返回总目录

目　　录

第一部分　C#语言概述

第一章　.NET 编程语言 C#

未来 5 年，我们的目标就是超越今天各自为营的 Web 站点，把 Internet 建成一个可以互相交换组件的地方。——比尔.盖茨

在本章中你将了解：

- Microsoft.NET 的概念
- .NET 框架
- C#语言在.NET 框架中的作用及其特性

1.1　Microsoft.NET——一场新的革命

1.1.1　什么是.NET

2000 年 6 月 22 日，不论对 Microsoft 还是对整个 IT 业界都将成为值得纪念的一天。这一天，微软公司正式推出了其下一代计算计划——Microsoft.NET(以下简称.NET)。这项计划将使微软现有的软件在 Web 时代不仅适用于传统的 PC，而且也能够满足目前呈强劲增长势头的新设备，诸如蜂窝电话以及个人数字助理（Personal Digital Assistant, PDA）等的需要。微软还计划通过创建新的工具来吸引软件开发人员和合作伙伴对 Microsoft.NET 的认同，并且开发出其他基于 Internet 的服务。

那么，你是否想知道：究竟什么是.NET?

请听听微软官员的声音：“……因特网的革命……从微软的角度来讲，我们就是要建设一个平台来创建并且支持新一代的应用。……我们必须有一套通用系统服务来支持这样的操作。这种观点就说明，我们还有下一个层次的发展，也就是说因特网下一步的发展，它将使因特网的作用远远超越展现一个网站。”

.NET 首先是一个开发平台，它定义了一种公用语言子集（Common Language Subset， CLS）,这是一种为符合其规范的语言与类库之间提供无缝集成的混合语。.NET 统一了编程类库，提供了对下一代网络通信标准，可扩展标记语言（Extensible Markup

Language，XML）的完全支持，使应用程序的开发变得更容易、更简单。Microsoft.NET 计划还将实现人机交互方面的革命，微软将在其软件中添加手写和语音识别的功能，让人们能够与计算机进行更好的交流，并在此基础上继续扩展功能，增加对各种用户终端的支持能力。最为重要的，.NET 将改变因特网的行为方式：软件将变成为服务。与 Microsoft 的其它产品一样，.NET 与 Windows 平台紧密集成，并且与其它微软产品相比它更进一步：由于其运行库已经与操作系统融合在了一起，从广义上把它称为一个运行库也不为过。

简而言之，.NET 是一种面向网络、支持各种用户终端的开发平台环境。微软的宏伟目标是让 Microsoft.NET 彻底改变软件的开发方式、发行方式、使用方式等等，并且不止是针对微软一家，而是面向所有开发商与运营商！.NET 的核心内容之一就是要搭建第三代因特网平台，这个网络平台将解决网站之间的协同合作问题，从而最大限度地获取信息。在 .NET 平台上，不同网站之间通过相关的协定联系在一起，网站之间形成自动交流，协同工作，提供最全面的服务。

1.1.2 我们为什么需要.NET

某一天，你出差到外地，在机场租借手机电话。在向该终端插入自己的 IC 卡后，自己的地址簿和计划簿被自动下载，随即它就变成了你个人专用的 PDA。这不是梦境！这是.NET 为我们描绘的一个未来生活的场景。

人们的需要总是无法满足，我们不断地问自己：“我们还应该有些什么？”需求推动着技术的进步。在二十一世纪，Internet 将成为商业活动的主要场所，B2B、B2C 等电子商务的运作方式，一对一营销的经营概念将网络的服务功能提高到了前所未有的程度。微软公司在此时提出.NET 有其深远的战略考虑：

改革商务模型。微软公司感觉到只靠销售软件包的商务模型没有什么前途，该公司打算今后将中心转移到可以在网络上使用“服务”型商务。这样，首要的问题就是解决网络上用来开发并执行“服务”的平台，这就是 Microsoft.NET。

提高软件开发生产效率，并且试图使应用软件的发布更为容易（再也不想因为 DLL 版本不同而烦恼，希望不用重新启动电脑就能够安装应用软件）。

改进用户界面，并能支持多种用户终端。用户界面演进的结果包括两方面的内容，一是完成传统的 PC 界面与基于 XML 的浏览器界面间的过渡，二是对自然语言和语音识别的支持，从而使用户与各种终端之间的沟通更加透明，真正达到网络互连的“3A”：Anywhere、Anytime、Any device。

今天，许多的人时常问：“除了上网看新闻，我们究竟还能干什么？”这是因为今天的互联网与旧式的大型计算机的工作模式还有许多相似之处，信息被储存在中央服务器内，而用户的所有操作都要依靠它们。让不同的网址之间相互传递有意义的信息，或者合作提供更广泛和更深层次的服务，还是一件十分困难的事。

现代人时常有一种困惑，感觉到如今生活在技术与机器架构的丛林中，我们在努力地去适应机器，适应技术，而不是机器和技术适应人类。科技以人为本还只是一个美好的愿望。这是因为我们还不能将控制信息的权利交给那些需要信息的人们。 .NET

的出现，意味着人们可以只用一种简单的界面就可以编写、浏览、编辑和分享信息，而且还可以得到功能强大的信息管理工具。由于使用的所有的文件都以符合网络协议的格式存在，所以所有的商业用户和个人用户都可以方便地查找和使用其中的信息，任何规模的公司都可以使用相同的工具与他们的供应商、商业伙伴和客户高效地沟通和分享信息，这样就创造出一种全新的协同工作模式。

总之，.NET 战略是一场软件革命：

- .NET 对最终用户来说非常重要，因为计算机的功能将会得到大幅度提升，同时计算机操作也会变得非常简单。特别地，用户将完全摆脱人为的硬件束缚：用户可以自由冲浪于因特网的多维时空，自由访问、自由查看、自由使用自己的数据，而不是束缚在便携式电脑的方寸空间——可通过任何桌面系统、任何便携式电脑、任何移动电话或 PDA 进行访问，并可对其进行跨应用程序的集成。
- .NET 对开发人员来说也十分重要，因为它不但会改变开发人员开发应用程序的方式，而且使得开发人员能创建出全新的各种应用程序，大幅提高软件生产率。.NET 将保证完全消除当今计算技术中的所有缺陷。.NET 定能实现确保用户从任何地点、任何设备都可访问其个人数据和应用程序的宏伟蓝图。
- .NET 把雇员、客户和商务应用程序整和成一个协调的、能进行智能交互的整体，而各公司无疑将是这场效率和生产力革命的最大受益者。.NET 承诺为人类创造一个消除任何鸿沟的商务世界。

1.1.3 .NET 的核心组件

.NET 的核心组件包括：

- 一组用于创建互联网操作系统的构建块，其中包括 Passport.NET（用于用户认证）以及用于文件存储的服务、用户首选项管理、日历管理以及众多的其它任务。
- 构建和管理新一代服务的基本结构和工具，包括 Visual Studio.NET、.NET 企业服务器、.Net Framework 和 Windows.NET。
- 能够启用新型智能互联网设备的.NET 设备软件。
- .NET 用户体验 。

1.2 .NET 与 C#

1.2.1 支持多种编程语言的.NET 结构框架

让我们翻开教科书，回顾一下近十年来软件开发的历史。

多年以前，当微软的组件对象模型（Component Object Model, COM）尚未推出时，软件的复用性对于开发人员仅仅是一种美好的憧憬。成千上万的程序员为了处理通信、接口和不同语言间的冲突而通宵达旦地艰辛劳动，但却收效甚微。COM 的出现改变了

这一切。通过将组件改变为通用、集成型的构件，开发人员正逐渐地从过去的繁复编程事务中解脱出来，可以选择自己最得心应手的编程语言进行编程。然而，软件组件与应用程序之间的联合仍然是松散的，不同的编程语言与开发平台限制了部件间的互用性，其结果是产生了日益庞大的应用程序与不断升级的软硬件系统。举个很简单的例子，只用五行 C 语言代码就能编写出的一个简单程序，若使用 COM 来编写，结果会是令人吃惊的：我们需要几百行代码。COM 在带来巨大价值的同时，也大大增加了开发开销。而.NET Framework 的出现使得一切问题都迎刃而解。实际上，在.NET Framework 中，所有的编程语言，从相对简单的 JScript 到复杂的 C++语言，一律是等同的。

Framework——框架，是开发人员对编程语言命令集的称呼。.Net 框架的意义就在于只用统一的命令集支持任何的编程语言。正如微软 Web 服务中心的成组产品经理 John Montgomery 所说：“只需简单地一用，.NET 框架便可消除各种异类框架之间的差异，将它们合并为一个整体。.NET 的作用不仅仅是将开发人员从必须掌握多种框架的束缚中解脱出来，通过创建跨编程语言的公共 API 集，.NET 框架可提供强大的跨语言继承性、错误处理和调试功能。现在，开发人员可以自由地选择他们喜欢的编程语言。.NET 平台欢迎所有人的垂顾。”.NET 将使编程人员梦想的语言互用性变成为近在眼前的现实。想想看，一个在 Visual Basic（VB）中定义的类能够在另一种与它完全不同的语言环境中使用、调试，甚至继承，这是多么令人兴奋的事情！

.NET 框架是.NET 平台的基础架构。其强大功能来自于公共语言运行时（Common Language Runtime,CLR 将在第二章中进行详细的解释）环境和类库。CLR 和类库（包括：Windows Forms，ADO.NET 和 ASP.NET）紧密结合在一起，提供了不同系统之间交叉与综合的解决方案和服务。.NET 框架创造了一个完全可操控的、安全的和特性丰富的应用执行环境。这不但使得应用程序的开发与发布更加简单，并且成就了众多种类语言间的无缝集成。

1.2.2 面向.Net 的全新开发工具——C#

在最近的一段时间里，C 和 C++一直是最有生命力的程序设计语言。这两种语言为程序员提供了丰富的功能，高度的灵活性和强大的底层控制能力。而这一切都不得不在效率上作出不同程度的牺牲。如果你使用过包括 C 和 C++在内的多种程序设计语言，相信你会深刻体会到它们之间的区别。比如与 Visual Basic 相比，Visual C++程序员为实现同样的功能就要花费更长的开发周期。由于 C 和 C++即为我们带来了高度的灵活性，又使我们必须要忍受学习的艰苦和开发的长期性，许多 C 和 C++程序员一直在寻求一种新的语言，以图在开发能力和效率之间取得更好的平衡。

今天，人们改进、开发出了许多语言以提高软件生产率，但这些或多或少都以牺牲 C 和 C++程序员所需要的灵活性为代价。这样的解决方案在程序员身上套上了太多的枷锁，限制了他们能力的发挥。它们不能很好地与原有的系统兼容，更为令人头痛的是，它们并不总是与当前的 Web 应用结合得很好。

理想的解决方案，是将快速的应用开发与对底层平台所有功能的访问紧密结合在

一起。程序员们需要一种环境：它与 Web 标准完全同步，并且具备与现存应用间方便地进行集成的能力。除此之外，程序员们喜欢它允许自己在需要时使用底层代码。

针对该问题，微软的解决方案是一种称之为 C#的程序语言。C#是一种现代的面向对象的程序开发语言，它使得程序员能够在新的微软.NET 平台上快速开发种类丰富的应用程序。.NET 平台提供了大量的工具和服务，能够最大限度地发掘和使用计算及通信能力。

由于其一流的面向对象的设计，从构建组件形式的高层商业对象到构造系统级应用程序，你都会发现，C#将是最合适的选择。使用 C#语言设计的组件能够用于 Web 服务，这样通过 Internet，可以被运行于任何操作系统上任何编程语言所调用。

不但如此，C#还能为 C++程序员提供快捷的开发方式，又没有丢掉 C 和 C++的基本特征——强大的控制能力。C#与 C 和 C++有着很大程度上的相似性，熟悉 C 和 C++的开发人员很快就能精通 C#。

1.3 C#语言的特点

C#在带来对应用程序的快速开发能力的同时，并没有牺牲 C 与 C++程序员所关心的各种特性。它忠实地继承了 C 和 C++的优点。如果你对 C 或 C++有所了解，你会发现它是那样的熟悉。即使你是一位新手，C#也不会给你带来任何其它的麻烦，快速应用程序开发（Rapid Application Development，RAD）的思想与简洁的语法将会使你迅速成为一名熟练的开发人员。

正如前文所述，C#是专门为.NET 应用而开发出的语言。这从根本上保证了 C#与.NET 框架的完美结合。在.NET 运行库的支持下，.NET 框架的各种优点在 C#中表现得淋漓尽致。让我们先来看看 C#的一些突出的特点，相信在以后的学习过程中，你将会深深体会到“#”——“SHARP”的真正含义。

- 简洁的语法
- 精心地面向对象设计
- 与 Web 的紧密结合
- 完整的安全性与错误处理
- 版本处理技术
- 灵活性与兼容性

1.3.1 简洁的语法

请原谅，虽然我们一再强调学习本书不需要任何的编程基础，但在这里还不得不提到 C++。

在缺省的情况下，C#的代码在.NET 框架提供的“可操控”环境下运行，不允许直接地内存操作。它所带来的最大特色是没有了指针。与此相关的，那些在 C++中被疯狂使用的操作符（例如：“::”、“->”和“.,”）已经不再出现。C#只支持一个“.”，对

于我们来说，现在需要理解的一切仅仅是名字嵌套而已。

C#用真正的关键字换掉了那些把活动模板库（Active Template Library，ALT）和COM 搞得乱糟糟的伪关键字，如 OLE_COLOR、BOOL、VARIANT_BOOL、DISPID_XXXXX 等等。每种 C#类型在.NET 类库中都有了新名字。

语法中的冗余是 C++中的常见的问题，比如“const”和“#define”、各种各样的字符类型等等。C#对此进行了简化，只保留了常见的形式，而别的冗余形式从它的语法结构中被清除了出去。

1.3.2 精心地面向对象设计

也许你会说，从 Smalltalk 开始，面向对象的话题就始终缠绕着任何一种现代程序设计语言。的确，C#具有面向对象的语言所应有的一切特性：封装、继承与多态，这并不出奇。然而，通过精心地面向对象设计，从高级商业对象到系统级应用，C#是建造广泛组件的绝对选择。

在 C#的类型系统中，每种类型都可以看作一个对象。C#提供了一个叫做装箱（boxing）与拆箱（unboxing）的机制来完成这种操作，而不给使用者带来麻烦，这在以后的章节中将进行更为详细的介绍。

C#只允许单继承，即一个类不会有多个基类，从而避免了类型定义的混乱。在后面的学习中你很快会发现，C#中没有了全局函数，没有了全局变量，也没有了全局常数。一切的一切，都必须封装在一个类之中。你的代码将具有更好的可读性，并且减少了发生命名冲突的可能。

整个 C#的类模型是建立在.NET 虚拟对象系统（Visual Object System，VOS）的基础之上，其对象模型是.NET 基础架构的一部分，而不再是其本身的组成成分。在下面将会谈到，这样做的另一个好处是兼容性。

借助于从 VB 中得来的丰富的 RAD 经验，C#具备了良好的开发环境。结合自身强大的面向对象功能，C#使得开发人员的生产效率得到极大的提高。对于公司而言，软件开发周期的缩短将能使它们更好地应付网络经济的竞争。在功能与效率的杠杆上人们终于找到了支点。

1.3.3 与 Web 的紧密结合

.NET 中新的应用程序开发模型意味着越来越多的解决方案需要与 Web 标准相统一，例如超文本标记语言（Hypertext Markup Language，HTML）和 XML。由于历史的原因，现存的一些开发工具不能与 Web 紧密地结合。SOAP 的使用使得 C#克服了这一缺陷，大规模深层次的分布式开发从此成为可能。

由于有了 Web 服务框架的帮助，对程序员来说，网络服务看起来就像是 C#的本地对象。程序员们能够利用他们已有的面向对象的知识与技巧开发 Web 服务。仅需要使用简单的 C#语言结构，C#组件将能够方便地为 Web 服务，并允许它们通过 Internet 被运行在任何操作系统上的任何语言所调用。举个例子，XML 已经成为网络中数据结构传送的标准，为了提高效率，C#允许直接将 XML 数据映射成为结构。这样就可以有

效地处理各种数据。

1.3.4 完全的安全性与错误处理

语言的安全性与错误处理能力，是衡量一种语言是否优秀的重要依据。任何人都会犯错误，即使是最熟练的程序员也不例外：忘记变量的初始化，对不属于自己管理范围的内存空间进行修改，……。这些错误常常产生难以预见的后果。一旦这样的软件被投入使用，寻找与改正这些简单错误的代价将会是让人无法承受的。C#的先进设计思想可以消除软件开发中的许多常见错误，并提供了包括类型安全在内的完整的安全性能。为了减少开发中的错误，C#会帮助开发者通过更少的代码完成相同的功能，这不但减轻了编程人员的工作量，同时更有效地避免了错误发生。

.NET 运行库提供了代码访问安全特性，它允许管理员和用户根据代码的 ID 来配置安全等级。在缺省情况下，从 Internet 和 Intranet 下载的代码都不允许访问任何本地文件和资源。比方说，一个在网络上的共享目录中运行的程序，如果它要访问本地的一些资源，那么异常将被触发，它将会无情地被异常扔出去，若拷贝到本地硬盘上运行则一切正常。内存管理中的垃圾收集机制减轻了开发人员对内存管理的负担。.NET 平台提供的垃圾收集器（Garbage Collection，GC）将负责资源的释放与对象撤销时的内存清理工作。

变量是类型安全的。C#中不能使用未初始化的变量，对象的成员变量由编译器负责将其置为零，当局部变量未经初始化而被使用时，编译器将做出提醒；C#不支持不安全的指向，不能将整数指向引用类型，例如对象，当进行下行指向时，C#将自动验证指向的有效性；C#中提供了边界检查与溢出检查功能。

1.3.5 版本处理技术

C#提供内置的版本支持来减少开发费用，使用 C#将会使开发人员更加轻易地开发和维护各种商业应用。

升级软件系统中的组件（模块）是一件容易产生错误的工作。在代码修改过程中可能对现存的软件产生影响，很有可能导致程序的崩溃。为了帮助开发人员处理这些问题，C#在语言中内置了版本控制功能。例如：函数重载必须被显式地声明，而不会像在 C++或 Java 中经常发生的那样不经意地被进行，这可以防止代码级错误和保留版本化的特性。另一个相关的特性是接口和接口继承的支持。这些特性可以保证复杂的软件可以被方便地开发和升级。

1.3.6 灵活性和兼容性

在简化语法的同时，C#并没有失去灵活性。尽管它不是一种无限制的语言，比如：它不能用来开发硬件驱动程序，在默认的状态下没有指针等等，但是，在学习过程中你将发现。它仍然是那样的灵巧。

如果需要，C#允许你将某些类或者类的某些方法声明为非安全的。这样一来，你

将能够使用指针、结构和静态数组，并且调用这些非安全的代码不会带来任何其它的问题。此外，它还提供了一个另外的东西（这样的称呼多少有些不敬）来模拟指针的功能——delegates，代表。再举一个例子：C#不支持类的多继承，但是通过对接口的继承，你将获得这一功能。

下面谈谈兼容性。

正是由于其灵活性，C#允许与 C 风格的需要传递指针型参数的 API 进行交互操作，DLL 的任何入口点都可以在程序中进行访问。C#遵守.NET 公用语言规范（Common Language Specification，CLS），从而保证了 C#组件与其它语言组件间的互操作性。元数据（Metadata）概念的引入既保证了兼容性，又实现了类型安全。

1.4 小　　结

Microsoft.NET 计划将彻底改变我们对因特网的认识，从而在这样一个网络时代彻底改变我们的生活。软件是一种服务，技术是我们的仆人，时间与地点将不再是我们面前的障碍。建立在 CLR 与类库基础上的.NET 框架是.NET 平台的核心组件之一，这为软件的可移植性与可扩展能力奠定了坚实的基础，并为 C#语言的应用创造了良好的环境。

C#是.NET 平台的通用开发工具，它能够建造所有的.NET 应用。其固有的特性保证了它是一种高效、安全、灵活的现代程序设计语言。从最普通的应用到大规模的商业开发，C#与.NET 平台的结合将为你提供完整的解决方案。

在本章中，我们提出了与.NET 以及与 C#语言相关的一些概念，例如 CLR、VOS 和 GC，也许你是初次接触它们，但不用担心，在以后的各章中我们将详细地介绍这些相关的概念与知识，相信通过学习，你将能够迅速掌握它们，并熟练地运用它们提供的各种特性。

复习题

（1）什么是.NET？

（2）简要说明.NET 战略的意义。

（3）.NET 的核心组件包括哪些？

（4）C#与其它语言相比有哪些突出特点？

第二章　运行环境——全面了解.NET

C#运行在.NET 平台之上，其各种特性与.NET 密切联系。它没有自己的运行库，许多强大的功能均来自.NET 平台的支持。因此，要想真正掌握 C#首先必须了解.NET。本章将向你介绍 C#的运行环境，重点放在.NET 公用语言运行时环境与公用语言规范上，最后介绍了.NET 的开发工具。

2.1　.NET 结构

.NET 包括四个组成部分：

- VOS 类型系统
- 元数据
- 公用语言规范
- 虚拟执行系统

下面分别对它们进行简要介绍。

2.1.1　虚拟对象系统

.NET 跨语言集成的特性来自于虚拟对象系统（VOS）的支持。

在不同语言间进行代码复用和应用集成中所遇到的最大问题，是不同语言类型系统间的相容性问题。可以想象，不同的语言虽然语法结构大体相同，但数据类型与语言环境本身的各种特点联系紧密，很难想象一种解释性的语言所拥有的数据类型会与一种编译语言相同，而即使相同的数据类型在不同的语言环境中表示的意义也存在差别。例如，同样是整数类型，在 MSSQL 中的长度是 32 位，而在 VB 中却是 16 位，至于日期时间与字符串类型在这方面的区别就更加明显了。

VOS 的建立就是为了改变这种状况。它既支持过程性语言也支持面向对象的语言，同时提供了一个类型丰富的系统来容纳它所支持的各种语言的特性。它在最大程度上屏蔽了不同语言类型系统间的转换，使程序员能够随心所欲地选择自己喜欢的语言（当然，这种语言必须支持.NET 应用）从事开发，保证了不同语言间的集成。

对于过程性语言，它描述了值的类型并指定了类型的所有值必须遵守的规则；在面向对象的语言方面，它统一了不同编程语言的对象模型。每一个对象在 VOS 中都被唯一标识以与其它对象相区别。

2.1.2 元数据

元数据是对 VOS 中类型描述代码的一种称呼。在编译程序将源代码转换成为中间代码时，它将自动生成，并与编译后的源代码共同包含在二进制代码文件中。元数据携带了源代码中类型信息的描述，这在一定程度上解决了版本问题：程序使用的类型描述与其自身绑定在一起。

在 CLR 定位与装载类型时，系统通过读取并解析元数据来获得应用程序中的类型信息，JIT 编译器获得加载的类型信息后，将中间语言代码翻译成为本地代码，在此基础上根据程序或用户要求建立类型的实例。由于整个过程中，CLR 始终根据元数据建立并管理对应特定应用程序的类型，从而保证了类型安全性。

此外，元数据在解决方法的调用，建立运行期上下文界限等方面都有着自己的作用。而关于元数据的一切都由.NET 在后台完成。

2.1.3 公用语言规范

公用语言规范（Common Language Specification，CLS），是 CLR 定义的语言特性集合，主要用来解决互操作问题。如果一个类库遵守 CLS，那么同样遵守 CLS 规范的其它编程语言将能够使用它的外部可见项。详细的内容见本章第二节。

2.1.4 虚拟执行系统

虚拟执行系统（Visual Execution System，VES）是 VOS 的实现，它用来驱动运行环境。元数据的生成与使用、公用语言规范的满足性检查以及应用程序执行过程中的内存管理均由它来完成。具体说来，VES 主要完成以下功能：

- 装入中间代码。
- 使用 JIT 将中间代码转换为本地码。
- 装入元数据。
- 代码管理服务——包括垃圾收集器和异常处理。
- 定制与调试服务。
- 线程和环境管理。

2.2 公用语言运行时环境与公用语言规范

了解了.NET 的结构之后，我们该看看.NET 利用其结构为我们创造的运行环境——公用语言运行时环境。它是 C#及其它支持.NET 平台的开发工具的运行基础。具体来说，它为我们的应用提供了以下益处：

- 跨语言集成的能力。
- 跨语言异常处理。
- 内存管理自动化。

- 强化的安全措施。
- 版本处理技术。
- 组件交互的简化模型。

2.2.1 理解 CLR

.NET 提供了一个运行时环境，叫做公用语言运行时，它管理着代码的执行，并使得开发过程变得更加简单。这是一种可操控的执行环境，其功能通过编译器与其它工具共同展现，你的代码将受益于这一环境。依靠一种以运行时为目标的（指完全支持运行时环境的）编译器所开发的代码叫做可操控代码。它得益于可操控环境的各种特性：跨语言集成、跨语言异常处理、增强的安全性、版本处理与开发支持、简单的组件交互模型以及调试服务。为了使运行时环境能够向可操控代码提供服务，语言编译器需要产生一种元数据，它将提供在你使用语言中的类型、成员、引用的信息。元数据与代码一起存储，每个可加载的 CLR 映像均包含了元数据。运行时环境使用元数据定位并载入类，在内存中展开对象实例，解决方法调用，产生本地代码，强制执行安全性，并建立运行时环境的边界。

运行时环境自动处理对象的展开与引用，当它们不再使用时负责它们的释放。被运行时环境进行这样的生命期管理的对象被称为可操控代码。自动内存管理消除了内存溢出，同时也解决了其它一些常见的语法错误。如果你的代码是可操控的，你仍然可以在需要的时候使用非可控代码，或者在你的.NET 应用中同时使用可控与非可控代码。由于语言编译器支持他们自己的类型，比如一些原始类型，你可能并不总是知道（也不必知道）你的数据是否是可控的。

CLR 使设计跨语言的组件与应用变得更加容易。以不同语言设计的对象能够彼此间进行通信，并且它们的行为能够紧密地综合与协调。举个例子，你定义了一个类，然后可以在另一种不同的语言中从该类中派生了一个类或者调用它其中的一个方法。你也可以向另一种语言中类的方法传递该类的一个实例。这种跨语言的集成之所以可能，因为以运行时间为目标的语言编译器与工具使用一种运行时间所定义的公用类型系统，他们遵守运行时的规则（公用语言规范）来定义新的类型，生成、使用、保持并绑定类型。

作为元数据的一部分，所有可控组件携带了关于它们所依赖的组件与资源的信息。运行时环境使用这些信息来保证你的组件或应用具有需要的所有东西的特定版本，其结果是你的代码将不会因为版本冲突而崩溃。注册信息与状态数据不再保存在难以建立与维护的注册表中，你所定义的类型及附属信息作为元数据被保存，这使得复制与移动组件的复杂程度得到降低。

编译工具用他们自己的方式向开发人员展现 CLR 的功能。这意味着运行时间的一些特性可能在不同的语言中的表现形式将会有所不同。你怎样体验运行时的特性将取决于你所使用的语言，比如说，如果你是一位 VB 开发人员，你可能注意到在运行时环境的帮助下，VB 语言比以前具有更多的面向对象的特性。

2.2.2 可操控执行的含义

前面的叙述中，我们多次提到了“可操控”这一概念。这意味着它指向的对象在执行过程中完全被运行时环境所控制。在执行过程中，运行时环境提供以下服务：自动内存管理、调试支持、增强的安全性及与非可操控代码的互操作性，例如 COM 组件。

在可控执行进程中的第一步是选择源代码的生成工具。如果你希望你的应用拥有 CLR 提供的优势，你必须使用一种（或多种）以运行时为目标的语言编译器，例如：VB、C#、VC 的编译器，或者一种第三方编译器如 PERL 或 COBOL 编译器。

由于运行时是一种多语言执行环境，它支持众多的数据类型和语言特性。你使用的语言编译器决定你将使用运行时的哪一部分功能子集。在代码中使用的语法由你的编译器决定，而不是运行时环境。如果你的组件需要被其他语言的组件完全使用，那么你必须在你组件的输出类型中使用 CLR 所要求的语言特征。

当你完成并编译你的代码时，编译器将它转换为微软中间语言（Microsoft Intermediate Language，MSIL），同时产生元数据。当你要执行你的代码时，这种中间语言被即时（Just In Time，JIT）编译器编译成为本地代码。如果安全策略需要的代码是类型安全的——通常情况下都是如此——JIT 编译器将在编译进程中对中间语言进行类型检查。一旦失败，在代码执行中将会触发异常。

2.2.3 CLR 的突出特色

跨语言集成的能力

CLR 包含了一个丰富的语言特性集，保证了它与各种程序设计语言的兼容性。这一特性集即公用语言规范，稍后将对其进行详细说明。

内存管理自动化

在执行过程中管理应用程序的资源是一项单调而困难的工作。它会将你的注意力从你本应解决的问题中引开。而垃圾收集机制完全解决了程序员在编程过程中头痛的问题，跟踪内存的使用，并知道何时将它们释放。

在面向对象的环境中，每种类型都标识了对你的应用有用的某种资源。为了使用这些资源，你需要为类型分配内存。在应用中，访问一种资源要通过以下步骤：

（1）为类型分配内存。

（2）初始化内存，设置资源的初始状态并使其可用。

（3）通过访问该类型的实例成员来访问资源。

（4）卸下将被清除的资源状态。

（5）释放内存。

这一看似简单的过程在实际的编程中是产生程序错误的主要来源之一。更可怕的是：内存中的错误往往导致不可预见的结果。如果你有过编程的经验，想想看，有多少次你的程序因为内存访问错误而崩溃？

CLR 要求所有的资源从可操控的堆（注：在此指一种内存结构）中分配。当一个

进程被初始化后，CLR 保留了一个未被分配的地址空间。这一区域叫做可操控堆。在堆中保持了指向下一个将被分配给对象的堆地址的指针（NEXT）。初始状态下，该指针是保留地址空间的基地址。一个应用使用新的操作产生对象。此操作首先检查新对象需要字节的大小是否会超出保留空间。如果对象大小合适，指向下一个地址的指针将指向堆中的这个对象，该对象的构造器被调用，新的操作返回对象的地址。

当一个应用请求建立一个对象时，地址空间可能不够大。堆将发现这一点（通过将新对象的大小与 NEXT 指针相加，并与堆的大小进行比较），这时垃圾收集器就将被调用。在这里，CLR 引入了“代”的概念。代，指堆中对象产生的先后。这样，垃圾收集器在将发生溢出时回收属于特定的“代”的对象，而不是回收堆中的所有对象。

（6）即时编译。

在各种语言的编译器对源代码进行编译之后，在 CLR 环境中产生的是中间代码（出于兼容性与跨语言集成的考虑），其内容虽然有效，但在转化为本地代码之前它本身是不可执行的。这就是 JIT 编译器需要完成的工作。

这里需要说明一个问题：为什么要即时编译，而不是一次性的将中间代码文件进行编译？答案很简单：原因在于效率。在大型的应用中，你很少会用到程序的全部功能，这种边执行边编译的措施比一次性的完全编译效率更高。

在 Windows 平台中，CLR 带有三个不同的 JIT 编译器：

（7）缺省的编译器——主编译器，由它进行数据流分析并输出经过优化的本地代码，所有的中间代码指令均可被它处理。

（8）PREJIT，它建立在主 JIT 编译器之上。其运行方式更像一个传统的编译器：每当一个.NET 组件被安装时它就运行。

（9）ECONOJIT，在并不充分优化的前提下，它能够快速完成 IL 代码到本地码的转换，编译速度与运行速度都很快。

为了配合编译器的工作，在.NET SDK 的安装路径下的/bin 目录中有一个负责管理 JIT 的应用程序：jitman.exe。具体的使用参见联机帮助。

（10）解决版本与发布问题。

在当前以组件为基础的系统中，开发人员和用户对于软件版本和发布中存在的问题已经十分熟悉了。当我们安装一个新的应用之后，我们很可能发现原本正常的某个应用程序奇怪地停止了工作。绝大多数开发人员将时间花在了确保所有注册表入口的一致性，以便激活 COM 类上。这就是所谓的“DLL 地狱”。

.NET 平台通过使用集合来解决这一问题。在这里，“集合”是一个专有名词，指类型与资源的发布单元，在很大程度上它等同于今天的 DLL。正像.NET 用元数据描述类型一样，它也用元数据描述包含类型的集合。通常说来，集合由四个部分组成：集合的元数据（集合的内部清单）、元数据描述的类型、实现类型的中间语言代码和一组资源。在一个集合中，以上四个部分并不是都必须存在，但是，集合中必须包含类型或资源，这样集合才有意义。

在.NET 中一个基本的设计方针是使用孤立的组件。一个孤立的集合的含义是指一个集合只能被一个应用所访问。在一台机器上，它不被多个应用共享，也不会受其它应用程序对系统的更改的影响。“孤立”赋予了开发人员在自己的程序中对代码的完全

控制权。任何共享代码都需要被明确地标识。同时，.NET 框架也支持共享集合的概念。一个共享集合指在一台机器上被多个应用共享的集合。共享集合需要严格地命名规定。有了.NET，应用程序间的共享代码是明确定义的。共享集合需要一些额外的规则来避免我们今天遇到的共享冲突问题。共享代码必须有一个全局唯一的名称，系统必须提供名称保护，并在每当引用共享集合时，CLR 将对版本信息进行检查，此外.NET 框架允许应用或管理员在明确说明的版本政策下重写集合的版本信息。

2.2.4 公用语言规范

使被不同语言的编译器所编译的对象能够相互理解的唯一方法，是所有在互操作过程中涉及的数据类型和语言特性对所有的语言来说是公共的。为了这个目的，公用运行时环境标识了一组语言特征的集合，称为公用语言规范（CLS）。如果你的组件在应用程序接口（Application Program Interface）中仅使用 CLS 的特征语言（包括子类），那么该组件能够被任何支持 CLS 的语言所编译的组件访问。所有支持 CLS 并仅使用 CLS 中的语言特征的组件被称为符合 CLS 的组件。

设计公用语言规范时遇到的一个最主要的挑战是选择适当的语言特性子集的大小。它应具有完全的表达能力，又应足够小，使得所有的语言能够容纳它。由于 CLS 是关于语言互用性的规范，它的规则仅应用于外部可见的条目中。CLS 假设语言间的互操作性仅在语言集合的边界发生交叉时才是重要的。也就是说，在单一的语言集中对于编程技术的使用没有任何限制。CLS 的规则仅作用于在定义它们的语言集合之外仍然可见的项上。这样就大大缩小了 CLS 的范围，减轻了系统的负担。

在 CLS 中是用 System.CLSCompliantAttribute 类来标识一个集合或者类是否是符合 CLS 规范的：在 System.CLSCompliantAttribute 的构造器中有一个 Boolean 型的返回值，代表了与之相关联的项是否符合 CLS 规范。

2.3 开 发 工 具

.NET 为使用与开发人员提供了功能强大、种类丰富的管理与开发工具，同时它们也是.NET 框架提供的服务，我们将它们列在下面，正是由于有了它们的支持.NET 才变得如此强大：

1. Visual Studio.NET：是.NET 的核心开发工具，包括微软提供的各种开发语言，其中有 Visual C#。

2. Assembly Generation Utility (al.exe)：用来建立集合的工具。它能够将资源文件或 MSIL 格式的文件转换为带有内容清单的集合。

3. Windows Forms ActiveX Control Importer (aximp.exe)：完成 COM 类库中类型定义的转换，使 ActiveX 控件能够在 Windows 窗口控件上使用。

4. Code Access Security Policy Utility (caspol.exe)：在用户与机器水平上修改安全策略。

5. Software Publisher Certificate Test Utility (Cert2spc.exe)：用于从 X.509 证书中生成软件出版证明书（SPC）。

6. Certificate Manager Utility (certmgr.exe)：管理证书、证书信任列表和证书回收列表。

7. Certificate Verification Utility (chktrust.exe)：检查证书签名的合法性。

8. Runtime Debugger (cordbg.exe)：运行时调试器，是一个命令行程序，帮助开发人员发现和调试基于 CLR 的应用程序中的错误。

9. Global Assembly Cache Utility (gacutil.exe)：允许你浏览与操纵全局集合缓存中内容的命令行程序。

10. MSIL Assembler (ilasm.exe)：MSIL 汇编程序，协助设计与实现 MSIL 生成器的程序。

11. MSIL Disassembler (ildasm.exe)：MSIL 反汇编程序，与 ilasm.exe 共同使用，将由 MSIL 代码产生的 Portable Executable 文件转换为文本文件。

12. Installer Utility (installutil.exe)：用来安装与卸载服务资源。

13. License Compiler (lc.exe)：产生可包含在可执行二进制文件中的二进制资源文件。

14. Certificate Creation Utility (makecert.exe)：生成 X.509 证书与用于数字签名的公用与私有密钥。

15. Permissions View Utility(permview.exe)：通过一个集合浏览许可集的工具。

16. Peverify Utility(peverify.exe)：检查中间语言与元数据是否符合类型安全认证要求。

17. Assembly Registration Tool(RegAsm.exe)：读取集合中的元数据并加上必要注册表入口信息，使得 COM 客户透明地建立 CLR 的类。

18. Services Registration Tool (RegSvcs.exe)：服务注册工具，它完成执行以下功能：装载与注册一个集合，为现有的 COM+1.0 应用生成、注册与安装类库。

19. Resource File Generator Utility(ResGen.exe)：资源文件生成器，用来将文本文件和 XML 格式的资源文件转换为 CLR 的二进制文件。

20. Secutil Utility(SecUtil.exe)：使得从集合中抽取的安全信息更加容易。

21. Set Registry Utility(setreg.exe)：改变注册表中公开密钥密码系统的设置。

22. Assembly Cache Viewer(shfusion.dll)：允许你使用 Windows 浏览器察看与操作全局集合缓存中的内容。

23. File Signing Utility(signcode.exe)：为 PE (portable executable)文件做标记。赋予程序员在组件安全约束的基础上对安全性有更多的控制权。

24. Shared Name Utility(Sn.exe)：帮助程序员以共享名称建立集合。

25. Soapsuds Utility(SoapSuds.exe)：使用远程技术帮助你编译与 Web 服务相通信的客户应用。

26. Isolated Storage Utility(storeadm.exe)：一种用来管理隔离存储区的命令行工具。

27. Type Library Exporter(TlbExp.exe)：命令行程序，生成由集合名称指示的包含集合中公共类型定义的类库。

28. Type Library Importer (TlbImp.exe)：将 COM 类库中的类型定义转换为在 CLR 中与元数据格式一致的类型定义。

29. Web Service Utility(WebServiceUtil.exe)：帮助建立 ASP.NET Web 服务与客户。

30. Windows Forms Class Viewer(wincv.exe)：能够在某种查找模式下快速查找类或者类序列的信息。

31. Windows Forms Designer Test Container(windes.exe)：允许开发人员测试开发出的视窗窗体控件在设计时的行为。

32. XML Schema Definition Tool(xsd.exe)：XML 计划定义工具。

2.4 小　　结

本章解释了与.NET 有关的概念并简要介绍了一些相关的技术。在了解了.NET 的结构之后，我们重点讨论了公用语言运行时环境和公用语言规范，最后给出了.NET 开发工具的清单。

在完成本章的学习之后，你已经了解了有关 C#运行环境的相关知识，这将为你深入学习 C#打下良好的基础。从下一章开始，我们将进入实际的编程实践中，您将会发现关于 C#的更多更有趣的东西。

复习题

（1）.NET 的结构由哪四部分组成？

（2）请简要总结 CLR 的作用。

（3）“可操控执行”的含义是什么？

（4）.NET 是怎样解决传统 Windows 程序设计中 DLL 的版本问题的？

（5）什么是 CLS，它的范围是怎样确定的？

第三章　编写第一个应用程序

介绍了 C#语言的这么多优点，您可能已经有些不耐烦了。好，那就让我们开始 C#的开发之路吧。

本章介绍如何生成您的第一个 C#程序。这是一个最基本的 C#应用程序，程序中的代码在全书中将经常出现。

我一直坚信，只有不断练习才是最好的学习方式。所以建议读者从本章开始，对书中所提供的程序示例，亲自进行编辑、编译和运行，在这个过程中，您将获得开发 C#程序的有益经验。

3.1　Welcome 程序

可以这么说，与用户没有任何交互的应用程序根本没有任何用处（病毒和黑客当然除外。然而即使是病毒程序的作者，也常常喜欢在自己得逞之后炫耀一番）。学习任何一门语言，绝大多数情况下人们都是从输入输出开始的。

第一个程序总是非常简单的。我们让用户通过键盘输入自己的名字，然后程序在屏幕上打印一条欢迎信息。程序的代码是这样的：

程序清单 3-1：

```
using System;
class Welcome
{
   static void Main() {
      Console.WriteLine("Please enter your name:");
      Console.ReadLine();
      Console.WriteLine("Welcome to you!");
   }
}
```

您可以在任意一种编辑软件中完成上述代码的编写，然后把文件存盘，文件名叫做 Welcome.cs。典型的 C#源文件通常都是以“.cs”作为文件的扩展名。

3.2　代 码 分 析

首先要提出的是，C#语言是大小写敏感的。这一点对于 C 和 C++程序员没什么问

题，只是要提醒一下 VB 和 Delphi 的程序员。

接下来让我为您逐条地分析上面的 C#程序语句。

3.2.1 名字空间

using System 表示导入名字空间。高级语言总是依赖于许多系统预定义的元素。如果您是 C 或 C++的程序员，那么您一定对使用#include 之类的语句来导入其它 C 或 C++源文件再熟悉不过了。C#中的含义与此类似，用于导入预定义的元素，这样在自己的程序中就可以自由地使用这些元素。

如果没有导入名字空间的话，我们该怎么办呢，程序还能保持正确吗？答案是肯定的。那样的话，我们就必须把代码改写成下面的样子：

程序清单 3-2：

```
class Welcome

{
  static void Main() {

        System.Console.WriteLine("Please enter your name:");

        System.Console.ReadLine();
         System.Console.WriteLine("Welcome to you!");
  }

}
```

也就是说，在每个 Console 前加上一个前缀“System.”。这个小原点“.”表示 Console 是作为 System 的成员而存在的。C#中抛弃了 C 和 C++中繁杂且极易出错的操作符像“::”和“->”等。C#中的复合名字一律通过“.”来连接。

System 是.Net 平台框架提供的最基本的名字空间之一。有关名字空间的详细使用方法我们将放在第十七章中详细介绍。在这里，只要我们学会怎样导入名字空间就足够了。

3.2.2 类和类的方法

让我们从写第一个程序时就记住：每个东西都必须属于一个类。如果您是 C 或 C++的程序员，请暂时忘掉那些全局变量。

在程序的第二行，class Welcome 声明了一个类，类的名字叫做 Welcome。这个程序为我们所作的事情就是依靠它来完成的。

和 C、C++中一样，源代码块被包含在一对大括号“{”和“}”中。每一个右括号“}”总是和它前面离它最近的一个左括号“{”相配套。如果左括号“{”和右括号“}”没有全部配套，那程序就是一个错误的程序。

static void Main()表示类 Welcome 中的一个方法。方法总是为我们完成某件工作的。

注意：在 C#程序中，程序的执行总是从 Main()方法开始的。一个程序中不允许出

现两个或两个以上的 Main()方法。对于习惯了写 C 控制台程序的读者，请牢记：C#中 Main()方法必须被包含在一个类中。

3.2.3 程序的输入和输出

程序所完成的输入输出功能都是通过 Console 来完成的。Console 究竟是什么呢？它是在名字空间中 System 已经为我们定义好的一个类，这里我们不用管它是怎么完成工作的，只要使用它就可以了。

上面的代码中，类 Console 为我们展现了两个最基本的方法：WriteLine 和 ReadLine。Console.ReadLine 表示接受输入设备输入，Console. WriteLine 则用于在输出设备上输出。

我们再为读者介绍 Console 中用于输入输出的另两个方法：Read 和 Write。它们和 ReadLine 与 WriteLine 的不同之处在于，ReadLine 和 WriteLine 执行时相当在显示时多加了一个回车键，而使用 Read 和 Write 时则光标不会自动转移到下一行。

让我们再对例子程序进行扩展，使得用户的输入对输出产生作用。

程序清单 3-3：

```
using System;

class Welcome
{
  static void Main() {

        Console.WriteLine("Please enter your name:");

        string name = Console.ReadLine();
         Console.WriteLine("Welcome to you,{0}!",name);
  }

}
```

我们用到了 string name = Console.ReadLine()这条语句。其中 string name 表示声明一个字符串类型的变量 name。系统定义的 Console 类提供的方法 ReadLine()的返回值类型为 string。所以，这句话表示从输入设备读取一个字符串，并把读取的值赋予变量 name。

再来看一下程序的最后一条输出语句：

```
Console.WriteLine("Welcome to you,{0}!",name);
```

这条语句表示在屏幕上对输出的字符串进行格式化。其中表示用方法的第二个参数来替代格式化后字符串相应的位置。对字符串进行格式化的参数可以是一个字符串，也可以是一个字符，或者是一个整数，等等。采用这种方式最多可以格式化三个变量。比如：

```
int x = 3;

string name1 = “Mike”;
```

```
string name2 = “John”;

Console.WriteLine("Welcome to you {0} times,{1} and {2}!",x,name1,name2);
```

和绝大多数编程语言一样，C#提供了字符串类型 string，它与 C 中的 MFC 为我们提供的类十分类似。C#中的 string 类型是一个引用类型（引用类型在第四章中我们有详细说明），为标准字符集。利用 string 可以方便地对字符串进行连接、截断等操作。比如，

```
string s = “Good” + “Morning”;

char x = s[3];
```

例子演示了字符串 s 由两个字符串“Good”和“Morning”相加得到。字符串还可以通过下标进行索引，得到一个字符。上面的例子中字符 x 的值为“o”。

所以，源程序 3-4 和源程序 3-3 的作用没什么区别：

程序清单 3-4：

```
using System;

class Welcome
{
  static void Main() {

      Console.WriteLine("Please enter your name:");

      string message = “Welcome to you “ + Console.ReadLine();
       Console.WriteLine(message);
  }

}
```

3.3 运 行 程 序

理解了源程序中每条代码的具体含义之后，下一步要做的就是让这个程序真正能够运行。不过对源代码即使还有不明白的地方也没有关系，在后续章节的学习中，您最终会熟练掌握这些概念的。

如果您的电脑上安装了 Visual Studio .Net，则可以在集成开发环境（Integrated Developer Environment，IDE）中直接选择快捷键或菜单命令，编译并执行源文件。

如果您不具备这个条件，那么您至少需要安装 Microsoft .Net Framework SDK，这样才能够不妨碍您在本书中继续学习 C#语言。实际上，.Net 平台内置了 C#的编译器。下面让我们使用这个微软提供的命令行编译器对我们的程序进行编译。

启动一个命令行提示符，在屏幕上输入一行命令：

```
csc   welcome.cs
```

我们假设您已经将 welcome.cs 文件保存在当前目录下。如果一切正常，welcome.cs 文件将被编译、运行，屏幕上出现一行字符，提示您输入姓名：

Please enter your name:

输入任意个字符并按下回车键，屏幕将打印出欢迎信息：

Welcome to you!

注意：和到目前为止我们使用过的绝大多数编译器不同，在 C#中，编译器只执行编译这个过程，而在 C 和 C++中要经过编译和链接两个阶段。换而言之，C#源文件并不被编译为目标文件（.obj），而是直接生成可执行文件（.exe）或动态链接库（.dll）。C#编译器中不需要包含链接器。

编译选项

我们可以灵活地使用 .Net 平台提供的命令行编译器的不同选项，选择不同的编译方式，从而灵活地对编译进行控制。

例如，如果我们希望对源文件 Welcome.cs 进行编译，生成名为 MyWelcome.exe 的可执行文件，我们可以采用这样的命令：

csc/out: MyWelcome.exe Welcome.cs

如果我们并不需要一个可执行文件，而仅仅是希望简单地检查源文件中是否存在语法错误，则命令可以写成：

csc/nooutput: Welcome.cs

如果不知道各个选项的具体含义，可以通过求助来获得：

csc/?

为方便读者，我们在表 3-1 中按字母排序的顺序列出了命令行编译器 csc 常用的参数及其用途。更详细的信息请参阅 C#联机帮助文档。

表 3-1　命令行编译器选项

选项	作用
@	指定响应文件
/?	列出编译命令选项
/addmodule	指定一个或多个模块作为装配的一部分
/baseaddress	指定载入动态链接库的首选地址
/bugreport	生成一个报告文件，其中包含程序 Bug 的详细信息
/checked	指定算术运算的溢出是否会导致程序在运行时抛出一个异常
/codepage	指定编译的所有源文件所使用的代码页
/debug	给出调试信息
/define	定义预处理程序的符号
/doc	由文件注释生成 XML 文件
/fullpaths	指定输出的绝对路径
/help	列出编译命令选项
/incremental	允许对源文件进行递增式编译
/linkresource	在装配时链接指定的 NET 资源

/main	指定 Main 方法所处的位置
/nologo	编译过程中不显示编译信息
/nooutput	编译源文件但不输出
/nostdlib	不导入标准库 (mscorlib.dll).

（续表）

选项	作用
/nowarn	编译过程中不生成警告信息
/optimize	指定编译时是否进行优化
/out	指定输出文件
/recurse	搜索子目录以寻找源文件
/reference	从包含装配的文件中导入元数据
/resource	把 NET 资源内嵌到输出文件
/target	指定输出文件的格式
/target:exe	输出文件为 exe 可执行文件
/target:library	输出文件为链接库
/target:module	输出文件为模块
/target:winexe	输出文件为 winexe 可执行文件
/unsafe	允许编译使用了不安全关键字的代码
/warn	设置警告级别
/warnaserror	把警告信息作为错误看待
/win32icon	把 .ico 图标文件插入到输出文件
/win32res	把 Win32 资源插入到输出文件

3.4 添 加 注 释

应用程序并不是只要你自己一个人能看懂就够了。不管以前计算机老师或者是编程书籍是否已经告诫过，这里我还要再一次强调：养成良好的代码注释的习惯。这是一名优秀的程序员必备的条件之一。代码注释不会浪费您的编程时间，它只会提高您的编程效率，使您的程序更加清晰、完整、友好。

C＃注释的方式和C++没有区别，每一行中双斜杠“／／”后面的内容，以及在分割符“／＊”和“＊／”之间的内容都将被编译器忽略。

这样，我们就可以采用“／／”进行单行注释，采用分割符“／＊”和“＊／”进行多行注释。让我们对 Welcome 程序加上注释：

程序清单 3-5：

```
／／源文件：welcome.cs
/＊说明：这里是我的第一个
C＃程序  ＊／
```

```
using System;

class Welcome
{
  static void Main() {

        Console.WriteLine("Please enter your name:");

        / / 要求用户输入姓名；

        Console.ReadLine();   / / 读取用户输入；
        Console.WriteLine("Welcome to you!");
        / * 本行代码用于打印欢迎信息；

             您可以在这里添加自己的代码；

            程序在这里结束 * /

    }

}
```

上面的注释似乎有些小题大做，但它毕竟说明了 C # 中注释的使用方法。

下面是对 C # 程序进行注释时要注意的两个问题：

首先，避免在“ / / ”之后的单行注解中使用反斜杠符号“\”。因为反斜杠符号“\”在 C # 中是一个续行符，这样做往往会导致你所不希望的结果出现。例如，当你写了类似于下面的代码：

```
Console.WriteLine(“The result is:{0}” , // \

        150 );
```

在编译这段代码时，“ / / ”表示逻辑上同一行剩余的所有文字被作为注释看待，而续行符“\”则将这一行同下一行连接起来，那么第二行也被作为注释的一部分。这时编译器找不到与第一行的左括号“(”相匹配的右括号“)”，因此编译出错。

其次，分割符“ / * ”和“ * / ”之间的注释不能有嵌套注释。这是因为， C # 编译器从遇到第一个分割符“ / * ”开始，将忽略下一个“ / * ”，直到遇上下一个与之匹配的分割符“ * / ”才认为注释结束，这样编译器就会对多余的“ * / ”报告错误，认为没有“ / * ”与之相匹配。

一些程序员经常习惯于把程序中不需要或暂时不需要的代码片段首尾分别加上分割符“ / * ”和“ * / ”。这时，如果这些代码片段本身又含有注释的话，我们所讨论的这种错误就会发生。这时，我们建议您换一种方式，把预处理语句 # if #endif 等贯穿在程序中使用，因为它们可以强制编译器忽略已包含注释的源代码片段，从而通过这种方式来实现源代码的嵌套注释。

以上提到的在注释中可能发生的两种问题，如果编译器报告错误，那已经是很幸运的了。如果是原本错误的程序被误认为是正确的加以编译，或者程序代码的原意因

此被曲解，造成的错误往往很难发现，那时造成的后果将会严重得多。

3.5 小　　结

在本章中我们写了第一个 C#程序“Welcome”。通过对该程序的分析和理解，我们学习到了：

- 如何在应用程序中与用户进行交互。
- 如何通过 System 预定义的类 Console 提供的方法来进行输入输出和对字符串进行格式化。
- 如何编译 C#源文件。
- 如何向代码中添加注释。

复习题

（1）C#程序通常都是如何开始执行的？
（2）Console 类为我们提供了那些输入输出的手段？
（3）说明如何使用编译器生成不同类型的输出文件。
（4）对源代码进行注释是必须的吗？为什么？
（5）说说在 C#代码中添加注释要注意那些问题。

第二部分　C#程序设计基础

第四章　数 据 类 型

应用程序总是需要处理数据，而现实世界中的数据类型多种多样，我们必须让计算机了解需要处理什么样的数据，以及采用哪种方式进行处理，按什么格式保存数据等等。比如，在编码程序中需要处理单个字符，在定购票系统需要打印货币金额，在科学运算中不同情况下我们需要不同精度的小数，这些都是不同的数据类型。

其实，任何一个完整的程序都可以看成是一些数据和作用于这些数据上的操作的说明。每一种高级语言都为开发人员提供一组数据类型，不同的语言提供的数据类型不尽相同。

在本书第一部分，我们为读者简要地介绍了 C#语言的主要特点和运行环境，并且创建了第一个简单的 C#应用程序。本章将会给你介绍 C#的数据类型系统。在这一章中，你将系统地学习 C#语言为我们提供的数据类型，以及使用这些数据类型时的要点。

对于程序中的每一个用于保存信息的量，使用时我们都必须声明它的数据类型，以便编译器为它分配内存空间。C#的数据类型可以分为两大部分：值类型和引用类型。

如果你是一名编程新手，本章将是你继续学习 C#的基础，应该牢牢掌握本章提到的各种数据类型。而对于有一定 C 和 C++经验的读者来说，对本章的内容不会感到困难，但在阅读本章和紧接着的几章里，要把注意力集中到文章中 C#独有的特点，尤其是与 C 和 C++的区别，这才是关键所在。

4.1 值　类　型

在具体讲解各种类型之前，我们先提一下变量的概念，在下一章我们将对变量作进一步的讨论。从用户角度来看，变量就是存储信息的基本单元；从系统角度来看，变量就是计算机内存中的一个存储空间。

下面我们开始介绍值类型。C#的值类型可以分为以下几种：

- 简单类型（Simple types ）
- 结构类型（Struct types）
- 枚举类型（Enumeration types）

简单类型，有时人们也称为纯量类型，是直接由一系列元素构成的数据类型。C# 语

言中为我们提供了一组已经定义的简单类型。从计算机的表示角度来看，这些简单类型可以分为整数类型、布尔类型、字符类型和实数类型。

4.1.1 整数类型

顾名思义，整数类型的变量的值为整数。数学上的整数可以从负无穷大到正无穷大，但是由于计算机的存储单元是有限的，所以计算机语言提供的整数类型的值总是在一定的范围之内。C#中有九种整数类型：短字节型（sbyte）、字节型（byte）、短整型（short）、无符号短整型（ushort）、整型（int）、无符号整型（uint）、长整型（long）、无符号长整型（ulong）。划分的依据是根据该类型的变量在内存中所占的位数。位数的概念是按照 2 的指数幂来定义的，比如说 8 位整数，则它可以表示 2 的 8 次方个数值，即 256。这和我们在 Windows 系统中显示属性是一样的，比如 16 位真彩色，表示一共可以显示 2 的 16 次方即 65536 种颜色。

这些整数类型在数学上的表示以及在计算机中的取值范围如表 4-1 中所示。

表 4-1 整数类型

数据类型	特征	取值范围
sbyte	有符号 8 位整数	在-128 到 127 之间
byte	无符号 8 位整数	在 0 到 255 之间
short	有符号 16 位整数	在-32,768 到 32,767 之间
ushort	无符号 16 位整数	在 0 到 65,535 之间
int	有符号 32 位整数	在-2,147,483,648 到 2,147,483,647 之间
uint	无符号 32 位整数	0 到 4,294,967,295 之间
long	有符号 64 位整数	在 9,223,372,036,854,775,808 和 9,223,372,036,854,775,807 之间
ulong	无符号 64 位整数	0 和 18,446,744,073,709,551,615 之间

程序清单 4-1：

```
using System;

class Test

{

   public static void Main()   {

      short x = 32766;

      x++;

      Console.WriteLine(x);

      x++;

      Console.WriteLine(x);

   }
```

```
    }
```

程序的输出为：

```
32767
-32768
```

上面的例子说明对于 short 类型的整数 x 已经超出了系统定义的范围（从-32768 到 32767 之间）。

4.1.2 布尔类型

布尔类型是用来表示“真”和“假”这两个概念的。这虽然看起来很简单，但实际应用非常广泛。我们知道，计算机实际上就是用二进制来表示各种数据的，即不管何种数据，在计算机的内部都是采用二进制方式处理和存储的。布尔类型表示的逻辑变量只有两种取值：“真”或“假”。在 C#中，分别采用 true 和 false 两个值来表示。

注意：在 C 和 C++中，用 0 来表示“假”，其它任何非 0 的式子都表示“真”。这种不正规的表达在 C#中已经被废弃了。在 C#中，true 值不能被其他任何非零值所代替。在其它整数类型和布尔类型之间不再存在任何转换，将整数类型转换成布尔型是不合法的:

```
bool x = 1  //错误，不存在这种写法。只能写成 x=true 或 x=false
```

4.1.3 实数类型

浮点类型

数学中的实数不仅包括整数，而且包括小数。小数在 C#中采用两种数据类型来表示：单精度（float）和双精度（double）。它们的差别在于取值范围和精度不同。计算机对浮点数的运算速度大大低于对整数的运算。在对精度要求不是很高的浮点数计算中，我们可以采用 float 型，而采用 double 型获得的结果将更为精确。当然，如果在程序中大量地使用双精度类浮点数，将会占用更多的内存单元，而且计算机的处理任务也将更加繁重。

- 单精度：取值范围在正负 1.5×10^{-45} 到 3.4×10^{38} 之间，精度为 7 位数。
- 双精度：取值范围在正负 5.0×10^{-324} 到 1.7×10^{308} 之间，精度为 15 到 16 位数。

十进制类型

C#还专门为我们定义了一种十进制类型(decimal)，主要用于方便我们在金融和货币方面的计算。在现代的企业应用程序中，不可避免地要进行大量的这方面的计算和处理，而目前采用的大部分程序设计语言都需要程序员自己定义货币类型等，这不能不说是一个遗憾。C#通过提供这种专门的数据类型，为我们弥补了这一遗憾，使我们

能够更为快捷地设计这方面的应用程序。

十进制类型是一种高精度、128 位数据类型，它所表示的范围从大约 1.0×10^{-28} 到 7.9×10^{28} 的 28 至 29 位有效数字。注意，该精度是用位数 (digits)而不是以小数位(decimal places)来表示的。运算结果准确到 28 个小数位。十进制类型的取值范围比 double 类型的范围要小得多，但它更精确。

当定义一个 decimal 变量并赋值给它时，使用 m 下标以表明它是一个十进制类型，如：

```
decimal d_value = 1.0m;
```

如果省略了 m，在变量被赋值之前，它将被编译器当作双精度（double）类型来处理。

4.1.4 字符类型

除了数字以外，计算机处理的信息，主要就是字符了。字符包括数字字符、英文字母、表达符号等，C#提供的字符类型按照国际上公认的标准，采用 Unicode 字符集。一个 Unicode 的标准字符长度为 16 位，用它可以来表示世界上大多种语言。可以按以下方法给一个字符变量赋值，如：

```
char c =  'A';
```

另外，我们还可以直接通过十六进制转义符（前缀\x）或 Unicode 表示法给字符型变量赋值（前缀\u），如下面对字符型变量的赋值写法都是正确的：

```
char c = '\x0032';      //
char c = '\u0032';      //
```

注意：在 C 和 C++中，字符型变量的值是该变量所代表的 ASCII 码，字符型变量的值作为整数的一部分，可以对字符型变量使用整数进行赋值和运算。而这在 C#中是被禁止的。

和 C、C++中一样，在 C#中仍然存在着转义符，用来在程序中指代特殊的控制字符。

表 4-2 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

4.1.5 结构类型

利用上面介绍过的简单类型，我们在进行一些常用的数据运算、文字处理似乎已经足够了。但是我们会经常碰到一些更为复杂的数据类型。比如，通讯录的记录中可以包含他人的姓名、电话和地址。如果按照简单类型来管理，每一条记录都要存放到三个不同的变量当中，这样工作量很大，也不够直观。有没有更好的办法呢？

正如上面的例子，在实际生活中，我们经常把一组相关的信息放在一起。把一系列相关的变量组织成为一个单一实体的过程，我们称为生成结构的过程。这个单一实体的类型就叫做结构类型，每一个变量称为结构的成员。结构类型的变量采用 struct 来进行声明，比如我们可以定义通讯录记录结构的定义：

```
struct PhoneBook{
        public string name;
        public string phone;
        public string address;
}
PhoneBook p1;
```

p1 就是一个 PhoneBook 结构类型的变量。上面声明中的 public 表示对结构类型的成员的访问权限，有关访问的细节问题我们将在第三部分详细讨论。对结构成员的访问通过结构变量名加上访问符“.”号，再跟成员的名称：

```
p1.name = “Mike”;
```

结构类型包含的成员类型没有限制，可以相同，也可以不同。比如，我们可以在通讯录的记录中在加上年龄这个成员：

```
struct PhoneBook {
   public string name;
   public uint age;
   public string phone;
   public string address;
}
```

我们甚至可以把结构类型作为另一个结构的成员的类型，这也没有任何问题：

```
struct PhoneBook {
   public string name;
   public uint age;
   public string phone;
   public struct address{
       public string city;
       public string street;
       public uint no;
   }
}
```

这里，“通讯录”这个结构中又包括了“地址”这个结构，结构“地址”类型包括城市、街道、门牌号码三个成员。

4.1.6 枚举类型

枚举（enum）实际上是为一组在逻辑上密不可分的整数值提供便于记忆的符号。比如，我们声明一个代表星期的枚举类型的变量：

```
enum WeekDay
    Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
};
WeekDay day;
```

注意：结构是由不同类型的数据组成的一组新的数据类型，结构类型的变量的值是由各个成员的值组合而成的。而枚举则不同，枚举类型的变量在某一时刻只能取枚举中某一个元素的值。比如，day 这个表示“星期”的枚举类型的变量，它的值要么是 Sunday，要么是 Monday 或其它的星期元素，但它在一个时刻只能代表具体的某一天，不能既是星期二、又是星期三。

```
day = Tuseday;
```

按照系统的默认，枚举中的每个元素类型都是 int 型，且第一个元素删去的值为 0，它后面的每一个连续的元素的值按加 1 递增。在枚举中，也可以给元素直接赋值，如下把星期天的值设为 1，其后的元素的值分别为 2,3…

```
enum WeekDay
    Sunday=1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
};
```

为枚举类型的元素所赋的值的类型限于 long、int、short 和 byte 等整数类型。

4.2 引 用 类 型

C#中的另一大数据类型是引用类型。“引用”这个词在这里的含义是，该类型的变量不直接存储所包含的值，而是指向它所要存储的值。也就是说，引用类型存储实际数据的引用值的地址。C#中的引用类型有四种：

- 类
- 代表
- 数组
- 接口

我们将在这里介绍前三种引用类型，而把对接口的深入讲述将放在第十五章进行。

4.2.1 类

类是面向对象编程的基本单位，是一种包含数据成员、函数成员和嵌套类型的数据结构。类的数据成员有常量、域和事件。函数成员包括方法、属性、索引指示器、运算符、构造函数和析构函数。类和结构同样都包含了自己的成员，但它们之间最主要的区别在于：类是引用类型，而结构是值类型。

类支持继承机制，通过继承，派生类可以扩展基类的数据成员和函数方法，进而达到代码重用和设计重用的目的。

有关类的概念将放在第十章详细讲解，这里请看一个类的定义：

```
class PhoneBook
{
   private string name;
   private string phone;
   private struct address{
       public string city;
       public string street;
       public uint no;
   }
   public string Phone{
       get{
            return phone;
       }
       set{
            phone = value;
       }
   public PhoneBook(string n){
       name = n;
   }
   public Edit()
   {;}
}
```

上面定义了 PhoneBook 这个类，类包括的数据成员有域 name、phone、address，属性 P1hone；类的函数成员有构造函数 PhoneBook(string n)，方法 Edit。

如果我们对某个类定义了一个变量，我们称它为类的一个实例。

下面我们介绍两个经常用到的类：

object 类

object 类是所有其它类型的基类，C#中的所有类型都直接或间接地从 object 类中继承。因此，对一个 object 的变量可以赋予任何类型的值：

```
int x =25;
object obj1;
obj1 = x;
object obj2 = 'A';
```

对 object 类型的变量声明采用 object 关键字，这个关键字是在.Net 框架结构为我们提供的预定义的名字空间 System 中定义的，是类 System.Object 的别名。

string 类

C#还定义了一个基本的类 string，专门用于对字符串的操作。同样，这个类也是在.Net 框架结构的名字空间 System 中定义的，是类 System. string 的别名。

字符串在实际中应用非常广泛，在类的定义中封装了许多内部的操作，我们只要简单地加以利用就可以了。可以用加号“+”合并两个字符串，采用下标从字符串中获取字符，等等。

```
string String1 = "Welcome";
string String2 = "Welcome " + " everyone";
char c = String1[0];
bool b =   (String 1== String2);
```

4.2.2 代表

在 C 和 C++的程序员看来，指针即是它们最强有力的工具之一，同时又给他们带来了很多苦恼之处。因为指针指向的数据类型可能并不相同，比如你可以把 int 类型的指针指向一个 float 类型的变量，而这时程序并不会出错。而且，如果你删除了一个不应该被删除的指针（比如 Windows 中指向主程序的指针），程序就有可能崩溃。由此可见，滥用指针给程序的安全性埋下了隐患。

正因为如此，在 C#语言中取消了指针这个概念。当然，对指针恋恋不舍的程序员仍然可以在 C#中使用指针，但必须声明这段程序是“非安全”（unsafe）的。而我们在这里要介绍的是 C#的一个引用类型——代表（delegate）。它实际上相当于 C#中的函数指针原型。与指针不同的是，代表在 C#是类型安全的。

在声明代表时，只需要指定代表指向的原型的类型，它不能有返回值，也不能带有输出类型的参数。比如我们可以声明一个指向 int 类型函数原型的代表：

```
delegate int MyDelegate();
```

如果我们声明了自己的一个代表，那么它就是对系统定义的类 System.delegate 的一个扩展。在代表的实例中，我们可以封装一个静态方法，也可以封装一个非静态的方法。我们看下面的例子：

程序清单 4-2：

```
using System;

delegate int MyDelegate();     //声明一个代表

public class MyClass
```

```
{
    public int InstanceMethod ()    {
        Console.WriteLine("Call the instance method.");
        return 0;
    }
     static public int StaticMethod ()    {
            Console.WriteLine("Call the static method.");
            return 0;
       }
}
public class Test
 {
      static public void Main ()
     {
            MyClass p = new MyClass();
            // 将代表指向非静态的方法 InstanceMethod
            MyDelegate d = new MyDelegate(p.InstanceMethod);
            // 调用非静态方法
            d();
           // 将代表指向静态的方法 StaticMethod
            d = new MyDelegate(MyClass.StaticMethod);
           // 调用静态方法
            d();
           }
}
```

程序的输出结果是：

call the instance method.

call the static method.

