第1章

另辟蹊径：解读.NET

本章主要说明.NET 平台的组成及其出现的意义，讲述了程序集同时具备可读性(对于开发者)和可执行性(对于用户)、CLR 的作用和它在.NET 平台中所处的重要位置，以及.NET 平台是怎样改变传统 Windows 的开发模式的。

1.1 前.NET 时代

在传统应用程序开发中，有各种各样的计算机开发语言，每种语言的使用方法并不完全一样，使用不同语言开发的人员之间很难形成交互点。于是不同的开发人员使用各种高级计算机语言编写出来的源代码，经过编译器编译之后，直接生成与平台(操作系统)关联的各种机器指令，如图 1-1 所示。

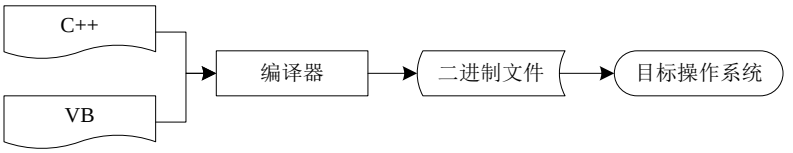


图 1-1 传统开发流程

由于不同的操作系统使用不同的指令集，因此使用编译器编译出来的二进制指令(文件)只能运行在某一特定的操作系统之中，如果想要让它运行在另外一种操作系统中，那么开发人员必须重新编译源代码，生成与另外一种操作系统相匹配的二进制指令。

出现这种情况的一个最主要的原因是在编译环节，因为编译器生成的二进制指令只能针对特定的操作系统。我们称这种编译方式为早期关联。也就是说，我们开发的程序在编译时就已经决定了它未来的运行环境。

随着计算机行业不停地发展，出现了各种各样的操作系统，以前一些不流行的操作系统也慢慢地开始流行了起来。这对开发人员来讲无疑是不利的，因为我们必须让我们现在开发出来的程序去适应各种各样的操作系统，而前面我们讲到，要想让我们的程序在不同的系统中运行，我们必须重新编译我们的源代码。

注：重新编译我们的源代码只是很笼统的一种说法，更多的时候是要修改源代码，原因将在后面的章节中提到。

为此，20 世纪 90 年代中期，Sun 公司推出了一个名叫 Java 的开发平台，主要目的是想让我们编写出来的程序能在任何一个操作系统之中运行，而不需要开发人员重新编译它的源代码。这个做法无疑是空前绝后的，开发人员不再需要为了让他们的程序去适应各种各样的操作系统而多做任何工作，“编译一次，到处运行”即是当时 Java 平台推广中最有名的一句广告语。

那么，到底 Sun 公司是怎样做到的“一次编译，到处运行”的呢？前面笔者提到过，之所以要为适应不同的操作系统而重新编译源代码，是因为编译器在编译阶段就把我们的源代码转换成了特定操作系统的二进制指令，在编译阶段，我们的程序就跟某一个操作系

统已经关联上了，因此，编译出来的二进制指令只能被该操作系统识别。Sun 公司要做的，就是在这个“编译阶段”让编译器编译出来的中间件不跟任何操作系统相关联，当这个中间件真正运行在某一个操作系统之中时，再由专门的程序将它翻译成与这个操作系统相匹配的指令。这样给人的感觉便是，我们只需要编译一次我们的程序，之后它就可以运行在任何一个系统之中了。

Java 平台程序从开发到运行的流程如图 1-2 所示。

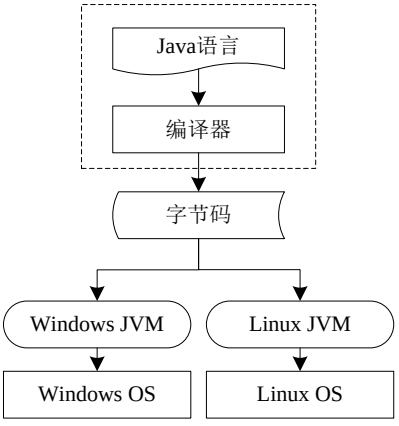


图 1-2　Java 平台程序开发运行流程图

图中虚线框内为开发人员负责的部分。理论上，不管最终程序运行在哪里，开发人员只要做一次同样的事情，剩下的环节对于开发人员来讲，都是不可见的。图中“字节码”就是前面提到过的中间件，它既与任何操作系统无关联，也不等同于传统开发过程中的“二进制文件”。图中 JVM 就是前面提到过的“专门的程序”，它能够将中间件翻译成与操作系统相匹配的指令。

Java 平台的出现在当时改变了整个软件行业的开发模式，尤其对开发人员来讲，是个极大的福利，开发人员从面向操作系统编程转变为面向平台编程。与此同时，微软紧跟其后，推出了与 Java 平台非常相似的一个开发平台，取名叫 Dot Net，简写为.NET。.NET 平台是微软的一个全新开发平台，虽然结构与 Java 平台相似，但是功能和出现的意义与 Java 千差万别，本章接下来几节会讲到这些。

注：上文提到的操作系统和平台的区别，为了避免混淆，本书中把操作系统称之为“操作系统”，而把 Java 和.NET 等称之为“平台”。另外，请注意 Java 语言和 Java 平台的区别。

1.2 .NET 的组成

何为平台？平台就是我们进行某项工作所需要的环境和条件。.NET 平台的出现彻底改变了传统 Windows 的开发模式，它重新定义了 Windows 的开发流程，解决了之前程序开发中遇到的问题，比如 COM 时代的 DLL 地狱(DLL HELL)、C++编程中的内存泄露、VB 编程中使用 COM 的困难等。它还集成了各种各样的语言，无论哪种语言都可以开发.NET 程序，并且每种语言编写出来的组件(程序集)之间可以毫无障碍地通信，甚至还可以从 VB.NET 编写的程序集 DLL 中派生出一个新的 C#类型，而并不需要知道该类型基类的 VB 源代码，这在以前是完全不可能做到的。除此之外，.NET 还真正实现了二进制兼容性，比如修改一个 DLL 中的公共类型，只要使用了该 DLL 的客户端没有使用该公共类型，那么客户端就不需要重新编译，这在之前也是完全不敢想象的。

.NET 平台之所以具备以上这些功能，完全得益于它的组成结构。.NET 平台主要由丰富的框架库、各种各样的开发语言、各种语言同时遵守的规范、公共语言运行时等组成。正是这些组成部分相互协调工作，才使得.NET 平台具有如此强大功能，如图 1-3 所示。

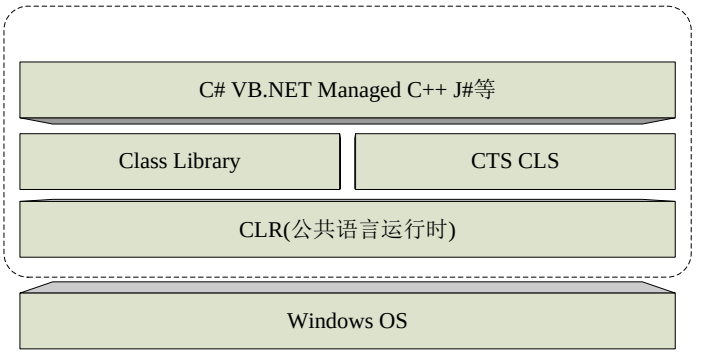


图 1-3 .NET 平台组成

图中列出的语言虽然仅仅为微软官方支持的几种语言，但理论上，任何一种语言只要遵守 CTS(公共类型系统)、CLS(公共语言规范)，并且有相应的编译器，那么它就可以成为.NET 平台支持的语言之一。

1.2.1 .NET 中的语言

.NET 平台官方给出的语言有 C#、VB.NET、Managed C++、J#和 F#，其中 J#主要是为了移植一些 J++项目，除了 C#和 VB.NET 之外，其他几种语言虽然平时也用得非常少，但是理论上，任何计算机语言只要能遵守公共类型系统和公共语言规范，并且有相应的.NET 编译器，那么这些语言就都可以成为.NET 平台语言大家庭中的一员。

.NET 中虽然语言种类繁多，但是各种语言之间却可以毫无障碍地进行交互，也就是说使用 C#编写的程序集，跟 VB.NET 编写的程序集是一样的，它们之间可以相互调用，这正是.NET 平台中的“语言集成”特性。

注：J++为微软早年以 Java 语言为基础开发出来的一种语言，在 JAVA 语言中增加了.NET 中特有的委托、事件等特性，后来由于版权问题，停止了支持该语言。

此外，上文中提及的程序集，是一种类似前面介绍 Java 平台时说到的“中间件”，将在下一节中讲到。

1.2.2　.NET 中的框架库

何为库？库就是别人已经写好了的，有组织有结构的代码集合，程序员可以直接拿过来使用，大到传统流行一时的 MFC、ATL 等框架，小到自己公司编写的一些通用工具类代码。.NET 平台本身是一个包含内容非常丰富的类库，范围涉及数据结构、网络、图形处理、加解密、线程、I/O、数据库等各个方面。作为一个开发人员，对“库”的概念应该不会陌生。

注：库和框架本身没有明显的区分定义，如果想了解更多，请参见本书第 2 章内容。

1.2.3　公共类型系统

前面讲到在.NET 平台中可以使用各种各样的语言进行程序开发，也就是说.NET 并不区分我们编写的某个类到底是使用 C#还是 Managed C++。使用 C#定义的一个接口和使用 VB.NET 定义的一个相同接口，最后产生的效果是一样的，都可以被 F#使用。那么这些语言定义的类型可以被.NET 平台相同对待的前提是什么呢？当然是必须遵守同一个类型规范，如果 C#中有 Interface 类型，而 VB.NET 中却没有 Interface 类型，那么就谈不上相同对待。

