# 第 1 章　.NET 体系结构

.NET 是 Microsoft 的一种开发模型，它使软件变得独立于平台和设备，并且支持利用 Internet 上的数据编程，.NET 可以使程序运行于各种异构平台。.NET 平台为创建新一代分布式 Web 应用程序提供了所有工具和技术（表示技术、构件技术和数据库技术）。.NET 平台支持标准的 Internet 协议，包括 HTTP（超文本传输协议）、XML（可扩展标记语言）和 SOAP（简单对象访问协议），从而实现了异构系统间应用程序的集成和通信。.NET 框架是.NET 的基本体系结构，它提供了具体的技术和服务（手段），在本章中将学习.NET 的体系结构。

**本章主要内容：**

- .NET Framework 的构成及其组件的概念。
- .NET Framework 的类库和命名空间之间的关系。

## 1.1　.NET 简介

.NET 是微软在本世纪初，应网络分布式运算需求而推出的一个应用程序开发和运行平台规范。该规范内容相当广泛，包含了诸如组件格式、编程语言、标准类和工具等各个方面。为推广和使用这个规范，作为该规范的首倡者，微软在公布这个规范的同时也推出了该规范在 Windows 平台上的一个实现——.NET Framework。.NET 框架是.NET 的基本体系结构，它提供了具体的技术和服务。现在人们所说的.NET，通常就是指微软的这个实现以及这个实现所包含的各项技术。

## 1.2　.NET Framework 概述

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

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

（2）提供一个将软件部署和版本控制冲突最小化的代码执行环境。

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

（4）提供一个可消除脚本环境或解释环境的性能问题的代码执行环境。

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

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

如果要使用 C#高效地开发应用程序，理解.NET Framework 就非常重要，.NET Framework 包括以下组件：

- 公共语言运行库（Common Language Runtime，CLR）。
- .NET Framework 类库（Framework Class Library，FCL）。
- 数据库访问组件（ADO.NET 和 XML）。
- 基于 ASP.NET 编程框架的网络服务（Web Services）和网络表单（WebForms）。
- Windows 桌面应用界面编程组件（WinForm）。

.NET Framework 的体系结构如图 1-1 所示。

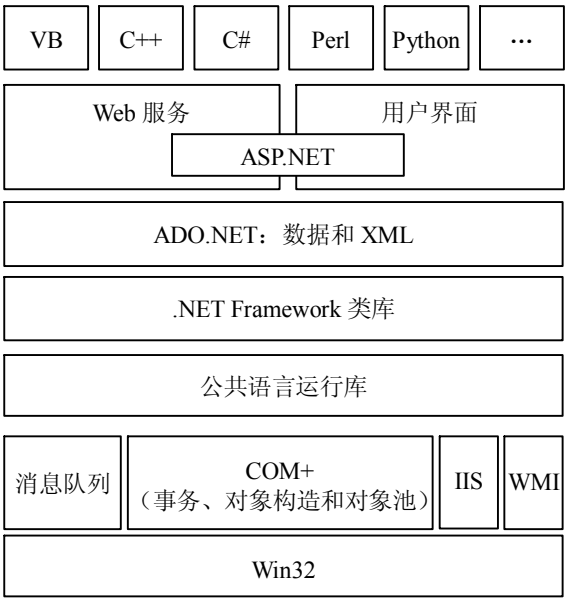


图 1-1　.NET Framework 的体系结构

公共语言运行库（CLR）简化了应用程序的开发，并为应用程序提供了一个强大和安全的执行环境。公共语言运行库是 .NET Framework 的基础。可以将公共语言运行库看做一个在执行时管理代码的代理，它提供内存管理、线程管理和远程处理等核心服务，并且还强制实施严格的类型安全以及可提高安全性和可靠性的其他形式的代码准确性。这类似于Java的虚拟机。事实上，代码管理的概念是公共语言运行库的基本原则。以公共语言运行库为目标的代码称为托管代码，不以公共语言运行库为目标的代码称为非托管代码。

.NET Framework 类库是一个综合性的面向对象的可重用类型的集合，它包含数量庞大的类，提供创建各种应用程序所需的广泛功能，简化了.NET 应用程序的开发，开发人员还可以通过创建自己的类库来扩展它们。

在基于.NET 的应用程序中，要访问数据库，使用 ADO.NET 组件是目前的最佳选择，它为非连接编程模型提供了改进的支持，也提供了丰富的 XML 支持。

ASP.NET 是一个建立在公共语言运行库上的编程框架，可以使用 ASP.NET 在服务器上构建功能强大的 Web 应用程序。ASP.NET Web 窗体提供了易用且功能强大的方法来构建动态的 Web 用户界面（UI）。

.NET Framework 提供了一系列工具和类来构建、测试和分发 XML Web Service。

.NET Framework 支持 3 种类型的用户界面：Web 窗体（通过 ASP.NET 工作）、Windows 窗体（运行在 Win32 客户端上）和控制台应用程序。

.NET 框架中最重要的元素是公共语言运行库（Common Language Runtime，CLR），公共语言运行库管理用.NET 语言编写的可执行代码像 Java 的虚拟机一样，它是.NET 体系结构的基础。公共语言运行库激活对象并在其上实施安全检查，在内存中部署、执行并且回收。

## 1.3 公共语言运行库

公共语言运行库（CLR）是整个.NET 框架的核心，它为.NET 应用程序提供了一个托管的代码执行环境。它实际上是驻留在内存里的一段代理代码，负责应用程序在整个执行期间的代码管理工作，比较典型的有：内存管理、线程管理、安全管理、远程管理、即时编译、代码强制安全类型检查等。这些都可称为.NET 框架的生命线。

公共语言运行库的组成如图 1-2 所示。

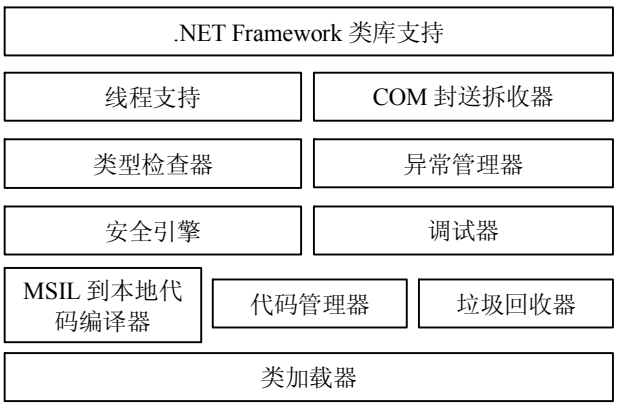


图 1-2 公共语言运行库的组成

下面对各个组成部分提供的功能进行简要介绍。

- 类加载器：管理元数据，加载和在内存中布局类。
- Microsoft 中间语言（MSIL）到本地代码编译器：通过即时编译把 Microsoft 中间语言转换成本地代码。
- 代码管理器：管理和执行代码。
- 垃圾回收器：为.NET Framework 下的所有对象提供自动生命期管理，支持多处理器，可扩展。
- 安全引擎：提供基于证据的安全，基于用户身份和代码来源。
- 调试器：使开发者能够调试应用程序和根据代码执行。
- 类型检查器：不允许不安全的类型转换和未初始化变量 MSIL 可被校验以保证类型安全。
- 异常管理器：提供和 Windows 结构化异常处理集成的异常处理机制。
- 线程支持：提供多线程编程支持。

