.NET 编程进阶笔记

——.NET 开发要点详解

目 录

目录

目 录 …… 2

前 言 …… 8

第一章 另辟蹊径：解读.NET …… 10

1.1 前.NET 时代 …… 10
1.2 .NET 组成 …… 11
1.2.1 .NET 中的语言 …… 12
1.2.2 .NET 中的框架库 …… 12
1.2.3 公共类型系统 …… 13
1.2.4 公共语言规范 …… 13
1.2.5 公共语言运行时 …… 13
1.2.6 .NET 程序运行流程 …… 14
1.3 .NET 中的程序集 …… 15
1.3.1 程序集与 EXE 文件的区别 …… 15
1.3.2 程序集组成 …… 16
1.3.3 程序集的特点 …… 17
1.4 .NET 的跨平台 …… 17
1.4.1 Write Once, Run Anywhere 的真实现状 …… 17
1.4.2 .NET 与 Java 平台出现的目的 …… 17
1.4.3 重新看待.NET …… 19
1.5 .NET 出现的意义 …… 19
1.6 本章回顾 …… 20
1.7 本章思考 …… 20

第二章 高屋建瓴：梳理编程约定 …… 22

2.1 代码中的 Client 与 Server …… 22
2.2 方法与线程的关系 …… 23
2.3 调用线程与当前线程 …… 25
2.4 阻塞方法与非阻塞方法 …… 25
2.5 UI 线程与线程 …… 26
2.6 原子操作 …… 27
2.7 线程安全 …… 28
2.8 调用（Call）与回调（CallBack） …… 31
2.9 托管资源与非托管资源 …… 32
2.10 框架（Frameworks）与库（Library） …… 33
2.11 面向（或基于）对象与面向（或基于）组件 …… 34

2.12 接口 …35
2.13 协议 …37
2.14 本章回顾 …40
2.15 本章思考 …40

第三章 编程之基础：数据类型 …42

3.1 引用类型与值类型 …42
3.1.1 内存分配 …43
3.1.2 字节序 …45
3.1.3 装箱与拆箱 …46
3.2 对象相等判断 …47
3.2.1 引用类型判等 …47
3.2.2 简单值类型判等 …48
3.2.3 复合值类型判等 …49
3.3 赋值与复制 …51
3.3.1 引用类型赋值 …51
3.3.2 值类型赋值 …52
3.3.3 传参 …53
3.3.4 浅复制 …56
3.3.5 深复制 …58
3.4 对象的不可改变性 …62
3.4.1 不可改变性定义 …62
3.4.2 定义不可改变类型 …63
3.5 本章回顾 …65
3.6 本章思考 …65

第四章 物以类聚：对象也有生命 …67

4.1 堆和栈 …67
4.2 堆中对象的出生与死亡 …69
4.2.1 引用和实例 …69
4.2.2 析构方法 …71
4.2.3 正确使用对象 …73
4.3 管理非托管资源 …75
4.3.1 释放非托管资源的最佳时间 …75
4.3.2 继承与组合中的非托管资源 …77
4.4 正确使用 IDisposable 接口 …82
4.4.1 Dispose 模式 …82
4.4.2 对象的 Dispose()和 Close() …85
4.5 本章回顾 …87
4.6 本章思考 …87

第五章 重中之重：委托与事件 …89

5.1 什么是.NET 中的委托 …89
5.1.1 委托的结构 …89
5.1.2 委托链表 …92
5.1.3 委托的“不可改变”特性 …97

5.1.4 委托的作用……99
5.2 事件与委托的关系……101
5.3 使用事件编程……103
5.3.1 注销跟注册事件同样重要……103
5.3.2 多线程中使用事件……103
5.3.3 委托链表的分步调用……105
5.3.4 正确定义一个使用了事件的类……107
5.4 弱委托……111
5.4.1 强引用与弱引用……111
5.4.2 弱委托定义……114
5.4.3 弱委托使用场合……115
5.5 本章回顾……115
5.6 本章思考……116

第六章 线程的升级：异步编程模型……117

6.1 异步编程的必要性……117
6.1.1 同步调用与异步调用……117
6.1.2 异步调用的优点……118
6.2 委托的异步调用……119
6.2.1 BeginInvoke 与 EndInvoke……119
6.2.2 AsyncCallback……121
6.2.3 处理异步调用时的异常……123
6.2.4 异步调用的应用……124
6.3 非委托的异步调用……126
6.3.1 异步方法……126
6.3.2 FileStream.BeginRead/FileStream.BeginWrite……128
6.4 异步编程时的注意事项……129
6.5 本章回顾……130
6.6 本章思考……130

第七章 可复用代码：组件的来龙去脉……132

7.1 .NET 中的组件……132
7.1.1 组件的定义……132
7.1.2 Windows Forms 中的组件……132
7.1.3 Windows Forms 中的控件……133
7.2 容器-组件-服务模型……134
7.2.1 容器的另类定义……134
7.2.2 容器与组件的合作……135
7.2.3 窗体设计器……139
7.3 设计时（Design-Time）与运行时（Run-Time）……142
7.3.1 组件的设计时与运行时……142
7.3.2 区分组件的当前状态……143
7.3.3 组件状态的应用……145
7.4 控件……145
7.4.1 控件基类……145
7.4.2 用户自定义控件……145

7.5 本章回顾 …… 148
7.6 本章思考 …… 148

第八章 经典重现：桌面 GUI 框架揭秘 …… 149

8.1 了解传统 Win32 应用程序的必要性 …… 149
8.2 Win32 应用程序结构 …… 150
8.2.1 运行平台决定程序结构 …… 150
8.2.2 Windows 消息循环 …… 151
8.2.3 窗口过程 …… 153
8.2.4 创建基于 Win32 的单窗体应用程序 …… 155
8.2.5 创建基于 Win32 的多窗体应用程序 …… 162
8.3 .NET Winform 程序与传统 Win32 程序的关联 …… 167
8.4 Windows Forms 框架 …… 168
8.5 Winform 程序结构 …… 171
8.5.1 UI 线程 …… 171
8.5.2 消息循环 …… 172
8.5.3 创建窗体 …… 176
8.5.4 窗口过程 …… 178
8.6 模式窗体与非模式窗体 …… 181
8.7 本章回顾 …… 186
8.8 本章思考 …… 186

第九章 沟通无碍：网络编程 …… 187

9.1 两种 Socket 通信方式 …… 187
9.1.1 IP 与端口 …… 187
9.1.2 基于连接的 TCP 通信 …… 188
9.1.3 基于无连接的 UDP 通信 …… 189
9.1.4 应用层协议 …… 191
9.1.5 .NET 中 Socket 编程相关类型 …… 192
9.2 TCP 通信实现 …… 194
9.2.1 定义应用层协议 …… 195
9.2.2 编写服务端 …… 196
9.2.3 编写客户端 …… 201
9.3 UDP 通信实现 …… 204
9.3.1 定义应用层协议 …… 204
9.3.2 编写客户端 …… 205
9.4 异步编程在网络编程中的应用 …… 211
9.4.1 异步发送数据 …… 212
9.4.2 异步实现“泵”结构 …… 213
9.5 本章回顾 …… 216
9.6 本章思考 …… 216

第十章 动力之源：代码中的“泵” …… 218

10.1 “泵”的概念 …… 218
10.1.1 现实生活中的“泵” …… 218
10.1.2 代码中的“泵” …… 219

10.1.3 代码中“泵”的作用 …… 221
10.2 常见“泵”结构 …… 222
10.2.1 桌面 GUI 框架 …… 223
10.2.2 Socket 通信 …… 224
10.2.3 Web 服务器 …… 224
10.3 “泵”对框架的意义 …… 231
10.3.1 重新回到框架定义 …… 231
10.3.2 框架离不开“泵” …… 231
10.4 本章回顾 …… 232
10.5 本章思考 …… 232

第十一章 规绳矩墨：模式与原则 …… 233

11.1 软件的设计模式 …… 233
11.1.1 观察者模式 …… 233
11.1.2 Windows Forms 中的观察者模式 …… 236
11.1.3 设计模式分类 …… 237
11.2 软件的设计原则 …… 238
11.2.1 Solid 原则介绍 …… 238
11.2.2 单一职责原则（SRP） …… 239
11.2.3 开闭原则（OCP） …… 240
11.2.4 里氏替换原则（LSP） …… 242
11.2.5 接口隔离原则（ISP） …… 244
11.2.6 依赖倒置原则（DIP） …… 245
11.3 设计模式与设计原则对框架的意义 …… 248
11.4 本章回顾 …… 248
11.5 本章思考 …… 249

第十二章 难免的尴尬：代码依赖 …… 251

12.1 从面向对象开始 …… 251
12.1.1 对象基础：封装 …… 251
12.1.2 对象扩展：继承 …… 255
12.1.3 对象行为：多态 …… 258
12.2 不可避免的代码依赖 …… 260
12.2.1 依赖存在的原因 …… 260
12.2.2 耦合与内聚 …… 262
12.2.3 依赖造成的“尴尬” …… 263
12.3 降低代码依赖 …… 265
12.3.1 认识“抽象”与“具体” …… 265
12.3.2 再看“依赖倒置原则” …… 266
12.3.3 依赖注入（DI） …… 269
12.4 框架的“代码依赖” …… 272
12.4.1 控制转换（IoC） …… 272
12.4.2 依赖注入（DI）对框架的意义 …… 273
12.5 本章回顾 …… 273
12.6 本章思考 …… 273

前 言

“勤于总结”一直都是学习的最好方式之一，编程也一样，总结的过程会促使您持续思考，从一个知识点不断地扩散，对于之前已经了解的知识，您可以理解得更加深刻，而对于那些在总结过程中接触到的新知识，也是一种额外的收获。一个不会总结的技术开发者难以成为一名优秀的程序员，笔者习惯将工作中的技术心得记录在自己的博客中，本书便是笔者从事.NET开发多年来的一个技术总结，希望这些技术总结能够给.NET开发者带来一些帮助。

1．本书特色

.NET 技术有着广泛的应用范畴，从 Windows 桌面应用，到 ASP.NET Web 应用，到 WCF 分布式应用，到 Window Mobile 嵌入式应用，到 Windows Phone 移动应用，到 ADO.NET 数据库应用，到 XML Web Service，.NET 无处不在。现在市面上不乏关于.NET 开发的书籍，笔者在本书中避免了介绍一些简单的基础内容，例如类型的声明、语法、开发环境介绍或者关键字的介绍等；同时也没有去介绍一些平时几乎用不到的高深莫测的技术，如代码安全性或者 IL 语言这样底层的内容。本书总结了笔者认为在实际.NET 开发工作中比较实用的一些知识点，如解释了诸如“原子操作”、“阻塞方法与非阻塞方法”、“框架与库”、“调用与回调”等这些常见但没有标准定义的编程术语；重点阐述了.NET 开发的三大基础知识点：数据类型、对象的生命期以及委托与事件；还详细介绍了“泵”结构在一些主流框架诸如 Windows 桌面 GUI 框架中的应用，以及它在 Socket 网络编程、Web 服务器开发等实际项目中起到的关键作用；在软件架构方面，本书分别从软件设计模式、软件设计原则以及代码依赖三个方面一一进行说明。总之，这是一本注重实际开发、接地气的.NET 技术书籍。

2．读者对象

本书适合已经入门、有一定编程经验并准备向高手迈进的.NET 开发者，包括：

- .NET 工程师
- 在校计算机相关专业.NET 方向大学生
- 从其它面向对象语言转向学习.NET 编程的开发者

3．主要内容

全书共分十二章。第 1~7 章为基础篇，分别涉及到了.NET 平台介绍、编程术语解释、数据类型、对象生命期、委托与事件、异步编程以及.NET 中的组件；第 8~12 章为设计篇，主要介绍到了代码中的“泵”结构、软件设计模式、软件设计原则以及代码依赖。其中第 8 章和第 9 章主要为了引入代码中“泵”的概念，为第 10 章讲“泵”结构做铺垫。

