**《.net编程先锋C#》第一章 C#简介**

欢迎您加入C#的世界！ 这一章将把您引进C#的天地，并回答一些相关的问题，如：您为什么要使用C#，C++和C#的主要有什么不同点，以及为什么C#使开发更容易而且还使您感到很有趣。

为什么是另外一种编程语言?

　 必须回答的一个问题：当您已经使用C++或VB从事企业开发时，为什么还要学习另一种语言? 市场式的回答就是："在企业计算领域，C#将会变成为用于编写"下一代窗口服务"(Next Generation Windows Services,简写为NGWS )应用程序的主要语言。" 这一章将对用参数请求提供支持，并陈列了C#的一些功能。这一章会使您开胃的。

　 C#语言自C/C++演变而来。但是，它现代、简单、完全面向对象和类型安全。如果您是C/C++程序员，学习曲线将会很平坦。许多C#语句直接借用您所喜爱的语言，包括表达式和操作符。假如不仔细看，简直会把它当成C++。

　 关于C#最重要的一点：它是现代的编程语言。它简化和现代化了C++在类、名字空间、方法重载和异常处理等领域。屏弃了C++的复杂性，使它更易用、更少出错。

　 对C#的易用有贡献的是减少了C++的一些特性，不再有宏、模板和多重继承。特别对企业开发者来说，上述功能只会产生更多的麻烦而不是效益。

　 使编程更方便的新功能是严格的类型安全、版本控制、垃圾收集(garbage collect)等等。所有的这些功能的目标都是瞄准了开发面向组件的软件。

　 在继续呈现出更多的功能之前，我想停下来并在下面说明C#至关重要的各种要素。

简单 现代 面向对象 类型安全 版本控制 兼容 灵活

简单

　 C#具有C++所没有的一 个优势就是学习简单。该语言首要的目标就是简单。很多功能(还不如说是缺少了C++的一些功能)有助于C#全方位的简单。

　 在C#中，没有C++中流行的指针。默认地，您工作在受管理的代码中，在那里不允许如直接存取内存等不安全的操作。我想没有C++程序员可以声称，从没有使用指针访问过不属于他们的内存。

　 与指针"戏剧性"密切相关的是"愚蠢的"操作。在C++中，有::、.、和->操作符，它们用于名字空间、成员和引用。对于新手来说，操作符至今仍是学习的一道难关。C#弃用其它操作符，仅使用单个操作符 "."。现在一个程序员所需要理解的就是嵌套名字的注解了。

　 您不必记住基于不同处理器架构的隐含的类型，甚至各种整型的变化范围。C#使用统一的类型系统，屏弃了C++多变的类型系统。这种系统允许您把各种类型作为一个对象查看，它是一个原始类型还是一个full-blown 类。和其它编程语言相比，由于加框(boxing)和消框(unboxing)的机制，把简单类型当作对象处理并不能获得性能的改善。稍后将详细解释加框和消框，但基本上仅当需要时才使用对象访问简单类型这种技术。

　首先，老练的程序员可能不喜欢它，但是整型和布尔型如今终归是两种完全不同的数据类型。这就意味着原来if语句中错误的赋值现在会被编译出错，因为if语句只接受布尔类型的值。再也不会出现误用赋值符为比较符这样的错误！

　C#同时也解决了存在于C++中已经有些年头的多余东西(redundancies)。这种多余包括常数预定义，不同字符类型等。鉴于多余表单已经从该语言中消失，故一般在C#中都可以使用表单了。

现代

　 您投入学习C#的努力是一笔大投资，因为C#是为编写NGWS 应用程序的主要语言而设计。您 将会发现很多自己用C++可以实现或者很费力实现的功能，在C#中不过是一部分基本的功能而已。

　 对于企业级的编程语言来说，新增的金融数据类型很受欢迎。您用到了一种新的十进制数据类型，它专用于金融计算方面。如果不喜欢这种现成简单的类型，根据您应用程序的特殊需求，可以很容易地创建出新的一种数据类型。

　 我已经提到，指针不再是您编程武器的一部分。不要太惊讶，全面的内存管理已经不是您的任务。运行时NGWS提供了一个垃圾收集器，负责C#程序中的内存管理。因内存和应用程序都受到管理，所以很必要增强类型安全，以确保应用的稳定性。

　 对于C++程序员，异常处理的确不是新的东西，但它是C#的主要功能。C#的异常处理与C++的不同点在于它是交叉语言的(运行时的另一个功能)。在没有C#之前，您必须处理怪异的HRESULTs，但现在由于使用了基于异常的健壮的出错处理， 这一切都 结束了。

　对于现代的应用程序，安全是首要的，C#也不会例外。它提供了元数据语法，用于声明下述NGWS安全模式的能力和许可。元数据是NGWS运行时的一个关键的概念，下一章将涉及到它更深的含义。

面向对象

　 您不会预料一种新语言不支持面向对象的功能吧? C#当然支持所有关键的面向对象的概念，如封装、继承和多态性。完整的C#类模式构建在NGWS运行时的虚拟对象系统(VOS，Virtual Object System)的上层，VOS将在下章描述。对象模式只是基础的一部分，不再是编程语言的一部分。

　 您一开始必须关注的事，就是不再有全局函数、变量或者是常量。所有的东西都封装在类中，包括事例成员(通过类的事例--对象可以访问)或静态成员(通过数据类型)。这些使C#代码更加易读且有助于减少潜在的命名冲突。

　 定义类中的方法默认是非虚拟的(它们不能被派生类改写)。主要论点是，这样会消除由于偶尔改写方法而导致另外一些原码出错。要改写方法，必须具有显式的虚拟标志。 这种行为不但缩减了虚拟函数表，而且还确保正确版本的控制。

　 使用C++编写类，您可以使用访问权限(access modifiers) 给类成员设置不同的访问等级。C#同样支持private、protected 和public 三种访问权限 ，而且还增加了第四种：internal。有关访问权限 的详细情况将在第五章 "类" 中说明。

　 您曾经创建了多少个类是从多基类派生出来的(ATL 程序员，您的投票不计在内！) ? 大多数情况，仅需从一个类派生出。多基类惹出的麻烦通常比它们解决的问题还多。那就是为什么C#仅允许一个基类。如果您觉得需要多重继承，可以运用接口。

　 一个可能出现的问题：在C#中不存在指针，如何模仿它? 这个问题的答案很有代表性，它提供了对NGWS运行时事件模式的支持。再次，我将把对它的全面解释放到第五章。

类型安全

　 我再次选指针作为一个例子。在C++中拥有一个指针，您能自由地把它强制转换成为任何类型，包括干出诸如把一个int\*(整型指针)强制转换成一个double \*(双精度指针)这样的傻事。只要内存支持这种操作，它就"干过"。这并不是您所想象的企业级编程语言的类型安全。

　 纲要性的问题，C#实施最严格的类型安全，以保护自己及垃圾收集器(garbage collector)。所以必须遵守C#中一些相关变量的规则：

　 您不能使用没有初始化的变量。对于对象的成员变量，编译器负责清零。而局部变量，则由您负责清零。当您使用一个没有初始化的变量时，编译器会教您怎么做。优点是能够避免由于使用不经初始化的变量计算结果而导致的错误，而您还不知道这些奇怪的结果是如何产生的。

　 C#取消了不安全的类型转换。不能把一个整型强制转换成一个引用类型(如对象)，而当向下转换时，C#验证这种转换是正确的。(也就是说，派生类真的是从向下转换的那个类派生出来的。)

　 边界检查是C#的一部分。再也不会出现这种情况：当数组实际只定义了n-1个元素，却超额地使用了n个元素。

　 算术运算有可能溢出终值数据类型的范围。C#允许在语句级或应用程序级检测这些运算。在允许检测溢出的情况下，当溢出发生时将会抛出一个异常。

　 在C#中，被传递的引用参数是类型安全的。

版本可控(Versionable)

　 在过去的几年中，几乎所有的程序员都至少有一次不得不涉及到众所周知的"DLL地狱"。该问题起因于多个应用程序都安装了相同DLL名字的不同版本。有时，老版本的应用程序可以很好地和新版本的DLL一起工作，但是更多的时候它们会中断运行。现在的版本问题真是令人头痛。

　 就象您将在第八章"用C#写组件"所看到的，NGWS runtime 将对您所写的应用程序提供版本支持。C#可以最好地支持版本控制。尽管C#不能确保正确的版本控制，但是它可以为程序员保证版本控制成为可能。有这种支持，一个开发人员就可以确保当他的类库升级时，仍保留着对已存在的客户应用程序的二进制兼容。

兼容

　 C#并没有存在于一个封闭的世界中。它允许使用最先进的NGWS的通用语言规定(Common Language Specification，简写为CLS)访问不同的API。CLS规定了一个标准，用于符合这种标准的语言的内部之间的操作。为了加强CLS的编译，C#编译器检测所有的公共出口编译，并在通不过时列出错误。

　 当然，您也想能够访问旧一点的COM对象。NGWS运行时提供对COM透明的访问。如何集成原来的代码将在第10章"非管理代码的内部操作"有介绍。

　 OLE 自动化是一种特殊的动物。任一个使用C++创建OLE自动化项目的人已经喜欢上各种各样的自动化数据类型。有个好消息就是C#支持它们，而没有烦锁的细节。

　 最后，C#允许您 用C 原型的API进持内部操作。可以从您的应用程序访问任何DLL中的入口点(有C的原型)。用于访问原始API的功能称作平台调用服务(Plaform Invocation Services ，缩写PInovke)，第10章将展示使用C API进行内部操作的一些例子。

灵活

　 上一部分的最后一段有可能提醒了程序员。您可能会问："难道就没有我要传递指针的API吗?" 您是正确的。不是仅有少数的这种API，而是很多(有点保守的估计)。这种对原始WIN32代码的访问有时导致对非安全类指定指针的使用(尽管它们中的一些由于受COM和PInvoke的支持可以解决)。

　 尽管C#代码的缺省状态是类型安全的，但是您可以声明一些类或者仅声明类的的方法是非安全类型的。这样的声明允许您使用指针、结构，静态地分配数组。安全码和非安全码都运行在同一个管理空间，这样暗示着当从安全码调用非安全码时不会陷入列集(marshaling)。