微软为.NET 平台定义了一套所有语言都必须遵守的规范——公共类型系统(Common Type System，CTS)，.NET 平台语言大家庭中所有语言都必须遵守这个规范，比如我们常见的值类型(ValueType)、引用类型(ReferenceType)、接口(Interface)、属性(Property)以及委托(Delegate)等。

1.2.4　公共语言规范

.NET 平台允许我们使用不同的语言开发应用程序，各种语言之间可以进行无障碍的交

互。比如我们既可以在 C#中派生出一个使用 VB.NET 编写类型的子类，也可以在 VB.NET 中捕获 Managed C++中抛出的异常，也就是说继承、多态、异常处理等这些功能之间都可以交叉使用。然而，每一种语言本身的规则又是不相同的，比如在 Managed C++语言中对区分大小写很敏感，而在 VB.NET 语言中则对区分大小写不敏感。正因为不同的语言之间可能支持不同的特性，而这些不同的特性又会影响语言之间的交互，所以微软为.NET 平台定义了一套所有语言必须遵守的规范——公共语言规范(Common Language Specification，CLS)，在.NET 平台语言大家庭中，所有语言都必须遵守这个规范。

最后需要指出的是，不仅开发人员需要了解以上规范，各语言编译器开发商同样也需要知道这些规范。

1.2.5 公共语言运行时

公共语言运行时(Common Language Runtime，CLR)是.NET 平台的核心。在使用各种语言开发的程序经过编译之后生成的中间件(程序集)，并不是传统意义上的二进制指令，而是一个类似代码数据的集合。这个代码数据集合并不能直接在操作系统中运行，如果说它还需要像在 Java 平台中一样，通过某种专门的程序将其转换成与操作系统相匹配的二进制指令，那么，CLR 就是.NET 平台中的这个“专门的程序”。由此来讲，.NET 平台中的程序需要经过两次“翻译”之后才能运行，一次由开发人员使用编译器进行编译，将源代码编译成中间件，另一次则由 CLR 将中间件翻译成与操作系统相匹配的二进制指令。

在传统程序开发过程中源代码的编译有且仅有一次，且直接生成二进制指令，一步到位。如果我们把传统编译称为“完全编译”，那么.NET 平台中的第一次编译就可以称之为“非完全编译”，如图 1-4 所示。

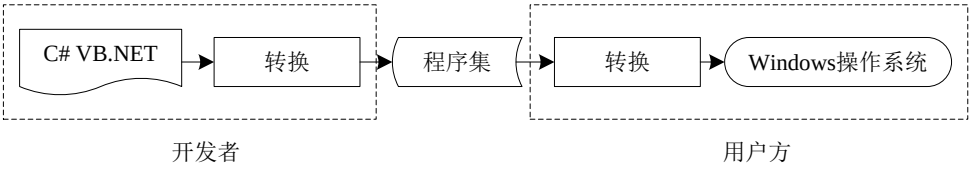


图 1-4 .NET 平台程序的两次转换

图 1-4 中第一次转换发生在开发者方，第二次转换发生在用户方。

注：上文说到的用户方是指程序的最终用户，CLR 一般安装在最终用户的操作系统中，它会自动运行，但 CLR 的工作过程对最终用户是不可见的。Windows Vista **以及之后的操作系统默认安装有不同版本的.NET CLR 程序。这便是为什么我们经常在** Windows XP **系统中运行一个.NET 程序时，系统会出现类似“没有相关运行环境”**

的错误提示。

在 CLR 中包含着一个名叫 JIT(Just-In-Time)的即时编译器，专门负责将中间件(程序集)转换成与操作系统相匹配的二进制指令，然后执行。CLR 的工作流程如图 1-5 所示。

JIT 编译器不会将所有的托管代码一次性编译成本地代码，而是当它需要执行某部分代码时才会将该部分代码(如果该代码从未编译过)编译成本代码。也就是说，有些托管代码可能从来都不会被二次编译。CLR 不仅有 JIT 这样的编译器，还具有统一内存管理、异常处理、安全检查、访问 COM 和 Win32 等功能。并且无论使用什么语言编写的托管代码，只要在 CLR 中运行，它都能为其提供这些服务。

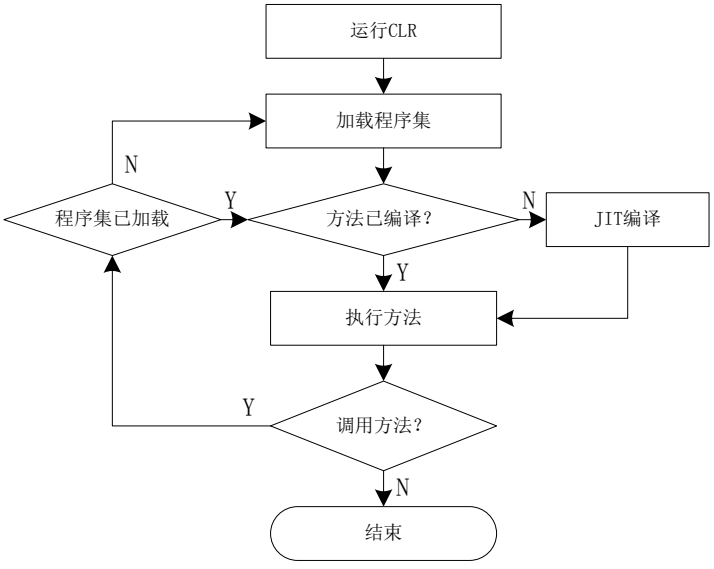


图 1-5　CLR 工作流程

注：与操作系统相关联的可以直接运行的代码称为本地代码(Native Code)或者非托管代码(Un Managed Code)，在传统开发过程中一次完全编译生成的二进制指令就是本地代码。在.NET 平台中经过非完全编译后生成的只能在 CLR 这样的环境中运行的代码称之为托管代码(Managed Code)，该解释在第 2 章中将有详细说明。

1.2.6　.NET 程序的运行流程

当开发人员在选择某种语言编写完程序后，该程序经过相应的编译器被编译成一种中间件(程序集)时，CLR 就会加载该中间件，使其运行，这便是.NET 程序的运行流程，如图 1-6 所示。

图 1-6 中各种语言编译出来的结果都是一样的，也就是说，CLR 只关心图中的程序集，不管这个程序集是通过什么语言开发的，这是.NET 平台支持多语言的前提。

从某种意义上讲，在.NET 平台中，开发人员从之前的“面向 Windows 操作系统编程”转变为了“面向 CLR 编程”。开发人员以往得到的是非托管代码，而现在得到的是托管代码。

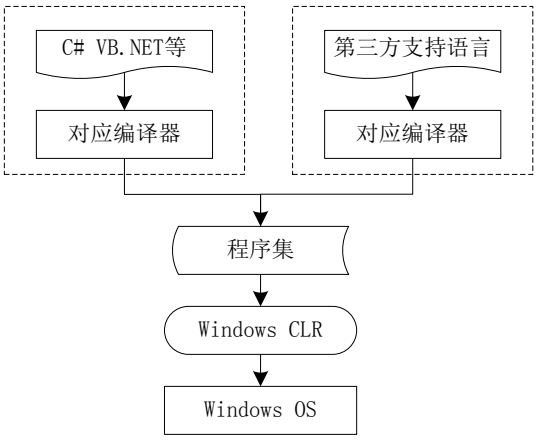


图 1-6　.NET 平台程序开发运行流程图

1.3　.NET 中的程序集

前面提到过，开发人员将源文件编译之后生成的中间件称为“程序集”。由于程序集文件名一般以.EXE 或者.DLL 结尾，很容易将它与传统开发过程中后缀名相同的 EXE 文件或者 DLL 文件混淆，但它们在本质上却是千差万别的。

1.3.1　程序集与 EXE 文件的区别

首先，它们出现的地方不同。程序集是面向公共语言运行库(CLR)的，是.NET 平台范畴内的东西，而 EXE 文件则是主要面向 Windows 操作系统的。

其次，它们的组成结构不一样。程序集中除了包含一种叫中间语言(Intermediate Language，IL)的代码之外还有很多其他东西，比如类型信息、版本信息、引用其他程序集信息、安全加密信息以及一些资源数据。也就是说，程序集是一种“自我描述”的文件，而 EXE 文件则主要包含二进制指令，是一个指令集合。

最后，它们的功能也不一样。程序集不仅可以在 CLR 中运行，它还可以在开发过程中发挥作用。开发人员可以直接从程序集中获取类型信息，比如以程序集中的某个类型为基类派生出新的类型，而后者不可能有这种功能。

总之，程序集是非完全编译的产物。它兼备了源代码和本地代码的特性，是一种介于源代码和本地代码之间的独立存在的一种数据结构，它同时具有可读性和可执行性。程序集与 EXE 文件的功能区别如图 1-7 所示。

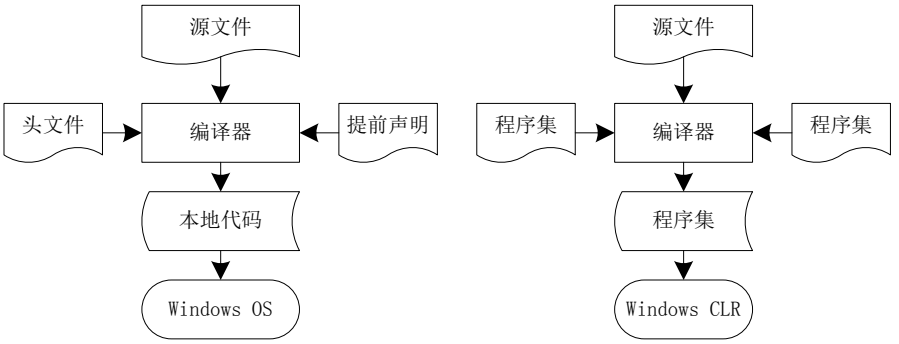


图 1-7　程序集与本地代码的功能区别

图 1-7 中左边是由编译器编译出来的本地代码，除了可以运行外，其余什么都做不了，而右边是由编译器编译出来的程序集，除了可以运行在 CLR 中外，还可以运用在开发阶段。在传统开发过程中，编译器不可能从一个二进制文件中读取里面的类型信息，只能通过类似 C++中的头文件等方式，来告诉编译器在代码中用到了哪些外部类型。