第一章主要讲述了.NET 平台的组成，强调.NET 平台出现的意义是改善了传统 Windows 开发模式，而非专门给我们提供一个跨平台的开发支持。

第二章是笔者从这几年的工作经历中总结出来的有关.NET 编程的一些术语概念，虽然有些可能没有官方定义，但是笔者认为了解这些概念对我们实际开发很有帮助。

第三章介绍了.NET 编程的基础，即数据类型。笔者分别从值类型与引用类型的内存分配、赋值与复制以及对象判等等方面进行一一说明。

第四章讲述了.NET 对象的“从生到死”，以及怎样管理好对象的非托管资源，最后提出了正确实现 Dispose 模式的方法，并说明为什么 Dispose 一个对象并不代表该对象死亡。

第五章讲了委托的结构以及它在.NET 编程中的重要地位，程序的执行过程就是方法的调

用过程，委托作为调用方法的一种手段，贯穿着整个.NET 编程领域，之后说明了.NET 中事件与委托的关系，并且提到了事件编程的标准规范。

第六章讲到了“异步编程模型”，阐述了异步编程的必要性，介绍怎样使用委托去异步调用一个方法，以及异步编程过程中需要注意的一些事项，比如跨线程访问、线程安全等等。

第七章介绍了.NET 编程中组件的定义，以及组件存在的意义，还介绍了“容器-组件-服务模型”，并提到了一般集成开发环境（如 Visual Studio）中窗体设计器的原理，之后介绍了组件的两种状态：运行时和设计时，以及区分组件当前状态的方法。

第八章主要重温了一下经典桌面 GUI 框架，介绍了“Windows 消息”、“消息泵”以及“窗口过程”等概念，本章引入了代码中“泵”的概念。

第九章介绍了两种 Socket 网络通信模式，即 TCP 通信与 UDP 通信，并强调了“泵”结构在网络通信中的应用。

第十章总结了“泵”结构的几个常见应用场景，如 Socket 通信、桌面 GUI 框架以及 Web 服务器的实现等。

第十一章介绍了软件开发中的设计模式与设计原则，这章讲到的设计模式几乎只是皮毛（实际上只是以“观察者模式”为例），另外详细介绍了软件设计中的五大设计原则，简称“Solid 原则”（英文名称首字母拼写而成）。

第十二章讲到了代码依赖不可完全避免的原因、代码依赖给开发过程造成的负面影响以及应该怎样去降低代码依赖程度。

4．关于注释与配图

全书中的配图和示例代码较多，笔者为了让代码显得更干净，再加上在代码中注释的空间不够，因此示例代码中索性就没有详细的中文注释（个别地方有简单英文单词注释），为了让读者更好地理解示例代码和配图，笔者在每段代码和配图下面都加上了详细的中文说明文字。本书中所有示例代码均为“说明性”代码，并不确保能在编译器中编译通过，代码语言为 C#，默认.NET 版本为.NET Framework 3.5，不过笔者认为本书中讲到的内容受.NET 版本以及语言影响不大。另外全书中以“注”字开头的提示文字更值得阅读，不容错过，因为那些很多并不是补充性说明，而是笔者认为那些更值得我们注意，所以提出来单独放到了一起。

5．勘误与支持

由于笔者技术功力有限，加上语言表达能力一般，书中如果出现错误或者一些不准确的地方，恳请各位读者提出批评指正。如发现错误或疏忽，可以发送邮件至 zhouzhi@syxysoft.com 与笔者取得联系，笔者会第一时间做出应答。如果在阅读本书时有任何疑问，也可以发送邮件与笔者进行讨论。

第一章 另辟蹊径：解读.NET

本章主要说明.NET 平台的组成以及出现的意义，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 平台是怎样改变了传统 Windows 开发模式。

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不一样，使用不同语言的开发人员之间很难有交互点。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）关联的机器指令，如下图 1-1：

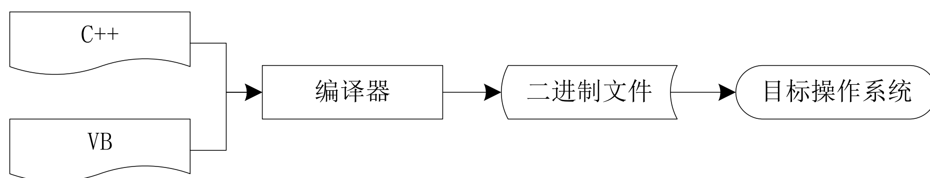


图 1-1 传统开发流程

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（二进制文件）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个系统中，开发人员必须重新编译源代码，生成与另外一个系统相匹配的二进制指令。

出现以上情况最主要的一个原因是在编译环节，因为编译器生成的二进制指令只能是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期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，或者说一些从前不流行的操作系统慢慢开始流行起来，这对开发人员来讲无疑是不利的，因为我们现在必须让我们开发出来的程序去适应各种各样的操作系统，而前面讲到过，想让我们的程序跑在不同的系统中，我们必须再重新编译我们的源代码。

注：重新编译我们的源代码只是表面上的，更多的时候是要修改源代码，原因后面有章节提到。

有问题总会有人想到解决问题的办法，上世纪中期，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跑在任何一个操作系统之中，而不需要开发人员重新编译他的源代码。这个想法无疑是空前绝后的，我们不再需要为了让我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”也成为了当时 Java 平台推广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，因此，编译出来的二进制指令只能被该操作系统识别。Sun 公司就是在这个“编译阶段”上做了手脚，让编译器编译出来的中间件不跟任何操作系统关联，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。这样给人的感觉就是，我们只需要编译一次我们的程序，之后它就可以跑在任何一个系统之中。

Java 平台程序开发到运行见下图 1-2：

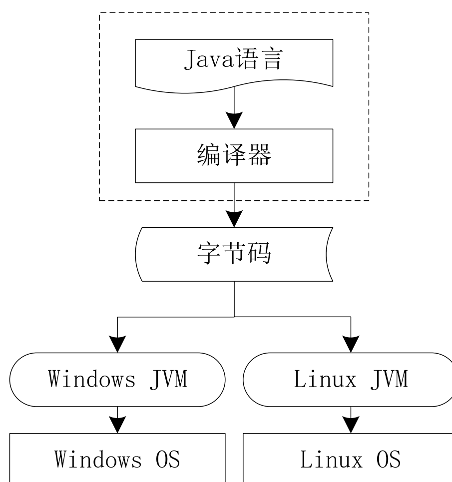


图 1-2 Java 平台程序开发运行流程图

如上图 1-2 所示，虚线框为开发人员负责的部分，理论上，不管最终程序运行在哪里，开发人员只需要做同样的事情一次，剩下的对于开发人员来讲，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 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 开发模式，它重新定义了 Windows 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，无论我们熟悉哪种语言都可以开发.NET 程序，每种语言编写出来的组件（程序集）之间可以毫无障碍地通讯，我们可以从 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，这个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，我们修改一个 DLL 中的公共类型，只要使用了该 DLL 的客户端没有用到该公共类型，那么客户端就不需要重新编译，这些在

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 平台主要包括：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。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，下图 1-3 为.NET 平台组成：

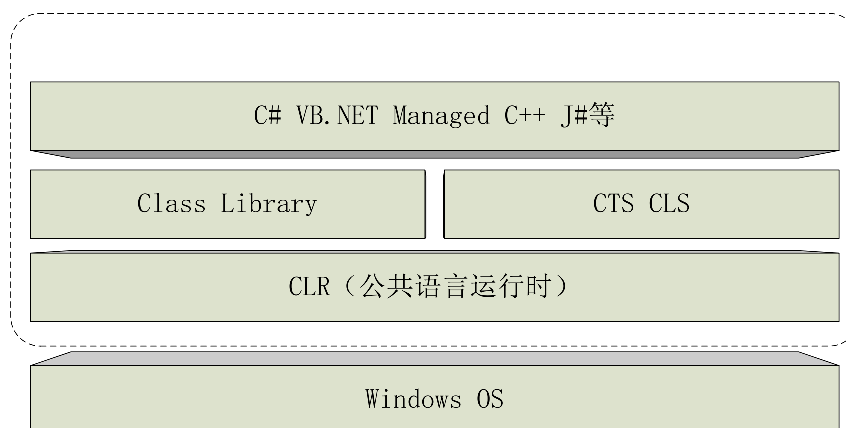


图 1-3 .NET 平台组成

图 1-3 中列出的语言仅仅为微软官方支持的几种语言。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，其它几种语言平时用得也是非常少的。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，都是可以成为.NET 平台语言大家庭中的一员。像 COBOL、Python、Delphi 等都有支持.NET 平台的版本。

.NET 中虽然语言种类繁多，但是却可以毫无障碍地交互，也就是说我们使用 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，这也就是.NET 平台中的“语言集成”特性。

> 注：
>
> [1]J++为微软早年以 JAVA 语言为基础弄出来的一种语言，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。
>
> [2]上文中提及到的程序集，是一种类似前面介绍 JAVA 平台时说到的“中间件”，下一节有说到。

1.2.2 .NET 中的框架库

何为库？库就是别人写好了的有组织有结构的代码集合，我们可以拿过来直接使用，大到传统流行一时的 MFC、ATL 等框架，小到自己公司写的一些通用工具类。.NET 平台本身包含一个非常丰富的类库，范围涉及到数据结构、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”的概念不应该陌生。

注：库和框架本身没有明显的区分定义，如果我们想了解更多，请参见第二章。

1.2.3 公共类型系统

前面讲到.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一个类到底是使用 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必须遵守的规范——公共类型系统（Common Type System），.NET 平台语言大家庭中所有语言都必须遵守这个规范。比如有我们常见的值类型（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，各种语言之间可以无障碍交互，我们可以在 C#中派生出一个使用 VB.NET 编写类型的子类，我们还可以在 VB.NET 中捕获 Managed C++中抛出的异常，也就是说继承、多态、异常处理等等这些都可以交叉使用。然而，每一种语言本身的语言规则是不相同的，如 Managed C++中大小写敏感，而 VB.NET 中大小写不敏感。正因为不同的语言之间可能支持不同的特性，而这些不同特性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 ），.NET 平台语言大家庭中的所有语言都必须遵守这个规范。

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）是.NET 平台的核心，可以把它当作.NET 平台的心脏。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，我们可以把它当作一个代码数据集合，这个代码数据集合并不能直接在操作系统中运行，还是需要像 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”。由此可见，.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。

传统开发过程中的编译有且仅有一次，源代码经过编译之后直接生成了二进制指令，一步到位，该过程由开发者负责。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为“非完全编译”，见下图 1-4：

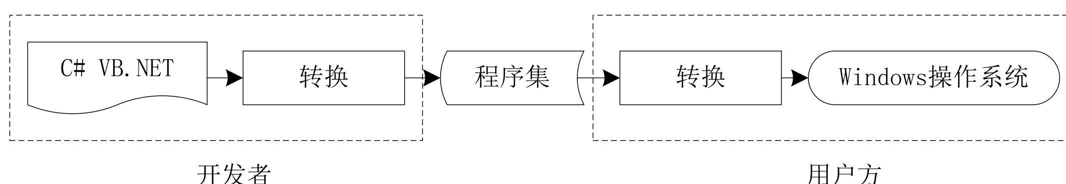


图 1-4 .NET 平台程序的两次转换

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，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。这也就是为什么我们经常在 WIN 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。