4.2.3 数组

在进行批量处理数据的时候，我们要用到数组。数组是一组类型相同的有序数据。数组按照数组名、数据元素的类型和维数来进行描述。C#中提供 System.Array 类是所有数组类型的基类。

数组的声明格式：

non-array-type [*dim-separators*] *array-instance name*；

比如我们声明一个整数数组：

```
int[] arr;
```

在定义数组的时候，可以预先指定数组元素的个数，这时在“[]”中定义数组的元素个数，它的个数可以通过数组名加圆点加“Length”获得。而在使用数组的时候，可以在“[]”中加入下标来取得对应的数组元素。C#中的数组元素的下标是从 0 开始的，也就是说，第一个元素对应的下标为 0，以后逐个增加。

在 C#中数组可以是一维的也可以是多维的，同样也支持矩阵和参差不齐的数组。一维数组最为普遍，用的也最多。我们先看一看下面的例子：

程序清单 4-3：

```
using System;

class Test

{

  static void Main() {

    int[] arr = new int[5];

    for (int i = 0; i < arr.Length; i++)

      arr[i] = i * i;

    for (int i = 0; i < arr.Length; i++)

      Console.WriteLine("arr[{0}] = {1}", i, arr[i]);

}

}
```

这个程序创建了一个基类型为 int 型的一维数组，初始化后逐项输出。其中 arr.Length 表示数组元素的个数。程序的输出为：

```
arr[0] = 0
arr[1] = 1
arr[2] = 4
arr[3] = 9
arr[4] = 16
```

上面的例子中我们用的是一维的，很简单吧！下面我们介绍多维的：

```
class Test
{
  static void Main() {            //可动态生成数组的长度
    string[] a1;           // 一维 string 数组
    string[,] a2;          // 二维
    string[,,] a3;         // 三维
    string[][] j2;         // 可变数组（数组）
          string[][][][] j3;    //多维可变数组
   }
}
```

在数组声明的时候可以对数组元素进行赋值，或者叫做对数组的初始化。也可以在使用的时候进行动态赋值。看下面的例子：

```
class Test
{
  static void Main() {
    int[] a1 = new int[] {1, 2, 3};
    int[,] a2 = new int[,] {{1, 2, 3}, {4, 5, 6}};
    int[,,] a3 = new int[10, 20, 30];
    int[][] j2 = new int[3][];
    j2[0] = new int[] {1, 2, 3};
    j2[1] = new int[] {1, 2, 3, 4, 5, 6};
    j2[2] = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9};
  }
}
```

上面的例子中我们可以看出数组初始化可以用几种不同的类型，因为它要求在一个初始化时要确定其类型。比如下面的写法是错误的：

```
class Test
{
  static void F(int[] arr) {}
```

```
    static void Main() {
      F({1, 2, 3});
    }
}
```

因为数组初始化时{1，2，3}并不是一个有效的表达式。我们必须要明确数组类型：

```
class  Test
{
   static void F(int[] arr) {}
   static void Main() {
      F(new int[] {1, 2, 3});
   }
}
```

4.3 装箱和拆箱

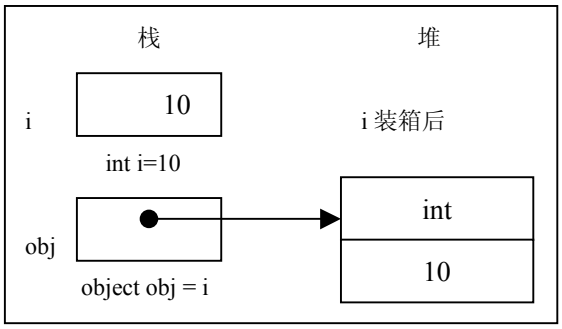
到目前为止，我们为大家讲解了有关 C#语言中的值类型和引用类型数据。这一节我们来了解一下 C#语言类型系统提出的 1 一个核心概念：装箱(boxing)和拆箱(unboxing)。装箱和拆箱机制使得在 C#类型系统中，任何值类型、引用类型和 object（对象）类型之间进行转换，我们称这种转化为绑定连接。简单地说，有了装箱和拆箱的概念，对任何类型的值来说最终我们都可以看作是 object 类型。

4.3.1 装箱转换

装箱转换是指将一个值类型隐式地转换成一个 `object` 类型，或者把这个值类型转换成一个被该值类型应用的接口类型（interface-type）。把一个值类型的值装箱，也就是创建一个 object 实例并将这个值复制给这个 object。比如：

```
int i = 10;
object obj = i;
```

用下图可以表示装箱的过程：



我们也可以用显式的方法来进行装箱操作：

```
int i = 10;
object obj = object(i);
```

我们可以假想存在一个 boxing 类型，其声明如下：

```
class T_Box
{
  T value;

        T_Box(T t) {
        value = t;
  } //该类型的构造函数。
}
```

这里 T 表示将要装箱的值的类型，它可以是 int、char、enum 等等。现在我们要将类型为 T 的值 v 装箱，其执行过程为：执行 new T_Box(v)，将其返回结果的实例作为对象类型的值，那么下面的语句：

```
int i = 10;
object obj = i;
```

等价于：

```
int i = 10;
object obj = new int_Box(i);       //将 i 装箱成对象 obj
```

我们看一下下面的程序。

程序清单 4-4：

```
using System

class Test{

   public static void Main(){

      int i = 10;
      object obj = i;         //对象类型
```

```
            if (obj is int) {
               Console.Write("The value of i is boxing! ");
            }
            i = 20;          // 改变 i 的值
            Console.WriteLine("int: i = {0}", i);
            Console.WriteLine("object: obj = {0}", obj);
        }
    }
```

输出结果为：

```
The value of i is boxing!
int: i = 20;
object: obj = 10;
```

这就证明了，被装箱的类型的值是作为一个拷贝赋给对象的。

4.3.2 拆箱转换

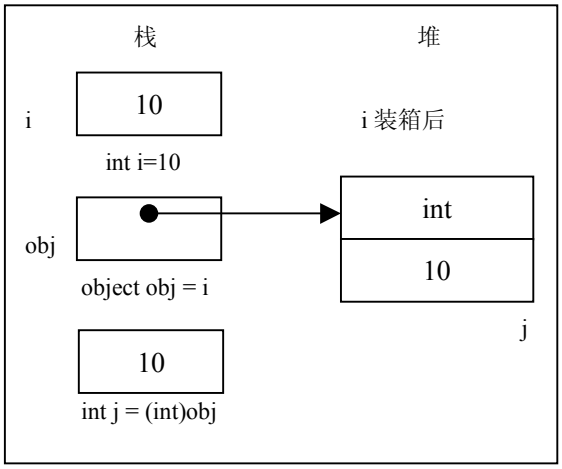
和装箱转换正好相反，拆箱转换是指将一个对象类型显式地转换成一个值类型，或是将一个接口类型显式地转换成一个执行该接口的值类型。

拆箱的过程分为两步：首先，检查这个对象实例，看它是否为给定的值类型的装箱值。然后，把这个实例的值拷贝给值类型的变量。

我们举个例子来看看将一个对象拆箱的过程。

```
int i = 10;
object obj = i;
int j = (int)obj;
```

这个过程用图来表示就是：



可以看出拆箱过程正好是装箱过程的逆过程。必须注意，装箱转换和拆箱转换必须遵循类型兼容原则。

4.4 小　　结

C#中的数据类型可以分为两大部分：值类型和引用类型。值类型的变量总是直接包含着自身的数据，而引用类型的变量是指向实际数据的地址。

C#的值类型包括整型、布尔型、实型、十进制型、结构和枚举。引用类型包括类、接口、代表和数组。

装箱和拆箱使我们可以把一个值类型当作一个引用类型来看待。装箱转换是指将一个值类型隐式地转换成一个 `object` 类型，拆箱转换是指将一个 `object` 类型显式地转换成一个值类型，它们互为逆过程。

复习题

（1）值类型和引用类型的区别在哪里？

（2）C#中为什么要把布尔类型从整数类型中单独分离出来？

（3）说说在什么情况下需要使用结构，在什么情况下需要使用枚举。

（4）写出下面程序运行的结果：

```
using System;

class Test

{

   public static void Main(){

      char c;

      c = 'W';

      Console.Write(c);

      c = 'E';

      Console.Write(c);

      c = '\n';

      Console.Write(c);

      c = 'L';

      Console.Write(c);

      c = '\a';
```

```
            Console.Write(c);
            c = 'C';
            Console.Write(c);
            c = '\t';
            Console.Write(c);
            c = 'O';
            Console.Write(c);
            c = '\v';
            Console.Write(c);
            c = 'M';
            Console.Write(c);
        }
    }
```

（5）对数组的初始化应注意那些问题？

（6）假设有一段程序对字符串进行加密，加密后的字符串的第一个字符是原字符串的最后一个字符，其余的每个字符是对应的原字符串中的前一个字符的值加上 3。比如“welcome”，末尾的字符为“e”，“welcom”依次加上 3 后成为“zhofrp”，故加密后的结果为“zhofrp”。程序由用户任意输入一个字符串，加密后输出。

（7）简述装箱和拆箱的过程。

第五章　变量和常量

有关变量和常量的知识是一门编程语言的基础知识，而每一门编程语言都有自己对变量和常量的命名和使用方式。本章将对 C#语言中的变量和常量进行讲解，主要内容包括：变量和常量各自的用途，如何对变量和常量进行命名，如何定义和初始化变量和常量。

5.1　变　　量

程序要对数据进行读、写、运算等操作。当需要保存特定的值或计算结果时，就需要用到变量（variable）。在用户看来，变量是用来描述一条信息的名称，在变量中可以存储各种类型的信息，比如：人的姓名、车票的价格、文件的长度，等等。

在计算机中，变量代表存储地址，变量的类型决定了存储在变量中的数值的类型。C#是一种安全类型语言，它的编译器存储在变量中的数值具有适当的数据类型。同时变量的值可以通过赋值或“++”和“—”运算符运算被改变。

使用变量的一条重要原则是：变量必须先定义后使用。

变量可以在定义时被赋值，也可以在定义时不被赋值。一个定义时被赋值的变量很好地定义了一个初始值。一个定义时不被赋值的变量没有初始值。要给一个定义时没有被赋值的变量赋值必须是在一段可执行的代码中进行。

5.1.1　命名变量

当我们需要访问存储在变量中的信息时，我们只需要使用变量的名称。为变量起名时要遵守 C#语言的规定：

- 变量名必须以字母开头。
- 变量名只能由字母、数字和下划线组成，而不能包含空格、标点符号、运算符等其它符号。
- 变量名不能与 C#中的关键字名称相同。这些关键字我们在附录 A 中给出。
- 变量名不能与 C#中的库函数名称相同。

但在 C#中有一点是例外，那就是允许在变量名前加上前缀“@”。在这种情况下，我们就可以使用前缀“@”加上关键字作为变量的名称。这主要是为了与其它语言进行交互时避免冲突。因为前缀“@”实际上并不是名称的一部分，其它的编程语言就会把它作为一个普通的变量名。在其它情况下，我们不推荐使用前缀“@”作为变量名的一部分。

下面给出了一些合法和非法的变量名的例子：

```
int i;   //合法
int No.1;    //不合法，含有非法字符
string total;  //合法
char use;    //不合法，与关键字名称相同
char @use;   //合法
float Main;  //不合法，与函数名称相同
```

尽管符合了上述要求的变量名就可以使用，但我们还是希望在给变量取名的时候，应给出具有描述性质的名称，这样写出来的程序便于理解。比如一个消息字符串的名字就可以叫 s_message；而 e90PT 就不是一个好的变量名。

我们可以在一条语句中命名多个类型相同的变量，如：

```
int a,b,c=50,d;
```

5.1.2 变量的类型

在 C#语言中，我们把变量分为七种类型，它们分别是：静态变量（static variables），非静态变量（instance variables），数组元素（array elements），值参数(value parameters)，引用参数（reference parameters），输出参数（output parameters），还有局部变量（local variables）。看下面的例子：

```
class A
{
  public static int x;
  int y;

  void F(int[] v, int a, ref int b, out int c) {
      int i = 1;
      c = a + b++;
  }
}
```

在上面的变量声明中，x 是静态变量，y 是非静态变量，v[0]是数组元素，a 是值参数，b 是引用参数，c 是输出参数，i 是局部变量。

静态变量

带有“static”修饰符声明的变量称为静态变量。一旦静态变量所属的类被装载，直到包含该类的程序运行结束时它将一直存在。静态变量的初始值就是该变量类型的默认值。为了便于定义赋值检查，静态变量最好在定义时赋值。如：static int a = 10；

非静态变量

不带有“static”修饰符声明的变量称为实例变量。如：

```
int a;
```

针对类中的非静态变量而言，一旦一个类的新的实例被创建，直到该实例不再被应用从而所在空间被释放为止，该非静态变量将一直存在。同样鉴于定义赋值检查，一个类的非静态变量也应该在初始化时赋值。

结构中的非静态变量随着结构的存在而存在。也就是说，当一个结构类型的变量存在或结束时，该结构类型中的变量也随之存在和结束。同样的，结构中的实例变量会随着结构类型变量的初始化而被初始化，如果该结构类型变量定义时未被赋值，那么其中的实例变量也不会被赋值。

数组元素

数组元素也是变量的一种，该变量随该数组实例的存在而存在。每一个数组元素的初始值是该数组元素类型的默认值。同样鉴于定义赋值检查，数组元素最好在初始时被赋值。

局部变量

局部变量是指在一个独立的程序块，一个 for 语句，switch 语句、或者 using 语句中声明的变量，它只在该范围中有效。当程序运行到这一范围时，该变量即开始生效，程序离开时变量就失效了。

与其它几种变量类型不同的是，局部变量不会自动被初始化，所以也就没有默认值。在进行赋值检查的时候，局部变量被认为没有被赋值。

在局部变量的有效范围内，在变量的定义以前就使用是不合法的，比如：

```
for(int i=0 ;i<10;i++){

  int num = a;       //非法，因为局部变量 a 还没有定义

  int a;

  int b = a;    //正确

 }
```

关于值参数、引用参数、输出参数，我们放在第十一章“方法”中进行详细介绍。

5.2 常　　量

常量就是其值固定不变的量。从数据类型角度来看，常量的类型可以是任何一种值类型或引用类型。

一个常量的声明，就是声明程序中要用到的常量的名称和它的值。和变量一样，

我们可以同时声明一个或多个给定类型的常量。常量声明的格式如下：

attributes constant-modifiers *const* type constant-declarators ;

其中，常量修饰符 constant-modifier 可以是：

- new
- public
- protected
- internal
- private

常量的类型 type 必须是以下之一：

sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, string, 枚举类型*(* enum-type)，或引用类型 *(* reference-type).

常量表达式 *constant-declarators* 表示常量的名字。

```
public const double X = 1.0, Y = 2.0, Z = 3.0;
```

5.3 小　　结

在本章节中，你学到了 C#中编程中用到的变量和常量的声明、表示、分类等。这些知识都是学习编程必须掌握的知识。

我们在本章了解了常量和变量的基础知识。实际编程过程中，它们往往是作为一个类的成员而存在的，在完成对本书第三部分的学习后，你将会熟练地把它们运用到实践当中去。

复习题

（1）说说给变量起名时应注意那些问题。

（2）说明静态变量与非静态变量的区别。

（3）说明使用局部变量应注意的问题。

第六章　类 型 转 换

在 C#语言中，一些预定义的数据类型之间存在着预定义的转换。比如，从 int 类型转换到 long 类型。C#语言中数据类型的转换可以分为两类：隐式转换（***implicit conversions***）和显式转换（***explicit conversions***）。本章我们将详细介绍这两类转换。

6.1　隐式类型转换

隐式转换就是系统默认的、不需要加以声明就可以进行的转换。在隐式转换过程中，编译器无需对转换进行详细检查就能够安全地执行转换。比如从 int 类型转换到 long 类型就是一种隐式转换。隐式转换一般不会失败，转换过程中也不会导致信息丢失。比如：

```
int i = 10;
long l = i;
```

我们在上一章介绍的装箱转换实际上就是一种隐式类型转换。在本节，我们还将讲解以下隐式转换的规则：

- 隐式数值换
- 隐式枚举转换
- 隐式引用转换

隐式转换发生的场合不一，包括函数成员调用，表达式计算和分配等。

6.1.1　隐式数值转换

隐式数值转换包括以下几种：

- 从 sbyte 类型到 short, int, long, float, double, 或 decimal 类型。
- 从 byte 类型到 short, ushort, int, uint, long, ulong, float, double, 或 decimal 类型。
- 从 short 类型到 int, long, float, double, 或 decimal 类型。
- 从 ushort 类型到 int, uint, long, ulong, float, double, 或 decimal 类型。
- 从 int 类型到 long, float, double, 或 decimal 类型。
- 从 uint 类型到 long, ulong, float, double, 或 decimal 类型。
- 从 long 类型到 float, double, 或 decimal 类型。
- 从 ulong 类型到 float, double, 或 decimal 类型。
- 从 char 类型到 ushort, int, uint, long, ulong, float, double, 或 decimal 类型。
- 从 float 类型到 double 类型。

其中，从 int, uint, 或 long 到 float 以及从 long 到 double 的转换可能会导致精度下降，但决不会引起数量上的丢失。其它的隐式数值转换则不会有任何信息丢失。

结合我们在数据类型中学习到的值类型的范围，我们可以发现，隐式数值转换实际上就是从低精度的数值类型到高精度的数值类型的转换。

从上面的 10 条我们可以看出，不存在到 char 类型的隐式转换，这意味着其它整型值不能自动转换为 char 类型。这一点需要特别注意。

下面的程序给出了隐式数值转换的例子。

程序清单 6-1：

```
using System;

class Test

{

  public static void Main()

  {

    byte x = 16;

    Console.WriteLine("x = {0}",x);

    ushort y = x;

    Console.WriteLine("y = {0}",y);

    y = 65535;

    Console.WriteLine("y = {0}",y);

    float z = y;

    Console.WriteLine("z = {0}",z);

  }

}
```

程序的输出将是：

```
x = 16;
y = 16;
y = 65535;
z = 65535;
```

如果我们在上面程序中的语句之后再加上一句：

```
y = y+1;
```

再重新编译程序时，编译器将会给出一条错误信息：

```
Can not implictly convert type 'int' to type 'ushort'
```

这说明，从整数类型 65536 到无符号短整型 y 不存在隐式转换。

6.1.2 隐式枚举转换

隐式枚举转换允许把十进制整数 0 转换成任何枚举类型，对应其它的整数则不存在这种隐式转换。还是让我们用例子来说明。

程序清单 6-2：

```
using System;

enum Weekday{

   Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday

};

class Test

{

   public static void Main() {

      Weekday day;

      day = 0;

      Console.WriteLine(day);

   }

}
```

程序的输出是：

```
0
```

但是如果我们把语句 day = 0 改写为 day = 1，编译器就会给出错误：

```
Can not implictly convert type 'int' to type 'enum'
```

6.1.3 隐式引用转换

隐式引用转换包括以下几类：

- 从任何引用类型到对象类型的转换。
- 从类类型 s 到类类型 t 的转换，其中 s 是 t 的派生类。
- 从类类型 s 到接口类型 t 的转换，其中类 s 实现了接口 t。
- 从接口类型 s 到接口类型 t 的转换，其中 t 是 s 的父接口。

从元素类型为 Ts 的数组类型 S 向元素类型为 Tt 的数组类型 T 转换，这种转换需要满足下列条件：

- S 和 T 只有元素的数据类型不同，但它们的维数相同。
- Ts 和 Tt 都是引用类型。

- 存在从 Ts 到 Tt 的隐式引用转换。
- 从任何数组类型到 System.Array 的转换。
- 从任何代表类型到 System.Delegate 的转换。
- 从任何数组类型或代表类型到 System.ICloneable 的转换。
- 从空类型（null）到任何引用类型的转换。

比如，下面的程序无法通过编译，因为数组的元素类型是值类型，C#中不存在这样的隐式转换：

程序清单 6-3：

```
using System;

class Test

{

  public static void Main()  {

    float[] float_arr = new float[10];

    int[] int_arr = new int[10];

    float_arr = int_arr;

  }

}
```

而下面这段程序则是正确的。

程序清单 6-4：

```
using System;

class Class1

{}

class Class2 : Class1

{}

class Test

{

  public static void Main()  {

    Class1[] class1_arr = new Class1[10];

    Class2[] class2_arr = new Class2[10];

    class1_arr = class2_arr;

  }
```

```
}
```

程序 6-5 很有趣，它给出了我们常用的值类型在系统环境中的原型定义。

程序清单 6-5：

```
using System;

class Test

{

   public static void Main()   {

      float[] float_arr = new float[10];

      double[] double_arr = new double[10];

      sbyte[] sbyte_arr = new sbyte[10];

      byte[] byte_arr = new byte[10];

      ushort[] ushort_arr = new ushort[10];

      int[] int_arr = new int[10];

      long[] long_arr = new long[10];

      string[] string_arr = new string[10];

      Console.WriteLine(float_arr);

      Console.WriteLine(double_arr);

      Console.WriteLine(sbyte_arr);

      Console.WriteLine(byte_arr);

      Console.WriteLine(ushort_arr);

      Console.WriteLine(int_arr);

      Console.WriteLine(long_arr);

      Console.WriteLine(string_arr);

   }

}
```

程序的输出结果是：

```
System.Single[];
System.Double[];
System.Sbyte[];
Syetem.Byte[];
```

Syetem.Int16[];

Syetem.Int32[];

Syetem.Int64[];

System.String[];

6.2 显式类型转换

显式类型转换，又叫强制类型转换。与隐式转换正好相反，显式转换需要用户明确地指定转换的类型。比如下面的例子把一个类型显式转换为类型：

```
long l = 5000;
int i = (int) l;
```

上一章介绍的拆箱转换就是一种显式转换。这里我们还将讲解以下转换的规则：

- 显式数值转换
- 显式枚举转换
- 显式引用转换

显式转换可以发生在表达式的计算过程中。它并不是总能成功，而且常常可能引起信息丢失。

显式转换包括所有的隐式转换，也就是说把任何系统允许的隐式转换写成显式转换的形式都是允许的，如：

```
int i = 10;
long l = (long)i;
```

6.2.1 显式数值转换

显式数值转换是指当不存在相应的隐式转换时，从一种数字类型到另一种数字类型的转换。包括：

- 从 sbyte 到 byte, ushort, uint, ulong, 或 char。
- 从 byte 到 sbyte 或 char。
- 从 short 到 sbyte, byte, ushort, uint, ulong, 或 char。
- 从 ushort 到 sbyte, byte, short, 或 char。
- 从 int 到 sbyte, byte, short, ushort, uint, ulong, 或 char。
- 从 uint 到 sbyte, byte, short, ushort, int, 或 char。
- 从 long 到 sbyte, byte, short, ushort, int, uint, ulong, 或 char。
- 从 ulong 到 sbyte, byte, short, ushort, int, uint, long, 或 char。
- 从 char 到 sbyte, byte, 或 short。
- 从 float 到 sbyte, byte, short, ushort, int, uint, long, ulong, char, 或 decimal。
- 从 double 到 sbyte, byte, short, ushort, int, uint, long, ulong, char, float, or 或

decimal。

- 从 decimal 到 sbyte, byte, short, ushort, int, uint, long, ulong, char, float, 或 double。

这种显式转换有可能丢失信息或导致异常抛出，转换按照下列规则进行：

- 对于从一种整型到另一种整型的转换，编译器将针对转换进行溢出检测，如果没有发生溢出，转换成功，否则抛出一个 OverflowException 异常。这种检测还与编译器中是否设定了 checked 选项有关。
- 对于从 float, double, 或 decimal 到整型的转换，源变量的值通过舍入得到最接近的整型值作为转换的结果。如果这个整型值超出了目标类型的值域，则将抛出一个 OverflowException 异常。
- 对于从 double 到 float 的转换，double 值通过舍入取最接近的 float 值。如果这个值太小，结果将变成正 0 或负 0；如果这个值太大，将变成正无穷或负无穷。如果原 double 值是 NaN，则转换结果也是 NaN。
- 对于从 float 或 double 到 decimal 的转换，源值将转换成小数形式并通过舍入取到小数点后 28 位（如果有必要的话）。如果源值太小，则结果为 0；如果太大以致不能用小数表示，或是无穷和 NaN，则将抛出 InvalidCastException 异常。
- 对于从 decimal 到 float 或 double 的转换，小数的值通过舍入取最接近的值。这种转换可能会丢失精度，但不会引起异常。

程序清单 6-6：

```
using System;

class Test
{
  static void Main() {
        long longValue = Int64.MaxValue;
        int intValue = (int) longValue;
        Console.WriteLine("(int) {0} = {1}", longValue, intValue);
  }
}
```

这个例子把一个 int 类型转换成为 long 类型，输出结果是：

```
(int) 9223372036854775807 = -1
```

这是因为发生了溢出，从而在显式类型转换时导致了信息丢失。

6.2.2 显式枚举转换

显式枚举转换包括以下内容：

- 从 sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, 或 decimal 到任何枚举类型。
- 从任何枚举类型到 sbyte, byte, short, ushort, int, uint, long, ulong, char, float,

double, 或 decimal。

- 从任何枚举类型到任何其它枚举类型。

显式枚举转换是这样进行的：它实际上是枚举类型的元素类型与相应类型之间的隐式或显式转换。比如，有一个元素类型为 int 的枚举类型 E，则当执行从 E 到 byte 的显式枚举转换时，实际上作的是从 int 到 byte 的显式数字转换；当执行从 byte 到 E 的显式枚举转换时，实际上是执行 byte 到 int 的隐式数字转换。

比如，对程序 6-2，我们改写如下。

程序清单 6-7：

```
using System;

enum Weekday{

   Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday

};

class Test

{

   public static void Main() {

      Weekday day;

      day = (Weekday)3;

      Console.WriteLine(day);

   }

}
```

程序的输出是：

```
3
```

6.2.3 显式引用转换

显式引用转换包括：

- 从对象到任何引用类型
- 从类类型 S 到类类型 T，其中 S 是 T 的基类。
- 从类类型 S 到接口类型 T，其中 S 不是密封类，而且没有实现 T。
- 从接口类型 S 到类类型 T，其中 T 不是密封类，而且没有实现 S。
- 从接口类型 S 到接口类型 T，其中 S 不是 T 的子接口。

从元素类型为 Ts 的数组类型 S 到元素类型为 Tt 的数组类型 T 的转换，这种转换需要满足下列条件：

- S 和 T 只有元素的数据类型不同，而维数相同。

- Ts 和 Tt 都是引用类型。
- 存在从 Ts 到 Tt 的显式引用转换。.
- 从 System.Array 到数组类型。
- 从 System.Delegate 到代表类型。
- 从 System.ICloneable 到数组类型或代表类型。

显式引用转换发生在引用类型之间，需要在运行时检测以确保正确。

为了确保显式引用转换的正常执行，要求源变量的值必须是 null 或者它所引用的对象的类型可以被隐式引用转换转换为目标类型。否则显式引用转换失败，将抛出一个 InvalidCastException 异常。

不论隐式还是显式引用转换，虽然可能会改变引用值的类型，却不会改变值本身。

6.3 小　　结

这一章详细讲述了 C#类型转换的隐式转换和显式转换，以及它们的实现。类型转换的引入增强了程序的灵活性，但不当的使用也可能导致数据丢失，或者引发异常。因此建议大家细看我们所列出的规则，并在实际使用中注意把握。

复习题

（1）哪些情况下类型转换会导致结果溢出，在哪些情况下又会导致发生异常？

（2）下面声明了一些数值类型的变量：

```
short s; int i; uint ui; long l; double d; demical m;
```

试分析下面哪些转换是正确的，哪些是错误的：

```
s = ui;        i = ui;

s = l;         l = s;

m = d;         d = m;

l = d;         m = i;
```

（3）在枚举类型和整数类型之间进行转换有什么规定？

（4）null 是否可以被转换为任何引用类型，为什么？

（5）分析下面的程序，试找出其中的错误；如果没有错误，写出程序的运行结果。

```
Using System;

enum Color{

   Red, Yellow, Blue, Green, Purple, Black, White

};
```

```
Class Test
{
   public static void Main(){
          Color[] color_arr = new Color[3];
          int[] int_arr = new int[]{1, 2, 3};
          color_arr = (Color[])int_arr;
            Console.WriteLine("The value of color[0] is:{0}",color[0]);
            Console.WriteLine("The value of color[1] is:{0}",color[1]);
              Console.WriteLine("The value of color[2] is:{0}",color[2]);
          }
   }
```

第七章　表　达　式

C#语言中的表达式类似于数学运算中的表达式，是操作符、操作对象和标点符号等连接而成的式子。操作符是用来定义类实例中表达式操作符的。表达式是指定计算的操作符、操作数序列。本章主要讲解操作符的使用，以及各类表达式的语法。

7.1　操　作　符

7.1.1　操作符的分类

表达式由操作数和操作符组成。表达式的操作符指出了对操作数的操作。比如操作符有+，-，/，和 new；操作数可以是文字、域、当前变量或表达式。

依照操作符作用的操作数的个数来分，C#中有三种类型的操作符：

- 一元操作符。一元操作符作用于一个操作数。一元操作符又包括前缀操作符和后缀操作符。
- 二元操作符。二元操作符作用于两位操作数，使用时在操作数中间插入操作符。
- 三元操作符。C#中仅有一个三元操作符“ ?:”，三元操作符作用于三个操作数，使用时在操作数中间插入操作符。

下面分别给出使用操作符的例子：

```
int x=5,y=10,z;

x++;    //后缀一元操作符

--x;    //前缀一元操作符

z = x + y;    //二元操作符

y = (X>10? 0 : 1); //三元操作符
```

7.1.2　操作符的优先级

当一个表达式包含多样操作符时，操作符的优先级控制着单个操作符求值的顺序。例如，表达式 x + y * z 按照 x + (y * z) 求值，因为“*”操作符比“+”操作符有更高的优先级。这和数学运算中的先乘除后加减是一致的。

表 7-1 总结了所有操作符从高到低的优先级顺序。

表 7-1　操作符从高到低的优先级顺序

类别	操作符
初级操作符	(x)　x.y　f(x)　a[x]　x++　x--　new type of　sizeof checked　unchecked
一元操作符	+　-　!　~　++x　--x　(T)x
乘、除操作符	*　/　%
加减操作符	+　-
移位操作符	<<　>>
关系操作符	<　>　<=　>=　is　as
等式操作符	==　!=
逻辑与操作符	&
逻辑异或操作符	^
逻辑或操作符	|
条件与操作符	&&
条件或操作符	||
条件操作符	?:
赋值操作符	=　*=　/=　%=　+=　-=　<<=　>>=　&=　^=　|=

当一个操作数出现在两个有相同优先级的操作符之间时，操作符按照出现的顺序由左至右执行。

除了赋值的操作符，所有的二进制的操作符都是左结合(***left-associative***)的，也就是说，操作按照从左向右的顺序执行。例如：x + y + z 按 (x + y) + z 进行求值

赋值操作符和条件操作符(?:)按照右接合(***right-associative***)的原则，即操作按照从右向左的顺序执行。如：x = y = z 按照 x = (y = z)进行求值。

建议在写表达式的时候，如果无法确定操作符的有效顺序，则尽量采用括号来保证运算的顺序，这样也使得程序一目了然，而且自己在编程时能够思路清晰。

7.2　算术操作符和算术表达式

C#中提供的算术操作符有五种：

- + 加法操作符
- – 减法操作符
- * 乘法操作符
- / 除法操作符
- % 求余操作符

在表达式的运算中，表达式总是按它们本身书写的顺序求值，如下例。

程序清单 7-1：

```
using System;

class Test
{
  static void F(int x, int y, int z) {
        Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
  }

        public static void Main() {
        int i = 0;
        F(i++, i++, i++);
  }
}
```

输出结果：

```
x = 0, y = 1, z = 2
```

7.2.1 加法运算

加法操作符可以运用于整数类型、实数类型、枚举类型、字符串类型和代表类型。这是通过操作符重载实现的，我们将在第十一章讲述操作符重载的内容，这里我们只需要知道这些操作符可以对不同类型的变量进行运算就可以了。加法操作符实际上定义了以下原型用于整数和浮点数运算：

```
int operator +(int x, int y);
uint operator +(uint x, uint y);
long operator +(long x, long y);
ulong operator +(ulong x, ulong y);
float operator +(float x, float y);
double operator +(double x, double y);
decimal operator +(decimal x, decimal y);
```

我们知道，在数学运算中结果可能是正无穷大、负无穷大，也可能结果不存在。在 C#中，这种情况的处理按照了国际上 IEEE 754 算法的规则。表 7-2 给出了在两个数相加时，操作数与目标类型的非零有限值、零值、无限值和 NaN 值（空值）的所有可能的组合。表中 x 和 y 是非零的有限值，z 是“x+y”的运算结果。如果 x 和 y 数值相同，但符号相反，则 z 为正零。如果“x+y”结果太大，，目标类型表示 z 是被认作和“x+y”符号相同的无穷。如果“x+y”大小，目标类型也无法表示，则 z 为和“x+y”同符号的零值。

表 7-2

	y	+0	–0	+∞	–∞	NaN
x	z	x	x	+∞	–∞	NaN
+0	y	+0	+0	+∞	–∞	NaN
–0	y	+0	–0	+∞	–∞	NaN
+∞	+∞	+∞	+∞	+∞	NaN	NaN
–∞	–∞	–∞	–∞	NaN	–∞	NaN

NaN	NaN	NaN	NaN	NaN	NaN	NaN

枚举型加法

对于枚举类型的变量，加法操作符的原型是：

```
E operator +(E x, U y);
E operator +(U x, E y);
```

此处 E 是枚举类型，U 是 E 的基类型。该运算等价于(E)((U)x + (U)y)

程序 7-2 演示了枚举型的加法运算。

程序清单 7-2：

```
using System;
enum Weekday{
   Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday
};
class Test
{
   public static void Main()  {
      Weekday day1=Weekday.Sunday;
      Weekday day2=Weekday.Saturday;
      Weekday day3=day1+6;
      Console.WriteLine(day1);
      Console.WriteLine(day2);
      Console.WriteLine(day3);
   }
}
```

程序输出结果为：

```
0
6
6
```

字符串加法

对于 `Object` 与 `String` 类型也可以进行加法运算，并且返回值总是 `String` 类型，这时加法操作符的原型是：

```
string operator +(string x, string y);

string operator +(string x, object y);

string operator +(object x, string y);
```

比如，字符串“Welcome ”和“to you ”相加的结果就是“Welcome to you”。

代表合并

加法操作符还可以作用于 delegate 类型的变量，这时我们称之为合并。原型为：

```
D operator +(D x, D y);
```

其中 D 是一个 delegate 类型。

式子中如果两个操作数是同一 delegate 类型 D 时，则加法操作符执行代表合并运算。如果第一个操作数为 null，那么结果是第二个操作数的值。反之，如果第二个操作数为 null 则结果是第一个操作数的值。

7.2.2 减法运算

减法操作符同样可以运用于整数类型、实数类型、枚举类型、字符串类型和代表类型。它的使用规则和加法操作符大体上没有什么区别。表 7-3 给出了对应的算法规则。

表 7-3

	y	+0	–0	+∞	–∞	NaN
x	z	x	x	–∞	+∞	NaN
+0	–y	+0	+0	–∞	+∞	NaN
–0	–y	–0	+0	–∞	+∞	NaN
+∞	+∞	+∞	+∞	NaN	+∞	NaN
–∞	–∞	–∞	–∞	–∞	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

枚举型减法

每种枚举型隐含下列预定义操作符。

```
U operator –(E x, E y);
```

其中 E 是枚举类型，U 是 E 的基类型。

该操作符等价于“(U)((U)X-(U)Y)”运算。换句话说，该操作符计算“X 和 Y 的”字数值的差，结果类型是枚举的基类型。

```
E operator –(E x, U y);
```

该操作符等价于(E)((U)x – y)。也就是说，该操作符指的是从枚举类型值中减去一个枚举基类型的值。

代表移去

加法操作符作用于 delegate 类型的变量时称为移去。原型为：

```
D operator –(D x, D y);
```

其中 D 是一个 delegate 类型。

7.2.3 乘法运算

乘法操作符用于执行整数和实数的乘法运算。它的算法规则见 7-4。

表 7-4

	+y	–y	+0	–0	+∞	–∞	NaN
+x	z	–z	+0	–0	+∞	–∞	NaN
–x	–z	z	–0	+0	–∞	+∞	NaN
+0	+0	–0	+0	–0	NaN	NaN	NaN
–0	–0	+0	–0	+0	NaN	NaN	NaN
+∞	+∞	–∞	NaN	NaN	+∞	–∞	NaN
–∞	–∞	+∞	NaN	NaN	–∞	+∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

7.2.4 除法运算

除法运算的算法规则见表 7-5。

表 7-5

	+y	–y	+0	–0	+∞	–∞	NaN
+x	z	–z	+∞	–∞	+0	–0	NaN
–x	–z	z	–∞	+∞	–0	+0	NaN
+0	+0	–0	NaN	NaN	+0	–0	NaN
–0	–0	+0	NaN	NaN	–0	+0	NaN
+∞	+∞	–∞	+∞	–	NaN	NaN	NaN
–∞	–∞	+∞	–∞	∞	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

在除法运算过程中，默认的返回值的类型与精度最高的操作数类型相同。比如，5/2 的结果为 2，而 5.0/2 结果为 2.5。如果两个整数类型的变量相除又不能整除的话，返回的结果是不大于相除之值的最大整数。看下面的例子。

程序清单 7-3：

```
using System;
```

```
class Test
{
   public static void Main()   {
      Console.WriteLine(5/3);
      Console.WriteLine(4/3);
      Console.WriteLine((5/3)==(4/3));
      Console.WriteLine(5.0/3);
      Console.WriteLine(4.0/3);
      Console.WriteLine((5.0/3)==(4.0/3));
   }
}
```

输出将是：

```
1
1
True
1.66666666667
1.33333333333
False
```

7.2.5 求余运算

“/”操作符用来求除法的商，而“%y”操作符则用来求除法的余数。C#中的求模运算即适用于整数类型，也同样适用于浮点数类型和十进制类型。例如，`5%3` 的结果为 2，`5%1.5` 的结果为 0.5。

7.3 赋值操作符和赋值表达式

赋值就是给一个变量赋一个新值。C#中提供的赋值表达式有：

`=` `+=` `-=` `*=` `/=` `%=` `&=` `|=` `^=` `<<=` `>>=`

赋值的左操作数必须是一个变量，属性访问器或索引访问器的表达式。

C#中可以对变量进行连续赋值，这时赋值操作符是右关联的，这意味着从右向左操作符被分组。例如，形如 a = b = c 的表达式等价于 a = (b = c)。

如果赋值操作符两边的操作数类型不一致，那就先要进行类型转换。

7.3.1 简单赋值

“=”操作符被称为简单赋值操作符。在一个简单赋值中，右操作数必须为某种类型的表达式，且该类型必须可以隐式地转换成左操作数类型。该运算将右操作数的值赋给作为左操作数的变量、属性或者索引器类型。简单赋值表达式的结果是被赋给左操作数的值。结果类型和左操作数的类型相同，且总是值类型。

7.3.2 复合赋值

形如 x *op*= y 的运算可以处理成形如 x *op* y 的二进制操作符重载方法。比如：

```
x += 5;     //等于  x = x + 5
x %= 3;     //等于  x = x % 3
x *= y+1;   //等于  x = x*(y+1)
```

复合赋值进行的步骤如下：

（1）如果所选操作符的返回类型可以隐式转换成 x 的数据类型，执行 x = x *op* y 的运算，除此之外，仅对 x 执行一次运算。

（2）否则，所选操作符是一个预定义操作符，所选操作符的返回值类型可以显式地转换成 x 的类型，且 y 可以隐式地转换成 x 的类型，那么该运算等价于 x = (T)(x *op* y) 运算，这里 T 是 x 的类型，除此之外，x 仅被执行一次。

（3）否则，复合赋值是无效的，且会产生编译时错误。

7.4 关系操作符和关系表达式

关系运算实际上是逻辑运算的一种，我们可以把它理解为一种“判断”，判断的结果要么是“真”，要么是“假”，也就是说关系表达式的返回值总是布尔值。C#定义关系操作符的优先级低于算术操作符，高于赋值操作符。

7.4.1 比较运算

C#中定义的比较操作符有：

- == 等于
- != 不等于
- < 等小于
- > 等大于
- <= 小于或等于
- >= 大于或等于

整数与实数

对于整数类型和实数类型，这六种比较操作符都可以适用。根据 IEEE 754 标准，比较运算符合下面的规则：

- 如果有一个操作数为 NAN(空)那么除“!=”外所有操作符结果为 false，“!=”的运算结果为“true”。对于任何两个操作数“x!=y”总等价于“!(x==y)”。可是当一个或两个操作数为 NaN(空)时，”<，>，<=，和>=操作符的结果和其相反操作符的逻辑非的结果是不同的。如，x 或 y 为 NaN，则 x<y 总是 false 的，但是!(x==y)为 true。
- 两个操作数不是 NaN 时，操作符根据下列顺序

–∞ < –max < … < –min < –0.0 == +0.0 < +min < … < +max < +∞

（其中 min 和 max 分别是指浮点格式所能表示的最小和最大的有限值）。

注意，在 C#中:

- 正零和负零被认为是相等的。
- 负无穷被认为小于其它任何值，只是等于另一个负无穷。
- 正无穷被认为大于其它任何值，只是等于另一个正无穷。

布尔类型

对于布尔类型的比较操作符实际上只有两种：

```
bool operator ==(bool x, bool y);
bool operator !=(bool x, bool y);
```

如果 x 和 y 都为 true 或 false，则“==”的结果为 true，否则为 false。

相反的，如果 x 和 y 都为 true 或都为 false，则“!=”的结果为 false，否则结果为 true。当操作数为 bool 类型时，“!=”操作符和 ^ 操作符有相等的结果。

枚举类型

每一种枚举类型隐式地提供下列预比较操作符。

```
bool operator ==(E x, E y);
bool operator !=(E x, E y);
bool operator <(E x, E y);
bool operator >(E x, E y);
bool operator <=(E x, E y);
bool operator >=(E x, E y);
```

执行“x op y”的结果，这里 x 和 y 是一个枚举类型 E，其基类为 U 的表达式，op 表示比较操作符，其等价于((U)x) *op* ((U)y)，即枚举类型比较操作符简单地比较两个操作数的基类型值。

引用类型

预定义引用类型等价操作符有：

```
bool operator ==(object x, object y);
bool operator !=(object x, object y);
```

该操作符返回两个引用类型是否等价的结果。

字符串

预定义字符串等价操作符有：

```
bool operator ==(string x, string y);
bool operator !=(string x, string y);
```

下列条件之一成立则认为两个字符串值相等：

- 两个字符串值都为 null，
- 两个字符串是字符串长度相同、对应的字符序列也相同的非空字符串。

注意：字符串等价操作符比较的是两个字符串的值，而不是字符串引用。当两个单独的字符串实例含有相同的字符串序列，则认为这两个字符串的值相等。引用类型等价操作符可以用于比较两个字符串引用，不是比较两个字符串的值。

代表类型

每一个代表元类型隐式地提供下列预定义比较操作符：

```
bool operator ==(System.Delegate x, D y);
bool operator ==(D x, System.Delegate y);
bool operator !=(System.Delegate x, D y);
bool operator !=(D x, System.Delegate y);
```

7.4.2 is 操作符

is 操作符被用于动态地检查运行时对象类型是否和给定的类型兼容。运算“e is T”的结果，其中，e 是一个表达式，T 是一个类型，返回值是一个布尔值。它表示，e 是否能通过引用转换、装箱转换或拆箱转换，成功地转换于 T 类型。

程序清单 7-4：

```
using System;

class Test

{

   public static void Main()  {

      Console.WriteLine(1 is int);

      Console.WriteLine(1 is float);

      Console.WriteLine(1.0 is float);

      Console.WriteLine(1.0 is double);

   }

}
```

输出为：

True

False

False

True

7.4.3 as 操作符

as 操作符用于通过引用转换或装箱转换将一个值显式地转换成指定的引用类型。不像显式类型转换，as 不会产生任何异常。如果转换不可以进行，那么结果值为 null。形如“e as T”的运算，e 定是一个表达式且 T 是一个引用类型。返回值的类型总是 T 的类型，并且结果总是一个值。

比如，当你在程序中写了下面的语句：

string s = ‘a’ as string;

虽然，字符型不能转换为字符串类型，程序仍然可以编译通过，只是有一个警告：

The given expression is never of the provided (‘string’) type.