程序集相对于编译器来讲，是可读的，相对于 CLR 来讲，是可以运行的。

注：由于程序集本身保存有版本、安全加密等信息，因此，程序集的可读性不仅仅体现在编译阶段，还体现在部署和运行阶段，这些信息都可以对其起到关键作用，其具体内容，后续将会讲到，此不赘述。

1.3.2　程序集的组成

程序集的组成结构如图 1-8 所示。

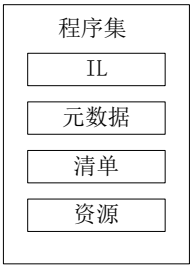


图 1-8　程序集组成

图 1-8 中程序集中的代码部分用 IL 表示，作为一种“中间语言”，它跟生成它的源语言(C#或者 VB.NET 等)是无关的，但用 C#和 VB.NET 定义同一个 class 时，编译之后生成的 IL 却是等效的。程序集中的元数据主要包含所在程序集中的命名空间、类型等信息，这些

信息通过某些手段可以被读出来。程序集中的清单主要记录程序集本身的一些信息，比如版本、安全加密以及引用其他程序集的一些信息。程序集中的资源指的是该程序集包含的一些可用资源，这些资源在程序集运行时可以使用，包括字符串、图片以及特殊文件等。

正是程序集的这种特殊组成结构，才使得程序集与传统 EXE 或 DLL 文件有着不一样的功能。

1.3.3　程序集的特点

程序集的特殊组成结构，是其具备特殊功能的前提。也正是程序集的存在，才使得.NET 平台开发模式比传统 Windows 开发模式更方便快捷。程序集有如下 4 个特征。

(1)　语言独立。
(2)　二进制兼容。
(3)　可重用性。
(4)　部署方便。

1.4　.NET 的跨平台

1.4.1　Write Once, Run Anywhere 的真实现状

Write Once, Run Anywhere 中的 Write 是相对于开发人员来讲的，而 Run 则是相对于终端用户来说的。这句话意思是说开发人员只需编写、调试、编译一次源程序，生成的可运行程序就可以在任何一个最终用户的任何一个操作系统之上运行。这句伟大而又霸气的话从 Java 平台刚一出来就流行了起来，同时它也成了曾经推广 Java 平台最为流行的一句广告词，影响着整个托管时代。

注：传统面向操作系统编程，编译出来的二进制指令可以直接运行在目标操作系统之上，我们称那个时代为“非托管时代”。现如今随着.NET 平台、Java 平台的流行，编译出来的中间件只能在一种环境中运行，表面上看是将程序交给这种环境托管运行，因此我们称这个时代为“托管时代”。

愿望总是美好的，但是真正要开发出百分百的 Write Once, Run Anywhere 这样的跨平台应用程序几乎是不可能的，原因其实前面讲到过，无论在 Java 平台还是.NET 平台中，开发人员将源程序编译之后生成的“中间件”，必须要经过第二次编译生成与目标操作系统相匹配的本地代码之后，才能运行。如果第二次编译阶段不能将这些“中间件”完全地或者说准确地转换成与目标操作系统相匹配的本地代码，其结果可想而知。

所以要让 Java 程序能够更好地跨平台，我们在开发阶段就应该选择性地使用一些框架库，尽量使用能适应所有操作系统的框架库，这也是为什么在 Java 平台中默认自带的框架库少之甚少的原因之一。为了满足开发需要，于是出现了越来越多的 Java 第三方库，但是这些第三方库大多数是针对特定操作系统的，因此，跨平台就显得越来越难以实现。

1.4.2 .NET 与 Java 平台出现的目的

很多人认为.NET 平台的作用就是为了让我们开发出跨平台的应用程序，而事实上，它跟 Java 平台有着不一样的目标。

.NET 平台的结构虽然跟 Java 平台极其类似，但它的出现只是为了优化传统 Windows 开发模式，使 Windows 开发流程更加方便快捷。而 Java 平台出现的主要目的则是为了解决程序跨平台运行的问题。如图 1-9 所示为两个平台的不同侧重点。

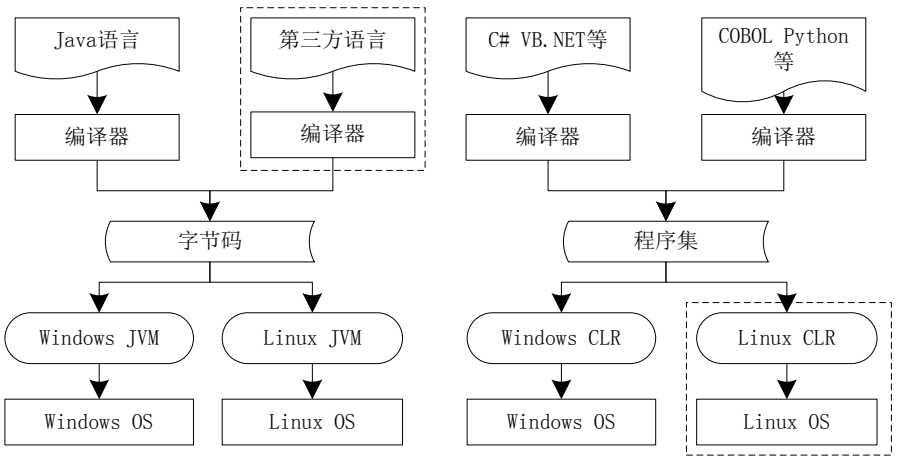


图 1-9 .NET 平台与 Java 平台的侧重点

图中虚线部分为平台之外的功能，理论上 Java 平台也支持多种语言，并且已有第三方语言在 Java 平台中小范围使用，只是没有得到推广。另外，理论上.NET 平台也可以支持跨平台，只要有对应操作系统的 CLR 存在，.NET 程序也可以运行在除了 Windows 之外的其他操作系统之中，如比较流行的 Mono，它能让我们的.NET 程序运行在 Linux 上。

那么，为什么两个平台有着相似的结构，却干着不一样的事情呢？这是因为两个平台所追求的目标不同。Sun 公司认为，在互联网世界中，他们要让一种语言在任何一个操作系统之中运行；微软则认为，在互联网世界中，他们要让所有的语言在同一个操作系统之中运行。很明显，微软所说的同一个操作系统就是指 Windows 操作系统。

注：有一种说法是，微软早年推广.NET 平台时也是打着跨平台的旗帜，只是后

来改变了路线，转而只支持 Windows，因此到目前为止，微软还没有发布过官方的非 Windows 的.NET CLR。不论哪种说法，.NET 平台现在的主要目的已不是支持跨平台了。

很多人有一个疑问，既然.NET 支持跨平台，也有一些第三方厂家发布了非 Windows 的.NET CLR，比如 Mono，那为什么还有很多.NET 程序不能从 Windows 移植到 Linux 呢？而且这个概率比 Java 平台大得多。其实在上一小节中笔者对此已经给出了答案，即程序是否能从一个操作系统移植到另外一个操作系统，关键要看我们在开发程序时使用了哪些框架库，因为有些框架库跟某个特定操作系统关联比较大(事实上，有很多库是针对某一个操作系统而开发的)，如果在源程序中使用了某种框架库，那么在另外的操作系统中是很难将其翻译出来的。比如.NET 平台中的 Windows Forms 框架，我们编写的 Winform 应用程序几乎不可能通过 Mono 移植到 Linux，原因很简单，微软开发的 Windows Forms 框架是针对 Windows 的，框架内部是通过调用 Windows 中类似 User32.dll、Gdi32.dll 这些 API 来实现的，仅此而已。

1.4.3　重新看待.NET

经过前面一小节的讨论，我们知道了微软发布.NET 平台的目的并不是为我们开发跨平台程序提供便利，因此我们应该清楚，如果我们的程序需要支持跨平台，那么最好不要选用.NET 开发平台。当然，如果我们开发的程序最终要运行在 Windows 之中的话，那么使用.NET 平台开发程序无疑又是最好的选择。

1.5　.NET 平台出现的意义

.NET 平台出现的意义主要体现在以下几个方面。

(1) .NET 平台将“面向对象”和“面向组件”融为一体。我们可以把程序集当作传统的 DLL 组件去使用，起到代码重用的效果，可以从中读取类型信息，从中派生出新的类型，在开发阶段就不需依赖于类似头文件或者提前声明这样的东西。程序集同时具备可读性和可执行性。

(2) .NET 平台集成了所有开发语言，可满足开发人员的不同需求。理论上，任何一种开发语言都可以成为.NET 平台中的一员，并且不同语言之间可以进行无障碍交互，我们可以从 C#编写的程序集中派生出 VB.NET 的类，也可以在 Managed C++中捕获由 C#编写的类型抛出的异常。

(3) .NET 平台解决了 COM 时代的 DLL HELL 问题。每个组件都是自我描述的，包含自己的版本信息，同一个组件的不同版本可以同时运行，依赖旧版组件的程序加载旧版组

件，依赖新版组件的程序可以加载新版组件，它们之间没有关联。

(4) .NET 平台真正实现了“二进制兼容性”。在传统编程中，一个组件修改了公共接口之后，使用这个组件的客户端必须重新编译，而在.NET 平台中，组件的任意一个公共接口修改之后，只要使用这个组件的客户端没有使用该公共接口，那么该客户端就不用重新编译，举例如下：

```
//Code 1-1
interface A
{
    void A_fun();
    void B_fun();
}
```