CLR 中包含一个名叫 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 的工作流程见下图 1-5：

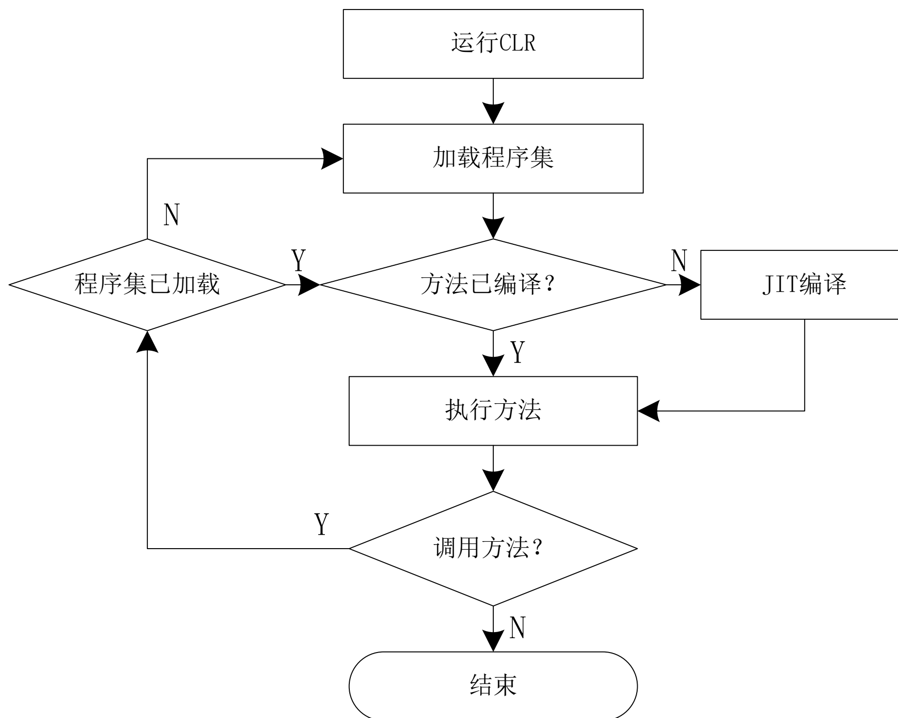


图 1-5 CLR 工作流程

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编译该部分代码（如果该代码从未编译过）。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 Just-In-Time 这样的编译器，还提供统一内存管理、异常处理、安全检查、访问 COM 和 Win32 等这些功能。无论我们使用什么语言编写的托管代码，只要在 CLR 中运行，它都能为我们提供这些服务。

注：与操作系统关联的可以直接运行的代码称为本地代码（NATIVECODE）或者非托管代码（UNMANAGEDCODE），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，像在.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（MANAGEDCODE），该解释在第二章中有详细说明。

1.2.6 .NET 程序运行流程

开发人员选择一种语言编写完程序后，经过相应编译器编译成一种中间件（程序集），CLR

加载该中间件，使其运行。见下图 1-6：

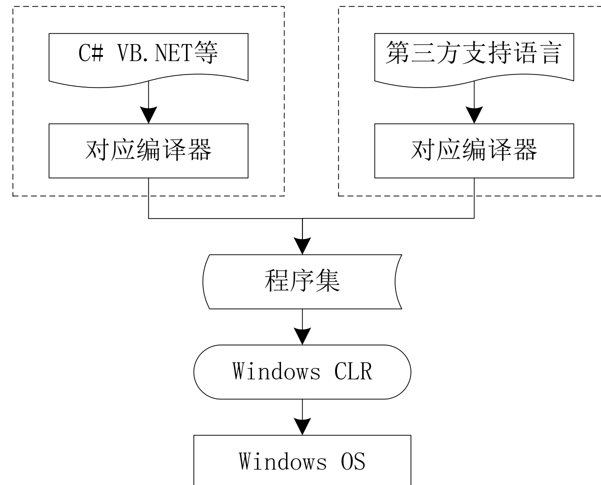


图 1-6 .NET 平台程序开发运行流程图

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，这个也就是.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“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。

1.3 .NET 中的程序集

前面几节提到过，开发人员将源文件编译之后生成的中间件，我们称之为“程序集”。因为程序集文件名一般以.EXE（或者.DLL）结尾，因此很容易将它与传统开发过程中的 EXE 文件（或者 DLL 文件）搞混淆。虽然它们的后缀名相同，但本质上却千差万别。

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 CLR 的，是.NET 平台范畴的东西，EXE 文件可以说主要是面向 Windows 操作系统的；

其次，它们的组成结构不一样，程序集中除了包含代码（一种叫 IL 语言的代码）之外还有很多其它东西，比如类型信息、版本信息、引用其它程序集信息、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”（self-Introduce）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，程序集不仅可以运行（运行在 CLR 中），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 文件的功能区别见下图 1-7：

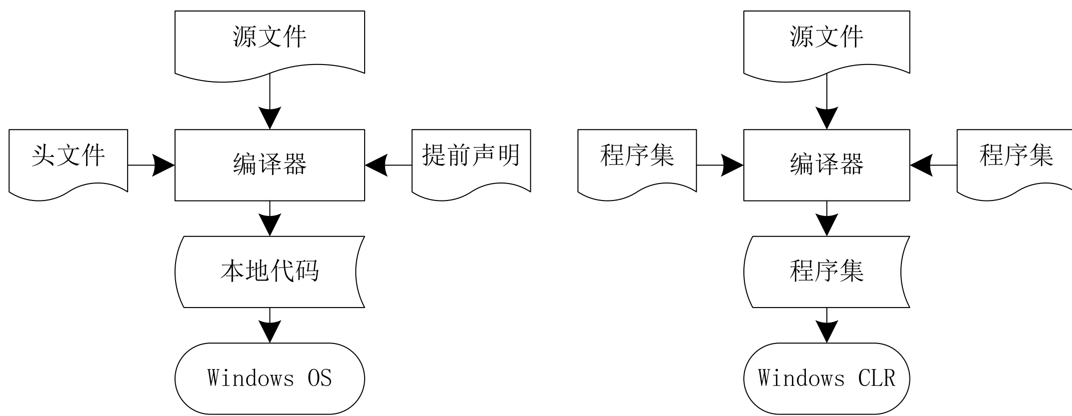


图 1-7 程序集与本地代码的功能区别

图 1-7 中左边由编译器编译出来的本地代码除了可以运行，其余什么都干不了，而右边由编译器编译出来的程序集除了可以运行在 CLR 中，还可以用在开发阶段。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：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，因为程序集本身保存有版本、安全加密等信息，这些信息都是可以起到关键作用，后续有讲到。

1.3.2 程序集组成

前面一小节中提到过，程序集中包含有代码，类型信息，版本信息，引用信息以及一些资源等。程序集的组成结构见下图 1-8：

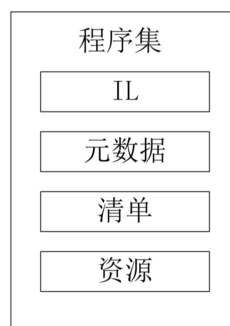


图 1-8 程序集组成

图 1-8 中 IL（Intermediate Language）是一种语言，程序集中的代码部分由它表示，从名字就可以看出它是一种“中间语言”，也就是说，它跟生成它的源语言（C#或者 VB.NET 等）是无关的，用 C#和 VB.NET 定义同一个 class，编译之后生成的 IL 是等效的。元数据主要包含这个程序集中的命名空间、类型等信息，这些信息通过某些手段是可以读出来的。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它程序集的一些信息。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是可以使用的，包括字符串、图片以及特殊文件等等。

正是程序集的这种特殊组成结构，才能使程序集与传统 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，是它具备特殊功能的前提。也正是程序集的存在，才使.NET 平台开发模式比传统 Windows 开发模式更方便快捷。程序集有如下四个特征：

（1）语言独立。

（2）二进制兼容。

（3）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"这句话中的 Write 是相对于开发人员来讲的，而 Run 则是相对于最终用户来说的。意思是说开发人员只需要编写、调试、编译一次源程序，生成的可运行程序可以跑在任何一个最终用户的任何一个操作系统之上。这句伟大而又霸气的话是从 Java 平台刚出来的时候流行起来的，同时它也成为了曾经推广 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跑在一种环境中，表面上看是将程序交给这种环境托管运行，我们称这个时代为"托管时代"。

愿望总是好的，但是真正要开发出百分百"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，结果会怎样？假如我们的源程序中本身就使用了 A 操作系统中独有的 A_fun 这个 API，我们却想让我们的程序编译之后跑在 B 操作系统之中，那么在第二次编译阶段，B 操作系统由于没有 A_fun 这个 API，编译器又应该怎样去翻译这个 A_fun 呢？这个时候，我们编写的程序也就只能跑在 A 操作系统中。

出现以上这种不能跨平台的情况，主要原因是在开发阶段源程序中就使用了一些与某个具体操作系统相关联的框架或者库，这些库最终并不能完整地（或者准确地）转换成另一操作系统中的本地代码。所以要让我们的 Java 程序能够更好的跨平台，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，这也是导致 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，越来越多的 Java 第三方库出现，而这些第三方库大多数是针对特定操作系统的，因此，跨平台就越来越难实现。

1.4.2 .NET 与 Java 平台出现的目的

到目前为止，在笔者对.NET 平台的所有介绍里面，几乎没有提到它关于“跨平台”的任何信息，对.NET 平台所有的介绍也全部限制在 Windows 操作系统之上，之所以这样，不是因为.NET 不具备“跨平台”的特点，只是为了纠正目前很多人对.NET 平台的误解，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。

前面对.NET 平台所有的介绍都是为了说明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。.NET 平台的结构虽然跟 Java 平台极其类似，但是后者主要目的是为了解决程序跨平台运行的问题。下图 1-9 为两个平台的侧重点：

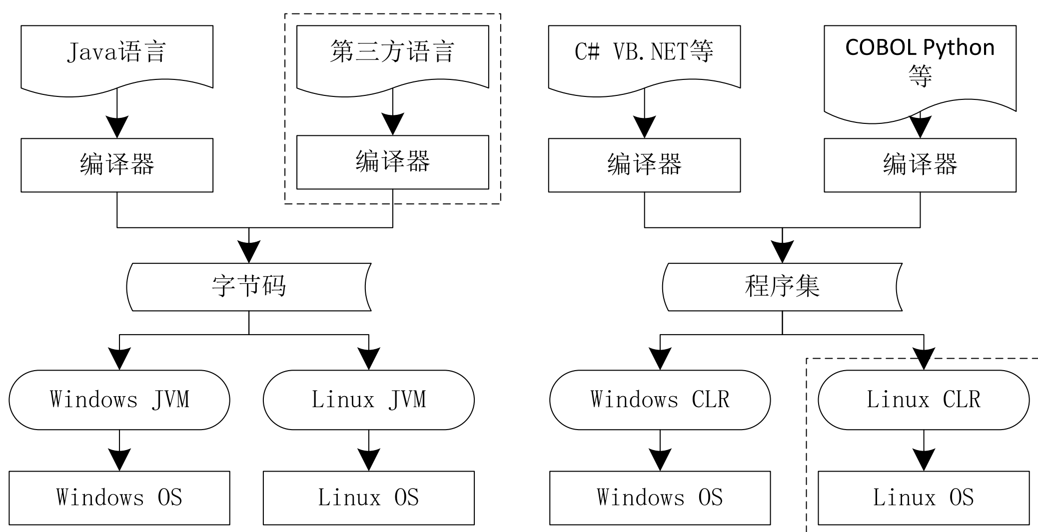


图 1-9 .NET 平台与 Java 平台的侧重点

图 1-9 中虚线部分为平台之外的功能，理论上 Java 平台也是可以支持多种语言，并且已经有第三方语言在 Java 平台中使用，只是小范围并没有得到推广。另一方面，.NET 平台也是可以支持跨平台的，只要有对应操作系统的 CLR 存在，.NET 程序也是可以运行在除了 Windows 之外的其它操作系统之中，比如比较流行的 Mono，它能让我们的.NET 程序跑在 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？这个主要因为两个平台追求的目标不一样。Sun 公司认为：在互联网世界中，我要让一种语言跑在任何一个操作系统之中；微软则认为：在互联网世界中，我要让所有的语言跑在同一个操作系统之中。很明显，微软所说的同一个操作系统就是指 Windows。

> 注：有一种说法是，微软早起推广.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。不管是哪种说法，.NET 平台现在的主要目的不是支持跨平台。

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？而且这个概率比 Java 平台大得多？其实上一小节中已经给出了答案，程序是否能从一个操作系统移植到另外一个操作系统，关键要看我们开发程序的时候使用了哪些框架库，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），我们源程序中如果使用了这种框架库，是很难在另外的操作系统中将其翻译出来的。很典型的一个例子就是.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，我们知道微软发布.NET 平台的目的，它不是为我们开发跨平台的程序提供便利，因此，我们应该清楚如果我们的程序需要支持跨平台，那么最好不要选择.NET 开发，当然，如果我们开发的程序最终运行在 Windows 之中，那么.NET 平台无疑是最好的选择。

1.5 .NET 出现的意义

前面讲过，虽然.NET 平台有着与 Java 平台类似的结构，但是它真正的主要目的是优化传统 Windows 开发模式，使 Windows 开发工作更加方便快捷。体现在以下几个方面：

