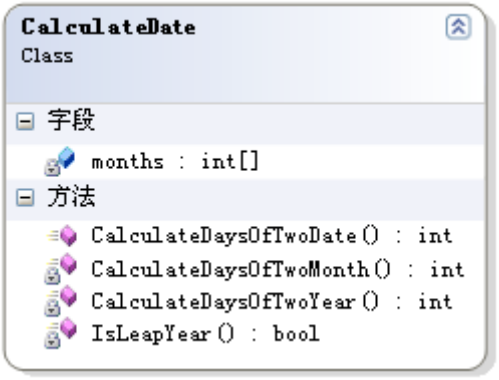

图 4 将函数移到类中

由于外界只需要调用 CalculateDaysOfTwoDate()一个函数，所以将此函数设置为公有（public），而其他三个函数则成为类的私有（private）成员，外界不可访问.

以下为调用此类完成计算两个日期间天数的代码示例：

```
MyDate  d1, d2;  //起始日期和结束日期
//1999年5月10日
d1.Year = 1999;
d1.Month = 5;
d1.Day = 10;
//2006年3月8日
d2.Year = 2006;
d2.Month = 3;
d2.Day = 8;
string  str  = "{0}年{1}月{2}日到{3}年{4}月{5}日共有天数：";
str = String.Format(str, d1.Year, d1.Month, d1.Day,
                         d2.Year, d2.Month, d2.Day);
CalculateDate obj=new CalculateDate(); //创建类CalculateDate对象obj
//调用对象obj的CalculateDaysOfTwoDate方法计算
int days = obj.CalculateDaysOfTwoDate(d1, d2);
Console.WriteLine(str + days);
```

对比前面结构化的程序，不难发现面向对象的程序具有以下几个特点：

（1）所有的函数都放入到一个类中，成为某个类的成员，**类是编程的基本单元**。

（2）外界不能直接调用类的成员函数，必须先创建一个对象，再通过对象来调用这些函数。

（3）只有声明为 public 的函数可以被外界调用（本例中为 CalculateDaysOfTwoDate()函数），其余声明为 private 的函数是私有的，外界无法访问。

从这个实例可以看出，面向对象程序与结构化程序有着很不一样的风格，但看不出来面向对象有何优越之处。

的确，对于这个小实例而言，面向对象程序与结构化程序相比没有明显的优越性，而且显得更麻烦，但如果是大规模的软件系统，则面向对象程序就有着结构化程序不可比拟的优势，简单地说：

对于大规模的系统，采用面向对象技术开发可以达到较高的开发效率与较低的维护成本，系统的可扩展性也更好。

拿本节的小例子而言，其实.NET Framework 本身就提供了两个类 DateTime 和 TimeSpan 可完成同样的功能：

```
DateTime dd1, dd2;
dd1 = new DateTime(2006, 3, 8);
dd2 = new DateTime(1999, 5, 10);
//两个日期对象相减，得到一个TimeSpan对象，Days是这一TimeSpan对象的属性
int ddays = (dd1 - dd2).Days;
Console.WriteLine(ddays);//结果：
```

对比一下，显然使用.NET Framework 提供的现成类比我们手工编写代码完成同样的工作开发效率要高得多。.NET Framework 中所提供的现成代码都是以面向对象的形式封装的。实践证明，当需要大规模地复用代码以提高软件生产率时，面向对象比结构化技术更有效。

1.2 面向对象的核心概念

从理论上说，面向对象技术拥有四大基本特性。

（1）封装

这是一种隐藏信息的特性。拿本节引例来说，类 CalculateDate 将数据结构与算法隐藏在类的内部，外界使用者无需知道具体技术实现细节即可使用此类。封装这一特性不仅大大提高了代码的易用性，而且还使得类的开发者可以方便地更换新的算法，这种变化不会影响使用类的外部代码。可以用以下公式展示类的封装特性：

封装的类=数据+对此数据所进行的操作（即算法）

通俗地说，封装就是：包起外界不必需要知道的东西，只向外界展露可供展示的东西。

在面向对象理论中，封装这个概念拥有更为宽广的含义。小到一个简单的数据结构，大到一个完整的软件子系统，静态的如某软件系统要收集数据信息项，动态的如某个工作处理流程，都可以封装到一个类中。

具备这种“封装”的意识，是掌握面向对象分析与设计技巧的关键。

（2）抽象

讲到抽象，不得不涉及到现代科学技术的基础——数学。

数学是一门抽象的科学，面对着纷繁复杂的世间万物，数学不理会各种事物的独特特性，而只抽取它们在数量上的特性，深刻揭示了“世间万物”在数量上表现出的共同规律，抽象正是数学的本质特征。

数学的一个分支——离散数学是计算机科学的根基之一，因此，计算机科学从诞生之日起，就与数学有着密不可分的联系，抽象思维也是计算机科学的主要思维方法之一。

在使用面向对象的方法设计一个软件系统时，首先就要区分出现实世界中的事物所属的类型，分析它们拥有哪些性质与功能，再将它们抽象为在计算机虚拟世界中才有意义的实体——类，在程序运行时，由类创建出对象，用对象之间的相互合作关系来模拟真实世界中事物的相互关联。

在从真实世界到计算机虚拟世界的转换过程中，抽象起了关键的作用。

（3）继承

真实世界中，事物之间有着一种派生关系，比如“食品”这一大类中包括“水果”这一子类，而“苹果”又包含在“水果”这一子类中。用图形表示如图 5：

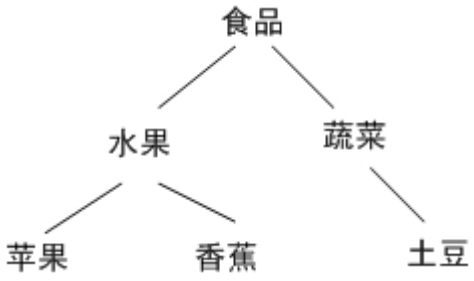

图 5 现实世界中的事物派生关系

在计算机世界中，以面向对象的观点不仅将上述事物抽象为类，而且将事物之间的派生

关系也一并模拟出来，这种关系称为“继承”（图 6）：

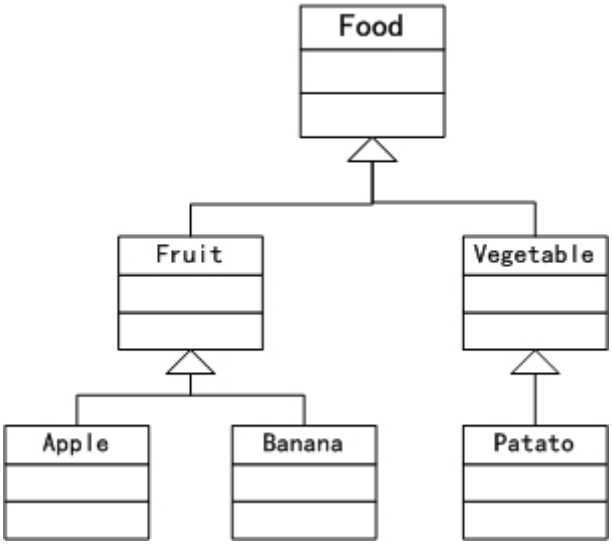

图 6 用“继承”模拟现实中的“派生”关系

在面向对象软件系统中，继承除了模拟现实世界中事物关系这一作用，还是一个“**重用已有代码而又不破坏已有代码**”的方法。举例来说：

现在要开发一个 B 项目，架构设计师发现以前完成的 A 项目中有部分类完全可以在 B 项目中重用，但需要增强这些类的功能以便适用于 B 项目。如果从 A 项目中直接抽取这些类的源代码并加以修改，虽然可以满足 B 项目的需要，但现在却需要维护两套功能类似的类代码，加大了管理的成本。在这种情况下，选择从 A 项目的类中用继承的方法派生出新类用在 B 项目中是一个可选的方案，既满足了新项目的需要，又避免了大量的重复代码与双倍的代码维护成本。

（4）多态

在现实生活中，我们常说：苹果**是一种**水果（Apple **is a** kind of fruit）。苹果与水果这两个概念之间其实是一种特性与共性的关系，简称为 **IS_A** 关系，其特点为：水果拥有的特性，苹果一定有。

在前面的小节中我们用继承模拟了 IS_A 关系，“水果（Fruit）”是基类，“苹果（Apple）”是子类，IS_A 关系可以用另一种方式来表达——**基类拥有的特性，子类一定有**。因此，可以把子类看成是一种“特殊”的基类。

举个例子：“给张三一个苹果”和“给张三一串香蕉”，虽然是两件不同的事，却可以统一说成“给张三一个水果”。

用“给张三一个水果”来指代“给张三一个苹果”和“给张三一串香蕉”，虽然在语义上比较“模糊”，但其适用性更广了，除了“苹果”和“香蕉”，以后还可以是“梨子”、“西瓜”、“葡萄”……，都可以用“给张三一个水果”来代表。

这种用一个比较抽象的事物来取代具体的事物的表达方法，在面向对象软件中用“多态”这一特性来模拟。

在编程时应用多态的方法，可以用一句话来表示：

用抽象的类编程[1]。

[1] 这里，我们将“接口”也看成是一种特殊的抽象类，抽象类中可以有非抽象方法，而接口中所有方法都是抽象方法。

即在代码中本应使用某一具体子类的地方使用较为抽象的基类对象，这种方法所带来的好处是多态的代码具有“变色龙”的特性，即在不同的条件下，同样代码可以完成不同的功能。

适当地在开发中应用多态特性，可以开发出可扩充性很强的系统。

（5）小结

面向对象的四大特性不是彼此独立的，“抽象”和“封装”更多地体现为一种思维方法，主要体现在面向对象系统的总体分析和设计中，“继承”和“多态”则多应用于具体子系统和软件模块的设计与编码过程中，而且“继承”是“多态”的基础。

1.3 面向对象技术与 Web 开发的关系

在早期静态网页的时代，网页主要是由 HTML 代码表达的，Web 服务器完成的工作只是应浏览器的请求传送网页，职责单一，除非需要开发 Web 服务器本身，开发以静态网页为主体的网站与面向对象技术并没有什么直接关系。

在早期的动态网页技术（如 ASP）出现之后，Web 服务器职责开始变得复杂起来，它不仅仅要完成原先的传送网页的工作，还需要完成在传送网页之前动态生成网页的工作。这时，动态网页多采用脚本语言比如 VBScript/JScript 等配合 HTML 来开发。

脚本语言与标准的面向对象语言相比，一般在语法上有所简化，在功能上也不如标准的面向对象语言强大，并且不支持面向对象的所有特性，同时往往也缺乏高效率的开发工具支持。

Java 平台的动态网页技术 JSP 应用 Serverlet 技术突破了网页开发不能使用面向对象语言与工具的限制，网页是由在 Web 服务器运行的 Serverlet 动态生成的，而 Serverlet 本身则使用全面向对象的 Java 语言开发。

在 JSP 出现并得到广泛应用不久之后，微软公司于 2002 年推出了 ASP 技术的替代技术——ASP.NET，在面向对象的道路上走得更远。

ASP.NET 将网页本身看成是一个类，当浏览器向 Web 服务器请求这个网页时，Web 服务器动态创建这个类的一个对象，然后调用它的特定方法生成 HTML 代码，再将生成的这些 HTML 代码发回给浏览器。

在开发 ASP.NET 网页时，开发者可以使用 C#/VB.NET 等面向对象的语言，并以与桌面程序一致的“所见即所得”与“事件驱动”开发方式编程。对比老的 ASP 技术，ASP.NET 技术的这些新特点，充分说明了微软 Web 开发技术已全面步入面向对象的时代。