- COM 封送拆收器：提供和 COM 组件之间的封送转换。
- .NET Framework 类库支持：通过和运行时集成代码来支持.NET Framework 类库。

实际上，CLR 代理了一部分传统操作系统的管理功能。通常将在 CLR 的控制下运行的代码称为托管代码，否则称为非托管代码。

### 1.3.1 非托管代码的运行原理

先来看一下 Windows 操作系统执行一个非托管程序的基本过程，如图 1-3 所示。

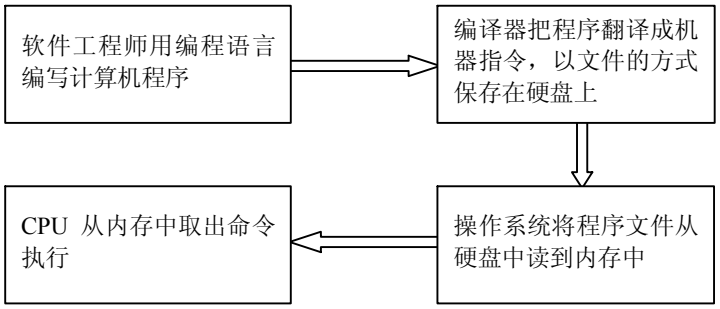


图 1-3 非托管程序的开发与执行过程

软件工程师写的程序经过编译器转为机器指令后一般以文件的形式保存在外部存储器中，当 CPU 执行程序时，首先把外部存储器中的程序指令代码读入到内存中。内存被分成许多块（称为内存单元），每个内存单元都有一个唯一的地址，指令就存放在以某个特定的地址（入口地址）开始的内存区域中。CPU 从入口地址处取出第一条指令，开始执行，然后再取出第二条指令，依此类推。

把一个程序从硬盘上装入内存执行，这是一个复杂的过程，这个功能由操作系统实现，开发具体应用程序的软件工程师不需要手动去写这部分代码。

可以看出，程序的运行必须依赖于操作系统，而且编译器生成的程序文件包含的是仅适用于特定 CPU 架构的机器指令。由于不同 CPU 架构的机器指令集不同，所以生成的这个可执行程序不能不加修改地在拥有不同 CPU 架构的计算机上运行。

以这种方式生成的机器指令代码就是前面提到过的非托管代码。某个包含非托管代码的程序如果不进行修改，不仅不能在不同 CPU 架构的计算机上执行，而且在不同的操作系统下也不能执行，比如一个 Windows 应用程序就无法直接在 Linux 下运行，反之亦然。图 1-4 很好地说明了非托管代码的运行原理。

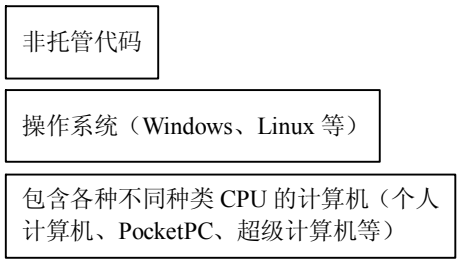


图 1-4 非托管代码的运行原理

### 1.3.2 托管代码的运行原理

显然，如果需要在拥有不同 CPU 架构的计算机和多种多样的操作系统上实现某一功能，必须针对每种操作系统和 CPU 架构编写特定的代码，这明显是一种重复和低效的劳动。那么程序能不能只写一次，却可以处处运行呢？

完全可以，这就是“跨平台”的设计思想。.NET 采用了这种设计思想，要支持跨平台这一特性，软件工程师写的程序经过编译器生成的结果就不能依赖于特定操作系统和特定 CPU 架构的机器指令，而必须是一种中间的、在各种操作系统和计算机硬件平台上都能执行的代码，这种代码.NET 称之为 MSIL（微软中间语言）指令。但程序最终还是要靠 CPU 执行，所以.NET 的 MSIL 指令仍然需要最终被翻译成本地 CPU 能执行的机器指令，这部分功能由一个运行在特定操作系统之上的软件系统来完成，这个软件系统被称为虚拟机（Virtual Machine，VM），CLR 就是.NET 虚拟机。这种运行在虚拟机之上的代码称为“托管代码”，MSIL 就是一种托管代码。

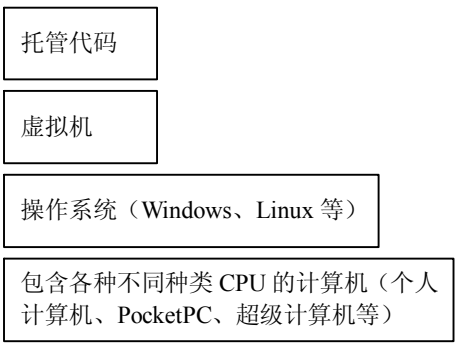


图 1-5 托管代码运行原理

图 1-5 所示为托管代码运行原理图，托管代码是一种对特定计算机和操作系统的中间语言，必须通过虚拟机才能将其转换为本地 CPU 能执行的机器指令。

如何跨语言应用呢？为了实现跨语言应用，微软采取了两项措施：一是开发了一种叫做 MSIL 的中间语言，它相当于是一种为虚拟机配套的汇编语言，该语言也叫做 IL（Intermediate Language）；二是提出了一个编译器的制作规范，这个规范叫做 CLS（公共语言规范，Common Language Specification）。

于是，那些各种不同的编程语言，只要配有按照 CLS 规范来制作的编译器，那么这种语言程序就能被编译成 IL 语言程序，也就能跨语言应用。

其实，只对语言进行统一还不能完全解决跨语言问题。因为，不同程序设计语言的差异不仅表现在语句的表达上，更重要是数据的内存布局。即不同语言的同一类型数据占用内存的大小和布局均有区别。例如，不同语言的函数在参数的传递顺序上就有所不同，所以以前在一个 Pascal 程序中调用 C 语言方法时就需要格外的小心，同理，在 C 程序中调用 Pascal 方法时也是如此。再例如，C 语言中的字符串是一个以 ASCII 码字符 NULL（在程序中通常用‘\0’表示）为结尾的字符数组；而在 Pascal 语言中，数组中的第一个字节就包含了字符串的长度，从而也就没有必要有一个类似的‘\0’结束标志。

综上所述，.NET 在使用 MSIL 进行了语言的统一之后，还需要对数据类型进行统一。为此，.NET 中就提供了一套统一的数据类型，这套数据类型就叫做公共类型系统（Common Type System，CTS）。也就是说，不同语言应用程序中的所有不符合 CTS 的数据类型，在经过编译器编译后都必须转换成统一后的类型，这样才能实现真正的跨语言应用。

由托管代码的运行原理可见，在.NET 中，编译被分为以下两个阶段（如图 1-6 所示）：

（1）把源代码编译为 MSIL。

（2）CLR 把 MSIL 编译为平台专用的代码。

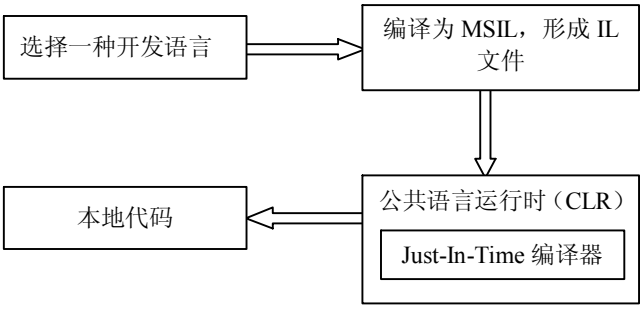


图 1-6　.NET 编译的两个阶段

这两个阶段的编译过程非常重要，因为 MSIL 托管代码是提供.NET 的许多优点的关键。托管代码与非托管代码相比除了介绍过的“跨平台”的优点外，还具有“跨语言”的优点。简言之，就是能将任何一种语言编写的源代码编译为 MSIL，编译好的中间语言代码可以与从其他语言编译过来的中间代码进行交互，从而能够重用其他语言开发的组件。