（1）将“面向对象”和“面向组件”结合于一体。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，但是又可以从中读取类型信息，从中派生出来新的类型，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）集成了所有开发语言，满足不同的开发人员。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，我们也可以在 Managed C++中捕获 C#编写的类型抛出的异常。

（3）解决了 COM 时代的 DLL HELL（DLL 地狱）问题。每个组件（程序集）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（side-by-side），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）真正实现了“二进制兼容性”。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。

```
//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）部署方便。.NET 平台中，一切私有程序集均可以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）之中，不像 COM 中需要将组件信息写入注册表。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由于不兼容而造成依赖旧版程序集的程序无法正常运行。

（6）统一了传统开发模式中各种各样的库。传统开发模式中存在各种各样的框架库，常见 MFC、ATL 以及 STL 等等，每种框架库的使用方法不一样，各种语言不能共享，.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，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）CLR 提供了丰富的功能。CLR 提供了内存管理功能，我们不用像传统 C++开发中那样担心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，让我们不管使用哪种语言，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 还提供了安全检查功能、异常处理功能等等。

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。

[3]DLL HELL（DLL 地狱）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，而这时候 B 程序还是要依赖旧版的 A.DLL，于是就会导致 B 程序无法正常运行。

1.6 本章回顾

每本关于.NET 技术的书籍总会有一章专门用来介绍.NET 这一平台，本书也不例外。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，这些缺陷激发了.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；本章最后总结了.NET 平台出现的意义，它完善了现有 Windows 软件开发模式，使得 Windows 软件开发模式更方便快捷。

1.7 本章思考

1.简述.NET 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运行在 CLR 中，你可以把它看成另外一个操作系统。

2.简述.NET 程序集（exe 文件、dll 文件）不能在无 CLR 环境的操作系统中运行的原因？

A：.NET 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 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..NET 中程序集的“可执行性”与“可读性”分别指什么？

A：.NET 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，程序集是可以在 CLR 中运行的。

第二章 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 13 条会在本书中出现的概念。

2.1 代码中的 Client 与 Server

我们一般说到 Client 和 Server，就会联想到网络通讯，TCP、UDP 或者 Socket 这些概念马上就浮现在头脑中。其实 Client 和 Server 不仅仅可以用来形容网络通讯双方，它还可以用来形容代码中两个有交互的代码块。

通讯结构中的 Client 与 Server 有信息交互，一般 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，同样涉及到信息交互，也同样可以看作这个对象为其提供服务。见下图 2-1：

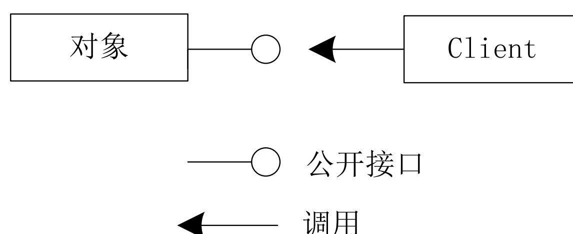


图 2-1 Client 与 Server 关系图

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，甚至都不在一台主机上。下面代码演示了 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
```

```
    }
}
```

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，我们看得见的只有方法，因此有时候我们很难分清楚某个方法到底会运行在哪个线程之中。

```
//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，一点线程的影子都看不见，

```
//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
        }
}
```

上面代码 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，那么这两个方法不可能同时执行，跟方法处在的位置无关，

```
//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
        }
}
```

上面代码 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，此时我们不需要考虑方法中访问的公共资源是否线程安全。

> 注：正常情况下，上面的结论成立，但是如果你非要在 DoSomething 中写了一些特殊代码，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，另一个 DoSomething 方法又开始执行，这个涉及到 Windows 消息循环的知识，本书第八章有讲到。

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它方法，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，

```
//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
    }
}
```

上面代码 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，这时候 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，一个方法耗时短，一调用马上就返回，我们叫它非阻塞方法。但是这个“很长”与“很短”根本就没有标准，

```
//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");
    }
}
```

上面代码 Code 2-6 中，Func1 相对于 Func2 来讲，耗时短，我们把 Func1 叫做非阻塞方法，Func1 不会阻塞它的调用线程，下面的 Console.WriteLine 很快就会执行；而相反，Func2 耗时长，我们把 Func2 叫做阻塞方法，Func2 会阻塞它的调用线程，下面的 Console.WriteLine 不能马上执行。

现实编程中，也没有严格标准，如果一个方法有可能耗时长，那么就把它当作阻塞方法。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 线程一般出现在 Winform 编程中，主要负责用户界面（User Interface）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。UI 线程中包含一个 Windows 消息循环，使用常见的 While 结构实现，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 循环存在，UI 线程才不会一开始就马上结束。

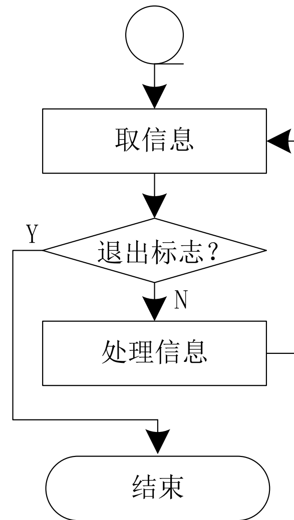


图 2-2 一个维持线程运行的循环结构

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，请参见第八章。使用任何语言开发的 WINDOWS 桌面应用程序都至少包含一个 UI 线程。

2.6 原子操作

所谓“原子”，即不可再分的意思。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，它只有三个状态：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，又继续执行的情况。原子操作又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，不可能有给 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。

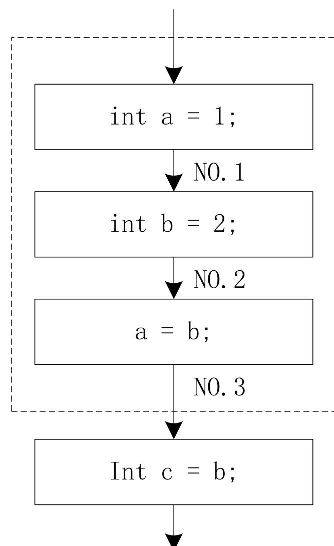


图 2-3 原子操作与非原子操作

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，是以这个操作在底层怎么实施去区分的，比如"A++;"只有一行代码，但是它不是原子操作，它底层实现是由许多原子操作组合而成。

2.7 线程安全

我们操作一个对象（比如调用它的方法或者给属性赋值），如果该操作为非原子操作，也就是说，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），这个时候可能会出现一种问题：前一个操作还未结束，后一个操作就开始了，前后两个操作一起就会出现混乱。

当多个线程同时访问一个对象（资源）时，如果每次执行可能得到不一样的结果，甚至出现异常，我们认为这个对象（资源）是“非线程安全”的。造成一个对象非线程安全的因素有很多，上面提到的由于非原子操作执行到一半就中断是一种，还有一种情况是多 CPU 情况中，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，所以还是有可能出现“对同一对象同时操作出现混乱”的情况。

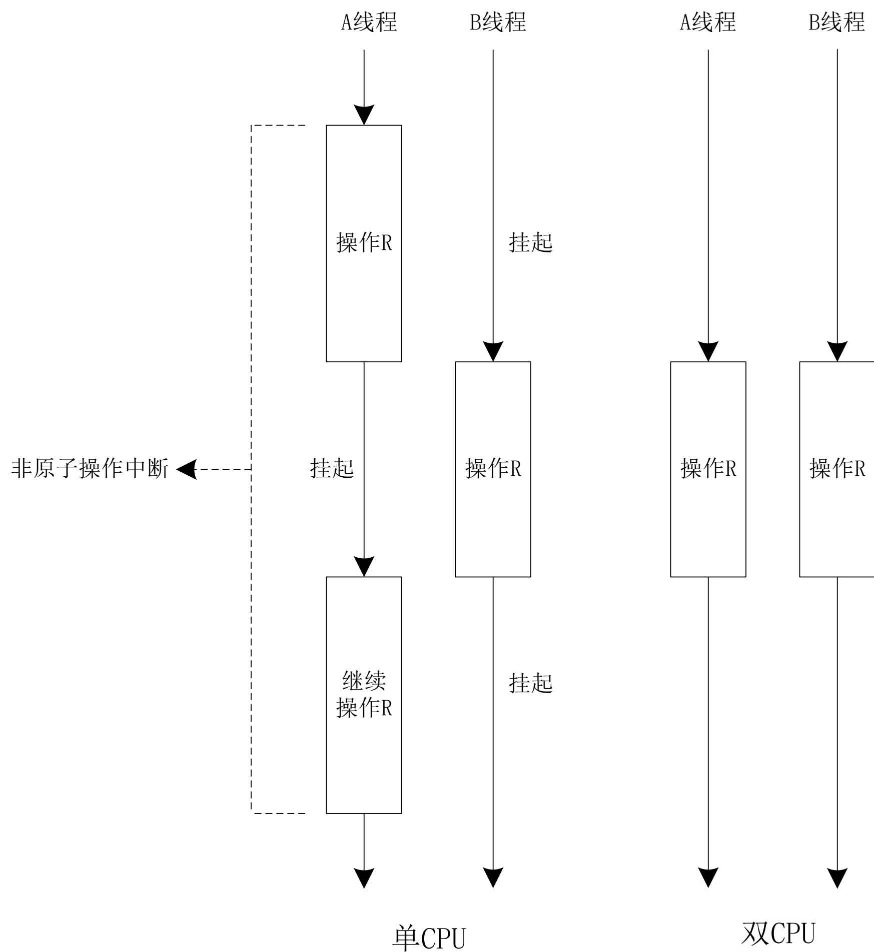


图 2-4 两种可能引起非线程安全的情况

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，可能出现异常。图中右边两个线程运行在双 CPU 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有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。如果一个对象，在多个线程访问它时，不会出现结果不稳定或异常情况，我们称该对象为“线程安全”的，也称访问它的方法是“线程安全”的。

```
//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
}
```

上面代码 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，我们常遇见的“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，否则就像前面讲的，程序出现异常或不稳定。

```
//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
            //...
        }
    }
}
```

上面代码 Code 2-8 中，DealControl 方法中需要操作 UI 控件，如果我们不知道 DealControl 到底会在哪个线程中运行，有可能在 UI 线程也有可能在非 UI 线程，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 包含有若干个线程安全的方法和属性，我们可以在非 UI 线程中使用它们。有 CONTROL.INVOKEREQUIRED 属性、CONTROL.INVOKE、CONTROL.BEGININVOKE（CONTROL.INVOKE 的异步版本，后续章节有讲到）、CONTROL.ENDINVOKE 以及 CONTROL.CREATEGRAPHICS 方法。跨线程访问这些方法和属性不会引起异常。

2.8 调用（Call）与回调（CallBack）

调用（Call）和回调（CallBack）是编程中最常遇见的概念之一，几乎出现在代码中的每一处，只是许多人并没有在意。现在最流行的解释是：调用指我们调用系统的方法，回调指系统调用我们写的方法。类似下面图 2-5 描述的：

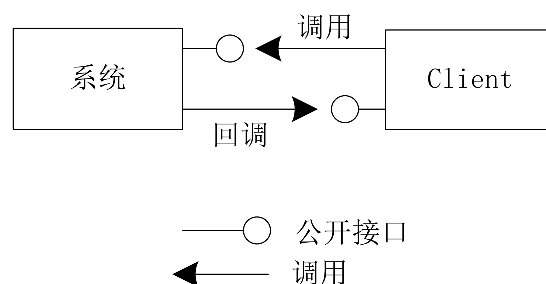


图 2-5 调用与回调的区别

上图 2-5 是目前对“调用”和“回调”的解释。但是需要清楚一点，本章第一节中已经讲到过，客户端（图中 Client）并不是绝对的，也就是说，Client 也有可能成为图中的“系统”部分，别人再调用它，它再回调另一个 client。

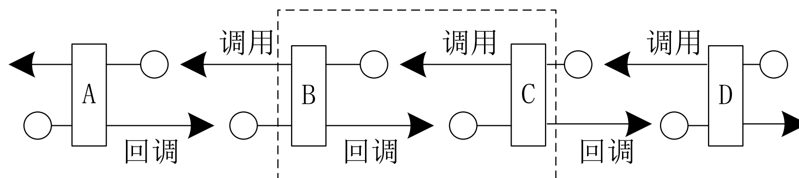


图 2-6 程序中调用与回调的关系

图 2-6 中描述一个程序中调用与回调的关系，我们平常对“调用”和“回调”的定义只局限在图中虚线框中，它只是一个小范围的规定。严格意义上讲，不应该有调用和回调之分，因为所有代码最终均由操作系统调用（甚至更底层）。

.NET 中的回调主要是通过委托（Delegate）来实现的，委托是一种代理，专门负责调用方法（委托的详细信息在本书第五章有讲到）。

2.9 托管资源与非托管资源

其实这里的“托管”跟第一章中讲到的托管环境、托管代码或者托管时代中的“托管”意思一样。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，也就是说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 中托管资源主要指“对象在堆中的内存”等；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 不会管理这些资源，需要开发人员人工去控制。.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。

图 2-7 一个堆中对象使用的资源

图 2-7 中虚线框表示“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。一个对象在使用完毕后（进入不可达状态，并不是死亡，第四章会讲到区别），我们应该确保它使用的（如果使用了）非托管资源能够及时释放，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 Garbage Collector）会帮我们处理。.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 等等，它们内部封装了非托管资源，没有使用非托管资源的类型也有很多，比如 Console、EventArgs、ArrayList 等等。

怎么完美地处理一个对象使用的非托管资源，是一门相当重要而且必学的技术，后面第四章有详细提到。

> 注：现在普遍有一种错误的观点就是，将 FILESTREAM、SOCKET 这样的类型对象称为非托管资源，这个是错误的，只能说这些对象使用到了非托管资源。

2.10 框架（Frameworks）与库（Library）

框架和类库都是一系列可以被重用的代码集合。不同的是，框架算是不完整的应用程序，理论上，我们不用写任何代码，框架本身可以运行起来；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，我们需要给它各种装饰，在这个过程中，我们可以使用类库，因为类库可以为我们提供一些封装好了的功能。下图 2-8 为框架、程序（开发人员编写）以及类库三者之间的关系：

图 2-8 框架程序类库之间的关系

图 2-8 中的调用关系其实是双向的，画出的箭头只显示了主要调用关系，即框架调用开发人员代码，后者再选择性调用一些类库。

从上图 2-8 中我们可以看出，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。