由于 ASP.NET 技术是全面向对象的，因此，要想掌握这一技术，**必须具备有扎实的面向对象理论基础**。

2 类与对象

从本节开始，介绍 C#面向对象编程的基本内容。

与使用 C 语言等结构化编程语言不一样，使用 C#编程，所有的程序代码几乎都放在类中，不存在独立于类之外的函数。因此，**类是面向对象编程的基本单元。**

在绝大多数面向对象语言中，一个类都可以包含两种成员：字段（Field）与方法（Method）。字段与方法这两个概念是面向对象理论的术语，是通用于各种面向对象语言的。而在各种的具体面向对象语言（比如 C#）中，可以简单地这样理解：

字段即变量，方法即函数。

类的字段一般代表类中被处理的数据，类的方法大多代表对这些数据的处理过程或用于实现某种特定的功能，方法中的代码往往需要访问字段保存的数据。

在 C#中，定义若干个变量，写若干个函数，将这些代码按以下格式汇集起来，再起个有意义的名字，就完成了一个类的定义：

```
[public|private] class 类名
{
     [public|private] 数据类型  变量名;
     [public|private] 数据类型  函数名(参数列表)
    {
    }
}
```

在上述类的定义中，方括号代表这部分可选，而竖线则代表多选一。声明为 public 的变量和函数可以被外界直接访问，与此对应，private 的变量与函数，则为类的私有成员，只能由类自己使用。

下面简要介绍组成类的基本成员。

2.1 类的字段

字段（Field）代表了类中的数据，在类的所有方法之外定义一个变量即定义了一个字段。在变量之前可以加上 public、private 和 protected 表示字段的访问权限。以下代码展示了在类 Student 中定义的两个公有字段 Age 和 SN，外界可以通过类 Student 创建的对象来读取或设置这两个字段的值。

可以在定义字段的同时给予一个初始值，如下所示：

```
class Student
{
   public int age=18;                //年龄
   public string SN="1220040110";    //学号
}
```

2.2 类的方法

（1）函数的概念

在程序开发过程中，经常发现多处需要实现或调用某一个公用功能（比如选择一个文件），这些功能的实现都需要书写若干行代码。如果在调用此功能的地方重复书写这些功能代码，将会使整个程序中代码大量重复，会增大开发工作量，增加代码维护的难度。

为了解决代码重复的问题，绝大多数程序设计语言都将完成某一公用功能的多个语句组合在一起，起一个名字用于代表这些语句的全体，这样的代码块被称为“**函数（function）**”。引入“函数”概念之后，程序中凡需要调用此公用功能的地方都可以只写出函数名，此名字就代表了函数中所包含的所有代码，这样一来，就不再需要在多个地方重复书写这些功能代码。

函数的出现，标志着软件开发进入了结构化编程的时代。调用和编写各种函数是程序员在结构化编程时的主要工作之一。

C#中一个函数的语法格式如下所示：

```
返回值类型  方法名(参数列表)
{
    语句1；
    语句2；
    //…
    return 表达式;
}
```

下面是一个典型的 C#函数示例：

```
int Add(int x, int y)
{
      return x + y;
}
```

函数需要向外界返回一个值，由 return 语句实现。

如果一个函数没有返回值或不关心其返回值，则将其返回值定义为 void。

```
  void f() //不返回任何值的函数
  {
      语句1；
      语句2；
      //…
      return ;  //在return后写一个空语句
  }
```

（2）方法的定义与使用

放在一个类中的函数（通常附加一个存取权限修饰符如 public 和 private）称为“方**法(method)**”。

访问一个方法的最基本方式是通过类创建的对象。例如以下代码在类 MathOpt 中定义了一个 Add()方法：

```
public class MathOpt
{
   public int Add(int x, int y)
   {
      return x + y;
   }
```

```
    }
```

则可以通过使用 new 关键字创建类 MathOpt 的对象来访问此 Add()方法：

```
    class Program
    {
        static void Main(string[] args)
        {
            //创建MathOpt类的对象
            MathOpt obj = new MathOpt();
            //通过对象调用类的方法，结果保存在局部变量中
            int result = obj.Add(100, 200);
            //……
        }
    }
```

（3）方法重载

方法重载是面向对象语言（如 C#）对结构化编程语言（如 C）的一个重要扩充，请看以下代码：

```
    class MathOpt
    {
        //整数相加
        public int Add(int x, int y)
        {
            return x + y;
        }
        //浮点数相加
        public double Add(double x, double y)
        {
            return x + y;
        }
    }
}
```

上述两个函数有以下独特之处：

（1）函数名相同，均为 Add；

（2）参数类型不同，一个为 int，另一个为 double。

这两个同名的函数彼此构成了“**重载**（Overload）”关系。

重载函数的调用代码：

```
    MathOpt mathobj = null;                    //定义MathOpt对象变量
    mathobj = new MathOpt();                   //创建对象
```

```
int IResult=mathobj.Add(100, 200);          //调用类的整数相加方法
double FResult = mathobj.Add(5.5, 9.2);     //调用类的浮点数相加方法
Console.WriteLine("100+200=" + IResult );  //输出结果
Console.WriteLine("5.5+9.2=" + FResult );  //输出结果
```

请注意标为粗体的两个方法调用语句。传给方法实参为浮点数时，将调用参数类型为 double 的 Add(double ，double)方法，传给方法的实参为整数时，调用参数类型为 int 的 Add(int,int)方法。

函数重载是面向对象语言对结构化语言特性的重要扩充，在面向对象的编程中应用极广。

两个构成重载关系的函数必须满足以下条件：

（1）函数名相同。

（2）参数类型不同，或参数个数不同。

需要注意的是，函数返回值类型的不同不是函数重载的判断条件。

比如，函数

```
public long Add(int x, int y){……}
```

就不与函数

```
public int Add(int x, int y) {……}
```

构成重载关系，因为这两个函数的实参类型和数目都相同，仅函数返回值类型不同。

另外要注意 C#是区分大小写的语言，因此，如果一个类中有以下两个函数：

```
public long add(int x, int y){……}
public int Add(int x, int y) {……}
```

则可以通过编译并且程序可以运行，但这并不是说这两个函数构成了重载关系，事实上，C#认为这是两个完全不同的函数，彼此之间一点关系也没有！

2.3 类的静态成员

类中的函数，如果在声明时没有加“static”关键字，则称之为类的“**实例方法（instance method）**”。加了 static 关键字的方法称为“**静态方法（static method）**”。

类似地，加了 static 关键字的字段称为“**静态字段（static field）**”。

（1）访问类的静态成员的基本方法

.NET Framework 提供了大量的静态方法供开发人员使用，最典型的是数学库函数，.NET Framework 将常用的数学函数放到了类 Math 中，例如以下代码计算 2 的 3 次方：

```
double ret = Math.Pow(2, 3);
```

对比前面介绍过的类实例方法的调用方式，可以发现静态方法在使用时不需要创建对象，而是按以下格式直接调用：

```
类名.静态方法名（参数列表）
```

类的实例方法可以直接访问类的公有静态字段，比如数学常数 π 就是 Math 类的公有静态字段，可以被用来计算圆周长：

```
//计算圆周长
double GetPerimeterOfCircle(Double radius)
{
      return 2 * Math.PI * radius;
}
```

（2）类静态成员的特性

类的静态成员有一个特殊的性质，先来看一个例子来说明这一点。

给类 StaticMembers 增加一个普通的实例方法 increaseValue()和实例字段 dynamicVar（参见示例项目 UseStaticMembers)：

```
class StaticMembers
{
   public static int staticVar=0;   //静态字段
   public int dynamicVar=0;
   public void increaseValue()
   {
      staticVar++;
      dynamicVar++;
   }
}
```

在 increaseValue()方法中，对类的静态字段 staticVar 和实例字段 dynamicVar 都进行了自增操作。

以下是测试代码：

```
   static void Main(string[] args)
   {
      StaticMembers obj=null;
      //创建100个对象
      for (int i = 0; i < 100; i++)
      {
         obj = new StaticMembers();
         obj.increaseValue();
```

```
        }
        //查看静态字段与普通字段的值
        Console.WriteLine("dynamicVar=" + obj.dynamicVar);
        Console.WriteLine("staticVar=" + StaticMembers.staticVar);
        //程序暂停，敲任意键继续
        Console.ReadKey();
    }
```

程序的运行结果：

```
dynamicVar=1
staticVar=100
```

因为类的静态成员拥有以下特性：

类的静态成员是供类的所有对象所共享的。

在本节示例中创建了 100 个对象，每个对象拥有 1 个 dyanmicVar 字段，一共有 100 个 dyanmicVar 字段，这些字段是独立的，“互不干涉内政”。而 staticVar 字段仅有一个，为所有对象所共享。因此，任何一个对象对 staticVar 字段的修改，都会被其他对象所感知（图 7）：

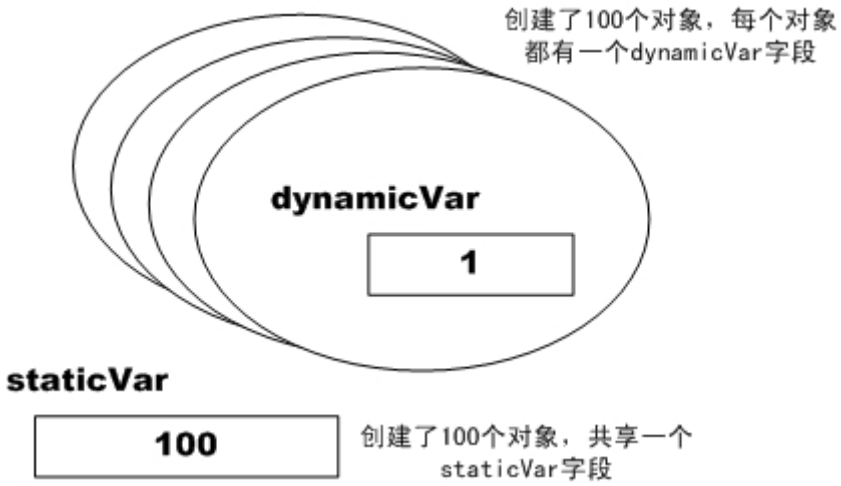

图 7 静态字段与实例字段

（3）类实例成员与静态成员的访问规则

在面向对象的程序中，对类的实例和静态成员，有以下访问规则：

- **位于同一类中的实例方法可直接相互调用。**
- **类的字段（包括实例字段和静态字段）可以被同一类中的所有实例方法直接访问。**
- **类中的静态方法只能直接访问类静态字段。**

上述规则中，需要特别注意在静态方法中直接调用类的实例方法是非法的。例如以下代码将无法通过编译。

```
class StaticMembers
{
```

```
    public static int staticVar=0;   //静态字段
    public static void staticFunc() //静态方法
    {
        increaseValue(); //静态方法中不能调用实例方法 ×
        dynamicVar++;    //静态方法中不能访问实例字段   ×
        staticVar++;     //静态方法可以访问静态字段 正确!
    }
    public int dynamicVar=0;
    public void increaseValue()
    {
        staticVar++;
        dynamicVar++;
    }
}
```

静态方法中只允许访问静态数据,那么,如何在静态方法中访问实例数据?