小结

　 C#语言从C和C++演变而来，它是给那些愿意牺牲C++一点底层功能，以获得更方便和更产品化的企业开发人员而创造的。C#现代、简单、面向对象和类型安全。尽管它借鉴了C和C++的许多东西，但是在一些诸如名字空间、类、方法和异常处理等特定领域，它们之间还存在着巨大的差异。

　 C#为您提供了方便的功能，如垃圾收集、类型安全、版本控制，等等。仅有的"代价"就是，代码操作默认是类型安全的，不允许指针。光是类型安全就可以搞定了。但是，如果您需要指针，仍可以通过非安全码使用它们，而且当调用非安全码时，不能含有列集。

## 第二章 理论基础－公用语言 运行环境

既然你已经具有了C#全面的印象，我也想让你了解NGWS runtime的全貌。C#依靠由NGWS提供的运行时；因此，有必要知道运行时如何工作，以及它背后所蕴含的概念。

所以，这一章分为两部分——它们是所有的概念和使用的基础。两部分的内容虽然有些重叠，但它有助于加深理解正在学习的概念。

2.1 NGWS Runtime

NGWS和NGWS Runtime为你提供了一种运行时环境。该运行时管理执行代码，并提供了使编程更容易的服务。只要你的编译器支持这种运行时，你就会从这种受管理的执行环境中得益。

你猜测C#编译器支持NGWS runtime很正确，但是不仅它支持NGWS runtime，VB和C++也支持。这些为支持运行时所创建的代码称作"受管代码"(managed code)。以下是你的应用程序从NGWS runtime那里所得到的利益：交叉语言集成(通过通用语言规范)、自动内存管理(垃圾收集)、交叉语言异常处理(统一展开)、增强安全(包括类型安全)、版本支持("DLL地狱"终结者)、组件交互简化模式。

因NGWS runtime 要提供所有的这些好处，编译器必须把元文件和受管代码一起发出。元文件描述代码中的类型，它和你的代码存在一起(与PE类似---PE为可变位执行文件)

正如你从很多种交叉语言功能所看到的，NGWS runtime主要是关于高度集成交叉多异编程语言(tight integration across multiple different programming languages)。这种支持可达到允许你从一个VB对象派生出一个C#类的程度(我后面会给出要讨论的文章)。

C#程序员将会喜欢的一个功能是，他们不必担心内存管理—也就是说不必担心臭名昭著的内存泄漏。NGWS runtime提供了内存管理，当对象和变量的生命期结束(不再被引用)时，垃圾收集器释放它们。我真的喜欢这个功能，因为在COM中的内存管理一直是我的一块心病。

应该鼓励配置一个管理应用程序或者组件。因为管理应用程序含有元数据文件，NGWS runtime可以利用这些信息，以确保你的应用程序具有它所需的各种规定版本。所产生的明显效果为，由于你的代码没有相互之间的依赖，很少可能出现中断。

这章余下来的将分为两部分，每一部分讨论NGWS runtime的各个方面，直到你的C#应用程序能执行为止。

1、中间语言(Intermediate Language，缩写IL)和元数据2、即时编译器(just-in-time compliers,简称JITers)

2.1.1 中间语言和元数据

由C#编译器生成的受管代码并不是原始代码，但它是中间语言(IL)代码。这种IL代码自身变成了NGWS runtime的受管执行进程的入口。IL代码明显的优势在于它是CPU无关的，这也意味着，你要用目标机器上的一个编译器才能把IL代码转换成原始代码。

尽管IL代码由编译器产生，但它并不是编译器提供给运行时仅有的东西。编译器同样产生有关你代码的元数据，它告诉运行时有关你代码的更多的东西，例如各种类型的定义、各种类型成员的签名以及其它数据。基本上，元数据是类型库、注册表内容和其它用于COM（组件对象模型）的信息。尽管如此，元数据还是直接和执行代码合并在一起，并不处在隔离的位置。

IL和元数据存放于扩展了PE格式的文件中(PE格式用于.exe和.dll文件)。当这样的一个PE文件被装载时，运行时从文件中定位和分离出元数据和IL。

在进一步说明之前，我想给你已有的IL指令的简短目录。尽管它不是一个完整的清单，也不需要你熟记和理解，但是它列出了你所必需的、C#程序所基于的知识基础。

算术和逻辑操作符、控制流、直接内存访问、堆栈操作、参数和局部变量、堆栈分配、对象模式、实例类型值、临界区、数组、分型位置

即时编译器(JITters)

2.1.2 即时编译器(JITters)

由C#或其它能产生受管代码的编译器所生成的受管代码就是IL码。虽然IL代码被包装在一个有效的PE文件中，但是你还是不能执行它，除非它被转换成为受管原始代码。这就是NGWS runtime 即时编译器(也称作JITters)大显身手的时候。

为什么你会对即时编译代码感到厌繁， 为什么不把整个IL PE文件编译成原始代码? 答案是时间——需要把IL代码编译成CPU规格的代码的时间。这种编译将更加有效率，因为一些程序段从来就没有被执行过。例如，在我的字处理器中，邮件合并功能从来就没有被编译。

从技术上说，全部的处理过程如下：当一个类型被装载时，装载器创建一个存根(stub)，并使它连接每一个类型的方法。当一个方法第一次被调用时，存根把控制交给JIT。JIT把IL编译为原始代码，且把存根指针指向缓冲了的原始代码。接着的调用将执行原始码。在某些位置上(At some point)，所有的IL都被转换成为原始代码，而JITter处于空闲状态。

正如我在前面提到的，JIT编译器有很多，不止一个。在Windows平台上，NGWS runtime装有3个不同的JIT编译器。

JIT——这是NGWS runtime默认使用的JIT编译器。它是一个后台(back end)优化的编译器 ，在前台(up front)实行数据流分析，并创建了高度优化的受管原始代码做为输出结果。JIT可以使用不严格的IL指令集编码，但是所需资源将十分可观。主要的限制在于内存足迹(footprint)、结果工作集，以及实行优化所消耗的时间。

EconoJIT—— 和主JIT相比，EconJIT的目标是把IL高速地转换成受管原始代码。它允许缓冲所产生的原始代码，但是输出码并不象主JIT生成的代码那样优化(代码小)。当内存紧张时，快速代码生成方案的优势将荡然无存。通过永久地抛弃无用的已JIT过的代码，你可以把更大的IL程序装入代码缓冲区。因为JIT编译快，执行速度也仍然很快。

PreJIT—尽管它是基于主JIT的，但操作起来更象是一个传 统的编译器。你安装了NGWS组件，它才能运行，才可以把IL代码编译成受管原始代码。当然最终的结果为，更快的装载时间和更快的应用程序启动时间(不需要更多的JIT编译)。

在所列出的JITters中，有两个是运行时的JITters。可是你怎么决定要使用哪一个JIT，它如何使用内存? 有一个称做"JIT编译管理器"的小应用程序(jitman.exe)，它存放于NGWS SDK安装目录下的bin目录中。当执行该程序时，它把一个图标加到系统任务条上，双击该图标打开程序对话框(见图2.1)。

图2.1 JIT编译管理器允许你设置各种相关性能的选项

尽管它是一个小小的对话框，可是你所选择的选项功能是相当强大的。每一个选项将在以下描述。

Use EconoJIT only 选项——当该复选框没有选上时，NGWS runtime使用默认的正常的JIT编译器。前面就曾经解释过两种JITter的区别。

Max Code Pitch Overhead(%)选项——该设置仅保留给EconoJIT。它控制了JIT编译时间和执行代码时间的百分比。如果超过了设定的域值，代码缓冲区得到扩充，以缩短JIT编译所消耗的时间。

Limit Size of Code Cache选项——该项默认为非选。没有选择该项意味着缓冲区将使用它所能得到的内存。如果你想限制缓冲区大小，复选该选项，这将允许你使用Max Size of Cache(bytes)选项。

Max Size of Cache(bytes)选项—控制容纳JIT代码的缓冲区的最大值。虽然你可以非常严格地限制这个值，但你还是应该小心，不能超过这个缓冲区所适合的最大值。否则该方法的JIT编译将会失败。

Optimize For Size选项——告诉JIT 编译器，优化的目的是为了使代码更小而不是能执行得更快。这个设置默认是关掉的。

Enable Concurrent GC[garbage collection]选 项——垃圾收集(GC)默认地运行在用户代码的线程中。意味GC发生时，可能会注意到回应有轻微的延迟。为防止出现该现象，打开当前GC。注意，当前GC比标准GC更慢，它仅在windows 2000上写时(the time of writing)有效。

当用C#创建项目时，你可能使用不同的设置试验过。当创建 UI-intensive应用程序时，你将会看到允许当前GC的最大差别。

2.2 虚拟对象系统(VOS)

到目前为止，你仅看到了NGWS runtime如何工作，但是并不了解它工作的技术背景以及为什么它要这样工作。这节都是关于 NGWS 虚拟对象系统的(VOS)。

以下为在VOS中形成声明、使用和管理类型模型时，NGWS runtime的规则。在VOS背后的思想是建立一个框架，在执行代码时不能牺牲性能，允许交叉语言集成和类型安全。

我提到的框架是运行时架构的基础。为了帮助你更好地了解它，我将它勾出四个区域。当开发C#应用程序和组件时，理解它们很重要。

VOS类型系统——提供丰富的类型系统，它打算支持全面编程语言的完全实施。

元数据——描述和引用VOS类型系统所定义的类型。元数据的永久格式与编程语言无关，但是，元数据拿自己当作一种互换机制(nterchange mechanism)来使用，这种互换是在在工具和NGWS的虚拟执行系统之间。

通用语言规范(CLS)——CLS定义了VOS中类型的子集，也定义了常规的用法。如果一个类库遵守CLS的规则，它确保类库可以在其它所有能实现CLS的编程语言上使用。

虚拟执行系统(VES)——这是VOS实时的实现。VES负责装入和执行为NGWS运得时编写的程序。

这四个部分一起组成了NGWS runtime架构。每一部分在下面小节中描述。

2.2.1 VOS类型系统

VOS类型系统提供丰富的类型系统，它打算支持多种编程语言的完全实施。所以，VOS必须都支持面向对象的语言和过程编程语言。

现在，存在着很多种近似但有点不兼容的类型。就拿整型当例子，在VB中，它是16位长，而在C++中，它是32位。还有更多的例子，特别是用在日期和时间以及数据库方面的数据类型。这种不兼容使应用程序的创建和维护不必要地复杂化，尤其当程序使用了多种编程语言时。