以上接口可以增加一个方法 void C_fun()或者将 A_fun()和 B_fun()换一下位置，使用该组件的客户端不需要重新编译，如果客户端中没有使用 B_fun()方法，我们还可以将 B_fun()删除，客户端仍然不需要重新编译。

(5) 部署程序集更方便。在.NET 平台中，一切私有程序集均可拷贝到与客户端同一目录之下，公共程序集统一放在 GAC(Global Assembly Cache，全局程序集缓存)之中，不像在 COM 中需要将组件信息写入注册表。并且在.NET 平台中同一程序集的不同版本可以同时运行，不需要考虑程序集升级之后因不兼容而造成依赖旧版程序集的程序无法正常运行的情况。

(6) .NET 平台统一了传统开发模式中各种各样的库。在传统开发模式中存在各种各样的框架库，常见的库有 MFC、ATL 以及 STL 等，但每种框架库的使用方法不一样，各种语言不能共享，而.NET 平台提供了一套丰富、功能强大的框架库，各种语言之间可以共享，使用方法类似，如果你掌握了 C#中 Console 类的用法，就等于你掌握了 VB.NET 中 Console 类的用法。

(7) .NET 平台为 CLR 提供了丰富的功能。CLR 不仅提供了内存管理功能，使得程序开发者不用再担心像传统 C++开发中那样程序中没有内存回收的代码而造成内存泄露；CLR 还提供了统一的 COM、Win32 访问功能，让开发者不管使用哪种语言，都可以轻松访问 Windows 操作系统中的非托管代码；CLR 还提供了调试功能，为编译器提供公开接口，让任何语言的编译器能够调试程序；CLR 还提供了安全检查、异常处理等功能。

注：①.NET 平台之所以具备以上特性，主要是因为它的第一次“非完全编译”。第一次非完全编译出来的中间件包含各种有用的信息，它不依赖于任何源语言，也不依赖于任何操作系统，只有在 CLR 中经过第二次编译，才会生成传统意义上的本地代码。②组件在不同场合有不同含义，这里的组件指“能够共享的二进制代码”。在.NET 平台中，可以把程序集当作组件，程序集作为可以共享的代码单元独立存在。③DLL

HELL 指的是，由于组件升级后不兼容的缘故，导致某一些依赖旧版组件的程序无法正常运行。比如由不同厂家发布的 A 程序和 B 程序同时依赖 A.DLL，由于某一次 A 程序升级了 A.DLL，新版的 A.DLL 并不兼容旧版，所以此时还是要依赖旧版的 A.DLL 才能运行的 B 程序，就会无法正常运行。

1.6 本章回顾

在本章开头，我们了解了在.NET 平台出现前，软件开发模式已经存在的一些缺陷激发了.NET 平台的出现；之后我们详细了解了.NET 平台的组成，以及它与 Java 平台的一些相同点和不同点，提到了.NET 跨平台的真实现状；最后本章总结了.NET 平台出现的意义，它不仅完善了现有 Windows 软件开发模式，还使得 Windows 软件开发模式变得更加方便快捷。

1.7 本章思考

1. 简述.NET 平台中 CTS、CLS 以及 CLR 的含义与作用。

A：CTS 指公共类型系统，是.NET 平台中各种语言必须遵守的类型规范；CLS 指公共语言规范，是.NET 平台中各种语言必须遵守的语言规范；CLR 指公共语言运行时，它是一个虚拟机，.NET 平台中所有的托管代码均需要在 CLR 中运行，可将其视为另外一个操作系统。

2. 简述.NET 平台中的程序集(exe 文件、dll 文件)在没有 CLR 环境的操作系统中不能运行的原因。

A：.NET 平台中的程序集并不是最终可以运行在操作系统中的机器指令，它只是介于源代码和机器指令之间的一个中间件，没有 CLR 的存在，就不能将该中间件转换成对应操作系统中的机器指令。换句话说，.NET 程序集不是传统意义上的可执行文件。

3. .NET 平台是否支持跨平台？与 Java 平台的区别在哪里？

A：支持，理论上讲，.NET 的跨平台和 Java 的跨平台没有差别。关于此问题，详见本章 1.4.2 小节。

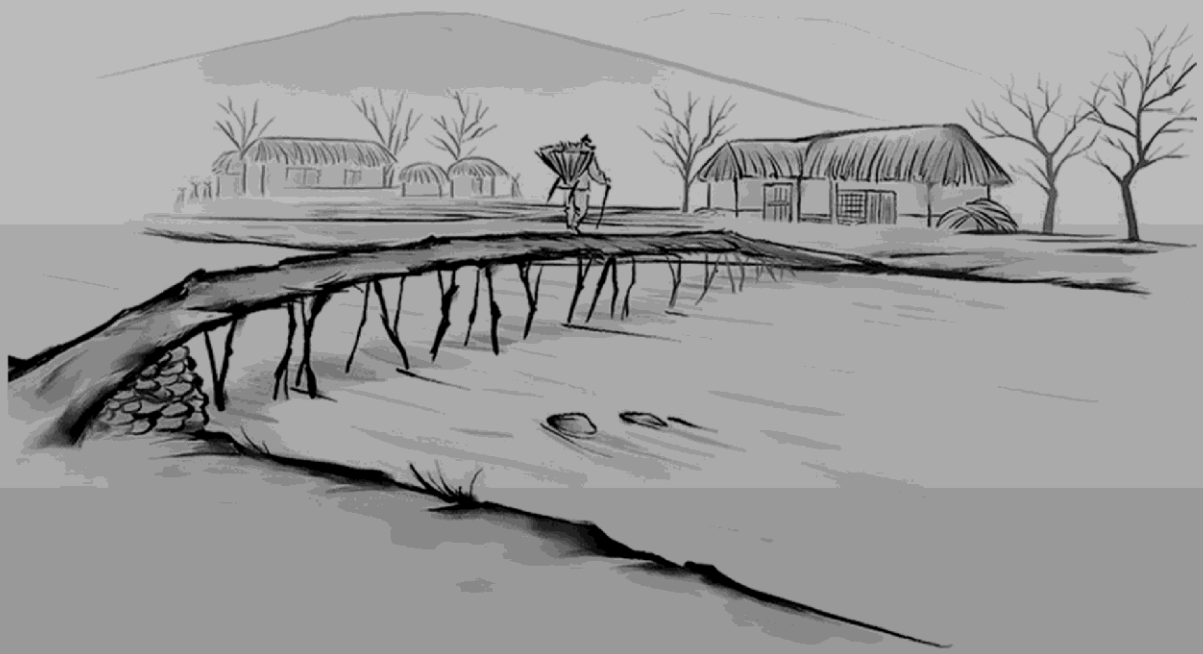
4. 在.NET 平台中程序集的“可执行性”与“可读性”分别指什么？

A：.NET 平台中的程序集是一种介于源代码和机器代码之间的中间件，对于开发者来讲，可以从程序集中读取出类似元数据、IL 代码或者资源数据等信息，而对于最终用户来讲，程序集只能在 CLR 中运行，无可读性。

第2章

高屋建瓴：梳理编程约定

在实际编程中，我们会遇见各种各样的概念，虽然有的并没有官方定义，但是我们可以自己给它取一个形象的名称。本章总结了在本书中出现的 13 条概念。



2.1 代码中的 Client 与 Server

一般来说，Client 和 Server 不仅仅可以用来形容进行网络通信的双方，还可以用来形容代码中两个有交互的代码块。

通常，通信结构中的 Client 与 Server 具有信息交互功能，并且 Server 为 Client 提供服务。代码中的一个方法调用另外一个对象的方法，同样涉及信息交互，同样可以看作是这个对象为其提供服务，如图 2-1 所示。

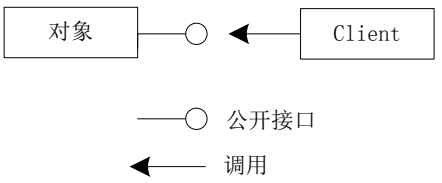


图 2-1 Client 与 Server 关系图

图 2-1 中“对象”称作 Server，Client 与 Server 可以不在一个程序集中，也可以不在同一个 AppDomain 里，更可以不在一个进程中，甚至都可以不在一台主机上。以下代码演示了 Client 与 Server 的关系：

```
//Code 2-1
class A
{
    //…
    public int DoSomething(int a,int b)
    {
        //do something here
        return a+b;
    }
}
class Program
{
    static void Main()
    {
        A a = new A();
        int result = a.DoSomething(3,4); //invoke public method a.DoSomething
    }
}
```

在代码 Code 2-1 中 a 对象是 Server，Program 是 Client，前者为后者提供服务。

注：Client 和 Server 不一定指的是对象，A 程序集调用 B 程序集中的类型，我

们可以把 A 当作 Client，把 B 当作 Server。Client 与 Server 也不是绝对的，在一定场合，Client 也可以看作是 Server。

2.2　方法与线程的关系

线程和方法没有一对一的关系，也就是说，一个线程可以调用多个方法，而一个方法又可以被多个线程调用。由于在代码中，看得见的只有方法，因此有时候程序员很难分清楚某个方法到底会运行在哪个线程之中。示例代码如下：

```
//Code 2-2
class Program
{
    static void DoSomething()
    {
        //do something here
    }
    static void Main()
    {
        //…
        Thread th1 = new Thread(new ThreadStart(DoSomething));
        Thread th2 = new Thread(new ThreadStart(DoSomething));
        th1.Start();
        th2.Start();
    }
}
```