很简单，可以在静态方法中创建对象，通过对象来访问即可。将静态方法 staticFunc() 修改如下即可通过编译：

```
    public static void staticFunc() //静态方法
    {
        //在静态方法中通过对象访问类的实例成员
        StaticMembers obj = new StaticMembers();
        obj.increaseValue(); //调用实例方法，OK!
        obj.dynamicVar++;    //访问实例字段，OK!
    }
```

2.4 类的属性

属性是一种特殊的“字段”。

先来看一个用于表示学生信息的类 Student：

```
class Student
{
    public String Name;          //姓名
    public DateTime Birthday;    //生日
    public int Age;              //年龄
}
```

Student 类中使用公有字段来表达学生信息，这种方式无法保证数据的有效性。比如外界完全可以这样使用 Student 类：

```
Student stu = new Student();
stu.Name = ""; //非法数据，名字怎能为空?
```

```
stu.Birthday = new DateTime(3000, 1, 3); //公元3000年出生，他来自未来世界？
stu.Age = -1; //年龄必须大于0!
```

在设计类时使用**属性**（Property）可以保证只有合法的数据可以传给对象。

以 Name 这个字段为例，它要求不能为空。

首先，定义一个私有的_Name 字段；

```
private String _Name="姓名默认值";
```

接着，即可定义一个 Name 属性：

```
public String Name
{
    get  //读
    {
        return _Name;
    }
    set  //写，使用隐含变量value
    {
        if (value.Length == 0)
            throw new Exception("名字不能为空");
        _Name = value;
    }
}
```

Name 属性由两个特殊的**读访问器（get）**和**写访问器（set）**组成。

当读取 Name 属性时，读访问器被调用，仅简单地向外界返回私有字段_Name 的值。

当设置 Name 属性时，写访问器被调用，先检查外界传入的值是不是空串，再将传入的值保存于私有字段中。

经过这样的设计，以下的代码在运行时会抛出一个异常提醒程序员出现了错误需要更正：

```
Student stu = new Student();
stu.Name = ""; //非法数据，名字怎能为空？
```

写访问器中有一个特殊的变量 value 必须特别注意，它代表了外界传入的值，例如以下代码向 Name 属性赋值：

```
Student stu = new Student();
stu.Name = "张三";
```

“张三”这一字串值将被保存到 value 变量中，供写访问器使用。

由上述例子可知，编写属性的方法如下：

（1）设计一个私有的字段用于保存属性的数据；

（2）设计 get 读访问器和 set 写访问器存取私有字段数据。

C#中还允许定义只读属性和只写属性。只读属性只有 get 读访问器，而只写属性只有 set 写访问器。

2.5 深入理解类与对象

（1） 类和对象的区别

请看示例程序 UnderstandClassAndObject：

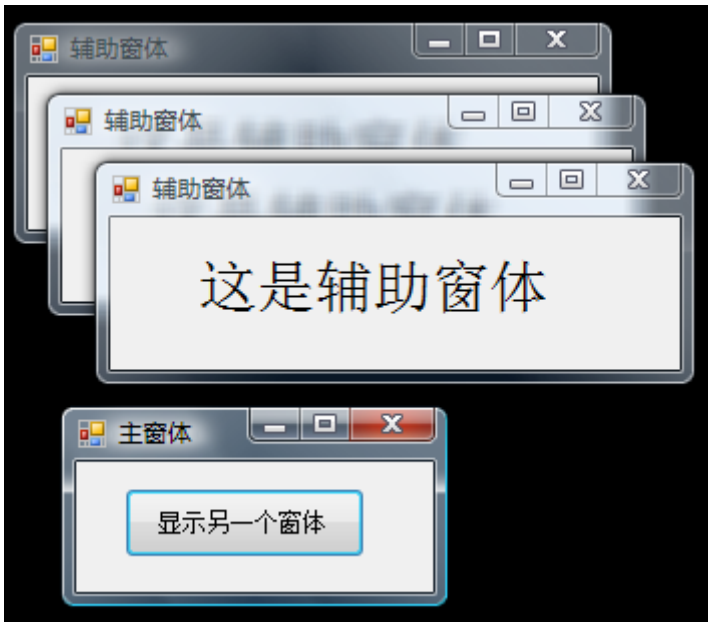

图 8 示例程序 UnderstandClassAndObject

如图 8 所示，每次点击主窗体上的“显示另一个窗体”按钮，都将在屏幕上显示一个“一模一样”的辅助窗体。

在这个示例程序中，一共有两个类：frmMain 代表主窗体，frmOther 代表辅助窗体。

在屏幕上显示的窗体其实是 frmMain 和 frmOther 类创建的对象，主窗体上的按钮单击事件响应代码如下：

```
private void btnShowOtherForm_Click(object sender, EventArgs e)
{
    //创建辅助窗体对象
    frmOther frm = new frmOther();
    //显示辅助窗体
    frm.Show();
}
```

可以看到，每次点击主窗体上的按钮，都会创建一个辅助窗体 frmOther 对象，这正是在屏幕上出现多个辅助窗体的原因所在。

从这个示例程序可以得到以下重要认识：

- **对象是以类为模板创建出来的。类与对象之间是一对多的关系。**

- **在 C#中，使用 new 关键字创建对象。**
- **在程序中“活跃”的是对象而不是类。**

在面向对象领域，对象有时又被称为是“类的实例”，**“对象”与“类的实例”这两个概念是等同的**。

（2）类的构造函数

当使用 new 关键字创建一个对象时，一个特殊的函数被自动调用，这就是类的构**造函数（constructor）**。

在 C#中，类的构造函数与类名相同，没有返回值。

```
class A
{
  //类A的构造函数
   public A()
   {
   }
}
```

类的构造函数在以类为模板创建对象时被自动调用。构造函数一般用于初始化类的私有数据字段。

（3）引用类型与值类型

.NET 将变量的类型分为“**值类型**”与“**引用类型**”两大类。诸如 int 和 float 之类的变量属于值类型，而“类”类型的变量则属于 “引用类型”。

值类型变量与引用类型变量在使用上是有区别的。

值类型的变量一定义之后就马上可用。比如定义“int i;”之后，变量 i 即可使用。

引用类型的变量定义之后，还必须用 new 关键字创建对象后才可以使用，我们在前面已经多次这样使用过引用变量了。

在 Visual Studio 随机文档中，详细地列出了每种数据类型属于值类型还是引用类型（表 1）：

表 1 C#的数据类型

类别		说明
值类型	简单类型	有符号整型： sbyte, short, int, long
		无符号整型： byte, ushort, uint, ulong
		Unicode 字符： char
		IEEE 浮点型： float, double
		高精度小数： decimal
		布尔型： bool
	枚举类型	enum E {...} 形式的用户定义类型
	结构类型	struct S {...} 形式的用户定义类型
引用类型	类类型	所有其他类型的最终基类： object
		Unicode 字符串： string

		class C {...} 形式的用户定义类型
	接口类型	interface I {...} 形式的用户定义类型
	数组类型	一维和多维数组，例如 int[] 和 int[,]
	委托类型	delegate T D(...) 形式的用户定义类型

值类型变量与引用类型变量的内存分配模型也不一样。为了理解清楚这个问题，读者首先必须区分两种不同类型的内存区域：**线程堆栈（Thread Stack）**和**托管堆（Managed Heap）**。

每个正在运行的程序都对应着一个进程（process），在一个进程内部，可以有一个或多个线程（thread），每个线程都拥有一块“自留地”，称为“线程堆栈”，大小为 1M，用于保存自身的一些数据，比如函数中定义的局部变量、函数调用时传送的参数值等，这部分内存区域的分配与回收不需要程序员干涉。

所有值类型的变量都是在线程堆栈中分配的。

另一块内存区域称为“堆（heap）”，在.NET 这种托管环境下，堆由 CLR 进行管理，所以又称为“托管堆（managed heap）”。

用 new 关键字创建的类的对象时，分配给对象的内存单元就位于托管堆中。

在程序中我们可以随意地使用 new 关键字创建多个对象，因此，托管堆中的内存资源是可以动态申请并使用的，当然用完了必须归还。

打个比方更易理解：托管堆相当于一个旅馆，其中的房间相当于托管堆中所拥有的内存单元。当程序员用 new 方法创建对象时，相当于游客向旅馆预订房间，旅馆管理员会先看一下有没有合适的空房间，有的话，就可以将此房间提供给游客住宿。当游客旅途结束，要办理退房手续，房间又可以为其他旅客提供服务了。

从表 1 可以看到，引用类型共有四种：类类型、接口类型、数组类型和委托类型。

所有引用类型变量所引用的对象，其内存都是在托管堆中分配的。

严格地说，我们常说的“对象变量”其实是类类型的引用变量。但在实际中人们经常将引用类型的变量简称为“对象变量”，用它来指代所有四种类型的引用变量。在不致于引起混淆的情况下，本书也采用了这种惯例。

在了解了对象内存模型之后，对象变量之间的相互赋值的含义也就清楚了。请看以下代码（示例项目 ReferenceVariableForCS）：

```
01     class A
02     {
03         public int i;
04     }
05     class Program
06     {
07         static void Main(string[] args)
08         {
09             A a ;
10             a= new A();
11             a.i = 100;
12             A b=null;
13             b = a;  //对象变量的相互赋值
14             Console.WriteLine("b.i=" + b.i); //b.i=?
15         }
```

```
16      }
```

注意第 12 和 13 句。

程序的运行结果是：

```
    b.i=100;
```

请读者思索一下：两个对象变量的相互赋值意味着什么？

事实上，两个对象变量的相互赋值意味着赋值后两个对象变量所占用的内存单元其内容是相同的。

讲得详细一些：

第 10 句创建对象以后，其首地址（假设为“1234 5678”）被放入到变量 a 自身的 4 个字节的内存单元中。

第 12 句又定义了一个对象变量 b，其值最初为 null（即对应的 4 个字节内存单元中为“0000 0000”）。

第 13 句执行以后，a 变量的值被复制到 b 的内存单元中，现在，b 内存单元中的值也为“1234 5678”。

根据前面介绍的对象内存模型，我们知道现在变量 a 和 b 都指向同一个实例对象。

如果通过 b.i 修改字段 i 的值，a.i 也会同步变化，因为 a.i 与 b.i 其实代表同一对象的同一字段。

整个过程可以用图 9 来说明：

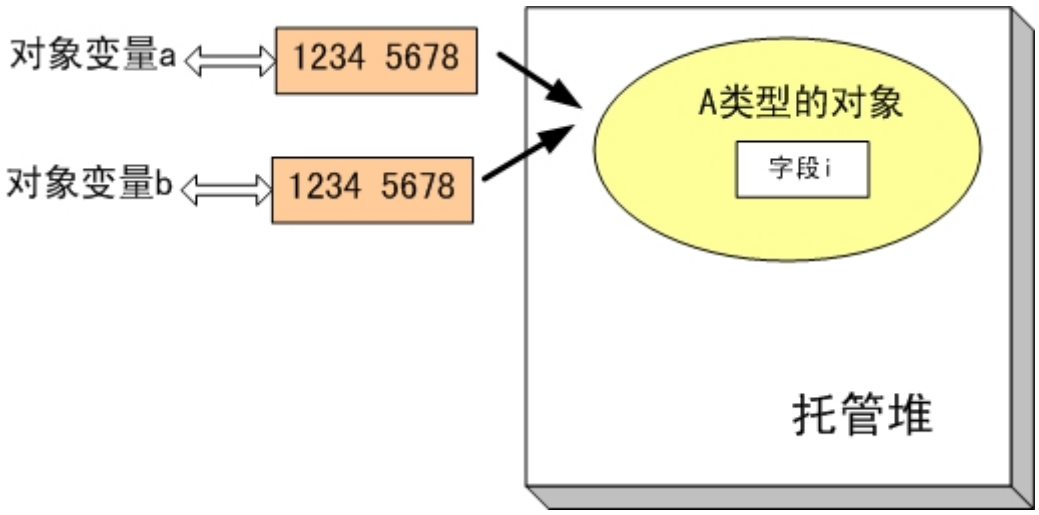

图 9 对象变量的相互赋值

由此得到一个重要结论：

对象变量的相互赋值不会导致对象自身被复制，其结果是两个对象变量指向同一对象。

另外，由于对象变量本身是一个局部变量，因此，**对象变量本身是位于线程堆栈中的。**

严格区分对象变量与对象变量所引用的对象，是面向对象编程的关键之一。

由于对象变量类似于一个对象指针，这就产生了“判断两个对象变量是否引用同一对象”的问题。

C#使用“＝＝”运算符比对两个对象变量是否引用同一对象，“!=”比对两个对象变量

是否引用不同的对象。参看以下代码：

```
//a1与a2引用不同的对象
A a1= new A();
A a2= new A();
Console.WriteLine(a1 == a2);//输出：false
a2 = a1;//a1与a2引用相同的对象
Console.WriteLine(a1 == a2);//输出：true
```