另一个问题是，因为编程语言之间存在着一些差别，你不能在一种语言中重用另一种语言创建的类型。(COM用二进制标准接口部分地解决了这个问题)。 当今代码重用肯定是有限的。

发布应用程序的最大障碍是各种编程语言的对象模型不统一。几乎每一方面都存在着差异：事件、属性、永久保存(persistence)等等。

VOS这里将改变 这种现象 。VOS定义了描述值的类型，并规定了类型的所有值所必须支持的一条合约。由于前面提到的支持面向对象和过程编程语言，就存在着两种值和对象。

对于值，类型存储于表述(representation)中，同样操作也在其中实行。对象更强大因为它显式地存于表述中。每一个对象都有一个区别于其它对象的识别号。支持不同的VOS类型在第四章 "C#类型"中提出。

2.2.2元数据

尽管元数据用于描述和引用由VOS类型系统定义的类型，但它还不能锁定到这个单个目标。当你写一个程序时，通过利用类型声明，你所声明的类型(假定它们是数值类型或引用类型)被介绍给NGWS runtime类型系统。类型声明在存于PE可执行文件内部的元数据中得到描述。

基本上，元数据用于各项任务：用于表示NGWS runtime用途的信息，如定位和装载类、 内存中这些类的事例、解决调用 、翻译IL为原始码、加强安全并设置运行时上下文边界。

你不必关心元数据的生成。元数据是由C#的"代码转IL编译器"(code-to-IL compiler,不是JIT编译器)生成的。代码转IL编译器发送二进制元数据信息给PE文件，是以标准的方式发送的，不象C++编译器那样，为出口函数创建它们自己的修饰名字。

你从元数据和可执行代码并存所获得的主要优势为，有关类型的信息同类型自身固定在一起，不会遍布很多地方。同样有助于解决存在于COM中的版本问题。进一步地，你可以在相同的上下文中使用不同的版本库，因为库不仅被注册表引用，也被包含在可执行代码中的元数据引用。

2.2.3通用语言规范

通用语言规范(CLS)并不是虚拟对象系统(VOS)真正的一部分，它是特殊的。CLS定义了VOS中的一个类型子集，也定义了必须符合CLS的常规用法。

那么，对此有什么迷惑呢?如果一个类库遵守CLS规则，其它编程语言同样也遵守CLS规则，那么其它编程语言的客户也可以使用类库。CLS是关于语言的交互可操作性(interoperability)。因此，常规用法必须仅遵循外部可访问项目 (externally visible items)如方法、属性和事件等等。

我所描述的优点是你可以做以下工作。用C#写一个组件，在VB中派生它，因加在VB中的功能是如此之强大，在C#中再次从VB类派生它。只要所有的外部可访问项遵守CLS规则，这样是可行的。

我在这本书中出示的代码不关心CLS协定。但在构建你的类库时要注意到CLS协定。我提供了表2.1，用以给类型和外部可访问项定义协定规则。

这个清单不完整。它仅包含一些很重要的项目。我不指出出现在本书中每一种类型的CLS协定，所以有个好主意：当你寻找CLS协定时，至少应该用浏览该表，以了解哪种功能有效。不要担心你不熟悉这章表中的每一个含义，在这本书中你会学到它们。

表2.1 通能语言规范中的类型和功能

Bool、char、byte、short、int、long、float、double、string、object(所有对象之母)、Arrays(数组)

数组的维数必须是已知的(>=1)，而且最小下标数必须为0。

要素类型必须是一个CLS类型。

类型(Types)

可以被抽象或隐藏。

零或更多的接口可以被实现。不同的接口允许拥有具有相同名字和签名的方法。

一个类型可以准确地从一个类型派生。允许成员被覆盖和被隐藏。

可以有零或更多的成员，它们是字段(fields)、方法、事件或者类型。

类型可以拥有零或更多个构造函数。

一种类型的可访问性可以是公共的或者对NGWS组件来说是局部的；但是，仅公共成员可以认为是类型接口的一部分。

所有的值型必须从系统值型继承。异常是一个枚举——它必须从系统枚举(System Enum)继承。

类型成员

类型成员允许隐藏或者覆盖另一种类型中的其它成员。

参数和返回值的类型都必须是 CLS 协定 类型。

构造函数、方法和属性可以被重载。

一个类型可以有抽象成员，但仅当类型不被封装时。

方法

一种方法可以是静态、虚拟或者实例。

虚拟和实例方法可以是抽象的，或者是一个实现。静态方法必须总拥有一个实现。

虚拟方法可能是最后的(或者不是)。

字段(Fields)

可以是静态或者是非静态。

静态字段可以被描述或只初始化。

属性

当获取和设置方法而不是使用属性语法时，属性可以公开。

获取的返回类型和设置方法的第一个参数必须是相同的CLS类型——属性的类型。

属性名字必须不同，不同的属性类型用于区分是不充分的。

由于使用方法实现属性访问，如果 PropertyName 是同一个类中定义的一个属性，你不能实现命名为 get\_PropertyName 和 set\_PropertyName 的方法。

属性可以被索引。

属性访问必须遵循这种命名格式：get\_ProName，set\_PropName。

枚举(Enumerations)

强调类型必须是byte、short、int 或long。

每一个成员是一个枚举类型的静态描述字段。

一个枚举不能实现任何接口。

你允许给多字段设定相同的值。

一个枚举必须继承系统枚举(隐含在C#中)

异常

可以被引发和被捕获。

自定义异常必须继承系统异常。

接口

可需要实现其它接口。

一个接口可以定义属性、事件和虚拟方法。实现取决于派生类。

事件

增加和取消方法必须是都提供或者都没有 ，每一种方法采用一个参数，它是一个从系统代表元(System Delegate)派生下来的类。

自定义属性

可以仅使用下更类型：Type(类型),char, char, bool, byte, short, int, long, float, double, enum (一种CLS 类型), and object.

代表元(Delegates)

可以被创建和被激活

标识符(Identifiers)

一个标识符的第一个字母必须来自一限制集。

通过大小写在单一范围内，不可能唯一地区别两个或更多个标识符(大小写不敏感)。

2.2.4虚拟执行系统(VES)

虚拟执行系统实现了虚拟对象系统。通过实现一个负责NGWS runtime的执行引擎(execution engine，缩写EE)创建VES。这个执行引擎执行你用C#编写和编译的应用程序。

下列组件为VES的一部分。

1、中间语言(IL)——被设计为很容易受各种各样的编译器所兼容 。在该框架之外，C++、VB和C#编译器都能够生成IL。

2、装入受管代码——这包括解决内存中的名字、 表层类(laying out classes )，并且创建JIT编译所必需的存根。通过执行经常性校验，包括加强一些访问规则，类装载器同样也增强了安全性。

3、用JIT转换IL成原始代码——IL代码并不是设计成为一种传统的解释字节代码或树型代码，IL转换是真正的编译。

4、装入元数据、校验类型安全和方法的完整性

5、垃圾收集(GC)和异常处理——两者都是基于堆栈格式的服务。受管代码允许你动态地跟踪堆栈。要动态地识别各个堆栈框架，JITter或其它编译器必须提供一个代码管理器。

6、描绘和查错服务——两者都取决于由源语言编译器所生成的信息。必须发出两个映射：一个映射从源语言结构发到指令流中的地址，一个映射从地址发到堆栈框架中的位置。当执行从IL到原始代码的转换时，这些映射被重新计算。

7、管理线程和上下文，还有远程管理——VES为受管代码提供这些服务。

虽然这个清单并不完整，但它足以让你理解运行时基于的由VES提供的低层架构。肯定将会有专门讨论运行时的书，而这本书将稍为深入地挖掘各种话题。

2.3 小结

这一章，我带你逛了一回运行时的世界。我描述了当创建、编译和配置C#程序时它是如何工作的。你学会了中间语言(IL)，还有元数据是如何用于描述被编译为IL的类型。元数据和IL都用于JITter检测和执行你的代码。你甚至可以选择用哪一种JITter来执行应用程序。

在这一章的第二部分，涉及到了运行时为何按这种方式工作的理论。你学了虚拟对象系统(VOS)和组成它的那部分。对于类库设计者最为感兴趣的就是通用语言规范(CLS)，它为基于VOS的语言交互操作设定规则。最后，你看到了虚拟执行系统(VES)如何通过NGWS runtime实现VOS。

## 第三章 第一个C#应用程序

3.0 选择一个编辑器

尽管我是一个顽固的Notepad狂，但这次我不建议用它编辑源码。原因是你正在与真正的编程语言打交道，使用Notepad编辑源码编译时可能产生大量的错误信息行(C++程序员知道我在说什么。)

你有几种选择。可以重新配置你信任的老式Visual C++ 6.0，使它能够和C#源文件一起工作。第二种选择是使用新的Visual Studio 7。第三，你可以用任何第三方程序编辑器，最好要支持行数、色彩编码、工具集成和良好的搜索功能。CodeWright就是其中一个例子，如图3.1所示。

图3.1 CodeWright 是你可以用于创建C#代码文件众多可能编辑器中的一个。

当然，在所提到的编辑器中，没有一个对创建C#程序来说是必要的。用Notepad肯定可以编辑。但是，如果你考虑到要编写更大的项目，最好还是忍痛割爱吧。

3.1 "Hello World" 代码

讨论编辑器有点离题 ，让我们把话题转回到一个非常出名的小应用程序。这个最短的C#版本应用程序见清单3.1。把它存起来，文件名为 helloworld.cs，以便使你能按照说明，完成诸如编译应用程序等其它余下来的步骤。

清单 3.1 最简单的 "Hello World "程序

1: class HelloWorld

2: {

3: public static void Main()

4: {

5: System.Console.WriteLine("Hello World");

6: }

7: }

在C#中，代码块(语句组)由大括弧({和})所括住。所以，甚至你以前没有C++的经验，你也可以说出Main()方法就是HelloWorld 类语句的一部分，因为类被括在所定义的大括弧中。