代码 Code 2-2 中的 DoSomething 方法可以同时运行在两个线程当中。

以上代码还是比较直观的情况，有时候，在代码中一点线程的影子都看不见。示例代码如下：

```
//Code 2-3
class Form1:Form
{
         //…
    private void DoSomething()
    {
        //do something here
        //maybe invoke UI controls
    }
    private btn1_Click(object sender,EventArgs e)
    {
        BackgroundWorker back = new BackgroundWorker();
        back.DoWork += back_DoWork;
        back.Start();
```

```
        DoSomething();  //NO.1
    }
    private void back_DoWork(object sender,DoWorkEventArgs e)
    {
        DoSomething();  //NO.2
    }
}
```

在代码 Code 2-3 中有两处调用了 DoSomething 方法，一个在 btn1.Click 的事件处理程序中，一个在 back.DoWork 的事件处理程序中，前者在 UI 线程中运行，而后者在非 UI 线程中运行，两者可以同时进行。

当我们不确定我们编写的方法到底会在哪些线程中运行时，我们最好需要特别注意一下，如果方法访问了公共资源，多个线程同时执行，那么这个方法时可能会引起资源异常。另外，只要我们确定了两个方法只会运行在同一个线程中，那么这两个方法就不可能同时执行，跟方法所处的位置无关。示例代码如下：

```
//Code 2-4
class Form1:Form
{
        //…
    private void DoSomething()
    {
        //do something here
        //maybe invoke UI controls
    }
    private void btn1_Click(object sender,EventArgs e)
    {
        DoSomething();  //NO.1
    }
    private void btn2_Click(object sender,EventArgs e)
    {
        DoSomething();  //NO.2
    }
}
```

在代码 Code 2-4 中 btn1.Click 和 btn2.Click 的事件处理程序中都调用了 DoSomething 方法，但是由于 btn1.Click 和 btn2.Click 的事件处理程序都在 UI 线程中运行，所以这两处的 DoSomething 方法不可能同时执行，只可能一前一后执行，此时我们不需要考虑方法中访问的公共资源的线程是否安全。

注：正常情况下，上面的结论成立，但是如果你非要在 DoSomething 中写一些特殊代码，比如 Application.DoEvents()，那么情况就不一定了，很有可能在 btn1_Click 中的 DoSomething 方法中调用 btn2_Click 方法，从而造成 DoSomething 方法还未结

束，而另一个 DoSomething 方法又开始执行，这里涉及 Windows 消息循环的知识，本书将在第 8 章中讲到。

2.3　调用线程与当前线程

前一节中说明了线程与方法的关系，一个线程很少只调用一个启动方法，多数情况下，启动方法中会调用其他方法，一个方法在哪个线程中运行，那么这个线程就是它的当前线程。示例代码如下：

```
//Code 2-5
class A
{
    public void DoSomething()
    {
        //do something here
        Console.WriteLine("currentthread is " + Thread.CurrentThread.Name);
    }
}
class Program
{
    //the start method of main thread
    static void Main()
    {
        A a = new A();
        a.DoSomething();  //NO.1
        Thread th1 = new Thread(new ThreadStart(th1_proc));
        th1.Start();
    }
    static void th1_proc()
    {
        A a = new A();
        a.DoSomething();  //NO.2
    }
}
```

在代码 Code 2-5 中，在 NO.1 处，主线程就是调用线程，它调用了 a.DoSomething 方法，此时在 a.DoSomething 中会输出主线程的 Name 属性值。在 NO.2 处，th1 才是调用线程，它调用了 a.DoSomething 方法，这时候 a.DoSomething 中会输出 th1 线程的 Name 属性值。

也就是说，哪个线程调用了方法，哪个线程就叫做这个方法的调用线程，方法在哪个线程中运行，哪个线程就是该方法的当前线程。

2.4 阻塞方法与非阻塞方法

首先，阻塞和非阻塞的概念是相对的，一个方法耗时很长才能返回，返回之前会一直阻塞调用线程，我们叫它阻塞方法；相反，一个方法耗时很短，一调用马上就能返回，我们叫它非阻塞方法。但是这个“很长”与“很短”是相对而言的，根本就没有标准。示例代码如下：

```
//Code 2-6
class Program
{
    static void Func1()
    {
        for(int i=0;i<100;++i)
        {
            Thread.Sleep(10);
        }
    }
    static void Func2()
    {
        for(int i=0;i<100;++i)
            for(int j=0;j<100;++j)
            {
                Thread.Sleep(10);
            }
    }
    static void Main()
    {
        Func1();  //NO.1
        Console.WriteLine("Func1 over");
        Func2();  //NO.2
        Console.WriteLine("Func2 over");
    }
}
```

在代码 Code 2-6 中，Func1 相对于 Func2 来讲，耗时短，我们把 Func1 叫作非阻塞方法，Func1 不会阻塞它的调用线程，其后的 Console.WriteLine 很快就会执行；而相反，Func2 耗时长，我们把 Func2 叫作阻塞方法，Func2 会阻塞它的调用线程，其后的 Console.WriteLine 不能马上执行。

另外，在编程中，需要注意阻塞方法和非阻塞方法的使用场合，有的线程中不应该调用阻塞方法，比如 Winform 中的 UI 线程。有时候一个类会提供两个功能相同的方法，一种是阻塞方法，它会阻塞调用线程，一直等到任务执行完毕才返回，另一种是非阻塞方法，

不管任务有没有执行完毕，马上就会返回，不会阻塞调用线程，至于任务何时执行完毕，它会以另一种方式通知调用线程。这两种调用方式也称为“同步调用”和“异步调用”，FileStream.Read 和 FileStream.BeginRead 就属于这一类。

注：同步调用和异步调用在后面的章节中将会讲到，异步编程模型(Asynchronous Programming Model)是.NET 中一项重要技术。我们既可以把耗时 10s 的方法称为阻塞方法，也可以把耗时 100ms 的方法称为阻塞方法，理论上没有标准。

2.5　UI 线程与线程

UI(User Interface)线程一般出现在 Winform 编程中，主要负责用户界面的消息处理。本质上，UI 线程跟普通线程没有什么区别。

一个线程只有不停地循环去处理任务才不会马上终止，也就是说，线程必须想办法去维持它的运行，不然很快就会运行结束。在 UI 线程中包含着一个 Windows 消息循环，它通过使用常见的 While 结构实现循环。该循环不停地获取用户输入，包括鼠标、键盘等输入信息，然后不停地处理这些信息，正因为有这样一个 While 循环的存在，UI 线程才不会一开始就马上结束，如图 2-2 所示。

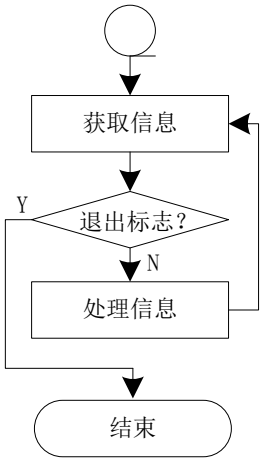


图 2-2　一个维持线程运行的循环结构

Winform 中的 UI 线程默认由 Program.Main 方法中的 Application.Run()进入，While 循环结构存在于 Application.Run()内部(或者由其调用)。

注：详细的 Windows 消息循环，请参见第 8 章。使用任何语言开发的 Windows 桌面应用程序都至少包含有一个 UI 线程。

2.6 原 子 操 作

所谓“原子”，即不可再分割的基本微粒。代码中的原子操作指代码执行的最小单元，也就是说，原子操作不可以被中断，它只有 3 个状态：未执行、正在执行和执行完毕，绝对没有执行到一半暂停下来，等待一会儿又继续执行的情况。因此原子操作又被称为程序中不可以被线程调度打断的操作。

比如给一个整型变量赋值“int a = 1;”，这个操作就是原子操作。在操作过程中不可能出现给 a 赋值到一半，操作暂停的情况。相反有很多操作，属于非原子操作，比如对一些集合容器的操作，向某些集合容器中增加删除元素等操作都是非原子操作，这些操作可能被打断，出现操作一半暂停的情况。非原子操作由许许多多的原子操作组成，原子操作与非原子操作的示例，如图 2-3 所示。

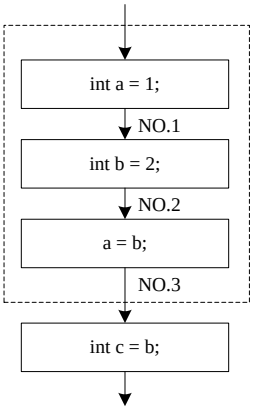


图 2-3　原子操作与非原子操作

图 2-3 中虚线框表示一个非原子操作，一个非原子操作由多个原子操作组成，虚线框中的操作可能在 NO.1、NO.2、NO.3 任何一个地方暂停。

注：原子操作与非原子操作不是以代码行数来区分的，而是以这个操作在底层怎么实施去区分的，比如“a++;”只有这一行代码，但它不是原子操作，它的底层实现是由许多原子操作组合而成的。

2.7 线 程 安 全

如果操作一个对象(比如调用它的方法或者给属性赋值)为非原子操作，即可能操作还没完成就暂停了，这个时候如果有另外一个线程开始运行同时也操作这个对象，访问了同样

的方法(或属性)，那么这时就可能会出现一个问题：前一个操作还未结束，后一个操作就开始了，前后两个操作一起出现混乱。

当多个线程同时访问一个对象时，如果每次执行都得到不一样的结果，甚至出现异常，那么这个对象便是“非线程安全”。造成一个对象非线程安全的因素有很多，除了前面提到的由于非原子操作执行到一半就中断以外，还有一种情况是由多个 CPU 造成的，即就算操作没有中断，由于多个 CPU 可以真正实现多线程同时运行，所以就有可能出现“对同一对象同时操作出现混乱”的情况，如图 2-4 所示。