7.4.4 关系表达式

用关系操作符将两个表达式连接起来的式子就是关系表达式。关系表达式的值就是关系操作符的返回值，即一个布尔值。关系表达式可以再作为关系操作符的操作数，也可以作为布尔值赋给赋值表达式。例如，下面都是合法的关系表达式：

a > b, (a=b)>c, (a>b)>(c<d), (a>b)==c

7.5 逻辑操作符和逻辑表达式

7.5.1 逻辑操作符

C#语言提供了三种逻辑操作符：

- && 逻辑与
- || 逻辑或
- ！ 逻辑非

其中，逻辑与和逻辑或都是二元操作符，要求有两个操作数。而逻辑非为一元操作符，只有一个操作数。它们的操作数都是布尔类型的值或者表达式。操作数为不同的组合时，逻辑操作符的运算结果可以用逻辑运算的“真值表”来表示，见表 7-6：

表 7-6 真值表

a	b	!a	a&&b	a||b

true	true	false	true	true
true	false	false	false	true
false	true	true	false	true
false	false	true	true	false

如果表达式中同时存在着多个逻辑运算符，逻辑非的优先级最高，逻辑与的优先级高于逻辑或。

7.5.2 逻辑表达式

用逻辑操作符将关系表达式或布尔值连接起来就是逻辑表达式。逻辑表达式的值仍然是一个布尔值。

在逻辑表达式的求值过程中，不是所有的逻辑操作符都被执行。有时候，不需要执行所有的操作符，就可以确定逻辑表达式的结果。只有在必须执行下一个逻辑操作符后才能求出逻辑表达式的值时，才继续执行该操作符。这种情况我们称为逻辑表达式的“短路”。

假设 a 是一个布尔值或逻辑表达式，bool-exp 是一个逻辑表达式，那么：

- a&&(bool-exp)　只有 a 当为 true 时，才继续判断值。如果 a 为 false 时，逻辑表达式的值已经确定为 false，不需要继续求值。
- a||(bool-exp)　只有 a 当为 false 时，才继续判断的值。如果 a 为 true 时，逻辑表达式的值已经确定为 true，不需要继续求值。

在熟练地掌握逻辑操作符和关系操作符以后，就可以使用逻辑表达式来表示各种复杂的条件。例如，给出一个年份，要判断它是不是一个闰年。我们知道，闰年的条件是：是 400 的倍数，或者是 4 的倍数但不是 100 的倍数。设年份为 year，闰年与否就可以用一个逻辑表达式来表示：

```
(year % 400)==0 || ((year % 4)==0 && (year % 100)!=0)
```

7.6 位 运 算

我们知道，任何信息在计算机中都是以二进制的形式保存的。位操作符就是对数据按二进制位进行运算的操作符。C#语言中的位操作符有：

- & 与
- | 或
- ^ 异或
- ~ 取补
- << 左移
- >> 右移

其中，取补只有一个操作数，而其它的位操作符都有两个操作数。这些运算都不

会产生溢出。位操作符的操作数为整型或者是可以转换为整型的任何其它类型。

与运算

操作数按二进制位进行与运算，运算规则为：

0&0=0

0&1=0

1&0=0

1&1=1

这说明，除了两个位均为 1，与运算结果为 1，其它情况下与运算结果均为 0。比如，2 和 10 进行与运算：

2 的二进制表示：　00000010

10 的二进制表示：　00001010

与运算的结果：　　00000010

1 所以，2&10 的结果为 2。

或运算

操作数按二进制位进行与运算，运算规则为：

0|0=0

0|1=1

1|0=1

1|1=1

这说明，除了两个位均为 0，或运算结果为 0，其它情况下或运算结果均为 1。比如，2 和 10 进行或运算：

2 的二进制表示：　00000010

10 的二进制表示：　00001010

或运算的结果：　　00001010

所以，2|10 的结果为 10。

异或运算

操作数按二进制位进行与运算，运算规则为：

0^0=0

0^1=0

1^0=0

1^1=1

这说明，当两个位相同时，异或运算结果为 0；不相同时异或运算结果为 1。比如，2 和 10 进行异或运算：

2 的二进制表示：　00000010

10 的二进制表示：　00001010

异或运算的结果：　00001000

所以，2^10 的结果为 8。

取补运算

取补运算对操作数的每一位取补，如对 10 取补结果为：

10 的二进制表示： 00001010

与运算的结果： 11110101

这个二进制对应的具体值与数据类型有关，下面的例子清楚地说明了这一点。

程序清单 7-5：

```
using System;

class Test

{

   public static void Main()   {

      short a = 10;

      ushort b = 10;

      int   c = 10;

      uint d = 10;

      Console.WriteLine(~10);

      Console.WriteLine("short: {0}",~a);

      Console.WriteLine("ushort: {0}",~b);

      Console.WriteLine("int: {0}",~c);

      Console.WriteLine("uint: {0}",~d);

   }

}
```

正确的输出是：

```
-11
short: -11
ushort: -11
int: -11
uint: 4294967285
```

移位运算

左移运算将操作数按位左移，高位被丢弃，低位顺序补 0。比如 10 的二进制为 00001010，左移一位为 00010100（20），左移二位为 00101000（40）。

右移运算时，如果操作数 x 是 int 或 long 型 时，x 的低位被丢弃，其它各位顺序依次右移，如果 x 是非负数，最高位设成零；如果 x 为负数，则最高位设为 1。而当 x 的类型为 uint 或 ulong 型时，x 的低位将被丢弃，其它各位顺序依次右移，高位设为 0。比如：

程序清单 7-6：

```
using System;
class Test
{
   public static void Main()   {
      int x = 16;
      Console.WriteLine(x);
      int y = x>>2;
      Console.WriteLine(y);
      y = y>>2;
      Console.WriteLine(y);
      y = y>>2;
      Console.WriteLine(y);
   }
}
```

上面这段程序的输出为：

```
16
4
1
0
```

如果把上面 x 的初始值设为-16，则程序的输出为：

```
-16
-4
-1
-1
```

7.7 其它特殊操作符

7.7.1 三元操作符

三元操作符“?:”，有时也称为条件操作符。

对条件表达式 b? x: y，先计算条件 b，然后进行判断。如果 b 的值为 true，计算 x 的值，运算结果为 x 的值；否则，计算 y，运算结果为 y 的值。一个条件表达式从不会又计算 x，也计算 y。条件操作符是向右关联的，也就是说，从左向右分组计算。例如：表达式 a? b: c? d: e 将按 a? b: (c? d: e)形式执行。

？：的每一个操作数必须是一个可以隐式转换成 bool 型的表达式或者执行操作符 true 类型的表达式，如果上述这两个条件一个也不满足，则发生运行时错误。

？：的第二和第三个操作数控制了条件表达式的类型。设 x 和 y 分别是第二个和第三个操作数的类型，那么

- 如果 x 和 y 为同一类型，则该类型即是条件表达式的类型。
- 否则，如果从 x 到 y 存在一个隐式转换，但不存在 y 到 x 的转换，那么 y 是条件表达式的类型。
- 否则，如果从 y 到 x 存在一个隐式转换，但不存在 x 到 y 的转换，那么 x 是条件表达式的类型。
- 否则，没有定义任何表达式类型，发生编译时错误。

7.7.2 自增和自减操作符

自增操作符 ++ 对变量的值加 1，而自减操作符 -- 对变量的值减 1。它们适合于 sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal 和任何 enum 类型。比如，假设一个整数 x 的值为 9，那么执行 x++之后的值为 10。

注意：自增和自减操作符的操作数必须是一个变量，一个属性访问器或一个索引指示器访问器，而不能是常量或者其它的表达式。比如 5++和（x+y）--都是非法的。如果操作数是一个访问器，那么这个访问器必须同时支持读和写。

自增和自减操作符又有前后缀之分。对于前缀操作符，遵循的原则是“先增减，后使用”，而后缀操作符则正好相反，是“先使用，后增减”。我们用例子来说明这个问题。

程序清单 7-7：

```
using System;

class Test

{

   public static void Main()   {

      int x = 5;

      int y = x++;

      Console.WriteLine(y);
```

```
        y = ++x;
        Console.WriteLine(y);
    }
}
```

第一次是先使用后加，所以输出为 5，第二次先加后使用，输出为 7。

再看一个例子。

程序清单 7-8：

```
using System;
class Test
{
    public static void Main()  {
        int x = 5;
        Console.WriteLine((x++)+(x++)+(x++));
        int y = (x++)+(x++)+(x++);
        Console.WriteLine(y);
    }
}
```

程序运行的结果是：

```
18
27
```

读者可能对输出感到难以理解。其实编译的过程是这样的：编译器先对整个表达式扫描，先把 x 的原值取出来，对表达式求解后，再对每个 x 执行++运算。

7.7.3 new 操作符

new 操作符用于创建一个新的类型实例。它有三种形式：

- 对象创建表达式，用于创建一个类类型或值类型的实例。
- 数组创建表达式，用于创建一个数组类型实例。
- 代表创建表达式，用于创建一个新的代表类型实例。

new 操作符暗示一个类实例的创建，但不一定必须暗示动态内存分配，这和 C++中对指针的操作不同。例如，下面三个式子分别创建了一个对象、一个数组和一个代表实例：

```
class  A{};  A a = new A;
```

```
int[] int_arr = new int[10];
delegate double DFunc(int x);       DFunc f = new DFunc(5);
```

7.7.4 typeof 操作符

typeof 操作符用于获得系统原型对象的类型。

例如：

程序清单 7-9：

```
using System;

class Test
{
  static void Main() {
        Console.WriteLine（typeof(int)）;
        Console.WriteLine（typeof(System.Int32)）;
        Console.WriteLine（typeof(string)）;
        Console.WriteLine（typeof(double[])）;
  };
}
```

产生如下输出。

```
Int32
Int32
String
Double[]
```

这表明 int 和 system.int32 是同一类型。

7.7.5 checked 和 unchecked 操作符

checked 和 unchecked 操作符用于整型算术运算时控制当前环境中的溢出检查。下列运算参与了 checked 和 unchecked 检查：

- 预定义的 ++ 和 -- 一元操作符，当其操作数类型为整型时。
- 预定义的 - 一元操作符，当其操作数为整型数时。
- 预定义的 ＋、－、＊、/ 等二元操作符，当两个操作数数据类型都是整型。
- 从一种整型到另一种整型地显式数据转换。

当上述运算之一产生一个目标类型无法表示的大数时，在使用了 checked 操作符的表达式中，如果运算是一个常量表达式，则产生运行时错误，否则，当运行时执行该运算时会溢出异常。而在使用了 unchecked 操作符的表达式中，返回值被截掉不符合目标类型的高位。

如果非常量表达式没有被任何 checked 或 unchecked 操作符或语句所包括时，运行

时计算该表达式的值，是否会产生溢出，依据于外部因素，如编译器状态、执行环境参数等。而对于一个常量表达式而言，总是默认为进行溢出检查。

使用了 unchecked 操作符后，溢出的发生不会导致编译错误。但这往往会出现一些不可预期的结果，所以使用 unchecked 操作符要小心。

下面的例子说明了 checked 和 unchecked 操作符的用法：

```
class Test
{
  static int x = 1000000;
  static int y = 1000000;

        static int F() {
        return checked(x * y); // 编译错误，抛出异常
  }

        static int G() {
        return unchecked(x * y);            // 返回值为 -727379968
  }

        static int H() {
        return x * y;             // 依赖于编译时的默认情况
  }
}
```

由于编译时没有对任何表达式进行计算，因此不会产生编译错误。当程序运行时，调用 F 方法会抛出一个 OverFlowException 异常，而调用 G 方法则返回－727379968（超出结果范围的低 32 位）。H 方法是否抛出异常则要看编译环境的默认情况了，调用它的结果有可能与 F 相同，也有可能与 G 相同。而对于常量表达式则是另一种情况：

```
class Test
{
  const int x = 1000000;
  const int y = 1000000;

        static int F() {
        return checked(x * y);        // 编译错误，溢出
  }

        static int G() {
        return unchecked(x * y);            // 返回值为-727379968
  }
```

```
        static int H() {
        return x * y;                          // 编译错误，溢出
    }
}
```

当计算 F 和 H 方法中的常量表达式时发生溢出，由于是在 checked 环境中，那么在编译时就报告错误。当计算 G 方法中的常量表达式时，也会发生溢出，但不会报告错误。

checked 和 unchecked 操作符只影响置于其后括号之中运算的溢出检查。它们不会影响作为表达式结果的被调用的方法成员。例如：

```
class Test
{
  static int F(int x, int y) {
        return x * y;
  }

        static int G() {
        return checked(F(1000000, 1000000));
  }
}
```

其中，G 方法中的 checked 不会影响函数 F 方法中的 X＊Y 运算，因此，X＊Y 运算在默认溢出检查的上下文中。

7.8 小　　结

C#语言为我们提供了丰富的操作符和表达式，用于各种情况下对数据的运算和处理。在本章中，我们重点介绍了算术操作符和算术表达式、赋值操作符和赋值表达式、关系操作符和关系表达式、逻辑操作符和逻辑表达式的使用，并且介绍了位运算的概念及其在 C#程序中的实现方式。我们还介绍了其它的一些操作符，如三元操作符、自增自减操作符、new 操作符、typeof 操作符、checked 和 unchecked 操作符等。

C#操作符和表达式的应用可以十分灵活，利用它们我们可以巧妙地处理许多问题。但它们也常常会出现一些混乱 ，在使用时需要小心谨慎。建议读者多通过编程实践来掌握这些操作符和表达式的用法。

复习题

（1）C#中的操作符可以分为哪几种？

（2）编写一个程序，利用求余运算完成 24 小时制和 12 小时制之间的转换。

（3）C#程序是如何进行复合赋值的？

（4）试说明关系运算和逻辑运算的区别。

（5）写出下面算术表达式的值，设 a=10，b=5.5，c=3：

```
int(a/c) + a/4 + b*c;

(a % c) - int(a % b) - (int)a % b;
```

（6）写出下面逻辑表达式的值，设 a=true，b=true，c=false，d=3，e=2.5：

```
a && (b || !c);

!(a && (d==int(e)) && (e/0.5)==5
```

（7）计算机在进行移位运算时的执行速度要比进行乘法运算时快得多。试编写一个方法，用移位运算代替 2 的指数幂运算。

（8）分析下面程序运行的结果：

```
using System;

class Test

{

   public static void Main()   {

      int x = 9;

      Console.WriteLine((x--)+(x--)+(x--));

      Console.WriteLine(x);

      int y = (--x)+(--x)+(--x);

      Console.WriteLine(y);

      Console.WriteLine(x);

   }

}
```

（9）试给出 is 和 as 操作符的使用实例。

（10）说明 checked 和 unchecked 操作符对表达式的运算结果会产生什么影响。

第八章　流 程 控 制

到目前为止，我们的程序还只能按照编写的顺序执行，中途不能发生任何变化。

然而，实际生活中并非所有的事情都是按部就班地进行，程序也是一样。为了适应自己的需要，我们经常必须要转移或者改变程序执行的顺序，达到这些目的的语句叫作流程控制语句。

和大多数编程语言相似，在程序模块中，C#可以通过条件语句控制程序的流程，从而形成程序的分支和循环。C#中提供了以下控制关键字：

- 选择控制：if、else、switch、case
- 循环控制：while、do、for、foreach
- 跳转语句：break、continue
- 编译控制：#if、#elif、#else、#endif
- 异常处理：try、catch、finally

8.1　条 件 语 句

当程序中需要进行两个或两个以上的选择时，可以根据条件判断来选择将要执行的一组语句。C#提供的选择语句有 if 语句和 switch 语句。

8.1.1　if 语句

if 语句是最常用的选择语句，它根据布尔表达式的值来判断是否执行后面的内嵌语句。

格式：

```
if (boolean-expression) embedded-statement
```

或

```
if (boolean-expression) embedded-statement
else    embedded-statement
```

当布尔表达式的值为真，则执行 if 后面的内嵌语句 `embedded-statement`。为假则程序继续执行。如果有 else 语句，则执行 else 后面的内嵌语句，否则继续执行下一条语句。

例如下面的例子用来对一个浮点数 x 进行四舍五入，结果保存到一个整数 i 中：

```
if (x – int(x) > 0.5)
{
 i = int(x)+ 1;
}
```

```
else
{
 i = int(x);
}
```

如果 if 或 else 之后的嵌套语句只包含一条执行语句，则嵌套部分的大括号可以省略。如果包含了两条以上的执行语句，对嵌套部分一定要加上大括号。

如果程序的逻辑判断关系比较复杂，通常会采用条件判断嵌套语句。if 语句可以嵌套使用，即在判断之中又有判断。具体形式如下：

```
if (boolean-expression)
{
 if (boolean-expression)
 {……};
 else
 {……};
……
}
else
{
 if (boolean-expression)
 {……};
 else
 {……};
……
}
```

此时应该注意，每一条 else 与离它最近且没有其它 else 与之对应的 if 相搭配。比如有下面一条语句：

```
if(x)if(y)F();else G();
```

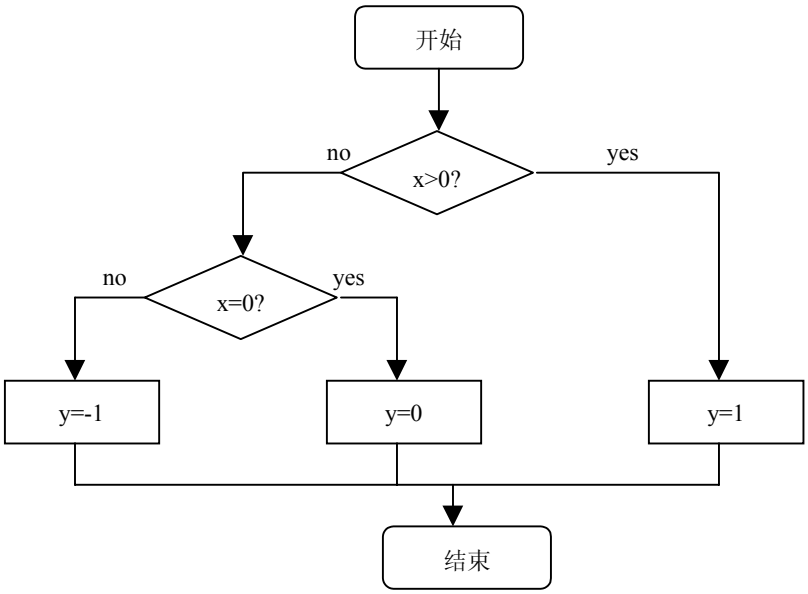
它实际上应该等价于下面的写法：

```
if(x){
   if(y){
      F();
   }
   else{
      G();
   }
}
```

举一个例子，设有一个数学函数的表达式为：

$$y=\begin{cases}-1,(x<0)\\0,(x=0)\\1,(x>0)\end{cases}$$

则相应的程序流程图为：



那么使用条件判断语句写的代码如下：

```
if(x>0){
    y = 1;
}
else{
    if(x==0){
        y = 0;
    }
    else{
        y = -1;
    }
}
```

注意：C#的 if 语句与 C、C++不同，即 if 后的逻辑表达式必须是布尔类型的。

请看一个判断是否往应用程序传送参数的例子。

程序清单 8-1：

```
using System;
class Test
{
  static void Main(string[] args) {
```

```
        if (args.Length == 0)
          Console.WriteLine("No arguments were provided");
        else    Console.WriteLine("Arguments were provided");
      }
    }
```

其中 args.Length == 0 是一个布尔表达式。但是，对于 C 或者 C++程序员，他们可能会习惯于编写像这样的代码：

```
    if (args.Length)
    {……}
```

这在 C#中是不允许的，因为 if 语句仅允许布尔（bool）数据类型的结果，而字符串的 Length 属性对象返回一个整形（integer）。编译器将出现报告错误信息。

8.1.2 switch 语句

if 语句每次判断只能实现两条分支，如果要实现多种选择的功能，那么可以采用 switch 语句。switch 语句根据一个控制表达式的值选择一个内嵌语句分支来执行。它的一般格式为：

```
switch(controlling-expression)
{
  case constant-expression:
    embedded-statements
  default:
    embedded-statements
}
```

switch 语句的控制类型,即其中控制表达式(controlling-expression)的数据类型可以是 sbyte，byte，short，ushort，uint，long，ulong，char，string 或枚举类型(enum-type)。每个 case 标签中的常量表达式(constant-expression)必须属于或能隐式转换成控制类型。如果有两个或两个以上 case 标签中的常量表达式值相同，编译时将会报错。switch 语句中最多只能有一个 default 标签。

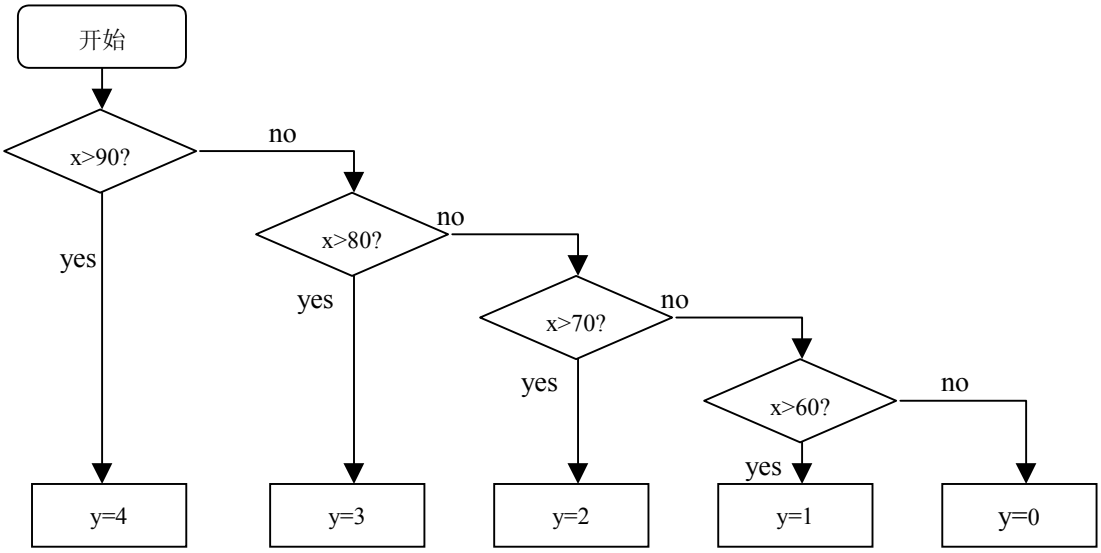
我们举一个例子来说明 switch 语句是如何实现程序的多路分支的。

假设考查课的成绩按优秀、良好、中等、及格和不及格分为五等，分别用 4、3、2、1、0 来表示，但实际的考卷为百分制，分别对应的分数为 90-100，80-90，60-80，60 分以下。下面的程序将考卷成绩 x 转换为考查课成绩 y。我们先看流程图。

代码如下：

```
int x = int(x/10);
switch(x)
{
 case 10: y=4;break;
 case 9:    y=4;break;
```

```
        case 8:    y=3;break;
        case 7:    y=2;break;
        case 6:    y=1;break;
        default: y=0;
      }
```



下面的例子判断传递给应用程序的参数的有无、位数。

程序清单 8-2：

```
using System;
class Test
{
  public static void Main(string[] args) {
    switch (args.Length) {
      case 0:
        Console.WriteLine("No arguments were provided");
        break;
      case 1:
        Console.WriteLine("One arguments was provided");
        break;
      default:
        Console.WriteLine("{0} arguments were provided");
        break;
    }
  }
}
```

使用 switch 语句时需注意以下几点：

不准遍历

c 和 c++语言允许 switch 语句中 case 标签后不出现 break 语句,但 c#不允许这样,它要求每个标签项后使用 break 语句或跳转语句 goto,即不允许从一个 case 自动遍历到其它 case，否则编译时将报错。

一个程序用于计算一年中已度过的天数，month 表示月份，day 表示日期，计算结果保存在 total 中。为简便起见，我们把闰年的情况排除在外。C 和 C++程序员会利用一点技巧来实现这个程序：

```
total=365;
switch(month){
        case 1: total-=31;
        case 2: total-=28;
        case 3: total-=31;
        case 4: total-=30;
        case 5: total-=31;
        case 6: total-=30;
        case 7: total-=31;
        case 8: total-=31;
              case 9: total-=30;
              case 10: total-=31;
              case 11: total-=30;
         case 12: total-=31;
              default: total+=day;
 }
```

然而这种写法在 C#中是不允许的。有经验的程序员会利用这个特点，但是很难保证任何人在编程时都不会忘记在 case 后加上 break 语句，这时往往会造成一些不易察觉的错误。所以在 C#中，如果标签项没有出现 break 语句或跳转语句 goto，编译器将会要求程序员加上。

如果想象 c,c++那样,执行完后继续遍历其它的语句,那也不难,只需要明确地加入这两条跳转语句即可:

- goto case label：跳至标签语句执行
- goto default：跳至 default 标签执行

那样，我们将上面的程序可以改写为：

```
        total=365;
 switch(month){
          case 1: total-=31; goto case 2;
          case 2: total-=28; goto case 3;
          case 3: total-=30; goto case 4;
          case 4: total-=31; goto case 5;
```

```
        case 5: total-=30; goto case 6;
        case 6: total-=31; goto case 7;
        case 7: total-=30; goto case 8;
        case 8: total-=31; goto case 9;
        case 9: total-=30; goto case 10;
        case 10: total-=31; goto case 11;
        case 11: total-=30; goto case 12;
        case 12: total-=31; goto default;
        default: total+=day;
    }
```

在避免了 c，c++中常出现的由于漏写 break 而造成的错误的同时，“不准遍历”的原则还使得我们可以任意排列 switch 语句中的 case 项而不会影响 switch 语句的功能。

另外，一般说来每个 switch 项都以 break，goto case 或 goto default 结束，但实际上任何一种不导致“遍历”的结构都是允许的，例如 throw 和 return 语句同样可以跳出控制之外，因而下例是正确的：

```
switch(i){
case 0:
    while(true)    F();
case 1:
    throw new ArgumentException();
case 2:
    return;
}
```

把字符串当成常量表达式

VB 的程序员可能已经习惯把字符串当成常量表达式来使用，但 C 和 C++却不支持这一点。但是，C#的 switch 语句与 c,c++的另一个不同点是， C#可以把字符串当成常量表达式来使用。所以 switch 语句的控制类型可以是 string 类型。

下面的例子是实现浮动窗口提示。在 Windows 操作系统中，我们把鼠标移到某一个控件上停留几秒钟，将会出现一个浮动提示，说明该控件的作用等。例子中的 GetButtonCaption 方法用于获得按钮上的文字。ShowMessage 表示在浮动提示窗口中显示信息。

```
string text = GetButtonCaption();
switch(text)
{
    case “OK”: ShowMessage（“save the change and exit”）；break;
    case “Retry”: ShowMessage（“return and retry”）；break;
    case “Abort”: ShowMessage（“Abort the change and exit”）；break;
    case “HELP”: ShowMessage（“Get help from system”）；break;
```

}

实际上，在老版本的 C 语言中，switch 语句的控制类型只允许是整数表达式或字符型表达式，而 ANSI 标准放宽了这一要求。C#中是对 switch 语句的控制类型的又一扩展。而且在 C#中 case 标签的引用值允许是 null。

8.2 循 环 语 句

循环语句可以实现一个程序模块的重复执行，它对于我们简化程序，更好地组织算法有着重要的意义。C#为我们提供了四种循环语句，分别适用于不同的情形：

- while 语句
- do-while 语句
- for 语句
- foreach 语句

8.2.1 while 语句

while 语句有条件地将内嵌语句执行 0 遍或若干遍。语句的格式为：

While (boolean-expression) embedded-statement

它的执行顺序是:

（1）计算布尔表达式 boolean-expression 的值；

（2）当布尔表达式的值为真时，执行内嵌语句 embedded-statement 一遍，程序转至第 1 步；

（3）当布尔表达式的值为假时，while 循环结束；

我们来看一个简单的例子，该例在数组中查找一个指定的值，如找到就返回数组下标，否则返回并报告：

程序清单 8-3：

```
using System;
class Test
{
  static int Find(int value, int[] array)
    {
    int i = 0;
    while (array[i] != value) {
      if (++i > array.Length)
        Console.WriteLine( “Can not find”);
    }
    return i;
  }
```

```
    static void Main(){
        Console.WriteLine(Find(3, new int[] {5, 4, 3, 2, 1}));
    }
}
```

while 语句中允许使用 break 语句结束循环,执行后续语句;也可以用 continue 语句来停止内嵌语句的执行，继续进行 while 循环。

我们使用下面的程序片段来计算一个整数 x 的阶乘值：

```
long y = 1;
while(true)
{
    y *= x;
    x--;
    if(x==0){
        break;
    }
}
```

8.2.2 do-while 语句

do-while 语句与 while 语句不同的是，它将内嵌语句执行一次（至少一次）或若干次。

```
do  embedded-statement  while(boolean-expression)
```

它按如下顺序执行：

（1）执行内嵌语句 embedded-statement 一遍；

（2）计算布尔表达式 boolean-expression 的值，为 true 则回到第一步，为 false 则终止 do 循环。

在 do-while 循环语句同样允许用 break 语句和 continue 语句实现与 while 语句中相同的功能。

我们看一下如何使用 do-while 循环来实现求整数的阶乘：

```
long y = 1;
do{
 y *= x;
 x--;
}
while(x>0)
```

8.2.3 for 语句

for 语句是 C#中使用频率最高的循环语句。在事先知道循环次数的情况下，使用 for 语句是比较方便的。for 语句的格式为：

```
for(initializer;condition;iterator) embedded-statement
```

其中 initializer，condition，iterator 这三项都是可选项。initializer 为循环控制变量做初始化，循环控制变量可以有一个或多个(用逗号隔开)；condition 为循环控制条件,也可以有一个或多个语句；iterator 按规律改变循环控制变量的值。

请注意，初始化、循环控制条件和循环控制都是可选的。如果忽略了条件，你就可以产生一个死循环，要用跳转语句（break 或 goto）才能退出。

```
for (;;){
  break; //  由于某些原因
}
```

for 语句执行顺序如下:

（1）按书写顺序将 initializer 部分(如果有的话)执行一遍，为循环控制变量赋初值；

（2）测试 condition(如果有的话)中的条件是否满足；

（3）若没有 condition 项或条件满足，则执行内嵌语句一遍，按 iterator 改变循环控制变量的值，回到第二步执行；

（4）若条件不满足，则 for 循环终止。

下面的例子非常简单，打印数字从 1 到 9，但它却清楚地显示出了 for 语句是怎样工作的。

```
for (int i = 0; i < 10; i++)
    Console.WriteLine(i);
```

for 语句可以嵌套使用，帮助我们完成大量重复性、规律性的工作。

下面的例子用于打印杨辉三角形。

程序清单 8-4：

```
using System;
class Test
{
    public static void Main()
    {
        int[,] a = new int[5,5];
        a[0,0] = 1;
        for(int i=1;i<=5;i++){
            a[i,0] = 1;
            a[i,i] = 1;
            for(int j=1;j<i;j++){
                a[i,j]=a[i-1,j-1]+a[i-1,j];
            }
        }
        for(int i=0;i<5;i++){
            for(int j=0;j<i;j++){
```

```
                    Console.WriteLine("{0} ",a[i][j])
                }
            Console.WriteLine();
            }
        }
    }
```

运行程序的结果为：

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

还以求整数的阶乘为例，代码我们可以这样写：

```
for(long y=1;x>0;x--)
    y *= x;
```

同样，可以用 break 和 continue 语句，来和循环判断符合语句中的逻辑表达式来配合使用，到达控制循环的目的。

仍然以打印数字为例，如果要求打印除 7 以外的 0 到 9 的数字，只要在 for 循环执行到 7 时，跳过打印语句就可以了。

```
for (int i = 0; i < 10; i++) {
    if (i==7)   continue;
      Console.WriteLine(i);
  }
}
```

8.2.4 foreach 语句

foreach 语句是在 C#中新引入的，C 和 C++中没有这个语句，而 Visual Basic 的程序员应该对它不会陌生。它表示收集一个集合中的各元素,并针对各个元素执行内嵌语句。语句的格式为：

```
foreach(type  identifier  in  expression) embedded-statement
```

其中类型(type)和标识符(identifier)用来声明循环变量，表达式(expression)对应集合。每执行一次内嵌语句，循环变量就依次取集合中的一个元素代入其中。在这里，循环变量是一个只读型局部变量，如果试图改变它的值或将它作为一个 ref 或 out 类型的参数传递,都将引发编译时错误。

foreach 语句中的 expression 必须是集合类型，如果该集合的元素类型与循环变量类型不一致，则必须有一个显示定义的从集合中的元素类型到循环变量元素类型的显式转换。

集合的概念相信大家都不陌生，它表示一组相同或相似的数据项总的描述。那么在 C#中，究竟什么样的类型算是集合类型呢？我们从语法上给出集合类型的定义：

- 该类型必须支持一个形为 GetEnumerator（）的公有的非静态方法，该方法的返回类型为结构、类或接口。
- 形为 GetEumerator（）的方法返回的结构、类或接口应当包含一个公有的非静态的方法 MoveNext（），该方法的返回类型为布尔型。
- 形为 GetEumerator（）的方法返回的结构、类或接口应当包含一个公有的非静态的属性 Current，该属性可以读出。

如果一个类型同时满足以上三个条件，该类型称为集合类型。Current 属性的类型叫作该集合类型的元素类型。

我们姑且不论集合类型的具体形式，只从 foreach 语句的使用角度举一个例子。

假设 Prime 是一个满足条件的集合类型，它的元素类型为 0 到 1000 以内的质数。MyInt 是我们自定义的一个类型，其范围为 200 到 300 之间的整数。下面这段程序用于在屏幕上打印出从 200 到 300 以内的所有质数。

程序清单 8-5：

```
using System;
using System.Collections;
class Test()
{
    public static void Main()
    {
        Console.WriteLine("See the prime number:");
        foreach(MyInt x in Prime)
            Console.WriteLine("{0} ",x);
    }
}
```

顺便说一句，数组类型是支持 foreach 语句的，对于一维数组，执行顺序是从下标为 0 的元素开始，一直到数组的最后一个元素；对于多维数组，元素下标的递增是从最右边那一维开始的，依次类推。

同样，break 和 continue 可以出现在 foreach 语句中,功能不变。

8.3 条 件 编 译

在开发应用程序的过程中，使用条件编译对我们来说是必不可少的。它能让我们有条件地将部分程序代码包括进来或排除在外。

条件编译属于编译预处理的范畴，因此在进入主题之前，我们先介绍一些预处理器的相关知识。

8.3.1 使用预处理指令

与 c++不同，C#没有独立的预处理器。C#中的预处理指令（pre-processing directives）仅仅用来与 C 保持一致，而并不是编译器开始编译代码之前的一个单独的处理步骤，它是作为词法分析的一部分来执行的。

预处理指令都以#号开头并位于行首（前面可以出现空格符）。在介绍条件编译之前，我们先学习两条用于定义符号和取消符号定义的预处理指令：#define 和 #undef。

#define 指令对于有一点 c 语言知识的读者来说再熟悉不过，它非常类似于 c 中的宏定义：

```
#define COUNT
```

这里定义了符号 COUNT，它的作用域是该定义所在的整个文件。需要注意的是，符号定义必须放在所有其它语句的前面,或者说是放在所有“实代码”（real code）的前面。所以下面的代码是错误的：

```
using   System
#define   COUNT
```

再看一个例子：

```
#define A
namespace N
{
      #define B
      #if B
      class Class1 {}
      #endif
}
```

由于在第二个#define 指令之前出现了“实代码”,因而是错误的。

然而，下面的代码却是合法的，因为#if 不是“实代码”：

```
 #define A
  #if A
      #define B
  #endif
  namespace N
  {
      #if B
      class Class1 {}
      #endif
  }
```

有时侯我们想在源程序中取消某个符号的定义，这时就要用到#undef 指令：

```
#undef   COUNT
```

与#define 一样，#undef 的作用域也是定义所在的整个文件，它也必须出现在所有

“实代码”之前。下面的例子定义了一个符号，然后又取消了它，第二个#undef 指令虽然不起任何作用，但却是合法的：

```
#define A
#undef A
#undef A
```

好了，简单地介绍了预处理语句之后，我们就可以来学习如何使用定义的符号进行条件编译了。

8.3.2 条件编译

条件编译指令有以下四种：

- #if
- #elif
- #else
- #endif

这些条件编译指令用来有条件地将部分程序段包括进来或排除在外。它们和 C#中的 if 语句有类似的作用。你可以在指令中使用逻辑操作符与(&&)，或(||)和取反操作符（!）等。它们在程序中的出现的先后顺序必须是这样：

一条#if 语句（必须有）
零或多条#elif 语句
零或一条#else 语句
一条#endif 语句（必须有）

下面我们通一些例子来说明它们的用法。

```
#define Debug
class Class1
{
#if Debug
  void Trace(string s) {}
#endif
}
```

执行时，由于第一行已经使用#define 指令定义了符号 Debug ，#if 的条件满足，这段代码等同于：

```
class Class1
{
  void Trace(string s) {}
}
```

再比如：

```
#define A
```

```
#define B
#undef C
class D
    {
    #if C
        void F() {}
     #elif A && B
    void I() {}
    #else
        void G() {}
    #endif
    }
```

不难看出，它实际编译的效果等同于：

```
class C
{
  void I() {}
}
```

在这个例子里，请大家注意#elif 指令的使用，它的含义是：”else if”，使用#elif 可以在#if 指令中加入一个或多个分支。

#if 指令可以嵌套使用，例如：

```
#define Debug              // Debugging on
#undef Trace        // Tracing off
class PurchaseTransaction
{
  void Commit() {
      #if Debug
          CheckConsistency();
          #if Trace
              WriteToLog(this.ToString());
          #endif
      #endif
      CommitHelper();
  }
}
```

预处理指令如果出现在其它输入输出元素中间就不会被执行。例如下面的程序试图在定义了 Debug 时显示 hello world，否则显示 hello everyone，但结果却令人哭笑不得：

程序清单 8-6：

```
using System;
```

```
class Hello
{
  static void Main() {
        System.Console.WriteLine(@"hello,
#if Debug
        world
#else
        everyone
#endif
        ");
  }
}
```

该程序输出结果如下：

```
hello,
#if Debug
   world
#else
   everyone
#endif
```

8.3.3 发出错误和警告信息

预编译和条件编译指令可以帮助我们在程序执行过程中发出编译的错误或警告，相应的指令是#warning 和#error，下面的程序展示了它们的用法。

程序清单 8-7：

```
#define DEBUG
#define RELEASE
#define DEMO VERSION
#if DEMO VERSION && !DEBUG
    #warning    you are    building a demo version
#endif
#if DEBUG && DEMO VERSION
    #error    you cannot build a debug demo version
#endif

using System;
class Demo
{
    public static void Main()
```

```
        {
            Console.WriteLine("Demo application");
        }
    }
```

在本例中，当你试图创建一个演示版时，会出现一个警告信息：

you are building a demo version，

当试图创建调试演示版时，会触发错误信息：

you cannot build a debug demo version。

8.4 异常处理语句

在编写程序时，不仅要关心程序的正常操作，也应该把握在现实世界中可能发生的各类不可预期的事件。比如用户错误的输入、内存不够、磁盘出错、网络资源不可用、数据库无法使用等。在程序中经常采用异常处理方法来解决这类现实问题。

C#中的异常提供了一种处理系统级错误和应用程序级错误的结构化的、统一的、类型安全的方法。C#的异常机制与 C++非常相似，除了以下几点不同之外：

- 在 C#里，所有异常都表现为一个类的实例，这个类继承自 System.Exception.类。而在 C++中，任何类型的任何值都可以表示异常。
- C#中一个终结块里的终结代码既可以在正常情况下执行，也可以在异常情况下执行，而在 C++中，不复制代码是难以做到这一点的。
- 在 C#中，系统级的异常如溢出，零除数等，由于有一个定义完好的异常类因而可以等同于应用程序级错误条件。

8.4.1 溢出的处理

大家知道，计算机进行数学计算时，往往会出现计算结果超出这个结果所属类型的值域的情况，这种情况我们称之为溢出。

C#中，对于溢出的处理有两种选择：

（1）你可以通过调整编译器的选项来检测整个程序。

（2）你可以声明可能发生溢出的模块，进行局部检测。

如果你选择第一种方案，即打算在整个程序中控制溢出检测，可以选中 C#编译器设置面板中的 Checked 选项（checked+）。

如果不希望溢出检测作用于整个程序，我们可以使用局部检测的方案。利用检测语句可以很容易做到。checked 和 unchecked 语句就是用来控制整型算术运算和转换中的溢出检测的，这我们在上一章中已经介绍过。

下面以一个计算阶乘的例子说明它们的用法。

程序清单 8-8：

```
Using   System;
```

```
class Factorial
{
   public static void Main(string[] args)
   {
       long nFactorial=1;
       long nComputerto=Int64.Parse(args[0]);
       long nCurDig=1;
       for(nCurDig=1;nCurDig<=nComputerto;nCurDig++)
            checked{nFactorial*=nCurDig;}
       Console.WriteLine("{0}! Is {1}",nComputerto,nFactorial);
    }
}
```

当阶乘运算发生溢出时,程序将发出异常信息：

```
System.OverflowException at Factorial.main(System.String[])
```

8.4.2 引发异常的条件

异常可以以两种不同的方式引发：

- throw 语句无条件，即时的抛出异常。
- C#语句和表达式执行过程中激发了某个异常的条件，使得操作无法正常结束，从而引发异常。例如整数除法操作分母为零时将抛出一个 System.DivideByZeroException 异常。

8.4.3 异常的处理

throw 语句

在学习如何处理异常之前，我们先介绍一下 throw 语句。

throw 语句抛出一个异常：

throw expression

带有表达式的 throw 语句抛出的异常是在计算这个表达式时产生的。这个表达式必须表示一个 System.Exception 类型或它的派生类型的值。如果对表达式的计算产生的结果是 null，则抛出的将是一个 NullReferenceException 异常。

不带表达式的 throw 语句我们稍后再介绍。

异常处理语句

异常是由 try 语句来处理的。

try 语句提供了一种机制来捕捉块执行过程中发生的异常。以下是它的三种可能的形式：

- try -catch(s)

- try -finally
- try -catch(s)-finally

在介绍 try 语句之前，我们先介绍一个重要的概念：异常传播。当一个异常被抛出以后，程序将控制权转移给 try 语句中第一个能够处理该异常的 catch 子句。这个从异常抛出到控制转移给合适的异常处理语句的过程就叫做异常传播。

异常传播包括重复执行以下步骤，直到找到一个与该异常相匹配的 catch 子句。

（1）由里层到外层的执行每一个包围抛出点（异常被抛出的最初位置）的 try 语句：

- 如果当前的 try 块包含一或多条 catch 子句，那么将按照这些子句的出现顺序对它们逐一检查以定位适合该异常的处理。所谓适合该异常的处理也就是第一个定义了该异常类型或其基类型的 catch 子句。一般 catch 子句（定义见下文）可以匹配任何类型的异常。一旦找到与该异常匹配的 catch 子句，异常传播也就结束了，程序控制转移到该 catch 子句执行。
- 否则，如果当前的 try 块含有 finally 块的话，就执行 finally 块。如果该块又抛出另一个异常，则当前处理的异常将被终止。如果没有，则当 finally 块执行完以后，再继续异常的处理。

（2）如果当前的成员函数调用中没能定位异常处理，则调用终止。并且在该成员函数调用点将该异常抛给调用者，重复执行上一步。

（3）如果该异常终止了当前线程或进程的所有成员函数调用，则说明该线程或进程中不存在对异常的处理，它将自行终止。

下面让我们回到 try 语句。

如果 catch 子句中指定了一个类类型，则它必须是 System.Exception 类型或它的派生类型。如果同时指定了类类型和标识符，就是声明了一个异常变量。异常变量相当于一个作用范围为整个 catch 块的局部变量。在 catch 块的执行过程中，异常变量描述了当前正在处理的异常。如果想引用异常对象（其中包括很多重要的错误信息），就必须定义异常变量。

在一个 catch 块中，可以用不含表达式的 throw 语句将该 catch 块捕捉到的异常再次抛出，对于异常变量的分配不会改变再次抛出的异常。

既没有定义异常类型也没有定义异常变量的 catch 子句称做一般 catch 子句。该 catch 子句一般在事先不能确定会发生什么样的异常的情况下使用。一个 try 语句中只能有一个一般 catch 子句，而且如果有的话,该 catch 子句必须排在其它 catch 子句的后面。

尽管 throw 语句(含表达式)只能抛出 System.Exception 类型或它的派生类型的异常，但其它语句并不受该规则限制，因而可以抛出其它类型的异常。比如用一般 catch 子句就可以捕捉这一类异常，然后利用一个不含表达式的 throw 语句就可将其再次抛出。

由于寻求异常的处理是通过按照 catch 子句出现的顺序逐一检查 catch 子句，因此如果有一个 catch 子句定义的异常类型与比它先出现的 catch 子句定义的类型相同或是它的派生类型，就会发生错误。下面的例子中我们演示了 try-catch 语句的使用以及再次抛出异常。

程序清单 8-9：

```
using System;

class Test
{
  static void F() {
        try {
              G();
        }
        catch (Exception e) {
              Console.WriteLine("Exception in F: " + e.Message);
              e = new Exception("F");
              throw;
        }
  }

        static void G() {
        throw new Exception("G");
  }

        static void Main() {
        try {
              F();
        }
        catch (Exception e) {
              Console.WriteLine("Exception in Main: " + e.Message);
        }
  }
}
```

F 方法捕捉了一个异常，向控制台写了一些诊断信息，改变异常变量的内容，然后将该异常再次抛出。这个被再次抛出的异常与最初捕捉到的异常是同一个。因此程序的输出结果如下：

```
Exception in F: G
Exception in Main: G
```

当 try 语句执行完以后，finally 块中的语句必将被执行。不论是否会发生由以下原因导致的程序控制转移：

- 普通操作的结果；
- 执行 break,continue，goto，或 return 语句的结果；
- 将异常传播到语句之外的结果。

我们用一个例子来证明 finally 子句的运行。

程序清单 8-10：

```
using   System;
class Jumptest
{
   public static void Main()
   {
       try
       {
            Console.WriteLine(“try”);
            goto leave;
        }
        finally
        {
             Console.WriteLine(“finally”);
        }
        leave:
              Console.WriteLine(“leave”);
     }
}
```

该程序的输出结果为：

```
try
finally
leave
```

由此可见，finally 子句总能被执行。因此我们可以利用 try-finally 来清除异常。

如果在执行 finally 块时抛出了一个异常，这个异常将被传播到下一轮 try 语句中去。如果在异常传播的过程中又发生了另一个异常，那么这个异常将被丢失。

最后，由于我们对待异常的态度往往是：捕捉、清除、继续执行程序，因此我们需要在程序中使用 try-catch(s)-finally 结构。

下面的例子计算函数值 $z=\sqrt{|x^2-y^2|}+x$。

```
float x,y,z;
        try{
        z = Math.Sqrt(x*x-y*y);
}
catch{
        z = Math.Sqrt(y*y-x*x);
}
finally {
        z = z+x;
```

```
}
```

其中第一个 try 语句捕捉负数开根号的异常，并与第二个 catch 语句配合达到取绝对值的目的。

8.5 小　　结

本章前半部分解释了如何使用 C#中的各种选择和循环语句。尽管你也许对它们很熟悉，但还是要留意它们与 C++不同的地方。对 if 来说，别忘了一定要用布尔表达式。尤其是，你一定要确保条件语句的短路不会阻止重要代码的运行。switch 语句不再支持遍历，而且你可以使用字符串标签，对于 C 程序员，这是一种新的用法。循环语句部分，我们说明了如何使用 for、foreach、while 和 do 语句。

条件编译中，我们讲述了如何将某些代码包括或排除。

在异常处理部分，我们较详细地介绍了异常传播的过程，并列举了几种解决办法，如用 checked 和 unchecked 控制溢出、用 try-catch-finally 处理所有异常等。

复习题

（1）if 语句的嵌套使用应注意哪些问题？

（2）为什么在 C#取消了 switch 语句中的遍历功能？我们如何在 C#程序中实现遍历？

（3）编写一个程序，对输入的四个整数，求出其中的最大值和最小值。

（4）试分别使用 for、while 和 do-while 语句编写程序，实现求前 n 个自然数之和。

（5）在数学中，有一种数列，各项为：

$$a_1 = 1;\quad a_2 = 1;\quad a_n = a_{n-2} + a_{n-1}\ (n>2);$$

试通过程序验证，当 n 趋向于无穷大时，数列前后项的比值 a_{n-1}/a_n 无限接近 0.618;

（6）如何在程序执行过程中发出编译的错误或警告？

（7）异常处理在程序中起到什么样的作用？

（8）说说异常处理语句 try、catch、finally 执行时的相互关系。

（9）在程序中使用异常处理除数为 0 的错误。

第三部分　面向对象的 C#

第九章　面向对象的程序设计

我们知道，C#源于 C 和 C++。1970 年，Brian 和 Dennis Ritchie 首创了 C 语言，由于其简单灵活的特点，很快成为世界上最流行的语言之一。

然而 C 语言是一个面向过程的程序设计语言。随着软件开发技术的进步，产生了面向对象的程序设计思想，二十世纪八十年代初美国 AT&T 贝尔实验室设计并实现了 C++语言，增加了对面向对象的程序设计的支持。

C#秉承了 C++面向对象的特性，支持面向对象的所有关键概念：封装、继承和多态性。整个 C#的类模型是建立在.Net 虚拟对象系统之上的，对象模型是基础架构的一部分而不再是编程语言的一部分。

本章不准备全面地介绍面向对象的理论，而是简要地论述面向对象的程序设计的基本技术，使读者对这方面的概念有一个初步的认识，为继续阅读本书后面的章节打下基础。

如果读者希望系统地了解面向对象的方法学，请参考这方面的专著。

通过对本章及对第三部分其它章节的学习后，读者将对 C#面向对象的程序设计有一个深刻的认识和理解。

在本章中，我们将简要地向读者介绍：

- 面向对象技术的由来。
- 面向对象系统的基本概念。
- 对象模型技术。
- 面向对象的分析的内容。
- 面向对象的设计的内容。
- Coad 和 Yourdon 面向对象的方法。

9.1　面向对象的基本概念

9.1.1 面向对象的技术的由来

随着计算机硬件技术的飞速发展，计算机的容量、速度迅速提高，计算机取得了越来越广泛的应用，这就对软件开发提出了更高的要求。然而软件技术的进步却远远滞后于硬件技术的进步，人们常常无法控制软件开发的周期和成本，软件的质量总是不尽人意，经常是用之不灵、弃之可惜，有的软件甚至无法交付。这种状况人们称之为“软件危机”。

人们认识到，为了摆脱软件危机，必须按照工程化的原则和方法来组织软件开发工作。二十世纪七十年代流行的面向过程的软件设计方法，目的主要是解决面向过程语言系统的设计问题。它主要强调程序的模块化和自顶向下的功能分解。在涉及大量计算的算法类问题上，从算法的角度揭示事物的特点，面向过程的分割是合适的。但是现在的软件应用涉及社会生活的方方面面，面对变动的现实世界，面向过程的设计方法暴露出越来越多的不足。例如：

- 功能与数据分离，不符合人们对现实世界的认识。要保持功能与数据的相容也十分困难。
- 基于模块的设计方式，导致软件修改困难。
- 自顶向下的设计方法，限制了软件的可重用性，降低了开发效率，也导致最后开发出来的系统难以维护。

为了解决结构化程序设计的这些问题，面向对象的技术应运而生。它是一种非常强有力的软件开发方法。它将数据和对数据的操作作为一个相互依赖、不可分割的整体，采用数据抽象和信息隐蔽技术，力图使对现实世界问题的求解简单化。它符合人们的思维习惯，同时有助于控制软件的复杂性，提高软件的生产效率，从而得到了广泛的应用，已成为目前最为流行的一种软件开发方法。

请读者注意，在这里我们并不是说面向对象的程序设计方法一定比面向过程的方法或者是其它的程序设计方法要好。每种方法都有自己的一套理论框架，也有相应的设计、分析、建模的方法，所以它们都有各自的优缺点。选择的依据关键是要看目标系统所要解决的是哪种类型的问题。如对于工程计算，面向过程的方法就可能比面向对象的方法更加适合。如果解决逻辑推理、机器证明等人工智能方面的问题，我们可能就需要采用面向逻辑的技术和方法。如果是设计知识库和专家系统，我们还可以选择面向规则的程序设计技术。

9.1.2 基本概念

可以这样认为：“面向对象=对象+类+继承+通信”。如果一个软件系统是使用这样四个概念来设计和实现的，我们认为这个软件系统是面向对象的。

对象（object）

对象是面向对象开发方法的基本成分。每个对象可用它本身的一组属性和其上的一组操作来定义。对象可以是现实生活中的一个物理对象，还可以是某一类概念实体的实例。比如，一辆汽车、一个人、一本书，乃至一种语言、一个图形、一种管理方

式，都可以作为一个对象。

从分析和设计的角度来看，对象表示了一种概念，它们把有关的现实世界的实体模型化。实体的有关声明有：描述实体，包括实体的属性和可以执行的操作。比如对于汽车这个对象，它的重量、颜色都可以作为对象的属性，它可以执行的操作可以是行驶、鸣笛等。

类（class）

类是一组具有相同数据结构和相同操作的对象的集合。类是对一系列具有相同性质的对象的抽象，是对对象共同特征的描述。比如每一辆汽车是一个对象的话，所有的汽车可以作为一个模板，我们就定义汽车这个类。

在一个类中，每个对象都是类的实例，可以使用类中提供的方法。从类定义中产生对象，必须有建立实例的操作，C++和 C#中的 new 操作符可用于建立一个类的实例，C#为我们提供的方法则更加安全。

继承（inheritance）

继承是使用已存在的定义作为基础建立新定义的技术。新类的定义可以是即存类所声明的数据和新类所增加的声明组合。新类复用即存的定义，而不要求修改即存类。即存类可以作为基类来引用，而新类可以作为派生类来引用。这种复用技术大大降低了软件的开发费用。

例如，汽车作为一个类已经存在，作为具有自身特征的卡车就可以从汽车类中继承。它同汽车一样，具有颜色、重量这些特征，可以行驶和鸣笛。它还具有一般汽车不一定具备的特征，比如可以载货等。

9.2 对象的模型技术

对象模型技术（object modeling technique，OMT）是美国通用电气公司提出的一套系统开发技术。它以面向对象的思想为基础，通过对问题进行抽象，构造出一组相关的模型，从而能够全面地捕捉问题空间的信息。

对象模型技术把分析时收集到的信息构造在三类模型中，即对象模型、功能模型和动态模型。三个模型从不同的角度对系统进行描述，分别着重于系统的一个方面，组合起来构成对系统的完整描述。形象地说，功能模型定义“做什么”，状态模型定义“何时做”，对象模型定义“对谁做”。

9.2.1 对象模型

对象模型描述系统的静态结构，包括类和对象，它们的属性和操作，以及它们之间的关系。构造对象模型的目的在于找出与应用密切相关的概念。

对象模型用包含对象及对象的关系图表示。图 9-1 给出了在对象模型中用于表示

类和对象的图形符号。

使用 OMT 建立对象模型的主要步骤如下：

（1）确定对象类；

（2）定义数据词典，用以描述类、属性和关系；

（3）用继承组织和简化对象类；

（4）测试访问路径；

（5）根据对象之间的关系和对象的功能将对象分组，建立模块。

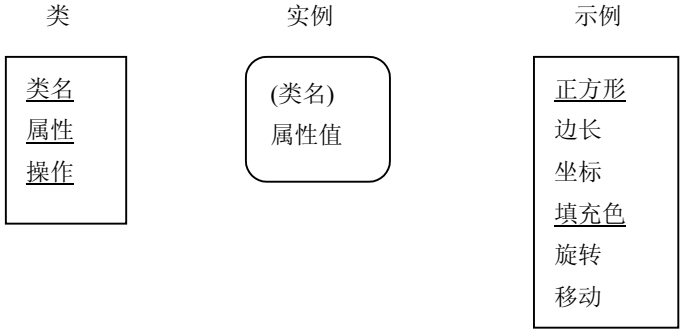


图 9-1　类与对象的表示方法

9.2.2　动态模型

动态模型着重于系统的控制逻辑，考察在任何时候对象及其关系的改变，描述这些涉及时序和改变的状态。

动态模型包括状态图和事件跟踪图。状态图是一个状态和事件的网络，侧重于描述每一类对象的动态行为。事件跟踪图则侧重于说明系统执行过程中的一个特点“场景”，也叫做脚本（scenarios），是完成系统某个功能的一个事件序列。脚本通常起始于一个系统外部的输入事件，结束于一个系统外部的输出事件。

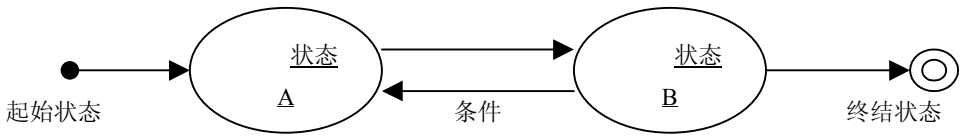


图 9-2　状态图的表示方法

对象到对象的单个消息叫做一个事件，在系统的一个特定的环境下发生的一系列事件叫做一个场景。在一个场景中，这一系列事件和交换事件的对象都可以放在一个事件跟踪图中表示。

建立动态模型的主要步骤有：

（1）准备典型的交互序列的场景；

（2）确定对象之间的事件，为每个场景建立事件跟踪图；

（3）为每个系统准备一个事件流图；

（4）为具有重要动态行为的类建立状态图；

（5）检验不同状态图中共享的事件的一致性和完整性。

9.2.3 功能模型

功能模型着重于系统内部数据的传送和处理。功能模型表明，通过计算，从输入数据能得到什么样的输出数据，但不考虑参加计算的数据按什么时序执行。功能模型由多个数据流图组成，它们指明从外部输入，通过操作和内部存储，直到外部输出的整个数据流情况。功能模型还包括了对象模型内部数据间的限制。

功能模型中的数据流图往往形成一个层次结构，一个数据流图的过程可以由下一层的数据流图作进一步的说明。

建立功能模型的主要步骤有：

（1）确定输入和输出值；

（2）用数据流图表示功能的依赖性；

（3）具体描述每个功能；

（4）确定限制；

（5）对功能确定优化的准则。

9.3 面向对象的分析

面向对象的分析 OOA（object-oriented analysis）是软件开发过程中的问题定义阶段，这一阶段最后得到对问题论域的清晰、精确的定义。传统的系统分析产生一组面向过程的文档，定义目标系统的功能。面向对象分析则产生一种描述系统功能和问题空间的基本特征的综合文档。

9.3.1 论域分析和应用分析

面向对象的分析过程可分为两个阶段，即论域分析阶段和应用分析阶段。

论域分析

论域分析是软件开发的基本组成部分，目的是使开发人员了解问题空间的组成，建立大致的系统实现环境。论域分析给出一组抽象，从高层表示论域知识，常常超出当前应用的范围，作为特定系统需求开发的参考。

论域分析实际上是一种学习，软件开发人员在这个阶段尽可能地理解当前系统与应用有关的知识。应放开思维，放宽考虑的领域，尽可能标识与应用有关的概念。论域分析的边界可能很模糊，有了广泛的论域知识，涉及到具体的应用时，就可以更快的进入情况，掌握应用的核心知识。而且，在用户改变对目标系统的需求时，广泛的分析可以帮助我们预测出目标系统在哪些方面会发生哪些变化。

通常进行小组分析，小组成员可以包括领域专家和分析员等。在分析过程中标识

出系统的基本概念：对象、类、方法、关系等。识别论域的特征，把这些概念集成到论域的模型中。论域的模型中必须包含概念之间的关系，还有关于每个单独概念的全部信息。这里信息起一种胶合作用，把标识出的相关概念并入论域综合视图中去。

应用分析

应用分析是依据在论域分析时建立起来的问题论域模型，并把问题论域模型用于当前特定的应用之中。

首先，通过收集到的用户信息来对论域进行取舍，把用户需求作为限制来使用，缩减论域的信息量。因此，论域分析的视野大小直接影响到应用分析保留的信息量。

一般说来，论域分析阶段产生的模型并不需要用任何基于计算机系统的程序设计语言来表示，而应用分析阶段产生的影响条件则通过某种基于计算机系统的程序设计语言来表示。

模型识别的要求可以针对一个应用，也可以针对多个应用。通常我们着重考虑两个方面，即应用视图和类视图。在类视图中，必须对每个类的规格说明和操作进行详细化，并表示出类之间的相互作用。

9.3.2 Coad 和 Yourdon 面向对象的分析过程

Coad 和 Yourdon 的方法是建立在信息模型化技术、面向对象的程序设计和知识库系统的基础之上的，方法分为面向对象的分析（OOA）和面向对象的设计（OOD）。我们在这里先讨论 Coad 和 Yourdon 面向对象的分析，在下一节中再讨论面向对象的设计。

Coad 和 Yourdon 认为面向对象的分析主要应该考虑：一个与特定应用有关的对象，对象与对象在结构和相互作用上的关系。在面向对象的分析中，需要建立分析模型来描述系统的功能。

OOA 的任务

面向对象的分析需要完成两个任务：

- 形式的说明所面对的应用问题，最终成为软件系统基本构成的对象，以及系统所必须遵从的、由应用环境所决定的规则和约束。
- 明确的规定构成系统的对象如何协同工作，完成指定的功能。

通过面向对象的分析建立的系统模型是以概念为中心的，因此称为概念模型。概念模型由一组相关的类组成。面向对象的分析可以自顶向下地逐层分解建立系统模型，也可以自底向上地从已经定义的类出发，逐步构造新的类。

概念模型构造和评审的顺序由五个层次构成：类和对象层、属性层、服务层、结构层、主题层。这五个层次不是构成软件系统的层次，而是分析过程中的层次，也可以说是问题的不同侧面，每个层次的工作都为系统的规格说明增加了一个组成部分。当五个层次的工作全部完成时，面向对象的分析的任务也就完成了。

OOA 的步骤

面向对象的分析通常按照下面的步骤进行：

（1）标识对象和类。可以从应用领域开始，逐步确定形成整个应用的基础的类和对象。这个步骤要通过分析领域中目标系统的责任、调查系统的环境，从而确定对系统有用的类和对象。

（2）标识结构。典型的结构有两种，一般—特殊结构和整体—部分结构。一般—特殊结构表示一般类是基类，特殊类是派生类；整体—部分结构表示聚合，由属于不同类的成员聚合成为新的类。

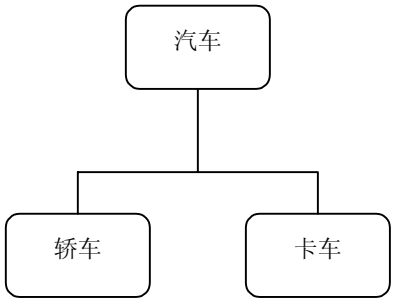


图 9-3　一般—特殊结构的例子

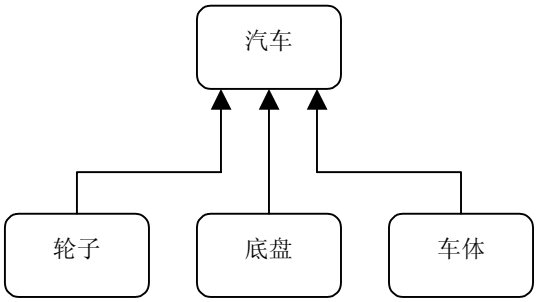


图 9-4　整体—部分结构的例子

（3）标识属性。对象所保存的信息称为它的属性。类的属性所描述的是状态信息，在类的某个实例中属性的值表示该对象的状态值。对于每个对象，我们都需要找出在目标系统中对象所需要的属性，而后将属性安排到适当的位置，找出实例连接，最后进行检查。对每个属性应该给出描述，由属性的名字和属性的描述来确定，并指定对该属性存在哪些特殊的限制（如只读、属性值限定于某个范围之内等）。

（4）标识服务。对象收到消息后执行的操作称为对象提供的服务，它描述了系统需要执行的处理和功能。定义服务的目的在于定义对象的行为和对象之间的通信。其具体步骤包括标识对象状态、标识必要的服务、标识消息连接和对服务的描述。可以用和流图类似的图形来指明服务。

（5）标识主题。对于包含大量类和对象的概念模型往往难以掌握，标识主题则对模型进行划分，给出模型的整体框架，划分出层次结构。但是 Coad 和 Yourdon 并没有提出如何标识主题的具体建议。在标识主题时，可以采取先识别主题，而后对主题进行改进和细化，最后将主题加入到分析模型当中的步骤进行。主题是一个与应用相关的，而不是人为任意引出的概念，主题层的工作有助于分析的结果。

9.4　面向对象的设计

从面向对象的分析到面向对象的设计是一个逐步扩充模型的过程。面向对象的分析时以实际问题为中心，可以不包括任何与特定计算机有关的问题，主要考虑“做什

么”的问题；面向对象的设计则是面向计算机的实地开发活动，考虑“怎么做”的问题。

9.4.1 高层设计和底层设计

面向对象的设计分为两个阶段，即高层设计和低层设计。

高层设计

高层设计阶段开发系统的结构，构造待开发软件的总体模型。在这个阶段，标识出在具体的计算机环境中进行问题求解所需要的概念，增加了一批需要的类，这些类包括那些使得软件系统能够与外部世界进行交互的类。

高层设计阶段的输出是适合应用软件要求的类、类之间的关系及应用子系统的视图规格说明。通常，利用面向对象的设计得到的系统框架如下图所示。

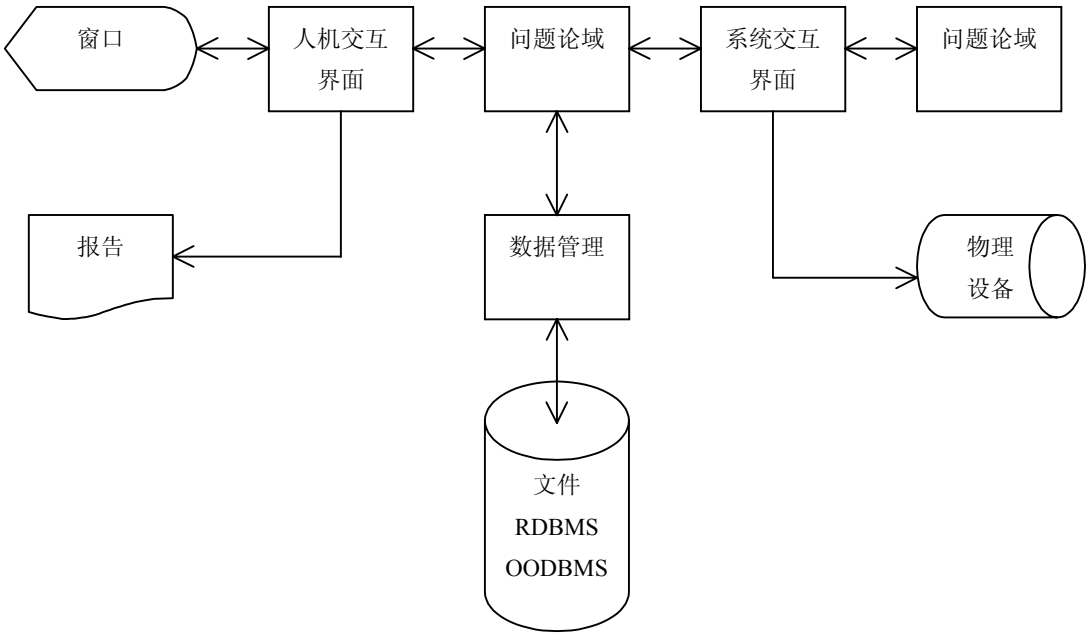


图 9-5　面向对象的设计导出的系统结构

高层设计过程中，应当使子系统的高层部件之间的通信量达到最小，把子系统中相互之间存在高度交互的类进行逻辑分组。

低层设计

低层设计集中于类的详细设计阶段。类设计的目标是形成单一概念的模型——一个独立的类表示一个概念，以及设计的部件应该是可复用的和可靠的。

类的设计过程中需要采用信息隐蔽、高内聚低耦合等设计原则。在面向对象的技术中，利用即存类的复用是一个很大的优点。

9.4.2 Coad 和 Yourdon 面向对象的设计过程

Coad 与 Yourdon 在设计阶段继续采用面向对象分析阶段中提到的五个层次（对象和类、结构、属性、服务和主题），这有助于从分析到设计的过渡。按照 Coad 与 Yourdon 面向对象设计方法，在设计阶段中利用这五个层次，建立系统的四个组成成分：问题论域、用户界面、任务管理和数据管理。

问题论域部分的设计

问题论域部分包括与我们所面对的应用问题直接相关的所有类和对象，这一工作实际上在面向对象的分析阶段已经开始，这时需要对它进行进一步的细化。

在面向对象的分析阶段，得到了与应用有关的概念模型，在面向对象的设计阶段，我们对分析得到的结果进行改进和增补。主要根据需求的变化，对面向对象的分析阶段产生的模型中的类和对象、结构、属性、操作进行组合和分解，根据面向对象的设计原则，增加必要的类、属性和关系。