C#应用程序(可执行)的入口点就是 static Main 方法，它必须包含在一个类中。仅有一个类能使用该标志定义，除非你告诉编译器它应使用哪一个 Main 方法(否侧，会产生一个编译错误)。

和C++相比，Main的第一个字母是大写的M，而不是你曾经使用过的小写字母。在这个方法中，你的程序开始并结束。方法中可以调用其它方法——如这个例子中，用于输出文本——或者创建对象并激活该方法。

正如你所看到的，Main方法返回一个void类型。

public static void Main()

尽管看到这些语句时，C++程序员肯定会觉得似曾相识，但是其他程序员并不如此。首先，public 的访问标志告诉我们这个方法可以被任何程序访问，这是它被调用的必要条件。其次，static 意味着没有先创建类的实例也可以调用方法——你所要做的就是用类名调用方法。

HelloWorld.Main();

但是，我不赞成在Main方法中执行这行代码，递归会导致堆栈溢出。另一重要的方面是返回类型。对于方法Main，可选择void (意味着根本就没有返回值)，或用int 为整型结果(应用程序返回的错误级别)。因此，两种可能的Main方法为：

public static void Main()

public static int Main()

C++程序员会同样知道后面我要提到的——可以传给应用程序的命令行参数数组。如：

public static void Main(string[] args)

我现在并不想详细地说明如何访问参数，但我想事先给C++程序员一个警告：和C++相比，应用程序路径不是这个数组的一部分。仅仅那些参数包含在这个数组中。

在对Main方法并不简短的介绍之后，让我们把注意力集中到唯一真正的代码行——这行代码在屏幕上显示"Hello Wold"。

System.Console.WriteLine("Hello World");

假如不是由于有了System，大家会马上猜到WriteLine是Console 对象的一个静态方法。那么System代表什么呢? 它是包含Console对象的名字空间(范围)，实际上并不是每次都在Console对象前加上名字空间的前缀，你可以象清单3.2所示范的那样，在应用程序中引入名字空间。

清单3.2 在应用程序中引入名字空间

1: using System;

2: 3: class HelloWorld

4: {

5: public static void Main()

6: {

7: Console.WriteLine("Hello World");

8: }

9: }

所有你要做的就是给System名字空间加一个using指令。在这之后，不再需要规定名字空间，就可以使用它们的方法和属性了。NGWS 框架体系中有很多的名字空间，我只对巨大的名字空间池中的少数几个对象进行探讨。但在第八章 "用C#写组件"将介绍为你的对象创建自己的名字空间。

3.2 编译应用程序

由于NGWS Runtime支持所有的编译器(VB、C++和C#)，你不必买一个单独的开发工具用来把应用程序编译成IL(中间语言)。但是，如果你从没有用过命令行编译器编译过应用程序(仅懂得编译名，而没有熟记)， 它还是你的首要选择。打开命令提示符并切换到存 helloworld.cs 的目录。敲入以下命令：

csc helloworld.cs

helloworld.cs 被编译并链接成hellworld.exe。因为源码没有错误(那当然！)，C#编译器没有出错提示，在整个编译过程没有丝毫停顿。如图3.2所示。

图3.2 使用命令行编译器 csc.exe 编译应用程序

现在你已经准备好运行第一个真正用C#编写的应用程序。简单地在命令行上敲入helloworld，输出结果为 "Hello World"。

在继续往下介绍之前， 我想稍为想象一下第一个应用程序和一个编译器开关的使用：

csc /out:hello.exe helloworld.cs

这个开关告诉编译器输出文件命名为hello.exe。虽然这不是什么绝招，但它是这本书中用到的未来编译器的基本功。

3.3 输入和输出

到目前为止，我仅仅演示了把简单的常量字符串输出到屏幕。尽管这本书只介绍了C#编程的概念而不介绍用户接口编程，但我需要让你迅速学会简单的屏幕输入和输出方法——相应于C的scanf 和 printf，或者C++的cin 和cout。我不能提供VB相应的函数，因为屏幕访问不是该核心语言的一部分。

你只需要能够读用户的输入并提示一些信息给用户。清单3.3 说明如何读一个用户请求的名字输入，并显示一条已定制好的"Hello" 信息。

Listing 3.3 从控制台读输入信息

1: using System;

2:

3: class InputOutput

4: {

5: public static void Main()

6: {

7: Console.Write("Please enter your name: ");

8: string strName = Console.ReadLine();

9: Console.WriteLine("Hello " + strName);

10: }

11: }

第7行使用Console对象的一个新方法用于提示文本信息给用户，它就是Write方法。它与WriteLine不同的地方在于它输出时不换行。我使用这种方法以便用户可以在信息提示的同一行输入名字。

在用户输入他的名字后(并按回车键)，ReadLine 方法读入了一个字符串变量。名字字符串连接到常量字符串"Hello"，并用我们早已熟悉的WriteLine方法显示出来(见图3.2)。

图3.3 编译和运行定制的Hello 应用程序

你几乎已学完了NGWS框架必要的输入和输出功能。但是，你还需要为用户显示多个值。为用户写一个格式串。清单3.4展示一个例子。

清单 3.4 使用不同的输出方法

1: using System;

2:

3: class InputOutput

4: {

5: public static void Main()

6: {

7: Console.Write("Please enter your name: ");

8: string strName = Console.ReadLine();

9: Console.WriteLine("Hello {0}",strName);

10: }

11: }

第9行包含了使用格式串的Console.WriteLine语句。格式串例子如下：

"Hello {0}"

{0}代替WriteLine方法的参数表中紧随格式串后的第一个变量。你可以用该技术格式化超过三个变量。

Console.WriteLine("Hello {0} {1}, from {2}",

strFirstname, strLastname, strCity);

当然，并不仅限于只使用字符串变量。你可以使用任何类型，这些类型在后面的第四章 "C#类型"中有讨论。

3.4 添加注释

当写代码时，你应为代码写注释条文，解释实现的内容、变更史等。尽管你注释中提供的信息(如果有的话)是给你写的，但是你还是必须遵守写C#注释的方法。清单3.5 显示采用的两种不同的方式。

清单3.5 给你的代码添加注释

1: using System;

2:

3: class HelloWorld

4: {

5: public static void Main()

6: {

7: // 这是单行注释

8: /\* 这种注释

9: 跨越多行 \*/

10: Console.WriteLine(/\*"Hello World"\*/);

11: }

12: }

"//" 符号用于单行注释。你可以用"//"注释当前所在行，或是跟在一个代码语句的后面：

int nMyVar = 10; // 胡说八道

所有在"//"后面的被认为是一条注释；所以，你可以同样用它们来注释一整行或一行源代码的部分。这种注释方式同C++中介绍的相似。

如果你的注释跨越多行，必须使用"/\* \*/"的字符组合。这种方式在C中有效。除了单行注释外，这种方式在C++和C#中还同样有效。因C/C++和C#都使用这种多行注释方式，所以它们也使用相同的终结符。请看下列代码行：

/\* Console.WriteLine("Hello World"); \*/

我使用"/\* \*/"简单地注释一整行。现在我假定这一行是很长代码的一部分，而且我决定要暂时禁用一个程序块：

/\*

...

/\* Console.WriteLine("Hello World"); \*/

...

\*/

这个结构所存在的问题为： "Hello World"那一行后面的"\*/"终止了始于第一行的"/\*"的注释，余下的代码对编译器有效，你将看到一些有趣的出错信息。至少 最后的"\*/"被标志为归属错误。我只不过想提醒一下，让你了解这种错误。（就是不允许嵌套）

3.5小结

在这一章中，你创建、编译并执行了第一个C#应用程序：著名的"Hello World"程序。我用这个短短的应用程序给你介绍有关Main方法，它是一个应用程序的入口点，也是出口点。这个方法可以没有返回值或返回一个整数错误级别。如果你的应用程序用参数调用，你可以(但不必要)读出并使用它们。

在编译和测试应用程序后，你学到了更多的由Console对象提供的有关输入和输出的方法。对于学习C#而言，它们足以创建出有意义的控制台例子，但用户接口的大部分将是WFC、WinForms或者ASP+。

## 第四章 C#类型

既然你知道了怎样创建一个简单的C#程序，我将会给你介绍C#的类型系统。在这一章中，你学到如何使用不同的值和引用类型，加框和消框机制能为你作些什么。尽管这一章的不侧重于例子，但你可以学到很多重要的信息，关于如何创建现成类型的程序。

4.1 值类型

各种值类型总是含有相应该类型的一个值。C#迫使你初始化变量才能使用它们进行计算。-变量没有初始化（需要赋值）不会出问题，因为当你企图使用它们时，编译器会告诉你。 每当把一个值赋给一个值类型时，该值实际上被拷贝了。相比，对于引用类型，仅是引用被拷贝了，而实际的值仍然保留在相同的内存位置，但现在有两个对象指向了它（引用它）。C#的值类型可以归类如下：（string str;这是定义）

·简单类型（Simple types ）

·结构类型（struct types）

·枚举类型（Enumeration types）

4.1.1 简单类型

在C#中出现的简单类型共享一些特性。第一，它们都是.NET系统类型的别名。第二，由简单类型组成的常量表达式仅在编译时而不是运行时受检测。最后，简单类型可以按字面被初始化。以下为C#简单类型归类：

·整型

·布尔型

·字符型 (整型的一种特殊情况)

·浮点型

·小数型

4.1.1.1 整型

C#中有9个整型。 sbyte 、byte、 short、 ushort、 int、 uint、 long、 ulong 和 char（单独一节讨论）。它们具有以下特性：

·sbyte型为有符号8位整数，取值范围在128~127之间。

·bytet型为无符号8位整数，取值范围在0~255之间。

·short型为有符号16位整数，取值范围在-32,768~32,767之间。

·ushort型为无符号16位整数，取值范围在0~65,535之间。

·int型为有符号32位整数，取值范围在-2,147,483,648~ 2,147,483,647之间。

·uint型为无符号32位整数，取值范围在 0 ~ 4,294,967,295之间。

·long型为64位有符号整数，取值范围在9,223,372,036,854,775,808~ 9,223,372,036,854,775,807之间。

·ulong型为64位无符号整数，取值范围在0 ~ 18,446,744,073,709,551,615之间。

VB和C程序员都可能会对int和long数据类型所代表的新范围感到惊讶。和其它的编程语言相比，在C#中，int不再取决于一个机器的字（word)的大小，而long被设成64位。