在第二阶段的编译过程中，JIT 编译器在将 MSIL 编译为本地代码时，并不是把整个 IL 文件一次编译完，而是采用即时编译，即每当 CLR 要执行一条函数调用指令时，由 JIT 编译器负责将这一函数所对应的 IL 指令动态编译成本地 CPU 可执行迟重的代码，CLR 同时将这些本地代码缓存起来，这样下次再调用此方法时就不需要重新编译了。这个过程要比一开始就编译整个应用程序代码的效率高得多，从而使得托管 IL 代码的执行速度几乎和内部代码的执行速度一样快。另外，JIT 编译器的编译过程是在运行时进行的，JIT 编译器会确切地知道程序运行在什么样的处理器上，因此可以针对特定的处理器来优化最后的可执行代码，而传统编译器的优化过程是独立于代码所运行的特定处理器的，例如 VS 6.0 为一般的 Pentium 机器进行了优化，所以它生成的代码就不能利用 Pentium III 处理器的硬件特性，传统编译器生成的代码就不能充分利用特定处理器的硬件特性，托管代码与非托管代码相比在性能上也有所提升。

另外，与非托管代码相比，托管代码还具有垃圾回收功能和代码访问安全性等优点，这部分内容会在后面章节作详细的介绍，所以这里只简单地介绍：垃圾回收功能就是为托管代码分配的内存，不需要程序员通过编程来释放它（例如 C++程序员利用 new 运算符分配内存时，一定要记得在不需要使用内存时用 delete 运算符将其释放掉），CLR 会利用垃圾回收器自动释放掉不再使用的内存，从而减少了程序员的工作。代码访问安全性就是在 JIT 编译器将 IL 指令转换成本地代码的过程中，CLR 将执行代码验证过程，以确保代码是类型安全的（例如避免将一种类型转换成与其不兼容的类型或出现非法指针等情况）。

## 1.4 .NET Framework 类库

正如 C 语言有 C 标准库一样，.NET 为所有运行在 CLR 环境下的应用程序提供了一个功能强大的公用代码库，即.NET Framework 类库（Framework Class Library，FCL），类库中包含数量庞大的类，这些类是以面向对象理论为基础而设计的，几乎封装了操作系统的所有对外编程接口（Application Program Interface，API），提供创建各种应用程序所需的广泛功能，而且这种类库是针对.NET 平台而非特定的编程语言的，这就使得使用 C#、VB.NET 和 C++.NET 等.NET 编程语言的程序员以一致的编程方式解决了不同语言之间相互集成的问题。

从开发人员的角度来看，编写托管代码的最大好处是可以使用.NET 基类库。开发人员使用基类库的方法是简单地实例化它们、调用它们的方法或者开发扩展其功能的派生类。

.NET 基类是一个内容丰富的托管代码类集合，它可以完成以前要通过 Windows API 来完成的绝大多数任务。这些类派生自与中间语言相同的对象模型，也基于单一继承性。无论.NET 基类是否合适，都可以实例化对象，也可以从它们派生自己的类。

.NET Framework 类库（FCL）中的 7000 多种类型——类、结构、接口、枚举和委托，组成了.NET Framework 的核心部分。一些 FCL 的类包含了 100 多个方法、属性和其他成员。所以学习 FCL 并不是轻松的事情。本书将主要说明如何使用.NET 基类库中的各种类，即各种基类是如何工作的。.NET 基类主要包括：

- IL 提供的核心功能，例如通用类型系统中的基本数据类型。
- Windows GUI 支持和控件。
- Web 窗体（ASP.NET）。
- 数据访问（ADO.NET）。
- 目录访问。
- 文件系统和注册表访问。
- 网络和 Web 浏览。
- COM 互操作性。

附带说一下，根据 Microsoft 源文件，.NET 基类几乎完全是用 C#编写的。

## 1.5 命名空间

因为.NET Framework 类库中包含数千个类，所以程序设计人员需要以快捷的方法找到所需要的类。将这些类库分组到命名空间中，将功能相似的类组织在一起，如进行文件、目录存取的 File、Directory 类就分类到 System.IO 命名空间下。命名空间也是.NET 避免类名冲突的一种方式，例如有两个程序员 A 和 B，他们处于同一个程序开发小组中，程序员 A 开发了一个名为 BankCustomer 的类，程序员 B 也开发了一个名为 BankCustomer 的类，那么在进行代码合并的时候，对于应用程序所创建的 BankCustomer 类对象而言，就不知道其引用的是哪个程序员所开发的 BankCustomer 类的成员，这样就造成了类名的冲突。利用命名空间可以解决这个冲突，使程序员 A 所开发的 BankCustomer 类位于命名空间 A 下，程序员 B 所开发的 BankCustomer 类位于命名空间 B 下，这样在应用程序中就可以用 A.BankCustomer 和

B.BankCustomer 来明显区分这两个类了，如图 1-7 所示。

```
namespace A
{
    public class BankCustomer{}
}
```

```
namespace B
{
    public class BankCustomer{}
}
```

图 1-7　命名空间解决类名冲突

命名空间不过是数据类型的一种组合方式，但命名空间中所有数据类型的名称都会自动加上该命名空间的名字作为其前缀。命名空间还可以相互嵌套。例如大多数用于一般目的的.NET 基类位于命名空间 System 中，基类 Array 在这个命名空间中，所以其全名是 System.Array。

.NET 需要在命名空间中定义所有的类型，例如可以把 Customer 类放在命名空间 YourCompanyName 中，则这个类的全名就是 YourCompanyName. Customer。

**注意**：如果没有显式提供命名空间，类型就添加到一个没有名称的全局命名空间中。

Microsoft 建议在大多数情况下都至少要提供两个嵌套的命名空间名，第一个是公司名，第二个是技术名称或软件包的名称，而类是其中的一个成员，例如 YourCompanyName.SalesServices.Customer。在大多数情况下，这么做可以保证类的名称不会与其他组织编写的类名冲突。

## 1.6　ADO.NET——数据和 XML

ADO.NET 是新一代的 ADO 技术，它为非连接的编程提供了更好的支持，而且在 System.Xml 命名空间中提供了丰富的 XML 支持，如图 1-8 所示。