```
//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();
    }
```

```
}
```

上面代码 Code 2-9 演示了从控制台程序（不使用框架开发）中获取用户输入的两个数据，然后输出两个数据之和，每个步骤的方法均由我们自己调用（NO.1. NO.2 以及 NO.3）。如果我们采用 Winform 程序（使用框架开发）实现，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;
    }
}
```

上面代码 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，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

这四个概念中最为熟悉的当然是“面向对象”，其它三个离我们有点遥远，平时接触不多。

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 特性，那么就说这种语言是基于对象的，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，我们称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，我们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。程序运行之后，分别加载这些独立的模块，各个模块负责完成自己的功能，我们称这种开发模式是基于组件的。基于组件开发模式中，除了二进制代码可以重用

外，还有另外一个优点，如果我们需要更新某一功能，或修复某一功能中的 bug，在不改变原有接口前提下，我们不用重新编译整个程序的源代码，而只需要重新编译某个组件源码即可。组件应该是语言独立的，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：基于组件开发中，我们只能重用已经编译完成的二进制代码，并不能从这个已经编译好的组件中读取其它信息，比如识别组件中的类型信息，派生出新的类型。面向组件指，在开发过程中，我们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，比如我们可以识别.NET 程序集中的类型信息，以此派生出新的类型。.NET 开发便是一种面向组件的开发模式。

> 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，我们需要知道，.NET 中的组件（程序集）并不包含传统意义的二进制代码。

2.12 接口

我们在阅读一些书籍或者网上浏览一些文章时，经常会碰到"接口"的概念，比如"一个类应该尽可能少的对外提供公共接口"、"我们应该先取得淘宝的支付接口权限"、"绘制图形时，我们需要调用系统的 DrawImage 接口"等等。那么，接口到底是什么？

其实我们碰到的这些"接口"概念跟它字面意思一样：对外提供的、可以完成某项具体功能的通道。比如我们电脑上的 USB 口，通过它，我们能够与电脑传输数据，还比如电视机的音量按钮，通过它，我们可以调节电视机喇叭发出声音的大小。接口是外界与系统（或模块）内部通讯的通道。

> 注："接口"的概念基于"封装"前提之上，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，接口用以下表示：

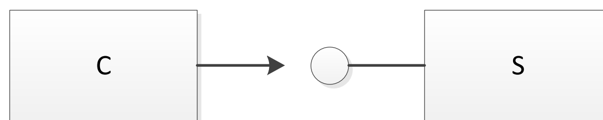


图 2-9 接口示意图

如上图 2-9 所示，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）、一个模块或者其它具有"封装"效果的单元个体。下图 2-10 显示某些场合存在的接口：

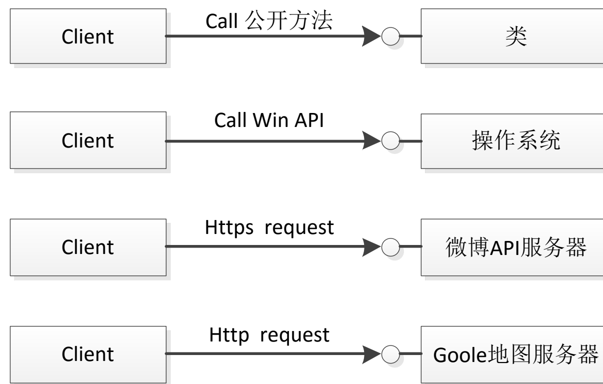


图 2-10 各种场合下的接口

如上图 2-10 显示了各种场合中的接口，可以看到，接口的概念不仅局限在代码层面。下表 2-1 显示了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	类的使用者不知道 People 类内部具体实现，但是可以与之通讯
2	操作系统	Win32 API，如 SetWindowText(hWnd,”text”); //设置某窗口标题	GUI 开发者	GUI 开发者不知道操作系统内部实现，但是可以与之通讯
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¶meter2=22	微博第三方应用开发者	微博第三方应用开发者不知道微博服务器内部实现，但是可以与之通讯
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不知道地图服务器内部实现，但是可以与之通讯

在.NET 编程中，还存在另外一种意义的“接口”，即我们使用 interface 关键字定义的接口类型，这种“接口”严格意义上讲跟我们刚才讨论的“接口”不能做相等比较。更准确来说，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，那么调用就不会成功。

注：有关“协议”，请参见下一节。

2.13 协议

协议，即约定、契约。两个（或两个以上）个体合作时需要共同遵守的准则，哪一方不遵守该准则，大部分时候将会导致合作失败，这个是现实生活中我们理解的“协议”。在计算机（编程）世界中，“协议”带来的效果同样如此。

计算机网络通信中，OSI（Open System Interconnection，开放系统互联模型）将网络分为7层，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，如下图2-11：

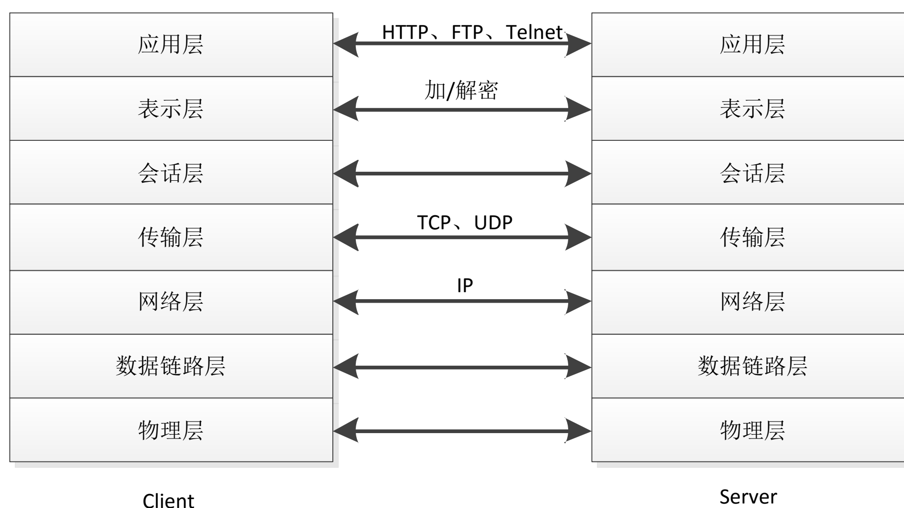


图 2-11 网络七层协议

如上图2-11所示，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，很多书籍中讲到了TCP/UDP协议结构，介绍了协议结构中每（几）个字节分别代表什么内容，数据发送方按照规定的格式填充该数据结构，数据接收方按照规定的格式去解析该数据结构，从而得到原始数据。不管TCP协议还是UDP协议，均属于传输层协议。对于某些高级语言（如C#、Java）开发者而言，接触这些协议的机会很少，更多时候，我们接触的是应用层协议，如HTTP协议、FTP协议等，除了这些主流、广为人知的协议外，我们自己在开发网络程序时，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，我们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（也就是协议）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；还比如在一些即时通信程序中，可能存在“文本消息”、“图片”、“表情”或者“文件”等一些数据类型，那么我们完全可以定义一个自己的应用层协议，见下图2-12：

图 2-12 自定义应用层协议

如上图2-12所示，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
}
```

如上代码 Code 2-11 所示，首先定义一个发送缓冲区（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）；对应的，接收方接收到数据后，按照预定义格式解析字节流。下图 2-13 显示顺序号为 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		byte
send_buffer[0x0000000c]	0x00	文本消息数据 "ABC"	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通讯中，由于数据是以"流"的形式传递的，前后发送的数据连接在一起，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，接收方可以按照该协议解析出一条完整的消息。详细参见本书中后续有关"网络编程"的第九章。

不仅网络通信需要"协议"的辅助，计算机世界中还有很多场合需要"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，下面表2-2显示了各种场合中的"协议"：

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	如各种编码规范：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，传统编译器将高级语言直接编译成与平台相关的机器码，机器码只能在指定平台上运行，CPU和编译器须遵守同一个规范
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）

到目前为止，我们讲到的"协议"都能很好地跟现实关联起来，或者说，它们都跟协议字面意思接近。其实在.NET程序开发过程中，也有一种"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，前面说过，代码中Client端与Server端需要交互，那么只有双方共同遵守某一约定，工作才能正常进行。这种协议在代码中具体体现在：

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。

具体示例代码见Code 2-12：