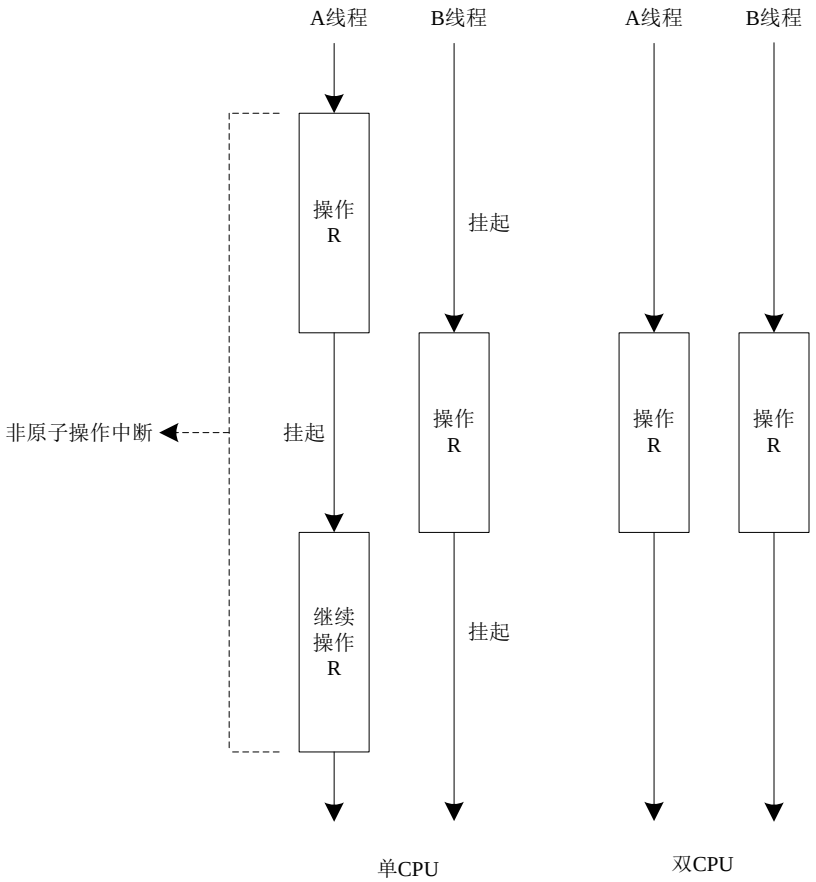


图 2-4　两种可能引起非线程安全的情况

图 2-4 中左边两个线程运行在单 CPU 系统中，A 线程中的非原子操作中断，对 R 的操作暂停，B 线程开始操作 R，前后两次操作相互干扰，可能会出现异常。图中右边两个线程运行在双 CPU 系统中，无论操作是否中断，都可能出现两个操作相互干扰的情况。

为了解决多线程访问同一资源可能引起的不稳定性，我们需要在操作方法中做一些改

进，最常见的方法是：对可能引起不稳定的操作加锁。在代码中使用 lock 代码块、互斥对象等来实现。因此，当一个对象，在多个线程访问它时，不会出现结果不稳定或异常的情况，我们就称该对象为“线程安全”，也称访问它的方法是“线程安全”的。示例代码如下：

```
//Code 2-7
class A
{
    //…
    int _a1 = 0;
    int _a2 = 0;
    object _syncObj = new object();
    public Int Result
    {
        get
        {
            lock(_syncObj)
            {
                if(_a2!=0)   //NO.1
                {
                    return _a1/_a2;  //NO.2
                }
                else
                {
                    return 0;
                }
            }
        }
    }
    public void DoSomething(int a1, int a2)
    {
        lock(_syncObj)
        {
            _a1 = a1;
            _a2 = a2;
        }
    }
    //other public methods
}
```

在代码 Code 2-7 中，单 CPU 时，如果没有 lock 块，多线程访问 A 类对象，一个线程在访问 A.Result 属性时，在判断 if(_a2!=0)为 true 后，可能在 NO.1 之后和 NO.2 之前处出现中断(线程挂起)。此时另一线程通过 DoSomething 方法修改_a2 的值为 0，中断恢复后，程序报错。双 CPU 中，如果没有 lock 块，多线程访问 A 类对象，情况更糟。一个线程访问 A.Result 属性时，不管在 NO.1 之后和 NO.2 之前会不会中断，另一个线程都有可能通过

DoSomething 方法修改_a2 的值为 0，程序报错。

另外，在 Winform 编程中，我们之所以经常会遇见“不在创建控件的线程中访问该控件”的异常，原因就是对 UI 控件的操作几乎都不是线程安全的(部分是)。一般 UI 控件只能由 UI 线程操作，其余的所有操作均需要投递到 UI 线程之中执行，否则就会像前面讲过的那样，程序出现异常或不稳定。示例代码如下：

```
//Code 2-8
class Form1:Form
{
    //…
    private btn1_Click(object sender,EventArgs e)
    {
        DealControl(null);  //NO.1
        Thread th1 = new Thread(new ThreadStart(th1_proc));
        th1.Start();
    }
    private void th1_proc()
    {
        DealControl(null);  //NO.2
    }
    private void DealControl(object[] args)
    {
        //…
        if(this.InvokeRequired)  //NO.3
        {
            this.Invoke((Action)delegate()  //NO.4
            {
                DealControl(args);
            });
        }
        else
        {
            //access ui controls directly
            //…
        }
    }
}
```

在代码 Code 2-8 中，在 DealControl 方法中需要操作 UI 控件，如果我们不知道 DealControl 最终会在哪个线程中运行(有可能在 UI 线程中运行，也有可能在非 UI 线程中运行)，那么我们可以使用 Control.InvokeRequired 属性去判断当前线程是否是创建控件的线程(UI 线程)，如果是，则该属性返回 false，可以直接操作 UI 控件，否则，返回 true，不能直接操作 UI 控件。代码中 NO.1 处直接在 UI 线程中调用 DealControl，DealControl 中可以直

接操作 UI 控件，NO.2 处在非 UI 线程中调用 DealControl，那么此时，就需要将所有的操作通过 Control.Invoke 投递封送到 UI 线程之中执行。

注：Control 含有若干个线程安全的方法和属性，常见的主要有 Control.InvokeRequired 属性、Control.Invoke 方法、Control.BeginInvoke 方法(Control.Invoke 的异步版本，后续章节有讲到)、Control.EndInvoke 方法以及 Control.CreateGraphics 方法。这些属性和方法都可以在非 UI 线程中使用，并且跨线程访问这些方法和属性时不会引起程序异常。

2.8 调用与回调

调用(Call)和回调(CallBack)是编程中最常见的概念，几乎出现在代码中的各个地方。目前在程序员中对这组概念最流行的解释是：调用是指程序员调用系统的方法，回调是指系统调用程序员写的方法，如图 2-5 所示。

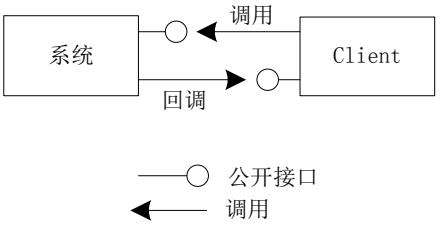


图 2-5 调用与回调的区别

需要注意的是，客户端(图中 Client 部分)并不是绝对的，即 Client 也有可能成为图中的“系统”部分，当它被调用时，它会再回调另一个 Client，如图 2-6 所示，描述的是在同一个程序中的调用与回调的关系。

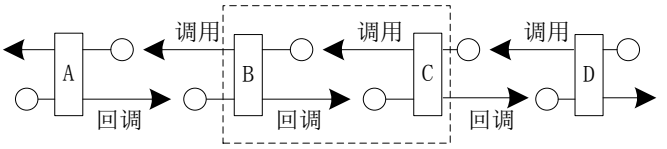


图 2-6 程序中调用与回调的关系

通常人们对“调用”和“回调”的定义只局限在图中虚线框内的部分。严格意义上讲，它只是一个小范围的规定，不应该有调用和回调之分，因为所有代码最终都是由操作系统来调用的。

.NET 平台开发中的回调主要是通过委托来实现的。委托是一种代理，专门负责调用方

法(委托的详细信息将在本书的第 5 章讲到)。

2.9 托管资源与非托管资源

在.NET 中，对象使用的资源分两种：一种是托管资源，一种是非托管资源。托管资源由 CLR 管理，不需要开发人员去人工控制，相对开发人员来讲，托管资源的管理几乎可以忽略。.NET 中的托管资源主要指“对象在堆中的内存”等资源。非托管资源指对象使用到的一些托管环境以外(比如操作系统)的资源。CLR 不会管理这些非托管资源，这就需要开发人员人工去管理控制。

.NET 中对象使用到的非托管资源主要有 I/O 流、数据库连接、Socket 连接、窗口句柄等各种直接与操作系统相关的资源，如图 2-7 所示，虚线框内部分表示“可能有”，即一个堆中对象可能使用到了非托管资源，但是它一定使用了托管资源。

图 2-7 一个堆中对象使用的资源

一个对象在使用完毕后(进入不可达状态，并不是死亡，第 4 章会讲到区别)，如果使用了非托管资源我们应该确保它能够及时释放，并归还给操作系统。至于托管资源，我们大部分时间不需要去关心，因为 CLR(具体应该是其中的 Garbage Collector 部分)会自动处理。在.NET 中使用了非托管资源的类型有很多，比如 FileStream、Socket、Font、Control(及其派生类)、SqlDataConnection 等，它们内部都封装了非托管资源；没有使用非托管资源的类型也有很多，比如 Console、EventArgs、ArrayList 等。