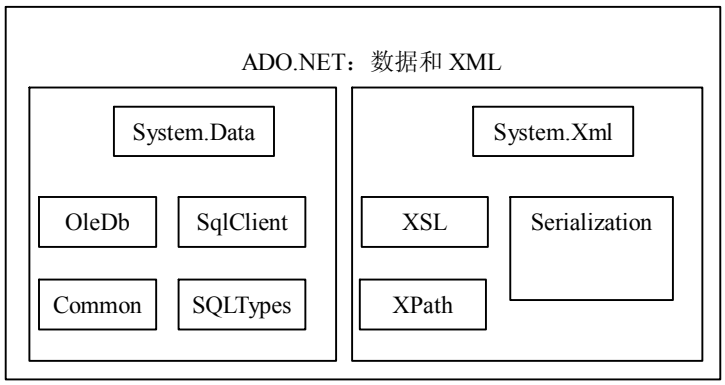


图 1-8　ADO.NET

System.Data 命名空间包含组成 ADO.NET 对象模型的类。粗略地划分，ADO.NET 对象模型被分为两层：链接层和非链接层。

System.Data 命名空间包含了 DataSet 类，它描述了多个表以及表之间的关系。这些数据集是完全自包含的数据结构，它们可以由多种数据源来填充。其中一种数据源是 XML；另一

种是 OLE DB；第三种是 Microsoft SQL Server 直连适配器。

System.Xml 命名空间提供了 XML 的支持。它包含了符合 W3C 标准的 XML 分析器和编写器。System.Xml.Xsl 命名空间提供了可扩展样式表语言转换。Xpath 的实现支持了在 XML 中数据图结构的导航。System.Xml.Serialization 命名空间为 XML Web Service 提供了完整的核心基础结构。

## 1.7 XML Web Service

XML Web Service 是.NET 平台的核心部分。它们是不同 Web 应用程序之间相互提供、使用数据与功能的基础机制，这些 Web 应用程序分布在单位内部或不同的单位之中。

XML Web Service 是为其他应用程序提供数据和服务的应用程序逻辑单元。应用程序可以通过工业标准 Web 协议和数据格式来访问 XML Web Service，如 HTTP、XML 和简单对象访问协议（Simple Object Protocol，SOAP），而不用考虑每个 XML Web Service 是如何实现的，如图 1-9 所示。

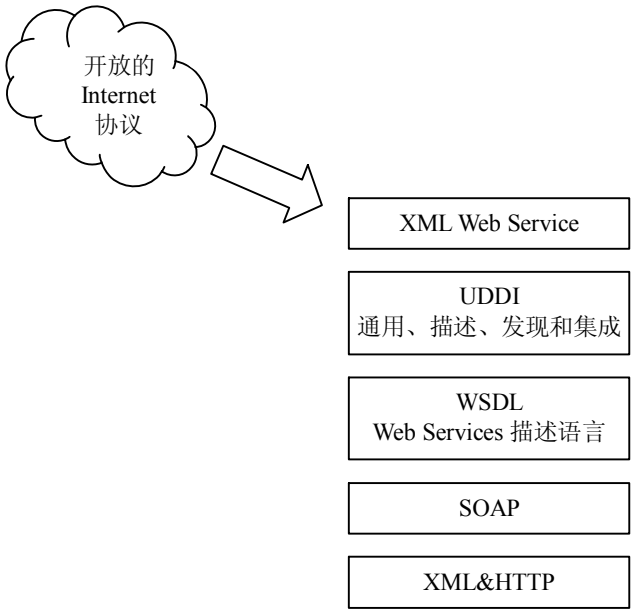


图 1-9 XML Web Service

1. XML 和 HTTP

XML Web Service 是使用 XML 和 HTTP 构建的，所以 XML Web Service 的运行可以不受防火墙的限制。另外，因为 XML 和 HTTP 是工业标准，所以任何支持 XML 和 HTTP 的平台都可以和 XML Web Service 协同工作。

2. SOAP

SOAP 定义了在与 XML Web Service 协同工作时如何格式化、发送和接收消息。SOAP 也是建立在 XML 和 HTTP 上的工业标准，所以任何支持 SOAP 的平台都可以支持 XML Web Service。

3. Web 服务描述语言

Web 服务描述语言（Web Services Description Language，WSDL）是一种 XML 格式，用来描述服务器提供的网络服务。可以使用 WSDL 创建一个文件，用来指明服务器提供的服务以及服务器支持的每个服务的操作集。

4. 通用、描述、发现和集成

通用、描述、发现和集成（Universal Discovery Description and Integration，UDDI）是注册和查找 XML Web Service 的工业标准。通过使用 UDDI，开发人员可以发现并使用在 Internet 上已经公开的可用的 XML Web Service。

## 1.8 Web 窗体和服务

ASP.NET 是建立在公共语言运行库之上的编程框架，公共语言运行库可用于在服务器上构建功能强大的 Web 应用程序。ASP.NET Web 窗体提供了易用且功能强大的方法来构建动态的 Web UI 页面。ASP.NET 的 XML Web Service 为构建分布式 Web 应用程序提供了构建模块。下面对一些比较常用的 ASP.NET 类进行描述，如图 1-10 所示。

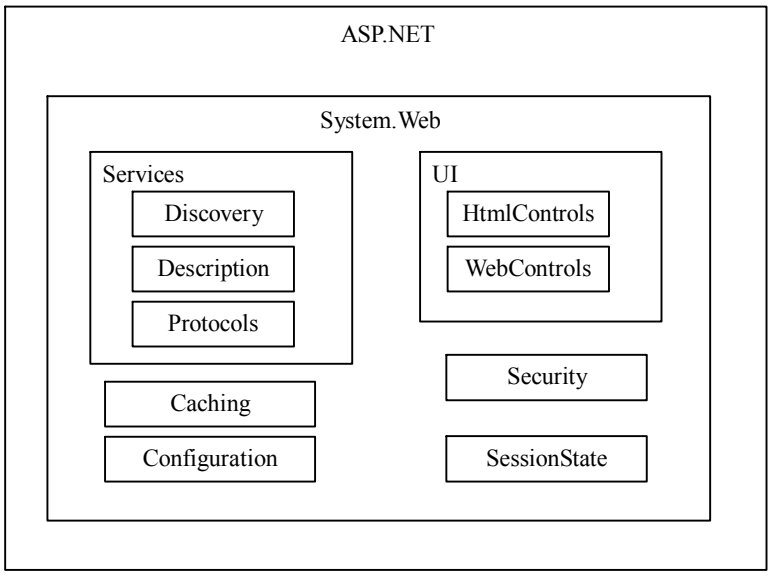


图 1-10 ASP.NET

1. System.Web

在 System.Web 命名空间中，有一些更底层的服务，如缓存、安全性和配置等，它们可以在 XML Web Service 和 Web UI 之间进行共享。

2. System.Web.Services

System.Web.Services 命名空间提供处理 XML Web Service 的类。

3. 控件

有两种类型的控件：HTML 控件和 Web 控件。System.Web.UI.HtmlControls 命名空间提供对 HTML 标记的直接映射，System.Web.UI. WebControls 命名空间能够使用模板来组织控件。

## 1.9 用 C#创建.NET 应用程序

C#可以用于创建控制台应用程序：仅使用文本、运行在 DOS 窗口中的应用程序。在进行单元测试类库、创建 Unix/Linux daemon 进程时，就要使用控制台应用程序。但是，我们常常使用 C#创建许多与.NET 相关的技术的应用程序，下面简要论述可以用 C#创建的不同类型的应用程序。

### 1.9.1 创建 ASP.NET 应用程序

ASP 是用于创建带有动态内容的 Web 页面的一种 Microsoft 技术。ASP 页面基本是一个嵌有服务器端 VBScript 或 JavaScript 代码块的 HTML 文件。当客户浏览器请求一个 ASP 页面时，Web 服务器就会发送页面的 HTML 部分并处理服务器端脚本。这些脚本通常会查询数据库的数据，在 HTML 中标记数据。ASP 是客户建立基于浏览器的应用程序的一种便利方式。