4.1.1.2 布尔型

布尔数据类型有true和false两个布尔值。可以赋于true或false值给一个布尔变量，或可以赋于一个表达式，其所求出的值等于两者之一：

bool bTest = (80 > 90);

与C和C++相比，在C#中，true值不再为任何非零值。不要为了增加方便而把其它整型转换成布尔型。

4.1.1.3 字符型

字符型为一个单Unicode 字符。一个Unicode字符16位长，它可以用来表示世界上多种语言。可以按以下方法给一个字符变量赋值：

char chSomeChar ='A';

除此之外，可以通过十六进制转义符（前缀\x）或Unicode表示法给变量赋值（前缀\u）：

char chSomeChar = '\x0065';

char chSomeChar = '\u0065';

不存在把char转换成其它数据类型的隐式转换。这就意味着，在C#中把一个字符变量当作另外的整数数据类型看待是行不通的——这是C程序员必须改变习惯的另一个方面。但是，可以运用显式转换：

char chSomeChar = (char)65;

int nSomeInt = (int)'A';

在C中仍然存在着转义符（字符含义）。要换换脑筋，请看表4.1。

Table 4.1 转义符（ Escape Sequences）

转义符 字符名

\'单引号

\" 双引号

\\ 反斜杠

\0 空字符

\a 感叹号（Alert ）

\b 退格

\f 换页

\n 新行

\r 回车

\t 水平 tab

\v 垂直tab

4.1.1.4 浮点型

两种数据类型被当作浮点型：float和double。它们的差别在于取值范围和精度：

float: 取值范围在 1.5x10^-45~ 3.4x10^38之间， 精度为7位数。

double: 取值范围在 5.0x10^-324 ~ 1.7x10^308之间， 精度为 15~16 位数。

当用两种浮点型执行运算时，可以产生以下的值：

正零和负零

正无穷和负无穷

非数字值（Not-a-Number，缩写NaN）

非零值的有限数集

另一个运算规则为，当表达式中的一个值是浮点型时，所有其它的类型都要被转换成浮点型才能执行运算。

4.1.1.5 小数型（The decimal Type)

小数型是一种高精度、128位数据类型，它打算用于金融和货币的计算。它所表示的范围从大约1.0x10^-28 到 7.9x10^28，具有28至29位有效数字。要注意，精度是以位数 (digits)而不是以小数位（decimal places)表示。运算准确到28个小数位的最大值。

正如你所看到的，它的取值范围比double的还窄，但它更精确。因此，没有decimal和double之间的隐式转换——往一个方向转换可能会溢出，往另外一个方向可能会丢失精度。你不得不运用显式转换。

当定义一个变量并赋值给它时，使用 m 后缀以表明它是一个小数型：

decimal decMyValue = 1.0m;

如果省略了m，在变量被赋值之前，它将被编译器认作double型。

4.1.2 结构类型

一个结构类型可以声明构造函数、常数、字段、方法、属性、索引、操作符和嵌套类型。尽管列出来的功能看起来象一个成熟的类，但在C#中，结构和类的区别在于结构是一个值类型，而类是一个引用类型。与C++相比，这里可以用结构关键字（struct）定义一个类。

使用结构的主要思想是用于创建小型的对象，如Point和FileInfo等等。你可以节省内存，因为没有如类对象所需的那样有额外的引用产生。例如，当声明含有成千上万个对象的数组时，这会引起极大的差异。

清单4.1 包含一个命名为IP的简单结构，它表示一个使用byte类型的4个字段的IP地址。我不包括方法等，因为这些工作正如使用类一样，将在下一章有详细的描述。

清单4.1 定义一个简单的结构

1: using System;

2:

3: struct IP

4: {

5: public byte b1,b2,b3,b4;

6: }

7:

8: class Test

9: {

10: public static void Main()

11: {

12: IP myIP;

13: myIP.b1 = 192;

14: myIP.b2 = 168;

15: myIP.b3 = 1;

16: myIP.b4 = 101;

17: Console.Write("{0}.{1}.",myIP.b1,myIP.b2);

18: Console.Write("{0}.{1}",myIP.b3,myIP.b4);

19: }

20: }

4.1.3 枚举类型

当你想声明一个由一指定常量集合组成的独特类型时，枚举类型正是你要寻觅的。最简单的形式，它看起来可能象这样：

enum MonthNames { January, February, March, April };

因我惯用缺省设置，故枚举元素是int型，且第一个元素为0值。每一个连续的元素按1递增。如果你想给第一个元素直接赋值，可以如下把它设成1：

enum MonthNames { January=1, February, March, April };

如果你想赋任意值给每个元素——甚至相同的值——这也没有问题：

enum MonthNames { January=31, February=28, March=31, April=30 };

最后的选择是不同于int的数据类型。可以在一条语句中按如此赋值：

enum MonthNames : byte { January=31, February=28, March=31, April=30 };

你可以使用的类型仅限于long、int、short和byte。

4.2 引用类型

和值类型相比，引用类型不存储它们所代表的实际数据，但它们存储实际数据的引用。在C#中提供以下引用类型给你使用：

·对象类型

·类类 型

·接口

·代表元

·字符串类型

·数组

4.2.1 对象类型

对象类型是所有类型之母——它是其它类型最根本的基类。因为它是所有对象的基类，所以可把任何类型的值赋给它。例如，一个整型：

object theObj = 123;

给所有的C++程序员一个警告：object并不等价于你可能正在寻找的void\*。无论如何，忘掉指针总是个好主意。

当一个值类型被加框（作为一个对象利用）时，对象类型就被使用了。这一章稍后会讨论到加框和消框

4.2.2 类类型

一个类类型可以包含数据成员、函数成员和嵌套类型。数据成员是常量、字段和事件。函数成员包括方法、属性、索引、操作符、构造函数和析构函数。类和结构的功能是非常相似的，但正如前面所述，结构是值类型而类是引用类型。

和C++相比，仅允许单继承。（你不能拥有派生一个新对象的多重基类。） 但是，C#中的一个类可以派生自多重接口，该接口在下一节将得到描述。

第五章 “类”专门讨论使用类编程。这一节仅打算给出C#类在哪里适合类型图的一个全貌。

4.2.3 接口

一个接口声明一个只有抽象成员的引用类型。跟C++中相似的概念为：一个结构的成员，且方法等于0。如果你不知道那些概念的任何东西，这里就是在C#中一个接口实际所做的。仅仅只存在着方法标志，但根本就没有执行代码。这就暗示了不能实例化一个接口，只能实例化一个派生自该接口的对象。

可以在一个接口中定义方法、属性和索引。所以，对比一个类，接口有什么特殊性呢？当定义一个类时，可以派生自多重接口，而你只能可以从仅有的一个类派生。

你可能会问："OK，但我必须实现所有的接口成员，那么我能从这个途径得到什么呢?" 我想举一个来自.NET的例子：很多类实现了IDictionary 接口。你可以使用简单的类型转换访问接口：

IDictionary myDict = (IDictionary)someobjectthatsupportsit;

现在你的代码可以访问字典了。可等等，我说很多类可以实现这个接口——所以，你可以在多个地方重用代码来访问IDictionary 接口！一旦学会，任何地方都可使用。

当你决定在类设计中使用接口时，学习更多关于面向对象的设计是个好主意。这本书不能教你这些概念，但你可以学习如何创建接口。以下的代码段定义接口IFace，它只有一个方法：

interface IFace

{

void ShowMyFace();

}

正如我所提到的，不能从这个定义实例化一个对象，但可以从它派生一个类。因此，该类必须实现ShowMyFace抽象方法：

class CFace:IFace

{

public void ShowMyFace()

{

Console.WriteLine("implementation");

}

}

接口成员和类成员的区别在于，接口成员不能被实现。因此，我不想在下一章中再次提到这一点。

4.2.4 代表元

一个代表元封装了具有一些标志的一个方法。基本上，代表元是类型安全和函数指针的安全版本(回调功能)。可以同时在一个代表元实例中同时封装静态和实例方法。

尽管你可以用代表员当作具有方法，但它们的主要用途是拥有有一个类事件。再次，我想把你引到下一章，那里会详细地讨论类。

4.2.5 字符串类型

C程序员可能会诧异，但当然，C#有一个用于操作字符串数据的基本字符串类型。字符串类直接派生自对象，且它是被密封的，这意味着再不能从它派生类。就象其它类型，字符串是预定义类System String的一个别名。

它的用法十分简单：

string myString = "some text";

合并字符串同样简单：

string myString = "some text" + " and a bit more";

而如果你想访问单个字符，所要做的就是访问下标：

char chFirst = myString[0];

当比较两个字符串是否相等时，简单地使用"=="比较操作符。

if (myString == yourString) ...