需要注意的是，如果“==”被用在值类型的变量之间，则比对的是变量的内容：

```
int i = 0;
int j = 100;
if (i == j)
{
    Console.WriteLine("i与j的值相等");
}
```

理解值类型与引用类型的区别在面向对象编程中非常关键。

3 命名空间与类库

在使用面向对象技术开发的现代软件系统中，经常拥有数百甚至上千个类，为了方便地管理这些类，面向对象技术引入了“命名空间（namespace）”的概念。

3.1 命名空间

命名空间可以看成是类的“容器”，它可以包含多个类，例如，以下代码创建了一个命名空间 MyDLL，在其中放置了两个类——MyPublicClass 和 MyInnerClass：

```
namespace MyDLL
{
    public class MyPublicClass
    {
    }
    class MyInnerClass
    {
    }
}
```

如果要使用命名空间中的类，需要指明其命名空间。例如：

```
MyDLL.MyPublicClass obj=new MyDLL.MyPublicClass();
```

如果希望少写一点代码，可以在源代码文件开头使用 using 语句引用 MyDLL 命名空间：

```
using MyDLL;
```

在源代码文件中引用了特定的命名空间之后，就可以直接使用其中的类而不再需要指明其所属的命名空间。

下面对命名空间这一概念做更进一步的讨论。

.NET Framework 使用命名空间来管理所有的类。如果把类比喻成书的话，则命名空间类似于放书的书架，书放在书架上，类放在命名空间里。

当我们去图书馆查找一本书时，需要指定这本书的编号，编号往往规定了书放在哪个书库的哪个书架上，通过逐渐缩小的范围：图书馆→书库→书架，最终可以在某个书架中找到这本书。

类似地，可以采用图书馆保存图书类似的方法来管理类，通过逐渐缩小的范围：最大的命名空间→子命名空间→孙命名空间→……，最终找到一个类。

所以，命名空间是可以嵌套的，请看以下示例代码：

```
namespace MyDLL
{
  //…
   namespace MyChildDLL
   {
      public class MyPublicChildClass
      {
      }
   }
}
```

这时，外界可通过以下格式的声明来使用此类型：

```
MyDLL.MyChildDLL.MyPublicChildClass obj2 =
        new MyDLL.MyChildDLL.MyPublicChildClass();
```

同样可以使用 using 语句缩短语句的长度。不再赘述。

3.2 类库

为了提高软件开发的效率，人们在整个软件开发过程中大量应用了软件工程的**模块化原则**，将可以在多个项目中使用的代码封装为可重用的软件模块，其于这些可复用的软件模块，再开发新项目就成为“**重用已有模块，再开发部分新模块，最后将新旧模块组装起来**”的过程。整个软件开发过程类似于现代工业的生产流水线，生产线上的每个环节都由特定的人员负责，整个生产线上的工作人员既分工明确又相互合作，大大地提高了生产效率。

在组件化开发大行其道的今天，人们通常将可以重用的软件模块称为“**软件组件**”。

在全面向对象的.NET 软件平台之上，软件组件的表现形式为**程序集（Assembly）**，可以通过在 Visual Studio 中创建并编译一个类库项目得到一个程序集。

在 Visual Studio 的项目模板中，可以很方便地创建类库（Class Library）项目（图 10）：

图 10 创建类库项目

Visual Studio 会自动在项目中添加一个名为 Class1.cs 的类文件，程序员可在此类文件中书写代码，或者添加新的类。一个类库项目中可以容纳的类数目没有限制，但只有声明为 public 的类可以被外界使用。

类库项目编译之后，会生成一个动态链接库（DLL：Dynamic Link Library）文件。这就是可以被重用的.NET 软件组件——程序集。默认情况下，类库文件名就是项目名加上“.dll”后缀。

每个类库项目都拥有一个默认的命名空间，可以通过类库项目的属性窗口来指定。

需要仔细区分“类库项目”、“程序集”和“命名空间”这三个概念的区别：

（1）每个类库项目编译之后，将会生成一个程序集。

（2）类库项目中可以拥有多个类，这些类可属于不同的命名空间。

（3）不同的类库项目可以定义相同的命名空间。

根据上述三个特性，可以得到以下结论：

“命名空间”是一个逻辑上的概念，它的物理载体是“程序集”，具体体现为“DLL”（或 EXE）文件。在 Visual Studio 中，可通过创建“类库”类型的项目生成程序集。

一个程序集可以有多个命名空间，而一个命名空间也可以分布于多个程序集。

一旦生成了一个程序集，在其他项目中就可以通过添加对这一程序集的引用而使用此程序集中的类。其方法是在“项目”菜单中选择“添加程序集”命令，激活“浏览”卡片，选择一个现有的程序集文件（DLL 或 EXE）。

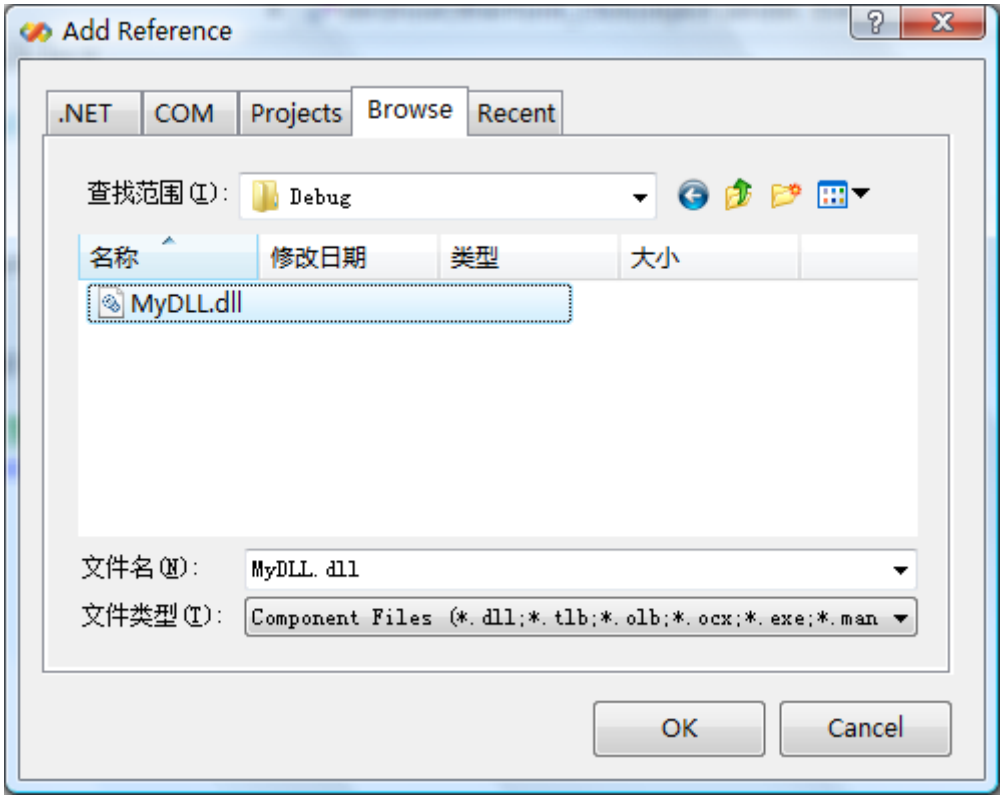

图 11 添加对程序集的引用

一个项目添加完对特定程序集的引用之后，就可以直接创建此程序集中的类了，当然要注意指明其命名空间。

4 继承

继承是面向对象编程中一个非常重要的特性，它也是另一个重要特性——多态的基础。

4.1 继承概念的引入

现实生活中的事物都归属于一定的类别。比如，狮子是一种（IS_A）动物。为了在计算机中摸拟这种关系，面向对象的语言引入了**继承**（inherit）的特性。

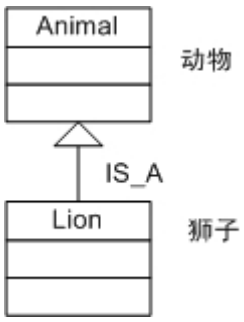

图 12 “继承”的引入

如图 12 所示，用类 Animal 代表动物，用类 Lion 代表狮子，用一个空心的三角箭头表示继承关系。

构成继承关系的两个类中，Animal 称为**父类**（parent class）或**基类**（base class），Lion 称为**子类**（child class）。

提示：

在一些书中，将父类称为**超类**（super class）。

“继承”关系有时又称为“派生”关系，“B 继承自 A”，可以说为“B 派生自 A”，或反过来说，“A 派生出 B”。

父类与子类之间拥有以下两个基本特性：

- **是一种（IS-A）关系**：子类是父类的一种特例。
- **扩充（Extends）关系**：子类拥有父类所没有的功能。

以下 C#代码实现了狮子类与动物类之间的继承关系：

```
class Animal
{
}
class Lion : Animal
{
}
```

可以看到，C#中用一个冒号间隔开父类和子类。

4.2 类成员的访问权限

面向对象编程的一大特点就是可以控制类成员的可访问性。当前主流的面向对象语言都拥有以下三种基本的可访问性（表 2）：

表 2 类成员的访问权限

可访问性	C#关键字	含义
公有	public	访问不受限制
私有	private	只有类自身成员可以访问
保护	protected	子类可以访问，其他类无法访问

（1）public 和 private

public 和 private 主要用于定义单个类的成员存取权限。

请看以下示例代码：

```
public class A
{
    public int publicI;
    private int privateI;
```

```
    protected int protectedI;
}
```

当外界创建一个 A 的对象后，只能访问 A 的公有实例字段 publicI：

```
A a = new A();
a.publicI = 100; //OK!
```

类 A 的私有实例字段 privateI 只能被自身的实例方法所使用：

```
public class A
{
    public int publicI;
    private int privateI;
    protected int protectedI;
    private void f()
    {
        privateI = 100; //OK!
    }
}
```

上述代码中，类 A 的私有方法 f()访问了私有字段 privateI。注意，只要是类 A 直接定义的实例方法，不管它是公有还是私有的，都可以访问类自身的私有实例字段。

（2）protected

在形成继承关系的两个类之间，可以定义一种扩充权限——**protected**。

当一个类成员被定义为 protected 之后，所有外界类都不可以访问它，但其子类可以访问。

以下代码详细说明了子类可以访问父类的哪些部分（示例项目 Inherits）：

```
class Parent
{
    public int publicField=0;
    private int privateFiled=0;
    protected int protectedField=0;
    protected void protectedFunc()
    { }
}
class Son:Parent
{
    public void ChildFunc()
    {
        publicField = 100;        //正确!子类能访问父类的公有字段
        privateFiled = 200;       //错误!子类不能访问父类的私有字段
```

```
            protectedField = 300;     //正确!子类能访问父类的保护字段
            protectedFunc();          //正确!子类能访问父类的保护方法
        }
}
```

当创建子类对象后，外界可以访问子类的公有成员和父类公有成员，如下所示：

```
    Son obj = new Son ();
    //可以调用子类的公有方法
    obj.ChildFunc();
    //可以访问父类的公有字段
    obj.publicField=1000;
```