但 ASP 也有缺点。第一，ASP 页面有时显示得比较慢，因为服务器端代码是解释性的，而不是编译的。第二，ASP 文件很难维护，因为它不是结构化的，服务器端的 ASP 代码和一般的 HTML 会混合在一起。第三，ASP 有时开发起来会比较困难，因为它不支持错误处理和语法检查。特别是如果使用 VBScript 并希望在页面中进行错误处理，就必须使用 On Error Resume Next 语句，通过 Err.Number 检查调用每个组件，以确保该调用正常进行。

ASP.NET 是 ASP 的修订版本，它解决了 ASP 的许多问题。但 ASP.NET 页面并没有替代 ASP，而是可以与原来的 ASP 应用程序在同一个服务器上同时并存。当然，也可以用 C#编写 ASP.NET。

有专门的书详细讨论了 ASP.NET，这里仅解释它的一些重要特性。

1. ASP.NET 的特性

首先，也是最重要的，ASP.NET 页面是结构化的。这就是说，每个页面都是一个继承了.NET 类 System.Web.UI.Page 的类，可以重写在 Page 对象的生存期中调用的一系列方法（可以把这些事件看成是页面所特有的，对应于原 ASP 的 global.asa 文件中的 OnApplication_Start 和 OnSession_Start 事件）。因为可以把一个页面的功能放在有明确含义的事件处理程序中，所以 ASP.NET 比较容易理解。

ASP.NET 页面的另一个优点是可以在 Visual Studio 可视化环境中创建它们，在该环境下可以创建 ASP.NET 页面使用的业务逻辑和数据访问组件。Visual Studio 解决方案包含了与应用程序相关的所有文件。而且，也可以在编辑器中调试传统的 ASP 页面。

最清楚的是，ASP.NET 的后台编码功能允许进一步采用结构化的方式。ASP.NET 允许把页面的服务器端功能单独放在一个类中，把该类编译为 DLL，并把该 DLL 放在 HTML 部分下面的一个目录中。放在页面顶部的后台编码指令将把该文件与其 DLL 关联起来。当浏览器请求该页面时，Web 服务器就会在页面的后台 DLL 中引发类中的事件。

最后 ASP.NET 在性能的提高上非常明显。传统的 ASP 页面是和每个页面请求一起进行解释，而 Web 服务器是在编译后高速缓存 ASP.NET 页面。这表示以后对 ASP.NET 页面的请求就比 ASP 页面的执行速度快得多。

ASP.NET 还易于编写通过浏览器显示窗体的页面。传统的方式是基于窗体的应用程序提

供一个功能丰富的用户界面，但较难维护，因为它们运行在非常多的不同机器上。因此，当用户界面是必不可少的，并且可以为用户提供扩展支持时，人们就会依赖基于窗体的应用程序。

2. Web 窗体

为了简化 Web 页面的结构，Visual Studio 提供了 Web 窗体。它们允许把控件从工具箱拖放到窗体上，再考虑窗体的代码，为控件编写事件处理程序。在使用 C#创建 Web 窗体时，就是创建一个继承于 Page 基类的 C#类，以及把这个类看做是后台编码的 ASP.NET 页面。

过去，Web 开发的困难使一些开发小组不愿意使用 Web。为了成功地进行 Web 开发，必须了解非常多的不同技术，例如 VBScript、ASP、DHTML、JavaScript 等。把窗体概念应用于 Web 页面，Web 窗体就可以使 Web 开发容易许多。

用于添加到 Web 窗体上的控件与 ActiveX 控件并不是同一种控件，它们是 ASP.NET 命名空间中的 XML 标记，当请求一个页面时，Web 浏览器会动态地把它们转换为 HTML 和客户端脚本。Web 服务器能以不同的方式显示相同的服务器端控件，产生一个对应于请求者特定 Web 浏览器的转换。这意味着现在很容易为 Web 页面编写相当复杂的用户界面，而不必担心如何确保页面运行在可用的任何浏览器上，因为 Web 窗体会完成这些任务。

3. XML Web 服务

目前，HTML 页面解决了 World Wide Web 上的大部分通信问题。有了 XML，计算机就可以用一种独立于设备的格式在 Web 上彼此通信。将来，计算机可以使用 Web 和 XML 交流信息，而不是专用的线路和专用的格式。XML Web 服务是为面向 Web 的服务而设计的，即远程计算机彼此提供可以分析和重新格式化的动态信息，最后显示给用户。XML Web 服务是计算机给 Web 上的其他计算机以 XML 格式显示信息的一种便利方式。

在技术上，.NET 上的 XML Web 服务是给请求的客户返回 XML 而不是 HTML 的 ASP.NET 页面。这种页面有后台编码的 DLL，它包含了派生自 WebService 类的类。Visual Studio IDE 提供的引擎简化了 Web 服务的开发。

公司选择使用 XML Web 服务主要有两个原因：一是因为它们依赖于 HTTP，而 XML Web 服务可以把现有的网络（HTTP）用作传输信息的媒介；二是因为 XML Web 服务使用 XML，该数据格式是自我描述的、非专用的、独立于平台的。

### 1.9.2 创建 Windows 窗体

C#和.NET 非常适合于 Web 开发，它们还为所谓的“胖客户端”应用程序提供了极好的支持，这种“胖客户端”应用程序必须安装在处理大多数操作的最终用户的机器上，这种支持来源于 Windows 窗体。

Windows 窗体是 Visual Basic 6 窗体的.NET 版本，要设计一个图形化的窗口界面，只需把控件从工具箱拖放到 Windows 窗体上即可。要确定窗口的行为，应为该窗体的控件编写事件处理例程。Windows Form 项目编译为.EXE，该 EXE 必须与.NET 运行库一起安装在最终用户的计算机上。

### 1.9.3 Windows 控件

Web 窗体和 Windows 窗体的开发方式一样，但应为它们添加不同类型的控件。Web 窗体使用 Web 服务器控件，Windows 窗体使用 Windows 控件。

Windows 控件比较类似于 ActiveX 控件。在执行 Windows 控件后，它会编译为必须安装到客户机器上的 DLL。

实际上，.NET SDK 提供了一个实用程序，为 ActiveX 控件创建包装器，以便把它们放在 Windows 窗体上。与 Web 控件一样，Windows 控件的创建需要派生于特定的类 System.Windows.Forms.Control。

## 1.10 本章小结

本章介绍了许多基础知识，主要介绍什么是.NET Framework、.NET Framework 的构成及其组件的概念、.NET 框架中最重要的元素公共语言运行库、公共语言运行库的组成及功能、托管代码和非托管代码的运行原理、.NET Framework 的类库和命名空间之间的关系；讨论了所有面向.NET 的语言如何编译为中间语言，之后由公共语言运行库进行编译和执行的过程。

## 习 题

1．列出.NET Framework 的组件。
2．公共语言运行库的用途是什么？
3．什么是托管环境？
4．什么叫托管代码？

# 第 2 章　托管执行环境

.NET 的核心是公共语言运行库（Common Language Runtime，CLR），顾名思义它是一个可被不同的编程语言所使用的运行库。公共语言运行库环境也称为托管环境，本章介绍托管执行的概念、托管代码的编译和执行原理，以及如何使用.NET Framework 公共语言运行库环境快速生成应用程序。