```
//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();
    }
}
```

如上代码 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）。双方都遵守了同一个协议，才能协调好工作。下图 2-14 显示了“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，详细参见本书第十二章。

2.14 本章回顾

本章共介绍了 13 个将在本书中遇到的概念（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！有些概念在其它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虐心绊脚石。

2.15 本章思考

1.下面代码 Code 2-13 中 MyContainer 类中的_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，可以在 MyContainer.Add()以及 MyContainer.GetAt()方法中加上锁（lock(object)）来解决该问题。

2.举例说明实际开发过程中遇见的框架和库有哪些。

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等。

第三章 编程之基础：数据类型

数据类型是编程的基础，每个程序员在使用一种平台开发程序时，首先得知道平台中有哪些数据类型，每种数据类型有哪些特点、又有着怎样的内存分配等。熟练掌握每种类型不仅有利于提高我们的开发效率，还能使我们开发出来的程序更加稳定、健全。.NET 中的数据类型共分为两种：引用类型和值类型，它们无论在内存分配还是行为表现上，均有着非常大的差别。

3.1 引用类型与值类型

关于对引用类型和值类型的定义，听得最多的是：值类型分配在线程栈中，而引用类型分配在堆中。这个定义并不准确（因为值类型也可以分配在堆中，而引用类型在某种场合也可以分配在栈中），或者说太抽象，它只是从内存分配的角度来区分值类型和引用类型，而对于内存分配，我们开发者是很难直观地去辨别。如果从代码角度来讲，.NET 中的值类型是指“派生自 System.ValueType 的类型”，而引用类型则指.NET 中排除值类型在外的所有其它类型。下图 3-1 显示了.NET 中的类型布局：

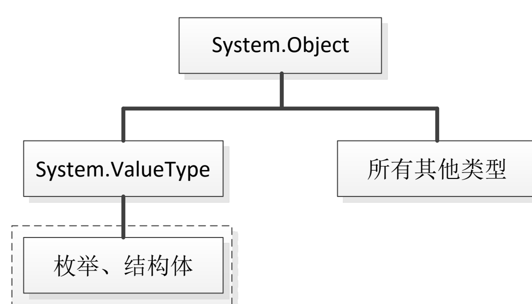


图 3-1 类型布局

如上图 3-1 所示，派生自 System.ValueType 的类型属于值类型（图中虚线部分，不包括 System.ValueType），所有其它类型均为引用类型（包括 System.Object、System.ValueType）。在以 System.Object 为根的庞大“继承树”中圈出一部分（图中虚线框），那么该小部分就属于“值类型”。

> 注：以上对值类型和引用类型的解释似乎有些难以理解，为什么“根”是引用类型，而某些“枝叶”却是值类型？这是因为.NET 内部对派生自 SYSTEM.VALUETYPE 的类型做了些“手脚”（这些对我们来讲是不可见的），使其跟其它类型（引用类型）具备不一样的特性。另外，.NET 中还有一些引用类型并不继承自 SYSTEM.OBJECT 类，比如使用 INTERFACE 关键字定义的接口，它根本不在“继承树”的范围之类，这样看来，像我们平时听见的“所有类型均派生自 SYSTEM.OBJECT 类型”的话似乎也不太准确，这些隐藏的不可告人的秘密都是.NET 内部做的一些处理，大部分并没有遵守主流规律。

通常值类型又分为两部分：

1）**简单值类型**：包括类似 int、bool、long 等.NET 内置类型，它们本质上也是一种结构体；
2）**复合值类型**：使用 Struct 关键字定义的结构体，如 System.Drawing.Point 等。复合值类型可以由简单值类型和引用类型组成，下面定义一个复合值类型：

```
//Code 3-1
struct MultipleValType
{
    int a;    //NO.1
    object c;    //NO.2
}
```

如上代码 Code 3-1 所示，MultipleValType 类型包含两个成员，一个简单值类型（NO.1 处），一个引用类型（NO.2 处）。

值类型均默认派生自 System.ValueType，又由于.NET 不允许多继承，因此我们既不可以在代码中显示定义一个派生自 System.ValueType 的结构体，同时也不可以让某个结构体继承自其它结构体。

引用类型和值类型各有自己的特性，这具体表现在内存分配、类型赋值（复制）、类型判等几个方面。

3.1.1 内存分配

本节开头就谈到，引用类型对象与值类型对象在内存中的存储方式不相同，使用 new 关键字创建的引用类型对象存储在（托管）堆中，而使用 new 关键字创建的值类型对象则分配在当前线程栈中。

> 注：堆和栈的具体概念请参见本书后面讲"对象生命期"的第四章。另外，使用类似"INT A = 0;"这种方式定义的简单值类型变量，跟使用 NEW 关键字"INT32 A = NEW INT32(0);"效果一样。

下面代码显示创建一个引用类型对象和一个值类型对象：

```
//Code 3-2
class Ref     //NO.1
{
    int a;
    Ref ref;
    public Ref(int a,Ref ref)
    {
        this.a = a;
        this.ref = ref;
    }
}
struct Val1    //NO.2
{
    int a;
    bool b;
    public Val1(int a,bool b)
    {
```

```
            this.a = a;
            this.b =b;
        }
}
struct Val2    //NO.3
{
        int a;
        Ref ref;
        public Val2(int a,Ref ref)
        {
            this.a = a;
            this.ref = ref;
        }
}
class Program
{
        static void Main()
        {
            Ref r = new Ref(0,new Ref(1,null));    //NO.4
            Val1 v1 = new Val1(2,true);    //NO.5
            Val2 v2 = new Val2(3,r);    //NO.6
        }
}
```

如上代码 Code 3-2 所示，先定义了一个引用类型 Ref（NO.1 处），它包含一个值类型和一个引用类型成员；然后定义了两个值类型（NO.2 和 NO.3 处），前者只包含两个简单值类型成员（int 和 bool 类型），后者包含一个简单值类型和一个引用类型成员；最后分别各自创建一个对象（NO.4、NO.5 以及 NO.6 处）。创建的三个对象在堆和栈中存储情况见下图 3-2：

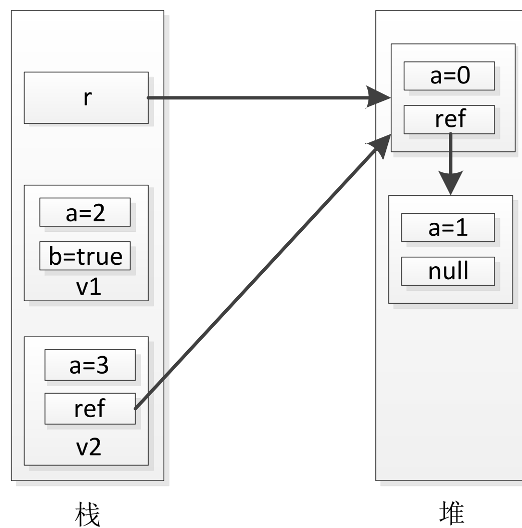


图 3-2 堆和栈中数据存储情况

如上图 3-2 所示，值类型对象 v1 和 v2 均存放在栈中，而引用类型对象均存放在堆中。

通常程序运行过程中，线程会读写各自对应的栈（因此有时候我们称“线程栈”），也就是说，“栈”才是程序进行读写数据的地方，那么程序怎么访问存放在堆中的数据（对象）呢？

这就需要在栈中保存一个对堆中对象的引用（索引），程序就可以通过该引用访问到存放在堆中的对象。

> 注：引用类型对象一般分为两部分：对象引用和对象实例，对象引用存放在栈中，程序使用该引用访问堆中的对象实例；对象实例存放在堆中，里面包含对象的数据内容，有关它们更详细介绍，请参见本书后面有关"对象生命期"的第四章。

3.1.2 字节序

我们知道，内存可以看作是一块具有连续编号的存储空间，编号有大有小，所以有高地址和低地址之分。如果以字节为单元进行编号，那么一块内存可以用下图 3-3 表示：

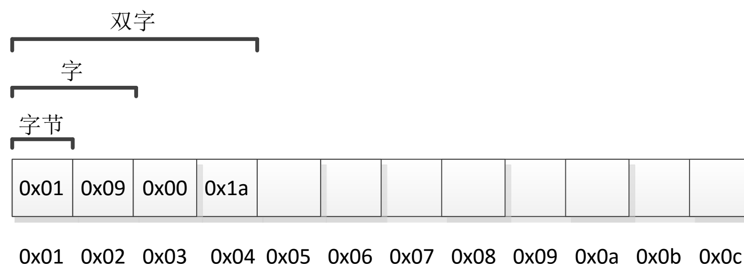


图 3-3 内存结构

如上图 3-3 所示，从左往右，地址编号依次增大，左侧称为“低地址”，右侧称为“高地址”。编号为 0x01 字节中存储数值为 0x01，编号为 0x02 字节中存储数值为 0x09，编号为 0x03 字节中存储数值为 0x00，编号为 0x04 字节中存储数值为 0x1a，每个字节中均可存放一个 0~255 之间的数值。那么这时候，如果我问你，图 3-3 中最左侧四个字节表示的一个 int 型整数为多少？你可能会这样去计算：0x01*2 的 24 次方+0x09*2 的 16 次方+0x00*2 的 8 次方+0x1a*2 的 0 次方，然后这样解释：高位字节在左边，低位字节在右边，将这样的一个二进制数转换成十进制数当然是这样计算。事实上，这种计算方法不一定正确，因为没有人告诉你高位字节一定在左边（低地址），而低位字节一定在右边（高地址）。

当占用超过一个字节的数值存放在内存中时，字节之间必然会有一个排列顺序，我们称之为“字节序”，这种顺序会因不同的硬件平台而不同。高位字节存放在低地址，而低位字节存放在高地址（如刚才那样），我们称之为“Big-Endian”；相反，高位字节存放在高地址，而低位字节存放在低地址，我们称之为“Little-Endian”。在使用高级语言编程的今天，我们大部分时间不用去在意“字节序”的差别，因为这些都有系统底层支撑模块帮我们判断完成。

.NET 中的值类型对象和引用类型对象在内存中同样遵循“字节序”的规律，如下面一段代码：

```
//Code 3-3
class Program
{
    static void Main()
    {
        int a = 0x1a09;
        int b = 0x2e22;
        int c = b;
    }
}
```

如上代码 Code 3-3 所示，变量 a、b、c 在栈中存储结构如下图 3-4：

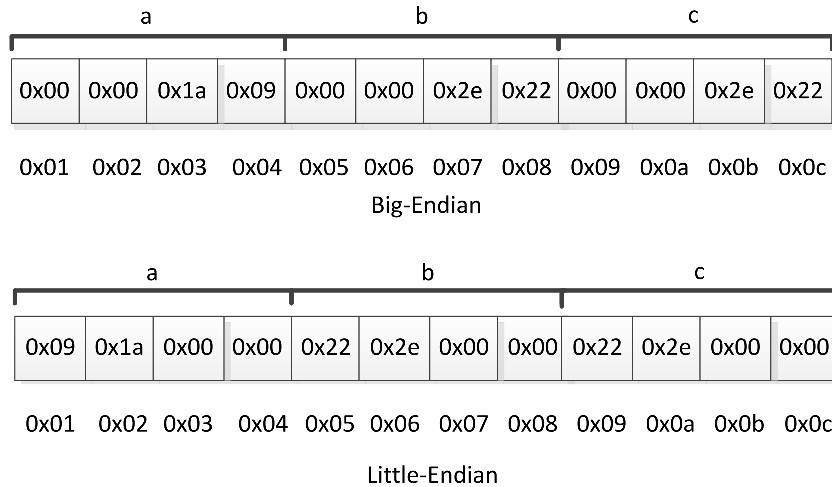


图 3-4 整型变量在栈中的存储结构

如上图 3-4 所示，图中右边为栈底（注意这里，通常情况下，栈底位于高地址，栈顶位于低地址）。依次将 c、b 和 a 压入栈，图中上部分为按“Big-Endian”的字节序存放数据，而图中下部分为按“Little-Endian”字节序存放数据。

3.1.3 装箱与拆箱

前面讲到，new 出来的值类型对象存放在栈中，new 出来的引用类型对象存放在堆中（栈中有引用指向堆中的实例）。如果我们把栈中的值类型转存到堆中，然后通过一个引用访问它，那么这种操作叫“装箱”；相反，如果我们把装箱后在堆中的值类型转存到栈中，那么就叫“拆箱”。下面代码 Code 3-4 表示装箱和拆箱操作：