由此可见，可以通过子类对象访问其父类的所有公有成员，事实上，外界根本分不清楚对象的哪些公有成员来自父类，哪些公有成员来自子类自身。

小结一下继承条件下的类成员访问权限：

- **所有不必让外人知道的东西都是私有的。**
- **所有需要向外提供的服务都是公有的。**
- **所有的“祖传绝招”，“秘不外传”的都是保护的。**

（3）internal

C#中还有一种可访问性，就是由关键字 internal 所确定的“内部”访问性。

internal 有点像 public，外界类也可以直接访问声明为 internal 的类或类的成员，但这只局限于同一个程序集内部。

读者可以简单地将程序集理解为一个独立的 DLL 或 EXE 文件。一个 DLL 或 EXE 文件中可以有多个类，如果某个类可被同一程序集中的类访问，但其他程序集中的类不能访问它，则称此类具有 internal 访问性。

例如类库项目 ClassLibrary1 可以生成一个独立的程序集（假定项目编译后生成 ClassLibrary1.DLL），其中定义了两个类 A 和 B：

```
namespace ClassLibrary1
{
   internal  class A
   {
      internal int InternalI=0;
   }

   public class B
   {
      public void f()
      {
         A a = new A();        //OK!
         a. InternalI= 100;    //OK!
      }
```

```
    }
}
```

由于类 B 与类 A 属于同一个程序集，所以，B 中的代码可以创建 A 的对象，并访问 A 的声明为 internal 的成员 InternalI。

在程序集 ClassLibrary1.DLL 之外，外界只能创建声明为 public 的类 B 的对象，不能创建声明为 internal 的类 A 的对象。

internal 是 C#的默认可访问性，这就是说，**如果某个类没有任何可访问性关键字在它前面，则它就是 internal 的**。

比如上面的类 A 也可以写成：

```
class A
{
   internal  int InternalI=0;
 }
```

它完全等同于

```
internal  class A
{
   internal int InternalI=0;
}
```

但要注意，在类中，如果省略成员的可访问性关键字，则默认为 private 的。

例如：

```
class A
{
    int InternalI=0;
 }
```

相当于

```
internal class A
{
    private int InternalI=0;
}
```

为便于读者查阅，将 C# 2.0 和 Visual Basic 2005 的存取权限总结如表 3 所示。

表 3 类型存取权限一览表

使用场合	C# 2.0	Visual Basic 2005	说明
Type （指类，接口等类型）	public	Public	访问不受限制
	internal	Friend	访问范围仅限于同一程序集

Member （指类型中的成员，比如类中的字段）	public	Public	访问不受限制
	internal	Friend	访问范围仅限于同一程序集
	protected	protected	访问范围限于自己或派生出来的子类型
	protected internal	Protected Friend	在同一程序集内访问不受限制，在不同程序集中仅由此类型派生出来的子类型可访问
	private	Private	仅自己可以访问

4.3 子类父类变量的相互赋值

构成继承关系的父类和子类对象之间有一个重要的特性：

子类对象可以被当成基类对象使用。

. 这是因为子类对象本就是一种（IS_A）父类对象，因此，以下代码是合法的：

```
Parent p;
Son c = new Son ();
p = c;  //正确，子类对象可以传给父类变量
```

上述代码中 Parent 是 Son 类的父类。

然而，反过来就不可以，以下代码是错误的：

```
c = p;  //错误，父类对象变量不可以直接赋值给子类变量
```

如果确信父类变量中所引用的对象的确是子类类型，则可以通过类型强制转换进行赋值，其语法格式为：

```
子类对象变量=(子类名称)基类对象变量;
```

或使用 as 运算符

```
子类对象变量=基类对象变量 as 子类名称;
```

示例代码如下：

```
c = (Child)p; //正确,父类对象变量引用的就是子类对象
```

或

```
c = p as Child ;
```

4.4 方法重载、隐藏与虚方法调用

由于子类对象同时“汇集了”父类和子类的所有公共方法，而 C#并未对子类和父类的方法名称进行过多限制，因此，一个问题出现了：

如果子类中某个方法与父类方法的签名一样（即方法名和方法参数都一样），那当通过子类对象访问此方法时，访问的是子类还是父类所定义的方法?

让我们先从子类方法与父类方法之间的关系说起。

总的来说，子类方法与父类方法之间的关系可以概括为以下三种。

- **扩充（Extend)**：子类方法，父类没有；
- **重载（Overload)**：子类有父类的同名函数，但参数类型或数目不一样；
- **完全相同**：子类方法与父类方法从方法名称到参数类型完全一样。

对于第一种“扩充”关系，由于子类与父类方法不同名，所以不存在同名方法调用的问题，重点分析一下后两种情况.

（1）重载（overload）

在前面介绍过方法重载的概念，在同一个类中构成重载的方法主要根据参数列表来决定调用哪一个。这一基本判断方法可以推广到类的继承情况。

例如，以下代码在子类和父类中定义了一个重载的方法 OverloadF()：

```
class Parent
{
   public void OverloadF()
   {
   }
}
class Child:Parent
{
   public void OverloadF(int i)
   {
   }
}
```

使用代码如下：

```
Child obj = new Child();
obj.OverloadF();    //调用父类的重载方法
obj.OverloadF(100);//调用子类的重载方法
```

可以看到，虽然重载的方法分布在不同的类中，但仍然可以将其看成是定义在同一个类中的，其使用方式与调用类的其他方法并无不同。

（2）隐藏（Hide）

当子类与父类拥有完全一样的方法时，称“子类**隐藏**了父类的同名方法”，请看示例项

目 HideExamples：

```
class Parent
{
    public void HideF()
    {
        System.Console.WriteLine("Parent.HideF()");
    }
}
class Child : Parent
{
    public void HideF()
    {
        System.Console.WriteLine("Child. HideF()");
    }
}
```

请注意现在子类和父类都拥有了一个完全相同的方法 HideF()，于是问题发生了，请看以下代码：

```
Child c = new Child();
c.HideF();//调用父类的还是子类的同名方法?
```

上述代码运行时，输出：

```
Child.HideF()
```

修改一下代码：

```
Parent p = new Parent();
p.HideF();//调用父类的还是子类的同名方法?
```

上述代码运行结果：

```
Parent.HideF()
```

由此可以得出一个结论：

当分别位于父类和子类的两个方法完全一样时，调用哪个方法由对象变量的类型决定。

然而，面向对象的继承特性允许子类对象被当成父类对象使用，这使问题复杂化了，请读者看以下代码，想想会出现什么结果？

```
Child c = new Child();
Parent p;
p = c;
```

```
p.HideF();//调用父类的还是子类的同名方法?
```

上述代码的运行结果是：

```
Parent.HideF()
```

这就意味着即使 Parent 变量 p 中实际引用的是 Child 类型的对象，通过 p 调用的方法还是 Parent 类的！

如果确实希望调用的子类的方法，应这样使用：

```
((Child)p).HideF();
```

即先进行强制类型转换。

回到前面 Parent 和 Child 类的定义，Visual Studio 在编译这两个类时，会发出一个警告：

```
警告 1    “HideExamples.Child.HideF()”隐藏了继承的成员
“HideExamples.Parent.HideF()”。如果是有意隐藏，请使用关键字 new。
```

虽然上述警告并不影响程序运行结果，却告诉我们代码不符合 C#的语法规范，修改 Child 类的定义如下：

```
class Child : Parent
{
    public new void HideF()
    {
        System.Console.WriteLine("Child.HideF()");
    }
}
```

“new”关键字明确告诉 C#编译器，子类隐藏父类的同名方法，提供自己的新版本。

由于子类隐藏了父类的同名方法，所以如果要在子类方法的实现代码中调用父类被隐藏的同名方法，请使用 base 关键字，示例代码如下：

```
base.HideF(); //调用父类被隐藏的方法
```

（3）重写（override）与虚方法调用

上述隐藏的示例中，由于子类隐藏了父类的同名方法，如果不进行强制转换，就无法通过父类变量直接调用子类的同名方法，哪怕父类变量引用的是子类对象。

这是不太合理的。我们希望**每个对象都只干自己职责之内的事**，即如果父类变量引用的是子类对象，则调用的就是子类定义的方法，而如果父类变量引用的就是父类对象，则调用的是父类定义的方法。这就是说，希望每个对象都“各人自扫门前雪，莫管他人瓦上霜”。

为达到这个目的，可以在父类同名方法前加关键字 virtual，表明这是一个虚方法，子类可以重写此方法：即在子类同名方法前加关键字 override，表明对父类同名方法进行了重写。

请看示例代码（示例项目 VirtualExamples）：

```
class Parent
{
   public virtual void OverrideF()
    {
        System.Console.WriteLine("Parent.OverrideF()");
    }
}
class Child : Parent
{
   public override void OverrideF()
    {
        System.Console.WriteLine("Child.OverrideF()");
    }
}
```

请看以下使用代码：

```
Child c = new Child();
Parent p;
p = c;
p.OverrideF();//调用父类的还是子类的同名方法？
```

上述代码的运行结果是：

```
Child.OverrideF()
```

这一示例表明，将父类方法定义为虚方法，子类重写同名方法之后，通过父类变量调用此方法，到底是调用父类还是子类的，由父类变量引用的真实对象类型决定，而与父类变量无关！

换句话说，同样一句代码：

```
p.OverrideF();
```

在 p 引用不同对象时，其运行的结果可能完全不一样！因此，如果我们在编程时只针对父类变量提供的对外接口编程，就使我们的代码成了“变色龙”，传给它不同的子类对象（这些子类对象都重写了父类的同名方法），它就干不同的事。

这就是面向对象语言的“**虚方法调用（Virtual Method Invoke）**”特性。

很明显，“虚方法调用”特性可以让我们写出非常灵活的代码，大大减少由于系统功能扩充和改变所带来的大量代码修改工作量。

由此给出以下结论：

面向对象语言拥有的“虚方法调用”特性，使我们可以只用同样的一个语句，在运行时根据对象类型而执行不同的操作。

5 抽象基类与接口

5.1 抽象类与抽象方法

在一个类前面加上“abstract”关键字，此类就成为了抽象类。

对应地，一个方法类前面加上“abstract”关键字，此方法就成为了抽象方法。

```
abstract class Fruit    //抽象类
{
    public abstract void GrowInArea(); //抽象方法
}
```

注意抽象方法不能有实现代码，在函数名后直接跟一个分号。

抽象类专用于派生出子类，子类必须实现抽象类所声明的抽象方法，否则，子类仍是抽象类。

抽象类一般用于表达一种比较抽象的事物，比如前面所说的“水果”，而抽象方法则说明此抽象类应该具有的某种性质，比如 Fruit 类中有一个抽象方法 GrowInArea()，说明水果一定有一个最适合其生长的地区，但不同的水果生长地是不同的。

从同一抽象类中继承的子类拥有相同的方法（即抽象类所定义的抽象方法），但这些方法的具体代码每个类都可以不一样，如以下两个类分别代表苹果（Apple）和菠萝（Pineapple）：

```
class Apple : Fruit //苹果
{
    public override void GrowInArea()
    {
        Console.WriteLine("南方北方都可以种植我。");
    }
}
class Pineapple : Fruit //菠萝
{
    public override void GrowInArea()
    {
        Console.WriteLine("我喜欢温暖，只能在南方看到我。");
    }
}
```

注意上述代码中的 override 关键字，这说明子类重写了基类的抽象方法。抽象类不能创建对象，一般用它来引用子类对象。

```
Fruit f;
f = new Apple();
f.GrowInArea();
f = new Pineapple();
```

```
f.GrowInArea();
```

运行结果：

```
南方北方都可以种植我。
我喜欢温暖，只能在南方看到我。
```

注意同一句代码“f.GrowInArea();”会由于 f 所引用的对象不同而输出不同的结果。可以看到，代码运行结果类似于上一节介绍的“虚方法调用”，两者没有本质差别。