**本章主要内容：**

- 托管执行环境中代码是怎样被编译和执行的。
- .NET 技术的相关重要概念。

## 2.1　概述

.NET Framework 中托管代码编译和执行的流程如图 2-1 所示。C#和 VB.NET 的代码首先被编译为微软中间语言（Microsoft Intermediate Language，MSIL）并存储在本地。当需要运行这些托管代码时，CLR 又对 MSIL 进行第二次编译（JIT 编译），将 MSIL 代码转换成本机代码（本机代码是针对当前 CPU 的可执行二进制代码）。

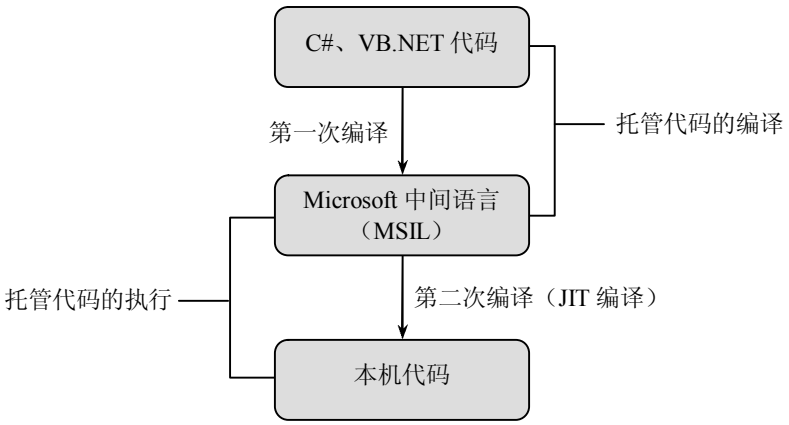


图 2-1　托管代码编译和执行的流程

对于.NET Framework 应用程序，安装在用户计算机上的程序并不是针对特定的 CPU 二进制代码，而是此应用程序的 MSIL，一种类似于以前隐藏在编译器内部的中间语言的代码。为什么要进行这种变化，将代码作为 MSIL 进行分发有什么好处？

很明显，这么做可以实现可移植性。.NET Framework 的核心并不需要在特定的操作系统和 CPU 上运行，也就是说它可以在非 Windows 操作系统和非 Intel 兼容处理器的系统上运行。

微软用以实现跨平台和跨语言的思想如图 2-2 所示。

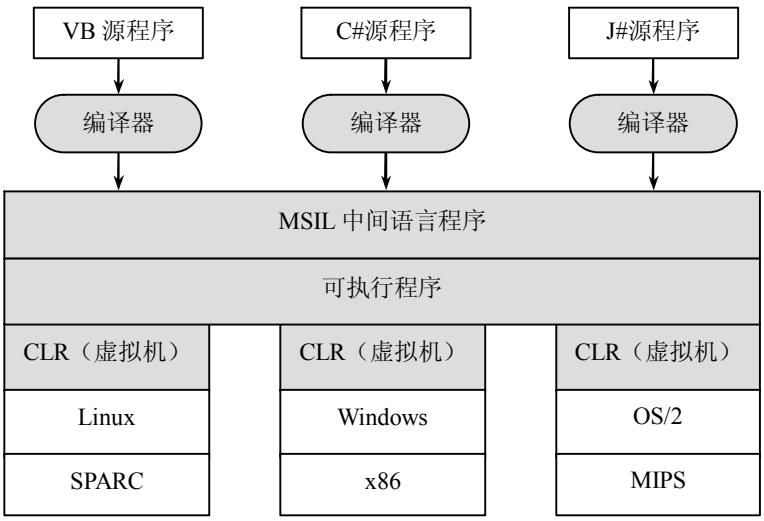


图 2-2　实现跨平台和跨语言的思想

如何跨语言应用？别无他法，创建一个统一语言是最直接的办法。那么原来就存在的各种语言怎么办？也别无他法，只能为它们配置专门的编译器，从而利用这些专门的编译器将各种不同的语言源程序编译成统一语言的中间程序。

为了实现上述目标，微软采取了两项措施：一是发展了一种叫做 MSIL 的中间语言，它相当于是一种为虚拟机配套的汇编语言，该语言也叫做 IL（Intermediate Language）；二是提出了一个编译器的制作规范，这个规范叫做 CLS（公共语言规范，Common Language Specification）。

于是，那些各种不同的编程语言，只要配有按照 CLS 规范来制作的编译器，那么这种语言程序就能被编译成 IL 语言程序，也就能跨语言应用。

## 2.2　编译和运行.NET Framework 应用程序

每种.NET Framework 支持的语言编译器都可以生成自描述的、托管的中间语言（Microsoft Intermediate Language，MSIL）代码。所有托管代码都通过使用公共语言运行库运行，公共语言运行库提供了跨语言集成、自动内存管理、跨语言异常处理、增强的安全性的编程模型。

### 2.2.1　编译器选项

下面编译和运行最简单的 C#程序，这是一个简单的控制台应用程序，它由把某条消息写到屏幕上的一个类组成。

在文本编辑器里输入下面的代码，并保存为后缀名为.cs 的文件（如 HelloDemoCS.cs）。

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

```
                Console.WriteLine ("hello world using c#!");
                Console.WriteLine("hello world using c#!");
                Console.WriteLine("hello world using c#!");
                Console.WriteLine("hello world using c#!");
                Console.WriteLine("hello world using c#!");
                Console.WriteLine("hello world using c#!");
        }
}
```

接下来编译这个程序。.NET Framework 包含一个 C#的命令编译器，此编译器名为 csc.exe。在 C# 中，通过使用 /? 开关选项可以获得完整的命令行选项列表，如下：

```
csc /?
```

命令行选项包括：/t 开关，指定编译目标；/r 开关，指定引用程序集；/out 开关，指定输出文件的名称。

（1）直接从命令行窗口中编译。

要编译一个 HelloDemoCS.cs 应用程序的源代码，输入如下：

```
csc   HelloDemoCS.cs
```

这个语法调用 C#编译器。在这个例子中，只需要指定要编译的文件名称。编译器将产生出可执行文件 HelloDemoCS.exe。

（2）用/target 或/t 指定编译目标。

可使用/target 或/t 开关选项指定目标文件类型，例如：

```
csc   /t:exe   HelloDemoCS.cs
```

（3）用/reference 或/r 来引用程序集。

当引用其他程序集时，应该使用/reference 或/r 编译开关选项，例如：

```
csc   /t:exe /r:assemb1.dll   HelloDemoVB.vb
```

托管代码是面向公共语言运行时的代码。创建托管代码需要如下步骤：

（1）选择编译器。为了获得公共语言运行库提供的优点，必须使用一个或多个针对运行库的语言编译器。

（2）将代码编译为 MSIL。编译器将源代码编译为 MSIL 并生成所需要的元数据。在执行时，实时（JIT）编译器将 MSIL 编译为本机代码。在此编译过程中，代码必须通过验证过程，该过程检查 MSIL 和元数据，以查看是否可以将代码确定为类型安全。托管代码在公共语言运行库提供的虚拟环境中执行，从而可以获得公共语言运行库提供的各种服务。

### 2.2.2 托管执行的过程

在.NET Framework 中，公共语言运行库为托管环境提供了基础结构，下面介绍在托管执行环境中编译和执行代码的一些基本概念，托管执行的过程如图 2-3 所示。