问题论域部分的设计包括：

（1）复用设计；

（2）把问题论域相关的类关联起来；

（3）加入一般化的类以建立类间协议；

（4）调整继承支持级别；

（5）改进性能；

（6）加入较低层的构件。

用户界面部分的设计

通常在面向对象的分析阶段给出了所需的属性和操作，在面向对象的设计阶段必须根据需求把交互的细节加入到用户界面的设计中，包括有效的人机交互所必须的实际显示和输入。

用户界面部分的设计主要由以下几个方面组成：

（1）用户分类；

（2）描述人及其任务的脚本；

（3）设计命令层；

（4）设计详细的交互；

（5）继续扩展用户界面原型；

（6）设计人机交互类（HIC）；

（7）根据图形用户界面进行设计。

任务管理部分的设计

所谓任务，是进程的别称，是执行一系列活动的一段程序。当系统中有许多并发行为时，需要依照各个行为的协调和通信关系，划分各种任务。以达到简化并发行为的设计和编码的目的。

任务管理主要包括任务的选择和调整，它的工作包括：

（1）识别事件驱动任务；
（2）识别时钟驱动任务；
（3）识别优先任务和关键任务；
（4）识别任务之间的协调者；
（5）对各个任务进行评审，保证它能够满足选择任务的过程标准；
（6）定义各个任务，说明它是什么任务、任务之间如何协调工作、如何通信。

数据管理部分的设计

数据管理部分提供了在数据管理系统中存储和检索对象的基本结构，包括对永久性数据的访问和管理。

数据管理的方法主要有三种：文件管理、关系数据库管理以及面向对象的数据库管理。

数据管理部分的设计包括：

（1）数据存放设计。数据存放设计选择数据存放的方式：文件存放、关系数据库表格存放，或者是面向对象的数据库存放。

（2）设计相应的操作。为每个需要存储的对象和类增加用于存储管理的属性和操作，在类和对象的定义中加以描述。

9.5 小　　结

面向对象是软件程序设计的一种新思想，这种思想的引入，是我们的程序设计更加贴近现实。传统的结构化设计是面向过程的，存在着许多自身无法克服的缺点，面向对象的设计解决了结构化设计的这些问题。面向对象的技术追求的是软件系统对现实世界的直接模拟，它把数据和对数据的操作封装成为一个整体。

对象的基本要素有对象、类、继承和消息。

对象模型技术把信息构造在三类模型中，它们是对象模型、状态模型和功能模型。每个模型从自己的角度对系统进行描述。

面向对象的分析分为论域分析和应用分析，面向对象的设计则可以分为高层设计和低层设计。

Coad 与 Yourdon 方法按照对模型进行构造和评审的顺序把概念模型分为五个层次：类和对象层、属性层、服务层、结构层和主题层。在设计阶段，主要对系统的四个组成部分进行设计，它们是问题论域部分、用户界面部分、任务管理部分和数据管理部分。

复习题

（1）采用面向对象的方法建立模型的思维过程是怎样的？
（2）简要地说明如何确定和划分类，类和对象是怎样的关系。

（3）对象之间有哪些关系，如何在对象之间进行通讯？

（4）类的设计准则有哪些？

（5）简述 Coad 与 Yourdon 方法的五个层次和四个组成部分。

（6）以交通工具为例，画出类的层次结构图。

（7）如何才能提高类的可复用性？在类的复用设计中，主要的继承关系有哪些？试通过举例来说明。

（8）在学习完本书第三部分后，回过头来复习面向对象的分析与设计，以一个学生学籍管理系统为例，使用 C#语言和你所熟悉的某种数据库语言，建立软件的体系结构。要求：

问题论部分，采用类的层次结构描述，区分类之间的继承。

数据管理部分，采用数据库语言建立结构。

人机交互部分，采用类的层次结构描述，注意用户界面的友好性。

任务管理部分，采用 C#的控制结构。

第十章　类

类是面向对象的程序设计的基本构成模块。从定义上讲，类是一种数据结构，这种数据结构可能包含数据成员、函数成员以及其它的嵌套类型。其中数据成员类型有常量、域和事件；函数成员类型有方法、属性、索引指示器、操作符、构造函数和析构函数。

本章介绍的内容包括：

- 类的声明
- 类的成员
- 类的构造函数和析构函数

10.1　类 的 声 明

类的声明格式如下：

attributes　class-modifiers　*class*　identifier　class-base　class-body　;

其中 attributes、class-modifiers、class-base 和 class-body 为可选项。Attributes 为属性集，class-modifiers 为类的修饰符，关键词 *class* 后跟随类的名称 identifier，class-base 和 class-body 表示继承方式和基类名。

类的修饰符

类的修饰符可以是以下几种之一或者是它们的组合（在类的声明中同一修饰符不允许出现多次）：

- *new*——仅允许在嵌套类声明时使用，表明类中隐藏了由基类中继承而来的、与基类中同名的成员。
- *public*——表示不限制对该类的访问。
- *protected*——表示只能从所在类和所在类派生的子类进行访问。
- *internal*——只有其所在类才能访问。
- *private*——只有对包.Net 中的应用程序或库才能访问。
- *abstract*——抽象类，不允许建立类的实例。
- *sealed*——密封类，不允许被继承。
- 在 6.5 节中我们将详细介绍抽象类和密封类的概念。

使用类的实例

使用 new 关键字可以建立类的一个实例，比如下面的代码：

```
class A {}
```

```
class B {
    void F{
        A a = new A();
    }
}
```

在类 B 的方法 F 中创建了一个类 A 的实例。

类的继承声明

我们使用如下代码表示类 B 从类 A 中继承：

```
class A {}
class B: A {}
```

有关 C#中的继承机制我们放在第四篇中进行详细讨论，在这里要事先声明的一点是：C#中的类只支持单继承。

10.2 类 的 成 员

类的成员可以分为两大类：类本身所声明的，以及从基类中继承而来的。

类的成员有以下类型：

- 成员常量，代表与类相关联的常量值。
- 域，即类中的变量。
- 成员方法，复杂执行类中的计算和其它操作。
- 属性，用于定义类中的值，并对它们进行读写。
- 事件，用于说明发生了什么事情。
- 索引指示器，允许像使用数组那样为类添加路径列表。
- 操作符，定义类中特有的操作。
- 构造函数和析构函数，分别用于对类的实例进行初始化和销毁。

包含有可执行代码的成员被认为是类中的函数成员，这些函数成员有方法、属性、索引指示器、操作符、构造函数和析构函数。

10.2.1 对类的成员的访问

在编写程序时，我们可以对类的成员使用不同的访问修饰符，从而定义它们的访问级别。

公有成员

C#中的公有成员提供了类的外部界面，允许类的使用者从外部进行访问。公有成员的修饰符为 public，这是限制最少的一种访问方式。

私有成员

C#中的私有成员仅限于类中的成员可以访问，从类的外部访问私有成员是不合法的。如果在声明中没有出现成员的访问修饰符，按照默认方式成员为私有的。私有成员的修饰符为 private。

保护成员

为了方便派生类的访问，又希望成员对于外界是隐藏的，这时可以使用 protected 修饰符，声明成员为保护成员。

内部成员

使用 internal 修饰符的类的成员是一种特殊的成员。这种成员对于同一包中的应用程序或库是透明的，而在包.Net 之外是禁止访问的。

使用下面的例子说明一下类的成员的访问修饰符的用法。

程序清单 10-1：

```
using System;
class Vehicle//定义汽车类
{
        public      int wheels; //公有成员：轮子个数
        protected   float weight;           //保护成员：重量
        public void F(){
              wheels = 4;//正确，允许访问自身成员
              weight = 10;      //正确，允许访问自身成员
        }
};
class train    //定义火车类
{
        public int num;   //公有成员：车厢数目
  private int passengers; //私有成员：乘客数
  private       float weight; //私有成员：重量
  public void F(){
        num = 5;   //正确，允许访问自身成员
        weight = 100;    //正确，允许访问自身成员
        Vehicle v1 = new Vehicle();
        v1.wheels = 4;   //正确，允许访问 v1 的公有成员
        //v1.weight = 6;         错误，不允许访问 v1 的保护成员，可改为：
        weight = 6;
  }
}
class Car:Vehicle  //定义轿车类
```

```
{
        int passengers;          //私有成员：乘客数
  public void F(){
        Vehicle v1 = new Vehicle();
        V1.wheels = 4;   //正确，允许访问 v1 的公有成员
        V1.weight = 6;          //正确，允许访问 v1 的保护成员
  }
}
```

10.2.2 this 保留字

保留字 this 仅限于在构造函数、类的方法和类的实例中使用，它有以下含义：

- 在类的构造函数中出现的 this 作为一个值类型，它表示对正在构造的对象本身的引用。
- 在类的方法中出现的 this 作为一个值类型，它表示对调用该方法的对象的引用。
- 在结构的构造函数中出现的 this 作为一个变量类型，它表示对正在构造的结构的引用。
- 在结构的方法中出现的 this 作为一个变量类型，它表示对调用该方法的结构的引用。

除此以外，在其它地方使用 this 保留字都是不合法的。

下面的代码演示了如何使用 this 保留字。

程序清单 10-2：

```
using System;
class A
{
  public int x;
  public void Main()
  {
        x = 5;
        Console.WriteLine("The value of x is: {0}",x);
        Console.WriteLine("The value of this.x is: {0}", this.x);
  }
}
```

程序运行的结果应该是：

```
The value of x is: 5
The value of this.x is: 5
```

注意：如果你是一名 C++的程序员，在使用 C#时请忘记 C++中的 this 指针，也忘掉诸如 this->和:: 之类的符号。只要有名字嵌套的概念，你使用 "." 就足够了。

下面再举一个例子来说明 this 的用法。

程序清单 10-3：

```
using System;
class Fact
{
  int x;
  public int GetFact()
  {
        float temp;
        int save = x;
        int a = 1;
        while(x>a)
        {
              a++;
              temp = this.x / a;
              x /= a;
              if((float)x != temp){
                    return –1;
              }
        }
        swap(this.x,save);
        return save;
  }
}
```

程序用于求某个整数是否为另一个整数的阶乘。如果是，类 Fact 的方法 GetFact（）返回该整数；否则，GetFact（）返回-1。

实际上，在 C#内部，this 被定义为一个常量。因此，使用 this++，this—-这样的语句都是不合法的。但是，this 可以作为返回值来使用。

我们知道，在 Windows 操作系统当中，当前窗口总是被加亮显示，我们称该窗口被激活。例如，在 Microsoft Word 中，我们可以同时打开多个文档。每个文档窗口作为 Microsoft Word 主窗口的一个子窗口，其中只有一个子窗口是当前激活的窗口。如果 Microsoft Word 没有打开任何文档，则主窗口作为当前激活的窗口。

下面的例子中，我们声明了一个窗口类 Window，假设类 Window 一次最多可以同时打开五个子窗口。Window 的方法 `GetActiveWindow`（）用于返回当前激活的窗口。如果打开了子窗口，则返回该子窗口的实例，否则返回主窗口本身。

程序清单 10-4：

```
using System;
class Window
{
```

```
        public Window[] m_ChildWindow = new Window[5];     //子窗口
    public bool IsHaveChild = false;   //是否拥有子窗口
    public bool IsActive;   //窗口是否被激活
    public Window GetActiveWindow()
    {
        if(IsHaveChild == false){
            IsActive = true;
            return this; //返回窗口自身
        }
        else{
            for(int i=0;i<5;i++){
                if(m_ChildWindow[i].IsActive == true){
                    return m_ChildWindow[i];
                    //返回激活的子窗口
                }
            }
        }
        return this;
      }
}
```

10.2.3 静态成员和非静态成员

若将类中的某个成员声明为 static，该成员称为静态成员。类中的成员要么是静态，要么是非静态的。一般说来，静态成员是属于类所有的，非静态成员则属于类的实例——对象。

以下示例代码演示了如何声明静态和非静态成员。

程序清单 10-5：

```
using System;
class Test
{
  int x;
  static int y;
        void F() {
        x = 1;                    // 正确,等价于 this.x = 1
        y = 1;                    // 正确,等价于 Test.y = 1
    }
```

```
        static void G() {
        x = 1;                  // 错误，不能访问 this.x
        y = 1;                  // 正确,等价于 Test.y = 1
    }
        static void Main() {
        Test t = new Test();
        t.x = 1;                // 正确
        t.y = 1;                // 错误，不能在类的实例中访问静态成员
        Test.x = 1;         // 错误，不能按类访问非静态成员
        Test.y = 1;         // 正确
    }
}
```

类的非静态成员属于类的实例所有，每创建一个类的实例，都在内存中为非静态成员开辟了一块区域。而类的静态成员属于类所有，为这个类的所有实例所共享。无论这个类创建了多少个副本，一个静态成员在内存中只占有一块区域。

10.2.4 成员常量

让我们再看一个成员常量的声明例子：

```
class A
{
        public const double X = 1.0;
        public const double Y = 2.0;
        public const double Z = 3.0;
}
```

关键字 const 用于声明常量，后跟数据类型的声明。类的常量可以加上以下修饰符：

- new
- public
- protected
- internal
- private

可以用一条语句同时声明多个常量，比如上例我们可以写成：

```
class A
{
        public const double X = 1.0, Y = 2.0, Z = 3.0;
}
```

10.3 构造函数和析构函数

10.3.1 构造函数

构造函数用于执行类的实例的初始化。每个类都有构造函数，即使我们没有声明它，编译器也会自动地为我们提供一个默认的构造函数。在访问一个类的时候，系统将最先执行构造函数中的语句。实际上，任何构造函数的执行都隐式地调用了系统提供默认的构造函数 base()。

如果我们在类中声明了如下的构造函数，

```
C(…) {…}
```

它等价于：

```
C(…): base() {…}
```

使用构造函数请注意以下几个问题：

- 一个类的构造函数通常与类名相同。
- 构造函数不声明返回类型。
- 一般地，构造函数总是 public 类型的。如果是 private 类型的，表明类不能被实例化，这通常用于只含有静态成员的类。
- 在构造函数中不要做对类的实例进行初始化以外的事情，也不要尝试显式地调用构造函数。

下面的例子示范了构造函数的使用：

```
class A
{
  int x = 0, y = 0, count;
        public A() {
        count = 0;
  }
        public A(int vx,int vy) {
        x = vx;
      y = vy;
  }
}
```

10.3.2 构造函数的参数

上一小节的例子中，类 A 同时提供了不带参数和带参数的构造函数。

构造函数可以是不带参数的，这样对类的实例的初始化是固定的。有时，我们在对类进行实例化时，需要传递一定的数据，来对其中的各种数据初始化，使得初始化

不再是一成不变的，这时，我们可以使用带参数的构造函数，来实现对类的不同实例的不同初始化。

在带有参数的构造函数中，类在实例化时必须传递参数，否则该构造函数不被执行。

让我们回顾一下 10.2 节中关于车辆的类的代码示例。我们在这里添加上构造函数，验证一下构造函数中参数的传递。

程序清单 10-6：

```
using System;
class Vehicle//定义汽车类
{
        public      int wheels; //公有成员：轮子个数
        protected   float weight;            //保护成员：重量
        public      Vehicle(){;}
        public      Vehicle(int w,float g){
                    wheels = w;
                    weight = g;
                }
  public void Show(){
                Console.WriteLine("the wheel of vehicle is:{0}",wheels);
                Console.WriteLine("the weight of vehicle is:{0}",weight);
        }
};
class train    //定义火车类
{
        public int num;   //公有成员：车厢数目
  private int passengers; //私有成员：乘客数
  private       float weight; //私有成员：重量
  public Train(){;}
  public Train(int n,int p,float w){
        num = n;
        passengers = p;
        weight = w;
  }
  public void Show(){
                Console.WriteLine("the num of train is:{0}",num);
                Console.WriteLine("the weight of train is:{0}",weight);
                 Console.WriteLine("the Passengers train car is:{0}", Passengers);
        }
}
```

```
class Car:Vehicle  //定义轿车类
{
        int passengers;          //私有成员：乘客数
        public Car(int w,float g,int p) : base(w,g)
  {
                wheels = w;
                weight = g;
                passengers = p;
            }
        new public void Show(){
            Console.WriteLine("the wheel of car is:{0}",wheels);
            Console.WriteLine("the weight of car is:{0}",weight);
             Console.WriteLine("the Passengers of car is:{0}", Passengers);
        }
}
class Test
{
  public static void Main(){
        Vehicle v1 = new Vehicle(4,5);
        Train t1 = new Train();
            Train t2 = new Train(10,100,100);
            Car c1 = new Car(4,2,4);
            v1.show();
            t1.show();
            t2.show();
            c1.show();
        }
}
```

程序的运行结果为：

```
the wheel of vehicle is:0
the weight of vehicle is: 0
the num of train is:0
the weight of train is:0
the Passengers of train is:0
the num of train is:10
the weight of train is:100
the Passengers of train is:100
the wheel of car is:4
the weight of car is:2
```

the Passengers of car is：4

10.3.3 析构函数

在类的实例超出范围时，我们希望确保它所占的存储能被收回。C#中提供了析构函数，用于专门释放被占用的系统资源。

析构函数的名字与类名相同，只是在前面加了一个符号“~”。析构函数不接受任何参数，也不返回任何值。如果你试图声明其它任何一个以符号“~”开头而不与类名相同的方法，和试图让析构函数返回一个值一样，编译器都会产生一个错误。

析构函数不能是继承而来的，也不能显式地调用。当某个类的实例被认为不再有效，符合析构的条件，析构函数就可能在某个时刻被执行。C++的程序员常常需要在析构函数中写上一系列 delete 语句来释放存储，而在 C#中，我们不必再为此担心了。垃圾收集器会帮助我们完成这些易被遗忘的工作。

10.4 小　　结

这一章我们介绍了 C#中的类的基本概念，展示了类的各种组成。

C#中的类是对数据结构的封装与抽象，是 C#最重要的组成部分。我们利用类定义各种新的数据类型，其中即包含数据内容，又包含对数据内容的操作。封装之后，类可以控制外界对它的成员的访问。

类的静态成员属于该类，非静态成员则属于这个类的某个实例。

在一个类的实例——对象的生命周期中，最先执行的代码就是类的构造函数。构造函数是用来初始化对象的特殊类型的函数。

不带参数的构造函数对类的实例的初始化是固定的。我们也可以使用带参数的构造函数，通过向它传递参数来对类的不同实例进行不同的初始化。构造函数同样可以使用默认参数。

当这个类的实例超出作用域时，或者由于其它理由被破坏时，析构函数将释放分配给该实例的任何存储区。

复习题

（1）在定义和使用 C#中的类时，应注意什么问题？

（2）类和类的实例之间是什么关系？用具体的例子说明怎样建立一个类的实例。

（3）类的成员有哪几种？

（4）试说明在哪种情况下需要指定类的成员为静态的。

（5）类的构造函数与类的成员方法有哪些不同之处？

（6）是否可以定义返回值为 void 型的类的构造函数。

（7）定义一个网络用户类，要处理的信息有用户 ID、用户密码、 email 地址。

在建立类的实例时，把以上三个信息都作为构造函数的参数输入，其中用户 ID 和用户密码是必须的，缺省的 email 地址是用户 ID 加上字符串“@hope.com”。

第十一章　方　　法

在面向过程的语言如 C 语言中，数据和对数据的操作通常分为两部分。在 C++语言中，大多数数据成为类的数据成员，而大多数对数据的操作放在了类的成员方法中。

C#实现了完全意义上的面向对象：任何事物都必须封装在类中，或者作为类的实例成员——没有全局常数、全局变量，也没有全局方法。

我们在上一章中学习了类的基本概念，并掌握了如何对类进行实例化。这一章我们将利用类的方法为类添加功能。

11.1　方法的声明

方法是类中用于执行计算或其它行为的成员。我们看一下方法的声明格式：

method-header　　*method-body*

其中方法头 *method-header* 的格式：

attributes　*method-modifiers*　*return-type*　　*member-name*　（　*formal-parameter-list* ）

传递给方法的参数在方法的形式化参数表 *formal-parameter-list* 中声明，我们将随后进行详细论述。

在方法的声明中，至少应包括方法名称、修饰符和参数类型，返回值和参数名则不是必须的。

注意：方法名 member-name 不应与同一个类中的其它方法同名，也不能与类中的其它成员名称相同。

修饰符

方法的修饰符 method-modifier 可以是：

- new
- public
- protected
- internal
- private
- static
- virtual
- sealed
- override

- abstract
- extern

对于使用了 abstract 和 extern 修饰符的方法，方法的执行体 *method-body* 仅仅只有一个简单的分号。其它所有的方法执行体中应包含调用该方法所要执行的语句。

返回值

方法的返回值的类型可以是合法的 C#的数据类型。C#在方法的执行部分通过 return 语句得到返回值，如

程序清单 11-1：

```
using System;
class Test
{
        public int max(int x,int y){
                if(x>y)
                        return x;
                else
                        return y;
        }
        public void Main(){
                Console.WriteLine("the max of 6 and 8 is:{0}",max(6,8));
        }
}
```

程序的输出是：

```
the max of 6 and 8 is:8
```

如果在 return 后不跟随任何值，方法返回值是 void 型的。

11.2 方法中的参数

C#中方法的参数有四种类型：

- 值参数，不含任何修饰符。
- 引用型参数，以 ref 修饰符声明。
- 输出参数，以 out 修饰符声明。
- 数组型参数，以 params 修饰符声明。

11.2.1 值参数

当利用值向方法传递参数时，编译程序给实参的值做一份拷贝，并且将此拷贝传递给该方法。被调用的方法不会修改内存中实参的值，所以使用值参数时，可以保证

实际值是安全的。在调用方法时，如果形式化参数的类型是值参数的话，调用的实参的表达式必须保证是正确的值表达式。在下面的例子中，程序员并没有实现他希望交换值的目的：

程序清单 11-2：

```
using System;
class Test
{
  static void Swap(int x, int y) {
        int temp = x;
        x = y;
        y = temp;
  }
        static void Main() {
        int i = 1, j = 2;
        Swap(i, j);
        Console.WriteLine("i = {0}, j = {1}", i, j);
  }
}
```

编译上述代码，程序将输出：

```
i = 1, j = 2
```

11.2.2 引用型参数

和值参不同的是，引用型参数并不开辟新的内存区域。当利用引用型参数向方法传递形参时，编译程序将把实际值在内存中的地址传递给方法。

在方法中，引用型参数通常已经初始化。再看下面的例子。

程序清单 11-3：

```
using System;
class Test
{
  static void Swap(ref int x, ref int y) {
        int temp = x;
        x = y;
        y = temp;
  }
        static void Main() {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine("i = {0}, j = {1}", i, j);
```

```
    }
  }
```

编译上述代码，程序将输出：

```
i = 2, j = 1
```

Main 函数中调用了 Swap 函数，x 代表 i ， y 代表 j。这样，调用成功地实现了 i 和 j 的值交换。

在方法中使用引用型参数，会经常可能导致多个变量名指向同一处内存地址。见示例：

```
class A
{
  string s;
        void F(ref string a, ref string b) {
        s = "One";
        a = "Two";
        b = "Three";
  }
        void G() {
        F(ref s, ref s);
  }
}
```

在方法 G 对 F 的调用过程中，s 的引用被同时传递给了 a 和 b。此时，s, a, b 同时指向了同一块内存区域。

11.2.3 输出参数

与引用型参数类似，输出型参数也不开辟新的内存区域。与引用型参数的差别在于，调用方法前无需对变量进行初始化。输出型参数用于传递方法返回的数据。

out 修饰符后应跟随与形参的类型相同的类型声明。在方法返回后，传递的变量被认为经过了初始化。

程序清单 11-4：

```
using System;
class Test
{
    static void SplitPath(string path, out string dir, out string name)    {
        int i = path.Length;
        while (i > 0) {
            char ch = path[i – 1];
            if (ch == '\\' || ch == '/' || ch == ':') break;
            i--;
```

```
        }
        dir = path.Substring(0, i);
        name = path.Substring(i);
    }


        static void Main() {
        string dir, name;
        SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
        Console.WriteLine(dir);
        Console.WriteLine(name);
    }
}
```

可以预计，程序的输出将会是：

```
c:\Windows\System\
hello.txt
```

我们注意到，变量 dir 和 name 在传递给 SplitPath 之前并未初始化，在调用之后它们则有了明确的值。

11.2.4 数组型参数

如果形参表中包含了数组型参数，那么它必须在参数表中位于最后。另外，参数只允许是一维数组。比如，string[]和 string[][]类型都可以作为数组型参数，而 string[,]则不能。最后，数组型参数不能再有 ref 和 out 修饰符。

程序清单 11-5：

```
using System;
class Test
{
  static void F(params int[] args) {
        Console.WriteLine("Array contains {0} elements:", args.Length);
        foreach (int i in args) Console.Write(" {0}", i);
        Console.WriteLine();
  }
        public static void Main() {
        int[] a = {1, 2, 3};
        F(a);
        F(10, 20, 30, 40);
        F();
  }
}
```

程序输出：

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

在上例中，第一次调用 F 是简单地把数组 a 作为值参数传递；第二次调用把已给出数值的数组传递给了 F；而在第三次调用中，F 创建了含有 0 个元素的整型数组作为参数传递。后两次调用完整的写法应该是：

```
F(new int[] {10, 20, 30, 40});
F(new int[] {});
```

11.3 静态和非静态的方法

C#的类定义中可以包含两种方法：静态的和非静态的。使用了 static 修饰符的方法为静态方法，反之则是非静态的。

静态方法是一种特殊的成员方法，它不属于类的某一个具体的实例。非静态方法可以访问类中的任何成员，而静态方法只能访问类中的静态成员。看这个例子：

```
class A
{
  int x;
  static int y;
  static int F() {
        x = 1;      //错误，不允许访问
        y = 2;  //正确，允许访问
  }
```

在这个类定义中，静态方法 F()可以访问类中静态成员 s，但不能访问非静态的成员 x。这是因为，x 作为非静态成员，在类的每个实例中都占有一个存储（或者说具有一个副本），而静态方法是类所共享的，它无法判断出当前的 x 属于哪个类的实例，所以不知道应该到内存的哪个地址去读取当前 x 的值。而 y 是非静态成员，所有类的实例都公用一个副本，静态方法 F 使用它就不存在什么问题。

那么，是不是静态方法就无法识别类的实例了呢？在 C#中，我们可以灵活地采用传递参数的办法。第十章我们提到了一个 windows 窗口程序的例子，这里我们再对这个例子进行一些改变。

程序清单 11-6：

```
using System;
class Window
{
  public string m_caption;      //窗口的标题
```

```
    public bool IsActive;          //窗口是否被激活
    public handle m_handle;      //窗口的句柄
    public static int m_total; //当前打开的窗口数目
    public handle Window(){
        m_total++; //窗口总数加 1
        // ……创建窗口的一些执行代码
        return m_handle;//窗口的返回值作为句柄
    }
    ~Window(){
        m_total--;//窗口总数减 1
        // ……撤消窗口的一些执行代码
    }
    public static string GetWindowCaption(Window w)
    {
        return w.m_caption;
    }
        //……窗口的其它成员
}
```

分析一下上面例子中的代码。每个窗口都有窗口标题 m_caption、窗口句柄 m_handle、窗口是否激活 IsActive 三个非静态的数据成员（窗口句柄是 Windows 操作系统中保存窗口相关信息的一种数据结构，我们在这个例子中简化了对句柄的使用）。系统中总共打开的窗口数目 m_total 作为一个静态成员。每个窗口调用构造函数创建，这时 m_total 的值加 1。窗口关闭或因为其它行为撤消时，通过析构函数 m_total 的值减 1。

我们要注意窗口类的静态方法 GetWindowCaption(Window w)。这里它通过参数 w 将对象传递给方法执行，这样它就可以通过具体的类的实例指明调用的对象，这时它可以访问具体实例中的成员，无论是静态成员还是非静态成员。

11.4 方法的重载

在前面的例子中，我们实际上已经看到了构造函数的重载。

程序清单 11-7：

```
using System;
class Vehicle//定义汽车类
{
    public      int wheels; //公有成员：轮子个数
    protected   float weight;          //保护成员：重量
```

```
        public      Vehicle(){;}
        public      Vehicle(int w,float g){
                    wheels = w;
                    weight = g;
                }
    public void Show(){
                Console.WriteLine("the wheel of vehicle is:{0}",wheels);
                Console.WriteLine("the weight of vehicle is:{0}",weight);
        }
};
```

类的成员方法的重载也是类似的。类中两个以上的方法（包括隐藏的继承而来的方法），取的名字相同，只要使用的参数类型或者参数个数不同，编译器便知道在何种情况下应该调用哪个方法，这就叫做方法的重载。

其实，我们非常熟悉的 Console 类之所以能够实现对字符串进行格式化的功能，就是因为它定义了多个重载的成员方法：

```
public static void WriteLine();
public static void WriteLine(int);
public static void WriteLine(float);
public static void WriteLine(long);
public static void WriteLine(uint);
public static void WriteLine(char);
public static void WriteLine(bool);
public static void WriteLine(double);
public static void WriteLine(char[]);
public static void WriteLine(string);
public static void WriteLine(Object);
public static void WriteLine(ulong);
public static void WriteLine(string, Object[]);
public static void WriteLine(string, Object);
public static void WriteLine(char[], int, int);
public static void WriteLine(string, Object, Object);
public static void WriteLine(string, Object, Object, Object);
```

我们举个例子来说明重载的使用。学生类中包含有学生姓名、性别、年龄、体重等信息。我们需要比较学生之间的年龄和体重，我们可以采用下面的代码。

程序清单 11-8：

```
using System;
class Student//定义学生类
{
        public string s_name;
```

```
        public int s_age;
        public float s_weight;
    public Student(string n,int a,float w){
        s_name = n;
        s_age = a;
        s_weight =w;
    }
        public int max_age(int x,int y){
            if(x>y) return x;
            else return y;
        }
        public float max_weight(float x, float y){
            if(x>y) return x;
            else return y;
    }
}
class Test
{
    public static void Main(){
        Student s1 = new Student("Mike",21,70);
        Student s2 = new Student("John",21,70);
        if(s1.max_age(s1.s_age,s2.s_age)==s1.s_age)
           Console.WriteLine("{0}'s age is bigger than {1}'s", s1.s_name,s2.s_name);
        else
           Console.WriteLine("{0}'s age is smaller than {1}'s", s1.s_name,s2.s_name);
        if(s1.max_weight(s1.s_weight,s2.s_weight)==s1.s_weight)
           Console.WriteLine("{0}'s weight is bigger than    {1}'s", s1.s_name,s2.s_name);
        else
            Console.WriteLine("{0}'s weight is smaller than {1}'s", s1.s_name,s2.s_name);
    }
}
```

由于 C#支持重载，我们可以给两个比较大小的方法取同一个名字 max，程序员在调用方法时，只需在其中带入实参，编译器就会根据实参类型来决定到底调用哪个重载方法。程序员只需要使用 max 这个方法名，剩下的就是系统的事了。那样，代码我们改写为：

程序清单 11-9：

```
using System;
class Student//定义学生类
{
```

```
        public string s_name;
        public int s_age;
        public float s_weight;
    public Student(string n,int a,float w){
        s_name = n;
        s_age = a;
        s_weight =w;
    }
        public int max (int x,int y){
            if(x>y) return x;
            else return y;
        }
        public float max (float x, float y){
            if(x>y) return x;
            else return y;
    }
}
class Test
{
    public static void Main(){
        Student s1 = new Student("Mike",21,70);
        Student s2 = new Student("John",21,70);
        if(s1.max (s1.s_age,s2.s_age)==s1.s_age)
            Console.WriteLine("{0}'s age is bigger than {1}'s", s1.s_name,s2.s_name);
        else
            Console.WriteLine("{0}'s age is smaller than {1}'s", s1.s_name,s2.s_name);
        if(s1.max (s1.s_weight,s2.s_weight)==s1.s_weight)
                Console.WriteLine("{0}'s weight is bigger than    {1}'s",
                s1.s_name,s2.s_name);
        else
            Console.WriteLine("{0}'s weight is smaller than {1}'s", s1.s_name,s2.s_name);
    }
}
```

11.5　操作符重载

11.5.1　问题的提出

在面向对象的程序设计中，自己定义一个类，就等于创建了一个新类型。类的实例和变量一样，可以作为参数传递，也可以作为返回类型。

在第七章中，我们介绍了系统定义的许多操作符。比如对于两个整型变量，使用算术操作符可以简便地进行算术运算：

```
class A
{
        public int x;
        public int y;
        public int Plus{
                return x+y;
        }
}
```

再比如，我们希望将属于不同类的两个实例的数据内容相加：

```
class B
{
        public int x;
}
class Test
{
  public int z;
  public static void Main{
        A a = new A();
        B b = new B();
        z = a.x + b.x;
  }
}
```

使用 a.x + b.x 这种写法不够简洁，也不够直观。更为严重的问题是，如果类的成员在声明时使用的不是 public 修饰符的话，这种访问就是非法的。

我们知道，在 C#中，所有数据要么属于某个类，要么属于某个类的实例，充分体现了面向对象的思想。因此，为了表达上的方便，人们希望可以重新给已定义的操作符赋予新的含义，在特定的类的实例上进行新的解释。这就需要通过操作符重载来解决。

11.5.2 使用成员方法重载操作符

C#中，操作符重载总是在类中进行声明，并且通过调用类的成员方法来实现。

操作符重载声明的格式为：

type operator operator-name (formal-param-list)

C#中，下列操作符都是可以重载的：

+ - ! ~ ++ -- true false

* / % & | ^ << >> == != > < >= <=

但也有一些操作符是不允许进行重载的，如：

=, *&&*, ||, *?:*, *new*, *typeof*, *sizeof*, *is*

一元操作符重载

顾名思义，一元操作符重载时操作符只作用于一个对象，此时参数表为空，当前对象作为操作符的单操作数。

下面我们举一个角色类游戏中经常遇到的例子。扮演的角色具有内力、体力、经验值、剩余体力、剩余内力五个属性，每当经验值达到一定程度时，角色便会升级，体力、内力上升，剩余体力和内力补满。“升级”我们使用重载操作符“++”来实现。

程序清单 11-10：

```
using System;
class Player
{
  public int neili;
  public int tili;
  public int jingyan;
  public int neili_r;
  public int tili_r;
  public Player()
  {
        neili = 10;
        tili = 50;
        jingyan = 0;
        neili_r = 50;
        tili_r = 50;
  }
  public static Player operator ++(Player p){
        p.neili = p.neili + 50;
        p.tili = p.tili + 100;
        p.neili_r = p.neili;
        p.tili_r = p.tili;
```

```
        return p;
    }
    public void Show()
    {
        Console.WriteLine("Tili: {0}",tili);
        Console.WriteLine("Jingyan: {0}",jingyan);
        Console.WriteLine("Neili: {0}",neili);
        Console.WriteLine("Tili_full: {0}",tili_r);
        Console.WriteLine("Neili_full: {0}",neili_r);
    }
}

class Test
{
    public static void Main(){
        Player man = new Player();
        man.Show();
        man++;
        Console.WriteLine("Now upgrading...:");
        man.Show();
    }
}
```

二元操作符重载

大多数情况下我们使用二元操作符重载。这时参数表中有一个参数，当前对象作为该操作符的左操作数，参数作为操作符的右操作数。

下面我们给出二元操作符重载的一个简单例子，即笛卡儿坐标相加。

程序清单 11-11：

```
Using system;
class DKR
{
    public int x,y,z;
    public DKR(int vx,int vy,int vz){
        x = vx;
        y = vy;
        z = vz;
    }
    public static DKR operator +(DKR d1,DKR d2)
    {
```

```
        DKR dkr = new DKR(0,0,0);
        dkr.x = d1.x + d2.x;
        dkr.y = d1.y + d2.y;
        dkr.z = d1.z + d2.z;
        return dkr;
    }
}

class Test
{
    public static void Main(){
        DKR d1 = new DKR(3,2,1);
        DKR d2 = new DKR(0,6,5);
        DKR d3 = d1+d2;
        Console.WriteLine("The 3d location of d3 is:{0},{1},{2}",d3.x,d3.y,d3.z);
    }
}
```

试着编译、运行该程序，看结果是否与预期的一致。

11.6 小　　结

C#中类的功能主要由类的成员方法来完成，类的成员方法也常常提供对类的私有成员的访问。本章首先介绍了方法的声明格式和如何给方法返回值，之后详细地介绍了传递给方法的四种参数。我们还讨论了：

- 静态方法与非静态方法。
- 方法的重载。
- 操作符重载。

复习题

（1）方法的四种参数分别是什么，它们各自有什么用处？

（2）使用 return 语句返回值和使用输出型参数有什么区别，它们各自适用于什么场合下使用？

（3）类的静态成员方法是否只能对静态的数据成员进行操作。

（4）类的静态成员方法属于类所公有，那么在调用静态方法时是否一定需要指明调用的类的实例。

（5）阅读下面的程序，找出其中错误并改正过来：

```
using system
```

```
class Triangle
{
  private int a;
  private int b;
  private int c;
  public Triangle(int va,int vb,int vc){
        a = va;
        b = vb;
        c = vc;
  }
  public void Main(){
        Triangle tr = new Triangle(2,2);
        Console.WriteLine("{0},{1}",tr.a,tr.b);
        return 0;
  }
}
```

（6）简述一元和二元操作符重载有什么区别。

（7）定义一个时钟类，以及类的三个属性：时、分、秒。定义在类上的各种操作，要求用操作符重载来完成。

（8）继续上一章网络用户的例子，为网络用户类添加注册功能，注册时对用户输入的 ID 进行合法性检查（小写字母开头，只能由字母、数字和下划线组成），并验证用户两次输入密码是否相同。

第十二章　域 和 属 性

为了保存类的实例的各种数据信息，C#给我们提供了两种方法：域和属性。其中，属性实现了良好的数据封装和数据隐藏。

12.1　域

12.1.1　声明

域表示与对象或类相关联的变量，声明格式如下：

attributes　field-modifiers　type　variable-declarators　;

域的修饰符 field-modifiers 可以是：

- *new*
- *public*
- *protected*
- *internal*
- *private*
- *static*
- *readonly*

实际上，域相当于 C++中的类的简单成员变量。在下面的代码中，类 A 包含了三个域：公有的 x 和 y，以及私有的 z。

```
class A
{
  public int x;
  public string y;
  private float z;
}
```

12.1.2　静态域和非静态域

静态域的声明是使用 static 修饰符，其它的域都是非静态域。静态域和非静态域分别属于 C#中静态变量和非静态变量。

若将一个域说明为静态的，无论建立多少个该类的实例，内存中只存在一个静态数据的拷贝。当这个类的第一个实例建立时，域被初始化。以后再进行类的实例化时，

不再对其进行初始化，所有属于这个类的实例共享一个副本。

与之相反，非静态域在类的每次实例化时，每个实例都拥有一份单独的拷贝。

下面的例子清楚地反映了二者之间的区别。

程序清单 12-1:

```
using System;
public class Count
{
        static int count;
        int number;
        public Count(){
                count = count + 1;
                number = count;
        }
        public void show(){
        Console.WriteLine("object{0}:count={1}", number,count);
 }
}

            class Test
     {
              public static void Main() {
                          Count a = new Count();
                          a.show();
                          Console.WriteLine();
                    Count b = new Count();
                          a.show();
                          b.show();
                          Console.WriteLine();
                    Count c = new Count();
                          a.show();
                          b.show();
                          c.show();
              }
            }
```

上面的例子中，类 Count 中域 count 被声明为静态，为所有类的实例所共享。类每进行一 1 次实例化，它的值就加 1，这个操作就在构造函数中实现，因而可以用于对系统中类的实例数进行计数。

域 number 用来存放当前实例的编号。当类被实例化时，在构造函数中就对编号进行赋值，从而可以看出实例化的顺序。

方法 show（）用来在屏幕上打印出当前类的实例数，还有类的各个实例的编号。

程序的运行结果应为：

```
object1:count=1
object1:count=2
object2:count=2
object1:count=3
object2:count=3
object3:count=3
```

从上面的例子中可以看出，无论何时，类的所有实例的 count 值都是相同的，说明它们共享一个数据，count 域只有一个副本。而每个实例的标号都是不同的，一旦实例化，标号就不再变化了。

12.1.3 只读域

域的声明中如果加上了 readonly 修饰符，表明该域为只读域。对于只读域我们只能在域的定义中和它所属类的构造函数中进行修改，在其它情况下，域是“只读”的。

熟悉 C 和 C++程序员可能习惯了使用 const 和#define 定义一些容易记住的名字来表示某个数值。static 和 readonly 修饰符可以起到同样的效果：

```
public class A
{
  public static readonly double PI = 3.14159;
  public static readonly Color White = new Color(255, 255, 255);
  public static readonly int kByte = 1024;
  ……//other members
}
```

这样，在程序中我们就可以直接使用 PI 来指代圆周率，white 来表示白色，等等。

那么，使用 static readonly 与使用 const 有什么区别呢？简单地说，const 型表达式的值在编译时形成，而 static readonly 表达式的值直到程序运行时才形成。看下面这个例子。

程序清单 12-2:

```
using System;
namespace Program1
{
  public class A
  {
        public static readonly int X = 1;
  }
}
```

```
namespace Program2
{
  class Test
  {
        static void Main() {
              Console.WriteLine(Program1.A.X);
        }
  }
}
```

假定名字空间 Program1 和 Program2 表示两个分别独立编译的程序（有关名字空间的概念我们将放在后续章节中介绍）。在这里，域 x 为静态只读的，它的值由于是在编译时形成的，所以无论是否改变 Program1 中 x 的值，只要不重新编译 Program2，Program2 的输出就不会发生变化。如果 Program2 已经安装在用户的系统上，对 Program1 的升级不会影响到旧的 Program2 的使用。这种技术有利于进行版本控制。

12.1.4 域的初始化

注意：在 C 和 C++中，未经初始化的变量是不能使用的。在 C#中，系统将为每个未经初始化的变量提供一个默认值。这虽然在某种程度上保证了程序的安全性，但对本应初始化为某个特殊值的变量忘记了初始化，也常常会导致程序的执行误入歧途。

对于静态变量、非静态的对象变量和数组元素，这些变量自动初始化为本身的默认值。对于所有引用类型的变量，默认值为 null。所有值类型的变量的默认值见下表所示。

表 12-1 部分类型的域初始化的值

变量类型	默认值
sbyte, byte, short, ushort, int, uint, long, ulong	0
char	\x0000
float	0.0f
double	0.0d
decimal	0.0m
bool	false
enum	0

对于 struct 类型的变量，默认的初始化将对构成该结构的每一个值类型初始化为上表中的默认值，对构成其的每一个引用类型初始化为 null。

如果在类中，没有显式地对域进行初始化，系统将赋予其一个默认值。域的默认初始化分为两种情况：对于静态域，类在装载时对其进行初始化；对于非静态域，在类的实例创建时进行初始化。在默认的初始化之前，域的值是不可预测的。

比如，下面的代码是合法的：

```
class Test
{
  static int a = b + 1;
  static int b = a + 1;
}
```

实际上等价于：

```
a = 1, b = 2
```

而下面的代码则是非法的：

```
class A
{
  int x = 1;
  int y = x + 1;
}
```

因为非静态变量 x 在类 A 实例化以前并没有初始化，代码 y = x + 1 无法得到正确的 x 的值。

12.2 属　　性

属性是对现实世界中实体特征的抽象，它提供了对类或对象性质的访问。比如，一个用户的姓名、一个文件的大小、一个窗口的标题，都可以作为属性。类的属性所描述的是状态信息，在类的某个实例中属性的值表示该对象的状态值。

C#中的属性更充分地体现了对象的封装性：不直接操作类的数据内容，而是通过访问器进行访问。它借助于 get 和 set 对属性的值进行读写，这在 C++中是需要程序员手工完成的一项工作。

12.2.1 声明

属性采用如下方式进行声明：

attributes property-modifiers type member-name { accessor-declarations }

属性的修饰符 *property-modifiers* 有：

- new
- public
- protected
- internal
- private
- static
- virtual
- sealed

- override
- abstract

以上修饰符中，static, virtual, override 和 abstract 修饰符不能同时使用。

属性的访问声明 *accessor-declaration* 必须用一对“{” 和 “}”大括号括起来，在其中给出对属性的值进行读写的操作说明。

虽然属性和域的语法比较类似，但不能把属性当做变量那样使用，也不能把属性作为引用型参数或输出参数来进行传递。

12.2.2 访问属性的值

在属性的访问声明中，对属性的值的读操作用 get 关键字标出，对属性的值的写操作用 set 关键字标出。我们看下面的例子。

程序清单 12-3:

```
using System;
public class File
{
  private string s_filename;
        public string Filename {
        get {
              return s_filename;
        }
        set {
              if (s_filename != value) {
                    s_filename = value;
                    }
        }
  }
}
public class Test
{
  public static void Main(){
        File f = new File();
        f.Filename = “myfile.txt”;
        string s = f.Filename;
}
}
```

在属性的访问声明中：

- 只有 set 访问器，表明属性的值只能进行设置而不能读出。
- 只有 get 访问器，表明属性的值是只读的，不能改写。

- 同时具有 set 访问器和 get 访问器，表明属性的值的读写都是允许的。

除了使用了 abstract 修饰符的抽象属性，每个访问器的执行体中只有分号“;”，其它所有属性的 get 访问器都通过 return 来读取属性的值，set 访问器都通过 value 来设置属性的值。

举个例子，旅馆对住宿人员进行登记，要记录的信息有：客人姓名、性别、所住的房间号、已住宿的天数。这里，客人的姓名和性别一经确定就不能再更改了，用户可以要求改变房间，住宿的天数当然也是不断变化的。我们在类的构造函数中对客人的姓名和性别进行初始化，在四个属性中，客人的姓名和性别是只读的，故只具有 get 访问器；房间号和住宿天数允许改变，同时具有 set 访问器和 get 访问器。

程序清单 12-4:

```
using System;
public class Customer
{
  public enum sex
  {
        man,
        woman,
        };
  private string s_name;
  public string Name{
        get {
              return s_name;
        }
  }
  private sex m_sex;
  public sex Sex{
        get {
              return m_sex;
        }
  }
  private string s_no;
  public string No{
        get {
              return s_no;
        }
        set {
              if (s_no != value) {
                    s_no = value;
              }
```

```
        }
    }
    private int i_day;
    public int Day{
        get {
            return i_day;
        }
        set {
            if (i_day != value) {
                i_day = value;
                }
        }
    }
    public void Customer(string name,sex sex,string no,int day)
    {
        s_name = name;
        m_sex = sex;
        s_no = no;
        i_day = day;
    }
}
```

12.3 小　　结

描述一个类的特性有两种方式，一是通过域，一是通过属性。域作为 public 类型的成员变量访问，而属性不能直接进行访问，必须通过访问器（accessors）进行。

本章中我们还讲述了静态域和非静态域的区别，在什么情况下使用只读域，以及变量默认的初始化值。

复习题

（1）域和属性有哪些相同和不同之处？

（2）如何设置类的属性读写权限？

（3）分析下面的不良代码的执行后果：

```
class Counter
{
  private int next;
```

```
    public int Next {
        get { return next++; }
    }
}
```

（4）继续网络用户类的扩展，添加用户网龄、真实姓名、性别、出生年月三条信息，采用属性对它们进行读写。

第十三章　事件和索引指示器

事件为类和类的实例提供了向外界发送通知的能力，而索引指示器则可以像数组那样对对象进行索引访问。在 C 和 C++中，没有事件和索引指示器的概念，它们是在C#首次提出的。

13.1　事　　件

形象地说，事件（event）就是类或对象用来“发出通知”的成员。通过提供事件的句柄，客户能够把事件和可执行代码联系在一起。

让我们一起先来看一个事件的例子。如果你熟悉 MFC 的话，理解这个例子应该不会很难。

程序清单 13-1:

```
public delegate void EventHandler(object sender, EventArgs e);
public class Button: Control
{
  public event EventHandler Click;
  protected void OnClick(EventArgs e) {
        if (Click != null) Click(this, e);
  }
  public void Reset() {
        Click = null;
  }
}
```

在这个例子中，Click 是类 Button 的一个域，我们可以获得它的值并进行修改。OnClick 方法用于触发 Click 事件。

13.1.1　事件的声明

事件的声明格式：

attributes *event-modifiers* `event` *type* *variable-declarators* `;`
attributes *event-modifiers* `event` *type* *member-name*
`{` *event-accessor-declarations* `}`

事件的修饰符 event-modifier 可以是：

- *new*

- *public*
- *protected*
- *internal*
- *private*
- *static*
- *virtual*
- *sealed*
- *override*
- *abstract*

static, virtual, override 和 abstract 修饰符同一时刻只能出现一个。事件的声明中可以包含事件访问说明，或者依靠编译器自动提供一个访问器；它也可以省略事件访问说明，一次定义一个或多个事件。上面的例子中就省略了这个说明。

注意：使用了 abstract 修饰符的抽象事件必须省略事件访问说明，否则编译器会提示错误。

事件所声明的类型必须是一个代表（delegate）类型，代表类型应预先声明，如上例中的 public delegate void EventHandler。

13.1.2 事件的预订和撤消

在随后的例子中，我们声明了一个使用 Button 类的登录对话框类。对话框类含有两个按钮：OK 和 Cancel 按钮。

程序清单 13-2:

```
public class LoginDialog: Form
{
  Button OkButton;
  Button CancelButton;
  public LoginDialog() {
        OkButton = new Button(...);
        OkButton.Click += new EventHandler(OkButtonClick);
        CancelButton = new Button(...);
        CancelButton.Click += new EventHandler(CancelButtonClick);
  }
  void OkButtonClick(object sender, EventArgs e) {
        // 处理 OkButton.Click 事件
  }
  void CancelButtonClick(object sender, EventArgs e) {
        // 处理 CancelButton.Click 事件
  }
}
```

在例子中使用了 Button 类的两个实例，事件的预订是通过为事件加上左操作符“+=”来实现的：

```
OkButton.Click += new EventHandler(OkButtonClick);
```

这样，只要事件被触发，方法就会被调用。

事件的撤消则采用左操作符“-=”：

```
OkButton.Click -= new EventHandler(OkButtonClick);
```

如果在类中声明了事件，我们又希望像使用域的方式那样使用事件，那么这个事件就不能是抽象的，也不能显式地包含事件访问声明。满足了这两个条件后，在任何可以使用域的场合都同样可以使用事件。

注意：对事件的触发相当于调用事件所表示的原型——delegate，所以对 delegate 型原型的调用必须先经过检查，确保 delegate 不是 null 型的。

13.1.3 事件访问器

如 Button 的例子所示，大多数情况下事件的声明都省略了事件访问声明。什么情况下使用事件访问声明呢？答案是：如果每个事件的存储开销太大，我们就可以在类中包含事件访问声明，按私有成员的规则存放事件句柄列表。

访问器的声明包括两种：添加访问器声明（*add-accessor-declaration*）和删除访问器声明（*remove-accessor-declaration*）。

访问器声明之后跟随相关执行代码的语句块。在添加访问器声明后的代码需要执行添加事件句柄的操作，在删除访问器声明后的代码需要执行删除事件句柄的操作。不管是哪种事件访问器，都对应相应的一个方法，这个方法只有一个事件类型的值参数，并且返回值为 void。

在执行预订操作时使用添加型访问器，在执行撤消操作时使用删除型访问器。访问器中实际上还包含了一个名为 value 的隐藏的参数，因而访问器在使用局部变量时不能再使用这个名字。

下面给出了使用访问器的例子。

程序清单 13-3:

```
class Control: Component
{
  // Unique keys for events
  static readonly object mouseDownEventKey = new object();
  static readonly object mouseUpEventKey = new object();
  // Return event handler associated with key
  protected Delegate GetEventHandler(object key) {...}
  // Add event handler associated with key
  protected void AddEventHandler(object key, Delegate handler) {...}
  // Remove event handler associated with key
  protected void RemoveEventHandler(object key, Delegate handler) {...}
```

```
    // MouseDown event
    public event MouseEventHandler MouseDown {
        add { AddEventHandler(mouseDownEventKey, value); }
        remove { AddEventHandler(mouseDownEventKey, value); }
    }
    // MouseUp event
    public event MouseEventHandler MouseUp {
        add { AddEventHandler(mouseUpEventKey, value); }
        remove { AddEventHandler(mouseUpEventKey, value); }
    }
}
```

13.1.4 静态事件

和域、方法等一样，在声明中使用了修饰符的事件称之为静态事件。静态事件不与具体的实例相关联，因此不能在静态事件的访问器中引用 this 关键字。此外，在静态事件声明时又加上 virtual, abstract 或 override 修饰符也都是不合法的。而对于非静态的事件，我们可以在事件的访问器中使用 this 来指代类的实例。

13.2 索引指示器

索引指示器（indexer）使得可以像数组那样对对象使用下标。它为我们提供了通过索引方式方便地访问类的数据信息的方法。

13.2.1 声明

还是让我们先来看一下索引指示器的声明格式：

attributes indexer-modifiers indexer-declarator { *accessor-declarations* }

索引指示器可以使用的修饰符 indexer-modifier 有：

- new
- public
- protected
- internal
- private
- virtual
- sealed
- override

- abstract

一对大括号“{}”之间是索引指示器的访问声明，使用 get 关键字和 set 关键字定义了对被索引的元素的读写权限。

例如，下面的例子用于打印出小组人员的名单。

程序清单 13-4:

```
using System
class Team
{
 string s_name = new string[8];
 public string this[int nIndex]
 {
        get{
              return s_name[nIndex];
        }
        set{
              s_name[nIndex] = value;
        }
  }
 }
 class Test
 {
        public static void Main(){
              Team t1 = new Team();
              for(int i=0; i<6 ;i++)
                    Console.WriteLine(t1[i]);
        }
 }
```

在许多情况下，某些数据信息应该是属于类或类的实例所私有的，需要限制对这些信息的访问。而我们有时又不希望这类数据对外界完全封闭。和属性一样，索引指示器为我们提供了控制访问权限的另一种办法。

13.2.2 实例

本实例给出运用索引指示器的一个简单例子。例子是一个网络应用程序：根据域名解析 IP 地址。

程序清单 13-5:

```
using System;
using System.Net;
```

```
class ResolveDNS
{
  IPAddress[] m_arrayIPs;
  Public void Resolve(string s_host){
        IPHostEntry ip = DNS.GetHostByName(s_host);
        m_arrayIPs = ip.AddressList;
  }
  public IPAddress this[int nIndex]{
        get{
              return m_arrayIPs[nIndex];
        }
  }
  public int IPLength{
        get{
              return m_arrayIPs.Length;
        }
  }
}

class TestApp
{
  public static void Main()
  {
        ResolveDNS resolver1 = new ResolveDNS();
        resolver1.Resolve(www.sohu.com);
        int n = resolver1.IPLength;
        Console.WriteLine("Get the IP Address of the host");
        Console.WriteLine();
        for(int i=0;i<n;i++)
              Console.WriteLine(resolver1[i]);
  }
}
```

程序的几点说明：

使用 System.Net 名字空间中的 DNS 类可以解析主机名。DNS 类中提供了一个静态方法 GetHostByName，这个方法返回一个 IPHostEntry 的对象，这个对象中含有 IP 地址列表。

在编译该程序时，必须在编译器中声明包含 System.Net 名字空间：

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

有关 csc 的编译参数可以使用 csc/? 来浏览。

13.3 小　　结

在编写类的时候，我们可以利用事件向客户说明发生了什么事情。事件可以在类的域或属性中加以说明，但事件的类型必须是代表型（delegate）的。我们可以预订事件来保证事件将被触发，也可以撤消事件。

索引指示器使我们可以像使用数组那样为类或类的实例添加路径列表，通过下标进行访问类中的信息。

复习题

（1）事件与 Windows 中的消息是否相同。

（2）如何预订和撤消一个事件？

（3）什么情况下使用静态事件？

（4）说明用户是否可以定义索引指示器的名称。

（5）使用索引指示器时，如何避免数组的越界问题？

（6）使用索引指示器能够为我们带来哪些方便？

（7）继续网络用户的例子。允许用户修改个人信息，用户凭用户 ID 和密码登录，修改完毕后提交或者放弃返回。使用事件处理用户登录和修改信息的提交。

第十四章　继　　承

为了提高软件模块的可复用性和可扩充性，以便提高软件的开发效率，我们总是希望能够利用前人或自己以前的开发成果，同时又希望在自己的开发过程中能够有足够的灵活性，不拘泥于复用的模块。今天，任何面向对象的程序设计语言都必须提供两个重要的特性：继承性（inheritance）和多态性（polymorphism）。

如果所有的类都处在同一级别上，这种没有相互关系的平坦结构就会限制了系统面向对象的特性。继承的引入，就是在类之间建立一种相交关系，使得新定义的派生类的实例可以继承已有的基类的特征和能力，而且可以加入新的特性或者是修改已有的特性，建立起类的层次。

同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果，这就是多态性。多态性通过派生类重载基类中的虚函数型方法来实现。

本章介绍了：

- C#中继承的基本概念。
- 方法覆盖。
- 如何通过虚方法来实现对象的多态性。
- 抽象和密封的概念。
- 属性及其访问器的重载。

14.1　C#的继承机制

14.1.1　概述

现实世界中的许多实体之间不是相互孤立的，它们往往具有共同的特征，也存在内在的差别。人们可以采用层次结构来描述这些实体之间的相似之处和不同之处。

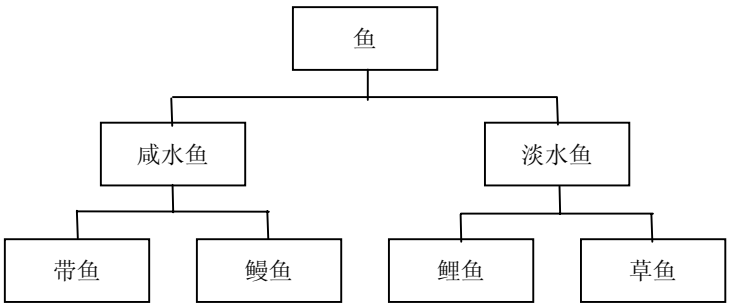


图 14-1　类的层次结构示例

上图反映了鱼类的派生关系。最高层的实体往往具有最一般最普遍的特征，越下层的事物越具体，并且下层包含了上层的特征，它们之间的关系是基类与派生类之间的关系。

为了用软件语言对现实世界中的层次结构进行模型化，面向对象的程序设计技术引入了继承的概念。一个类从另一个类派生出来时，派生类从基类那里继承特性。派生类也可以作为其它类的基类。从一个基类派生出来的多层类形成了类的层次结构。

注意：C#中，派生类只能从一个类中继承。这是因为，在 C++中，人们在大多数情况下不需要一个从多个类中派生的类。从多个基类中派生一个类，这往往会带来许多问题，从而抵消了这种灵活性带来的优势。

C#中，派生类从它的直接基类中继承成员：方法、域、属性、事件、索引指示器。除了构造函数和析构函数，派生类隐式地继承了直接基类的所有成员。

第十章中提到了一个关于车辆的例子，让我们再回顾一下这个例子，从中获得基类与派生类的感性认识。

程序清单 14-1:

```
using System;
class Vehicle       //定义汽车类
{
        int wheels; //公有成员：轮子个数
        protected float weight;         //保护成员：重量
        public      Vehicle(){;}
        public      Vehicle(int w,float g){
                    wheels = w;
                    weight = g;
                }
        public void Speak(){
                Console.WriteLine("the w vehicle is speaking!");
        }
};
class Car:Vehicle  //定义轿车类：从汽车类中继承
{
        int passengers;           //私有成员：乘客数
        public Car(int w,float g,int p) : base(w,g)
        {
                    wheels = w;
                    weight = g;
                    passengers = p;
        }
}
```

Vehicle 作为基类，体现了“汽车”这个实体具有的公共性质：汽车都有轮子和重

量。Car 类继承了 Vehicle 的这些性质，并且添加了自身的特性：可以搭载乘客。

C#中的继承符合下列规则：

- 继承是可传递的。如果 C 从 B 中派生，B 又从 A 中派生，那么 C 不仅继承了 B 中声明的成员，同样也继承了 A 中的成员。Object 类作为所有类的基类。
- 派生类应当是对基类的扩展。派生类可以添加新的成员，但不能除去已经继承的成员的定义。
- 构造函数和析构函数不能被继承。除此以外的其它成员，不论对它们定义了怎样的访问方式，都能被继承。基类中成员的访问方式只能决定派生类能否访问它们。
- 派生类如果定义了与继承而来的成员同名的新成员，就可以覆盖已继承的成员。但这并不因为这派生类删除了这些成员，只是不能再访问这些成员。
- 类可以定义虚方法、虚属性以及虚索引指示器，它的派生类能够重载这些成员，从而实现类可以展示出多态性。

14.1.2 覆盖

我们上面提到，类的成员声明中，可以声明与继承而来的成员同名的成员。这时我们称派生类的成员覆盖（hide）了基类的成员。这种情况下，编译器不会报告错误，但会给出一个警告。对派生类的成员使用 new 关键字，可以关闭这个警告。