可以按照以下公式编写代码：

抽象类 抽象类变量名=new 继承自此抽象类的具体子类名（）；

一个抽象类中可以包含非抽象的方法和字段。因此：

包含抽象方法的类一定是抽象类，但抽象类中的方法不一定是抽象方法。

5.2 抽象属性

除了方法可以是抽象的之外，属性也可以是抽象的，请看以下代码：

```
abstract class Parent
{
    public abstract String Message   //抽象属性
    {
        get;
        set;
    }
}

class Child : Parent
{
    private String _msg;
    public override  String Message
    {
        get
        {
            return _msg;
        }
        set
        {
            _msg=value;
        }
    }
}
```

使用代码：

```
Parent p = new Child();
p.Message = "Hello";
```

5.3 接口

来看以下这句话：

```
鸭子是一种鸟，会游泳，同时又是一种食物。
```

如何在面向对象的程序中表达这种关系？

如果使用 C++，可以设计成让鸭子（Duck）类继承自两个父类（鸟 Bird 和食物 Food）。但在 C# 中所有的类都只能有一个父类，此方法不可行。

为了解决这一问题，C#引入了接口（interface）这一概念，并规定“**一个类可以实现多个接口**”。

（1）接口的定义与使用

关键字 interface 用于定义接口（示例项目 InterfaceExamples）：

```
//定义两个接口
public interface ISwim
{
    void Swim();
}
public interface IFood
{
    void Cook();
}
```

接口可以看成是一种“纯”的抽象类，它的所有方法都是抽象方法。

可以用与继承相同的语法定义一个类实现某些接口：

```
//定义一个抽象类
public abstract class Bird
{
    public abstract void Fly();

}
//继承自一个抽象类，实现两个接口
public class Duck : Bird, IFood, ISwim
{
    //实现ISwim接口
```

```
    public  void Swim()
    {
        Console.WriteLine("是鸭子就会游泳");
    }
    //实现IFood接口
    public void Cook()
    {
        Console.WriteLine("鸭子经常被烧烤，北京烤鸭就很有名");
    }
    //实现抽象类Bird中的抽象方法
    public override void Fly()
    {
        Console.WriteLine("只有野鸭才会飞");
    }
}
```

可以看到，抽象类定义了对象所属的类别，而接口实际上定义了一种对象应具有的行为特性。

可按以下公式使用接口：

接口类型名 变量名＝new 实现了接口的类型名（）；

示例代码如下：

```
static void Main(string[] args)
{
    Duck d = new Duck();
    //Duck对象d可以使用3种方法：
    //1.自身定义的；
    //2.父类定义的
    //3.接口定义的
    d.Fly();
    d.Cook();
    d.Swim();
    //将子类（Duck）对象赋给基类变量
    Bird b = d;
    //现在只能使用基类定义的Fly()方法
    b.Fly();
    //将Duck对象赋给ISwin接口变量
    ISwim s = d;
    //现在只能使用接口定义的Swim()方法
    s.Swim();
    //将Duck对象赋给另一个实现的接口IFood接口变量
    IFood f = d;
```

```
        //现在只能使用接口定义的Cook()方法
        f.Cook();
    }
```

请读者仔细地阅读上述代码的注释，由于 Duck 类继承自抽象基类 Bird，又实现了 ISwim 和 IFood 两个接口，所以，Duck 对象拥有这三者所定义的所有方法，并且可以赋值给这三种类型的变量。

需要注意的是，虽然程序中始终都只有一个 Duck 对象，但将其赋值给不同类型的变量后，其可以使用的方法是不一样的。

（2）显式实现接口

上面讲到，某个类可以实现多个接口，当创建一个此类的对象之后，通过引用这个对象的对象变量可以访问其所有的公有方法（包括自身的公有方法以及由接口定义的公有方法以）。在这种情况下，根本分不清哪些方法是由接口定义的，哪些是由类自己定义的。C#提供了一种“显式接口”实现机制，可以区分开这两种情况，一个示例代码如下：

```
interface IMyInterface
{
    void func();
}
public class A:IMyInterface
{
    void IMyInterface.func()
    {
        //……
    }
    public void func2()
    {
        //……
    }
}
```

请注意在方法 func 前以粗体突出显示的接口名称，这就是 C#对接口 IMyInterface 的显式实现方式。

当类 A 显式实现接口 IMyInterface 之后，只能以下面这种方式访问接口定义的方法：

```
IMyInterface  a = new A();
a.func();
```

以下代码将不能通过编译：

```
A  a = new A();
a.func();
```

由此得到一个结论：

如果一个类显式实现某个接口，则只能以此接口类型的变量为媒介调用此接口所定义的方法，而不允许通过类的对象变量直接调用。

或者这样说：

被显式实现的接口方法只能通过接口实例访问，而不能通过类实例直接访问。

6 多态

多态编程并非什么新鲜的技术，在前面介绍继承与接口时，就多次使用基类变量引用子类对象，或使用接口变量引用实现了此接口的对象。这其实就是多态编程。

多态编程的基本原理是：

使用基类或接口变量编程。

在多态编程中，基类一般都是抽象基类，其中拥有一个或多个抽象方法，各个子类可以根据需要重写这些方法。

或者使用接口，每个接口都规定了一个或多个抽象方法，实现接口的类根据需要实现这些方法。

因此，多态的实现分为两大基本类别：继承多态和接口多态。

6.1 继承多态

假设某动物园管理员每天需要给他所负责饲养的狮子、猴子和鸽子喂食。我们用一个程序来模拟他喂食的过程。

首先，建立三个类分别代表三种动物（图 13）：

图 13 三种动物

饲养员用一个 Feeder 类来表示。由于三种动物吃的东西不一样，Feeder 类就必须拥有三个喂动物的公有方法（图 14）：

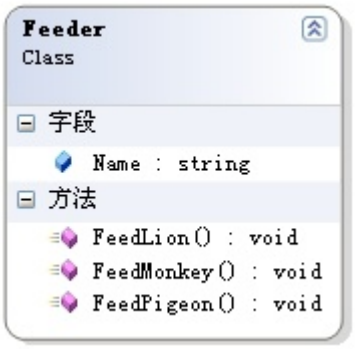

图 14 饲养员类

饲养员小李喂食的过程如下：

```
static void Main(string[] args)
{
    Monkey m = new Monkey();
    Pigeon p = new Pigeon();
    Lion l = new Lion();

    Feeder f = new Feeder();
    f.Name = "小李";

    f.FeedMonkey();  //喂猴子
    f.FeedPigeon();  //喂鸽子
    f.FeedLion();    //喂狮子
}
```

如果动物园领导看小李工作努力，又把大熊猫交给他管理。这时，我们的程序不得不给 Feeder 类增加第四个方法：FeedPanda()。

万一小李后来又不管理鸽子了，那不又得从 Feeder 类中删除 FeedPigeon()方法吗？

这种编程方式很明显是不合理的。

可以应用多态的方法解决。

很明显，狮子、猴子和鸽子都是一种动物，因此，可以建立一个 Animal 抽象基类，让狮子、猴子和鸽子从其派生出来（图 15）。

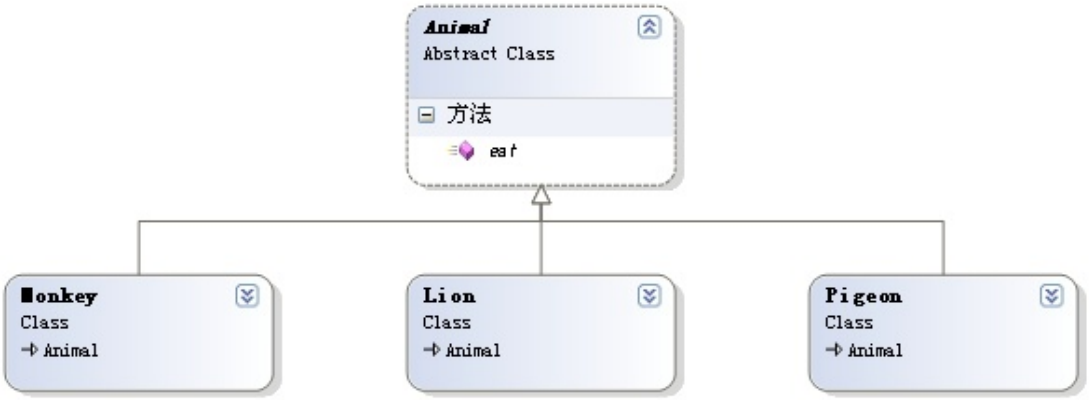

图 15 建立继承体系

由于不同的动物吃不同的食物，所以在 Animal 类中定义一个抽象的 eat()方法，由子类负责实现此方法。

```
abstract class Animal
{
    public abstract void eat();
}
//狮子
class Lion:Animal
{
    public override  void eat()
    {
        //吃肉
    }
}
//猴子
class Monkey:Animal
{
    public override void eat()
    {
        //吃香蕉
    }
}
//鸽子
class Pigeon:Animal
{
    public override void eat()
    {
        //吃大米
    }
}
```

现在，可以将 Feeder 类的三个喂养方法合并为一个 FeedAnimal（图 16）：

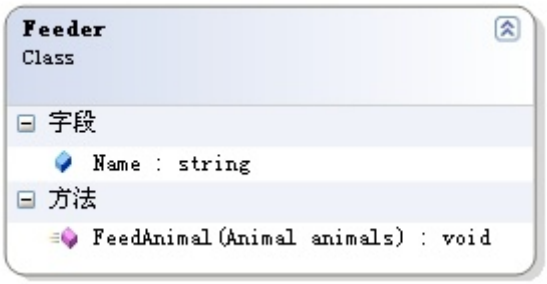

图 16 方法合并

Feeder 类代码如下：

```
//动物园饲养员
class Feeder
{
   public String Name;

   public void FeedAnimal(Animal animals)
   {
      animals.eat();
   }
}
```

现在，喂养过程变为：

```
static void Main(string[] args)
{
      Monkey m = new Monkey();
      Pigeon p = new Pigeon();
      Lion l = new Lion();

      Feeder f = new Feeder();
      f.Name = "小李";

      f.FeedAnimal(m);//喂猴子
      f.FeedAnimal(p);//喂鸽子
      f.FeedAnimal(l);//喂狮子
}
```

上述代码中有连续三句的动物喂养语句，还可以进一步使用多态的方法消除之。
修改 Feeder 类的定义，给其增加一个新方法：FeedAnimals()，新方法完成的功能是喂

养一群动物，它接收一个类型为 Animal 的数组：

```
//动物园饲养员
class Feeder
{
   //……
    //喂养一群动物
    public void FeedAnimals(Animal[] ans)
    {
        foreach (Animal an in ans)
        {
            an.eat();
        }
    }
}
```

喂养过程现在的代码如下：

```
    static void Main(string[] args)
    {
        //动物数组
        Animal[] ans={new Monkey(),new Pigeon(),new Lion()};
        Feeder f = new Feeder();
        f.Name = "小李";
        f.FeedAnimals(ans);
    }
```

上述代码中数组 ans 的元素类型为 Animal，因此，可以在其中存入任何一个 Animal 的子类。具有这种特性的数组称为“**多态数组**”。

简化后的代码变得非常简洁。对比最早的使用三个独立方法实现的方法，新代码适应性大大增强。不管有几种动物，也不管每种动物有多少只，只要将所有这些动物都“塞”进多态数组中，Feeder 类的 FeedAnimals 方法不用改就可以使用。