我只不过想提到，尽管字符串是一个引用类型，比较时是比较值，而不是比较引用(内存地址)。

字符串类型几乎用于这本书的每一个例子中，而且在这些例程中，我会介绍给你一些由字符串对象所显露的极其有趣的方法。

4.2.6 数组

一个数组包含有通过计算下标访问的变量。所有包含于数组中且被当作元素的变量必须是同一类型。这种类型自然被称为"数组类型"。数组可以存储整数对象、字符串对象或者 你提出的任何对象。

数组的维数就是所谓的排(rank)，它决定了相关数组元素的下标数。最常用的数组是一维数组(第一排)。一个多维数组具有的排数大于1 。每个维的下标始于0，终于维的长度减1 。

应有足够的理论支持。让我们看一下用一个数组初始化器( array initializer)初始化的数组：

string[] arrLanguages = { "C", "C++", "C#" };//前面的声明使得数组元素为string

该简写效果等同以下：

arrLanguages[0]="C"; arrLanguages[1]="C++"; arrLanguages[2]="C#";

而编译器为你做了所有的工作。当然，它将同样为多维数组初始化器工作：

int[,] arr = {{0,1}, {2,3}, {4,5}};//前面的声明使得数组元素为int

它是以下的简写：

arr[0,0] = 0; arr[0,1] = 1;

arr[1,0] = 2; arr[1,1] = 3;

arr[2,0] = 4; arr[2,1] = 5;

如果你不想事先初始化一个数组，但知道了它的大小，该声明就象这样：

int[,] myArr = new int[5,3];

如果数组的大小必须动态地被计算，用于数组创建的语句可以象这样写：

int nVar = 5;

int[] arrToo = new int[nVar];

正如我在这一节开始所陈述的，你可以往数组里面塞任何东西，只要所有的元素类型都相同。因此，如果你想把任何东西放进一个数组，就声明它的类型为对象：

4.3 加框和消框

这一章的课程中，我已经给出了各式各样的值类型和引用类型。由于速度的原因，你会使用值类型——它除了占据一定空间的内存块外，就没有什么了。但是，有时对象的方便性就象值类型一样好用。

这就是加框和消框登上了舞台的地方，加框和消框是C#类型系统的核心概念。通过允许一个值类型转换成类型对象或从类型对象转换成值类型，这种机制形成了值类型和引用类型之间的捆绑连接。任何东西终究是一个对象——但是，仅当需要它们是对象时。

4.3.1 加框转换

给一个值加框指隐式地把任何值类型转换成类型对象。当一个值类型被加框时，一个对象实例就被分配，且值类型的值被拷贝给新的对象。

看以下例子：

int nFunny = 2000;

object oFunny = nFunny;

第二行的赋值暗示调用一个加框操作。nFunny整型变量的值被拷贝给oFunny对象。现在整型变量和对象变量都同时存在于栈中，但对象的值居留在堆中。

那么，它暗示着什么呢?它们的值互相独立——在它们之间没有连接。(oFunny没有引用nFunny的值。) 以下代码说明了结果：

int nFunny = 2000;

object oFunny = nFunny;

oFunny = 2001;

Console.WriteLine("{0} {1}", nFunny, oFunny);

当代码改变oFunny的值时，nFunny的值并没有改变。只要你脑袋中有这个copy动作，就能够使用值类型的对象功能，发挥出你的巨大优势！

4.3.2 消框转换

和加框相比，消框是显式操作——必须告诉编译器，你想从对象中抽取出哪一种值类型。当执行消框操作时，C#检测所请求的值类型实际上存储在对象实例中。经过成功的校验，该值被消框。

这就是消框如何执行：

int nFunny = 2000;

object oFunny = nFunny;

int nNotSoFunny = (int)oFunny;

如果错误地请求一个double值

double nNotSoFunny = (double)oFunny;//之所以产生错误因为加框之前为int而非double

通用语言运行时(Common Language Runtime，简写CLR)将会引发一个InvalidCastException异常。你可以在第7章 "异常处理" 中学到更多有关异常处理的知识。

4.4 小结

在这一章中，你学到了C#中用到的各种类型。简单的值类型包括整型、布尔型、浮点型和小数型。你会非常经常地用到一些类型，进行数学和金融的计算，还有逻辑表达。

在介绍引用类型之前，我显示了一个看起来象类的结构类型。它几乎如一个类般地运作，但它只是一个值类型，这使它更加适合需要有大量的小对象的场合。

引用类型起始于所有对象之母的objedt本身。object是C#中所有对象的基类，且它同样用于值类型的加框和消框。除此之外，我还让你领略了代表元、字符串和数组。

令C#程序员十分神气的类型就是类。它是C#面向对象编程的心脏，下一章整章专门让你迅速理解这个激动人心且功能强大的类型。

## 第五章 类

前一章讨论了数据类型和它们的用法。现在我们转移到C#中至关重要的结构——类。没有了

类，就连简单的C#程序都不能编译。这一章假定你知道了一个类的基本组成部分：方法、属性、构

造函数和析构函数。 C#在其中增加了索引和事件。

在这一章中，你学到下列有关类的话题。

。 使用构造函数和析构函数

。给类写方法

。给一个类增加属性存取标志

。实现索引

。创建事件并通过代表元为事件关联客户

。应用类、成员和存取修饰符。

5.1 构造函数和析构函数

在你可以访问一个类的方法、属性或任何其它东西之前， 第一条执行的语句是包含有相

应类的构造函数。甚至你自己不写一个构造函数，也会有一个缺省的构造函数提供给你。

class TestClass

{

public TestClass(): base() {} // 由编译器提供

}

一个构造函数总是和它的类名相同，但是，它没有声明返回类型。总之，构造函数总是

public的，你可以用它们来初始化变量。

public TestClass()

{

// 在这给变量

// 初始化代码等等。

}

如果类仅包含静态成员(能以类型调用，而不是以实例调用的成员)，你可以创建一个

private的构造函数。

private TestClass() {}

尽管存取修饰符在这一章的后面将要大篇幅地讨论，但是private意味着从类的外面不可能

访问该构造函数。所以，它不能被调用，且没有对象可以自该类定义被实例化。

并不仅限于无参数构造函数——你可以传递初始参数来初始化成员。

public TestClass(string strName, int nAge) { ... }

作为一个C/C++程序员，你可能习惯于给初始化写一个附加的方法，因为在构造函数中没有

返回值。当然，尽管在C#中也没有返回值，但你可以引发一个自制的异常，以从构造函数获得返回

值。更多有关异常处理的知识在第七章 "异常处理"中有讨论。

但是，当你保留引用给宝贵的资源，应该想到写一个方法来解决：一个可以被显式地调用来

释放这些资源。问题是当你可以在析构函数(以类名的前面加"~"的方式命名)中做同样的事情时，为

何还要写一个附加的方法.

public ~TestClass()

{

// 清除

}

你应该写一个附加方法的原因是垃圾收集器，它在变量超出范围后并不会立即被调用，而仅当

间歇期间或内存条件满足时才被触发。当你锁住资源的时间长于你所计划的时间时，它就会发生。

因此，提供一个显式的释放方式是一个好主意，它同样能从析构函数中调用。

public void Release()

{

// 释放所有宝贵的资源

}

public ~TestClass()

{

Release();

}

调用析构函数中的释放方法并不是必要的——总之，垃圾收集会留意释放对象。但没有忘记清

除是一种良好的习惯。

5.2 方法

既然对象能正确地初始化和结束，所剩下来的就是往类中增加功能。在大多数情况下，功能的

主要部分在方法中能得到实现。你早已见过静态方法的使用，但是，这些是类型(类)的部分，不是

实例(对象)。

为了让你迅速入门，我把这些方法的烦琐问题安排为三节：

。方法参数

。改写方法

。方法屏蔽

5.2.1 方法参数

因方法要处理更改数值，你多多少少要传递值给方法，并从方法获得返回值。以下三个部分涉及

到由传递值和为调用者获取返回结果所引起的问题。

。输入参数

。引用参数

。输出参数

5.2.1.1 输入参数

你早已在例子中见过的一个参数就是输入参数。你用一个输入参数通过值传递一个变量给一个方

法——方法的变量被调用者传递进来的值的一个拷贝初始化。清单5.1 示范输入参数的使用。

清单 5.1 通过值传递参数

1: using System;

2:

3: public class SquareSample

4: {

5: public int CalcSquare(int nSideLength)

6: {

7: return nSideLength\*nSideLength;

8: }

9: }

10:

11: class SquareApp

12: {

13: public static void Main()

14: {

15: SquareSample sq = new SquareSample();

16: Console.WriteLine(sq.CalcSquare(25).ToString());

17: }

18: }

因为我传递值而不是引用给一个变量，所以当调用方法时(见第16行)，可以使用一个常量表达式

(25)。整型结果被传回给调用者作为返回值，它没有存到中间变量就被立即显示到屏幕上 。

输入参数按C/C++程序员早已习惯的工作方式工作。如果你来自VB，请注意没有能被编译器处理

的隐式ByVal或ByRef——如果没有设定，参数总是用值传递。

这点似乎与我前面所陈述的有冲突：对于一些变量类型，用值传递实际上意味着用引用传递。

迷惑吗? 一点背景知识也不需要：COM中的东西就是接口，每一个类可以拥有一个或多个接口。一个

接口只不过是一组函数指针，它不包含数据。重复该数组会浪费很多内存资源；所以，仅开始地址

被拷贝给方法，它作为调用者，仍然指向接口的相同指针。那就是为什么对象用值传递一个引用。

5.2.1.2 引用参数

尽管可以利用输入参数和返回值建立很多方法，但你一想到要传递值并原地修改它(也就是在相

同的内存位置)，就没有那么好运了。这里用引用参数就很方便。

void myMethod(ref int nInOut)

因为你传递了一个变量给该方法(不仅仅是它的值)，变量必须被初始化。否则，编译器会报警。

清单 5.2 显示如何用一个引用参数建立一个方法。

清单 5.2 通过引用传递参数

1: // class SquareSample

2: using System;

3:

4: public class SquareSample

5: {

6: public void CalcSquare(ref int nOne4All)

7: {

8: nOne4All \*= nOne4All;

9: }

10: }

11:

12: class SquareApp

13: {

14: public static void Main()

15: {

16: SquareSample sq = new SquareSample();

17:

18: int nSquaredRef = 20; // 一定要初始化

19: sq.CalcSquare(ref nSquaredRef);

20: Console.WriteLine(nSquaredRef.ToString());

21: }

22: }

正如所看到的，所有你要做的就是给定义和调用都加上ref限定符。因为变量通过引用传递，你

可以用它来计算出结果并传回该结果。但是，在现实的应用程序中，我强烈建议要用两个变量，一

个输入参数和一个引用参数。

5.2.1.3 输出参数

传递参数的第三种选择就是把它设作一个输出参数。正如该名字所暗示，一个输出参数仅用于从

方法传递回一个结果。它和引用参数的另一个区别在于：调用者不必先初始化变量才调用方法。这

显示在清单5.3中。

清单 5.3 定义一个输出参数

1: using System;

2:

3: public class SquareSample

4: {

5: public void CalcSquare(int nSideLength, out int nSquared)

6: {

7: nSquared = nSideLength \* nSideLength;

8: }

9: }

10:

11: class SquareApp

12: {

13: public static void Main()

14: {

15: SquareSample sq = new SquareSample();

16:

17: int nSquared; // 不必初始化

18: sq.CalcSquare(15, out nSquared);

19: Console.WriteLine(nSquared.ToString());