前面汽车类的例子中，类 Car 继承了 Vehicle 的 Speak（）方法。我们可以给 Car 类也声明一个 Speak（）方法，覆盖 Vehicle 中的 Speak，见下面的代码。

程序清单 14-2:

```
using System;
class Vehicle//定义汽车类
{
        public      int wheels; //公有成员：轮子个数
        protected   float weight;           //保护成员：重量
        public      Vehicle(){;};
        public      Vehicle(int w,float g){
                    wheels = w;
                    weight = g;
                }
  public void Speak(){
                Console.WriteLine("the w vehicle is speaking!");
        }
}
class Car:Vehicle  //定义轿车类
{
        int passengers;          //私有成员：乘客数
        public Car(int w,float g,int
```

```
                wheels = w;
                weight = g;
                passengers = p;
            }
    new public void Speak(){
            Console.WriteLine("Di-di!");
    }
    }
```

注意：如果在成员声明中加上了 new 关键字修饰，而该成员事实上并没有覆盖继承的成员，编译器将会给出警告。在一个成员声明同时使用 new 和 override 则编译器会报告错误。

14.1.3 base 保留字

base 关键字主要是为派生类调用基类成员提供一个简写的方法。我们先看一个例子程序代码：

```
class A
{
  public void F(){
        //     F 的具体执行代码
   }
   public int this[int nIndex]{
        get{};
        set{};
   }
}

class B
{
public void G(){
        int x = base[0];
        base.F();
}
}
```

类 B 从类 A 中继承，B 的方法 G 中调用了 A 的方法 F 和索引指示器。方法 F 在进行编译时等价于：

```
public void G(){
  int x = (A（this）)[0];
        (A（this）).F();
```

}

使用 base 关键字对基类成员的访问格式为：

base . identifier

base [expression-list]

14.2 多 态 性

在面向对象的系统中，多态性是一个非常重要的概念，它允许客户对一个对象进行操作，由对象来完成一系列的动作，具体实现哪个动作、如何实现由系统负责解释。

14.2.1 C#中的多态性

“多态性”一词最早用于生物学，指同一种族的生物体具有相同的特性。

在 C#中，多态性的定义是：同一操作作用于不同的类的实例，不同的类将进行不同的解释，最后产生不同的执行结果。C#支持两种类型的多态性：

编译时的多态性

编译时的多态性是通过重载来实现的。我们在第十一章中介绍了方法重载和操作符重载，它们都实现了编译时的多态性。

对于非虚的成员来说，系统在编译时，根据传递的参数、返回的类型等信息决定实现何种操作。

运行时的多态性

运行时的多态性就是指直到系统运行时，才根据实际情况决定实现何种操作。C#中，运行时的多态性通过虚成员实现。

编译时的多态性为我们提供了运行速度快的特点，而运行时的多态性则带来了高度灵活和抽象的特点。

14.2.2 虚方法

当类中的方法声明前加上了 virtual 修饰符，我们称之为虚方法，反之为非虚。使用了 virtual 修饰符后，不允许再有 static, abstract, 或 override 修饰符。

对于非虚的方法，无论被其所在类的实例调用，还是被这个类的派生类的实例调用，方法的执行方式不变。而对于虚方法，它的执行方式可以被派生类改变，这种改变是通过方法的重载来实现的。

下面的例子说明了虚方法与非虚方法的区别。

程序清单 14-3:

```
using System;
```

```
class A
{
  public void F() { Console.WriteLine("A.F"); }
  public virtual void G() { Console.WriteLine("A.G"); }
}
class B: A
{
  new public void F() { Console.WriteLine("B.F"); }
  public override void G() { Console.WriteLine("B.G"); }
}
class Test
{
  static void Main() {
        B b = new B();
        A a = b;
        a.F();
        b.F();
        a.G();
        b.G();
  }
}
```

例子中，A 类提供了两个方法：非虚的 F 和虚方法 G。类 B 则提供了一个新的非虚的方法 F，从而覆盖了继承的 F；类 B 同时还重载了继承的方法 G。那么输出应该是：

```
A.F
B.F
B.G
B.G
```

注意到本例中，方法 a.G() 实际调用了 B.G，而不是 A.G。这是因为编译时值为 A，但运行时值为 B，所以 B 完成了对方法的实际调用。

14.2.3 在派生类中对虚方法进行重载

先让我们回顾一下普通的方法重载。普通的方法重载指的是：类中两个以上的方法（包括隐藏的继承而来的方法），取的名字相同，只要使用的参数类型或者参数个数不同，编译器便知道在何种情况下应该调用哪个方法。

而对基类虚方法的重载是函数重载的另一种特殊形式。在派生类中重新定义此虚函数时，要求的是方法名称、返回值类型、参数表中的参数个数、类型、顺序都必须与基类中的虚函数完全一致。在派生类中声明对虚方法的重载，要求在声明中加上 override 关键字，而且不能有 new, static 或 virtual 修饰符。

还是让我们用汽车类的例子来说明多态性的实现。

程序清单 14-4:

```
using System;
class Vehicle//定义汽车类
{
        public      int wheels; //公有成员：轮子个数
        protected   float weight;            //保护成员：重量
        public      Vehicle(int w,float g){
                    wheels = w;
                    weight = g;
        }
  public virtual void Speak(){
                Console.WriteLine("the w vehicle is speaking!");
        }
};

class Car:Vehicle  //定义轿车类
{
        int passengers;          //私有成员：乘客数
        public Car(int w,float g,int p) : base(w,g)
  {
                    wheels = w;
                    weight = g;
                    passengers = p;
        }
  public override void Speak(){
                Console.WriteLine("The car is speaking:Di-di!");
        }
}

class Truck:Vehicle      //定义卡车类
{
        int passengers;          //私有成员：乘客数
  float load;        //私有成员：载重量
        public Truck (int w,float g,int p，float l) : base(w,g)
  {
                    wheels = w;
                    weight = g;
                    passengers = p;
```

```
                load = l;
        }
    public override void Speak(){
            Console.WriteLine("The truck is speaking:Ba-ba!");
        }
}

class Test
{
    public static void Main(){
        Vehicle v1 = new Vehicle();
        Car c1 = new Car(4,2,5);
        Truck t1 = new Truck(6,5,3,10);
        v1.Speak();
        v1 = c1;
        v1.Speak();
        c1.Speak();
    v1 = t1;
    v1.Speak();
        t1.Speak();
      }
    }
```

分析上面的例子，我们看到：

- Vehicle 类中的 Speak 方法被声明为虚方法，那么在派生类中就可以重新定义此方法。
- 在派生类 Car 和 Truck 中分别重载了 Speak 方法，派生类中的方法原型和基类中的方法原型必须完全一致。
- 在 Test 类中，创建了 Vehicle 类的实例 v1，并且先后指向 Car 类的实例 c1 和 Truck 类的实例 t1。

运行该程序，结果应该是：

```
The Vehicle is speaking!
The car is speaking:Di-di!
The car is speaking:Di-di!
The truck is speaking:Ba-ba!
The truck is speaking:Ba-ba!
```

这里，Vehicle 类的实例 v1 先后被赋予 Car 类的实例 c1，以及 Truck 类的实例 t1 的值。在执行过程中，v1 先后指代不同的类的实例，从而调用不同的版本。这里 v1 的 Speak 方法实现了多态性，并且 v1.Speak（）究竟执行哪个版本，不是在程序编译时确定的，而是在程序的动态运行时，根据 v1 某一时刻的指代类型来确定的，所以还体现

了动态的多态性。

14.3 抽象与密封

14.3.1 抽象类

有时候，基类并不与具体的事物相联系，而是只表达一种抽象的概念，用以为它的派生类提供一个公共的界面。为此，C#中引入了抽象类（abstract class）的概念。

注意：C++程序员在这里最容易犯错误。C++中没有对抽象类进行直接声明的方法，而认为只要在类中定义了纯虚函数，这个类就是一个抽象类。纯虚函数的概念比较晦涩，直观上不容易为人们接受和掌握，因此 C#抛弃了这一概念。

抽象类使用 abstract 修饰符，对抽象类的使用有以下几点规定：

- 抽象类只能作为其它类的基类，它不能直接被实例化，而且对抽象类不能使用 new 操作符。抽象类如果含有抽象的变量或值，则它们要么是 null 类型，要么包含了对非抽象类的实例的引用。
- 抽象类允许包含抽象成员，虽然这不是必须的。
- 抽象类不能同时又是密封的。

如果一个非抽象类从抽象类中派生，则其必须通过重载来实现所有继承而来的抽象成员。请看下面的示例：

```
abstract class A
{
  public abstract void F();
}
abstract class B: A
{
  public void G() {}
}
class C: B
{
  public override void F() {
        // F 的具体实现代码
  }
}
```

抽象类 A 提供了一个抽象方法 F。类 B 从抽象类 A 中继承，并且又提供了一个方法 G；因为 B 中并没有包含对 F 的实现，所以 B 也必须是抽象类。类 C 从类 B 中继承，类中重载了抽象方法 F，并且提供了对 F 的具体实现，则类 C 允许是非抽象的。

让我们继续研究汽车类的例子。我们从“交通工具”这个角度来理解 Vehicle 类的

话，它应该表达一种抽象的概念，我们可以把它定义为抽象类。由轿车类 Car 和卡车类 Truck 来继承这个抽象类，它们作为可以实例化的类。

程序清单 14-5:

```
using System;
abstract class Vehicle    //定义汽车类
{
        public      int wheels; //公有成员：轮子个数
        protected   float weight;            //保护成员：重量
        public      Vehicle(int w,float g){
                    wheels = w;
                    weight = g;
        }
  public virtual void Speak(){
              Console.WriteLine("the w vehicle is speaking!");
        }
};

class Car:Vehicle  //定义轿车类
{
        int passengers;          //私有成员：乘客数
        public Car(int w,float g,int p) : base(w,g)
  {
                    wheels = w;
                    weight = g;
                    passengers = p;
        }
  public override void Speak(){
              Console.WriteLine("The car is speaking:Di-di!");
        }
}

class Truck:Vehicle      //定义卡车类
{
        int passengers;          //私有成员：乘客数
  float load;         //私有成员：载重量
        public Truck (int w,float g,int p，float l) : base(w,g)
  {
                    wheels = w;
                    weight = g;
```

```
            passengers = p;
            load = l;
        }
    public override void Speak(){
            Console.WriteLine("The truck is speaking:Ba-ba!");
        }
}
```

14.3.2 抽象方法

由于抽象类本身表达的是抽象的概念，因此类中的许多方法并不一定要有具体的实现，而只是留出一个接口来作为派生类重载的界面。举一个简单的例子，“图形”这个类是抽象的，它的成员方法“计算图形面积”也就没有实际的意义。面积只对“图形”的派生类比如“圆”、“三角形”这些非抽象的概念才有效，那么我们就可以把基类“图形”的成员方法“计算面积”声明为抽象的，具体的实现交给派生类通过重载来实现。

一个方法声明中如果加上 abstract 修饰符，我们称该方法为抽象方法（abstract method）。

如果一个方法被声明也是抽象的，那么该方法默认也是一个虚方法。事实上，抽象方法是一个新的虚方法，它不提供具体的方法实现代码。我们知道，非虚的派生类要求通过重载为继承的虚方法提供自己的实现，而抽象方法则不包含具体的实现内容，所以方法声明的执行体中只有一个分号“;”。

只能在抽象类中声明抽象方法。对抽象方法，不能再使用 static 或 virtual 修饰符，而且方法不能有任何可执行代码，哪怕只是一对大括号中间加一个一个分号“{;}”都不允许出现，只需要给出方法的原型就可以了。

“交通工具”的“鸣笛”这个方法实际上是没有什么意义的，接下来我们利用抽象方法的概念继续改写汽车类的例子：

程序清单 14-6:

```
using System;
abstract class Vehicle    //定义汽车类
{
        public      int wheels; //公有成员：轮子个数
        protected   float weight;             //保护成员：重量
        public      Vehicle(int w,float g){
                    wheels = w;
                    weight = g;
        }
    public abstract void Speak();
};
```

```
class Car:Vehicle  //定义轿车类
{
        int passengers;          //私有成员：乘客数
        public Car(int w,float g,int p) : base(w,g)
  {
                wheels = w;
                weight = g;
                passengers = p;
        }
  public override void Speak(){
            Console.WriteLine("The car is speaking:Di-di!");
        }
}

class Truck:Vehicle      //定义卡车类
{
        int passengers;          //私有成员：乘客数
  float load;        //私有成员：载重量
        public Truck (int w,float g,int p，float l) : base(w,g)
  {
                wheels = w;
                weight = g;
                passengers = p;
                load = l;
        }
  public override void Speak(){
            Console.WriteLine("The truck is speaking:Ba-ba!");
        }
}
```

还要注意，抽象方法在派生类中不能使用 base 关键字来进行访问。例如，下面的代码在编译时会发生错误：

```
class A
{
  public abstract void F();
}
class B: A
{
  public override void F() {
```

```
        base.F();                                   // 错误， base.F 是抽象方法
    }
}
```

我们还可以利用抽象方法来重载基类的虚方法，这时基类中虚方法的执行代码就被“拦截”了。下面的例子说明了这一点：

```
class A
{
  public virtual void F() {
        Console.WriteLine("A.F");
  }
}
abstract class B: A
{
  public abstract override void F();
}
class C: B
{
  public override void F() {
        Console.WriteLine("C.F");
  }
}
```

类 A 声明了一个虚方法 F，派生类 B 使用抽象方法重载了 F，这样 B 的派生类 C 就可以重载 F 并提供自己的实现。

14.3.3 密封类

想想看，如果所有的类都可以被继承，继承的滥用会带来什么后果？类的层次结构体系将变得十分庞大，类之间的关系杂乱无章，对类的理解和使用都会变得十分困难。有时候，我们并不希望自己编写的类被继承。另一些时候，有的类已经没有再被继承的必要。C#提出了一个密封类（sealed class）的概念，帮助开发人员来解决这一问题。

密封类在声明中使用 sealed 修饰符，这样就可以防止该类被其它类继承。如果试图将一个密封类作为其它类的基类，C#将提示出错。理所当然，密封类不能同时又是抽象类，因为抽象总是希望被继承的。

在哪些场合下使用密封类呢？密封类可以阻止其它程序员在无意中继承该类，而且密封类可以起到运行时优化的效果。实际上，密封类中不可能有派生类，如果密封类实例中存在虚成员函数，该成员函数可以转化为非虚的，函数修饰符 virtual 不再生效。

让我们看下面的例子：

```
abstract class A
{
  public abstract void F();
}

sealed class B: A
{
  public override void F() {
  // F 的具体实现代码
  }
}
```

如果我们尝试写下面的代码：

```
class C: B{  }
```

C#会指出这个错误，告诉你 B 是一个密封类，不能试图从 B 中派生任何类。

14.3.4 密封方法

我们已经知道，使用密封类可以防止对类的继承。C#还提出了密封方法（sealed method）的概念，以防止在方法所在类的派生类中对该方法的重载。

对方法可以使用 sealed 修饰符，这时我们称该方法是一个密封方法。

不是类的每个成员方法都可以作为密封方法，密封方法必须对基类的虚方法进行重载，提供具体的实现方法。所以，在方法的声明中，sealed 修饰符总是和 override 修饰符同时使用。请看例子代码。

程序清单 14-7:

```
using System;
class A
{
  public virtual void F() {
        Console.WriteLine("A.F");
  }
  public virtual void G() {
        Console.WriteLine("A.G");
  }
}
class B: A
{
  sealed override public void F() {
        Console.WriteLine("B.F");
  }
```

```
  override public void G() {
        Console.WriteLine("B.G");
  }
}
class C: B
{
  override public void G() {
        Console.WriteLine("C.G");
  }
}
```

类 B 对基类 A 中的两个虚方法均进行了重载，其中 F 方法使用了 sealed 修饰符，成为一个密封方法。G 方法不是密封方法，所以在 B 的派生类 C 中，可以重载方法 G，但不能重载方法 F。

14.4 继承中关于属性的一些问题

和类的成员方法一样，我们也可以定义属性的重载、虚属性、抽象属性以及密封属性的概念。

与类和方法一样，属性的修饰也应符合下列规则：

属性的重载

- 在派生类中使用修饰符的属性，表示对基类中的同名属性进行重载。
- 在重载的声明中，属性的名称、类型、访问修饰符都应该与基类中被继承的属性一致。
- 如果基类的属性只有一个属性访问器，重载后的属性也应只有一个。但如果基类的属性同时包含 get 和 set 属性访问器，重载后的属性可以只有一个，也可以同时有两个属性访问器。

注意：与方法重载不同的是，属性的重载声明实际上并没有声明新的属性，而只是为已有的虚属性提供访问器的具体实现。

虚属性

- 使用 virtual 修饰符声明的属性为虚属性。
- 虚属性的访问器，包括 get 访问器和 set 访问器，同样也是虚的。

抽象属性

- 使用 abstract 修饰符声明的属性为抽象属性。
- 抽象属性的访问器也是虚的，而且没有提供访问器的具体实现。这就要求在非虚的派生类中，由派生类自己通过重载属性来提供对访问器的具体实现。

- abstract 和 override 修饰符的同时使用，不但表示属性是抽象的，而且它重载了基类中的虚属性。这时属性的访问器也是抽象的。
- 抽象属性只允许在抽象类中声明。
- 除了同时使用 abstract 和 override 修饰符这种情况之外，static, virtual, override 和 abstract 修饰符中任意两个不能再同时出现。

密封属性

- 使用 sealed 修饰符声明的属性为密封属性。类的密封属性不允许在派生类中被继承。密封属性的访问器同样也是密封的。
- 属性声明时如果有 sealed 修饰符，同时也必须要有 override 修饰符。

从上面可以看出，属性的这些规则与方法十分类似。对于属性的访问器，我们可以把 get 访问器看成是一个与属性修饰符相同、没有参数、返回值为属性的值类型的方法；把 set 访问器看成是一个与属性修饰符相同、仅含有一个 value 参数、返回类型为 void 的方法。还记得第十章中客户住宿的例子吗？还是让我们扩展这个例子来说明属性在继承中的一些问题。

程序清单 14-8:

```
using System;
public enum sex
{
  woman,
  man,
};
abstract public class People
{
  private string s_name;
  public virtual string Name{
        get {
              return s_name;
        }
  }
  private sex m_sex;
  public virtual sex Sex{
        get {
              return m_sex;
        }
  protected string s_card;
  public abstract string Card
```

```
    {
        get; set;
    }
}
```

上面的例子中声明了“人”这个类，人的姓名 Name 和性别 Sex 是两个只读的虚属性；身份证号 Card 是一个抽象属性，允许读写。因为类 People 中包含了抽象属性 Card，所以 People 必须声明是抽象的。下面我们为住宿的客人编写一个类，类从 People 中继承。

程序清单 14-9:

```
class Customer: People
{
  string s_no;
  int i_day;
    public string No{
            get {
                return s_no;
            }
            set {
                if (s_no != value) {
                    s_no = value;
                }
                }
            }
    public int Day{
        get {
            return i_day;
        }
        set {
            if (i_day != value) {
                i_day = value;
                }
        }
    }
        public override string Name {
        get { return base.Name; }
    }
        public override sex Sex {
        get { return base.Sex }
    }
```

```
        public override string Card {
        get {
            return s_ card;
        }
            set {
            s_ card = value;
        }
    }
}
```

在类 Customer 中，属性 Name、Sex 和 Card 的声明都加上了 override 修饰符，属性的声明都与基类 People 中保持一致。Name 和 Sex 的 get 访问器、Card 的 get 和 set 访问器都使用了 base 关键字来访问基类 People 中的访问器。属性 Card 的声明重载了基类 People 中的抽象访问器。这样，在 Customer 类中没有抽象成员的存在，Customer 可以是非虚的。

14.5 小　　结

继承是面向对象系统中一个非常重要的概念。C#语言为我们提供了一整套设计良好的继承机制，包括：

- 派生类对基类的继承。
- 方法的继承。
- 属性及其访问器的继承。

在 C#中还提供了抽象和密封的概念，给继承方式带来了高度的灵活性，大大方便了开发人员设计自己的类的层次结构体系。包含了抽象方法或抽象属性的类必须是抽象类，抽象类的这些成员交给派生类去实现。密封类不允许被继承，密封方法和密封属性不允许被重载。抽象和密封的概念是本章的难点，希望读者认真掌握。

复习题

（1）继承是否破坏了对象的封装性。

（2）举一个例子，说明派生类对基类成员方法的覆盖和重载有什么区别。

（3）举例说明 base 保留字的用法。在哪些情况下不允许使用 base 保留字？

（4）多态性的含义是什么？C#为我们提供了哪两种多态性？说明它们之间有什么区别。

（5）用虚方法实现面向对象系统的多态性，会给我们带来什么好处？

（6）C#中为什么要提出密封类的概念？

（7）找出下列代码中的错误：

```
abstract class A
{
  int y;
  public virtual int X {
        get { return 0; }
  }
  public virtual int Y {
        get { return y; }
        set { y = value; }
  }
  public abstract int Z { get; set; }
}
class B: A
{
  int z;
  public override int X {
        get { return base.X + 1; }
  }
  public override int Y {
        set { base.Y = value < 0? 0: value; }
  }
  public abstract override int Z {
        get { return z; }
        set { z = value; }
  }
}
```

（8）设计一个信用卡通用付帐系统。系统可以使用三个银行的信用卡，其中两个是跨地区的银行，一个是本地银行。跨地区的银行提供的信用卡又分为三种：本地卡、外地卡、通存通兑卡。系统不处理外地卡付帐。实现的功能有：付帐、查询、转帐、取款。利用抽象类和虚方法的概念实现本系统。

第四部分　深入了解 C#

第十五章　接　　口

Windows2000 的推出是许多人盼望已久的事情，它带来的多种新特性令人为之兴奋不已。对于一名程序设计人员来说，最关注的一些问题有：在 Windows2000 操作系统中将组件对象模型（COM）与 Microsoft 事务服务器（MTS）合二为一，命名为 COM+；全新的应用程序编程接口（Application Programming Interface）特性，等等。

那么，这一切对于软件开发人员来说意味着什么呢？如何能够在新的视窗系统下高效地编写可靠的桌面应用程序和分布式应用程序？本章将向读者介绍有关的问题。

首先，本章讲解了组件化程序设计的基本概念，随后详细地论述了如何从组件编程的角度，利用 C#定义和实现接口，为我们设计组件级的应用程序。

接口是一种新的、基于组件的编程概念。如果读者有过一些 COM 方面的基础知识，对阅读本章将有一定的帮助。

15.1　组件编程技术

从软件业的发展历程来看，程序设计方法经历了多次变革。每当一种程序设计方法不能适应应用软件发展的需要时，人们就会努力寻找一种新的方法来解决这种“软件危机”。 组件化程序设计就是程序设计的一种新的变革，它结合了对象技术和组件技术两种特性，更为适合现代企业级应用程序的开发需要。这一节我们将向读者简要地介绍组件和分布式应用程序设计的基础知识。如果您希望了解更多组件化程序设计的知识，请参考这方面论述的专著。

15.1.1　应用程序的体系结构

一个应用程序的体系结构是应用程序结构的一种概念性描述。

当前，随着信息技术的飞速发展，现代企业中大多采用了分布式计算机系统。日益激烈的竞争要求应用程序尽量缩短开发周期，并且具有高度的灵活性，以适应变化多端的市场需要。这一切都对分布式应用程序的开发人员在业务方面和技术方面提出了挑战。

传统的分布式应用程序大多是两层的客户机/服务器模式（Client/Server），客户机直接连接到服务器上，在客户机上负责处理数据和执行客户端应用程序。

这种两层的应用程序体系结构存在着许多限制：客户连接的开销、服务器数据格式的限制、可扩展性等。尤其是当客户的数目未知，或者客户数目可能非常庞大时，两层的应用程序体系结构将无法处理这种情况。

为了提高分布式应用程序的灵活性和可重用性，可以在两层的体系结构中再加入第三层，这就是三层式应用程序体系结构：

表示层向用户提供数据，展现用户接口。

商业层用以实施商业逻辑，表示层使用商业层提供的服务。

数据访问层执行具体的数据访问服务，包括检索和存储。

在三层式应用程序体系结构中，商业层不固定地连接到任何客户，也不关心数据的存储方式。修改任何一层，都不会对其它层产生不良影响。每种服务都是独立的，并且可用新的方式进行组合，创建新的应用程序。这种三层结构方便开发人员创建高伸缩性的应用程序。

注意：三层式应用程序并不意味着三台独立的计算机。三层体系结构是一个逻辑模型，具体采用哪种物理模型依赖于提供服务的位置。

15.1.2 组件

组件的概念和特点

组件的英文名为“component”，也称为元件。实际上组件并不是一种新概念，它在许多成熟的工程领域有着十分广泛的应用。比如我们组装计算机，自己并不一定要了解 CPU、主板、光驱等配件的工作原理，而只需要知道如何将这些配件组装在一起。

软件行业的组件系统比其它许多行业发展的都要慢。在计算机软件发展的早期，一个应用系统往往是一个单独的应用程序。随着人们对软硬件需求的不断增加，应用更加复杂，程序更加庞大，系统开发的难度也越来越大。

从软件模型的角度考虑，人们希望把庞大的应用程序分割成为多个模块，每个模块完成独立的功能，模块之间协同工作。这样的模块我们称为组件。这些组件可以进行单独开发、单独编译、单独测试；把所有的组件组合在一起就得到了完整的系统。许多人都认为，未来的应用程序都将利用组件实现。

组件化的软件结构为我们带来了极大的好处。但是为了能够通过组装现有的组件来创建应用程序系统，我们必须解决几个技术上的关键问题：

- 采用一个标准方式来规范组件的定位和使用，这样将大大减少在人员培训上的开销，提高了组件的通用性。
- 提供与对象进行交互操作的标准方式。组件和对象所处的具体位置不应该影响程序员的开发方式，也不妨碍它们之间的交互操作，即我们所说的“位置透明性”。
- 要便于创建组件的版本。对软件的升级应该具有灵活性，组件的更新不会对现有的应用程序的运行造成不良影响。

- 提供满足用户需要的安全性。

接口

了解了组件的基本含义后，我们还必须进一步理解接口（interface）的含义。接口描述了组件对外提供的服务。在组件和组件之间、组件和客户之间都通过接口进行交互。因此组件一旦发布，它只能通过预先定义的接口来提供合理的、一致的服务。这种接口定义之间的稳定性使客户应用开发者能够构造出坚固的应用。一个组件可以实现多个组件接口，而一个特定的组件接口也可以被多个组件来实现。

组件接口必须是能够自我描述的。这意味着组件接口应该不依赖于具体的实现，将实现和接口分离彻底消除了接口的使用者和接口的实现者之间的耦合关系，增强了信息的封装程度。同时这也要求组件接口必须使用一种与组件实现无关的语言。目前组件接口的描述标准是 IDL 语言。

由于接口是组件之间的协议，因此组件的接口一旦被发布，组件生产者就应该尽可能地保持接口不变，任何对接口语法或语义上的改变，都有可能造成现有组件与客户之间的联系遭到破坏。

每个组件都是自主的，有其独特的功能，只能通过接口与外界通信。当一个组件需要提供新的服务时，可以通过增加新的接口来实现，不会影响原接口已存在的客户。而新的客户可以重新选择新的接口来获得服务。

15.1.3 组件化程序设计

组件化程序设计方法继承并发展了面向对象的程序设计方法。它把对象技术应用于系统设计，对面向对象的程序设计的实现过程作了进一步的抽象。我们可以把组件化程序设计方法用作构造系统的体系结构层次的方法，并且可以使用面向对象的方法很方便地实现组件。

组件化程序设计强调真正的软件可重用性和高度的互操作性。它侧重于组件的产生和装配，这两方面一起构成了组件化程序设计的核心。组件的产生过程不仅仅是应用系统的需求，组件市场本身也推动了组件的发展，促进了软件厂商的交流与合作。组件的装配使得软件产品可以采用类似于搭积木的方法快速地建立起来，不仅可以缩短软件产品的开发周期，同时也提高了系统的稳定性和可靠性。

组件程序设计的方法有以下几个方面的特点：

- 编程语言和开发环境的独立性。
- 组件位置的透明性。
- 组件的进程透明性。
- 可扩充性。
- 可重用性。
- 具有强有力的基础设施。
- 系统一级的公共服务。

C#语言由于其许多优点，十分适用于组件编程。但这并不是说 C#是一门组件编程

语言，也不是说 C#提供了组件编程的工具。我们已经多次指出，组件应该具有与编程语言无关的特性。请读者记住这一点：组件模型是一种规范，不管采用何种程序语言设计组件，都必须遵守这一规范。比如组装计算机的例子，只要各个厂商为我们提供的配件规格、接口符合统一的标准，这些配件组合起来就能协同工作。组件编程也是一样。我们只是说，利用 C#语言进行组件编程将会给我们带来更大的方便。

15.2 接 口 定 义

从技术上讲，接口是一组包含了函数型方法的数据结构。通过这组数据结构，客户代码可以调用组件对象的功能。

15.2.1 声明

接口声明实际上就是一种定义新的接口的类型声明，声明的格式如下：

attributes *interface-modifiers* `interface` *identifier* *interfacebase* *interface-body* `;`

接口仅可使用下列修饰符：

- new
- public
- protected
- internal
- private

在一个接口定义中同一修饰符不允许出现多次，new 修饰符只能出现在嵌套接口中，表示覆盖了继承而来的同名成员。

The public, protected, internal, and private 修饰符定义了对接口的访问权限。

在接口的声明体中，可以定义接口的成员。接口的成员可以是方法、属性、索引指示器和事件。

下面的例子定义了一个名为 IControl 的接口，接口中包含一个成员方法 Paint：

```
interface IControl
{
  void Paint();
}
```

15.2.2 接口的继承

接口具有不变性，但这并不意味着接口不再发展。类似于类的继承性，接口也可以继承和发展。

注意：接口继承和类继承不同。首先，类继承不仅是说明继承，而且也是实现继

承；而接口继承只是说明继承。也就是说，派生类可以继承基类的方法实现，而派生的接口只继承了父接口的成员方法说明，而没有继承父接口的实现。其次，C#中类继承只允许单继承，但是接口继承允许多继承，一个子接口可以有多个父接口。

接口可以从零或多个接口中继承。从多个接口中继承时，用“:”后跟被继承的接口名字，多个接口名之间用“,”分割。被继承的接口应该是可以访问得到的，比如从 private 类型或 internal 类型的接口中继承就是不允许的。接口不允许直接或间接地从自身继承。

和类的继承相似，接口的继承也形成接口之间的层次结构。

请看下面的例子。

程序清单 15-1：

```
using System;
interface IControl
{
  void Paint();
}
   interface ITextBox: IControl
{
  void SetText(string text);
}
   interface IListBox: IControl
{
  void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {}
```

对一个接口的继承也就继承了接口的所有成员，上面的例子中，接口 ITextBox 和 IListBox 都从接口 IControl 中继承，也就继承了接口 IControl 的 Paint 方法。接口 IComboBox 从接口 ITextBox 和 IListBox 中继承，因此它应该继承了接口 ITextBox 的 SetText 方法和 IListBox 的 SetItems 方法，还有 IControl 的 Paint 方法。

15.3 接口的成员

15.3.1 接口成员的定义

接口可以包含一个和多个成员，这些成员可以是方法、属性、索引指示器和事件，但不能是常量、域、操作符、构造函数或析构函数，而且不能包含任何静态成员。下面例子中接口 IExample 包含了索引指示器、事件 E、方法 F、属性 P 这些成员：

```
interface IExample
{
  string this[int index] { get; set; }
        event EventHandler E;
        void F(int value);
        string P { get; set; }
}
public delegate void EventHandler(object sender, EventArgs e);
```

接口成员默认访问方式是 public。接口成员声明不能包含任何修饰符，比如成员声明前不能加 abstract, public, protected, internal, private, virtual, override 或 static 修饰符。

接口的成员之间不能相互同名。继承而来的成员不用再声明，但接口可以定义与继承而来的成员同名的成员，这时我们说接口成员覆盖了继承而来的成员，这不会导致错误，但编译器会给出一个警告。关闭警告提示的方式是在成员声明前加上一个 new 关键字。但如果没有覆盖父接口中的成员，使用 new 关键字会导致编译器发出警告。

15.3.2 对接口成员的访问

对接口方法的调用和采用索引指示器访问的规则与类中的情况也是相同的。如果底层成员的命名与继承而来的高层成员一致，那么底层成员将覆盖同名的高层成员。但由于接口支持多继承，在多继承中，如果两个父接口含有同名的成员，这就产生了二义性（这也正是 C#中取消了类的多继承机制的原因之一），这时需要进行显式的声明。

程序清单 15-2：

```
using System;
interface ISequence
{
  int Count { get; set; }
}
   interface IRing
{
  void Count(int i);
}
 interface IRingSequence: ISequence, IRing {}
class C
{
  void Test(IRingSequence rs) {
        //rs.Count(1);              错误, Count 有二义性
        //rs.Count = 1;    错误, Count 有二义性
        ((ISequence)rs).Count = 1;          // 正确                    ((IRing)rs).Count(1);          // 正
```

```
确调用 IRing.Count
  }
}
```

上面的例子中，前两条语句 x.Count(1)和 x.Count = 1 会产生二义性，从而导致编译时错误，因此必须显式地给 X 指派父接口类型，这种指派在运行时不会带来额外的开销。

再看下面的例子：

程序清单 15-3：

```
using System;
interface IInteger
{
  void Add(int i);
}
   interface IDouble
{
  void Add(double d);
}
interface INumber: IInteger, IDouble {}
class C
{
  void Test(INumber n) {
        //n.Add(1); 错误
        n.Add(1.0);                    // 正确
        ((IInteger)n).Add(1);   // 正确
        ((IDouble)n).Add(1);          // 正确
  }
}
```

调用 n.Add(1) 会导致二义性，因为候选的重载方法的参数类型均适用。但是，调用 n.Add(1.0) 是允许的，因为 1.0 是浮点数，参数类型与方法 IInteger.Add（）的参数类型不一致，这时只有 IDouble.Add 才是适用的。不过只要加入了显式的指派，就决不会产生二义性。

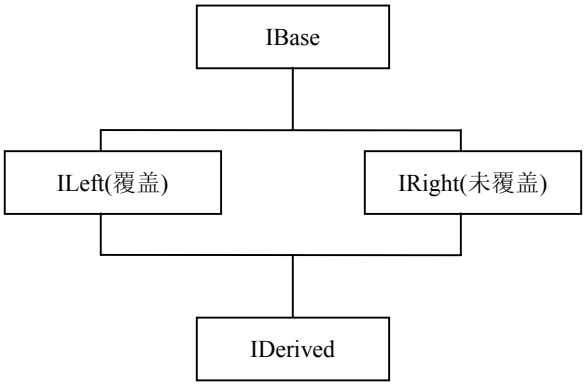
接口的多重继承的问题也会带来成员访问上的问题。

```
interface IBase
{
  void F(int i);
}
```

```
   interface ILeft: IBase
{
  new void F(int i);
}
   interface IRight: IBase
{
  void G();
}
interface IDerived: ILeft, IRight {}
class A
{
  void Test(IDerived d) {
        d.F(1);                          // 调用 ILeft.F
        ((IBase)d).F(1);        // 调用 IBase.F
        ((ILeft)d).F(1);        // 调用 ILeft.F
        ((IRight)d).F(1);       // 调用 IBase.F
  }
}
```

上例中，方法 IBase.F 在派生的接口 ILeft 中被 Ileft 的成员方法 F 覆盖了。所以对 d.F(1) 的调用实际上调用了。虽然从 IBase——IRight——IDerived 这条继承路径上来看， ILeft.F 方法是没有被覆盖的。

我们只要记住这一点：一旦成员被覆盖以后，所有对其的访问都被覆盖以后的成员“拦截”了。



接口多继承中底层成员对高层成员的覆盖

15.3.3 接口成员的全权名

使用接口成员也可采用全权名（fully qualified name）。接口的全权名称是这样构成的：接口名加小圆点“.”再跟成员名。比如对于下面两个接口：

```
interface IControl
{
  void Paint();
}
   interface ITextBox: IControl
{
  void SetText(string text);
}
```

其中 Paint 的全权名是 IControl.Paint ，SetText 的全权名是 ITextBox.SetText。当然，全权名中的成员名称必须是在接口中已经声明过的，比如使用 ITextBox.Paint.就是不合理的。

如果接口是名字空间的成员，全权名还必须包含名字空间的名称。

```
namespace System
{
  public interface ICloneable
  {
        object Clone();
  }
}
```

那么 Clone 方法的全权名是 System.ICloneable.Clone。

15.4 接口的实现

15.4.1 类对接口的实现

前面我们已经说过，接口定义不包括方法的实现部分。接口可以通过类或结构来实现。我们主要讲述通过类来实现接口。用类来实现接口时，接口的名称必须包含在类声明中的基类列表中。

下面的例子给出了由类来实现接口的例子。其中 ISequence 为一个队列接口，提供了向队列尾部添加对象的成员方法 Add（），IRing 为一个循环表接口，提供了向环中插入对象的方法 Insert(object obj)，方法返回插入的位置。类 RingSquence 实现了接口 ISequence 和接口 IRing。

程序清单 15-4：

```
using System;
```

```
interface ISequence
{
  object Add();
}
   interface IRing
{
  int Insert(object obj);
}
   class RingSequence: ISequence, IRing
{
  public object Add() {…}
        public int Insert(object obj) {…}
}
```

如果类实现了某个接口，类也隐式地继承了该接口的所有父接口，不管这些父接口有没有在类声明的基类表中列出。

```
using System;
interface IControl
{
  void Paint();
}
   interface ITextBox: IControl
{
  void SetText(string text);
}
   class TextBox: ITextBox
{
  public void Paint() {…}
  public void SetText(string text) {…}
}
```

这里，类 TextBox 不仅实现了接口 ITextBox，还实现了接口 ITextBox 的父接口 IControl。

前面我们已经看到，一个类可以实现多个接口。再看下面的例子：

```
using System;
interface IControl
{
  void Paint();
}
```

```
interface IDataBound
{
  void Bind(Binder b);
}
public class Control: IControl
{
  public void Paint() {…}
}
public class EditBox: Control, IControl, IDataBound
{
  public void Paint() {…}
  public void Bind(Binder b) {…}
}
```

上例中，类 EditBox 从 Control 类继承并同时实现了 IControl and IDataBound 接口。EditBox 中的 Paint 方法来自 IControl 接口，Bind 方法来自 IDataBound 接口，二者在 EditBox 类中都作为公有成员实现。 当然，在 C# 中我们也可以选择不作为公有成员实现接口。

如果每个成员都明显地指出了被实现的接口，通过这种途径被实现的接口我们称之为显式接口成员（explicit interface member）。用这种方式我们改写上面的例子：

```
public class EditBox: IControl, IDataBound
{
  void IControl.Paint() {…}
  void IDataBound.Bind(Binder b) {…}
}
```

显式接口成员只能通过接口调用。例如：

```
class Test
{
  static void Main() {
        EditBox editbox = new EditBox();
        editbox.Paint();   // error: no such method
        IControl control = editbox;
        control.Paint();   // calls EditBox's Paint implementation
  }
}
```

上述代码中对 editbox.Paint()的调用是错误的，因为 editbox 本身并没有提供这一方法。control.Paint()是正确的调用方式。

注意：接口本身不提供所定义的成员的实现，它仅仅说明这些成员，这些成员必须依靠实现接口的类或其它接口的支持。

15.4.2 显式接口成员执行体

为了实现接口，类可以声明显式接口成员执行体（Explicit interface member implementations）。显式接口成员执行体可以是一个方法、一个属性、一个事件或者是一个索引指示器的声明，声明与该成员对应的全权名应保持一致。

```
using System;
interface ICloneable
{
  object Clone();
}
   interface IComparable
{
  int CompareTo(object other);
}
   class ListEntry: ICloneable, IComparable
{
  object ICloneable.Clone() {…}
  int IComparable.CompareTo(object other) {…}
}
```

上面的代码中 ICloneable.Clone 和 IComparable.CompareTo 就是显式接口成员执行体。

不能在方法调用、属性访问以及索引指示器访问中通过全权名访问显式接口成员执行体。事实上，显式接口成员执行体只能通过接口的实例，仅仅引用接口的成员名称来访问。

显式接口成员执行体不能使用任何访问限制符，也不能加上 abstract, virtual, override 或 static 修饰符。

显式接口成员执行体和其他成员有着不同的访问方式。因为不能在方法调用、属性访问以及索引指示器访问中通过全权名访问，显式接口成员执行体在某种意义上是私有的。但它们又可以通过接口的实例访问，也具有一定的公有性质。使用显式接口成员执行体通常有两个目的：

- 因为显式接口成员执行体不能通过类的实例进行访问，这就可以从公有接口中把接口的实现部分单独分离开。如果一个类只在内部使用该接口，而类的使用者不会直接使用到该接口，这种显式接口成员执行体就可以起到作用。
- 显式接口成员执行体避免了接口成员之间因为同名而发生混淆。如果一个类希望对名称和返回类型相同的接口成员采用不同的实现方式，这就必须要使用到显式接口成员执行体。如果没有显式接口成员执行体，那么对于名称和返回类型不同的接口成员，类也无法进行实现。

只有类在声明时，把接口名写在了基类列表中，而且类中声明的全权名、类型和返回类型都与显式接口成员执行体完全一致时，显式接口成员执行体才是有效的。例

如：

```
class Shape: ICloneable
{
  object ICloneable.Clone() {…}
  int IComparable.CompareTo(object other) {…}
}
```

这是一个无效的声明，因为 Shape 声明时基类列表中没有出现接口 IComparable。下面的代码同样也有错误：

```
class Shape: ICloneable
{
  object ICloneable.Clone() {…}
}
   class Ellipse: Shape
{
  object ICloneable.Clone() {…}
}
```

在 Ellipse 中声明 ICloneable.Clone 是错误的，因为 Ellipse 即使隐式地实现了接口 ICloneable，ICloneable 仍然没有显式地出现在 Ellipse 声明的基类列表中。

接口成员的全权名必须对应在接口中声明的成员。如下面的例子中，Paint 的显式接口成员执行体必须写成 IControl.Paint。

```
using System;
interface IControl
{
  void Paint();
}
   interface ITextBox: IControl
{
  void SetText(string text);
}
   class TextBox: ITextBox
{
  void IControl.Paint() {…}
  void ITextBox.SetText(string text) {…}
}
```

15.4.3 接口映射

类必须为在基类表中列出的所有接口的成员提供具体的实现。在类中定位接口成员的实现称之为接口映射（***interface mapping***）。

映射，数学上表示一一对应的函数关系。接口映射的含义也是一样，接口通过类来实现，那么对于在接口中声明的每一个成员，都应该对应着类的一个成员来为它提供具体的实现。

类的成员及其所映射的接口成员之间必须满足下列条件：

- 如果 A 和 B 都是成员方法，那么 A 和 B 的名称、类型、形参表（包括参数个数和每一个参数的类型）都应该是一致的。
- 如果 A 和 B 都是属性，那么 A 和 B 的名称、类型应当一致，而且 A 和 B 的访问器也是类似的。但如果 A 不是显式接口成员执行体，A 允许增加自己的访问器。
- 如果 A 和 B 都是时间，那么 A 和 B 的名称、类型应当一致。
- 如果 A 和 B 都是索引指示器，那么 A 和 B 的类型、形参表（包括参数个数和每一个参数的类型）应当一致。而且 A 和 B 的访问器也是类似的。但如果 A 不是显式接口成员执行体，A 允许增加自己的访问器。

那么，对于一个接口成员，怎样确定由哪一个类的成员来实现呢？即一个接口成员映射的是哪一个类的成员？在这里我们叙述一下接口映射的过程。假设类 C 实现了一个接口 IInterface，Member 是接口 IInterface 中的一个成员。在定位由谁来实现接口成员 Member，即 Member 的映射过程是这样的：

（1）如果 C 中存在着一个显式接口成员执行体，该执行体与接口 IInterface 及其成员 Member 相对应，则由它来实现 Member 成员。

（2）如果条件（1）不满足，且 C 中存在着一个非静态的公有成员，该成员与接口成员 Member 相对应，则由它来实现 Member 成员。

（3）如果上述条件仍不满足，则在类 C 声明的基类列表中寻找一个 C 的基类 D，用 D 来代替 C。

（4）重复步骤（1）—（3），遍历 C 的所有直接基类和非直接基类，直到找到一个满足条件的类的成员。

（5）如果仍然没有找到，则报告错误。

下面是一个调用基类方法来实现接口成员的例子。类 Class2 实现了接口 Interface1，类 Class2 的基类 Class1 的成员也参与了接口的映射，也就是说，类 Class2 在对接口 Interface1 进行实现时，使用了类 Class1 提供的成员方法 F 来实现接口 Interface1 的成员方法 F：

```
interface Interface1
{
  void F();
}
   class Class1
{
  public void F() {}
  public void G() {}
}
```

```
 class Class2: Class1, Interface1
{
  new public void G() {}
}
```

注意：接口的成员包括它自己声明的成员，而且包括该接口所有父接口声明的成员。在接口映射时，不仅要对接口声明体中显式声明的所有成员进行映射，而且要对隐式地从父接口那里继承来的所有接口成员进行映射。

在进行接口映射时，还要注意下面两点：

- 在决定由类中的哪个成员来实现接口成员时，类中显式说明的接口成员比其它成员优先实现。
- 使用 Private、protected 和 static 修饰符的成员不能参与实现接口映射。

例如：

```
interface ICloneable
{
  object Clone();
}
 class C: ICloneable
{
  object ICloneable.Clone() {…}
  public object Clone() {…}
}
```

例子中成员 ICloneable.Clone 称为接口 ICloneable 的成员 Clone 的实现者，因为它是显式说明的接口成员，比其它成员有着更高的优先权。

如果一个类实现了两个或两个以上名字、类型和参数类型都相同的接口，那么类中的一个成员就可能实现所有这些接口成员：

```
interface IControl
{
  void Paint();
}
 interface IForm
{
  void Paint();
}
 class Page: IControl, IForm
{
  public void Paint() {…}
}
```

这里，接口 IControl 和 IForm 的方法 Paint 都映射到了类 Page 中的 Paint 方法。当然也可以分别用显式的接口成员分别实现这两个方法：

```
interface IControl
{
  void Paint();
}
   interface IForm
{
  void Paint();
}
   class Page: IControl, IForm
{
  public void IControl.Paint() {
        //具体的接口实现代码
  }
  public void IForm.Paint() {
        //具体的接口实现代码
  }
}
```

上面的两种写法都是正确的。但是如果接口成员在继承中覆盖了父接口的成员，那么对该接口成员的实现就可能必须映射到显式接口成员执行体。看下面的例子：

```
interface IBase
{
  int P { get; }
}
   interface IDerived: IBase
{
  new int P();
}
```

接口 IDerived 从接口 IBase 中继承，这时接口 IDerived 的成员方法覆盖了父接口的成员方法。因为这时存在着同名的两个接口成员，那么对这两个接口成员的实现如果不采用显式接口成员执行体，编译器将无法分辨接口映射。所以，如果某个类要实现接口 IDerived，在类中必须至少声明一个显式接口成员执行体。采用下面这些写法都是合理的：

```
//一、对两个接口成员都采用显式接口成员执行体来实现
class C: IDerived
{
  int IBase.P
       get {
         ……//具体的接口实现代码
         }
```

```
        }
          int IDerived.P(){
          ……//具体的接口实现代码
          }
    }

//二、对 Ibase 的接口成员采用显式接口成员执行体来实现
class C: IDerived
{
    int IBase.P
        get {
              ……//具体的接口实现代码
          }
        }
    public int P(){
          ……//具体的接口实现代码
          }
  }

  //三、对 IDerived 的接口成员采用显式接口成员执行体来实现
  class C: IDerived
  {
    public int P
          get {
              ……//具体的接口实现代码
          }
    }
    int IDerived.P(){
              ……//具体的接口实现代码
    }
  }
```

另一种情况是，如果一个类实现了多个接口，这些接口又拥有同一个父接口，这个父接口只允许被实现一次。

```
using System;
interface IControl
{
  void Paint();
}
```

```
        interface ITextBox: IControl
{
  void SetText(string text);
}
        interface IListBox: IControl
{
  void SetItems(string[] items);
}
        class ComboBox: IControl, ITextBox, IListBox
{
  void IControl.Paint() {…}
          void ITextBox.SetText(string text) {…}
          void IListBox.SetItems(string[] items) {…}
}
```

上面的例子中，类 ComboBox 实现了三个接口：IControl, ITextBox 和 IListBox。如果认为 ComboBox 不仅实现了 IControl 接口，而且在实现 ITextBox 和 IListBox 的同时，又分别实现了它们的父接口 IControl。实际上，对接口 ITextBox 和 IListBox 的实现，分享了对接口 IControl 的实现。

15.4.4 接口实现的继承机制

一个类继承了它的基类提供的所有接口的实现。

如果不显式地重新实现接口，派生类就无法改变从基类中继承来的接口映射。

```
  interface IControl
{
  void Paint();
}
   class Control: IControl
{
  public void Paint() {…}
}
   class TextBox: Control
{
  new public void Paint() {…}
}
```

上面的例子中，TextBox 中的 Paint 方法覆盖了 Control 中的 Paint 方法 ，但却没有改变 Control.Paint 对 IControl.Paint 的映射，并且在类的实例和接口的实例中对 Paint 方法的调用会产生下面这样的结果：

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();              // invokes Control.Paint();
t.Paint();              // invokes TextBox.Paint();
ic.Paint();             // invokes Control.Paint();
it.Paint();             // invokes Control.Paint();
```

但是，当一个 interface 方法被映射到类中的一个虚方法时，派生类就可以重载这个虚方法，并且改变这个接口的实现。我们改写一下上例中的代码：

```
interface IControl
{
  void Paint();
}
   class Control: IControl
{
  public virtual void Paint() {…}
}
   class TextBox: Control
{
  public override void Paint() {…}
}
```

上面代码的实际效果是：

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();              // 调用 Control.Paint();
t.Paint();              // 调用 TextBox.Paint();
ic.Paint();             // 调用 Control.Paint();
it.Paint();             // 调用 TextBox.Paint();
```

因为显式说明的接口成员不能被声明为虚的，因此无法重载显式说明的接口实现。这时最好采用显式说明的接口实现来调用另一个方法，这个被调用的方法可以被声明为虚的，允许被派生类重载。例：

```
interface IControl
{
  void Paint();
}
```

```
class Control: IControl
{
  void IControl.Paint() { PaintControl(); }
  protected virtual void PaintControl() {…}
}
class TextBox: Control
{
  protected override void PaintControl() {…}
}
```

这里，从 Control 中派生的类可以通过重载 PaintControl 方法来具体实现 IControl.Paint。

15.4.5 接口的重实现

我们已经介绍过，派生类可以对基类中已经定义的成员方法进行重载。类似的概念引入到类对接口的实现中来，叫做接口的重实现（re-implementation）。

继承了接口实现的类可以对接口进行重实现。这个接口要求是在类声明的基类列表中出现过的。对接口的重实现也必须严格地遵守首次实现接口的规则，派生的接口映射不会对为接口的重实现所建立的接口映射产生任何影响。下面的代码给出了接口重实现的例子：

```
interface IControl
{
  void Paint();
}
class Control: IControl
{
  void IControl.Paint() {…}
}
    class MyControl: Control, IControl
{
  public void Paint() {}
}
```

实际上就是：Control 把 IControl.Paint 映射到了 Control.IControl.Paint 上，但这并不影响在 MyControl 中的重实现。在 MyControl 中的重实现中，IControl.Paint 被映射到 MyControl.Paint 之上。

在接口的重实现时，继承而来的公有成员声明和继承而来的显式接口成员的声明参与到接口映射的过程。

```
using System;
```

```
interface IMethods
{
  void F();
  void G();
  void H();
  void I();
}
   class Base: IMethods
{
  void IMethods.F() {}
  void IMethods.G() {}
  public void H() {}
  public void I() {}
}
   class Derived: Base, IMethods
{
  public void F() {}
  void IMethods.H() {}
}
```

这里，接口 IMethods 在 Derived 中的实现把接口方法映射到了 Derived.F, Base.IMethods.G, Derived.IMethods.H, 还有 Base.I.