如何才能完美地处理一个对象使用的非托管资源，这是一门相当重要而且必须具备的技术，在第 4 章中有详细讲述。

注：现在普遍有一种错误的观点，即将 FileStream、Socket 这样的类型对象称为非托管资源。这是错误的，只能说这些对象只是使用了非托管资源。

2.10　框 架 与 库

框架(Frameworks)和类库(Library)都是一系列可以被重复使用的代码集合。二者的不同之处在于，框架是不完整的应用程序，理论上，不用写任何代码，框架本身就能运行起来；而类库多半指能够提供一些具体功能的类集合，它包含的内容和功能一般都比框架更简单。

我们使用框架去开发一个应用程序，其实就是在框架的基础上写一些扩展代码，框架就像一个没有装修的毛坯房屋，需要给它各种装饰。在装饰的过程中，可以使用类库提供的一些已封装好了的功能，如图 2-8 所示，为框架、程序以及类库三者之间的关系。

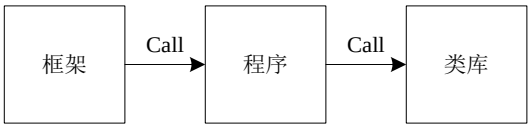


图 2-8　框架、程序以及类库之间的关系

图 2-8 中的调用关系其实是双向的，箭头所指部分显示了主要调用关系，即框架调用开发人员的代码，后者再选择性地调用一些类库。

从图 2-8 中可以看出，整个应用程序的最终控制权并不在开发人员手中，而是在框架方，这种现象称为“控制转换”(Inversion Of Control，IOC)，即程序的运行流程由框架控制。几乎所有框架都遵循这个规则。示例代码如下：

```
//Code 2-9
class Program
{
    //…
    static int GetTotal(int first,int second)
    {
        return first + second;
    }
    static void Main()
    {
        int first,second;
        Console.WriteLine("Input first:");
        first = int.Parse(Console.ReadLine());     //NO.1
        Console.WriteLine("Input second:");
        second = int.Parse(Console.ReadLine());    //NO.2
        int total = GetTotal(first,second);        //NO.3
        Console.WriteLine("the total is:" + total);
        Console.Read();
    }
}
```

代码 Code 2-9 演示了从控制台程序(即不使用框架开发)中获取用户输入的两个数据，然后输出这两个数据之和的过程。其中每个步骤的方法均由开发者自己调用(NO.1、NO.2 以及 NO.3)。

如果开发者要采用 Winform 程序(即使用框架开发)实现上述过程，则示例代码如下：

```
//Code 2-10
class Form1:Form
{
    public Form1()
    {
        //…
        this.btn1.Click+=(EventHandler)(delegate(object sender,EventArgs e)
        {
            int first = int.Parse(txtFirst.Text);      //NO.1
            int second = int.Parse(txtSecond.Text);    //NO.2
            int total = GetTotal(first,second);        //NO.3
            MessageBox.Show("the total is:" + total);
        });
    }
    private int GetTotal(int first,int second)
    {
        return first + second;
    }
}
```

代码 Code 2-10 演示了从窗体界面中的 txtFirst 和 txtSecond 两个文本框中获取数据，然后计算出两个数据之和的过程。其中每个步骤的方法都是由系统(框架)调用(在 btn1.Click 事件处理程序中)。使用框架开发的程序，代码中大部分方法都属于“回调方法”。

注：控制转换原则又称为 Hollywood Principle，即 Don’t call us, we will call you. 意思是指好莱坞制片公司会主动联系演员，而不需要演员自己去找电影制片公司。

2.11 面向(或基于)对象与面向(或基于)组件

面向(或基于)对象，面向(或基于)组件，在这组概念中，.NET 开发中经常用到的是“面向对象”，其解释分别如下。

基于对象：如果一种编程语言有封装的概念，能够将数据和操作封装在一起，形成一个整体，同时它又不具备像继承、多态这些 OO 的特性，那么这种语言就是基于对象的，比如 JavaScript。

面向对象：在基于对象的基础之上，还具备继承、多态特性的编程语言，则该编程语言是面向对象的，比如 C#、Java。

基于组件：组件是共享二进制代码的基本单元，它是一个已经编译完成的模块，可以在多个系统中重用。在软件开发中，开发者事先定义好固定接口，然后将各个功能分开独立开发，最后生成各自独立的模块。在程序运行之后，系统分别加载这些独立的模块，各个模块负责完成自己的功能，这种开发模式便是基于组件的。基于组件开发模式除了二进制代码可以重用外，还有一个优点，那就是如果需要更新某一功能，或修复某一功能中的bug，在不改变原有接口的前提下，我们不用重新编译整个程序的源代码，而只要重新编译某个组件源码即可。组件源码的语言是独立的，一种语言开发出来的组件，理论上任何一种语言都可以使用它。

面向组件：在基于组件开发中，开发者只能重用已经编译完成的二进制代码，并且不能从这个已经编译好的组件中读取其他信息，比如识别组件中的类型信息，派生出新的类型等。面向组件是指在开发过程中，开发者不仅能够重用组件中的代码，还能以该组件为基础，扩展出新的组件，比如识别.NET 程序集中的类型信息，以此派生出新的类型等。.NET 开发便是一种面向组件模式的开发。

注：如果说面向对象是强调类型与类型之间的关系，那么面向组件就是强调组件与组件之间的关系。另外，在.NET 中的组件并不包含传统意义的二进制代码。

2.12　接　　口

接口，即对外提供的、可以完成某项具体功能的通道。比如计算机上的 USB 口，通过它可以传输数据，再比如电视机的音量按钮，通过它可以调节电视机声音的大小。在.NET 开发中，接口是外界与系统(或模块)内部通信的通道。

注:“接口”概念的前提基于“封装”之上，如果没有“封装”，那么就没有“外界”与“内部”之说。

在软件一般架构设计图中，接口用以下方法表示，如图 2-9 所示。

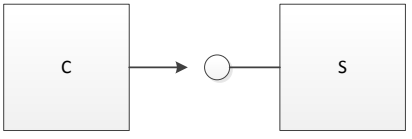


图 2-9　接口示意图

圆圈代表对外公开的通道，S 的内部细节对外界 C 是不可见的。注意图中的 S 不一定代表一个类，它可以是一个系统(跟 C 所属不同的系统)、一个模块或者其他具有“封装”功能的单元个体。

如图 2-10 所示为某些场合存在的接口。

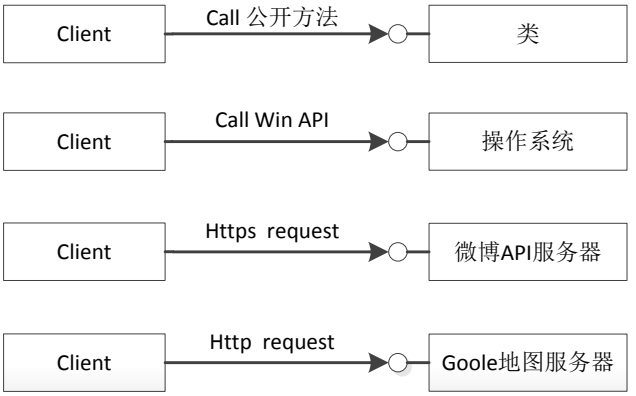


图 2-10　各种场合下的接口

图 2-10 显示了各种场合中的接口，从中可以看出，接口的概念并非仅局限于代码层面。表 2-1 显示的是各种接口的表现形式。

表 2-1　各种场合中接口的具体表现形式

序号	场合	接口的表现形式	谁是外界	说明
1	类	类的公开方法，如： People p = new People(); p.Walk();	类的使用者	类的使用者虽然不知道People类的内部具体实现，但是可以与之通信
2	操作系统	Win32 API，如： SetWindowText(hWnd,”text”); //设置某窗口标题	GUI 开发者	GUI 开发者虽然不知道操作系统的内部实现，但是可以与之通信
3	微博开放平台	https 协议 url，如加载最新微博： https://api.weibo.com/2/statuses/public_timeliti.json?parameter1=12&parameter2=22	微博第三方应用开发者	微博第三方应用开发者虽然不知道微博服务器的内部实现，但是可以与之通信
4	Google 地图服务	http 协议 url，如查询指定城市地理坐标信息： http://maps.googleapis.com/maps/api/geocode/xml?address=london&sensor=false	地图第三方应用开发者	地图第三方应用开发者虽然不知道地图服务器的内部实

				现，但是可以与之通信

在.NET 编程中，还存在另外一种意义的“接口”，即开发者使用 interface 关键字定义的接口类型。这种“接口”严格意义上讲跟前面所讨论的“接口”不能做相等比较。更准确地来说，它代表编程过程中的一种“协议”，是代码中调用方和被调用方必须遵守的契约，如果某一方不遵守这种“协议”，那么调用就不会成功。

2.13　协　　议

协议，即约定、契约，是两个或两个以上的个体在合作时需要共同遵守的准则。任何一方不遵守该准则，那么都将会导致合作失败，这是现实生活中人们通常所说的“协议”。在计算机编程世界中，“协议”带来的效果同样如此。

在计算机网络通信中，开放系统互联模型(Open System Interconnection，OSI)将网络分为 7 层功能，且每层均有多种协议，通信双方必须分别遵守各层中对应的协议，如图 2-11 所示。