20: }

21: }

5.2.2 改写方法

面向对象设计的重要原则就是多态性。不要理会高深的理论，多态性意味着：当基类程序员已

设计好用于改写的方法时，在派生类中，你就可以重定义(改写)基类的方法。基类程序员可以用

virtual 关键字设计方法：

virtual void CanBOverridden()

当从基类派生时，所有你要做的就是在新方法中加入override关键字：

override void CanBOverridden()

当改写一个基类的方法时，你必须明白，不能改变方法的访问属性——在这章的后面，你会学

到更多关于访问修饰符的知识。

除了改写基类方法的事实外，还有另一个甚至更重要的改写特性。当把派生类强制转换成基类

类型并接着调用虚拟方法时，被调用的是派生类的方法而不是基类的方法。

((BaseClass)DerivedClassInstance).CanBOverridden();

为了演示虚拟方法的概念，清单 5.4 显示如何创建一个三角形基类，它拥有一个可以被改

写的成员方法（ComputeArea）。

清单 5.4 改写一个基类的方法

1: using System;

2:

3: class Triangle

4: {

5: public virtual double ComputeArea(int a, int b, int c)

6: {

7: // Heronian formula

8: double s = (a + b + c) / 2.0;

9: double dArea = Math.Sqrt(s\*(s-a)\*(s-b)\*(s-c));

10: return dArea;

11: }

12: }

13:

14: class RightAngledTriangle:Triangle

15: {

16: public override double ComputeArea(int a, int b, int c)

17: {

18: double dArea = a\*b/2.0;

19: return dArea;

20: }

21: }

22:

23: class TriangleTestApp

24: {

25: public static void Main()

26: {

27: Triangle tri = new Triangle();

28: Console.WriteLine(tri.ComputeArea(2, 5, 6));

29:

30: RightAngledTriangle rat = new RightAngledTriangle();

31: Console.WriteLine(rat.ComputeArea(3, 4, 5));

32: }

33: }

基类Triangle定义了方法ComputeArea。它采用三个参数，返回一个double结果，且具有公共访

问性。从Triangle类派生出的是RightAngledTriangle，它改写了ComputeArea 方法，并实现了自己

的面积计算公式。两个类都被实例化，且在命名为TriangleTestApp的应用类的Main() 方法中得到

验证。

我漏了解释第14行：

class RightAngledTriangle : Triangle

在类语句中冒号（：）表示RightAngledTriangle从类 Triangle派生。那就是你所必须要做的

，以让C#知道你想把 Triangle当作RightAngledTriangle的基类。

当仔细观察直角三角形的ComputeArea方法时，你会发现第3个参数并没有用于计算。但是，利

用该参数就可以验证是否是“直角”。如清单5.5所示。

清单 5.5 调用基类实现

1: class RightAngledTriangle:Triangle

2: {

3: public override double ComputeArea(int a, int b, int c)

4: {

5: const double dEpsilon = 0.0001;

6: double dArea = 0;

7: if (Math.Abs((a\*a + b\*b - c\*c)) > dEpsilon)

8: {

9: dArea = base.ComputeArea(a,b,c);

10: }

11: else

12: {

13: dArea = a\*b/2.0;

14: }

15:

16: return dArea;

17: }

18: }

该检测简单地利用了毕达哥拉斯公式，对于直角三角形，检测结果必须为0。如果结果不为0，类

就调用它基类的 ComputeArea来实现。

dArea = base.ComputeArea(a,b,c);

例子的要点为：通过显式地利用基类的资格检查，你就能轻而易举地调用基类实现改写方法。

当你需要实现其在基类中的功能，而不愿意在改写方法中重复它时，这就非常有帮助。

5.2.3 方法屏蔽

重定义方法的一个不同手段就是要屏蔽基类的方法。当从别人提供的类派生类时，这个功能特

别有价值。看清单 5.6，假设BaseClass由其他人所写，而你从它派生出 DerivedClass 。

清单 5.6 Derived Class 实现一个没有包含于 Base Class中的方法

1: using System;

2:

3: class BaseClass

4: {

5: }

6:

7: class DerivedClass:BaseClass

8: {

9: public void TestMethod()

10: {

11: Console.WriteLine("DerivedClass::TestMethod");

12: }

13: }

14:

15: class TestApp

16: {

17: public static void Main()

18: {

19: DerivedClass test = new DerivedClass();

20: test.TestMethod();

21: }

22: }

在这个例子中， DerivedClass 通过TestMethod()实现了一个额外的功能。但是，如果基类的

开发者认为把TestMethod()放在基类中是个好主意，并使用相同的名字实现它时，会出现什么问题

呢？（见清单5.7）

清单 5.7 Base Class 实现和 Derived Class相同的方法

1: class BaseClass

2: {

3: public void TestMethod()

4: {

5: Console.WriteLine("BaseClass::TestMethod");

6: }

7: }

8:

9: class DerivedClass:BaseClass

10: {

11: public void TestMethod()

12: {

13: Console.WriteLine("DerivedClass::TestMethod");

14: }

15: }

在优秀的编程语言中，你现在会遇到一个真正的大麻烦。但是，C#会给你提出警告：

hiding2.cs(13,14): warning CS0114: 'DerivedClass.TestMethod()' hides inherited member

'BaseClass.TestMethod()'. To make the current method override that implementation, add

the override keyword. Otherwise add the new keyword.

（hiding2.cs(13,14):警告 CS0114:'DerivedClass.TestMethod()' 屏蔽了所继承的成员

'BaseClass.TestMethod()'。要想使当前方法改写原来的实现，加上 override关键字。否则加上新

的关键字。）

具有了修饰符new，你就可以告诉编译器，不必重写派生类或改变使用到派生类的代码，你的方法就

能屏蔽新加入的基类方法。清单5.8 显示如何在例子中运用new修饰符。

清单 5.8 屏蔽基类方法

1: class BaseClass

2: {

3: public void TestMethod()

4: {

5: Console.WriteLine("BaseClass::TestMethod");

6: }

7: }

8:

9: class DerivedClass:BaseClass

10: {

11: new public void TestMethod()

12: {

13: Console.WriteLine("DerivedClass::TestMethod");

14: }

15: }

使用了附加的new修饰符，编译器就知道你重定义了基类的方法，它应该屏蔽基类方法。但是，如果

你按以下方式编写：

DerivedClass test = new DerivedClass();

((BaseClass)test).TestMethod();

基类方法的实现就被调用了。这种行为不同于改写方法，后者保证大部分派生方法获得调用。

5.3 类属性

有两种途径揭示类的命名属性——通过域成员或者通过属性。前者是作为具有公共访问性的成员变量而被实现的；后者并不直接回应存储位置，只是通过 存取标志(accessors)被访问。

当你想读出或写入属性的值时，存取标志限定了被实现的语句。用于读出属性的值的存取标志记为关键字get，而要修改属性的值的读写符标志记为set。在你对该理论一知半解以前，请看一下清单5.9中的例子，属性SquareFeet被标上了get和set的存取标志。

清单 5.9 实现属性存取标志

1: using System;

2:

3: public class House

4: {

5: private int m\_nSqFeet;

6:

7: public int SquareFeet

8: {

9: get { return m\_nSqFeet; }

10: set { m\_nSqFeet = value; }

11: }

12: }

13:

14: class TestApp

15: {

16: public static void Main()

17: {

18: House myHouse = new House();

19: myHouse.SquareFeet = 250;

20: Console.WriteLine(myHouse.SquareFeet);

21: }

22: }

House类有一个命名为SquareFeet的属性，它可以被读和写。实际的值存储在一个可以从类内部访问的变量中——如果你想当作一个域成员重写它，你所要做的就是忽略存取标志而把变量重新定义为：

public int SquareFeet;

对于一个如此简单的变量，这样不错。但是，如果你想要隐藏类内部存储结构的细节时，就应该采用存取标志。在这种情况下，set 存取标志给值参数中的属性传递新值。（可以改名，见第10行。）

除了能够隐藏实现细节外，你还可自由地限定各种操作：

get和set：允许对属性进行读写访问。

get only：只允许读属性的值。

set only：只允许写属性的值。

除此之外，你可以获得实现在set标志中有效代码的机会。例如，由于种种原因（或根本没有原因），你就能够拒绝一个新值。最好是没有人告诉你它是一个动态属性——当你第一次请求它后，它会保存下来，故要尽可能地推迟资源分配。

5.4 索引

你想过象访问数组那样使用索引访问类吗 ？使用C#的索引功能，对它的期待便可了结。

语法基本上象这样：

属性 修饰符 声明 { 声明内容}

具体的例子为

public string this[int nIndex]

{

get { ... }

set { ... }

}

索引返回或按给出的index设置字符串。它没有属性，但使用了public修饰符。声明部分由类型string和this 组成用于表示类的索引。get和set的执行规则和属性的规则相同。（你不能取消其中一个。） 只存在一个差别，那就是：你几乎可以任意定义大括弧中的参数。限制为，必须至少规定一个参数，允许ref 和out 修饰符。

this关键字确保一个解释。索引没有用户定义的名字，this 表示默认接口的索引。如果类实现了多个接口，你可以增加更多个由InterfaceName.this说明的索引。

为了演示一个索引的使用，我创建了一个小型的类，它能够解析一个主机名为IP地址——或一个IP地址列表(以http://www.microsoft.com为例 )。这个列表通过索引可以访问，你可以看一下清单

5.10 的具体实现。

清单 5.10 通过一个索引获取一个IP地址

1: using System;

2: using System.Net;

3:

4: class ResolveDNS

5: {

6: IPAddress[] m\_arrIPs;

7:

8: public void Resolve(string strHost)

9: {

10: IPHostEntry iphe = DNS.GetHostByName(strHost);

11: m\_arrIPs = iphe.AddressList;

12: }

13:

14: public IPAddress this[int nIndex]

15: {

16: get

17: {

18: return m\_arrIPs[nIndex];

19: }

20: }

21:

22: public int Count

23: {

24: get { return m\_arrIPs.Length; }

25: }

26: }

27:

28: class DNSResolverApp

29: {

30: public static void Main()

31: {

32: ResolveDNS myDNSResolver = new ResolveDNS();

33: myDNSResolver.Resolve("http://www.microsoft.com");

34:

35: int nCount = myDNSResolver.Count;

36: Console.WriteLine("Found {0} IP's for hostname", nCount);