前面我们说过，类在实现一个接口时，同时隐式地实现了该接口的所有父接口。同样，类在重实现一个接口时，同时隐式地重实现了该接口的所有父接口。

```
using System;
interface IBase
{
  void F();
}
   interface IDerived: IBase
{
  void G();
}
   class C: IDerived
{
  void IBase.F() {
        //对 F 进行实现的代码…
  }
        void IDerived.G() {
        //对 G 进行实现的代码…
```

```
  }
}
   class D: C, IDerived
{
  public void F(){
        //对 F 进行实现的代码…
  }
  public void G() {
        //对 G 进行实现的代码…
  }
}
```

这里，对 IDerived 的重实现也同样实现了对 IBase 的重实现，把 IBase.F 映射到了 D.F。

15.5 抽象类与接口

和非抽象类一样，抽象类也必须提供在基类列表中出现的所有接口成员的实现。不同的是，抽象类允许将接口的方法映射到抽象的成员方法。

```
interface IMethods
{
  void F();
  void G();
}
   abstract class C: IMethods
{
  public abstract void F();
  public abstract void G();
}
```

上例中，所有 C 的非抽象的派生类必须重载 C 中的抽象方法来提供对接口的实现。

注意：显式说明的接口成员不能是抽象的，但它允许调用抽象的方法，如下例所示：

```
interface IMethods
{
  void F();
  void G();
}
```

```
  abstract class C: IMethods
{
  void IMethods.F() { FF(); }
 void IMethods.G() { GG(); }
 protected abstract void FF();
 protected abstract void GG();
}
```

15.6 小　　结

组件化程序设计方法继承并发展了面向对象的程序设计方法。它把对象技术应用于系统设计，对面向对象的程序设计的实现过程作了进一步的抽象。我们可以把组件化程序设计方法用作构造系统的体系结构层次的方法，并且可以使用面向对象的方法很方便地实现组件。

接口是组件之间的协议，描述了组件对外提供的服务。从技术上讲，接口是一组包含了函数型方法的数据结构。通过这组数据结构，客户代码可以调用组件对象的功能。

接口可以从父接口中继承。接口的继承首先是说明性继承，不是实现性继承，它的实现需要通过类或结构来实现；其次接口继承可以是多继承。

接口包含的成员有方法、属性、索引指示器和事件。由于接口允许多继承，在可能发生二义性的地方可以采用全权名来避免。

可以用类来实现接口。在类中定位接口成员的实现称之为接口映射。类必须为接口的所有成员提供具体的实现，包括接口中显式声明的成员，以及接口从父接口中继承而来的成员。同样，在对接口的实现过程中可以采用显式接口成员执行体来避免产生二义性。

派生类可以对基类已经实现的接口进行重实现。

抽象类也可以实现接口，但接口成员必须映射到抽象类的抽象成员。抽象类的派生类如果是非抽象类，则必须通过方法重载来实现接口成员。

复习题

（1）简述组件化的思想，以及它为软件的开发带来了哪些好处。

（2）说明组件化程序设计与面向对象的程序设计有什么区别和联系。

（3）接口在组件中起到什么样的作用？

（4）试从实现接口成员的角度说说接口的成员为什么不可以是操作符。

（5）试说明对接口成员的访问方式与对类的成员的访问方式有什么不同之处。

（6）说明显式接口成员执行体与类中的其它成员之间有什么区别。在哪种情况下需要使用显式接口成员执行体？

（7）接口的多继承给我们带来了哪些问题，我们应如何解决这些问题？

（8）试论述接口映射的过程。

（9）假设一个类中采用了显式接口成员执行体实现了某个接口成员，可否在它的派生类类中重载这个成员？

（10）下面的代码是一个字典程序的例子，试分析接口 IDictionary 的各个成员分别映射到哪个类的哪个方法。

```
interface Idictonary
{
  void LoadLibrary(string filename);
  void FreeLibrary(string filename);
  void RestoreLibrary(string filename);
  bool InsertWord(string word);
  bool LookupWord(string word,string resultword);
  void DeleteWord(string word);
}

class Dict
{
  public void LoadLibrary(string filename){……};
  public void FreeLibrary(string filename) {……};
  public bool InsertWord(string word) {……};
  public string LookupWord(string word) {……};
  public void DeleteWord(string word) {……};
}

class NewDict : Dict, IDictionary
{
  void LoadLibrary(string filename) {……};
  void FreeLibrary(string filename) {……};
  public void RestoreLibrary(string filename) {……};
  public void InsertWord(string word) {……};
  public bool LookupWord(string word,string resultword) {……};
  public void DeleteWord(string word) {……};
}
```

第十六章　组织应用程序

在传统的 Windows 应用程序中，动态链接库（Dynamic-Link Library，DLL）是一个非常重要的组成部分。在建立应用程序的可执行文件时，不需要将 DLL 链接到程序中，而是在程序运行时动态装载 DLL。除了很小的程序以外，实际工作的一个应用程序通常都由若干的编译单元共同组成。出于方便，我们常常将大型的程序分为若干个相互联系的可执行程序和动态链接库。

有经验的程序员一定深有体会，使用现有的各种语言开发工具来编写动态链接库是一件非常困难，同时又是非常需要耐心的事情。然而现在你将会发现，用 C#来编写动态链接库和写简单的可执行程序几乎没什么两样——只要一些简单的标记，编译器将会为我们完成绝大多数的工作

C# 程序是通过使用名字空间来组织的。名字空间即可以作为应用程序的内部结构体系 ，也可以作为应用程序的外部结构体系。在作为外部结构体系时，程序中的一些元素可以被导出到其它的程序。使用指示符导入名字空间有助于使用方便。

16.1　基 本 概 念

16.1.1　动态链接库

你是否有过这样的经历？在配置不是很好的机器上运行一些不是很完善的 Windows 应用程序，系统速度越来越慢，直至弹出一个警告对话框：“系统资源严重不足！请关闭部分 Windows 应用程序……”。这往往是静态链接的缘故。使用普通的函数库，在程序链接时将库中的代码拷贝到可执行文件中，这叫作静态链接。假设有多个程序同时执行，并且都调用了同一个动态链接库，这时内存中就会保留许多重复的代码副本。

使用动态链接库则不一样。只有程序在执行时才将库代码装入内存。对于同一个动态链接库，无论有多少个应用程序同时在使用它，内存中都只有一个动态链接库的副本。如果动态链接库不再被任何程序使用，系统就将它调出内存，这就减少了应用程序对内存的要求。

动态链接库是一种程序模块，它不仅可以包含可执行代码，而且通常还包含了各种类型的预定义的数据和资源，扩大了库文件的使用范围。Windows 操作系统使用了许多动态链接库。比如我们使用 Visual C++建立 MFC 应用程序时，如果在 AppWizard 向导中选择了使用 MFC 作为动态链接库，那么所有这种类型的程序至少都共用了“C:\Windows\System”目录下的 MFC 动态链接库文件 Msvcrt.dll 和 Mfc42.dll。许多设

备的驱动程序也是用动态链接库实现的，扩展名一般为.drv。

动态链接库技术常常用于开发大型软件系统。一个大型系统如果只通过一个可执行文件来完成，那程序就太庞大了，而且可能有许多重复的功能。这时如果将程序分解，由一系列主程序和动态链接库组成，这就减少了开发难度和工作量，提高了访问的速度，更有利于对整个系统的管理。

动态链接库的另一个用途是软件产品国际化。开发人员可以将依赖于各国语言的资源分离开来，各自放进专门的动态链接库中。各国不同的用户可以在安装和运行时，选择适当语言版本的动态链接库，这样主程序不用改变就可以用于全球范围。这是实现软件国际化的一项技术。

知道了动态链接库的这些特点，相信读者不难明白为什么动态链接库的应用那么广泛了。

16.1.2 编译单元

我们再首先介绍一下编译单元（Compilation units）的概念。顾名思义，编译单元是能够被编译器进行编译的最小单位。编译单元定义了源文件的整体结构。一个编译单元的声明格式如下：

using-directives attributes namespace-member-declarations

一个 C#程序包含了一个或多个编译单元，每一个编译单元包含在一个独立的源文件中。当 C#程序被编译时，编译器对程序的所有编译单元进行统一处理。编译单元也可能互相依赖。

编译单元的使用指示符只对本单元的属性和名字空间成员声明产生影响，但不会影响到其它的编译单元。

16.1.3 名字空间和装配

到现在为止，除了依赖于一些系统提供的类（比如一开始我们就提到的 System.Console 类），我们介绍的程序主要都是依靠自身来实现的。但更普遍的情况是，现实世界中的应用程序可能包含许多不同的部分。举个例子，一个应用程序系统可能需要依赖于许多不同的组件，一些是内部开发的，另一些是从别的软件开发商处购买的。

使用名字空间和装配使得这种基于组件的系统成为可能。名字空间提供了一个逻辑上的层次结构体系，它即可以作为应用程序的内部结构体系 ，也可以作为应用程序的外部结构体系。在作为外部结构体系时，程序中的一些元素可以被导出到其它的程序。

装配用于应用程序的打包（packaging）和部署（deployment）。装配和模块所扮演的角色类似，都可以作为类型的物理容器。一个装配可能包含若干个相互独立的模块，可以包含许多类型、作用于这些类型上的可执行代码，以及对其它装配的引用。

装配有两种类型：应用程序和库。应用程序都有一个入口，通常扩展名为“.exe” 。

库则不包含入口，通常扩展名为“.dll”。

16.2 使用名字空间

16.2.1 名字空间的声明

名字空间的声明是关键字“*namespace*”后跟名字空间名和名字空间主体，然后还可以跟一个分号。格式如下：

namespace qualified-identifier namespace-body *;*

名字空间的声明要么在编译单元（源文件）的第一行出现，要么作为成员出现在其它名字空间的声明之中。首次声明的名字空间是全局名字空间的成员，在别的名字空间内部声明的名字空间作为外部名字空间的成员。在这两种情况下，名字空间都不允许出现重名。

名字空间隐式地使用 public 修饰符，在声明时不允许使用任何访问修饰符。

下面这种形式可以采用非嵌套的语法来实现名字空间的嵌套声明：

```
namespace N1.N2
{
  class A {}
  class B {}
}
```

上述代码等价于：

```
namespace N1
{
  namespace N2
  {
        class A {}
        class B {}
  }
}
```

16.2.2 成员与类型声明

名字空间的成员可以是一个类型（类、结构、接口、枚举或代表），也可以是另一个名字空间。一个编译单元或名字空间主体中可以包含多个成员声明，这些声明给编译单元或名字空间主体中添加了新的成员。

名字空间中的类型声明可以是类的声明、结构的声明、接口的声明、枚举的声明，

或者是代表的声明。

一个类型声明可以在编译单元的第一行作为顶层声明出现，也可以出现在编译单元的内部，作为成员声明。类型声明还可以是在名字空间、类或结构的内部作为成员声明出现。

对类型声明的访问权限与类型声明所处的位置有关：

- 对在编译单元中一开始就声明的类型，访问权限可以是公有（public）的，也可以是内部（internal）的。默认的访问权限是内部的。
- 在类中声明的类型，访问权限可以是公有（public）的，保护（protected）的，内部（internal）的，或是私有（private）的。默认的访问方式是私有的。

在结构中声明的类型，访问权限可以是公有（public）的，内部（internal）的，或是私有（private）的。默认的访问方式是私有的。

16.2.3 改写“Welcome”程序

为了演示名字空间的基本用法，我们把第三章的“Welcome” 程序分为库和控制台两部分。库用来提供显示的消息，控制台程序执行显示。

首先让我们看一下库文件，其中包含了一个 WelcomeMessage 类。

程序清单 16-1：

```
// 库文件 WelcomeLibrary.cs
namespace MyProgram.CSharp.FirstApp
{
  public class WelcomeMessage
  {
        string m_message;
        public WelcomeMessage()
        {
              m_message = “Welcome !”;
        }
        public string Message {
                        get {
                              return m_message;
                        }
                  set {
                        m_message = value;
              }
        }
  }
}
```

上面的代码展示了在名字空间 MyProgram.CSharp.FirstApp 定义的类

WelcomeMessage，类 WelcomeMessage 提供了可读的属性 Message。我们看到，名字空间可以嵌套，比如声明：

```
namespace MyProgram.CSharp.FirstApp
{…}
```

实际上是以下多层名字空间的嵌套：

```
namespace MyProgram
{
  namespace CSharp
  {
        namespace FirstApp
        {…}
  }
}
```

“Welcome”程序组件化的下一步就是编写客户控制台程序，客户程序将使用库中提供的类 WelcomeMessage。实际上 WelcomeMessage 这个类的全名应该是 MyProgram.CSharp.FirstApp. WelcomeMessage，但这种方法使用的书写的名字太长，显得过于笨拙。一个简单的方法是预先导入名字空间，这样就可以不通过全名来使用名字空间的所有类型。

程序清单 16-2：

```
// 客户程序源文件 WelcomeApp.cs
using System;
using MyProgram.CSharp.FirstApp;
class WelcomeApp
{
  public static void Main() {
        WelcomeMessage M = new WelcomeMessage();
        System.Console.WriteLine(M.Message);
        System.Console.WriteLine("Please Enter your name:");
        string input = System.Console.ReadLine();
        M.Message = "Welcome" + input;
        System.Console.WriteLine(M.Message);
        }
  }
```

上例中，导入了名字空间 MyProgram.CSharp.FirstApp 后，每次出现的 WelcomeMessage 实际上就是 MyProgram.CSharp.FirstApp. WelcomeMessage 的简写。

我们写的代码可以编译后产生一个包含类 WelcomeMessage 的库，以及一个包含类 WelcomeApp 的应用程序。具体的编译步骤的细节，可能会因为使用编译器或工具的不同而不同。使用 Visual Studio 7.0 提供的命令行编译器，正确的语法应该是：

```
csc /target:library WelcomeLibrary.cs
```

上面这条语句产生 WelcomeLibrary.dll 类库。

```
csc /reference: WelcomeLibrary.dll WelcomeApp.cs
```

上面这条语句产生 WelcomeApp.exe 应用程序。

16.3 使用指示符

使用指示符的目的是为了方便使用其它的名字空间中定义的名字空间和类型。名字空间的指示符有两种：别名使用指示符（*using-alias-directive*）和名字空间使用指示符（*using-namespace-directive*）。

16.3.1 别名使用指示符

C# 支持使用别名。别名使用指示符定义一个别名，以后就可以使用这个别名来代替一个类型。这在两个库的名字可能发生冲突的情况下非常有用。别名还可以避免使用冗长的名字空间。如下面的代码定义了 MessageSource 作为类 WelcomeMessage 的别名：

```
using MessageSource = MyProgram.CSharp.FirstApp. WelcomeMessage;
```

别名使用指示符为我们提供了一个标识符，在整个编译单元或是在名字空间主体之中，这个标识符作为名字空间或类型的别名。别名使用指示符的语法格式为：

```
using identifier = namespace-or-type-name ;
```

可以在成员声明中包含别名使用指示符，这个别名可以用来指代名字空间，也可以用来指代类型，例如：

```
namespace N1.N2
{
  class A {}
}
   namespace N3
{
  using A = N1.N2.A;
  class B: A {}
}
```

那么在名字空间 N3 的成员声明中，A 作为 N1.N2.A 的别名，则：

```
class B: A {}
```

表示 N3.B 从类 N1.N2.A.中继承。下面例子中的写法也可以达到同样的效果：

```
namespace N3
{
  using R = N1.N2;
```

```
  class B: R.A {}
}
```

注意别名的标识符不能与同一编译单元或名字空间中的其它成员同名。例如下面的代码就是不合法的，因为名字空间 N3 已经包含了成员 A：

```
namespace N3
{
  class A {}
}
   namespace N3
{
  using A = N1.N2.A;          // 错误, A 已经存在
}
```

在用到别名使用指示符时，其中的别名只是在特定的编译单元或名字空间中有效。看下面的例子：

```
namespace N3
{
  using R = N1.N2;
}
   namespace N3
{
  class B: R.A {}             // 错误, R 在这里是未知
}
```

别名 R 只在包含它名字空间的主体中有效，在名字空间的第二次声明中 R 是未知的。但把别名使用指示符放在编译单元中，别名对于两处声明都是有效的：

```
using R = N1.N2;
namespace N3
{
  class B: R.A {}
}
   namespace N3
{
  class C: R.A {}
}
```

16.3.2 名字空间使用指示符

使用名字空间指示符为编译单元或名字空间主体导入了另一个名字空间的类型，对这些类型的使用无需全名。使用名字空间指示符的语法格式：

```
using   namespace-name   ;
```

在编译单元或名字空间的成员声明中，名字空间中包含的类型可以被成员直接使用。例如：

```
namespace N1.N2
{
  class A {}
}
   namespace N3
{
  using N1.N2;
  class B: A {}
}
```

一个名字空间使用指示符导入了在给出的名字空间中包含的类型，但没有引入嵌套的名字空间。见下例：

```
namespace N1.N2
{
  class A {}
}
   namespace N3
{
  using N1;
  class B: N2.A {}        // 错误, N2 未知
}
```

这里，名字空间使用指示符导入了 N1 中包含的类型，但没有导入 N1 中的嵌套。所以，在类 B 的声明中引用 N2.A 是错误的。

和别名使用指示符不同，名字空间使用指示符可以导入在编译单元或名字空间主体中预定义的类型。

在使用名字空间的时候，如果在编译单元或名字空间主体中定义的成员与名字空间使用指示符导入的成员名字相同，编译器将认作是前者。也就是说，编译单元或名字空间主体中定义的成员覆盖了名字空间使用指示符导入的同名成员。

```
namespace N1.N2
{
  class A {}
  class B {}
}
   namespace N3
{
  using N1.N2;
  class A {}
}
```

上面的例子中，因为名字空间 N3 中声明了成员 A，所以 A 表示的是 N3.A，而不是 N1.N2.A。

如果在编译单元或名字空间主体中，通过名字空间使用指示符导入了多个名字空间，而这些名字空间中又包含了同名的类型，对这类名字的使用将会引起混淆。

```
namespace N1
{
  class A {}
}
   namespace N2
{
  class A {}
}
   namespace N3
{
  using N1;
  using N2;
  class B: A {}                    // 错误, A 有二义性
}
```

由于名字空间 N1 和 N2 都包含了成员 A，而名字空间 N3 又同时导入了 N2 和 N1，那么在名字空间 N3 中就不能引用成员 A，因为这时编译器无法判断指的是 N1.A 还是 N2.A。要解决这个矛盾，我们要么给成员加上具体的限制，要么用名字空间使用指示符表明究竟引用的是哪个成员。如果表示的是 N1.A，那么下面的写法就是正确的：

```
namespace N3
{
  using N1;
  using N2;
  using A = N1.A;
  class B: A {}                    // A 表示 N1.A
}
```

和别名使用指示符一样，名字空间使用指示符并没有在编译单元或名字空间的声明中添加新的成员，并且只对它所在的编译单元或名字空间有作用，对其它的编译单元和名字空间则没有影响。

最后要说明的一点是，在同一个编译单元或名字空间主体中，对名字空间使用指示符没有顺序上的限制，它们之间不会相互影响。

16.4 程 序 示 例

上面我们介绍了对应用程序结构进行组织的方法，接下来我们举一个综合应用的例子。程序实现的功能很简单，但其中用到了我们讲解的各方面的知识。在程序中，我们利用名字空间把应用程序功能进行分割，并且用到了在一个名字空间中包含多个类、在一个可执行文件中调用多个动态链接库的方法。程序中还用到了异常处理、类的继承、派生类对虚方法的重载、多态性的实现等概念，希望读者能够在阅读程序时认真注意这些用法，来加深对基本概念的理解。

该程序是一个小游戏，游戏中随机产生矩形、正方形、直角三角形、等腰直角三角形四种图形。游戏开始给每个用户 1000 分，然后由用户自行押注，根据押注的多少和随机产生的图形面积计算用户赢取的分数。程序中用到了名字空间 System 中提供的类 Random，该类提供了用于产生一个随机数的方法。

程序清单 16-3：

```
//MyShape.cs——源文件，用于定义图形类，作为其它图形的基类
using System;
namespace MyShape
{
   public class Shape
   {
      public virtual void Draw()
      {;}       //虚方法，用于图形绘制
      public virtual int GetArea(){
         return 0;    //虚方法，用于计算图形面积
      }
   }
}

//rect.cs——源文件，用于定义矩形类和正方形类
using System;
namespace MyShape
{
   public class Rectangle : Shape     //定义矩形类
   {
      protected int a;
      protected int b;       //矩形的边长
      public Rectangle(int va,int vb)
      {
         a = va;
         b = vb;
      }
      public override int GetArea()          //重载虚方法，计算矩形面积
```

```
        {
            int area=a*b;
            return area;
        }
        public override void Draw()     //重载虚方法，在屏幕上绘制矩形
        {
          Console.WriteLine("Rectangle:");
          Console.WriteLine("* * * * *");
          Console.WriteLine("*               *");
          Console.WriteLine("*               *");
          Console.WriteLine("* * * * *");
        }
    }

    public class Square : Rectangle            //定义正方形类
    {
        public Square(int va) : base(va,va)
        {;}
        public override void Draw()   //重载，绘制正方形
        {
          Console.WriteLine("Square");
          Console.WriteLine("* * * * *");
          Console.WriteLine("*               *");
          Console.WriteLine("*               *");
          Console.WriteLine("*               *");
          Console.WriteLine("* * * * *");
        }
    }
}

//triangle.cs——源文件，用于三角形
using System;
namespace MyShape
{
    //定义普通三角形，作为其它三角形的基类
    public class Triangle : Shape
    {
        protected int a;
        protected int b;
```

```
    protected int c;
    public Triangle(int va,int vb,int vc)
    {
      a = va;
      b = vb;
      c = vc;
    }
    public override int GetArea()
    {
      int s=(a+b+c)/2;
      int area=(int)(Math.Sqrt(s*(s-a)*(s-b)*(s-c)));
      return area;
    }
  }
  //定义直角三角形
  public class RectTriangle : Triangle
  {
    new protected int a;
    new protected int b;
     public RectTriangle(int va,int vb) :
base(va,vb,(int)(Math.Sqrt(va*va+vb*vb)))
    {
      a = va;
      b = vb;
    }
    public override int GetArea()
    {
      int area=(int)(a*b/2);
      return area;
    }
    public override void Draw()
    {
      Console.WriteLine("RectTriangle");
      Console.WriteLine("*");
      Console.WriteLine("* *");
      Console.WriteLine("*     * ");
      Console.WriteLine("*   *   *");
    }
  }
```

```
    //定义等腰直角三角形
    public class RectEqualTriangle : RectTriangle
    {
        new protected int a;
        public RectEqualTriangle(int va) : base(va,va)
        {
            a = va;
        }
        public override int GetArea()
        {
            int area=(int)(a*a/2);
            return area;
        }
        public override void Draw()
        {
            Console.WriteLine("RectEqualTriangle");
            Console.WriteLine("*");
            Console.WriteLine("* *");
            Console.WriteLine("*     * ");
            Console.WriteLine("*         *");
            Console.WriteLine("* * * * *");
        }
    }
}

//Mymessage.cs——源文件，用于定义程序显示的一些信息
using System;
namespace MyMessage
{
    public class Message
    {
        public void Begin()
        {
          Console.WriteLine("***********                ***********");
          Console.WriteLine("*             *            *           *");
          Console.WriteLine("**********   *           *   **********");
          Console.WriteLine("*            *   *        *   *          *");
          Console.WriteLine("*            *   *      *   *          *");
          Console.WriteLine("*              *   *    *   *          *");
```

```
        Console.WriteLine("*                  *   * *   *                 *");
        Console.WriteLine("*                    *   *   *                  *");
        Console.WriteLine("*                      *     *                  *");
        Console.WriteLine("*                        * *                    *");
        Console.WriteLine("*                          *                    *");
        Console.WriteLine("*                  SHAPE GAME                   *");
        Console.WriteLine("***********************************");
      }
      public bool Ask()
      {
        Console.WriteLine("Press 0 to exit the game");
        Console.WriteLine("Press any other key to continue the game");
        Console.WriteLine();
        int c = Console.Read();
        if(c == 48)
            return false;
        return true;
      }
    }
}

//client.cs——客户程序
using System;
using MyShape;
using MyMessage;
class ClientTest
{
   public static void Main()
   {
        int score = 1000; //总分
        int win;      //每一局赢取的分数
        int choice;  //随机获得的图形号
        int bet;      //每一局下的注
        string s;
        Shape sp = new Shape();
        Random ran = new Random();
        Message msg = new Message();
        msg.Begin();
        while(true)
```

```
{
  if(!msg.Ask())
    break;
  Console.WriteLine("Your Score: {0}",score);
  Console.WriteLine("Enter your bet: ");
  Console.ReadLine();
  s = Console.ReadLine();
     //如果押注的输入不正确，进行异常处理，并默认下注为 100 分
  try {
     bet = s.ToInt32();
  }
  catch{
    bet = 100;
  }
  if(bet < score)
    score -= bet;
  else{
    bet = score;
    score = 0;
  }
  Console.WriteLine("Remain Score: {0}",score);
  win = 0;
  for(int i=0;i<3;i++)
  {
    choice = ran.Next() % 4;       //随机数发生器
    switch(choice)
    {
       case 0:
          sp = new RectTriangle(5,4);
          goto end;
       case 1:
          sp = new RectEqualTriangle(5);
          goto end;
       case 2:
          sp = new Rectangle(5,4);
          goto end;
       case 3:
          sp = new Square(5);
    }
```

```
                end:
                  //利用多态性，计算得分
                   sp.Draw();
                  win += sp.GetArea()*(i+1)*bet/100;
                  Console.WriteLine("Your win: {0}",win);
               }
                  score += win;
                  Console.WriteLine("Your Score: {0}",score);
               if(score<100)
               {
            Console.WriteLine("Your remain score is not enough
                   to play");
                  break;
               }
            }
         }
      }
```

对源代码进行编译的命令为：

```
csc/target:library /out:MyShape.dll    MyShape.cs    rect.cs    triangle.cs
csc/target:library /out:MyMessage.dll MyMessage.cs
csc/reference:MyShape.dll; MyMessage.dll    client.cs
```

16.5 小　　结

本章为读者讲解了程序编译的不同方式：动态链接库和可执行文件。使用名字空间可以为 C#可执行文件或者库导入其它的动态链接库。把一个大型程序分割为一系列的可执行文件和动态链接库将大大方便我们对应用程序的组织和管理。

为了方便对名字空间的使用，我们可以视具体情况采用别名使用指示符和名字空间使用指示符。

本章最后举出了一个较为完整的程序例子，读者可以从中学习到组织应用程序结构的方法。

复习题

（1）名字空间为我们管理组织 C#程序带来了哪些方便之处？

（2）举例说明动态链接和静态链接的不同点。

（3）C#程序的装配有哪两种，它们通常分别在哪种条件下使用？

（4）名字空间中定义的类型是否都可以被导出，以便供其它程序使用？

（5）说说别名使用指示符和名字空间使用指示符之间有什么区别，以及它们各自的使用场合。

（6）说明别名使用指示符和名字空间使用指示符各自的使用有效范围。

第十七章　文 件 操 作

文件管理是操作系统的一个重要组成部分，而文件操作就是对用户在编写应用程序时进行文件管理的一种手段。

目前有许多文件系统。在我们使用过的从 DOS、Windows3.X、Windows95、WindowsNT、Windows2000 这些操作系统中，用到了我们非常熟悉的 FAT、FAT32、NTFS 等文件系统。这些文件系统在操作系统内部实现时有不同的方式，然而它们提供给用户的接口是一致的。只要按照正规的方式来编写代码，而且程序不涉及到操作系统的具体特性，那么生成的应用程序就可以不经过改动，而在不同的操作系统上移植。因此，在编写对文件操作的代码时，我们不需要考虑具体的实现方式，只需要利用语言环境给我们提供的外部接口。

一个完整的应用程序，肯定要涉及到对系统和用户的信息进行存储、读取、修改等操作，还常常需要设计自己的文件格式。因此，有效地实现文件操作，是一个良好的应用程序所必须具备的内容。

C#为我们提供了文件操作的强大功能。利用.Net 环境所提供的功能，我们可以方便地编写 C#程序，实现文件的存储管理、对文件的读写等各种操作。

17.1　.Net 框架结构提供的 I/O 方式

在 System.IO 名字空间中提供了多种类型，用于进行数据文件和数据流的读写操作。这些操作可以同步进行，也可以异步进行。

17.1.1　文件和流

文件（file）和流（stream）即有区别又有联系。文件是在各种媒质上（可移动磁盘、硬盘、CD 等）永久存储的数据的有序集合。它是一种进行数据读写操作的基本对象。通常情况下文件按照树状目录进行组织，每个文件都有文件名、文件所在路径、创建时间、访问权限等属性。

从概念上讲，流非常类似于单独的磁盘文件，它也是进行数据读取操作的基本对象。流为我们提供了连续的字节流存储空间。虽然数据实际存储的位置可以不连续，甚至可以分布在多个磁盘上，但我们看到的是封装以后的数据结构，是连续的字节流抽象结构。这和一个文件也可以分布在磁盘上的多个扇区一样。

除了和磁盘文件直接相关的文件流以外，流有多种类型，流可以分布在网络中、内存中或者是磁带中。

17.1.2 支持输入输出操作的类型

Stream

System.IO 为我们提供了一个抽象类 Stream，Stream 类支持对字节的读写操作。在 Stream 类中包括了对异步操作的支持。

既然 Stream 是抽象类，所有其它流的类就都必须从 Stream 类中继承。Stream 类及其子类共同构成了一个数据源和数据存储的视图，从而封装了操作系统和底层存储的各个细节，使程序员把注意力集中到程序的应用逻辑上来。

流包含以下基本操作：

读操作（Reading）。即读出流中的数据，把数据存放在另一种数据结构中，比如数组。

写操作（Writing）。即从另一种数据结构中读出数据，存放至流对象中。

搜索操作（Seeking）。即从流中的当前位置开始搜索，定位到指定的位置。

由于数据视图的不同，一些流可能不同时支持以上的所有操作。比如网络流就不支持搜索操作。Stream 类提供了 CanRead, CanWrite 和 CanSeek 三种属性，来表示流是否支持这些操作。

BinaryReader 和 BinaryWriter

BinaryReader 和 BinaryWriter 这两个类提供了从字符串或原始数据到各种流之间的读写操作。

File 和 Directory

File 类支持对文件的基本操作，包括创建、拷贝、移动、删除和打开一个文件。Directory 类则用于执行常见的各种目录操作，如创建、移动、浏览目录及其子目录。

File 类和 Directory 类都是密封类。不像抽象类 Stream，File 类和 Directory 类可以被实例化，但它们不能被其它类继承。

File 类和 Directory 类的基类都是抽象类 FileSystemEntry。

Stream

File 类的静态方法主要是用于创建 FileStream 类。一个 FileStream 类的实例实际上代表一个磁盘文件，它通过 Seek（）方法进行对文件的随机访问，也同时包含了流的标准输入、标准输出、标准错误等。FileStream 默认对文件的打开方式是同步的，但它同样很好地支持异步操作。

TextReader 和 TextWriter

TextReader 和 TextWriter 类都是抽象类。和 Stream 类的字节形式的输入和输出不同，它们用于 Unicode 字符的输入和输出。

StringReader 和 StringWriter

StringReader 和 StringWriter 在字符串中读写字符。

StreamReader 和 StreamWriter

StreamReader 和 StreamWriter 在流中读写字符。

BufferedStream

BufferedStream 是为诸如网络流的其它流添加缓冲的一种流类型。其实，FileStream 流自身内部含有缓冲，而 MemoryStream 流则不需要缓冲。一个 BufferedStream 类的实例可以由多个其它类型的流复合而成，以达到提高性能的目的。缓冲实际上是内存中的一个字节块，利用缓冲可以避免操作系统频繁地到磁盘上读取数据，从而减轻了操作系统的负担。

MemoryStream

MemoryStream 是一个无缓冲的流，它所封装的数据直接放在内存中，因此可以用于快速临时存储、进程间传递信息等。

NetworkStream

NetworkStream 表示在互联网络上传递的流。

当使用名字空间 System.IO 中提供的类时，对存储数据的访问权限必须符合操作系统的安全性要求。

注意：不要使用这些类来编写应用程序对网络文件进行的操作。因为 Internet 默认的安全政策是不允许对文件直接访问。可以使用 IsolatedStroage 类来处理网络文件。

17.2 文件存储管理

17.2.1 目录管理

.Net 框架结构在名字空间 System.IO 中为我们提供了 Directory 类来进行目录管理。利用它，我们可以完成对目录及其子目录进行创建、移动、浏览等操作，甚至还可以定义隐藏目录和只读目录。

Directory 的构造函数形式如下：

```
public Directory( string path);
```

其中的参数 path 表示目录所在的路径。

Directory 的主要属性有：

- Attributes：0x01 表示只读，0x02 表示隐藏。

- Name：当前路径名。
- Parent：上一级父目录名。
- Root：所在根目录名。
- CreationTime：目录创建时间。
- LastAccessTime：上一次访问目录的时间。
- LastWriteTime：上一次修改目录的时间。

我们常用到的 Directory 类的一些成员方法有：

- CreateDirectory(string path)：创建子目录。
- CreateDirectories(string path)：创建多级子目录。
- CreateFile(string filename)：在当前目录下创建一个新文件。
- Delete()：删除目录。

下面的例子中，我们利用 Directory 的 CreateDirectory 和 CreateDirectories 方法创建一级和二级子目录。

程序清单 17-1：

```
using System;
using System.IO;
class DirectoryTest
{
   public static void Main()
   {
      Directory d = new Directory("c:\\c#");
      Directory d1;
      Directory d2;
      try{
         d1 = d.CreateSubdirectory("file1");
      }
      catch(IOException e)
      {
         Console.WriteLine("directory file1 failed because: {0}",e);
         return;
      }
      try{
         d2 = d.CreateSubdirectories("file1\\file2");
         //等于 d2 = d1.CreateSubDirectory("file2");
      }
      catch(IOException e)
      {
         Console.WriteLine("directory file2 failed because: {0}",e);
         return;
```

```
        }
        Console.WriteLine("Create directory successfully!");
    }
}
```

上面例子中的异常 IOException 表示准备创建的目录名已经存在。System.IO 中提供了各种输入输出的异常，便于我们了解任务失败的原因所在，或者是给程序的用户提示信息。

同样我们可以进行目录删除操作。

程序清单 17-2：

```
using System;
using System.IO;
class DirectoryTest
{
    public static void Main()
    {
        Directory d = new Directory("c:\\c#\\file1");
        Directory[] subdir;
        try{
            subdir = d.GetDirectories();
        }
        catch(DirectoryNotFoundException e)
        {
            Console.WriteLine("Can not find such a directory because: {0}",e);
            return;
        }
        foreach(Directory dir in subdir){
            dir.Delete();
        }
        d.Delete();
        Console.WriteLine("Delete successfully!");
    }
}
```

17.2.2 文件管理

File 类通常和 FileStream 类协作来完成对文件的创建、删除、拷贝、移动、打开等操作。

File 类的构造函数形式为：

```
public File(string fileName);
```

同样，File 类也有自己的属性，如绝对路径名 DerictoryName、创建时间 CreationTmie、上次访问时间 LastAccessTime、上次修改时间 LastWriteTime、文件长度 Length 等。

File 类为我们提供的方法主要有 CopyTo、MoveTo、Delete 等，利用它们可以完成一些基本的文件管理。

下面给出的例子用于简单的文件拷贝。这是一个带命令行参数的可执行文件，第一个参数表示源文件的绝对路径名，第二个参数表示目标文件的绝对路径名。

程序清单 17-3：

```
using System;
using System.IO;
class FileTest
{
  public static void Main(String[] args)
  {
    //验证参数输入是否正确
    string sourcename;
    string targetname;
    try{
      sourcename = args[0];
    }
    catch
    {
      Console.WriteLine("Please input the name of the source file correctly");
      return;
    }
    try{
      targetname = args[1];
    }
    catch
    {
      Console.WriteLine("Please input the name of the target file correctly");
      return;
    }
    //进行文件拷贝操作
    File file;
    try{
      file = new File(sourcename);
    }
    catch
```

```
        {
            Console.WriteLine("Can not open the source file named '{0}'",sourcename);
            return;
        }
        try{
            file.CopyTo(targetname,true);
        }
        catch
        {
            Console.WriteLine("Can write the target file named '{0}'",targetname);
            return;
        }
        Console.WriteLine("File copys successfully!");
    }
}
```

上面我们用到了 File 的 CopyTo 方法。若该为 MoveTo 方法，则程序可以用于文件的移动操作。

下面的例子则用于删除当前目录下的所有文件。

程序清单 17-4：

```
using System;
using System.IO;
class FileTest
{
    public static void Main()
    {
        Console.WriteLine("Are you sure to delete all the files in current directory?");
        Console.WriteLine("Press 'Y' then enter to continue,any other key to abort");
        int a = Console.Read();
        if(a == 'Y' || a == 'y'){
            Console.WriteLine("Deleting the files...");
        }
        else{
            Console.WriteLine("canceled by user");
            return;
        }
        Directory dir = new Directory(".");
        foreach (File f in dir.GetFiles()) {
            f.Delete();
        }
```

```
    }
  }
```

其中，语句 foreach (File f in dir.GetFiles())表示查找当前所有文件。Directory 类的成员方法 GetFiles 有两种重载形式：

```
public File[] GetFiles();
public File[] GetFiles( string searchCriteria);
```

后面一种方法中的参数表示搜索文件的条件。

我们对程序 17-4 可以稍做改动，把最后一段代码改写为：

```
foreach (File f in dir.GetFiles("*.exe")) {
    f.Delete();
}
```

这样，程序实际上变成了删除当前目录下的所有扩展名为.exe 的文件。

值得一提的是，作者在使用 File.Delete 方法时遇到了一个有趣的现象。文件删除后，在 Windows 的回收站中找不到，但是使用工具软件 Norton Unerase Wizard 却成功地找到并恢复了被删除的文件，而且文件名还保存完好（Windows 彻底删除的文件恢复时文件名的第一个字符已被删除）。看来 File.Delete 方法也并不是彻底地抹去文件的内容，而只是对被删除的文件作出某个标记而已。

17.3 读 写 文 件

17.3.1 按文本模式读写

StreamReader 和 StreamWriter 类为我们提供了按文本模式读写数据的方法。下面的例子是从一个文本文件中读取内容并打印在屏幕上。

程序清单 17-5：

```
using System;
using System.IO;
class FileTest
{
  public static void Main()
  {
    StreamReader srd;
    try{
      srd = File.OpenText("c:\\c#\\file1\\file1.txt");
    }
    catch
    {
      Console.WriteLine("File open failed");
```

```
            return;
        }
        while (srd.Peek()!=-1)
        {
            String str = srd.ReadLine();
            Console.WriteLine (str);
        }
        Console.WriteLine ("The end of the file has been reached");
        srd.Close();
    }
}
```

上面用到的方法 OpenText 是 File 类的一个静态方法，不能被某个具体的 File 类的实例调用。它表示从一个已经存在的文本文件中读取一个文本流，并保存在一个 StreamReader 实例中。下面的例子则是向文本文件中写入文本流。

程序清单 17-6：

```
using System;
using System.IO;
class FileTest
{
    public static void Main()
    {
        StreamWriter srd;
        try{
            srd = File.CreateText("c:\\c#\\file1\\file2.txt");
        }
        catch
        {
            Console.WriteLine("File create failed");
            return;
        }
        srd.WriteLine ("Web Address:");
        srd.WriteLine ("www.sohu.com");
        srd.WriteLine ("www.263.net");
        srd.WriteLine ("www.microsoft.com/china");
        srd.WriteLine ("www.sina.com.cn");
        srd.Close();
    }
}
```

17.3.2 按二进制模式读写

System.IO 还为我们提供了 BinaryReader 和 BinaryWriter 类，用于按二进制模式读写文件。它们提供的一些读写方法是对称的，比如针对不同的数据结构，BinaryReader 提供了 ReadByte，ReadBoolean，ReadInt，ReadInt16，ReadDouble，ReadString 等方法，而 BinaryWriterz 则提供了 WriteByte，WriteBoolean，WriteInt，WriteInt16，WriteDouble，WriteString 方法。

下面我们举一个通讯录的例子来说明 BinaryReader 和 BinaryWriter 类是如何使用的。

程序清单 17-7：

```
using System;
using System.IO;
struct Record{
   public string name;
   public UInt16 age;
   public string phone;
   public string address;
};      //通讯录的记录格式

class PhoneBook
{
   FileStream f_srm;
   string s_filename;
   Record m_record;
   public PhoneBook(string filename)
   {
      s_filename = filename;
   }

   public int Open()
   {    //显示初始化界面
      Console.WriteLine("1: List Record");
      Console.WriteLine("2: Add New Record");
      Console.WriteLine("3: Erase Record");
      Console.WriteLine("0: Exit");
      int i = Console.Read() - '0';
      return i;
   }
   public void AddNew()
```

```
{    //向通讯录中添加记录
   try{
      f_srm = new FileStream(s_filename,FileMode.OpenOrCreate);
   }
   catch
   {
      Console.WriteLine("PhoneBook Error!");
      return;
   }
   BinaryWriter pw = new BinaryWriter(f_srm);
   pw.Seek(0,SeekOrigin.End);
   string s_temp;
   Console.Write("Name: ");
   Console.ReadLine();
   m_record.name = Console.ReadLine();
   pw.WriteString(m_record.name);
   reenter:
   Console.Write("Age: ");
   s_temp = Console.ReadLine();
   try{
      m_record.age = s_temp.ToUInt16();
   }
   catch
   {
      Console.WriteLine("The format of age is error,please ReEnter");
      goto reenter;
   }
   pw.Write(m_record.age);
   Console.Write("Phone Number: ");
   m_record.phone = Console.ReadLine();
   pw.WriteString(m_record.phone);
   Console.Write("Address: ");
   m_record.address = Console.ReadLine();
   pw.WriteString(m_record.address);
   f_srm.Close();
}
public void List()
{    //浏览所有记录
   try{
```

```
            f_srm = new FileStream(s_filename,FileMode.Open);
        }
        catch
        {
            Console.WriteLine("PhoneBook Error!");
            return;
        }
        BinaryReader pr = new BinaryReader(f_srm);
        Console.WriteLine("Name        Age     Phone Number        Address");
        Console.WriteLine("****************************************");
        while(true)
        {
        try{
        m_record.name = pr.ReadString();
        m_record.age = pr.ReadUInt16();
        m_record.phone = pr.ReadString();
        m_record.address = pr.ReadString();
        }
        catch{return;}
        Console.WriteLine("{0,-9}{1,-6}{2,-18}{3}",m_record.name,
m_record.age,m_record.phone,m_record.address);
        }
        f_srm.Close();
    }
    public void Erase()
    {    //清空所有记录
        try{
            f_srm = new FileStream(s_filename,FileMode.Create);
        }
        catch
        {
            Console.WriteLine("PhoneBook Error!");
            return;
        }
        f_srm.Close();
        Console.WriteLine("Now the book is empty!");
    }
}
```

```
class Test
{
   public static void Main()
   {
      PhoneBook pbook = new PhoneBook("c:\\c#\\file\\Phone.book");
      int i = pbook.Open();
      switch(i)
      {
       case 1:
          pbook.List();
          break;
       case 2:
          pbook.AddNew();
          break;
       case 3:
          pbook.Erase();
       }
   }
}
```

其中有这么一条语句：

```
Console.WriteLine("{0,-9}{1,-6}{2,-18}{3}",m_record.name,
m_record.age,m_record.phone,m_record.address);
```

它表示对输出进行格式化。“{}”中使用的第二个数表示输出占据的屏幕位数。正数表示从右边起计，而负数表示从左边起计。

17.4 异步文件操作

先来说说同步和异步操作之间的主要区别。在同步 I/O 操作中，方法将一直处于等待状态，直到 I/O 操作完成。而在异步 I/O 操作中，在开始了 I/O 操作后，程序的方法可以转移去执行其它的操作，这样大大提高了程序执行的效率。

由于 Windows 是一个多任务操作系统，在同一时刻系统可能会接受到多个 I/O 操作请求，要求对磁盘文件执行各种操作。如果采用同步方式，那么每时每刻最多只能有一个 I/O 操作在进行，而其它的任务都处于等待状态，系统的利用率将会大为降低。异步 I/O 操作则较好地解决了这种性能上的问题。

Stream 类支持在同一个流中既可以进行同步读写，也可以进行异步读写。Stream 类是一个抽象类，它为我们提供了 BeginRead、BeginWrite、EndReader、EndWrite、Read、Write、Seek 等成员方法，协同完成对流的读写操作。所有这些方法都是虚方法。因此，

在我们自己设计 Stream 类的派生类时，我们在类用于读写的成员方法 Read 和 Write 中应该重载这些方法，并同时设计它们同步和异步的执行代码。BeginRead, EndRead, BeginWrite, 和 EndWrite 方法默认为我们提供的是异步读写操作方式，如果你的派生类的 Read 和 Write 方法执行同步操作时，那么程序提供的效率不会很好。只有当它们执行异步操作时，我们才能有效地提高程序的执行效率。

Stream 类还提供了 ReadByte 和 WriteByte 方法，用于一次读写一个字节。它们在默认情况下实际上是调用了 Read 和 Write 方法的同步操作。

如果在实际情况下，一个流不支持同步或者异步的读写方式，这时我们就需要编写自己的方法来抛出一个异常。

下面的代码是在.NET 联机帮助中提供的一个异步读写操作的例子，程序模拟了一个多处理器系统的工作。

程序清单 17-8：

```
using System;
using System.IO;
using System.Threading;
using BenchUtil;
public class BulkImageProcAsync{
    public const String ImageBaseName = "tmpImage-";
public const int numImages = 200;
public const int numPixels = 512*512;
    // ProcessImage has a simple O(N) loop, and we can vary the number
    // of times we repeat that loop to make the app more CPU-bound or
// more IO-bound.
public static int processImageRepeats = 20;
    // Threads must decrement NumImagesToFinish, and protect
    // their access to it via a mutex.
    public static int NumImagesToFinish = numImages;
    public static Object NumImagesMutex = new Object[0];
    // WaitObject is signalled when all image processing is done.
    public static Object WaitObject = new Object[0];
    internal static PerfTimer Pf = new PerfTimer("Asynchronous Bulk Image Processor");
public class ImageStateObject    {
        public byte[] pixels;
          public int imageNum;
  }
    public static void MakeImageFiles()    {
      int sides = (int) Math.Sqrt(numPixels);
      Console.Write("Making "+numImages+" "+sides+"x"+sides+" images...  ");
        byte[] pixels = new byte[numPixels];
```

```
                for(int i=0; i<numPixels; i++)
                pixels[i] = (byte) i;
                        for(int i=0; i<numImages; i++) {
                                FileStream fs = new FileStream(ImageBaseName+i+
                                        ".tmp", FileMode.Create, FileAccess.Write, FileShare.None,
                                8192, false);
                                fs.Write(pixels, 0, pixels.Length);
                                FlushFileBuffers(fs.GetHandle());
                                fs.Close();
                                }
                                Console.WriteLine("Done.");
                        }
            public static void ReadInImageCallback(IAsyncResult asyncResult)    {
          ImageStateObject state = (ImageStateObject) asyncResult.AsyncState;
        Console.WriteLine("Image "+state.imageNum+" was read.
    "+(asyncResult.CompletedSynchronously ? "synchronously" : "asynchronously"));
                Stream stream = (Stream) asyncResult.AsyncObject;
                int bytesRead = stream.EndRead(asyncResult);
                if (bytesRead != numPixels)
                   throw new Exception("In ReadInImageCallback, got wrong number    of bytes
      from the image!    got: "+bytesRead);
                 ProcessImage(state.pixels, state.imageNum);              stream.Close();
                // Now write out the image.
                // Using async IO here probably swamps the threadpool, since
                // there are blocked threadpool threads on soon-to-be-//spawned
                // threadpool threads.
                FileStream fs = new FileStream(ImageBaseName+state.imageNum+".done",
FileMode.Create, FileAccess.Write, FileShare.None, 4096, false);
                fs.Write(state.pixels, 0, numPixels);
                //IAsyncResult writeResult = fs.BeginWrite(state.pixels,
              //0, numPixels, null, null);             //fs.EndWrite(writeResult);
                fs.Close();
                // Release memory as soon as possible, especially global state.
                state.pixels = null;                        // Record that an image is done now.
                lock(NumImagesMutex) {                  NumImagesToFinish--;
                     if (NumImagesToFinish==0) {
                          Monitor.Enter(WaitObject);
                          Monitor.Pulse(WaitObject);
                          Monitor.Exit(WaitObject);
```

```
                    }
                }
        }
            public static void ProcessImage(byte[] pixels, int imageNum)       {
                //Console.WriteLine("ProcessImage "+imageNum);
                // Do some CPU-intensive operation on the image.
                for(int i=0; i<processImageRepeats; i++)
                    for(int j=0; j<numPixels; j++)
                        pixels[j] += 1;
             //Console.WriteLine("ProcessImage "+imageNum+" done.");       }
            public static void ProcessImagesInBulk()       {
                Console.WriteLine("Processing images...    ");
                //int timer = Pf.StartTimer("ProcessImages");
                int timer = Pf.StartTimer("Total Time");
                NumImagesToFinish = numImages;
                    AsyncCallback readImageCallback = new
               AsyncCallback(ReadInImageCallback);
                for(int i=0; i<numImages; i++) {
                    ImageStateObject state = new ImageStateObject();
                        state.pixels = new byte[numPixels];                    state.imageNum = i;
                    // Because very large items are read only once, the buffer
                    // on the file stream can be very small to save memory.
                    FileStream fs = new FileStream(ImageBaseName+i+".tmp", FileMode.Open,
FileAccess.Read, FileShare.Read, 1, true);
                    fs.BeginRead(state.pixels, 0, numPixels, readImageCallback, state);
                }             // Ensure all image processing is done.
                // If not, block until all are finished.
               bool mustBlock = false;
                lock (NumImagesMutex)
                    if (NumImagesToFinish > 0)
                        mustBlock = true;
                    }
                    if (mustBlock) {
                      Console.WriteLine("All worker threads are queued... Blocking until they
complete.    numLeft: "+NumImagesToFinish);
                      Monitor.Enter(WaitObject);                    Monitor.Wait(WaitObject);
                      Monitor.Exit(WaitObject);
                  }
                Pf.StopTimer(timer);
```

```
                Pf.OutputStoppedTime();
        }
        public static void Cleanup()    {
                for(int i=0; i<numImages; i++) {
                    File.Delete(ImageBaseName+i+".tmp");
                    File.Delete(ImageBaseName+i+".done");
                }
        }
            public static void TryToClearDiskCache()    {
                // Try to force all pending writes to disk, AND to clear the
                // disk cache of any data.          byte[] bytes = new byte[100*(1<<20)];
                for(int i=0; i<bytes.Length; i++)                bytes[i] = 0;
                bytes = null;          GC.Collect();          Thread.Sleep(2000);      }
            public static void Main(String[] args)    {
                Console.WriteLine("Bulk image processing sample application, using async IO");
                Console.WriteLine("Simulates applying a simple transformation to "+numImages+"
\"images\"");
                Console.WriteLine("(ie, Async FileStream & Threadpool benchmark)");
                Console.WriteLine("Warning - this test requires "+(numPixels * numImages * 2)+"
bytes of tmp space");
                if (args.Length==1) {
                    processImageRepeats = Int32.Parse(args[0]);
                    Console.WriteLine("ProcessImage inner loop - "+processImageRepeats);
                }
              MakeImageFiles();
              TryToClearDiskCache();
                    ProcessImagesInBulk();
              Cleanup();
          }
            [dllimport("KERNEL32", SetLastError=true)]
        static extern void FlushFileBuffers(int handle);
        }
```

这里是采用同步方法实现同样功能的程序。

程序清单 17-9：

```
        using System;
        using System.IO;
        using System.Threading;
        using BenchUtil;
        public class BulkImageProcSync
```

```
{
        public const String ImageBaseName = "tmpImage-";
    public const int numImages = 200;
    public const int numPixels = 512*512;
        // ProcessImage has a simple O(N) loop, and we can vary the number
        // of times we repeat that loop to make the app more CPU-bound or
    // more IO-bound.
    public static int processImageRepeats = 20;
        internal static PerfTimer Pf = new PerfTimer("Synchronous Bulk Image Processor");
        public static void MakeImageFiles()       {
            int sides = (int) Math.Sqrt(numPixels);
            Console.Write("Making "+numImages+" "+sides+"x"+sides+" images...    ");
            byte[] pixels = new byte[numPixels];
            for(int i=0; i<numPixels; i++)
                pixels[i] = (byte) i;
            for(int i=0; i<numImages; i++) {
                FileStream fs = new FileStream(ImageBaseName+i+".tmp", FileMode.Create,
FileAccess.Write, FileShare.None, 8192, false);
                fs.Write(pixels, 0, pixels.Length);
                    FlushFileBuffers(fs.GetHandle());                    fs.Close();
            }
            Console.WriteLine("Done.");
      }
         public static void ProcessImage(byte[] pixels, int imageNum)       {
            //Console.WriteLine("ProcessImage "+imageNum);
            // Do some CPU-intensive operation on the image
            for(int i=0; i<processImageRepeats; i++)
                for(int j=0; j<numPixels; j++)
    pixels[j] += 1;
            //Console.WriteLine("ProcessImage "+imageNum+" //done.");       }
        public static void ProcessImagesInBulk()       {
            Console.WriteLine("Processing images...    ");
            int timer = Pf.StartTimer("Total Time");
            byte[] pixels = new byte[numPixels];
            for(int i=0; i<numImages; i++) {
                FileStream input = new FileStream(ImageBaseName+i+".tmp",
            FileMode.Open, FileAccess.Read, FileShare.Read, 4196, false);
                input.Read(pixels, 0, numPixels);                    input.Close();
                ProcessImage(pixels, i);
```

```
                    FileStream output = new FileStream(ImageBaseName+i+
                    ".done", FileMode.Create, FileAccess.Write,
                    FileShare.None, 4196, false);
                output.Write(pixels, 0, numPixels);                    output.Close();
                }
            Pf.StopTimer(timer);
            Pf.OutputStoppedTime();
        }
    public static void Cleanup()
    {
                for(int i=0; i<numImages; i++) {
                 File.Delete(ImageBaseName+i+".tmp");
                 File.Delete(ImageBaseName+i+".done");
            }
        }
        public static void TryToClearDiskCache()       {
            byte[] bytes = new byte[100*(1<<20)];
            for(int i=0; i<bytes.Length; i++)
                bytes[i] = 0;
            bytes = null;
          GC.Collect();
           Thread.Sleep(2000);
        }
        public static void Main(String[] args)       {
            Console.WriteLine("Bulk image processing sample application, using synchronous
IO");
            Console.WriteLine("Simulates applying a simple transformation to "+numImages+"
\"images\"");
            Console.WriteLine("(ie, Sync FileStream benchmark)");
            Console.WriteLine("Warning - this test requires "+(numPixels * numImages * 2)+"
bytes of tmp space");
            if (args.Length==1) {
                processImageRepeats = Int32.Parse(args[0]);
                Console.WriteLine("ProcessImage inner loop - "+processImageRepeats);
            }
      MakeImageFiles();
      TryToClearDiskCache();
            ProcessImagesInBulk();
            Cleanup();
```

```
        }
        [dllimport("KERNEL32", SetLastError=true)]
        static extern void FlushFileBuffers(int handle);
    }
```

17.5 小　　结

应用程序中常常需要保存和读取一些信息，这时我们就会不可避免地遇到文件操作。

在 C#语言中进行文件操作，我们不需要关心文件的具体存储是哪种格式，只需要利用框架结构封装的对文件操作的统一接口，就可以保证程序在不同的文件系统上能够良好地移植。

.Net 框架结构在 System.IO 名字空间中提供了多种类型，用于进行数据文件和数据流的读写操作。其中我们经常用到的有：File，Stream，FileStream，BinaryReader，BinaryWriter，StreamReader，StreamWriter 等。其中 Stream 是抽象类，不允许直接使用类的实例，我们可以利用系统提供的或者创建自己的派生类。

我们详细地介绍了使用 File 类和 Directory 类进行目录和文件的管理，以及采用 StreamReader 类、StreamWriter 类、BinaryReader 类、BinaryWriter 类进行文本模式和二进制模式的文件的读写操作。在最后一小节，我们介绍了对文件的同步和异步操作方式。

复习题

（1）说说文件和流之间的区别和联系。

（2）说明 Directory 类为我们提供了哪些目录管理的功能，它们是通过哪些成员方法来实现的？

（3）使用 Directory 类和 File 类，编写一个拷贝整个目录（包括目录、子目录及目录下的所有文件）的程序。

（4）编写一个查找具有某一相同扩展名的所有文件的例子。

（5）内存文件流类 MemoryStream 与其它 I/O 类有什么不同之处？

（6）说明文件的同步读写方式和异步读写方式的区别在那里。

（7）利用 StreamReader 和 StreamWriter 类编写一个简单的文本编辑器。

（8）回顾第二部分中网络用户的例子，设计自己的文件结构，并编写程序来保存用户信息。

第十八章　高 级 话 题

18.1　注册表编程

18.1.1　注册表概述

Windows 操作系统的注册表中包含了有关计算机运行方式的配置信息，其中包括 Windows 操作系统配置信息、应用程序配置信息、专用用户设备配置信息、环境配置信息等。另外，在 WindowsNT 和 Windows2000 操作系统中，注册表中还包含了安全性、网络管理等配置信息。

在 Windows95 中提供了用于用户编辑注册表的工具 Regedit，它具有对注册表进行操作的强大功能。在 WindowsNT 和 Windows2000 操作系统中，相应的工具为 Regedt32。让我们在 Windows 的“开始”菜单中选择“运行”，键入 Regedit（假设是在 Windows9x 操作系统中），将打开注册表编辑器，界面如下图所示。

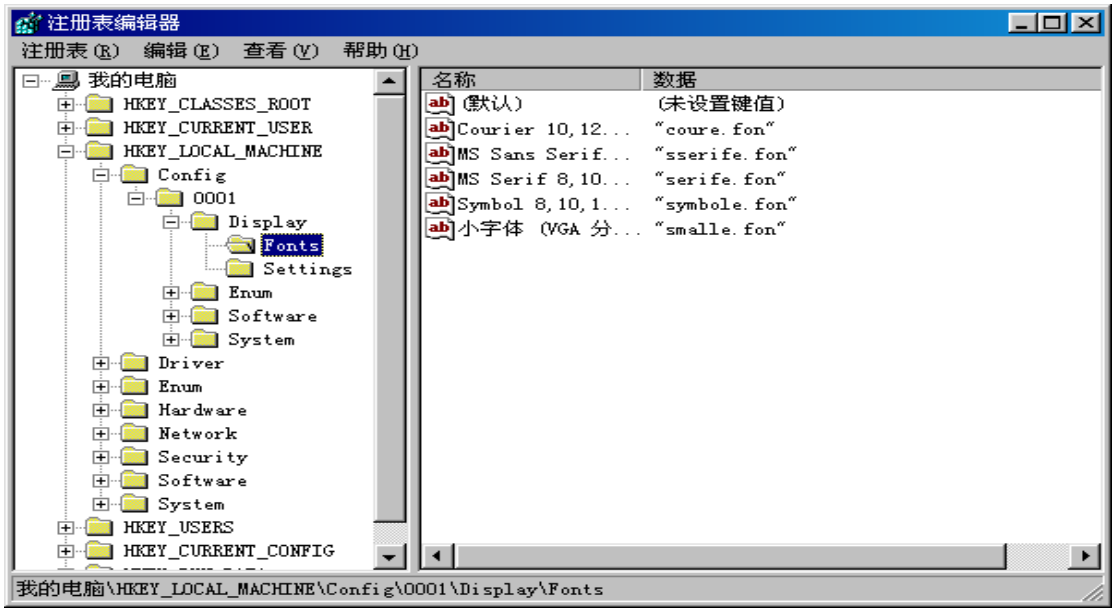


图 18-1　注册表编辑器

我们把在注册表编辑器中左边出现的称为“主键”，主键之间形成层次结构，主键

的下一级主键称为该主键的子键。主键可以对它赋一个或多个值，值的名称称为键值，注册表编辑器右边出现的“名称”一栏就是键值，“数据”一栏为键值的值。

键值的值的类型分为三种，即字符串（REG_SZ）、二进制（REG_BINARY）、还有双字（REG_DWORD）。WindowsNT 的注册表中还包括扩展字符串值（REG_EXPAND_SZ）和多字符串值（REG_MULTI_SZ）。

从图 18-1 中我们可以看到，Windows 操作系统注册的表是按类似于目录的树状结构来组织的。其中第二级子目录包含了六个预定义主键，把注册表分为六大部分，这六个部分的功能描述如下：

HKEY_CLASSES_ROOT 该主键中包含了文件的扩展名和应用程序的关联信息，以及 Windows Shell 和 OLE 用于存储注册类表的信息。该主键下的子键决定了如何在资源管理器和桌面中显示该类文件及其图标。它不是一个单独的分支，它实际上是从 HKEY_LOCAL_MACHINE\SOFTWARE\Classes 映射过来的。

- HKEY_CURRENT_USER 该主键包含了指向主键中当前用户的信息，如当前用户窗口信息、桌面设置信息、远程网络地址信息等。
- HKEY_LOCAL_MACHINE 该主键包含了本地计算机关于软件和硬件的安装和配置信息，其中的信息与特定的用户无关，可供所有用户在登录系统时使用。
- HKEY_USERS 该主键记录了当前 Windows 登录用户的设置信息。用户每次登录系统时，就在该主键下生成一个用户登录名的子键，在该子键下保存了该用户的桌面设置、背景位图、应用程序快捷键、显示字体等信息。这些信息都可以通过 Windows 系统中控制面板等一些工具来进行设置。一般应用程序不直接访问 HKEY_USERS 主键，而是通过主键 HKEY_CURRENT_USER 进行访问。
- HKEY_CURRENT_CONFIG 该主键机器当前的硬件配置信息，它实际上也不是独立存在的，而是指向 HKEY_LOCAL_MACHINE\CONFIG 结构中的某个子键信息。这些配置可以根据当前所连接的网络类型或硬件及硬件驱动软件安装的改变而改变。在 Windows NT 中没有该主键。
- HKEY_DYN_DATA 该主键保存一些实时动态的数据信息。这些动态数据在 Windows 系统启动时生成，包括各个设备在系统启动时的状态和运行情况。硬件驱动程序常常要对该主键进行操作。

另外，在 Windows NT 操作系统中，还存在着 HKEY_PERFORMANCE_DATA 主键，它集中管理一些实时的数据信息，比如网络驱动程序等。

在微软的 Windows2000 操作系统当中，我们只能看见 HKEY_CURRENT_USER、HKEY_CURRENT_USER 、HKEY_LOCAL_MACHINE 、HKEY_USERS 、HKEY_CURRENT_ CONFIG 五个基本主键。

18.1.2 C#对注册表编程的支持

在应用程序安装时，常常需要利用注册表来登记应用程序的名称、运行路径、用户对应用程序的配置信息等。许多应用程序在运行时也常常需要访问注册表。

.Net 框架结构在 Microsoft.Win32 名字空间内提供了两个类用于注册表操作：

Registry 和 RegistryKey。这两个类都是密封类，不允许被其它类继承。

Registry 类中提供了 7 个公有的静态域，分别代表 Windows 注册表中的 7 个基本主键，它们是：

- Registry.ClassesRoot，对应于 HKEY_CLASSES_ROOT 主键。
- Registry.CurrentUser，对应于 HKEY_CURRENT_USER 主键。
- Registry.LocalMachine，对应于 HKEY_ LOCAL_MACHINE 主键。
- Registry..Users，对应于 HKEY_USERS 主键。
- Registry.CurrentConfig，对应于 HKEY__CURRENT_CONFIG 主键。
- Registry.DynDta，对应于 HKEY_DYN_DATA 主键。
- Registry.PerformanceData，对应于 HKEY_ PERFORMANCE_DATA 主键。

RegistryKey 类中封装了对 Windows 注册表的基本操作。对注册表的操作必须符合系统提供的权限，否则不能完成指定的操作，程序将抛出一个异常。

创建子键

创建子键的成员方法的原型为：

```
public RegistryKey CreateSubKey(string subkey);
```

其中，参数 subkey 表示要创建的子键的名字或子键的全路径名。如果创建成功，返回值就是被创建的子键，否则为 null。

打开子键

打开子键的成员方法原型为：

```
public RegistryKey OpenSubKey(string name);
public RegistryKey OpenSubKey(string name, bool writable);
```

name 参数表示要打开的子键名称或全路径名，writable 参数表示被打开的主键是否可以被修改。

第一个方法对打开的子键默认是只读的，如果希望对打开的主键进行写操作，使用第二个方法，并把 writable 参数值设为 true。

名字空间 Microsoft.Win32 中还为我们提供了另一个方法，用于打开远程机器上的注册表进行操作。方法原型为：

```
public static RegistryKey OpenRemoteBaseKey( RegistryHive hKey, string machineName);
```

删除子键

DeleteSubKey 方法用于删除指定的子键，方法原型为：

```
public void DeleteSubKey (string subkey);
```

使用 DeleteSubKey 方法时，如果子键之中还包含子键，则删除失败，并返回一个异常。如果要彻底删除子键目录，即删除子键以及子键以下的全部子键，可以使用 DeleteSubKeyTree 方法。该方法原型为：

```
public void DeleteSubKeyTree(string subkey);
```

读取键值

读键的方法原型为：

```
public object GetValue(string name);
public object GetValue(string name, object defaultValue);
```

name 参数表示键的名称，返回类型是一个 object 类型。如果方法中指定的键不存在，则方法返回一个 null。我们在使用 GetValue 方法时，可以不必关心该键的值类型究竟是字符串、二进制还是 DWORD 类型，只要使用正确的返回类型就可以了。比如，我们希望读取一个字符串类型的键值，代码就可以这样写：

```
string s_value = key.GetValue("Type");
```

其中 key 表示一个主键。

如果不确定键值是否存在，而又不希望得到一个 null 返回值，那就使用第二个方法 GetValue(string name, object defaultValue)，其中的参数 defaultValue 表示默认的返回值。如果读取失败，返回值就是传递给参数 defaultValue 的值。

设置键值

设置键值的方法原型为：

```
public void SetValue(string name, object value);
```

同样，我们在使用该方法修改键值时，不用费心去分辨究竟该传递哪种值类型，方法将会识别是哪种类型，并把相应类型的值赋予指定的键。

18.1.3 注册表编程示例

示例一：修改“开始”菜单

我们知道，Windows 文件存储采用的是树型目录结构，在这个结构中，Windows 桌面代表的是最上面一层。Windows 注册表中对于桌面的设置，大都放在 HKEY_USERS 和 HKEY_CURRENT_USER 中。其中，“开始”菜单中的“运行”菜单，“查找”菜单，“设置”菜单中的“控制面板”和“打印机”，都可以通过在“HKEY_CURRENT_USER\\Software\\Microsoft\\Windows\\CurrentVersion\\Policies\\Explorer”主键下新建 DWORD 键值来屏蔽：

- “NoClose=1”：屏蔽“关闭系统”。
- “NoRun=1”：屏蔽“运行”。
- “NoFind=1”：屏蔽“查找”。
- “NoSetFolders=1”：屏蔽“设置”菜单中的“控制面板”和“打印机”。
- “NoSetTaskBar=1”：屏蔽“设置”菜单中的“任务栏和开始菜单”。
- “NoLogOff=1”：屏蔽“注销”。
- “NoRecentDocsMenu=1”：屏蔽“文档”。

图 18-2　“开始”菜单

下面的程序就实现了屏蔽“开始”菜单中的“运行”、“查找”、“设置”菜单中的“控制面板”和“打印机”。

程序清单 18-1：

```
using System;
using Microsoft.Win32;
class RegTest
{
   public static void Main()
   {
      RegistryKey key = Registry.CurrentUser;
      RegistryKey key1 = key.CreateSubKey("\\Software\\Microsoft\\
Windows\\CurrentVersion\\Policies\\Explorer");
      key1.SetValue("NoFind",1);
      key1.SetValue("NoRun",1);
      key1.SetValue("NoSetFolders",1);
      key1.Close();
   }
}
```

示例二：在“新建”中添加自己的文件类型

下面的例子在桌面的右键快捷菜单中或在 Windows 资源管理器的“新建”中添加自己的文件类型。假设添加的是扩展名为“.cs”的文件，说明文件为“cs files”，默认文件的打开方式采用 Windows 中的写字板 notepad.exe 打开。

程序清单 18-2：

```
using System;
using Microsoft.Win32;
class RegTest
{
   public static void Main()
   {
      RegistryKey key1 = Registry.ClassesRoot.CreateSubKey(".cs");
      key1.SetValue("","cs");
      RegistryKey key2 = key1.CreateSubKey("ShellNew");
      key2.SetValue("NullFile","");
      key1.Close();
      key2.Close();
      key1 = Registry.ClassesRoot.CreateSubKey("cs");
      key1.SetValue("","csharp file");
      key2 = key1.CreateSubKey("DefaultIcon");
      key2.SetValue("","c:\\windows\\notepad.exe,1");
      key2.Close();
      key2 = key1.CreateSubKey("shell\\open\\command");
      key2.SetValue("","c:\\windows\\notepad.exe");
   }
}
```