6.2 接口多态

我们从分析一个现成软件的设计方法入手了解接口多态的形式与使用方法。

请读者仔细研究一下 Word 的绘图功能（图 17）：

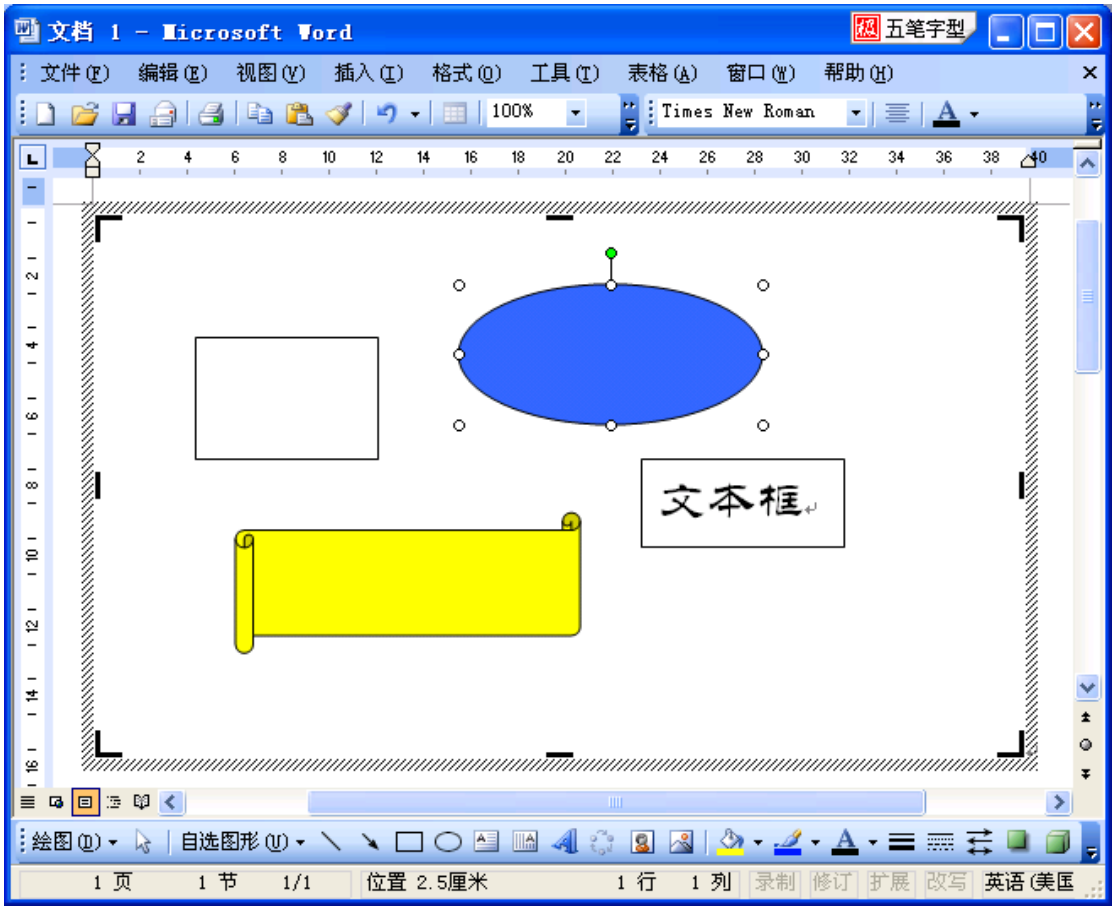

图 17 Word 的绘图功能

如果读者比较熟悉 Word 的绘图功能的话，一定知道在 Word 中可以方便地绘制各种图形，而且可以对图形做各种操作，比如移动、缩放、旋转、填充等。

Word 的绘图功能其实相当于一个小型的矢量绘图软件。

矢量绘图软件的设计与开发，是一个典型的面向对象程序设计与开发过程。

下面我们从面向对象的角度，思索一下如何克隆 Word 的绘图功能为一个小型的绘图软件。

首先，这个软件应该拥有绘制多种图形的功能。因此，可以建立一个图形对象继承体系（图 18）：

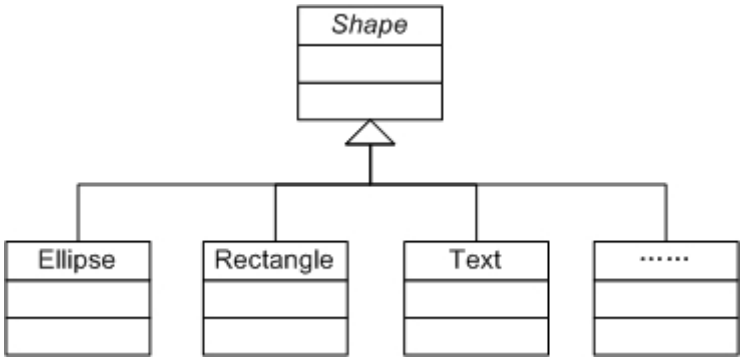

图 18 图形对象的继承体系

拥有了这样一个图形对象的继承树之后，每一绘图页上的图形都可以看成是 Shape 对象的集合[2]。比如图 18 所示的绘图页上就拥有一个 Ellipse 对象、一个 Text 对象和一个 Rectangle 对象。将这些对象放入到一个多态对象集合中，就保存了当前页面上的图形。

用户在页面上放置一个图形，相当于往多态集合中增加一个对象，删除一个图形，相当于在多态对象集合中删除一个对象。

将这一多态集合保存到磁盘文件，就相当于保存了用户的绘图结果。当打开磁盘文件时，在内存中重建多态对象集合，依次访问集合中的每一个对象，通知它们“自己绘制自己”，就重现了用户上次保存时的绘图结果。从这点出发，就知道应该给 Shape 基类增加一个抽象方法：DrawMyself()，让每个图形子类去负责实现具体的绘图方法。

现在考虑对图形对象进行的操作。

一旦用户在页面上放置了一个图形对象，他就可以使用移动、缩放、旋转、填充图形等功能。如何用面向对象的特性来实现这些图形操作？是否也可以用继承来实现？

象 C#和 Java 这类的面向对象语言，一个类只能有一个父类，而且，不同的图形对象可以拥有不同的操作，比如对于矩形对象，可以同时改变其长和宽，但对直线对象而言，只有改变长度是允许的操作，因此，不太适合使用继承来实现操作图形对象的功能。

比较好的方案是将常见操作抽象为接口。哪种图形对象可以进行哪种操作，就让它实现这个接口（图 0-19）：

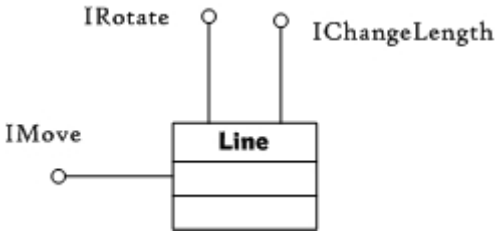

图 0-19 用接口模拟图形操作功能

如图 0-19 所示，直线 Line 对象实现了三个接口，因而就可以移动（IMove），旋转（IRotate）和改变长度（IChangeLength）。

因此，就有可能编写这样一个方法，移动任何一个“可移动的”图形对象：

```
public void MoveShape(IMove obj)
{
        Obj.Move();
}
```

类似地，可以编写这样一个方法，同时移动多个“可移动的”对象：

```
public void MoveShapes(IMove[] objs)
{
    Foreach(Shape s in objs)
    {
        s.Move();
```

[2] 在.NET 中，ArrayList、数组、List 等的实例都可以包容多个对象，都是对象集合。它们的使用方法请查看 MSDN。

```
        }
    }
```

上述代码都是针对接口变量进行编程的，不涉及任何具体的图形类，因而，可以移动任何一个或一组实现了特定接口的图形对象。

这就是“接口多态”在实际编程中的应用。

可以看到，接口多态与继承多态其编程方式与作用都是类似的。但由于一个类可以实现多个接口，所以，接口多态较继承多态更灵活，因而在编程中也用得更广。

多态是面向对象技术中最精华的部分之一。大量的精巧软件设计方案都建立在对多态特性的巧妙应用上。

在编程中应用多态，可以将其简化为以下两句：

- **应用继承实现对象的统一管理。**
- **应用接口定义对象的行为特性。**

对比传统的不使用多态的编程方式，使用多态的好处是：

当要修改程序并扩充系统时，需要修改的地方较少，对其他部分代码的影响较小。

7 委托

委托是一种新的面向对象语言特性，在历史比较长的面向对象语言比如 C++中并未出现过。微软公司在设计运行于.NET Framework 平台之上的面向对象语言（如 C#和 Visual Basic.NET）时引入了这一新特性。

7.1 理解委托的概念

我们都很熟悉常用的数据类型（如 int）的使用方法：先定义一个变量，然后再给其赋值，如下所示：

```
int i;  //定义变量
i=100;  //给变量赋值
```

委托（delegate）也可以看成是一种数据类型，可以用于定义变量。但它是一种特殊的数据类型，它所定义的变量能接收的数值只能是一个函数，更确切地说，委托类型的变量可以接收一个函数的地址，很类似于 C＋＋语言的函数指针。

简单地说：

委托变量可看成是一种类型安全的函数指针，它只能接收符合其要求的函数地址。

来看一个例子（示例项目 FirstDelegateExample），这是一个控制台项目。

示例项目中定义了一个类 MathOpt，其中有一个方法 Add：

```
public class MathOpt
{
    public int Add(int argument1, int argument2)
    {
        return argument1 + argument2;
```

```
    }
}
```

示例项目接着定义了一个委托数据类型 MathOptDelegate，注意加粗的“delegate”关键字：

```
public delegate int MathOptDelegate(int value1,int value2);
```

示例项目中上述定义语句放在两个类（MathOpt 和 Program）之外。
定义好了委托数据类型，在 Main()方法中即可定义一个此委托类型的变量：

```
MathOptDelegate oppDel;
```

接着可以给此变量赋值：

```
MathOpt obj = new MathOpt();
oppDel = obj.Add;
```

您注意到了吗？**委托变量接收一个对象的方法引用**。
赋值之后的委托变量可以当成普通函数一样使用：

```
Console.WriteLine(oppDel(1, 2)); //输出 3
```

上述语句实际上相当于以下语句：

```
Console.WriteLine(obj.Add(1, 2)); //输出 3
```

示例项目 FirstDelegateExample 的完整代码如下所示：

```
public class MathOpt
{
    public int Add(int argument1, int argument2)
    {
        return argument1 + argument2;
    }
}

public delegate int MathOptDelegate(int value1, int value2);

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

```
        MathOpt obj = new MathOpt();
        oppDel = obj.Add;
        Console.WriteLine(oppDel(1, 2)); //输出 3
    }
}
```

从这个示例项目中可以得到这样一个直观的印象：**委托可以看成是一个函数的“容器”，将某一具体的函数“装入”后，就可以把它当成函数一样使用**。

其实，委托并不是函数的“容器”，它是一个派生自 Delegate 的类，但从使用角度出发，将其理解为函数“容器”也是可以的。

那么，是不是所有的函数都可以赋值给委托类型 MathOptDelegate 的变量 oppDel 呢？

请注意 MathOptDelegate 的定义语句，它规定了委托类型 MathOptDelegate 的变量只能接收这样的函数：

拥有两个 int 类型的参数，并且返回值类型也是 int。

只要是满足上述要求的函数，不管其名字如何，也不管它是静态的还是实例的，都可以传给委托类型 MathOptDelegate 的变量 oppDel，并通过 oppDel 来“间接地”调用它们。

定义委托类型时对函数的要求被称为函数的“**签名（signature）**”。

函数的签名规定了函数的参数数目和类型，以及函数的返回值，体现了函数的本质特征。

每一个委托都确定了一个函数的签名。拥有不同签名的函数不能赋值给同一类型的委托变量。

因此，**一个委托类型的变量，可以引用任何一个满足其要求的函数。**

7.2 委托的组合与分解

委托变量可以代表某一函数，使用委托变量就相当于调用一个函数。如果仅是这么简单，那么直接调用函数不就行了吗？为什么还要引入“委托”这一特性？

事实上，委托不仅可以代表一个函数，还可以组合“一堆”的函数，然后批量执行它们。

请看示例程序 UseDelegate，它展示了委托变量之间的组合与分解。

项目中定义了一个委托类型 MyDelegate：