1. 将源代码编译为托管模块

可以用自己喜欢的语言来编写代码，前提是使用的该语言编译器能够编译面向 CLR 的代码。微软已经创建了几种面向 CLR 的语言编译器：C#、Visual Basic、JScript、J#（一个 Java 语言编译器），以及一个中间语言汇编器。除此之外，其他一些公司也在创建面向 CLR 的编译器，如 COBOL、Fortran 等。在这里可以将编译器看做是一个语法检查器和“正确代码”的分析器。它们对源代码进行检查，确保我们编写的所有东西都有意义，最后输出描述我们意图的

指令序列。但是，不管用哪种编译器，最后生成的结果都是一个托管模块。托管模块是一个需要 CLR 才能执行的标准 Windows 可移植可执行文件（Portable Executable，简称 PE 文件）。托管模块主要包括：编译器将源代码编译为 MSIL，这是一组可以有效地转换为本机代码且独立于 CPU 的指令，可以将其看成是一种“面向对象”的汇编语言，它提供了许多功能强大的 IL 指令，比如 call 指令用于调用一个方法，newobj 指令用于创建一个对象。在可以执行代码前，必须将 MSIL 转换为 CPU 特定的代码，这通常是由 JIT 编译器完成的，这部分内容我们稍后介绍。由于公共语言运行库为它支持的每种计算机结构都提供了一种或多种 JIT 编译器，因此可以在任何受支持的 CPU 上对同一组 MSIL 进行 JIT 编译和执行。当编译器产生 MSIL 时，它也产生元数据，元数据描述代码中的类型、每种类型的成员的签名、代码引用的成员和运行库在执行时使用的其他数据，公共语言运行库在运行时将会用到元数据。例如，元数据向外界表明以下信息：我这个模块中有多少个类、每个类中有多少个成员、它们引用了哪些外部模块中的类型等。

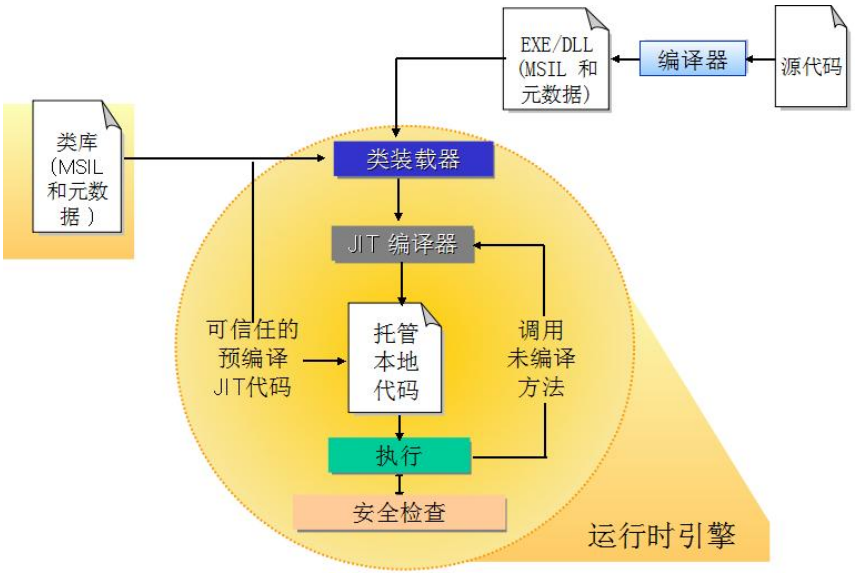


图 2-3　托管执行的过程

2. 执行代码

当用户执行托管程序时，操作系统加载器加载公共语言运行库，由 CLR 负责执行托管模块的 MSIL 代码。代码是如何执行的呢？

代码运行时，CLR 的类加载器从外部存储器中将模块加载到内存中并向元数据咨询该类的信息。代码的执行过程便是调用方法的过程，加载后，CLR 对方法的 MSIL 执行广泛的分析，当一个方法被第一次调用时，CLR 将检测出该方法引用了哪些类型，举例来说，我们在编程时调用了类库中 System.Console 类的 WriteLine 方法来向控制台输出一些信息，那么当程序中 WriteLine 方法被第一次调用时，CLR 会检测出这个方法引用了类库中的 System.Console 类，然后 CLR 会尝试加载 System.Console 类所在的类库程序集文件，当类库程序集文件被加载时，CLR 将自动检查是否有该程序集的预编译版本存在，如果存在，CLR 将加载预编译的代码，不再需要额外的运行时 JIT 编译，若不存在，则被调用方法的 MSIL 代码需要再经过 JIT

编译器动态地编译为本地 CPU 指令代码执行。在进行即时编译的过程中，CLR 同时检查这些 IL 指令是否违反了一些安全规则，这个过程称为验证过程，下面是验证过程可以检验的一些情形：

（1）不能从未初始化的内存中读取数据。

（2）每个方法调用都必须传入正确的参数个数，并且各个参数的类型要正确匹配。

（3）每个方法的返回值都必须被正确地使用等。

如果 IL 代码被确定是“不安全”的，那么 CLR 会停止编译并中断程序的执行。上面即时编译和代码验证的过程仅仅在第一次调用某个方法时发生。CLR 将编译好的本地代码缓存起来，第二次调用时就直接调用缓存中的本地代码，从而避免了再次编译带来的性能损失。这种进行 JIT 编译然后执行代码的过程一直重复到程序中所有代码执行完成时为止。

在执行过程中，托管代码接受自动内存管理、安全性、跨语言调试支持、增强的部署和版本控制支持等服务。

以上是对托管执行过程的一个介绍，接下来对托管执行过程中涉及到的重要概念进行较详细的说明。

### 2.2.3 元数据

元数据是一组数据表，在编译程序时将源代码转换成中间代码时自动生成，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。在 CLR 定位与装载类型时，系统通过读取并解析元数据来获取应用程序中的类型信息。JIT 编译器获得加载的信息后，将中间语言代码译成本地代码，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类型，从而保证了类型安全。元数据在解决方法调用、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。当执行代码时，运行库将元数据加载到内存中，并引用它来发现有关代码的类、成员、继承等信息。

元数据包含以下信息：

- 类型的名称。
- 类型的可见性，可为 public 或 assembly。
- 基类。
- 所实现的接口。
- 所实现的方法。
- 所暴露的属性。
- 所提供的事件。

此外，还可以包括更多的细节信息。例如每一个方法的描述，包括方法的参数及其类型、方法的返回值类型等。Visual Studio 使用元数据提供智能感知（IntelliSense）功能，可向开发人员显示对于所输入的类名有哪些相应的方法可用。

.NET 提供了提取元数据的基类，称为反射 API（System.Reflection），包含在 System.Reflection 和 System.Reflection.Emit 这两个命名空间中，利用该类可以获得.NET 程序集中的元数据。

元数据有很多用途，但是下面的用途是最为重要的：

- 查找和加载类：元数据省去了源代码编译时对头文件和库文件的需求。因为元数据和MSIL 包含在同一个 PE 文件中，也就是说该文件已经包含了所有被引用的类型和成员的信息，所以在编译该文件时编译器可以直接从托管模块中读取元数据来获得这些信息，从而省去了源代码编译时对头文件和库文件的需求。
- 强制安全性：元数据可能包含（也可能不包含）运行代码所需要的权限。安全系统使用权限来阻止代码访问其没有获得访问权限的资源。