```
//Code 3-4
class Program
{
    static void Main()
    {
        int a = 1;    //NO.1
        object b = a;    //NO.2
        int c = (int)b;    //NO.3
    }
}
```

如上代码 Code 3-4 所示，NO.1 定义一个整型变量 a，它存放在栈中，NO.2 处进行装箱操作，将栈中的 a 的值复制一份到堆中，并且使用 b 引用指向它，NO.3 处将装箱后堆中的值复制一份到栈中，整个过程栈和堆中的变化情况见下图 3-5：

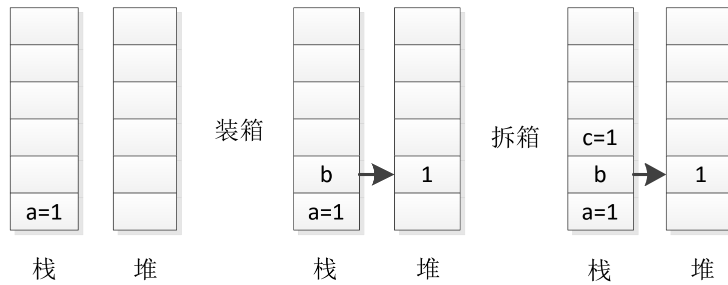


图 3-5 装/拆箱栈和堆中变化过程

如上图 3-5 所示，装箱时将栈中值复制到堆中，拆箱时再将堆中的值复制到栈中。

使用时间短、主要是为了存储数据的类型应该定义为值类型，存放在栈中，随着线程中方法的调用完成，栈中的数据会不停地自动清理出栈，再加上栈一般情况下容量都比较有限，因此，建议类型设计的时候，值类型不要过大，而把那种体积大、程序需要长时间使用的类型定义为引用类型，存放在堆中，交给 GC 统一管理。同时，拆装箱涉及到频繁的数据移动，影响程序性能，应尽量避免频繁的拆装箱操作发生。

> 注：图 3-5 中栈的存储是连续的，而堆中存储可以是随机的，具体原因参见本书后续有关"对象生命期"的第四章。

3.2 对象相等判断

在面向对象的世界里，随处充满着“对象”的影子，那么怎么去判断对象的相等性呢？所谓相等，指具有相同的组成、属性、表现行为等，两个对象相等并不一定要求相同。.NET 对象的相等性判断主要包括以下三个方面：

3.2.1 引用类型判等

引用类型分配在堆中，栈中只存放对堆中实例的一个引用，程序只能通过该引用才能访问到堆中的对象实例。对引用类型来讲，只有栈中的两个引用指向堆中的同一个实例时，才能说这两个对象相等（其实是同一个对象），其余任何时候，对象都不相等，就算两个对象中包含的数据一模一样。用图 3-6 表示为：

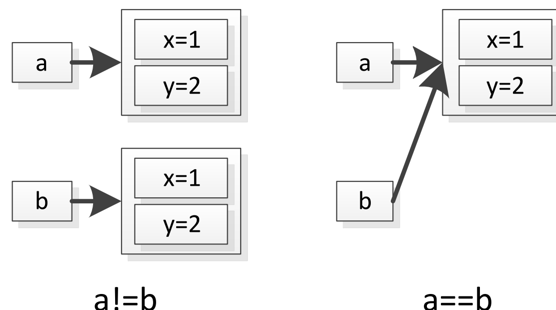


图 3-6 引用类型判等

如上图 3-6 所示，左边的 a 和 b 分别指向堆中不同的对象实例，虽然实例中包含相同的内容，但是它两不相等；右边的 a 和 b 指向堆中同一个实例，因此它们相等。

可以看出，对于引用类型来讲，判断两个对象是否相等很简单，直接判断两个对象引用是否指向堆中同一个实例，若是，则相等；其余任何情况都不相等。

注：熟悉 C/C++中指针的读者应该很清楚，两个不同的整型变量 A 和 B，虽然 A 的值和 B 的值相等（比如都为 1），但是它们两的地址肯定不相等（参见前面讲到的"字节序"）。.NET 中引用类型判等其实就是比较对象在堆中的地址，不同的对象地址肯定不相等（就算内容相等）。另外，.NET 中的 STRING 类型是一种特殊的引用类型，它不遵守引用类型的判等标准，只要两个 STRING 包含相同的字符串，那么就相等，STRING 类型判等更符合值类型的判等标准。

3.2.2 简单值类型判等

简单值类型包括.NET 内置类型，比如 int、bool、long 等，这一类的比较准则跟现实中所说到的"相等"概念相似，只要两者的值相等，那么两者就相等，见如下代码：

```
//Code 3-5
class Program
{
	static void Main()
	{
		int a = 10;
		int b = 11;
		int c = 10;
	}
}
```

如上代码 Code 3-5 所示，a 和 c 相等，与 b 不相等。为了与引用类型判等进行区分，见下图 3-7：

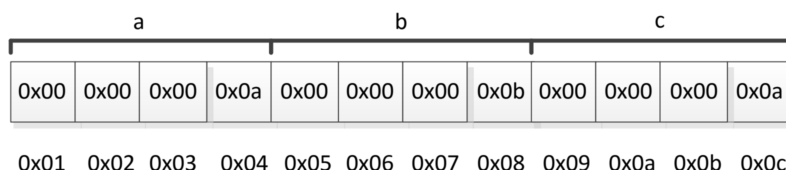


图 3-7 简单值类型在栈中的存储情况

如上图 3-7 所示，假设按照"Big-Endian"的字节序排列，右边是栈底，程序依次将 c、b 以及 a 压入栈。我们可以看到，如果比较 a 和 c 的内容，"a==c"成立；但是如果比较 a 和 c 的地址，很明显，a 的（起始）地址为 0x01，而 c 的（起始）地址为 0x09，它两的地址不相等。

简单值类型的比较只关注两者包含的内容，而不去关心两者的地址，只要它们的内容相等，那么它们就相等。复合值类型也是比较两者包含的内容，只是复合值类型可能包含多个成员，需要挨个成员进行一一比较，详见下一小节。

注：虽然笔者很不想在.NET 的书籍中提到有关指针（地址）的话题，但是为了说明"引用类型判等"的标准与"值类型判等"的标准有何区别，还是稍微提到了指针。我们可以很容易对比发现，引用类型判等其实就是比较对象在堆中的地址，而对象在堆中的地址就是由栈中的引用来表示的，地址不同，栈中引用的值肯定不相等，把栈中引用想象成一个存储堆中地址的变量，完全可以用简单值类型的判等标准去判断引用

是否相等。

3.2.3 复合值类型判等

前面讲过，复合值类型由简单值类型、引用类型组成。既然也是值类型的一种，那么它的判等标准和简单值类型一样，只要两个对象包含的内容依次相等，那么它们就相等。下面代码 Code 3-6 定义了两种复合值类型，一种只由简单值类型组成，一种由简单值类型和引用类型组成：

```
//Code 3-6
struct MultipleValType1    //NO.1
{
    int _a;
    int _b;
    public MultipleValType1(int a,int b)
    {
        _a = a;
        _b = b;
    }
}
struct MultipleValType2    //NO.2
{
    int _a;
    int[] _ref;
    public MultipleValType2(int a,int[] ref)
    {
        _a = a;
        _ref = ref;
    }
}
class Program
{
    static void Main()
    {
        MultipleValType1 mvt1 = new MultipleValType1(1,2);    //NO.3
        MultipleValType1 mvt2 = new MultipleValType1(1,2);    //NO.4
        // mvt1 equals mvt2    return true;
        MultipleValType2 mvt3 = new MultipleValType2(2,new int[]{1,2,3}); //NO.5
        MultipleValType2 mvt4 = new MultipleValType2(2,new int[]{1,2,3}); //NO.6
        //mvt3 equals mvt4    retturn false;
    }
}
```

如上代码 Code 3-6 所示，创建两个复合值类型，一个只包含简单值类型成员（NO.1 处），另一个包含简单值类型成员和引用类型成员（NO.2 处），最后创建了两对对象 mvt1 和 mvt2（NO.3 和 NO.4 处）、mvt3 和 mvt4（NO.5 和 NO.6 处），它们都存放在栈中。mvt1 和 mvt2 相等，因为它两包含相等的成员（_a 都等于 1，_b 都等于 2），相反，mvt3 和 mvt4 却不相等，虽然看起

来它两初始化是一样的(_a 都等于 1,_ref 都指向堆中一个 int[]数组,并且数组中的值也相等),原因很简单，按照前面关于“引用类型判等”的标准，mvt3 中的_ref 和 mvt4 中的_ref 根本就不是指向堆中同一个对象实例（即 mvt3._ref!=mvt4._ref）。为了更好地理解这其中的区别，请见下图 3-8：

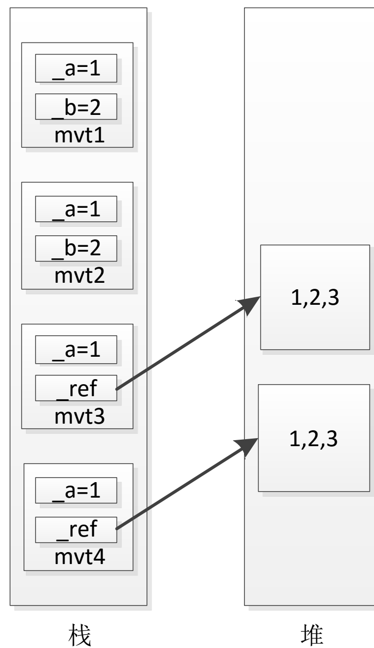


图 3-8 复合值类型内存分配

如上图 3-8 所示，创建的 4 个对象均存放在栈中，mvt1 和 mvt2 包含相等的成员，因此它两相等，但是 mvt3 和 mvt4 包含的引用类型成员_ref 并不相等，它们指向堆中不同的对象实例，因此 mvt3 和 mvt4 不相等。

对于值类型而言，判断对象是否相等需要按以下几个步骤：

（1）若是简单值类型，则直接比较两者内容，如 int、bool 等；

（2）若是复合值类型，则遍历对应成员：

1）若成员是简单值类型，则按照“简单值类型判等”的标准进行比较；

2）若成员是引用类型，则按照“引用类型判等”的标准进行比较；

3）若成员是复合值类型，则递归判断。

值类型判等是一个“递归”的过程，只要递归过程中有一次比较不相等，那么整个对象就不相等。详见下图 3-9：

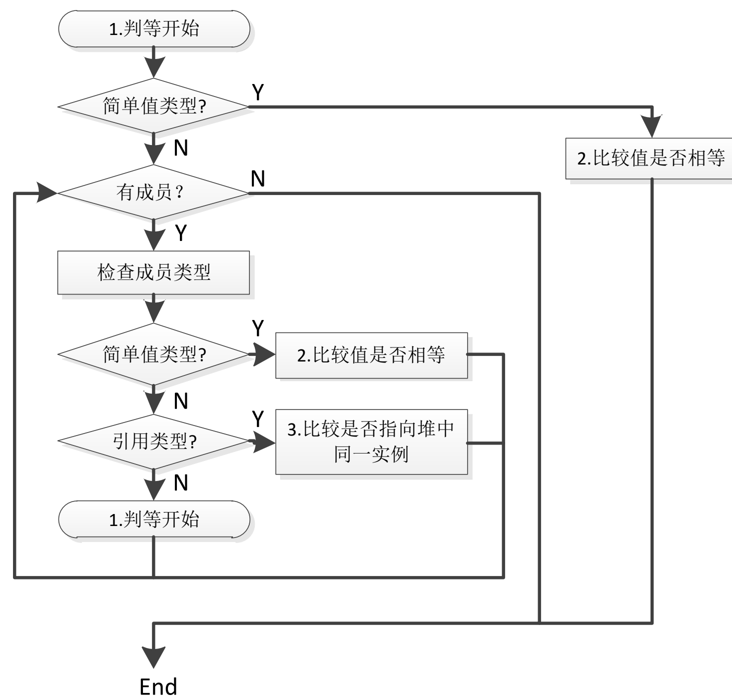


图 3-9 值类型判等流程

3.3 赋值与复制

3.3.1 引用类型赋值

通过赋值运算符“=”操作后，运算符两头的变量应该是相等的，这个是永远不会变的，不管在什么计算机语言中，也不管是值类型还是引用类型甚至其它类型。将.NET 中一个引用类型变量赋值给另外一个引用类型变量后，两个变量相等，既然相等，那么两个变量（栈中引用）肯定是指向堆中同一个实例，见下代码 Code 3-7:

```
//Code 3-7
class RefType    //NO.1
{
    int _a;
    bool _b;
    public RefType(int a,bool b)
    {
        _a = a;
        _b = b;
    }
}
class Program
{
    static void Main()
    {
        RefType r = new RefType(1,true);    //NO.2
```

```
        RefType r2 = r;      //NO.3
    }
}
```

如上代码 Code 3-7 所示，代码先创建了一个引用类型 RefType（NO.1 处），然后创建该类型的一个对象（NO.2 处），变量 r 指向该对象在堆中的实例，最后将变量 r 赋值给变量 r2（NO.3 处），该操作执行后，r 和 r2 相等，都指向堆中同一个实例。见下图 3-10：

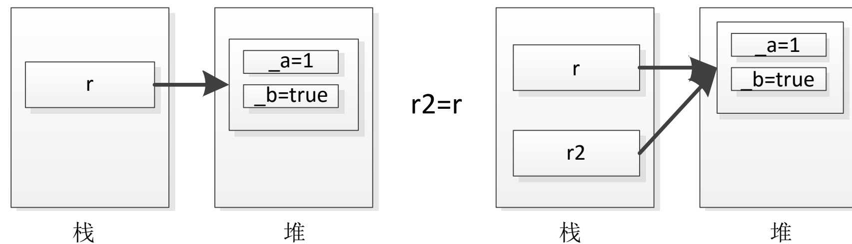


图 3-10 引用类型赋值

如上图 3-10 所示，赋值后，栈中两个变量指向堆中的同一实例。

可以看出，引用类型变量赋值后，对两个变量中任何一个操作，都会影响另外一个，这种情况会发生在引用类型“传参”过程中，详见 3.3.3 小节。

3.3.2 值类型赋值

既然赋值后，两个变量是相等的，既然相等，那么两个变量中包含的内容也应该一一相等。这个对于简单值类型来讲，很好理解，整型变量 a（值为 1）赋值给整型变量 b 后，a 和 b 的值相等（值都等于 1）；对于复合值类型来讲，赋值的过程就是成员的一一赋值，最后对应的成员一一相等，下面代码 Code 3-8 显示了复合值类型赋值情况：