37: for (int i=0; i < nCount; i++)

38: Console.WriteLine(myDNSResolver[i]);

39: }

40: }

为了解析主机名，我用到了DNS类，它是System .Net 名字空间的一部分。但是，由于这个名字空间并不包含在核心库中，所以必须在编译命令行中引用该库：

csc /r:System.Net.dll /out:resolver.exe dnsresolve.cs

解析代码是向前解析的。在该 Resolve方法中，代码调用DNS类的静态方法GetHostByName，它返回一个IPHostEntry对象。结果，该对象包含有我要找的数组——AddressList数组。在退出Resolve 方法之前，在局部的对象实例成员m\_arrIPs中，存储了一个AddressList array的拷贝(类型IPAddress 的对象存储在其中)。

具有现在生成的数组 ，通过使用在类ResolveDNS中求得的索引，应用程序代码就可以在第37至38行列举出IP地址。(在第6章 "控制语句"，有更多有关语句的信息。) 因为没有办法更改IP地址，所以仅给索引使用了get存取标志。为了简单其见，我忽略了数组的边界溢出检查。

5.4 事件

当你写一个类时，有时有必要让类的客户知道一些已经发生的事件。如果你是一个具有多年编程经验的程序员，似乎有很多的解决办法，包括用于回调的函数指针和用于ActiveX控件的事件接收（event sinks)。现在你将要学到另外一种把客户代码关联到类通知的办法——使用事件。

事件既可以被声明为类域成员（成员变量），也可以被声明为属性。两者的共性为，事件的类型必定是代表元，而函数指针原形和C#的代表元具有相同的含义。

每一个事件都可以被0或更多的客户占用，且客户可以随时关联或取消事件。你可以以静态或者以实例方法定义代表元，而后者很受C++程序员的欢迎。

既然我已经提到了事件的所有功能及相应的代表元，请看清单5.11中的例子。它生动地体现了该理论。

清单5.11 在类中实现事件处理

1: using System;

2:

3: // 向前声明

4: public delegate void EventHandler(string strText);

5:

6: class EventSource

7: {

8: public event EventHandler TextOut;

9:

10: public void TriggerEvent()

11: {

12: if (null != TextOut) TextOut("Event triggered");

13: }

14: }

15:

16: class TestApp

17: {

18: public static void Main()

19: {

20: EventSource evsrc = new EventSource();

21:

22: evsrc.TextOut += new EventHandler(CatchEvent);

23: evsrc.TriggerEvent();

24:

25: evsrc.TextOut -= new EventHandler(CatchEvent);

26: evsrc.TriggerEvent();

27:

28: TestApp theApp = new TestApp();

29: evsrc.TextOut += new EventHandler(theApp.InstanceCatch);

30: evsrc.TriggerEvent();

31: }

32:

33: public static void CatchEvent(string strText)

34: {

35: Console.WriteLine(strText);

36: }

37:

38: public void InstanceCatch(string strText)

39: {

40: Console.WriteLine("Instance " + strText);

41: }

42: }

第4行声明了代表元（事件方法原形），它用来给第8行中的EventSource类声明TextOut事件域成员。你可以观察到代表元作为一种新的类型声明，当声明事件时可以使用代表元。

该类仅有一个方法，它允许我们触发事件。请注意，你必须进行事件域成员不为null的检测，因为可能会出现没有客户对事件感兴趣这种情况。

TestApp类包含了Main 方法，也包含了另外两个方法，它们都具备事件所必需的信号。其中一个方法是静态的，而另一个是实例方法。

EventSource 被实例化，而静态方法CatchEvent被预关联上了 TextOut事件：

evsrc.TextOut += new EventHandler(CatchEvent);

从现在起，当事件被触发时，该方法被调用。如果你对事件不再感兴趣，简单地取消关联：

evsrc.TextOut -= new EventHandler(CatchEvent);

注意，你不能随意取消关联的处理函数——在类代码中仅创建了这些处理函数。为了证明事件处理函数也和实例方法一起工作，余下的代码建立了TestApp 的实例，并钩住事件处理方法。

事件在哪方面对你特别有用？你将经常在ASP+中或使用到WFC (Windows Foundation Classes)时，涉及到事件和代表元。

5.5 应用修饰符

在这一章的学习过程中，你已经见过了象public、virtual等修饰符。欲以一种易于理解的方法概括它们，我把它们划分为三节：

。类修饰符

。成员修饰符

。存取修饰符

5.5.1 类修饰符

到目前为止，我还没有涉及到类修饰符，而只涉及到了应用于类的存取修饰符。但是，有两个修饰符你可以用于类：

abstract——关于抽象类的重要一点就是它不能被实例化。只有不是抽象的派生类才能被实例化。派生类必须实现抽象基类的所有抽象成员。你不能给抽象类使用sealed 修饰符。

sealed——密封 类不能被继承。使用该修饰符防止意外的继承，在.NET框架中的类用到这个修饰符。

要见到两个修饰符的运用，看看清单5.12 ，它创建了一个基于一个抽象类的密封类（肯定是一个十分极端的例子）。

清单 5.12 抽象类和密封类

1: using System;

2:

3: abstract class AbstractClass

4: {

5: abstract public void MyMethod();

6: }

7:

8: sealed class DerivedClass:AbstractClass

9: {

10: public override void MyMethod()

11: {

12: Console.WriteLine("sealed class");

13: }

14: }

15:

16: public class TestApp

17: {

18: public static void Main()

19: {

20: DerivedClass dc = new DerivedClass();

21: dc.MyMethod();

22: }

23: }

5.5.2 成员修饰符

与有用的成员修饰符的数量相比，类修饰符的数量很少。我已经提到了一些，这本书即将出现的例子描述了其它的成员修饰符。

以下是有用的成员修饰符：

abstract——说明一个方法或存取标志不能含有一个实现。它们都是隐式虚拟，且在继承类中，你必须提供 override关键字。

const——这个修饰符应用于域成员或局部变量。在编译时常量表达式被求值，所以，它不能包含变量的引用。

event ——定义一个域成员或属性作为类型事件。用于捆绑客户代码到类的事件。

extern——告诉编译器方法实际上由外部实现。第10章 “和非受管代码互相操作” 将全面地涉及到外部代码。

override——用于改写任何基类中被定义为virtual的方法和存取标志。要改写的名字和基类的方法必须一致。

readonly——一个使用 readonly修饰符的域成员只能在它的声明或者在包含它的类的构造函数中被更改。

static——被声明为static的成员属于类，而不属于类的实例。你可以用static 于域成员、方法、属性、操作符甚至构造函数。

virtual——说明方法或存取标志可以被继承类改写。

5.5.3 存取修饰符

存取修饰符定义了某些代码对类成员（如方法和属性）的存取等级。你必须给每个成员加上所希望的存取修饰符，否则，默认的存取类型是隐含的。

你可以应用4个 存取修饰符之一：

public——任何地方都可以访问该成员，这是具有最少限制的存取修饰符。

protected——在类及所有的派生类中可以访问该成员，不允许外部访问。

private——仅仅在同一个类的内部才能访问该成员。甚至派生类都不能访问它。

internal——允许相同组件（应用程序或库）的所有代码访问。在.NET组件级别，你可以把它视为public，而在外部则为private。

为了演示存取修饰符的用法，我稍微修改了Triangle例子，使它包含了新增的域成员和一个新的派生类（见清单 5.13）。

清单 5.13 在类中使用存取修饰符

1: using System;

2:

3: internal class Triangle

4: {

5: protected int m\_a, m\_b, m\_c;

6: public Triangle(int a, int b, int c)

7: {

8: m\_a = a;

9: m\_b = b;

10: m\_c = c;

11: }

12:

13: public virtual double Area()

14: {

15: // Heronian formula

16: double s = (m\_a + m\_b + m\_c) / 2.0;

17: double dArea = Math.Sqrt(s\*(s-m\_a)\*(s-m\_b)\*(s-m\_c));

18: return dArea;

19: }

20: }

21:

22: internal class Prism:Triangle

23: {

24: private int m\_h;

25: public Prism(int a, int b, int c, int h):base(a,b,c)

26: {

27: m\_h = h;

28: }

29:

30: public override double Area()

31: {

32: double dArea = base.Area() \* 2.0;

33: dArea += m\_a\*m\_h + m\_b\*m\_h + m\_c\*m\_h;

34: return dArea;

35: }

36: }

37:

38: class PrismApp

39: {

40: public static void Main()

41: {

42: Prism prism = new Prism(2,5,6,1);

43: Console.WriteLine(prism.Area());

44: }

45: }

Triangle 类和 Prism 类现在被标为 internal。这意味着它们只能在当前组件中被访问。

请记住“.NET组件”这个术语指的是包装（ packaging,），而不是你可能在COM+中用到的组件。

Triangle 类有三个 protected成员，它们在构造函数中被初始化，并用于面积计算的方法中。由于这些成员是protected 成员，所以我可以在派生类Prism中访问它们，在那里执行不同的面积计算。

Prism自己新增了一个成员m\_h，它是私有的——甚至派生类也不能访问它。

花些时间为每个类成员甚至每个类计划一种保护层次，通常是个好主意。当需要引入修改时，全面的计划最终会帮助你，因为没有程序员会愿意使用“没有文档”的类功能。

5.6 小结

这章显示了类的各种要素，它是运行实例（对象）的模板。在一个对象的生命期，首先被执行的代码是个构造函数。构造函数用来初始化变量，这些变量后来在方法中用于计算结果。

方法允许你传递值、引用给变量，或者只传送一个输出值。方法可以被改写以实现新的功能，或者你可以屏蔽基类成员，如果它实现了一个具有和派生类成员相同名字的方法。

命名属性可以被当作域成员（成员变量）或属性存取标志实现。后者是get和set存取标志，忽略一个或另外一个，你可以创建仅写或仅读属性。存取标志非常适合于确认赋给属性的值。

C#类的另外一个功能是索引，它使象数组语法一样访问类中值成为可能。还有，如果当类中的某些事情发生时，你想客户得到通知，要让它们与事件关联。

当垃圾收集器调用析构函数时，对象的生命就结束了。由于你不能准确地预测这种情况什么时候会发生，所以应该创建一个方法以释放这些宝贵的资源，当你停止使用它们时。