元数据的其他用途包括：

- 解析方法调用。
- 设定运行时上下文边界。
- 提供反射功能。

元数据提供了互操作性的支持：

（1）因为为确定的类型提供了通用的格式，元数据允许不同的组件、工具和运行时支持互操作。

（2）利用元数据，可以在运行时获得程序集的方法、属性、类型和事件。

（3）.NET 提供了程序集登记、类型输出、类型引入和 XML 模式 4 个工具软件支持互操作。

元数据确保了语言的集成，是.NET 的必要元素，没有元数据，.NET 不能支持内存管理、安全管理、内存部署、类型检查等功能，可以说，没有元数据就不可能有.NET。

### 2.2.4 Microsoft 中间语言（MSIL）

通过前面的学习，我们理解了 Microsoft 中间语言显然在.NET Framework 中有非常重要的作用。C#开发人员应该明白，C#代码在执行前要编译为中间语言（实际上，C#编译器仅编译为托管代码），这是有意义的，客观地说，在.NET Framework 环境中进行工作的开发人员并不需要完全理解 MSIL，但是至少要稍微了解一下 MSIL 的内容。

现在简单讨论一下 IL 的主要特征，因为面向.NET 的所有语言在逻辑上都需要支持 IL 的主要特征。

中间语言的主要特征如下：

- 面向对象和使用接口。
- 值类型和引用类型之间的巨大差别。
- 强数据类型。
- 使用异常来处理错误。
- 使用特性（Attribute）。

实际上，MSIL 是 CLR 的汇编语言。关于这些 MSIL 指令集，值得注意的一点是，它与 CLR 的 CTS 抽象有着十分紧密的对应关系。它能直接支持对象、值类型，甚至是装箱和拆箱操作；而且，某些操作（如创建新实例）与高级语言中的一些极为常见的操作符相似，而这些操作方法在典型机器指令中很少见。

对于希望直接处理那些低级机器码的开发人员，.NET Framework 提供了称为 Ilasm 的 MSIL 汇编器。不过，开发人员一般不会去使用此工具，既然可以使用更简单、更强大的语言获得相同的结果，就不必用 MSIL 编写了。

### 2.2.5 程序集

程序集（Assembly）是包含编译好的、面向.NET Framework 的代码的逻辑单元，类似于逻辑.dll。程序集是一个或多个托管模块，以及一些资源文件的逻辑组合。程序集是组件复用，以及实施安全策略和版本控制的最小单位。每个程序集由组成功能单元的所有物理文件组成，这些物理文件包括所有托管模块以及资源或数据文件。

从概念上说，程序集提供了将一组文件作为单个实体的方法。

程序集是一个逻辑上的构件，并不存在单一文件将所有必要文件包裹成一个程序集。实际上，只看磁盘目录列表不可能说出哪些文件属于同一个程序集。要想搞清楚一个特定的程序集到底由哪些文件构成，必须查看该程序集的清单。如前所述，模块的特性包含了模块内的类型信息，与此相类似，程序集清单包含了“程序集内的所有模块及其他文件”的有关信息，换句话说清单就是程序集的特性，清单包含于程序集的某个文件里，并且包含了程序集的信息以及“组成程序集的文件”的信息。Visual Studio 这样的工具不但会为每个编译模块生成元数据，还会为每个程序集生成相应清单。

程序集的一个重要特性是它们包含的元数据描述了对应代码中定义的类型和方法。程序集也包含描述程序集本身的元数据，这种程序集元数据包含在一个称为程序集清单的区域中，可以检查程序集的版本及其完整性。

由前面的分析可见，通常程序集可能由以下 4 个元素组成：

- 程序集清单，包含程序集元数据。
- 类型元数据。
- 实现这些类型的 Microsoft 中间语言（MSIL）代码。
- 资源集。

对上面 4 个元素而言，只有程序集清单是必需的，但也需要资源或类型来向程序集提供任何有意义的功能（程序集=托管模块+程序集清单），将在第 4 章中详细介绍程序集。

### 2.2.6 应用程序域

在 Windows 操作系统中，每个进程都有自己的虚拟地址空间。以前使用进程边界来隔离在同一台计算机上运行的应用程序。每个应用程序被加载到单独的进程中，以进程间地址空间的分离来达到一个应用程序与运行在同一台计算机上的其他应用程序相隔离。这种隔离是有必要的，因为我们不能信任应用程序的代码，一个应用程序完全有可能读写一个无效的内存地址。这种将每个 Windows 进程放在一个独立的地址空间的方法提高了应用程序的健壮性，因为这样一个进程就不会干扰另一个进程的运行。

但是，因为 Windows 进程需要许多操作系统资源，上面这种以进程作为边界分隔应用程序的方法在性能上会大打折扣。例如，我们同时在计算机上运行多个应用程序，就会启动多个进程，太多的进程会损伤系统性能，并限制系统中可用的资源。那么，如何在降低这种性能损失的基础上来实现应用程序间的隔离呢？

在前面讲到托管执行的过程时，指出托管代码必须先通过 CLR 对其进行的一个验证过程，然后才能运行。通过验证的代码被认为是类型安全的，我们可以确保这些类型安全的代码不会访问无效的内存地址，因此也就不会干扰另一个应用程序的代码。这意味着我们可以在一个单

独的 Windows 虚拟地址空间内运行多个托管应用程序。在 CLR 中，一个托管应用程序称为一个应用程序域（AppDomain），CLR 提供了在一个单独的操作系统进程中执行多个托管应用程序的能力。在一个操作系统进程中可以容纳多个应用程序域，这些应用程序域之间是相互独立的，一个应用程序域中的代码不能直接访问另一个应用程序域中的对象，相应地，一个应用程序域中出现的异常也不会影响到其他应用程序域，应用程序域的另一个特性是能够在不停止整个进程的情况下卸载某个应用程序域。这样就可以减少进程的数量，从而提高性能，降低资源需求，而应用程序仍然可以保持良好的健壮性。

## 2.3 本章小结

本章介绍了使用 C#编写一个简单的控制台程序的过程，介绍了托管执行环境中代码是怎样被编译和执行的，重点介绍了如何创建一个.NET 应用程序及涉及到的重要概念；本章.NET 的重要概念非常多，也比较难理解，需要仔细思考。

难理解的概念包括：命名空间、应用程序域、元数据及作用、公共语言运行库环境（CLR）和程序集。

## 习 题

1．.NET 产生的代码叫什么？

2．哪一个.NET 组件将 MSIL 编译成特定于 CPU 的本机代码？

3．下面哪一个命名空间声明的类与其他 3 个不同？

① 
```
namespace MySchool.FirstNamespace
{
    class ClassA {}
    class ClassB {}
}
```
② 
```
namespace MySchool.FirstNamespace
{
    class ClassA {}
}
namespace MySchool.FirstNamespace
{
    class ClassB {}
}
```
③ 
```
namespace MySchool
{
    namespace FirstNamespace
    {
        class ClassA {}
    }
    class ClassB {}
}
```

④
```
namespace MySchool
{
    namespace FirstNamespace
    {
        class ClassA {}
        class ClassB {}
    }
}
```

4．哪个组件把 MSIL 编译成本地代码？

5．.NET 编译器生成的代码叫什么？

6．什么是元数据？

7．.NET Framework 使用中间代码的好处是什么？

8．元数据包含哪些内容？