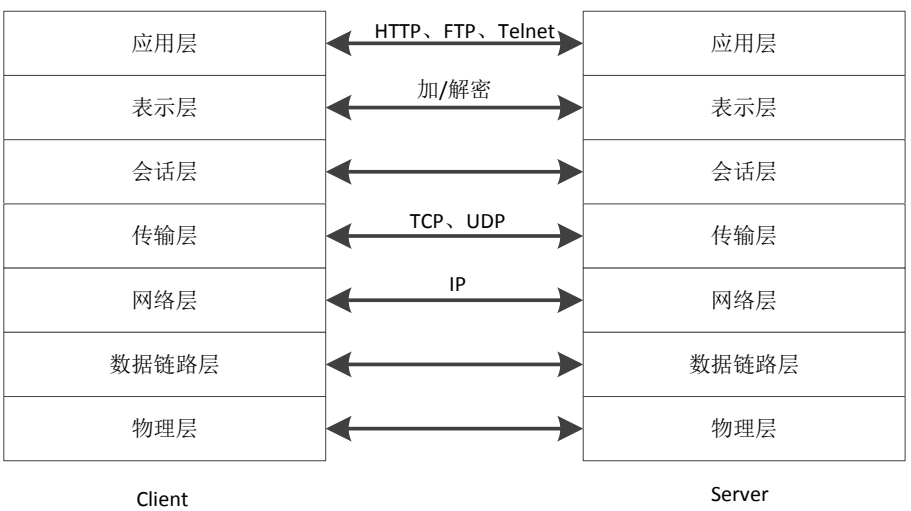


图 2-11　网络七层协议

数据发送方必须按照规定协议封装数据，然后才能发送给另一方；同理，数据接收方也必须按照对应协议解析接收到的数据包，然后才能获得发送方发送的原始数据。在实际通信编程中，这些“封装/解析”的步骤均已被计算机底层模块完成，因此对用户来讲，这些过程都是透明的，它们一直都在，并且是双方通信的关键。

网络通信协议是一种数据结构，不管是 TCP 协议还是 UDP 协议，均属于传输层协议。

对某些高级语言(如 C#、Java)开发者而言，接触这些协议的机会很少，更多时候，接触的是应用层协议，如 HTTP 协议、FTP 协议等。除了这些广为人知的协议外，在开发网络程序时，开发者也可以自己定义自己的应用层协议，如在编写雷达航迹显示系统时，可以将接收到的原始雷达数据进行预处理，以某一种预先定义的数据结构(即协议)转发给其他人，其他人按照预先定义好的数据结构(协议)去解析接收到的数据包；又如在一些即时通信程序中，可能存在文本消息、图片、表情、文件等一些数据类型，开发者完全可以定义一个自己的应用层协议，如图 2-12 所示。

图 2-12　自定义应用层协议

在图 2-12 中，第一个字节表示消息类型，是文本消息还是表情，可以通过该字节区分；第 2～5 个字节表示双方通信次数；第 6～9 个字节表示“数据区”长度，之后的 N 个字节表示发送的“原始数据”；倒数两个字节为一些附加数据；最后一个字节为校验码，整个数据结构的长度为：(1+4+4+数据区长度+2+1)个字节。发送方填充完整个数据结构，然后发送给接收方，接收方接收到数据后，按照已知的数据结构格式去解析，获得其中的原始数据。发送“文本消息”的示例代码如下：

```
//Code 2-11
public static void SendStringMsg(int sequence, string msg)
{
byte[] msg_buffer = Encoding.Unicode.GetBytes(msg);
byte[] send_buffer = new byte[12 + msg_buffer.Length];
// NO.1   1 + 4 + 4 + N + 2 + 1
using (MemoryStream ms = new MemoryStream(send_buffer))
{
using (BinaryWriter bw = new BinaryWriter(ms))
{
            bw.Write((byte)1);              //NO.2
            bw.Write(sequence);             //NO.3
            bw.Write(msg_buffer.Length);    //NO.4
            bw.Write(msg_buffer);           //NO.5
            bw.Write((short)0);             //NO.6
            bw.Write((byte)0);              //NO.7
        }
    }
    //send 'send_buffer' to receiver with socket...   NO.8
}
```

在代码 Code 2-11 中，首先定义了一个发送缓冲区(NO.1 处)，因为 12 个字节已固定，

所以缓冲区的长度应该是：12+文本消息长度，然后依次将消息类型(NO.2 处)、顺序号(NO.3 处)、数据长度(NO.4 处)、文本消息内容(NO.5 处)、附加字(NO.6 处)和校验码(NO.7 处)写入缓冲区，发送方按照预定义格式填充字节流缓冲区，再将其发送给对方(NO.8 处)。相应地接收方在接收到数据后，按照预定义格式解析字节流，如图 2-13 所示，这是顺序号为“10”，文本消息为“ABC”时发送缓冲区 send_buffer 中的内容。

名称	值		类型
send_buffer[0x00000000]	0x01	类型 1	byte
send_buffer[0x00000001]	0x0a		byte
send_buffer[0x00000002]	0x00	顺序号 10	byte
send_buffer[0x00000003]	0x00		byte
send_buffer[0x00000004]	0x00		byte
send_buffer[0x00000005]	0x06		byte
send_buffer[0x00000006]	0x00	数据长度 6 Unicode编码	byte
send_buffer[0x00000007]	0x00		byte
send_buffer[0x00000008]	0x00		byte
send_buffer[0x00000009]	0x41		byte
send_buffer[0x0000000a]	0x00		byte
send_buffer[0x0000000b]	0x42	文本消息数据 "ABC"	byte
send_buffer[0x0000000c]	0x00		byte
send_buffer[0x0000000d]	0x43		byte
send_buffer[0x0000000e]	0x00		byte
send_buffer[0x0000000f]	0x00	附加字 0	byte
send_buffer[0x00000010]	0x00		byte
send_buffer[0x00000011]	0x00	校验字 0	byte

图 2-13　发送缓冲区中的内容

注：在 TCP 通信中，由于数据是以“流”的形式传递的，因此，当前后发送的数据连接在一起时，接收方无法区分单个的消息(找不到消息边界)，若按照上面提到的预先定义一个传输协议，接收方则可以按照该协议解析出一条完整的消息。详见本书第 9 章有关“网络编程”介绍。

不仅网络通信需要“协议”的辅助，计算机世界中的很多场合都需要“协议”的辅助，如加密和解密、编码和解码以及 CPU 执行机器指令、计算机通过 USB 口与外设交换数据等，参见表 2-2。

表 2-2　各种场合中的协议

序　号	场　合	协　议	说　明
1	加密/解密	使用的同一套算法	加密和解密的算法必须配套，否则会解密失败
2	编码/解码	使用的同一种编码规范	如 Unicode、UTF-8、ASCLL，编码和解码必须使用同一套规范，否则会出现乱码
3	CPU 执行机器指令	CPU 和编译器使用的同一套 CPU 指令集	CPU 和编译器必须使用同一套指令集。传统编译器是将高级语言直

			接编译成与平台相关的机器码，而机器码只能在指定平台上运行，因此 CPU 和编译器须遵守同一个规范

续表

序　号	场　合	协　议	说　明
4	USB 接口	计算机和外设使用的同一种 USB 规范	计算机与外设必须使用同一种 USB 规范，如 USB1.0、USB1.1 或 USB2.0，否则两者之间不能正常交互(不考虑兼容情况)

至此，以上协议都跟其字面意思接近。在.NET 程序开发过程中，还有一种比较重要的“协议”，它便是使用关键字 interface 声明的接口。使用 interface 声明的接口也是一种“协议”，它规定了代码调用方与代码被调用方共同遵守的一种规范，这种协议在代码中具体体现在以下两个方面。

(1) 调用方必须存在一个接口引用。

(2) 被调用方必须实现该接口。

其具体示例代码如下：

```
//Code 2-12
interface IWalkable         //NO.1
{
    void Walk();
}
class People:IWalkable      //NO.2
{
    public void Walk()
    {
        //…
    }
}
class Program
{
    static void Main()
    {
        IWalkable w = new People();
        Func(w);
    }
    static void Func(IWalkable w)  //NO.3
    {
        w.Walk();
    }
}
```

在代码 Code 2-12 中，NO.1 处定义了一个协议(接口)，被调用方(NO.2 处)遵守了该协议

(实现接口)，调用方也遵守了该协议(NO.3 处，包含一个接口类型参数)。只有双方都遵守了同一个协议，才能更好地协调工作，如图 2-14 所示为 “协议”在代码调用中起到的作用。

图 2-14　代码调用中的协议

注：代码中使用 interface 声明的“接口”在面向抽象编程中起到了非常重要的作用，详见本书第 12 章相关介绍。

2.14　本 章 回 顾

本章共介绍了 13 个概念(术语)，或许我们曾经了解过某些概念的含义，但一直处于似懂非懂的状态，那么阅读完本章，你肯定会拍下脑袋，高呼：原来是这样！有些概念在其他地方几乎找不到准确的解释，比如“线程和方法的关系”、“库与框架区别”以及“代码中的协议”等；另外一些概念虽然能找到一些解释说明，但并没有像本章讲得这么详细。总之，本章定会扫清我们在编程道路上遇见的绊脚石。

2.15　本 章 思 考

1.下面代码中的_int_list 成员是否是线程安全的，为什么？

```
//Code 2-13
class MyContainer
{
    List<int> _int_list = new List<int>();
    public void Add(int item)
    {
        _int_list.Add(item);
    }
    public int GetAt(int index)
    {
        return _int_list[index];
    }
}
```

A :**不是线程安全的 ,因为无论是** MyContainer.Add()**方法还是** MyContainer.GetAt()**方法，均可以同时在多个线程中运行，这就意味着可能存在多个线程同时访问集合容器**_int_list**。对此可以通过在** MyContainer.Add()**以及** MyContainer.GetAt()**方法中加上锁的方法来解决该问题。**

2. 举例说明在实际开发过程中遇到的框架和库有哪些。

A：**框架有**：ASP.NET MVC、ASP.NET Webforms、Windows Forms、WCF、WPF **以及**

SilverLight 等；库包括公司内部的一些通用库，如 MySQL 数据库访问工具库、日志记录工具库、字符串处理工具库、图片处理工具库以及加解密工具库等。