18.2 在 C #代码中调用 C++和 VB 编写的组件

如果不能和用别的编程语言编写的组件进行交互，这种编程技术的含金量就会大打折扣。.Net 环境为我们提供了不同编程语言编写的组件之间相互调用的良好机制。只要按照.Net 可操控代码的标准来编写组件，对于客户程序来说，调用的组件是哪种语言编写的都无关紧要，调用的方式实际上没有什么区别。

下面我们先后使用 C++、VB 和 C#编写了自己的组件。这是一个简化的字典组件，字典在构造时没有装载语言库，需要使用 LoadLibrary 来完成。使用 FreeLibrary 方法把语言库卸载。属性 CurrentLibrary 表示当前的语言库。在 C#中同时调用了用三种语言编写的组件，通过比较相信读者将会一目了然。

18.2.1 C++组件

先看 C++中的组件。

程序清单 18-3：

```
//DICTVC.cpp
#using <mscorlib.dll>
using namespace System;
namespace DICTVC
{
    __gc public class DictionaryComponent      //定义字典组件
    {
  private:
        String* LanguageName;
        String* AvailableLanguage[];
        public:
        DictionaryComponent ()
        {
            LanguageName = null;
            AvailableLanguage = new String*[4];
            AvailableLanguage[0] = new String(L"Chinese");
            AvailableLanguage[1] = new String(L"English");
            AvailableLanguage[2] = new String(L"German");
            AvailableLanguage[3] = new String(L"French");
  }
        bool LoadLibrary(String* language)
  {
              for(int i=0;i<4;i++){
               if (language == AvailableLanguage[i])
                      break;
              }
              if(i == 4){
                      return false;
              }
             LanguageName = language;
              return true;
        }
        void FreeLibrary(){
           LanguageName = null;
        }
```

```
        __property string* get_CurrentLibrary{
            return LanguageName;
        }
    }
}
```

首先，我们声明了一个新的名字空间，用来封装创建的新的类。

```
namespace DICTVC {…};
```

注意到，名字空间不能被嵌套，但可以被分离在多个文件中。一个简单的源代码文件也可以包含多个没有嵌套名字空间。VB 和 C#中，所有的类都是可操控的。但在C++定义的名字空间中，我们可以包含可操控的和未操控的代码，所以我们需要特别指定我们新建的类是可操控的。

```
__gc public class DictionaryComponent {…};
```

这个声明表示类将被在运行时创建，而且受控于垃圾收集器管理的堆中。我们也可以在编译时指定 /CLR 选项来保证程序中所有的类都是可操控的。

类的构造函数在每次创建类实例时都会被调用，构造函数的名称与类的名称相同，而且没有返回类型。

```
public:
    DictionaryComponent() {…}
```

而且，因为类应该是可操控的，所以我们必须显式地通知编译器数组是一个可操控的对象。因此，在指派字符串时，我们使用了修饰符__gc 。

注意：C++没有对变量进行默认的初始化的功能，必须自己手动设置 LanguageNamed 的值。

```
LanguageName = null;
AvailableLanguage = new String*[4];
```

下面是 LoadLibrary 方法，带有一个字符串类型的参数，返回一个布尔值：

```
bool LoadLibrary (String* language) {
……
    if(i == 4)
        return false;
    LanguageName = language;
        return true;
}
```

下面是 FreeLibrary 方法，没有参数，也没有返回值。

```
void FreeLibrary(){
        LanguageName = null;
    }
```

最后，我们创建了一个只读的属性 Count：

```
__property string* get_CurrentLibrary{
        return LanguageName;
```

```
}
```

编译这个 C++ 组件时的命令选项稍微有些复杂：

```
cl /CLR /c DICTVC.cpp
link -noentry -dll /out:..\Bin\DICTVC.dll DICTVC
```

这里，我们需要使用/CLR 选项来告诉编译器，生成的代码装配必须是可操控的。我们还需要一条独立的命令来执行链接（和 C#中直接编译产生目标代码不同，C++中必须经过编译和链接两个过程）。装配是指在 .NET 结构框架中能够被部署、进行版本定义、可重用的物理单元。每一个装配都建立了一个类型的集合，集合的元素包含基于运行时的类和其它的资源，它们将按照装配的指定协同工作。装配的指定说明了哪些组件是装配的一部分，哪些类型是从装配中导出的，以及类型的独立性。运行时环境将使用装配来定位和绑定你所引用到的类型。

为了方便起见，我们假设组件在当前目录的“..\Bin”子目录中被创建。为了指定组件的创建位置，我们使用/out 选项来指定输出文件的路径全名。

注意，即使我们在指定的输出文件名中包含了.dll 扩展名，我们还必须要附加的-dll 选项来告诉编译器我们将要创建的是一个 DLL，而不是一个可执行文件。

18.2.2 VB 组件

再看 VB 中组件。

程序清单 18-4：

```
'DICTVB.vb'
Option Explicit
Option Strict
Imports System
Namespace DICTVB
    Public Class DictionaryComponent
        Private AvailableLanguage (4) As String
        Private LanguageName As String
        Public Sub New()
            MyBase.New
            AvailableLanguage (0) = "Chinese"
            AvailableLanguage (1) = "English"
            AvailableLanguage (2) = "Japanese"
            AvailableLanguage (3) = "Korean"
        End Sub

        Public Function LoadBinary(ByVal language as String)
            As Boolean
            DIM i As Int
```

```
            For i = 1 To 4
                IF    language = AvailableLanguage (i) then
                      Exit For
                End IF
             Next i
              IF i=4 then
                 LoadBinary = false;
              ELSE
                 LanguageName = language;
            LoadBinary = true;
              End IF
            End Function

        Public Sub FreeLibrary()
              LanguageName = “”
            End Sub

            ReadOnly Property CurrentLibrary () As String
               Get
                  CurrentLibrary = LanguageName;
               End Get
            End Property
         End Class
      End Namespace
```

和 C++以及 C#类似，在代码中我们指定了名字空间和类的名称（老版本 VB 使用文件名来指代类的名称）。

上面的 VB 代码中声明了 Strict 选项，这是用来控制变量类型的转换是隐式的还是显式的。隐式转换不需要任何特别的语法说明，而显式转换则需要添加强制转换操作符。如果在代码中关闭了这个选项，只有一些允许的普遍类型的转换，比如说从整型到双精度型的转换，才能够隐式地进行。隐式转换和显式转换我们在书中第二部分曾进行过详细的说明。

在 VB 中，类的构造函数的名称是 New ，而不像 C#中那样，构造函数名称与类的名称相同。因为构造函数没有返回值，所以 VB 中构造函数的是作为一个过程（SUB）来实现的，而不是作为一个函数（Function）。VB 中的构造函数的形式如下：

```
Public Sub New()
    …
End Sub
```

请注意下面这个声明：

```
MyBase.New
```

这个声明是必须的，它表示调用基类中的构造函数。而在 C 和 C++中，调用基类的构造函数是由编译器自动完成的。

接下来我们看 LoadLibrary 方法。在 VB 中，有返回值的子过程称做函数。LoadLibrary 函数带有一个字符串类型的参数，返回一个布尔值。

```
Public Function LoadBinary(ByVal language as String)
    As Boolean
.............
End Function
```

过程 FreeLibrary 用于卸载字典语言库。

```
Public Sub FreeLibrary()
    LanguageName = “”
End Sub
```

最后，我们创建了一个只读的属性 Count：

```
ReadOnly Property CurrentLibrary () As String
  Get
    CurrentLibrary = LanguageName;
  End Get
End Property
```

命令行的编译十分简单，唯一的一点不同就是：为了方便，我们把输出的组件定位到“..\Bin”子目录：

```
vbc DICTVB.vb /out:..\Bin\DICTVB.dll /t:library
```

18.2.3 C#组件

为了比较起见，我们也使用 C#编写了一个类似的组件，具体的代码就不用再详细进行分析了。

程序清单 18-5：

```
//DICTCS.cs
using System;
namespace DICTCS
{
 public Class DictionaryComponent
 {
    private string LanguageName;
    private string[] AvailableLanguage = new string[4];;
    public DictionaryComponent ()
    {
            AvailableLanguage[0] = "Chinese";
            AvailableLanguage[1] = "English";
```

```
            AvailableLanguage[2] = "German";
            AvailableLanguage[3] = "French";
        }
            public bool LoadLibrary(String language)
        {
                for(int i=0;i<4;i++){
                    if (language == AvailableLanguage[i])
                        break;
                }
                if(i == 4)
                    return false;
                 LanguageName = language;
                 return true;
            }
            public void FreeLibrary(){
                LanguageName = null;
            }
            public string CurrentLibrary{
                get{
                return LanguageName;
                }
        }
    }
}
```

这里，编译的命令比我们常用到的稍微复杂了一些：

```
csc /out:..\Bin\DICTCS.dll /target:library DICTCS.cs
```

和 C++ 一样，我们使用/out 选项把编译后的组件输出到当前目录的“..\Bin”子目录。同样，我们也需要使用 /target:library 编译选项来告诉编译器创建一个 DLL。

18.2.4 C# 中的客户程序

程序清单 18-6：

```
//ClientCS.cs
using System;
using DICTVC;
using DICTCS;
using DICTVB;

class Test
```

```
{
public static void Main()
{
  Console.Write("Input the language name of library to load:”);
  string language = Console.ReadLine();
//调用 C#中的组件
   DICTCS.DictionaryComponent CsDict = new
         DICTCS. DictionaryComponent();
   Console.WriteLine("Dictionary from C# DictionaryComponent");
   if(CsDict.LoadLibrary(language)==false){
         Console.WriteLine("Library not found in C# DictionaryComponent!”);
      }
      else{
         Console.WriteLine("Library load successfully in C# DictionaryComponent”);
      }
  //调用 C++中的组件
      DICTVC. DictionaryComponent VcDict = new
      DICTVC. DictionaryComponent ();
      Console.WriteLine("\n Dictionary from C ++
            DictionaryComponent");
      if(VcDict.LoadLibrary(language)==false){
      Console.WriteLine("Library not found in C++ DictionaryComponent!”);
      }
      else{
         Console.WriteLine("Library load successfully in C++ DictionaryComponent”);
      }
  //调用 VB 中的组件
      DICTVB. DictionaryComponent VbDict = new
      DICTVB. DictionaryComponent ();
      Console.WriteLine("\n Dictionary from VB
            DictionaryComponent");
      if(VbDict.LoadLibrary(language)==false){
         Console.WriteLine("Library not found in VB DictionaryComponent!”);
      }
      else{
      Console.WriteLine("Library load successfully in VB DictionaryComponent”);
      }
      //显示各个字典中的语言信息
      Console.WriteLine(“The language of libiary in
```

```
            C# dictionary is: {0}",CsDict.CurrentLibrary);
        Console.WriteLine("The language of libiary in
            C++ dictionary is: {0}",VcDict.CurrentLibrary);
        Console.WriteLine("The language of libiary in
            VB dictionary is: {0}",VbDict.CurrentLibrary);
        //卸载字典语言库
        CsDict.FreeLibrary();
        VcDict.FreeLibrary();
        VbDict.FreeLibrary();
    }
}
```

在上面的例子中，我们引用了名字空间的全名，因为定义的各个组件名字均为DictionaryComponent。如果不使用名字空间的全名，对组件的使用将产生混淆。如果你在声明组件时采用下面的语法，我们就不需要引入全名：

```
using VC DictionaryComponent = CompVC.DictionaryComponent;
using CS DictionaryComponent = CompCS.DictionaryComponent;
using VB DictionaryComponent = CompVB.DictionaryComponent;
```

事实上，如果我们在 C++中调用 C#或者 VB 编写的组件，除了一些范围的指定外，客户程序的代码基本上没有什么两样。

编译该 C#客户程序也十分简单，现在让我们把应用程序的可执行文件同样输出到当前目录的“..\Bin”子目录，这只需要使用/reference 编译选项就可以了：

```
csc /reference:..\Bin\DICTCS.dll;..\Bin\DICTVB.dll;
..\Bin\DICTVC.dll /out:..\Bin\ClientCS.exe ClientCS.cs
```

下面让我们检验一下程序的运行效果，在屏幕上键入 Client.exe 命令，并按提示输入语言名称：

```
Input the language name of library to load:
```

如果我们输入“English”，程序运行的结果将是：

```
Dictionary from C# DictionaryComponent
Library load successfully in C# DictionaryComponent
Dictionary from VC DictionaryComponent
Library load successfully in VC DictionaryComponent
Dictionary from VB DictionaryComponent
Library load successfully in VB DictionaryComponent
The language of library in C# dictionary is: English
The language of library in VC dictionary is: English
The language of library in VB dictionary is: English
```

再运行程序，这一次我们输入“Japanese”，程序运行的结果将是：

```
Dictionary from C# DictionaryComponent
Library not found in C# DictionaryComponent!
```

```
Dictionary from VC DictionaryComponent
Library not found in VC DictionaryComponent!
Dictionary from VB DictionaryComponent
Library load successfully in VB DictionaryComponent
The language of library in C# dictionary is:
The language of library in VC dictionary is:
The language of library in VB dictionary is: Japanese
```

18.3 版 本 控 制

版本控制（***Versioning***）主要是为了解决组件的版本不兼容的问题。版本兼容的方式有：

- 源代码级兼容：依赖于旧版本的代码在重新编译之后能够与新版本兼容。
- 二进制级兼容：依赖于旧版本的应用程序无需重新编译就能与新版本兼容。

大多数语言根本不支持二进制级的版本兼容，其中许多在源代码级的兼容问题上也表现的不尽如人意。事实上，许多语言由于自身的缺陷，不改写客户代码就不可能实现组件的升级。

举个例子，假设基类的作者写了一个叫 Base 的类。在第一个版本中，类 Base 中没有包含方法 F。一个叫 Derived 的类从 Base 中继承，并且声明了一个方法 F。类 Derived 和类 Base 一同被交付给客户使用，并且配置到很多客户机和服务器上。

```
// Author A
namespace A
{
  public class Base         // version 1
  {
  }
}
// Author B
namespace B
{
  class Derived: A.Base
  {
        public virtual void F() {
              System.Console.WriteLine("Derived.F");
        }
  }
}
```

到目前为止，程序的运行一切正常。然后，Base 类的作者提供了一个新版本，给类 Base 添加了一个方法 F。

```
// Author A
namespace A
{
  public class Base         // version 2
  {
        public virtual void F() {              // added in version 2
              System.Console.WriteLine("Base.F");
        }
  }
}
```

新版本的 Base 应该和旧版本保持源代码级兼容和二进制级兼容。不幸的是，类 Base 中的新方法与类 Derived 中的 F 产生了混淆。Derived 应该重载 Base 中的 F 吗？看上去不应该，因为 Derived 已经被编译，那时 Base 中甚至还没有 F！但是，如果 Derived 中的 F 不重载 Base 中的 F，而它又必须符合基类 Base 的声明，可是在写 Derived 类时还不存在该声明。比如某种情况下，Base 中的 F 可能要求被重载。

在解决版本问题时，C#要求开发人员清楚地表达他们的意图。在原始代码中，类 Base 不包括方法 F，所以不存在什么问题。Derived 中的 F 是作为一个新方法，而不是重载基类中的方法。

```
// Author A
namespace A
{
  public class Base
  {
  }
}
// Author B
namespace B
{
  class Derived: A.Base
  {
        public virtual void F() {
              System.Console.WriteLine("Derived.F");
        }
  }
}
```

一旦类 Base 添加了方法 F 并且提交了新版本，Derived 的二进制版本的意图也是很清楚的：方法 F 是语义独立的，不会被作为重载来对待。但是，如果 Derived 被重新

编译，这就产生了混淆，编译器将给出一个警告，并且在默认情况下，Derived 中的 F 覆盖了 Base 中的 F。编译器的警告用于提示 Derived 的作者主要在 Base 中方法 F 的存在。如果 Derived 中的 F 和 Base 中的 F 根本不存在什么关系，Derived 的作者可以使用 new 关键字声明自己的意图，从而关闭警告。

```
// Author A
namespace A
{
  public class Base                // version 2
  {
        public virtual void F() { // added in version 2
              System.Console.WriteLine("Base.F");
        }
  }
}
// Author B
namespace B
{
  class Derived: A.Base  // version 2a: new
  {
        new public virtual void F() {
              System.Console.WriteLine("Derived.F");
        }
  }
}
```

另一种情况是，经过对程序应用环境的调查，Derived 的作者决定让 F 重载 Base 中的方法 F 更为合适。这时他就应该给 Derived 中的 F 加上 override 关键字。

```
// Author A
namespace A
{
  public class Base                          // version 2
  {
        public virtual void F() { // added in version 2
              System.Console.WriteLine("Base.F");
        }
  }
}
// Author B
namespace B
{
```

```
class Derived: A.Base  // version 2b: override
{
    public override void F() {
        base.F();
        System.Console.WriteLine("Derived.F");
    }
}
```

当然 Derived 的作者还有另外一个选择，就是改变方法 F 的名字，以从根本上避免名字冲突。这种做法打破了 Derived 的源代码级兼容和二进制级兼容。这就得看具体的情况了，如果 Derived 类不导出到其它的程序，改名不失为一个好办法，它提高了程序的可靠性，对应方法 F 的含义不会再出现任何混淆。

18.4 代 码 优 化

在编写基于.NET 架构的应用程序，尤其是分布式时，我们常常极为关注应用程序的执行效率。如果掌握了如何编写高效率的代码，就能大幅度地提高应用程序的执行速度，并有助于减少应用程序的瓶颈（bottlenecks）。

18.4.1 撤消

我们知道，类 object 是 .NET 架构中的其它一个类的基类。类 object 是在 System 中定义的，它并没有声明析构函数，而是定义了一个保护类型的成员方法 Finalize。

如果.NET 的运行时垃圾收集器认为一个对象可以安全从内存中移出时，垃圾收集器就调用该对象的撤消方法 Finalize，把对象移出内存，释放占用的系统资源。我们应该把对一些重要性能的考虑牢记在脑海里。

如果你有 C++的开发背景，你也许会倾向于采用对象的析构函数来撤消对象。不过当你在 .NET 环境下编程，你就不能再试图依靠析构函数来执行对象的撤消。

对象的撤消方法 Finalize 会对程序的性能产生下列不良影响：

自己拥有撤消方法的对象在释放资源时将耗费更长的时间。

垃圾收集器并不按照一定的顺序来撤消对象，也并不保证每一个对象的撤消方法都能被正确的调用。

如果本应该撤消的对象引用了另一个暂时还不能撤消的对象，这个对象也不能撤消。

如果同时有大量的对象在等待撤消，这将会极为耗费系统资源，并降低系统性能。

每个需要清除的对象都必须执行撤消。为了优化性能，在必须使用 Finalize 方法时，你可以重载一个 Close 方法，当你需要清除某个对象时，你就可以调用 Close 方法，从而强迫垃圾收集器调用撤消方法，把该对象设为 null。调用 GC.SupressFinalize（）方

法（GC 是 System 中提供的一个类），可以为你代码中的元数据设置一个标记，告诉运行时 GC 不要撤消这个类。这样，你就可以在对象已没有用处时立即动手释放它。

下面的代码展示了如何设计一个重载 Close 方法的类：

```
public class MyClass{
    public override void close(){
        GC.SupressFinalize(this)
    }
    protected overide void Finalize(){
        close();
    }
}
```

18.4.2 事务

当可操控的代码必须要和未操控的代码进行交互时，事务（transition）就发生了。这通常出现在当你需要平台调用服务（Platform Invocation Services，PInvoke）来访问未操控的动态链接库的静态指针入口，或者是访问 COM 提供的其它方法时。

每一次事务都会带来少量的开销，据估计每调用一次事务大约要执行 10 到 40 条指令。因此，最好的编程习惯是在代码中尽量少调用事务。如果情况必须，那就谨慎地使用事务。在使用 API 函数时，应尽可能地每次执行多个动作，而不是重复调用，但每次只执行少量的动作。

18.4.3 值类型

公共语言环境支持两种类型：值类型和引用类型。值类型表示在内存中占据实际的数据位数，引用类型只表示数据在内存中的位置。我们知道，在运行时，对象类型、接口类型、指针类型都被作为引用类型来对待，而其它的主要类型都被定义为值类型。

究竟使用哪种类型，要看具体情况而定。

在一些情况下，值类型在性能上更有优势。如果对象被分配 GC 堆中，这时值类型就被分配在栈中，从而具有更快的运行速度。这是因为它们没有与类相关联的开销，所以不需要调用类的构造函数。另外，值类型的成员通常会被自动初始化为默认值，一般是 0 或 null。你可以通过从 System.ValueType 中派生来定义自己的值类型。

18.4.4 字符串

字符串在运行时是不可变的。如果我们修改了字符串中的数据，实际上我们不是在修改原来的字符串，而是创建了字符串的一个新的实例。为了避免这种情况的发生，我们可以使用 the System.Text.Stringbuilder 类来创建一个 StringBuilder 对象，这样就可以避免创建新的对象实例，而只是修改原来的对象。

在下面的例子中，字符串 MyString1 和 MyString2 相连接，实际上我们建立了第三

个字符串对象，把连接后的值“Please enter your name”赋予了它。这样就降低了代码的性能，因为我们创建了新的对象，分配了新的空间，而已经分配的空间实际上是浪费了。

```
string MyString1 = "Please ";
string MyString2 = "enter your name";
MyString1 = string.Concat(MyString, MyNewString);
```

如果我们采用 StringBuilder 类，就可以解决上面的例子中的性能问题。StringBuilder 不会创建新的对象。看改进后的代码：

```
StringBuilder MyStringBuilder = new StringBuilder ("Please ");
String MyNewString = "enter your name";
MyStringBuilder.Append(MyNewString);
```

18.4.5 内联

内联表示在编译时，在对每一个方法的调用处都加上实际的方法代码，而不是只包含对该方法的引用。这样做的结果是虽然增加了输出文件的长度，但是由于减少了对方法调用的开销，从而加快了程序的运行速度。

声明内联方法是在方法的定义中加上 inline 关键字。

在.NET 环境中使用内联方法可以减少对虚方法的使用，降低系统的开销。同样，我们也推荐尽量使用密封类和密封方法。

18.5 小　　结

本章讨论了一些 C#编程中较为复杂的问题，提供了内容丰富的各种编程经验。如果你希望了解 C#深入编程的关键技术，仔细阅读本章的内容一定会大有益处。

本章介绍了以下内容：

- 如何使用 C#进行注册表编程。
- 如何在 C#源程序中调用 VB 和 C++语言编写的组件。
- 如何利用名字空间技术实现版本控制。
- 如何提高 C#程序的执行效率。

复习题

（1）简述 Windows 操作系统中注册表所起的作用，以及注册表的结构。

（2）如何通过编程在注册表中生成一个新键和删除一个键？

（3）C#客户程序调用其它语言编写的组件与调用 C#组件有什么区别？

（4）简述版本控制对于应用程序的作用何在。

（5）有哪几种途径可以提高 C#应用程序的执行效率？

第五部分　附　　录

附录 A　关　键　字

关键字是系统预定义的保留字。编译器在扫描源程序时，遇到关键字将作出专门的解释，负责执行特定的任务。我们也可以认为关键字是语句的一部分。我们不能用关键字来定义各种类型的名称。不过，C#允许我们使用关键字前面符号@来作为自定义的名称。比如，我们不允许采用 as，但可以采用@as 来作为一个变量声明的名称。下面列出了 C#中定义的关键字：

abstract	enum	long	stackalloc
as	event	namespace	static
base	explicit	new	string
bool	extern	null	struct
break	false	object	switch
byte	finally	operator	this
case	fixed	out	throw
catch	float	override	true
char	for	params	try
checked	foreach	private	typeof
class	goto	protected	uint
const	if	public	ulong
continue	implicit	readonly	unchecked
decimal	in	ref	unsafe
default	int	return	ushort
delegate	interface	sbyte	using
do	internal	sealed	virtual
double	is	short	void
else	lock	sizeof	while

附录 B　错　误　码

错误 CS0001　编译器内部错误
错误 CS0003　内存溢出
错误 CS0004　提升为错误的警告
错误 CS0005　编译器选项后应跟正确的参数
错误 CS0006　找不到动态链接的元数据文件
错误 CS0007　.Net 环境初始化错误
错误 CS0008　从文件中读取元数据错误
错误 CS0009　不能打开元数据文件
错误 CS0010　名字空间与类型的声明不能同名
错误 CS0011　引用的类找不到基类
错误 CS0012　找不到引用类型的定义
错误 CS0013　向文件中保存元数据时发生未知错误
错误 CS0014　找不到文件名
错误 CS0015　类型名太长
错误 CS0016　无法输出文件
错误 CS0017　输出文件有多个入口
错误 CS0019　双目操作符不适用于指定的操作数
错误 CS0020　除数为 0
错误 CS0021　不能对类型表达式使用索引
错误 CS0022　错误的多重索引
错误 CS0023　单目操作符不适用于指定的操作数
错误 CS0024　无法打开源文件
错误 CS0025　找不到标准库文件
错误 CS0026　在静态属性、方法、域初始化时使用了非法的关键字
错误 CS0027　当前环境中关键字无效
警告（4 级）CS0028　函数声明带有错误的入口标识
错误 CS0029　类型之间无法进行隐式转换
错误 CS0030　类型之间无法进行转换
错误 CS0031　常数值不能转换为类型值
错误 CS0032　无法打开渐增编译文件
错误 CS0033　写渐增编译文件时发生磁盘 I/O 错误
错误 CS0034　双目操作符对指定的操作数存在二义性
错误 CS0035　单目操作符对指定的操作数存在二义性
错误 CS0036　输出参数不能使用[in]特征

错误 CS0037　不能给值类型赋 null 值
错误 CS0038　非静态的类型成员不能被嵌套的类型访问
错误 CS0040　调试初始化产生错误
错误 CS0041　保存调试信息错误
错误 CS0042　创建调试信息文件错误
错误 CS0043　PDB 文件格式错误，将删除文件并重新编译
错误 CS0050　方法返回类型的访问权限低于方法的访问权限
错误 CS0051　方法参数类型的访问权限低于方法的访问权限
错误 CS0052　域类型的访问权限低于域的访问权限
错误 CS0053　属性类型的访问权限低于属性的访问权限
错误 CS0054　索引指示器返回类型的访问权限低于索引指示器的访问权限
错误 CS0055　索引指示器参数类型的访问权限低于索引指示器的访问权限
错误 CS0056　操作符返回类型的访问权限低于操作符的访问权限
错误 CS0057　操作符参数类型的访问权限低于操作符的访问权限
错误 CS0058　代表类型的访问权限低于代表的访问权限
错误 CS0059　代表类型的访问权限低于代表的访问权限
错误 CS0060　基类的访问权限低于派生类的访问权限
错误 CS0061　父接口的访问权限低于子接口的访问权限
错误 CS0065　事件属性必须同时有两个访问器
错误 CS0066　事件必须是代表型
错误 CS0067　事件在声明的类中从未使用过
错误 CS0068　接口中的事件不能有初始化
错误 CS0069　接口中的事件不能有访问器
错误 CS0070　事件不能出现在操作符“+=”或“－=”左边
错误 CS0071　事件的显示接口执行体必须按照属性的语法格式
错误 CS0075　强制转换负数时应使用括号
错误 CS0076　保存的枚举器名称不能被使用
错误 CS0077　as 操作符必须同引用类型一起使用
错误 CS0100　参数名字重复
错误 CS0101　名字空间中包含了同名的类型
错误 CS0102　类中已经包含了指示符的定义
错误 CS0103　名称在类或名字空间中不存在
错误 CS0104　引用存在二义性
错误 CS0105　名字空间使用指示符在名字空间中已经出现过
错误 CS0106　修饰符对当前项目无效
错误 CS0107　多个访问修饰符
警告（1 级）CS0108　成员覆盖了继承的同名成员，应使用 new 修饰符
警告（4 级）CS0109　成员未覆盖了继承的同名成员，不应使用 new 修饰符
错误 CS0110　常量之间循环定义

错误 CS0111　具有相同参数类型的方法已被定义过

错误 CS0112　静态成员方法不能使用 override、virtual 或 abstract 修饰符

错误 CS0113　重载成员方法不能使用 new、virtual 或 abstract 修饰符

警告（2 级）CS0114　方法覆盖了继承的同名方法，如果要重载，使用 override 修饰符，否则使用 new 修饰符

错误 CS0115　找不到合适的方法进行重载

错误 CS0116　名字空间不能直接包含域或方法等成员

错误 CS0117　类型中不能包含函数定义

错误 CS0118　构造函数名称应指代另一构造函数

错误 CS0119　构造函数名称指代的构造函数在当前环境中无效

错误 CS0120　非静态的域、方法和属性成员要求对象引用

错误 CS0121　方法之间的调用存在二义性

错误 CS0122　因保护级别不能访问成员

错误 CS0123　方法声明与代表的类型不符

错误 CS0126　需要一个可转换的类型

错误 CS0127　返回值为 void 类型的方法应在表达式后使用 return 语句

错误 CS0128　当前范围内已定义了局部变量

错误 CS0131　语句的左边应为一个变量、属性或索引指示器

错误 CS0132　静态构造函数不能有参数

错误 CS0133　表达式中的变量应为常量

错误 CS0134　在嵌套名字空间中不能使用全权名

错误 CS0135　声明之间相互冲突

错误 CS0136　局部变量的名称在当前范围内不能使用，该名称已有其它含义

错误 CS0138　名字空间使用指示符仅对名字空间有效

错误 CS0139　break 或 continue 语句没有跳出的范围

错误 CS0140　标签重复

错误 CS0143　未对类定义构造函数

错误 CS0144　对抽象类或接口不能建立实例

错误 CS0145　对常量域应赋值

错误 CS0146　基类定义发生循环

错误 CS0148　代表没有有效的构造函数

错误 CS0149　应使用方法名称

错误 CS0150　应使用常量值

错误 CS0151　应使用整数类型

错误 CS0152　switch 语句中已经存在的标签

错误 CS0153　goto 语句只在 switch 语句中有效

错误 CS0154　因为缺少读访问器，属性或索引指示器不能使用

错误 CS0155　catch 或 throw 的类型应该从 System.Exception 中继承

错误 CS0156　throw 语句不能在 catch 子句以外使用

错误 CS0157　无法离开 finally 子句
错误 CS0158　在包含的范围内标签屏蔽了另一个标签
错误 CS0159　goto 语句中标签不存在
错误 CS0160　前面的 catch 子句已经捕获了所有的意外
错误 CS0161　方法中不是所有的路径都能返回值
错误 CS0162　检测到执行不到的代码
错误 CS0163　控制无法从一个 case 语句转入另一个 case 语句
错误 CS0164　标签未被引用
错误 CS0165　可能使用未被赋值的局部变量
错误 CS0167　代表缺少调用方法
警告（3 级）CS0168　声明的变量未被使用
警告（3 级）CS0169　保护域成员未被使用
错误 CS0170　可能使用未被赋值的域
错误 CS0171　在构造函数结束之前应给域赋值
错误 CS0172　因类型之间可以相互进行隐式转换，无法确定表达式的类型
错误 CS0173　因类型之间不存在隐式转换，无法确定表达式的类型
错误 CS0174　base 引用需要一个基类
错误 CS0175　此处不能使用 base 关键字
错误 CS0176　在类的实例中无法访问静态成员，应使用类型名
错误 CS0177　方法结束之前需要给输出参数赋值
错误 CS0178　数组初始化结构错误
错误 CS0179　外部方法含有执行体
错误 CS0180　成员不能同时是外部的和抽象的
错误 CS0181　未知特征
错误 CS0182　特征参数必须为常量、方法、域、属性或类
警告（1 级）CS0183　表达式总是提供的类型
警告（1 级）CS0184　表达式总是不提供的类型
错误 CS0185　类型不是 lock 语句中需要的引用类型
错误 CS0186　当前环境中不能使用 null
错误 CS0187　操作符未对此类型作出定义
错误 CS0188　在所有域被赋值之前不能使用实例
错误 CS0190　_arglise 型构造函数仅在使用变量的方法中有效
错误 CS0191　不能给只读域赋值
错误 CS0192　只读域不能为 ref 或 out 型
错误 CS0193　或操作符只能对指针使用
错误 CS0196　一个指针只能指向一个值
错误 CS0198　静态的只读域不能被赋值
错误 CS0199　静态的只读域不能为 ref 或 out 型
错误 CS0200　只读的属性、索引指示器属性不能被赋值

错误 CS0201　不能作为语句使用
错误 CS0202　对 GetEnumerrator 的调用必须返回类或结构
错误 CS0203　方法不能作为变量使用
错误 CS0204　局部变量的取值范围不能超过 65536
错误 CS0205　不能调用基类的方法或属性
错误 CS0206　属性、索引指示器不能作为 ref 或 out 型传递
错误 CS0207　访问器不能被声明为 unsafe
错误 CS0208　对可控类型不能取地址或长度
错误 CS0209　在 fixed 语句中声明的局部变量必须为指针类型
错误 CS0210　fixed 语句中声明时必须初始化
错误 CS0211　不能对表达式取地址
错误 CS0212　在 fixed 语句中的初始化时不能取非 fixed 型表达式的地址
警告（3 级）CS0213　不需要使用 fixed 语句来取 fixed 型表达式的地址
错误 CS0214　指针必须在 unsafe 环境中使用
错误 CS0215　true 或 false 操作符必须为 bool 类型
错误 CS0216　操作符需要一个已定义的操作符与之匹配
错误 CS0217　用户自定义操作符必须与参数有相同的返回类型
错误 CS0218　类型必须包含操作符 true 和 false 的声明
警告（3 级）CS0219　变量已赋值但未使用
警告（3 级）CS0220　checked 模式下编译时操作符溢出
错误 CS0221　常值不能转换为类型（可用 unchecked 语法）
错误 CS0223　不允许空字符串 case 标号
错误 CS0224　方法不能同时有 paras 参数和 varargs
错误 CS0225　paras 参数必须是一维数组
错误 CS0227　unsafe 编译时才能出现非操控代码
错误 CS0228　类型不包含成员定义或定义不可达
错误 CS0229　成员间定义模糊
错误 CS0230　foreach 语句缺少类型和标识符
错误 CS0231　params 参数表和-arglist 参数不能有下列变量
错误 CS0233　sizeof 只能用于 unsafe 模式
错误 CS0234　类名或名字空间名未定义
错误 CS0235　代表中不能有-arglist
错误 CS0236　域初始化不能引用非静态的域、方法或属性
错误 CS0500　抽象类成员不能声明实体
错误 CS0501　非抽象或外部的成员函数必须声明实体
错误 CS0502　类是抽象且封闭的
错误 CS0503　抽象方法不能标记为虚
错误 CS0504　常变量不能标记为静态
错误 CS0505　成员不能覆盖继承的非函数成员

错误 CS0506　函数不能覆盖继承的函数，因函数不是 virtual、abstract 或 override 的性质

错误 CS0507　覆盖继承成员函数 2 时，函数 1 不能改变访问调节器

错误 CS0508　函数 1 覆盖继承成员函数 2 时不能改变返回类型

错误 CS0509　不能从封口类型继承

错误 CS0513　抽象类包含于非抽象类中

错误 CS0514　静态构造函数不能有外部 this 指针和基构造函数调用

错误 CS0515　静态构造函数不允许访问调节器

错误 CS0516　构造函数不能调用自身

错误 CS0517　无基类不能调用基类构造函数

错误 CS0518　未定义或引入预定义类型

错误 CS0519　名字与预定义名字空间冲突

错误 CS0520　名字与预定义类型冲突

错误 CS0522　结构不能调用基类构造函数

错误 CS0523　结构成员互相引用构成圈

错误 CS0524　接口不能声明类型

错误 CS0525　接口不能包含域

错误 CS0526　接口不能包含构造函数

错误 CS0527　接口类型表中有非接口类型

错误 CS0528　接口已处于接口列中

错误 CS0529　接口互相继承构成圈

错误 CS0531　接口成员不能有定义

错误 CS0533　派生类成员隐藏了继承的基类抽象成员

错误 CS0534　方法没有执行继承的抽象方法

错误 CS0535　类没有执行接口成员

错误 CS0536　类没有执行接口成员，类成员是静态、非公有或返回类型错误

错误 CS0538　外部接口声明中的名字不是接口

错误 CS0539　外部接口声明中的成员不是接口成员

错误 CS0540　包含类型不执行接口

错误 CS0541　外部接口不能在类和结构中声明

错误 CS0542　用户自定义成员名不能与包含类型相同

错误 CS0543　计数器的值过大

错误 CS0544　属性不能覆盖继承的非属性

错误 CS0545　函数不能覆盖，因为没有可覆盖属性

错误 CS0546　不能覆盖，因为没有可覆盖属性

错误 CS0547　属性和索引器不能有 void 类型

错误 CS0548　属性和索引器必须至少有一个存取程序

错误 CS0549　函数是封口类的新虚成员

错误 CS0550　加入了一个接口成员找不到的访问程序

错误 CS0551　外部接口执行缺少访问程序
错误 CS0552　不允许用户自定义与接口有关的转换
错误 CS0553　不允许用户自定义与基类有关的转换
错误 CS0554　不允许用户自定义与派生类有关的转换
错误 CS0555　不允许用户自定义包含类型之间的转换
错误 CS0556　用户自定义转换必须以包含类型为源类型或目标类型
错误 CS0557　类中不能出现重复的用户自定义转换
错误 CS0558　用户自定义操作符必须是静态或公有
错误 CS0559　++和-操作符的返回类型和参数必须是包含类型
错误 CS0560　访问程序 1 不能覆盖被函数隐藏的访问程序 2
错误 CS0561　函数不能覆盖由专门编译器产生的方法
错误 CS0562　unary 操作符的参数必须是包含类型
错误 CS0563　二进制操作符的一个参数必须是包含类型
错误 CS0567　接口不能包含操作符
错误 CS0568　结构不能包含外部无参数构造函数
错误 CS0569　方法不能覆盖非方法
错误 CS0570　类引用了语言不支持的类型
错误 CS0571　函数不能在外部调用操作符和访问器
错误 CS0572　不能通过表达式引用类型，请尝试 path-to-type
错误 CS0573　结构中不能有域初始化实例
错误 CS0574　析构函数名必须与类名匹配
错误 CS0575　只有类类型可以有构造函数
错误 CS0576　名字系统已有 system 的别名
错误 CS0577　构造函数、析构函数、操作符和外部接口执行的 condictional 属性无效
错误 CS0578　返回类型为空的函数 condictionak 属性无效
错误 CS0579　属性重复
错误 CS0580　太多未命名变量定义了属性
错误 CS0581　该属性不适用于命名变量
错误 CS0582　接口成员的 conditional 属性无效
错误 CS0583　内部编译错，请用/bugreport 选项创建故障报告并提交给问题报告处理人
错误 CS0584　内部编译错；阶段符号
错误 CS0585　内部编译错；阶段
错误 CS0586　内部编译错；阶段
错误 CS0587　内部编译错；阶段
错误 CS0588　内部编译错；LEX 阶段
错误 CS0589　内部编译错；PARSE 阶段
错误 CS0590　用户自定义操作符不能返回 void 类型

错误 CS0591　变量属性包含非法值
错误 CS0592　属性赋给错误的类型
错误 CS0594　浮点常量越界
错误 CS0595　只能在空类型中指定该属性
错误 CS0596　guid 属性必须于 comimport 属性一起指定
错误 CS0597　该属性缺少未命名变量
错误 CS0598　无效的 guid 特征参数，该参数的变量必须是形式为 **XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX** 的常字符串
错误 CS0599　属性隶属于非法命名变量
错误 CS0601　dllimport 属性必须在静态或外部方法中指定
警告（1 级）CS0602　不赞成使用旧特性，请使用新特性
错误 CS0608　方法不能同时标记为 dllimport 和 sysnative
错误 CS0609　不能在标记为 override 的索引器中建立同样的属性
错误 CS0610　域不能是 System.TypedReference 类型
错误 CS0611　数组元素不能是 System.TypedReference 类型
错误 CS0612　成员已作废
错误 CS0614　不能定义与 system.variant 有关的用户自定义转换
错误 CS0616　不是属性类型
错误 CS0617　试图访问私有成员失败
警告（1 级）CS0618　成员已作废
错误 CS0619　成员已作废
错误 CS0620　索引器不能有 void 类型
警告（1 级）错误 CS0621　抽象成员和虚成员不能为私有
错误 CS0622　不能用数组初始化表达式给非数组类型初始化
错误 CS0623　数组初始化只能用于域或变量的初始化
错误 CS0624　returnshresult 属性只能位于 dllimport 或成员为 COM 标准接口的方法中
错误 CS0625　标记有 structlayout 的域类型实例必须有 structoffset 属性
错误 CS0626　没有标记 dllimport 或 sysnative 的外部方法不能执行
错误 CS0627　非外部方法不能有 sysnative 属性
错误 CS0628　封口类定义了新保护成员
错误 CS0629　接口执行过程中不能使用 conditional 属性
错误 CS0630　联合的成员是类或引用类型
错误 CS0631　索引器不能是 ref 或 out 参数
错误 CS0632　命名的属性变量不能是只读类型
错误 CS0633　传递给名字属性的变量必须是有效的标识符
错误 CS0634　变量仅对 System.Interop.UnmanagedType.CustomMarshaller 类型有效
错误 CS0635　System.Interop.UnmanagedType.CustomMarshaller 需要命名变量

ComType 和 Marshal

错误 CS0636　structoffset 属性只能位于标记有 structlayout 的类型中

错误 CS0637　静态域和常域不允许 structlayoff 属性

错误 CS0638　未定义全局标识符

错误 CS0640　生成集合前不允许建立全局属性集

错误 CS0641　该属性只有在 System.Attribute 的派生类中有效

警告（3 级）CS0642　空语句可能导致错误

错误 CS0643　属性变量重复命名

错误 CS0644　类不能从基类继承

错误 CS0645　标识符过长

错误 CS0646　包含索引指示器的类型不能指定 System.Reflection.DefaultMember Attribute

错误 CS0647　发表属性出错

错误 CS0648　语言不支持的类型

警告（3 级）CS0649　域未分配值，将始终用缺省值

错误 CS0650　数组声明符语法错

错误 CS0651　语法错，全局属性必须是文件范围

错误 CS0652　常量越界

错误 CS0653　不能应用抽象的属性类

错误 CS0654　无变量表的方法引用

错误 CS1001　需要标识符

错误 CS1002　需要“;”

错误 CS1003　语法错误，缺少字符

错误 CS1004　双重修饰符

错误 CS1005　非法间接引用类型

错误 CS1007　属性访问器已定义

错误 CS1008　缺少 byte, sbyte, short, ushort, int, uint, long, 或 ulong 类型

错误 CS1009　无法识别的脱离顺序

错误 CS1010　串没有正确定界

错误 CS1011　字符赋值为空

错误 CS1012　多字符赋值给同一字符变量

错误 CS1013　无效数字

错误 CS1014　缺少 get 或 set 方法

错误 CS1015　需要一个对象、串或类类型

错误 CS1016　需要命名的属性变量

错误 CS1017　try 语句已经包含一个空 catch 块

错误 CS1018　需要关键字 this 或 base

错误 CS1019　需要可重载的 unary 操作符

错误 CS1020　需要可重载的二进制操作符

错误 CS1021　整数常量越界
错误 CS1022　缺少类或名字空间定义或文件尾
错误 CS1023　内嵌语句不能是声明或标号语句
错误 CS1024　需要预处理器指令
错误 CS1025　缺少单行注释或行尾标志
错误 CS1026　缺少“)”
错误 CS1027　缺少#endif 指令
错误 CS1028　不该有的预处理器指令
错误 CS1029　#error：文字
警告（1 级）CS1030　#warning：文字
错误 CS1031　缺少类型定义
错误 CS1032　必须在文件开始处#define 或 #undef 预处理器符号
错误 CS1033　编译器限制越界：文件不能超过最大行数限制
错误 CS1034　编译器限制越界：行不能超过最大字符数限制
错误 CS1035　找到文件尾标记，需要“*/”
错误 CS1036　缺少“(”或“ .”
错误 CS1037　缺少可重载操作符
错误 CS1038　缺少#endregion 指令
错误 CS1039　无法终止的字符串
错误 CS1040　预处理器指令必须位于行首第一个非空字符处
错误 CS1041　缺少标识符
错误 CS1042　关键字不能包含统一代码逃逸顺序
错误 CS1501　没有对带有数值类型参数的方法的重载
错误 CS1502　最佳的重载方法声明中含有无效的参数
错误 CS1503　无法对方法的参数进行类型转换
错误 CS1504　无法打开源文件
错误 CS1505　编译选项缺少文件说明
错误 CS1506　输出文件不在装配目录下
错误 CS1507　建立模块时无法链接源文件
错误 CS1508　装配中已使用了源标识符
错误 CS1509　引用的文件不是一个装配，换用选项/add module
错误 CS1510　ref 或 out 类型参数必须为一个值
错误 CS1511　关键字 base 在静态方法中无效
错误 CS1512　关键字 base 在当前环境中无效
错误 CS1513　缺少“}”
错误 CS1514　缺少“{”
错误 CS1515　源文件被包含多次
错误 CS1516　源文件被指定多次
错误 CS1517　无效的处理器表达式

错误 CS1518　缺少类、代表、枚举、接口、结构或联合
错误 CS1519　使用类、结构或接口成员的声明无效
错误 CS1520　类、结构或接口的方法应有返回值
错误 CS1521　无效的基类型
警告（1 级）CS1522　switch 语句块为空
警告（1 级）CS1523　switch 语句代码前应有 case 或 default 关键字
错误 CS1524　缺少 catch 或 finally
错误 CS1525　表达式中含有无效字符
错误 CS1526　new 表达式中的类型后应有“()”或“[]”
错误 CS1527　名字空间的元素不能显式声明为私有的或保护的
错误 CS1528　缺少“;”或“=”
错误 CS1529　using 子句应该在名字空间其它元素之前
错误 CS1530　对名字空间的元素不允许使用 new 或 unsafe 关键字
错误 CS1532　属性或索引指示器属性不能同时重载一个访问器而覆盖另一个访问器
错误 CS1533　代表不能直接调用
错误 CS1534　二元操作符重载只能有两个参数
错误 CS1535　一元操作符重载只能有一个参数
错误 CS1536　void 参数类型无效
错误 CS1537　别名使用指示符在名字空间中已经出现过
错误 CS1538　不能通过包含类的派生类型来引用类型
错误 CS1539　非法的分割符
错误 CS1540　不能通过全权名访问保护类型成员
错误 CS1541　选项无效：符号中不能包含目录
错误 CS1542　‘dll’不能加入到装配中，它已存在于另一个装配中，应使用/R 选项
错误 CS1545　语言不支持属性或索引指示器属性引用的类型，尝试直接调用读或写访问器
错误 CS1546　语言不支持属性或索引指示器属性引用的类型，尝试直接调用访问器
错误 CS1547　当前环境中不允许使用 void 关键字
错误 CS1548　标记装配时密码错误
错误 CS1549　找不到适当的密码服务
错误 CS1551　索引指示器至少应有一个参数
错误 CS1552　指定数组类型时，“[]”应该出现在参数名称前
错误 CS1553　非法声明，应使用“修饰符+参数类型+……”的格式
错误 CS1554　非法声明，应使用“类型+操作符+……”的格式
错误 CS1555　程序中 Main 方法找不到包含的类
错误 CS1556　Main 方法的所有者应是一个有效的类或结构
错误 CS1557　第一个输出文件中找不到包含 Main 方法的类，而在第二个输出文

件中找到

错误 CS1558　类中没有合适的 Main 方法
错误 CS1559　导入的对象不能作为程序的入口使用
错误 CS1560　#line 指定的文件名太长
错误 CS1562　无源输出：应指定选项/out
错误 CS1563　输出的文件没有任何源文件
错误 CS1565　指定选项冲突：win32 res 和 win32 icon
错误 CS1566　读源文件错误
错误 CS1567　生成 win32 资源文件时出错
错误 CS1569　生成 XML 文档错
警告（1 级）CS1570　XML 注释生成错误格式的 XML 文档
错误 CS1571　XML 注释参数标记重复
警告（2 级）错误 CS1572　找不到 XML 注释含有的参数标记
警告（4 级）错误 CS1573　XML 注释含有的参数标记与实际参数不符
警告（1 级）错误 CS1574　XML 注释含有的 cref 特征项找不到
错误 CS1575　分配堆栈的表达式在类型后应有“[]”
错误 CS1576　#line 指示符后缺少指定的行数
错误 CS1577　装配错误
错误 CS1578　缺少文件名、单行注释或行结尾
错误 CS1579　foreach 语句声明的类型没有定义，或者不可访问
错误 CS1580　XML 注释含有的 cref 特征项中有无效的参数个数
错误 CS1581　XML 注释含有的 cref 特征项中有无效的返回值
错误 CS1583　文件不是有效的 win32 资源文件
错误 CS1585　成员修饰符的关键字应出现在成员类型或名称前
错误 CS1586　数组创建时应有数组长度或数组的初始化
错误 CS1600　编译被用户中断
错误 CS1900　警告级别应在 0~4 之间
错误 CS1901　指定的选项冲突：0 警告级别，警告提升为错误
错误 CS1904　无效的警告数目
错误 CS2000　编译器初始化错误
错误 CS2001　找不到源文件
错误 CS2002　源文件指定多次
错误 CS2003　响应文件包含多次
错误 CS2005　命令行选项缺少指定的文件
错误 CS2006　命令行语法错误：switch 语句缺少文字
错误 CS2007　无法识别的命令行选项
错误 CS2008　没有指定的输入
错误 CS2011　无法打开响应文件
错误 CS2012　无法打开文件进行修改

错误 CS2013　映像基准数无效
错误 CS2014　旧的命令行选项已过时，请使用新的
错误 CS2015　文件不是源代码文件，而是二进制文件
错误 CS2016　代码页无效或没有安装
错误 CS2017　对同一输出文件不能使用/main 和/dll
错误 CS2018　打不开消息文件“cscmsgc.dll”
错误 CS2019　/target 目标类型无效，请指明‘exe’、‘winexe’、‘library’或‘module’
错误 CS2020　只有首先输出的文件才能建立目标，而不是‘module’
错误 CS2021　文件名太长或无效
错误 CS3000　带有变量参数的方法不符合 CLS
错误 CS3001　参数类型不符合 CLS
错误 CS3002　方法返回值不符合 CLS
错误 CS3003　变量类型不符合 CLS
错误 CS3004　混合和分解的 unicode 字符不符合 CLS
错误 CS3005　只有大小写不同的指示符不符合 CLS
错误 CS3006　只有 ref 和 out 不同的指示符不符合 CLS
错误 CS3008　指示符不符合 CLS
错误 CS3009　基类型或接口不符合 CLS
错误 CS3010　符合 CLS 的接口不能有不符合 CLS 的成员
错误 CS3011　不符合 CLS 的成员不能是抽象的
错误 CS3012　在模块中不能指定符合 CLS 的属性
错误 CS3013　添加的模块不符合 CLS，或者缺少符合 CLS 的特征
错误 CS3014　成员不能被标记为符合 CLS，因为装配没有标记为符合 CLS 的
错误 CS3015　数组类型的方法特征参数不符合 CLS
错误 CS5000　未知的编译器错误
警告（1 级）CS5001　程序输出文件不包含已定义的入口

附录 C　.Net 名字空间成员速查

Microsoft.ComServices

类、结构、枚举

BootstrapHelper　ComponentServices　ContextUtil　RegistrationErrorInfo RegistrationHelper　SecurityCallContext　SecurityCallers　SecurityIdentifier SecurityIdentity　SharedPropertyGroupManager　SubAuthorityArray ApplicationAccessControlAttribute ApplicationQueuingAttribute ConstructionEnabledAttribute　DescriptionAttribute　EventClassAttribute GenericPropertyAttribute　InterfacePropertyAttribute　MethodPropertyAttribute AutoCompleteAttribute　ApplicationPropertyAttribute　ApplicationActivationAttribute ApplicationIDAttribute　ApplicationNameAttribute　ComponentPropertyAttribute COMTIIntrinsicsAttribute　EventTrackingEnabledAttribute　ExceptionClassAttribute IISIntrinsicsAttribute　JustInTimeActivationAttribute　LoadBalancingSupportedAttribute MustRunInClientContextAttribute　SynchronizationAttribute　TransactionAttribute ComponentAccessControlAttribute　InterfaceQueuingAttribute　ObjectPoolingAttribute SecurityRoleAttribute　RegistrationHelperTx　BOID　XACTTRANSINFO AccessChecksLevelOption　ActivationOption　AuthenticationOption ImpersonationLevelOption　InstallationFlags　PropertyLockMode　PropertyReleaseMode SynchronizationOption　TransactionOption　TransactionVote

接口

IGetContextProperties　IObjectConstruct　IObjectConstructString IObjectContext　IObjectContextInfo　IObjectControl　IRegHelperInternal IRegistrationHelper　ITransaction　SharedProperty　SharedPropertyGroup

Microsoft.Win32

类、结构、枚举

Registry　RegistryKey　RegistryHive

System

类、结构、枚举

Object　Activator　AppDomainFlags　Array　Attribute　AttributeUsageAttribute CLSCompliantAttribute　ContextStaticAttribute　FlagsAttribute LoaderOptimizationAttribute　MTAThreadAttribute　NonSerializedAttribute ObsoleteAttribute　ParamArrayAttribute　SerializableAttribute　STAThreadAttribute ThreadStaticAttribute　BitConverter　Buffer　CallContext　Console　Convert　DBNull

Delegate MulticastDelegate UnhandledExceptionEventHandler AsyncCallback CrossAppDomainDelegate EventHandler Empty Environment EventArgs UnhandledExceptionEvent Exception SystemException TypeInitializationException TypeLoadException EntryPointNotFoundException TypeUnloadedException UnauthorizedAccessException WeakReferenceException AccessException FieldAccessException MethodAccessException MissingMemberException MissingMethodException MissingFieldException AppDomainUnloadedException AppDomainUnloadInProgressException ArgumentException ArgumentNullException ArgumentOutOfRangeException DuplicateWaitObjectException ArithmeticException DivideByZeroException NotFiniteNumberException OverflowException ArrayTypeMismatchException BadImageFormatException CannotUnloadAppDomainException ContextMarshalException CoreException ExecutionEngineException IndexOutOfRangeException StackOverflowException FormatException InvalidCastException InvalidOperationException MulticastNotSupportedException NotImplementedException NotSupportedException PlatformNotSupportedException NullReferenceException OutOfMemoryException RankException ServicedComponentException URIFormatException ApplicationException GC LocalDataStoreSlot LogicalCallContext MarshalByRefObject ServicedComponent __ComObject AppDomain ContextBoundObject Math OperatingSystem Radix Random Type String TimeZone URI Value ValueType Void ArgIterator Boolean Byte Char Currency DateTime Decimal Double Enum LoaderOptimization PlatformID TypeCode UriHostNameType URIPartial AttributeTargets Guid Int16 Int32 Int64 IntPtr ParamArray RuntimeArgumentHandle RuntimeFieldHandle RuntimeMethodHandle RuntimeTypeHandle SByte Single TimeSpan TypedReference UInt16 UInt32 UInt64 UIntPtr Version WeakReference

接口

IAsyncResult ICloneable IComparable IConvertible ICustomFormatter IFormattable ILogicalThreadAffinative IServiceObjectProvider

System.CodeDOM

类、结构、枚举

CodeAttributeArgument CodeAttributeArgumentCollection CodeAttributeBlock CodeAttributeDeclaration CodeAttributeDeclarationCollection CodeCatchClause CodeCatchClauseCollection CodeClassCollection CodeClassMemberCollection CodeExpression CodeFieldReferenceExpression CodeIndexerExpression CodeLiteralExpression CodeMethodInvokeExpression CodeObjectCreateExpression CodeParameterDeclarationExpression CodePrimitiveExpression CodePropertyReferenceExpression CodeThisReferenceExpression

CodeTypeOfExpression CodeTypeReferenceExpression CodeArrayCreateExpression CodeBaseReferenceExpression CodeBinaryOperatorExpression CodeCastExpression CodeDelegateCreateExpression CodeDelegateInvokeExpression CodeExpressionCollection CodeLinePragma CodeNamespace CodeLiteralNamespace CodeNamespaceImportCollection CodeParameterDeclarationExpressionCollection CodeStatement CodeThrowExceptionStatement CodeTryCatchFinallyStatement CodeVariableDeclarationStatement CodeAssignStatement CodeAttachEventStatement CodeClassMember CodeLiteralClassMember CodeMemberEvent CodeMemberField CodeMemberMethod CodeClassConstructor CodeConstructor CodeMemberProperty CodeClass CodeClassDelegate CodeCommentStatement CodeDelegateInvokeStatement CodeDetachEventStatement CodeForLoopStatement CodeIfStatement CodeLiteralStatement CodeMethodInvokeStatement CodeMethodReturnStatement CodeNamespaceImport CodeStatementCollection CodeBinaryOperatorType FieldDirection MemberAttributes

System.CodeDOM.Compiler

类、结构、枚举

CodeGenerator CodeCompiler CSharpCodeGenerator JSCodeGenerator JSDoubleCompilationCodeGenerator VBCodeGenerator CompilerError CompilerErrorCollection CompilerParameters CompilerResults Executor TempFiles IndentedTextWriter

接口

IBatchCompiler ICodeCompiler ICodeGenerator

System.Collections

类、结构、枚举

ArrayList BitArray CaseInsensitiveComparer CaseInsensitiveHashCodeProvider Comparer Dictionary Hashtable CaseInsensitiveHashtable NameObjectCollectionBase NameValueCollection ObjectList Queue SortedList CaseInsensitiveSortedList Stack StringCollection StringTable DictionaryEntry

接口

ICollection IComparer IDictionary IDictionaryEnumerator IEnumerable IEnumerator IHashCodeProvider IList

System.Collections.Bases

类、结构、枚举

TypedCollectionBase TypedDictionaryBase TypedReadonlyCollectionBase

System.ComponentModel

类、结构、枚举

LookForClassInfo MemberAttribute MergablePropertyAttribute NotificationAttribute PersistableAttribute PersistContentsAttribute PersisterAttribute PrivatePersisterAttribute ProvidePropertyAttribute ReadOnlyAttribute RecommendedAsConfigurableAttribute RunInstallerAttribute ServerExplorerBrowsableAttribute ServerExplorerReadOnlyAttribute ShowInToolboxAttribute TypeConverterAttribute ValueEditorAttribute BindableAttribute BrowsableAttribute CategoryAttribute SysCategoryAttribute ComponentEditorAttribute DefaultEventAttribute DefaultPropertyAttribute DefaultValueAttribute DescriptionAttribute SysDescriptionAttribute DesignerAttribute DesignerCategoryAttribute DesignOnlyAttribute DispidAttribute EditorAttribute EventDescriptorAttribute ExtenderPropertyAttribute ExtenderProvidedPropertyAttribute HelpAttribute ImmutableObjectAttribute LicenseProviderAttribute ListBindableAttribute LocalizableAttribute BaseComponentEditor ClassInfo CompModSwitches Component Container CreationArgument EventDescriptorCollection EventHandlerList License LicenseContext LicenseManager LicenseProvider LicFileLicenseProvider MemberAttributeCollection MemberDescriptor PropertyDescriptor ReflectPropertyDescriptor ExtendedPropertyDescriptor EventDescriptor ReflectEventDescriptor PersistInfo CreationBundle PropertyDescriptorCollection SubProperty SyntaxCheck TypeConverter TypeListConverter VariantConverter BooleanConverter ByteConverter CharConverter CollectionConverter ArrayConverter CultureInfoConverter CurrencyConverter DateTimeConverter DecimalConverter DoubleConverter EnumConverter AlphabeticalEnumConverter ExpandableObjectConverter GuidConverter Int16Converter Int32Converter Int64Converter ReferenceConverter ComponentConverter SingleConverter StringConverter TimeSpanConverter TypeDescriptor ValidatingEventArgs ValueEditor CancelEventHandler CollectionChangeEventHandler PropertyChangedEventHandler RefreshEventHandler TableChangedEventHandler ValidatingEventHandler CancelEventArgs CollectionChangeEventArgs PropertyChangedEventArgs RefreshEventArgs TableChangedEventArgs InvalidEnumArgumentException Win32Exception LicenseException ObjectDisposedException WarningException MarshalByRefComponent Sys BindableSupport CollectionChangeAction LicenseUsageMode PersistableSupport TableChangedType TableSortDirection ValueEditorStyles

接口

IClassInfo IComNativeDescriptorHandler IComponent IContainer ICustomTypeDescriptor IExtenderProvider IPersistable ISite ISupportInitialize