```
delegate void MyDelegate(string s);
```

接着定义了一个拥有两个静态方法的类 MyClass：

```
class MyClass
{
    public static void Hello(string s)
    {
        Console.WriteLine("您好, {0}!", s);
    }
    public static void Goodbye(string s)
    {
        Console.WriteLine("再见, {0}!", s);
    }
}
```

请仔细看以下代码：

```
class Program
{
   static void Main(string[] args)
   {
       MyDelegate a, b, c, d;

      // 创建引用 Hello 方法的委托对象 a：
      a = MyClass.Hello ;
      Console.WriteLine("调用委托变量 a:");
      a("a");

      // 创建引用 Goodbye 方法的委托对象 b：
      b = MyClass.Goodbye;
      Console.WriteLine("调用委托变量 b:");
      b("b");

      // a 和 b 两个委托合成 c，
      c = a + b;
      Console.WriteLine("调用委托变量 c:");
      c("c=a+b");  // c 将按顺序调用两个方法

      // 从组合委托c中移除 a ，只留下b，用d代表移除结果，
      d = c - a;
      Console.WriteLine("调用委托变量 d:");
      d("d=c-a");// 后者仅调用 Goodbye 方法：
   }
}
```

上述代码中委托变量 c 组合了两个委托变量 a 和 b，因而，它拥有两个函数，当执行“c("c=a+b");”时，将导致 MyClass 类的两个静态函数都被执行。

象 c 这样的委托变量又称为“**多路委托变量**”。

可以用加法运算符来组合单个委托变量为多路委托变量。类似地，也可以使用减法运算符来从一个多路委托变量中移除某个委托变量。

上述示例的运行结果为：

```
调用委托变量 a:
您好, a!
调用委托变量 b:
再见, b!
调用委托变量 c:
您好, c=a+b!
```

```
再见, c=a+b!
调用委托变量 d:
再见, d=c-a!
```

8 事件

事件的主要特点是一对多关联，即一个事件源，多个响应者。在具体技术上，.NET Framework 的事件处理机制是基于多路委托实现的。

8.1 事件与多路委托

先看一个多路委托示例项目 MulticastDelegateLinkExample。
首先定义一个委托：

```
public delegate void MyMultiDelegate(int value );
```

接着，定义事件发布者与响应者类：

```
//事件发布者类
public class  Publisher
{
    public MyMultiDelegate handlers; //事件响应者清单
}

//事件响应者类
public class Subscriber
{
    //事件处理函数
    public void MyMethod(int i )
    {
        Console.WriteLine(i);
    }
}
```

以下为模拟实现事件响应的代码：

```
    static void Main(string[] args)
    {
        //一个事件源对象
        Publisher p = new Publisher();
        //两个事件响应者
        Subscriber s1 = new Subscriber();
```

```
        Subscriber s2 = new Subscriber();
        //可以直接调用Delegate类的静态方法组合多个委托
        p.handlers = System.Delegate.Combine(p.handlers, new
              MyMultiDelegate(s1.MyMethod)) as MyMultiDelegate;
        p.handlers = System.Delegate.Combine(p.handlers, new
              MyMultiDelegate(s2.MyMethod)) as MyMultiDelegate;
        //或调用＋＝运算符组合委托
        //p.handlers += new MyMultiDelegate(s1.MyMethod);
        //p.handlers += new MyMultiDelegate(s2.MyMethod);
        //最简单的写法
        //p.handlers += s1.MyMethod;
        //p.handlers += s2.MyMethod;

        //直接调用委托变量，代表激发事件
        p.handlers(10);
    }
```

上述代码执行到最后一句时，将会调用两个事件响应者 s1 和 s2 的事件响应函数 MyMethod，在控制台窗口输出两个整数：

```
10
10
```

上面这个例子中，事件的激发是在 Main()函数中引发的（即上述代码的最后一句），而真实的事件不应允许由外界引发，必须由事件源对象自己引发。

为了限制事件的激发只能由事件源对象自己引发，C#引入了一个新的关键字——event，为此需要修改 UseMultiDelegateExample 项目（参看项目 UseEventExample）：

```
public delegate void MyMultiDelegate(int value);

//事件发布者类
public class Publisher
{
    public event MyMultiDelegate handlers; //定义一个事件
    //激发事件
    public void FireEvent()
    {
        handlers(10);
    }
}

//事件响应者类
public class Subscriber
{
```

```
    //事件处理函数
    public void MyMethod(int i)
    {
        Console.WriteLine(i);
    }
}
```

与前不同之处在于 Publisher 类给 handlers 字段增加了一个 event 关键字，并提供了一个新的用于激发事件的方法 FireEvent()。

以下为模拟实现事件响应的代码：

```
    static void Main(string[] args)
    {
        Publisher p = new Publisher();
        Subscriber s1 = new Subscriber();
        Subscriber s2 = new Subscriber();
        //声明为事件的委托无法直接调用Combine方法
        //以下两句将无法通过编译
        //p.handlers = System.Delegate.Combine(p.handlers,
                  new MyMultiDelegate(s1.MyMethod)) as MyMultiDelegate;
        //p.handlers = System.Delegate.Combine(p.handlers,
                  new MyMultiDelegate(s2.MyMethod)) as MyMultiDelegate;
        //必须使用+=运算符给事件追加委托
        p.handlers+=new MyMultiDelegate(s1.MyMethod);
        p.handlers+=new MyMultiDelegate(s2.MyMethod);
        //声明为事件的委托也不能直接调用，下面这句无法通过编译
        //p.handlers(10);
        //只能通过类的公有方法间接地引发事件
         p.FireEvent();
     }
```

请注意上述代码中被注释掉的代码，它们是无法通过编译的，只能使用“+=”给 handles 事件追加委托，也只能通过类的公有方法来间接地激发此事件。

对比以上两个示例，不难看出事件与多路委托其实大同小异，只不过多路委托允许在事件源对象之外激发事件罢了。

8.2 Visual Studio 窗体事件机制剖析

在一个 C# Windows 应用程序中，往窗体上拖一个按钮 button1，在窗体设计器中双击这一按钮，Visual Studio 将会生成一个函数框架并在代码编辑器中打开：

```
private void button1_Click(object sender, EventArgs e)
{
}
```

这个函数就是按钮单击事件的事件处理函数。

注意这一函数有两个参数，第一个参数代表了事件源对象，第二个参数代表与事件相关的信息。

对每一个 C#窗体，Visual Studio 都会自动生成一个“窗体名.Desinger.cs”文件，打开它，可以看到以下框架代码：

```
01     partial class Form1
02     {
03         //……
04         private void InitializeComponent()
05         {
06             this.button1 = new System.Windows.Forms.Button();
07             //……
08             this.button1.Click += new
                       System.EventHandler(this.button1_Click);
09             //……
10         }
11         //……
12         private System.Windows.Forms.Button button1;
13     }
```

第 08 句很清晰地说明了按钮的单击事件其实是一个 System.EventHandler 类型的委托。

EventHandler 委托是.NET Framework 预定义的众多事件委托之一，查询 Visual Studio 文档，可以看到以下定义：

```
    public delegate void EventHandler (Object sender,  EventArgs e )
```

这是一个通用的事件委托声明，被用在许多地方（比如鼠标的单击事件）。

但要注意，**不同的事件拥有不同的委托声明**，比如 MouseMove 事件所对应的委托就不是 EventHandler 类型的。

一个对象可以激发多个事件，在 Visual Studio 的属性窗口中激活“事件”面板，可以看到指定对象可激发的事件列表（图 20）：

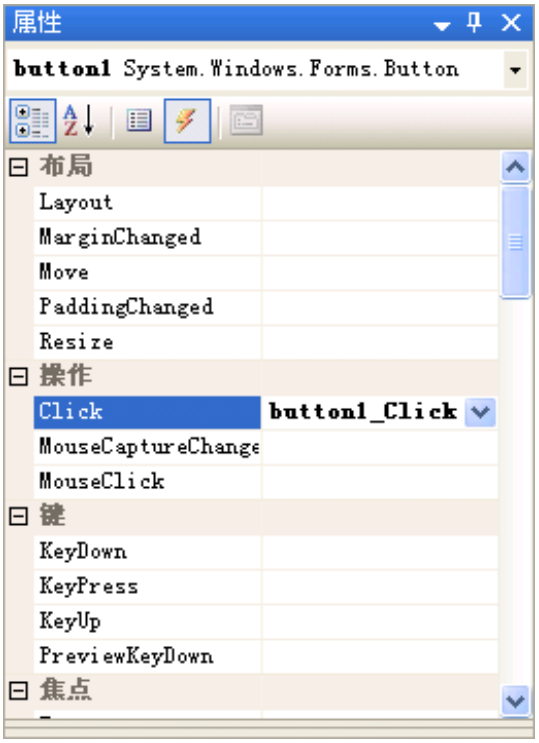

图 20 按钮对象可激发的事件

图 20 所示为按钮对象可激发的事件清单，已写好事件处理程序的事件名（图中为 Click 事件）后头跟着的就是事件处理函数名（图中为 button1_Click），在图中选定的事件中双击，Visual Studio 会自动生成相应的事件响应函数框架，同样地会在“窗体名.Desinger.cs”文件中增加一行组合委托的语句。

例如，在图 20 中双击 KeyDown 这一行，将会在 Form1.cs 文件中生成一个事件响应函数框架：

```
    private void button1_KeyDown(object sender, KeyEventArgs e)
    {
    }
```

并在“Form1.Designer.cs”文件中增加一行：

```
this.button1.KeyDown += new
          System.Windows.Forms.KeyEventHandler(this.button1_KeyDown);
```

请注意，KeyEventHandler 是.NET Framework 又一个预定义的事件委托之一，专用于响应键盘事件：

```
public event KeyEventHandler KeyDown;
```

KeyEventHandler 的声明如下：

```
public delegate void KeyEventHandler ( Object sender,  KeyEventArgs e)
```

对于键盘事件，程序员往往需要知道是具体按键值，这些信息被封装到了事件参数 KeyEventArgs 中。比如想检测用户在点击按钮 button1 时是否压住了 CTRL 键，此事件响应函数 button1_KeyDown()可这样写：

```
private void button1_KeyDown(object sender, KeyEventArgs e)
{
    if ((e.Control == true) && (e.KeyCode == Keys.Enter))
        MessageBox.Show("您按下了Ctrl_Enter键");
}
```

可以看到，用户按键的信息全都是由 KeyEventArgs 类型的事件参数 e 所提供的，而 KeyEventArgs 又派生至 EventArgs 类。

```
public class KeyEventArgs : EventArgs
```

由此我们可以明白 Visual Studio 中对可视化窗体控件的事件处理机理：

所有的.NET Framework 可视化窗体控件的预定义事件，都是某一对应的“事件名＋Handler”委托类型的变量。与此事件相关的信息都封装在“事件名＋Args”类型的事件参数中，此事件参数有一个基类 EventArgs，它是所有事件参数的基类。

明了上述内部机理，对于我们在程序中定义自己的事件非常有好处，尤其是开发一个自定义的可视化控件时，如果需要增加新的事件类型，我们应尽量遵循.NET Framework 的定义事件的框架，给事件取一个名字，定义一个“事件名＋Handler”的事件委托类型，再从 EventArgs 派生出自定义事件的参数，取名为“事件名＋Args”。

8.3 事件小结

面向对象的软件系统有许多都是事件驱动的，比如 ASP.NET 就采用了“事件驱动”的编程方式。

所谓“事件驱动”的开发方式，就是指整个系统包含许多的对象，这些对象可以引发多种事件，软件工程师的主要开发工作就是针对特定的事件书写代码响应它们。

.NET 事件处理机制建立在委托的基础之上，而这两者都是 ASP.NET 技术的基础之一。因此，必须牢固地掌握好委托和事件这两种编程技术，才能为掌握 ASP.NET 技术扫清障碍。