ISynchronizeInvoke　ITable　ITypeDescriptorContext　IValueAccess

System.ComponentModel.Design

类、结构、枚举

AccessedThroughPropertyAttribute　InheritanceAttribute
NotifyParentPropertyAttribute　ParenthesizePropertyNameAttribute　PropertyTabAttribute
RefreshPropertiesAttribute　ToolboxItemAttribute　ValueProviderCollection
AddRemoveComponentActionUnit　CodePersister　ComponentModelPersister
CodeStream　CommandID　ComponentChangeBaseActionUnit
PropertyChangeActionUnit　EventHandlerSetActionUnit　ComponentDesigner
CompoundActionUnit　CompoundPropertyChangeActionUnit
DesigntimeLicenseContextSerializer　DocumentCollection　InheritanceService　Localizer
MemberModifier　MenuCommand　DesignerVerb　PropertyValue
ServiceObjectContainer　StandardCommands　StandardToolWindows　ValueEditorHost
ValueProvider　ResourceValueProvider　ValueProviderService　DesigntimeLicenseContext
InheritedPropertyDescriptor　ValueEditorConverter　NumericFormatTypeEditor
IntegerEditor　BlankIntegerEditor　AliasEditor　DateTimeFormatEditor
NumericFormatEditor　ByteEditor　ContentAlignmentEditor　CultureInfoEditor
CurrencyEditor　DatePickerEditor　DecimalEditor　GuidEditor　ImmediateStringEditor
NameEditor　ObjectPropertiesEditor　OldArrayEditor　OldCollectionEditor
ReadOnlyValueEditor　ReferenceEditor　NonEditableReferenceEditor　StringEditor
StringListEditor　TimeSpanEditor　VariantEditor　ActionUnitEventHandler
ActiveDocumentEventHandler　BatchOperationEventHandler
ComponentChangedEventHandler　ComponentChangingEventHandler
ComponentEventHandler　ComponentRenameEventHandler
ContainerSelectorActiveEventHandler　DocumentEventHandler　TestCommand
ToolboxComponentsCreatedEventHandler　TypeChangedEventHandler
UIActivationStateEventHandler　ServiceCreatorCallback　BinaryEditor　CollectionEditor
ArrayEditor　DateTimeEditor　ObjectSelectorEditor　ActionUnitEventArgs
ActiveDocumentEventArgs　BatchOperationEventArgs　ComponentChangedEventArgs
ComponentChangingEventArgs　ComponentEventArgs　ComponentRenameEventArgs
ContainerSelectorActiveEventArgs　DocumentEventArgs
ToolboxComponentsCreatedEvent　TypeChangedEventArgs　UIActivationStateEventArgs
CheckoutException　ArrayDialog　ByteViewer　ActionType　AdornmentType
ContainerSelectorActiveEventArgsType　DisplayMode　HelpContextType
HelpKeywordType　InheritanceLevel　PropertyTabScope　RefreshProperties
SelectionRules　SelectionStyles　SelectionTypes　ValueExpressionLevel

接口

IActionUnit　IApplicationExtensibilityService　ICheckoutService

ICodeGenSupressionService ICodeNavigatorService ICodePersisterTaskListService IComponentChangeService IComponentCodePersistanceService ICompoundActionUnit IDeferLoadService IDesigner IDesignerEventService IDesignerFilter IDesignerHost IDesignerOptionService IDictionaryService IEditorSite IEventPropertyService IExtenderListService IExtenderProviderService IExtensibilityService IFilePersistanceService IHelpService IInheritanceService IMenuCommandService IReferenceService IResourceService ISelectionService IServiceObjectContainer ITestCommandService ITypeDescriptorFilterService IUIActivationService IUndoService IValueProvider IValueProviderCollection IValueProviderService

System.ComponentModel.Design.CodeModel

类、结构、枚举

DelegateTypeInfo

接口

ICodeArrayCreateExpression ICodeBlockComment ICodeCast ICodeClass ICodeClassImport ICodeConstructor ICodeDataMember ICodeDelegate ICodeDOMStatement ICodeElement ICodeElementLocation ICodeElements ICodeEventSetStatement ICodeLocalVariableStatement ICodeMember ICodeMembers ICodeMethod ICodeMethodInvokeExpression ICodeMethodInvokeStatement ICodeNameReferenceExpression ICodeNamespaceImport ICodeObjectCreateExpression ICodeParameter ICodeParameters ICodeProperty ICodePropertySetStatement ICodeReturnValueStatement ICodeSourceFile ICodeStatement ICodeStatements

System.Configuration

类、结构、枚举

BaseConfigCollection BaseConfigItem ConfigManager LevelOfService QueryCell Selector ConfigQuery FileSelector LocalMachineSelector AppDomainSelector HttpSelector NullSelector ConfigException ConfigExceptionType PutFlags QueryCellOp

接口

IConfigCollection IConfigItem

System.Configuration.Assemblies

类、结构、枚举

AssemblyHash AssemblyHashAlgorithm AssemblyVersionCompatibility ProcessorID

System.Configuration.Core

类、结构、枚举

AppDomain BindingPolicy CodeBaseHint

System.Configuration.Install

类、结构、枚举

Installer TransactedInstaller AssemblyInstaller ComponentInstaller InstallContext WfcInstallContext InstallerCollection InstallEventHandler InstallEventArgs InstallException UninstallAction

System.Configuration.Interceptors

类、结构、枚举

ListAppend ListMerge ListPrepend NativeCatalogInterceptor PropertyOverride HttpFileSelector ListSelector ConfigInterceptorException RequestParams ListMergeDirective

接口

IConfigMerger IConfigReader IConfigTransformer

System.Configuration.Schema

类、结构、枚举

PropertySchemaCollection QueryMeta RelationMeta SchemaFile ConfigSchema ConfigTypeSchema PropertySchema ConfigTypeSchemaMetaFlags ConfigTypeSchemaSchemaGeneratorFlags RelationMetaMetaFlags

System.Configuration.Web

类、结构、枚举

AppSettings Assembly Authentication AuthorizationItem CodeAccess Compilation Compiler ConfigSection Cookie Credential CustomErrors DiscoverySearchPatternType Globalization HttpHandler HttpHandlerFactory HttpModule Impersonation Location MimeImporterType MimeInfoType MimeReflectorType ParameterReaderType Passport ProcessModel ProtocolImporterType ProtocolInfoType ProtocolReflectorType ProtocolType ReferenceResolverType ReturnWriterType SdlHelpGenerator SessionState Trace User WebControl

System.Core

类、结构、枚举

HashCodes SystemEvents UNDONE Utils PowerModeChangedEventHandler SessionEndedEventHandler SessionEndingEventHandler TimerElapsedEventHandler

UserPreferenceChangedEventHandler PowerModeChangedEventArgs SessionEndedEventArgs SessionEndingEventArgs TimerElapsedEventArgs UserPreferenceChangedEventArgs CprException EmptyEnumeratorException PowerModes SessionEndReasons UserPreferenceCategories

接口

IIndexedCollection

System.Data

类、结构、枚举

AdapterDesignerAttribute DataCategoryAttribute DataSysDescriptionAttribute PropertiesCollection DataColumn ObjectStoreColumn DataSet DataSetView DataTable ObjectStoreTable DataView RelatedView DataRowViewColumnDescriptor DataRowViewRelationDescriptor DataSetViewListItemTableDescriptor BaseCollection ColumnsCollection ConstraintsCollection RelationsCollection RowsCollection TablesCollection TableSettingsCollection Constraint ForeignKeyConstraint UniqueConstraint DataError DataExpression DataFilter DataKey DataRelation DataRow DataRowBuilder DataRowViewTypeDescriptor DataRowView DataSetViewListItemTypeDescriptor DataSetViewListItem DataUtil StreamFormats TableSetting ConvertEventHandler DataColumnChangeEventHandler DataRowChangeEventHandler DataRowCreatedEventHandler ListChangedEventHandler MergeFailedEventHandler PositionChangingEventHandler StateChangeEventHandler ConvertEventArgs DataColumnChangeEventArgs DataRowChangeEventArgs IndexEventArgs ListChangedEventArgs MergeFailedEventArgs PositionChangingEventArgs StateChangeEventArgs DataException DeletedRowInaccessibleException DuplicateNameException InRowChangingEventException InvalidBindingException InvalidConstraintException InvalidExpressionException SyntaxErrorException EvaluateException MissingPrimaryKeyException NoNullAllowedException ReadOnlyException RowNotInTableException VersionNotFoundException ConstraintException DataSys Range AcceptRejectRule AggregateTypes CascadeAction CommandType DataRowAction DataRowState DataRowVersion DataViewRowState DBCommandBehavior DBObjectState EnforceConstraintsRule IsolationLevel ListChangedType MappingType MissingMappingAction MissingSchemaAction ParameterDirection PropertyAttributes ReadState Rule SchemaType SortDirection SortOrder StatementType TreePosition UpdateRowSource UpdateStatus XmlContent XmlOrder

接口

ICollectionInfo IColumnMapping IColumnMappings IDataErrorInfo IDataList IDataParameter IDataReader IDataRecord IDataSetCommand

IEditableList IEditableObject IFilter IIndexedList IListSource ILiveList INameTable IPresentedProperties IPropertyDescriptor ISortedList ITableMapping ITableMappings ITypedElements ITypedList ITypedObject

System.Data.ADO

类、结构、枚举

ADODataSetCommand ADOConnection ADOCommandConverter ADOParameterConverter ADOError ADOErrors ADOParameter ADOParameters ADOProperties ADOProperty ADOCommand ADODataReader ADOInfoMessageEventHandler ADORowUpdatedEventHandler ADORowUpdatingEventHandler ADOSchemaMappingEventHandler ADOInfoMessageEventArgs ADORowUpdatedEventArgs ADORowUpdatingEventArgs ADOSchemaMappingEventArgs ADOException ADODBType

System.Data.Internal

类、结构、枚举

DataSetCommand DBDataSetCommand DBConnection DBColumnDescriptor DataColumnMappingConverter DataTableMappingConverter DataColumnMapping DataColumnMappings DataStorage DateTimeStorage DecimalStorage DoubleStorage Int16Storage Int32Storage Int64Storage ObjectStorage SByteStorage SingleStorage SparseStorage StringStorage TimeSpanStorage UInt16Storage UInt32Storage UInt64Storage BooleanStorage ByteStorage CharStorage DataTableMapping DataTableMappings DBCommand DBDataReader DBDataRecord DBEnumerator SchemaChangedEventHandler SchemaChangingEventHandler InfoMessageEventArgs RowUpdatedEventArgs RowUpdatingEventArgs SchemaChangedEventArgs SchemaChangingEventArgs SchemaMappingEventArgs

接口

IDataParameters

System.Data.SQL

类、结构、枚举

SQLDataSetCommand SQLConnection SQLCommandConverter SQLParameterConverter SQLCommand SQLDataReader SQLError SQLErrors SQLParameter SQLParameters SQLInfoMessageEventHandler SQLRowUpdatedEventHandler SQLRowUpdatingEventHandler SQLSchemaMappingEventHandler SQLRowUpdatedEventArgs SQLRowUpdatingEventArgs SQLSchemaMappingEventArgs SQLInfoMessageEventArgs SQLException SQLPooledConnection SQLDataType

接口

IGetDispenser ISQLDataRecord ITransaction ITransactionExport ITransactionExportFactory

System.Data.SQLTypes

类、结构、枚举

SQLBinary SQLString SQLException SQLNullValueException SQLTruncateException SQLBit SQLBoolean SQLByte SQLDateTime SQLDouble SQLGuid SQLInt16 SQLInt32 SQLInt64 SQLMoney SQLNumeric SQLSingle

接口

INullable

System.Diagnostics

类、结构、枚举

MonitoringDescriptionAttribute ConditionalAttribute DebuggableAmbivalentAttribute DebuggableAttribute DebuggerHiddenAttribute DebuggerStepThroughAttribute CountersData InstancesData EventLogInstaller PerformanceCounterInstaller EventLog EventLogEntry PerformanceCounter Process ProcessModule ProcessStartInfo ProcessThread EventLogEventHandler CoreSwitches Counter CounterCreationData CounterCreationDataCollection CounterSampleCalculator Debug Debugger EventLogNames InstanceData PerformanceCounterCategory PerformanceCounterManager SerializableRegistryKey StackFrame StackTrace Switch TraceSwitch BooleanSwitch Trace TraceListener DefaultTraceListener EventLogTraceListener TextWriterTraceListener TraceListeners EventLogEvent CounterSample EventLogEntryType PerformanceCounterType ProcessPriorityClass ProcessWindowStyle ThreadPriorityLevel ThreadState ThreadWaitReason TraceLevel

接口

ICollectData

System.Diagnostics.SymbolStore

类、结构、枚举

SymBinder SymDocument SymDocumentType SymDocumentWriter SymLanguageType SymLanguageVendor SymMethod SymReader SymScope SymVariable SymWriter SymbolToken SymAddressKind

接口

ISymbolBinder ISymbolDocument ISymbolDocumentWriter ISymbolMethod ISymbolNamespace ISymbolReader ISymbolScope ISymbolVariable ISymbolWriter

System.DirectoryServices

类、结构、枚举

DSDescriptionAttribute DirectoryEntry DirectorySearcher DirectoryEntryCollection DirectoryEntryProperty DirectoryEntryPropertyCollection DirectoryEntryPropertyValueCollection SearchResults SearchResultsEntry SearchResultsProperty SearchResultsPropertyCollection SearchResultsPropertyValueCollection SortOption ReferralChasingOptions SearchScope SortDirection

System.Drawing

类、结构、枚举

GraphicsAdvancedAttribute ToolboxBitmapAttribute ColorConverter CursorConverter FontConverter ImageConverter ImageFormatConverter PointConverter RectangleConverter SizeConverter IconConverter Brushes ColorTranslator Cursor Cursors Icon ImageAnimator Pens SystemBrushes SystemColors SystemIcons SystemPens Brush SolidBrush TextureBrush Font FontFamily Graphics Image Bitmap Pen Region StringFormat Color Point PointF Rectangle RectangleF Size SizeF BrushStyle ContentAlignment FontStyle GraphicsUnit KnownColor PenStyle PolyFillMode StringAlignment StringDigitSubstitute StringFormatFlags StringTrimming StringUnit

System.Drawing.Design

类、结构、枚举

PointEditor RectangleEditor SizeEditor PropertyValueUIHandler PropertyValueUIItemInvokeHandler PropertyValueUIItem ToolboxItem DrawingToolboxItem UITypeEditor ColorEditor COM2ExtendedUITypeEditor ContentAlignmentEditor CursorEditor FontEditor FontNameEditor IconEditor ImageEditor MetafileEditor BitmapEditor ToolboxItemInfoEnum UITypeEditorEditStyle

接口

IPropertyValueUIService IToolboxUser

System.Drawing.Drawing2D

类、结构、枚举

Blend ColorBlend PathData RegionData HatchBrush LinearGradientBrush PathGradientBrush CustomLineCap AdjustableArrowCap GraphicsContainer GraphicsPath GraphicsPathIterator GraphicsState Matrix CombineMode CompositingMode CompositingQuality CoordinateSpace DashStyle FillMode FlushIntention HatchStyle InterpolationMode LinearGradientMode LineCap

LineJoin MatrixOrder PathPointType PenAlignment PenType PixelOffsetMode QualityMode RenderingHint SmoothingMode WarpMode WrapMode

System.Drawing.Imaging

类、结构、枚举

APMFileHeader BitmapData ColorMap ColorMatrix ColorPalette Decoder Encoder EncoderParameter EncoderParameters FrameDimension ImageAttributes ImageCodecInfo ImageFormat MetafileHeader METAHEADER PropertyItem Metafile ColorAdjustType ColorChannelFlags ColorMapType ColorMatrixFlags ColorMode EmfPlusRecordType EmfType EncoderParameterValueType ImageCodecFlags ImageFlags ImageLockMode MetafileFrameUnit MetafileType PaletteFlags PixelFormat

System.Drawing.Printing

类、结构、枚举

PrintDocument MarginsConverter PrintEventHandler PrintPageEventHandler QueryPageSettingsEventHandler Margins PageSettings PaperSize PaperSource PreviewPageInfo PrintController StandardPrintController PreviewPrintController PrinterResolution PrinterSettings PrinterUnitConvert PrintEventArgs QueryPageSettingsEventArgs PrintPageEventArgs InvalidPrinterException Duplex PaperKind PaperSourceKind PrinterResolutionKind PrinterUnit PrintRange

System.Drawing.Text

类、结构、枚举

FontCollection InstalledFontCollection PrivateFontCollection GenericFontFamilies HotkeyPrefix LineSpacing TextRenderingHint

System.Globalization

类、结构、枚举

Calendar GregorianCalendar JapaneseCalendar JulianCalendar KoreanCalendar ThaiBuddhistCalendar HebrewCalendar HijriCalendar CharacterInfo CompareInfo CultureInfo DateTimeFormatInfo NumberFormatInfo RegionInfo SortKey StringInfo TextElementEnumerator TextInfo CompareOptions CultureTypes DateTimeStyles GregorianCalendarTypes NumberStyles UnicodeCategory DaylightTime

System.IO

类、结构、枚举

IODescriptionAttribute FileSystemWatcher FileSystemEventHandler

RenamedEventHandler FileSystemEventArgs RenamedEventArgs IOException DirectoryNotFoundException EndOfStreamException FileNotFoundException PathTooLongException InternalBufferOverflowException BinaryReader BinaryWriter FileSystemEntry Directory File Stream BufferedStream FileStream MemoryStream TextReader StreamReader StringReader TextWriter StreamWriter StringWriter ChangedFilters FileAccess FileMode FileShare FileSystemAttributes SeekOrigin WatcherChangeTypes WatcherTarget WaitForChangedResult

System.IO.IsolatedStorage

类、结构、枚举

IsolatedStorageException IsolatedStorage IsolatedStorageFile IsolatedStorageFileStream IsolatedStorageScope

System.Management

类、结构、枚举

ManagementBaseObject ManagementObject ManagementClass ManagementEventWatcher ManagementObjectSearcher CompletedEventHandler EventArrivedEventHandler ObjectPutEventHandler ObjectReadyEventHandler ProgressEventHandler StoppedEventHandler ManagementEventArgs ObjectPutEventArgs ObjectReadyEventArgs ProgressEventArgs StoppedEventArgs CompletedEventArgs EventArrivedEventArgs ManagementException ManagementObjectCollection ManagementOperationWatcher ManagementOptions ObjectGetOptions PutOptions ConnectionOptions DeleteOptions EnumerationOptions GetInstancesOptions GetSubclassesOptions QueryOptions GetRelatedClassOptions GetRelatedOptions GetRelationshipClassOptions GetRelationshipOptions EventWatcherOptions InvokeMethodOptions ManagementPath ManagementQuery ObjectQuery SchemaQuery WQLSchemaQuery RelatedClassQuery RelationshipClassQuery SelectClassQuery ClassEnumerationQuery DataQuery InstanceEnumerationQuery WQLDataQuery RelatedObjectQuery RelationshipQuery SelectQuery EventQuery WQLEventQuery ManagementScope Method MethodSet Property PropertySet Qualifier QualifierSet AuthenticationLevel CIMType ComparisonSettings ImpersonationLevel ManagementStatus PutType TextFormat

接口

IWmiEventSource

System.Messaging

类、结构、枚举

MessagingDescriptionAttribute MessageQueueInstaller Message MessageQueue XmlMessageFormatter PeekCompletedEventHandler ReceiveCompletedEventHandler

PeekAsyncEventArgs ReceiveAsyncEventArgs MessageQueueException ActiveXMessageFormatter BinaryMessageFormatter DefaultPropertiesToSend MessagePropertyFilter MessageQueueCriteria MessageQueuesEnumerator MessagesEnumerator Acknowledgement AcknowledgeType CryptoProviderType EncryptionAlgorithm EncryptionRequired HashAlgorithm MessagePriority MessageType

接口

IMessageFormatter

System.Net

类、结构、枚举

WebHeaders HttpContinueDelegate ProtocolViolationException WebException AuthenticationManager AuthenticationScheme Authorization CredentialCache DefaultControlObject DNS EndPoint IPEndPoint IPXEndPoint GlobalProxySelection HttpExtension HttpStatusCode HttpVersion IPAddress IPHostEntry NetworkCredential ProxyData ServicePoint ServicePointManager SingleCredential SocketAddress WebListener WebRequest HttpWebRequest WebRequestFactory WebResponse HttpWebResponse DnsPermission SocketPermission WebPermission ConnectionModes NetworkAccess TransportType WebStatus

接口

IAuthenticationModule ICertificatePolicy ICredentialLookup IProxyInfo ISelectProxy IWebRequestCreate

System.Net.Sockets

类、结构、枚举

SocketException NetworkStream AddressFamily InvalidSocketConstants LingerOption MulticastOption ProtocolFamily ProtocolType Socket SocketErrors SocketMsgFlags SocketOption SocketShutdown SocketType TCPClient TCPListener UDPClient SelectMode

System.Reflection

类、结构、枚举

AssemblyCompanyAttribute AssemblyConfigurationAttribute AssemblyCopyrightAttribute AssemblyDefaultAliasAttribute AssemblyDescriptionAttribute AssemblyInformationalVersionAttribute AssemblyProductAttribute AssemblyTitleAttribute AssemblyTrademarkAttribute DefaultMemberAttribute MemberFilter ResolveEventHandler TypeFilter CustomAttributeFormatException AmbiguousMatchException

ReflectionTypeLoadException InvalidFilterCriteriaException TargetException TargetInvocationException TargetParameterCountException AssemblyNameProxy Assembly AssemblyName Binder ManifestResourceInfo MemberInfo MethodBase MethodInfo ConstructorInfo PropertyInfo TypeDelegator EventInfo FieldInfo Missing Module ParameterInfo Pointer StrongNameKeyPair UnmanagedMarshal AssemblyNameFlags BindingFlags CallingConventions EventAttributes FieldAttributes MemberTypes MethodAttributes MethodImplAttributes ParameterAttributes PropertyAttributes ResourceAttributes ResourceLocation TypeAttributes InterfaceMapping ParameterModifier

接口

ICustomAttributeProvider IReflect

System.Reflection.Emit

类、结构、枚举

AssemblyBuilder CustomAttributeBuilder EventBuilder ILGenerator LocalBuilder MethodRental OpCodes ParameterBuilder PropertyBuilder SignatureHelper MethodBuilder ConstructorBuilder EnumBuilder TypeBuilder FieldBuilder ModuleBuilder AssemblyBuilderAccess FlowControl OpCodeType OperandType PackingSize PEFileKinds StackBehaviour EventToken FieldToken Label LocalToken MethodToken OpCode ParameterToken PropertyToken SignatureToken StringToken TypeToken

System.Resources

类、结构、枚举

ResourceClass MissingManifestResourceException ResourceManager ResourceReader ResourceSet ResXResourceSet ResourceWriter ResXFileRef ResXResourceReader ResXResourceWriter StaticResourceManager

接口

IResourceReader IResourceWriter

System.Runtime.CompilerServices

类、结构、枚举

AssemblyCultureAttribute AssemblyDelaySignAttribute AssemblyKeyFileAttribute AssemblyKeyNameAttribute AssemblyOperatingSystemAttribute AssemblyProcessorAttribute AssemblyVersionAttribute CompilationRelaxationsAttribute CompilerGlobalScopeAttribute DiscardableAttribute NotInGCHeapAttribute RuntimeHelpers

System.Runtime.InteropServices

类、结构、枚举

ComAliasNameAttribute ComConversionLossAttribute ComEmulateAttribute ComImportAttribute ComRegisterFunctionAttribute ComSourceInterfacesAttribute ComUnregisterFunctionAttribute ComVisibleAttribute DispIdAttribute DllImportAttribute ExposeHResultAttribute FieldOffsetAttribute GlobalObjectAttribute GuidAttribute HasDefaultInterfaceAttribute IDispatchImplAttribute ImportedFromTypeLibAttribute InAttribute InterfaceTypeAttribute MarshalAsAttribute MethodImplAttribute NoComRegistrationAttribute NoIDispatchAttribute OutAttribute PredeclaredAttribute PreserveSigAttribute ProgIdAttribute StructLayoutAttribute TypeLibFuncAttribute TypeLibTypeAttribute TypeLibVarAttribute ObjectCreationDelegate COMEmulateException ExternalException SEHException COMException InvalidComObjectException InvalidOleVariantTypeException SafeArrayRankMismatchException SafeArrayTypeMismatchException VTableCallsNotSupportedException CallConvCdecl CallConvFastcall CallConvStdcall CallConvThiscall DispatchWrapper ErrorWrapper ExtensibleClassFactory FUNCDESC Marshal RegistrationServices Root TypeAttr TYPEATTR TypeLibAttr TypeLibConverter UnknownWrapper CALLCONV CallingConvention CharSet ComInterfaceType ExporterEventKind FUNCFLAGS FUNCKIND GCHandleType IDispatchImplType IDLFLAG ImporterEventKind INVOKEKIND LayoutKind LIBFLAGS MethodImplOptions PARAMFLAG PInvokeMap SYSKIND TYPEFLAGS TYPEKIND TypeLibExporterFlags TypeLibFuncFlags TypeLibTypeFlags TypeLibVarFlags UnmanagedType VarEnum ArrayWithOffset BIND_OPTS ELEMDESC FILETIME GCHandle HandleRef IDLDesc IDLDESC PARAMDESC STATSTG TypeDesc TYPEDESC

接口

ICustomMarshaler ITypeLibConverter ITypeLibExporterNotifySink ITypeLibImporterNotifySink UCOMIBindCtx UCOMIConnectionPoint UCOMIConnectionPointContainer UCOMIEnumMoniker UCOMIEnumString UCOMIEnumVARIANT UCOMIMoniker UCOMIPersistFile UCOMIRunningObjectTable UCOMIStream UCOMITypeInfo UCOMITypeLib

System.Runtime.InteropServices.Expando

接口

IExpando

System.Runtime.Remoting

类、结构、枚举

ContextAttribute Synchronization ThreadAffinity URLAttribute Embedded

OneWayAttribute ProxyAttribute ServicedComponentProxyAttribute SchemaType SoapMethodAttribute SoapOptionAttribute XmlAttribute XmlElement CrossContextDelegate HeaderHandler MessageSurrogateFilter RemotingException ServerException ClientSponsor Lease ObjectHandle ChannelServices Context ContextProperty Header HttpHandlerInfo InternalMessageWrapper MethodCallMessageWrapper MethodReturnMessageWrapper InternalRemotingServices LeaseManager LeaseSink LifetimeServices MethodCall ConstructionCall MethodResponse ConstructionResponse ObjRef RealProxy RemotingServices RemotingSurrogateSelector ReturnMessage SoapServices TrackingServices XmlNamespaceEncoder ActivatorLevel LeaseState SoapMethodOption SoapOption WellKnownObjectMode

接口

IActivator IChannel IChannelInfo IChannelReceiver IChannelSender IComponentServices IConstructionCallMessage IConstructionReturnMessage IContextAttribute IContextProperty IContextPropertyActivator IContributeClientContextSink IContributeDynamicSink IContributeEnvoySink IContributeObjectSink IContributeServerContextSink IDynamicMessageSink IDynamicProperty IEnvoyInfo ILease IMessage IMessageCtrl IMessageSink IMethodCallMessage IMethodMessage IMethodReturnMessage IObjectHandle IRemotingFormatter IRemotingTypeInfo ISchemaType ISponsor ITrackingHandler

System.Runtime.Remoting.Channels.HTTP

类、结构、枚举

HTTPChannelBase HTTPChannel HTTPChannelRequest HTTPChannelResponse HTTPMessageSink HttpRemotingHandler HttpRemotingHandlerFactory

System.Runtime.Remoting.Channels.SMTP

类、结构、枚举

MailAttachment MailMessage SMTPChannelBase SMTPChannel SmtpMail SMTPMessageSink SMTPRegisterSink CdoDSNOptions CdoEventStatus MailEncoding MailFormat MailPriority

接口

ISMTPMessage ISMTPOnArrival

System.Runtime.Remoting.Channels.TCP

类、结构、枚举

TCPChannel

System.Runtime.Serialization

类、结构、枚举

DeserializationEventHandler DeserializationEvent SerializationException CloningService Formatter FormatterConverter FormatterServices ObjectIDGenerator ObjectManager SerializationBinder SerializationInfo SerializationInfoEnumerator SurrogateSelector StreamingContextStates SerializationEntry StreamingContext SurrogateInfo

接口

IDeserializationEventListener IFormatter IFormatterConverter IObjectReference ISerializable ISerializationSurrogate ISurrogateSelector

System.Runtime.Serialization.Formatters

类、结构、枚举

InternalCV InternalFE InternalNI InternalPR InternalSAI InternalSOI InternalSOR InternalSOW InternalST ObjectWriter SerTrace ServerFault SoapFault SoapMessage FormatterAssemblyStyle FormatterTopObjectStyle FormatterTypeStyle InternalArrayTypeE InternalElementTypeE InternalMemberTypeE InternalMemberValueE InternalNameSpaceE InternalObjectPositionE InternalObjectTypeE InternalParseStateE InternalParseTypeE InternalPrimitiveTypeE InternalSerializerTypeE

接口

IFieldInfo ISerParser ISerWriter ISoapMessage ITrace

System.Runtime.Serialization.Formatters.Binary

类、结构、枚举

BinaryFormatter BinarySerializer BinaryArrayTypeEnum

System.Runtime.Serialization.Formatters.Soap

类、结构、枚举

SoapFormatter SoapSerializer

System.Security

类、结构、枚举

DynamicSecurityMethodAttribute SuppressUnmanagedCodeSecurityAttribute UnverifiableCodeAttribute SecurityException VerifierException XMLSyntaxException CodeAccessPermission PermissionSet NamedPermissionSet SecurityElement SecurityManager SecurityZone

接口

IEvidenceFactory IPermission ISecurityEncodable ISecurityPolicyEncodable

IStackWalk

System.Security

类、结构、枚举

DynamicSecurityMethodAttribute SuppressUnmanagedCodeSecurityAttribute UnverifiableCodeAttribute SecurityException VerifierException XMLSyntaxException CodeAccessPermission PermissionSet NamedPermissionSet SecurityElement SecurityManager SecurityZone

接口

IEvidenceFactory IPermission ISecurityEncodable ISecurityPolicyEncodable IStackWalk

System.Security.Cryptography.X509Certificates

类、结构、枚举

X509Certificate

System.Security.Permissions

类、结构、枚举

SecurityAttribute CodeAccessSecurityAttribute EnvironmentPermissionAttribute FileDialogPermissionAttribute FileIOPermissionAttribute IsolatedStoragePermissionAttribute IsolatedStorageFilePermissionAttribute PermissionSetAttribute PrincipalPermissionAttribute PublisherIdentityPermissionAttribute ReflectionPermissionAttribute RegistryPermissionAttribute SecurityPermissionAttribute SiteIdentityPermissionAttribute StrongNameIdentityPermissionAttribute UIPermissionAttribute URLIdentityPermissionAttribute ZoneIdentityPermissionAttribute EnvironmentPermission FileDialogPermission FileIOPermission IsolatedStoragePermission IsolatedStorageFilePermission PublisherIdentityPermission ReflectionPermission RegistryPermission SecurityPermission SiteIdentityPermission StrongNameIdentityPermission UIPermission URLIdentityPermission ZoneIdentityPermission PrincipalPermission StrongNamePublicKeyBlob EnvironmentPermissionAccess FileIOPermissionAccess IsolatedStorageContainment PermissionState ReflectionPermissionFlag RegistryPermissionAccess SecurityAction SecurityPermissionFlag UIPermissionClipboard UIPermissionWindow

接口

IUnrestrictedPermission

System.Security.Policy

类、结构、枚举

PolicyException AllMembershipCondition ApplicationDirectory ApplicationDirectoryMembershipCondition CodeGroup FirstMatchCodeGroup NetCodeGroup UnionCodeGroup Evidence Hash HashMembershipCondition PermissionRequestEvidence PolicyLevel PolicyStatement Publisher PublisherMembershipCondition Site SiteMembershipCondition SkipVerificationMembershipCondition StrongName StrongNameMembershipCondition Url URLMembershipCondition WebPage Zone ZoneMembershipCondition PolicyStatementAttribute

接口

ICodeGroup IIdentityPermissionFactory IMembershipCondition

System.Security.Principal

类、结构、枚举

GenericIdentity GenericPrincipal MTSPrincipal SecurityCallContext SecurityCallers SecurityIdentity WindowsIdentity WindowsImpersonationContext WindowsPrincipal PrincipalPolicy WindowsAccountType

接口

IIdentity IPrincipal

System.ServiceProcess

类、结构、枚举

SPDescriptionAttribute ServiceInstaller ServiceProcessInstaller ServiceBase ServiceController ServiceInstallerDialog ServiceControllerStatus ServiceInstallerDialogResult ServiceStart ServiceType

System.Text

类、结构、枚举

Decoder Encoder Encoding UnicodeEncoding UTF7Encoding UTF8Encoding ASCIIEncoding CodePageEncoding StringBuilder

System.Text.RegularExpressions

类、结构、枚举

ResourceClass MatchEvaluator Capture Group Match CaptureCollection MatchCollection Regex

System.Threading

类、结构、枚举

IOCompletionCallback ThreadExceptionEventHandler ThreadStart TimerCallback WaitCallback WaitOrTimerCallback ThreadExceptionEventArgs

SynchronizationLockException ThreadAbortException ThreadInterruptedException ThreadStateException ThreadStopException Interlocked Monitor Overlapped ReaderWriterLock RegisteredWaitHandle Thread ThreadPool Timeout Timer WaitHandle AutoResetEvent ManualResetEvent Mutex ApartmentState ThreadPriority ThreadState LockCookie NativeOverlapped

System.Timers

类、结构、枚举

TimersDescriptionAttribute Schedule Timer OccurredEventHandler OccurredEventArgs RecurrencePattern WeeklyPattern DailyPattern DaysOfMonthPattern MonthlyPattern RecurrencePatterns DaysOfWeek Months OccurrenceInMonth

接口

IRecurrencePattern

System.Web

类、结构、枚举

WebCategoryAttribute WebSysDescriptionAttribute HttpClientCertificate HttpApplicationState HttpCookieCollection HttpFileCollection HttpModuleCollection HttpCacheValidateHandler HttpException HttpWriter WebSys HttpCacheability HttpCacheRevalidation HttpValidationStatus ProcessShutdownReason ProcessStatus TraceEnableEnum TraceModeEnum ConfigureCapabilities HttpApplication HttpCachePolicy HttpCacheVary HttpCapabilities HttpBrowserCapabilities HttpContext HttpCookie HttpDebugHandler HttpExceptionInfo HttpMethodNotAllowedHandler HttpNotFoundHandler HttpNotImplementedHandler HttpPostedFile HttpRequest HttpResponse HttpRuntime HttpServerUtility HttpStaticObjectsCollection HttpUrl HttpUtility HttpWorkerRequest ProcessInfo ProcessModelInfo StaticFileHandler TraceContext

接口

IHttpAsyncHandler IHttpHandler IHttpHandlerFactory IHttpModule

System.Web.Caching

类、结构、枚举

CacheItemRemovedCallback CacheItemPriorities CacheItemPriorityDecay CacheItemRemovedReason Cache CacheDependency OutputCacheModule

System.Web.Configuration

类、结构、枚举

ConfigurationException ConfigXmlException ConfigXmlElement

ConfigurationInput ConfigXmlCursor ConfigXmlDocument ConfigXmlTokenTable DictionarySectionHandler HttpHandlersSectionHandler HttpModulesConfigurationHandler IgnoreSectionHandler SettingContainer SingleTagSectionHandler

接口

IConfigurationSectionHandler

System.Web.Security

类、结构、枚举

CookieAuthenticationEventHandler DefaultAuthenticationEventHandler PassportAuthenticationEventHandler WindowsAuthenticationEventHandler CookieAuthenticationEvent DefaultAuthenticationEvent PassportAuthenticationEvent WindowsAuthenticationEvent AuthenticationModes CookieAuthPasswordFormat CookieAuthentication CookieAuthenticationModule CookieAuthenticationTicket CookieIdentity DefaultAuthenticationModule FileAuthorizationModule PassportAuthenticationModule PassportIdentity UrlAuthorizationModule WindowsAuthenticationModule

System.Web.Services

类、结构、枚举

WebServicesDescriptionAttribute WebMethodAttribute WebServiceAttribute WebService TransactionMode WebServicesConfiguration WebServicesConfigurationSectionHandler

System.Web.Services.Description

类、结构、枚举

HtmlFormElementInfos HtmlOptionInfos HttpGetRequestResponseInfos HttpParameterInfos HttpPostRequestResponseInfos ImportedParameters ImportInfos MimeInfos ProtocolInfos ServiceDescriptions SoapAddressInfos SoapExtendsInfos SoapFaultInfos SoapHeaderInfos SoapImplementsInfos SoapInteractionInfos SoapInterfaceInfos TextMatchInfos HtmlInputType HtmlFormElementInfo HtmlInputInfo HtmlSelectInfo HtmlOptionInfo HttpGetRequestInfo HttpMimeGroupInfo HttpPostRequestInfo HttpResponseInfo HttpParameterInfo HttpPostServiceInfo HttpRequestResponseInfo HttpGetRequestResponseInfo HttpPostRequestResponseInfo ImportedParameter ImportedReturn ImportInfo MimeInfo TextInfo XmlMimeInfo AnyMimeInfo HtmlFormInfo MimeInfoImporter TextInfoImporter XmlMimeInfoImporter AnyMimeInfoImporter FormInfoImporter MimeInfoReflector XmlMimeInfoReflector FormInfoReflector ProtocolInfo SoapProtocolInfo HttpGetProtocolInfo HttpGetServiceInfo HttpPostProtocolInfo

ProtocolInfoImporter SoapProtocolInfoImporter HttpProtocolInfoImporter HttpGetProtocolInfoImporter HttpPostProtocolInfoImporter ProtocolInfoReflector SoapProtocolInfoReflector HttpProtocolInfoReflector HttpGetProtocolInfoReflector HttpPostProtocolInfoReflector ServiceDescription SoapAddressInfo SoapExtendsInfo SoapExtensionImporter SoapExtensionReflector SoapImplementsInfo SoapInteractionInfo SoapOneWayInfo SoapRequestResponseInfo SoapInterfaceInfo SoapServiceInfo SoapTypeRefInfo SoapFaultInfo SoapHeaderInfo SoapMessageInfo SoapRequestInfo SoapResponseInfo TextMatchInfo

System.Web.Services.Discovery

类、结构、枚举

DiscoveryClientResults DiscoveryReferences DiscoveryClientDocuments DiscoveryClientReferences DiscoveryExceptionDictionary DiscoveryClientProtocol DiscoveryClientResult DiscoveryDocument DiscoveryReference SchemaReference ContractReference DiscoveryDocumentReference DiscoveryRequestHandler DiscoverySearchPattern XmlSchemaSearchPattern ContractSearchPattern DiscoveryDocumentSearchPattern DynamicDiscoveryDocument ExcludePathInfo SoapBinding

System.Web.Services.Protocols

类、结构、枚举

HttpMethodAttribute MatchAttribute SoapExtensionAttribute SoapHeaderAttribute SoapMethodAttribute SoapHeaders CookieCollection ClientProtocol HttpClientProtocol SoapClientProtocol SoapException SoapHeaderException WebServiceRedirectException WebServiceTimeoutException LogicalMethodTypes SoapHeaderDirection SoapMessageStage AsyncResult Cookie HttpClientRequest HttpClientResponse HttpServerRequest HttpServerResponse HttpServerValueCollection LogicalMethodInfo MimeFormatter MimeParameterReader ValueCollectionParameterReader HtmlFormParameterReader UrlParameterReader MimeParameterWriter UrlEncodedParameterWriter UrlParameterWriter HtmlFormParameterWriter MimeReturnReader NopReturnReader TextReturnReader XmlReturnReader AnyReturnReader MimeReturnWriter XmlReturnWriter PatternMatcher ServerProtocol SoapServerProtocol DiscoveryServerProtocol HttpServerProtocol ServerType SoapExtension SoapHeader SoapUnknownHeader SoapMessage SoapServerMessage SoapClientMessage WebServiceHandlerFactory

System.Web.UI

类、结构、枚举

ConstructorNeedsTagAttribute ControlBuilderAttribute LiteralContentAttribute

PersistenceTypeAttribute TagPrefixAttribute TemplateAttribute ToolboxDataAttribute ValidationPropertyAttribute DataBindingCollectionConverter BuildTemplateMethod ImageClickEventHandler RenderMethod ImageClickEventArgs IndentedTextWriter HtmlTextWriter Html32TextWriter ClientTarget EncodingTypeEnum PersistenceMode AttributeCollection AutomaticHandlerMethodInfos BaseControlBuilder ControlBuilder ObjectTagBuilder PropertyBuilder TemplateBuilder RootBuilder BaseParser BatchTemplateParser TemplateParser ApplicationFileParser TemplateControlParser UserControlParser PageParser BatchDependencyWalker CompiledTemplateBase CompiledTemplateBuilder Control DataboundLiteralControl LiteralControl TemplateControl UserControl Page ControlCollection CssStyleCollection DataBinder DataBinding DataBindingCollection FieldDescriptor LosFormatter ObjectConverter PropertyConverter PropertySetter SimpleWebHandlerParser SourceReference StateBag StateItem TwoCharPeekableTextReader ValidatorCollection

接口

IAttributeAccessor INamingContainer IParserAccessor IPostBackDataHandler IPostBackEventHandler IStateManager ITagNameToTypeMapper ITemplate IValidator

System.Web.UI.Design

类、结构、枚举

ControlDesigner HtmlControlDesigner UserControlDesigner WebControlDesigner ReadWriteControlDesigner TemplatedControlDesigner TemplateEditingFrame DataFieldConverter DataSourceConverter DataBindingCollectionEditor UrlEditor ImageUrlEditor XmlFileEditor DataBindingValueUIHandler DHTMLBehavior IdentityBehavior SourceViewEventSink WebControlPersister

接口

IDataSourceProvider ISourceViewEventSink IWebFormReferenceManager IWebFormsBehavior IWebFormsBuilderService IWebFormsControlDesigner IWebFormsDesigner IWebFormServices IWebFormServices2

System.Web.UI.Design.WebControls

类、结构、枚举

BaseDataListComponentEditor DataGridComponentEditor DataListComponentEditor AdRotatorDesigner BaseValidatorDesigner ButtonDesigner CalendarDesigner CheckBoxDesigner HyperLinkDesigner LabelDesigner LinkButtonDesigner ListControlDesigner PageletDesigner PanelDesigner BaseDataListDesigner DataGridDesigner DataListDesigner RepeaterDesigner RegexTypeEditor ListItemsCollectionEditor TableCellsCollectionEditor

TableRowsCollectionEditor CalendarAutoFormatDialog RegexEditorDialog

System.Web.UI.Design.WebControls.ListControls

类、结构、枚举

ColumnCollectionEditor

System.Web.UI.HtmlControls

类、结构、枚举

HtmlControl HtmlImage HtmlInputControl HtmlInputFile HtmlInputHidden HtmlInputImage HtmlInputRadioButton HtmlInputText HtmlInputButton HtmlInputCheckBox HtmlContainerControl HtmlForm HtmlGenericControl HtmlSelect HtmlTable HtmlTableCell HtmlTableRow HtmlTextArea HtmlAnchor HtmlButton HtmlTableCellCollection HtmlTableRowCollection

System.Web.UI.WebControls

类、结构、枚举

Style TableItemStyle DataGridPagerStyle TableStyle WebColorConverter FontNamesConverter FontUnitConverter UnitConverter OptionalIntegerConverter TargetConverter ValidatedControlConverter AdCreatedEventHandler CommandEventHandler DataGridCommandEventHandler DataGridItemEventHandler DataGridPageChangedEventHandler DataGridSortCommandEventHandler DataListCommandEventHandler DataListItemEventHandler DayRenderEventHandler MonthChangedEventHandler RepeaterCommandEventHandler RepeaterItemEventHandler ServerValidateEventHandler AdCreatedEventArgs CommandEventArgs DataGridCommandEventArgs DataListCommandEventArgs RepeaterCommandEventArgs DataGridItemEventArgs DataGridPageChangedEventArgs DataGridSortCommandEventArgs DataListItemEventArgs RepeaterItemEventArgs FontUnit Unit DataGridControlBuilder DataListControlBuilder HyperLinkControlBuilder LabelControlBuilder LinkButtonControlBuilder RepeaterControlBuilder TableCellControlBuilder TextBoxControlBuilder Repeater RepeaterItem WebControl AdRotator BaseDataList DataGrid DataList Button Calendar CheckBox RadioButton DataListItem HyperLink Image ImageButton Label BaseValidator CompareValidator CustomValidator RangeValidator RegularExpressionValidator RequiredFieldValidator LinkButton ListControl RadioButtonList CheckBoxList DropDownList ListBox Panel Table TableCell TableHeaderCell TableRow DataGridItem TextBox ValidationSummary Xml CalendarDay Column EditCommandColumn HyperLinkColumn TemplateColumn BoundColumn ButtonColumn ColumnCollection DataGridItemCollection DataKeyCollection DataListItemCollection DayRenderEventArgs FontInfo

GeneratedImage ListItem ListItemCollection MonthChangedEventArgs PagedDataSource RepeaterItemCollection RepeatInfo SelectedDatesCollection TableCellCollection TableRowCollection BorderStyle ButtonColumnType CalendarSelectionMode DayNameFormat FirstDayOfWeek FontSize GridLines HorizontalAlign ImageAlign ListItemType ListSelectionMode NextPrevFormat PagerMode PagerPosition RepeatDirection RepeatLayout TextAlign TextBoxMode TitleFormat UnitType ValidationCompareOperator ValidationDataType ValidationSummaryDisplayMode ValidatorDisplay VerticalAlign

接口

IDesignerAccessor IRepeatInfoUser

System.Web.Util

类、结构、枚举

AspCompatCallback TransactedCallback WorkItemCallback CabCreator EventLog MailAttachment MailMessage Profiler Sec SmtpMail Transactions UrlPath Wildcard WildcardPath WildcardUrl WildcardDos WorkItem MailEncoding MailFormat MailPriority TransactionSupport

接口

IManagedContext

System.WinForms

类、结构、枚举

WinCategoryAttribute ColumnHeader CommonDialog FileDialog OpenFileDialog SaveFileDialog FontDialog PageSetupDialog PrintDialog ColorDialog DataGridColumn DataGridDropdownColumn DataGridTextBoxColumn DataGridValueColumn DataGridBoolColumn DataGridTable ErrorProvider FolderBrowser HelpProvider StatusBarPanel TabItem Timer ToolBarButton TrayIcon DataGridPreferredColumnWidthTypeConverter KeysConverter ListBindingConverter ListPropertyConverter OpacityConverter SelectionRangeConverter TreeNodeConverter BorderConverter EdgeConverter ListItemConverter ImageIndexConverter BindingsCollection ControlBindingsCollection ListManagerBindingsCollection GridColumnsCollection GridTablesCollection AutoSizeEventHandler ColumnClickEventHandler ControlEventHandler CurrentChangingEventHandler DateBoldEventHandler DateRangeEventHandler DateTimeFormatEventHandler DateTimeFormatQueryEventHandler DateTimeUserStringEventHandler DateTimeWmKeyDownEventHandler DocumentReadyEventHandler DragEventHandler DrawItemEventHandler ErrorEventHandler GiveFeedbackEventHandler HelpEventHandler InputLangChangeEventHandler InputLangChangeRequestEventHandler

InvalidateEventHandler ItemChangedEventHandler ItemCheckEventHandler
ItemDragEventHandler KeyEventHandler KeyPressEventHandler
LabelEditEventHandler LayoutEventHandler LinkClickEventHandler
LinkLabelLinkClickedEventHandler MeasureItemEventHandler MethodInvoker
MouseEventHandler NavigateEventHandler NodeLabelEditEventHandler
PaintEventHandler PropertyTabChangedEventHandler
PropertyValueChangedEventHandler QueryAccessibilityHelpEventHandler
QueryContinueDragEventHandler ReadyStateEventHandler RequestResizeEventHandler
ScrollEventHandler SelectionChangedEventHandler SplitterEventHandler
StatusBarDrawItemEventHandler StatusBarPanelClickEventHandler
ToolBarButtonClickEventHandler TreeViewCancelEventHandler TreeViewEventHandler
UICuesEventHandler UpDownChangedEventHandler UpDownEventHandler
ViewModelessEventHandler ViewStateLoadEventHandler ViewStateSaveEventHandler
FilePersistEventHandler PrintControllerWithStatusDialog AutoSizeEventArgs
ColumnClickEventArgs ControlEventArgs CurrentChangingEventArgs
DateBoldEventArgs DateRangeEventArgs DateTimeFormatEventArgs
DateTimeFormatQueryEventArgs DateTimeUserStringEventArgs
DateTimeWmKeyDownEventArgs DocumentReadyEventArgs DragEventArgs
DrawItemEventArgs StatusBarDrawItemEventArgs ErrorEventArgs
FilePersistEventArgs GiveFeedbackEventArgs HelpEventArgs
InputLangChangeEventArgs InputLangChangeRequestEventArgs InvalidateEventArgs
ItemChangedEventArgs ItemCheckEventArgs ItemDragEventArgs KeyEventArgs
KeyPressEventArgs LabelEditEventArgs LayoutEventArgs LinkClickEventArgs
LinkLabelLinkClickedEventArgs MeasureItemEventArgs MouseEventArgs
StatusBarPanelClickEventArgs NavigateEventArgs NodeLabelEditEventArgs
PaintEventArgs PropertyTabChangedEventArgs PropertyValueChangedEventArgs
QueryAccessibilityHelpEventArgs QueryContinueDragEventArgs ReadyStateEventArgs
RequestResizeEventArgs ScrollEventArgs SelectionChangedEventArgs
SplitterEventArgs ToolBarButtonClickEventArgs TreeViewEventArgs
TreeViewCancelEventArgs UICuesEventArgs UpDownChangedEventArgs
UpDownEventArgs ViewModelessEventArgs ViewStateLoadEventArgs
ViewStateSaveEventArgs DataStreamFromComStream NativeWindow
OwnerDrawPropertyBag TreeNode Control RichControl ScrollableControl
ContainerControl Form PrintPreviewDialog ThreadExceptionDialog PropertyGrid
UpDownBase DomainUpDown NumericUpDown UserControl Panel TabPage
MultiplexPanel ScrollBar VScrollBar HScrollBar Splitter StatusBar TabBase
TabControl TabStrip ToolBar TrackBar TreeView ValueEditorHost ValueEdit
AxHost ButtonBase CheckBox RadioButton Button DataGrid DateTimePicker
FormatControl Label LinkLabel ListControl ComboBox ListBox CheckedListBox

TextBoxBase RichTextBox TextBox DataGridTextBox GroupBox ListView MDIClient MonthCalendar PictureBox PrintPreviewControl ProgressBar ImageList Menu MenuItem ContextMenu MainMenu ToolTip BitVector32 DataGridCell Message AccessibleObject AmbientProperties Application BindingManager Border Clipboard ControlPaint CreateParams DataFormats DataGridRow DataGridAddNewRow DataGridRelationshipRow DataObject Edge FeatureSupport OSFeature FileVersionInfo GridTablesFactory Help ImageListStreamer InputLanguage ListBinding ListItem ListManager RelatedListManager MessageBox RichTextBoxIMEColor Screen SelectionRange SendKeys StringDictionary StringSorter SystemInformation TreeNodeCollection Value WinFormsSecurity AccessibleEvents AccessibleNavigation AccessibleRoles AccessibleSelection AccessibleStates AnchorStyles Appearance ArrangeDirection ArrangeStartingPosition BootMode Border3DSide Border3DStyle BorderStyle BoundsSpecified ButtonBorderStyle ButtonState CaptionButton CharacterCasing CheckState ColorDepth ColumnHeaderStyle ComboBoxStyle ControlStyles DataGridLineStyle DataGridNavigationModes DataGridParentRowsLabelStyle DateTimePickerFormats Day DialogResult DisplayType DockStyle DragAction DragDropEffects DrawItemState DrawMode EdgeStyle ErrorBlinkStyle ErrorIconAlignment FlatStyle FloatMode FloodFillType FolderBrowserFolders FolderBrowserStyles FormBorderStyle FormStartPosition FormWindowState FrameStyle HorizontalAlignment IMEMode ItemActivation ItemBoundsPortion Keys LeftRightAlignment LinkBehavior LinkState ListViewAlignment MDILayout MenuGlyph MenuMerge MouseButtons Orientation PictureBoxSizeMode PositionModes PropertySort RichTextBoxFinds RichTextBoxIMEOptions RichTextBoxScrollBars RichTextBoxSelectionAttribute RichTextBoxSelectionTypes RichTextBoxStreamType RichTextBoxWordPunctuations RightToLeft ScrollBars ScrollButton ScrollEventArgsType SecurityIDType SelectionMode Shortcut SizeGripStyle SizeRelationship SortOrder SpecialFolder StatusBarPanelAutoSize StatusBarPanelBorderStyle StatusBarPanelStyle StructFormat TabAlignment TabAppearance TabDrawMode TabSizeMode TickStyle ToolBarAppearance ToolBarButtonStyle ToolBarTextAlign TreeViewAction TriangleDirection UICues VerticalAlignment View

接口

IBorderStyleEdges IButtonControl ICommandExecutor ICompletion IContainerControl IDataGridColumnEditingNotificationService IDataGridEditingService IDataGridHost IDataObject IFeatureSupport IFileReaderService IMessageFilter IWin32Window IWindowTarget

System.WinForms.ComponentModel

类、结构、枚举

InstallerTypeAttribute RunInstallerAttribute Format GenericFormat NumericFormat ObjectFormat DateTimeFormat TypeListEditor RedirectValueEditor FormatEventHandler BooleanReturnMethodInvoker OrderSelector FormatMode UserMode CollectionInfo FormatEventArgs FormatInfo ThreadNotify

接口

IComponentEditorPageSite IEditorHost

System.WinForms.Design

类、结构、枚举

WinFormsComponentEditor CreateControlActionUnit CreateMenuItemActionUnit CreateComponentActionUnit Win32FormsPersister CompositionDesigner ControlDesigner RichControlDesigner RichEditDesigner RichTextBoxDesigner ScrollBarDesigner SplitterDesigner TextBoxBaseDesigner TrackBarDesigner TreeViewDesigner UpDownBaseDesigner AxHostDesigner ButtonDesigner ComboBoxDesigner DataGridDesigner DateTimePickerDesigner LabelDesigner ListBoxDesigner ListViewDesigner MonthCalendarDesigner ParentControlDesigner ScrollableControlDesigner DocumentDesigner FormDocumentDesigner UserControlDocumentDesigner PanelDesigner TabPageDesigner TabBaseDesigner TabControlDesigner GroupBoxDesigner PictureBoxDesigner ProgressBarDesigner RadioButtonDesigner ImageListDesigner OpenFileDialogDesigner SaveFileDialogDesigner TrayIconDesigner MenuCommands ClipboardValueProvider AdvancedBindingPropertyDescriptor DesignBindingPropertyDescriptor AdvancedBindingConverter ControlBindingsConverter DesignBindingConverter DataSourceConverter WinFormValueEditor FormatValueValueEditor FormatValueEditor BeforeReferenceAddedEventHandler WindowRemovedEventHandler ToolboxItemCreatorCallback WinFormsToolboxItem AdvancedBindingEditor AnchorEditor DataMemberFieldEditor DataMemberListEditor DesignBindingEditor DockEditor FileNameEditor HelpFileFileNameEditor FolderNameEditor ImageIndexEditor LinkAreaEditor LinkedDataMemberFieldEditor ValueEditorUITypeEditor DataGridColumnCollectionEditor ImageCollectionEditor StringCollectionEditor ListControlStringCollectionEditor StringArrayEditor TreeNodeCollectionEditor WindowRemovedEventArgs AdvancedBindingPicker ComponentEditorForm DataGridAutoFormatDialog ComponentTray ComponentEditorPage DesignBindingPicker DesignerFrame TabOrder AdvancedBindingObject AxImporter AxParameterData AxWrapperGen BeforeReferenceAddedEventArgs ClipboardDataHolder CommandSet DesignBinding DesignBindingValueUIHandler EventHandlerService OleDragDropHandler

PropertyTab EventsTab SelectionUIHandler

接口

ComponentClipboardService IEventHandlerService IMenuEditorService IMouseHandler IMultiWindowService IOleDragClient IOverlayService ISelectionUIHandler ISelectionUIService ISplitWindowService IToolboxService ITrayHolderService ITypeLoader IUIService IWinFormsDesigner IWinFormsEditorService

System.Xml

类、结构、枚举

ValidationEventHandler XmlNodeChangedEventHandler ValidationEventArgs XmlException MTNameTable NameTable XmlCharType XmlConvert XmlImplementation XmlNamedNodeMap XmlAttributeCollection XmlNavigator DataDocumentNavigator DocumentNavigator XmlNode XmlNotation XmlAttribute XmlDocument XmlDataDocument XmlDocumentFragment XmlEntity XmlLinkedNode XmlProcessingInstruction XmlCharacterData XmlComment XmlSignificantWhitespace XmlText XmlWhitespace XmlCDataSection XmlDeclaration XmlDocumentType XmlElement XmlEntityReference XmlNodeChangedEventArgs XmlNodeList XmlReader XmlTextReader XmlNodeReader XmlResolver XmlUrlResolver XmlSchemaCollection XmlWriter XmlTextWriter EntityHandling Formatting ReadState TreePosition Validation WhitespaceHandling WriteState XmlNodeChangedAction XmlNodeType XmlSpace

接口

XmlNameTable

System.Xml.Serialization

类、结构、枚举

XmlArrayAttribute XmlArrayItemAttribute XmlAttributeAttribute XmlElementAttribute XmlEnumAttribute XmlIgnoreAttribute XmlIncludeAttribute XmlRootAttribute XmlTextAttribute XmlTypeAttribute XmlArrayItemAttributes XmlElementAttributes XmlAttributeEventHandler XmlNodeEventHandler XmlSerializationFixupCallback XmlAttributeEvent XmlNodeEvent XmlQualifiedName CodeIdentifier CodeIdentifiers XmlAttributeOverrides XmlAttributes XmlCodeExporter XmlMapping XmlMembersMapping XmlTypeMapping XmlMemberInfo XmlMemberMapping XmlReflectionImporter XmlSchemaExporter XmlSchemaImporter XmlSerializationReader XmlSerializationWriter XmlSerializer XmlSerializerNamespaces XmlArrayType XmlForm XmlReference

System.Xml.Serialization.Schema

类、结构、枚举

XmlSchemaAnnotations XmlSchemaItems XmlSchemas XmlSchemaTopItems XmlSchema XmlSchemaAnnotated XmlSchemaAttributeGroup XmlSchemaNamedAttributeGroup XmlSchemaDataType XmlSchemaNamedDataType XmlSchemaFacet XmlSchemaMaxBoundFacet XmlSchemaMaxExclusiveFacet XmlSchemaMaxInclusiveFacet XmlSchemaMinBoundFacet XmlSchemaMinExclusiveFacet XmlSchemaMinInclusiveFacet XmlSchemaNumericFacet XmlSchemaPrecisionFacet XmlSchemaScaleFacet XmlSchemaLengthFacet XmlSchemaMaxLengthFacet XmlSchemaMinLengthFacet XmlSchemaPatternFacet XmlSchemaPeriodFacet XmlSchemaEncodingFacet XmlSchemaEnumerationFacet XmlSchemaNotation XmlSchemaOccuringItem XmlSchemaAny XmlSchemaConstraint XmlSchemaKey XmlSchemaKeyref XmlSchemaUnique XmlSchemaElement XmlSchemaField XmlSchemaGroupBase XmlSchemaSequence XmlSchemaAll XmlSchemaChoice XmlSchemaGroup XmlSchemaNamedGroup XmlSchemaType XmlSchemaNamedType XmlSchemaAnnotation XmlSchemaAnyAttribute XmlSchemaAppInfo XmlSchemaAttribute XmlSchemaDocumentation XmlSchemaImport XmlSchemaInclude XmlSchemaContent XmlSchemaContentProcessing XmlSchemaTypeBlock XmlSchemaTypeDerivation XmlSchemaUseType

接口

IXmlSchemaNamedItem

System.Xml.XPath

类、结构、枚举

XPathException Querytype

System.Xml.Xsl

类、结构、枚举

XsltException NSParamList XslTransform

参 考 资 料

[1] 王燕编著，《面向对象的理论与 C++实践》，清华大学出版社，1997 年。

[2] 周之英编著，《现代软件工程》，科学出版社，2000 年。

[3] 郑人杰等编著，《软件工程》，清华大学出版社，1999 年。

[4] 潘爱民编著，《COM 原理与应用》，清华大学出版社，1999 年。

[5] Microsoft Corporation，“C# Language Specification.”White paper，2001：http://www.microsoft.com/msdn.