```
//Code 3-8
struct MultipleValType1    //NO.1
{
    int _a;
    bool _b;
    public ValType(int a,bool b)
    {
        _a = a;
        _b = b;
    }
}
struct MultipleValType2    //NO.2
{
    int _a;
    int[] _ref;
    public MultipleValType(int a,int[] ref)
    {
        _a = a;
        _ref = ref;
    }
```

```
}
class Program
{
    static void Main()
    {
        MultipleValType1 mvt1 = new MultipleValType(1,true);    //NO.3
        MultipleValType1 mvt2 = mvt1;    //NO.4
        MultipleValType2 mvt3 = new MultipleValType2(1,new int[]{1,2,3}); //NO.5
        MultipleValType2 mvt4 = mvt3;    //NO.6
    }
}
```

如上代码 Code 3-8 所示，代码中创建了两种复合值类型 MultipleValType1 和 MultipleValType2，一种只由简单值类型组成（NO.1 处），另外一种由简单值类型和引用类型组成（NO.2 处），然后定义一个 MultipleValType1 类型的变量 mvt1（NO.3 处），将它赋值给变量 mvt2（NO.4 处），之后还定义了一个 MultipleValType2 类型的变量 mvt3（NO.5 处），将它赋值给变量 mvt4（NO.6 处）。赋值操作后，mvt1 和 mvt2 相等，mvt3 和 mvt4 相等，见下图 3-11：

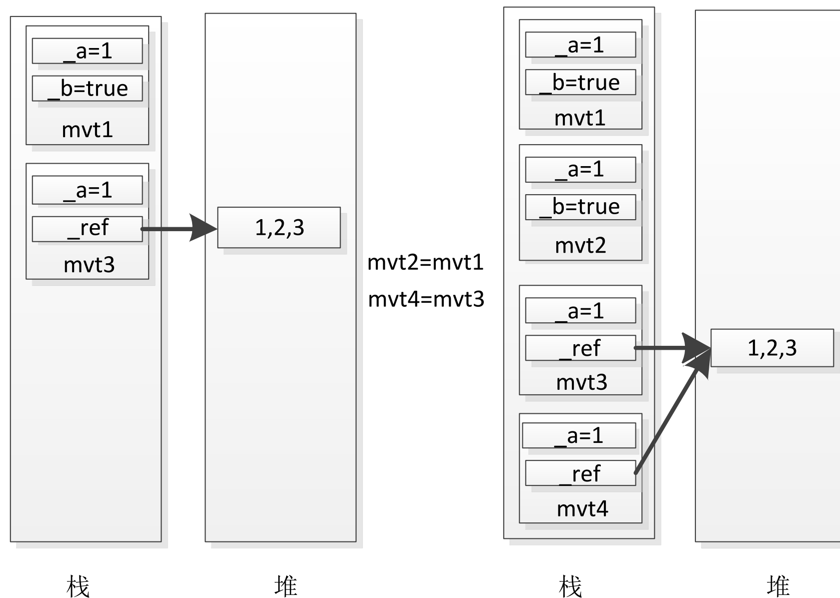


图 3-11 值类型赋值

如上图 3-11 所示（**为了方便绘图，注意图中栈里的数据并没按正确顺序存储**），值类型变量赋值时，逐个成员依次赋值，当成员中不包含引用类型时，赋值后两个对象完全独立（如 mvt1 和 mvt2），操作其中一个不会影响另外一个；当成员中包含引用类型时，由于引用类型成员指向堆中同一实例，赋值后两个对象不完全独立（如 mvt3 和 mvt4），操作其中一个有可能影响另外一个。

引用类型赋值和值类型赋值有很大的区别，前者传递的是对象在堆中的“地址”（索引、引用），其它都不变；后者会发生一次完整的“成员赋值”，逐个成员依次赋值。正因为这种赋值差异的存在，导致引用类型和值类型在“传参”过程中有明显差别，详见下一小节。

3.3.3 传参

所谓“传参”，其实就是赋值，将实参赋给形参。由于.NET 中的引用类型与值类型的赋值存在差异，所以在传递参数时，我们需要分清参数到底是什么类型，这个非常重要，因为引用

类型传参后，实参和形参指向的是堆中同一个实例，使用形参操作堆中的实例，直接能影响到实参指向的实例；而值类型传参后，形参是实参的一个副本，形参和实参包含有相同的内容，一般对形参进行的操作不会影响到实参，但也有例外，比如值类型中包含有引用类型的成员，不管在形参还是实参中，这个引用类型成员指向堆中同一个实例，通过该引用类型成员，形参照样可以影响到实参。

> 注："形参"指方法执行时，用于接收（存储）外部传值的临时变量，它在方法体内部可见；"实参"则是调用方在调用方法时，用来给方法传递参数的变量，它在方法体内部不可见。
>
> ```
> VOID FUNC(INT A)
> {
>     //这里的 A 是形参
>     //…
> }
> ```
>
> 调用方法代码:
>
> ```
> INT A = 1;
> FUNC(A);  //这里的 A 是实参，实参 A 将值赋给形参 A
> ```

下面代码演示值类型传参和引用类型传参：

```
//Code 3-9
class RefType    //NO.1
{
    public int a;
    public bool b;
    public RefType(int a,bool b)
    {
        this.a = a;
        this.b = b;
    }
}
struct ValType    //NO.2
{
    public int a;
    public int[] ref;
    public ValType(int a,int[] ref)
    {
        this.a = a;
        this.ref = ref;
```

```
        }
}
class Program
{
        static void Main()
        {
                RefType rt = new RefType(1,true);    //NO.3
                UpdateRef(rt);    //NO.4
                // rt.a == 2
                ValType vt = new ValType(1,new int[]{1,2,3});    //NO.5
                UpdateVal(vt);    //NO.6
                // vt.a == 1    and    vt.ref contains {2,2,3}
        }
        static void UpdateRef(RefType tmp)
        {
                tmp.a += 1;    //NO.7
        }
        static void UpdateVal(ValType tmp)
        {
                tmp.a += 1;    //NO.8
                tmp.ref[0] += 1;    //NO.9
        }
}
```

如上代码 Code 3-9 所示，代码中定义了一个引用类型 RefType（NO.1 处）和一个值类型 ValType（NO.2 处），然后分别创建一个变量 rt（NO.3 处）和 vt（NO.5 处），将这两个变量作为实参，分别传递给 UpdateRef()方法和 UpdateVal()方法，UpdateRef 方法改变形参 tmp 中成员 a 的值（加 1，NO.7 处），这时候，实参 rt 的成员 a **也会受到**影响（也会加 1）；UpdateVal 方法中改变形参 tmp 中成员 a 的值（加 1，NO.8 处），实参 vt 的成员 a **不会受到**影响，但是，UpdateVal 方法中使用形参 tmp 中引用类型成员 ref 去改变堆中数组中的值时（首元素加 1，NO.9 处），实参 vt 中的数组**也会受到**影响。详细过程参见下图 3-12：

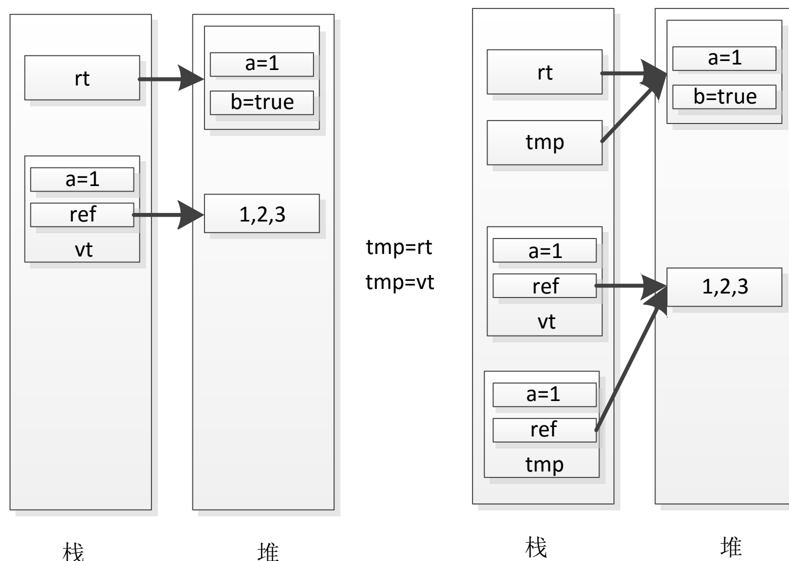


图 3-12 引用类型传参与值类型传参

如上图 3-12 所示（**为了方便绘图，注意图中栈里的数据并没按正确顺序存储**），左边为传参之前栈和堆中的情况，右边为传参之后栈和堆中的情况。可以看出，引用类型传参时，形参 tmp 可以完全操控实参 rt 指向的实例，而值类型传参时，形参 tmp 不能操控 vt 中值类型成员，但是仍然可以通过引用类型成员 ref 操控 vt 中 ref 指向的堆中实例。

引用类型传参和值类型传参各有优点，可以分场合使用不同的类型进行传参，有些时候我们在调用方法时，希望方法在操作形参的时候，同时影响到实参，也就是说希望方法的调用能够对方法体外部变量起到效果，那么我们可以使用引用类型传参；如果仅仅是为了给方法传递一些必需的数值，让方法能够正常运行，不需要方法的执行能够影响到外部变量，那么我们可以使用值类型（不包含引用类型成员）传参。

不管哪种类型传参，我们会发现，当有引用类型出现的时候（值类型中可以包含引用类型成员），方法体内总能通过形参去影响到实参，尤其是在调用一些别人开发好的类库，由于我们根本不知道类库中一些方法的具体实现，所以很难确定调用的方法会不会影响到我们传进去的实参，如果此时我们恰恰不希望方法能够影响到我们传递进去的实参，那该怎么做才能确保方法执行时（后），我们的实参不会受到影响呢？我们可以以实参为基础，复制出一个一模一样的副本，然后将该副本传递给方法，这样一来，方法体内部就不会影响到原来的对象。复制分两种，一种叫“浅复制”，另一种叫“深复制”，详见下一小节。

3.3.4 浅复制

所谓“浅复制”，类似值类型赋值，将源对象的成员一一进行拷贝，生成一个全新的对象。新对象与源对象包含相同的组成，值类型赋值就是一种“浅复制”。对于引用类型，怎么实现浅复制呢？下面代码 Code 3-10 演示引用类型怎样实现浅复制：

```
//Code 3-10
class RefType:IClonable    //NO.1
{
    int _a;
    int[] _ref;
    public RefType(int a,int[] ref)
    {
        _a = a;
        _ref = ref;
    }
    public Object Clone()
    {
        return new RefType(_a,_ref);    //NO.2
    }
}
class Program
{
    static void Main()
    {
        RefType rt = new RefType(1,new int[]{1,2,3});    //NO.3
        RefType rt2 = rt;    //NO.4
        RefType rt3 = rt.Clone();    //NO.5
```

```
    }
}
```

如上代码 Code 3-10 所示，代码定义了一个引用类型 RefType（NO.1 处），它包含一个值类型成员和一个引用类型成员。RefType 实现了 IConable 接口，该接口包含一个 Clone()方法，意为克隆出一个新对象，在实现 Clone()方法时，方法中调用了 RefType 的构造方法创建一个全新的 RefType 对象（NO.2 处），并将其返回。显而易见，新创建的副本与源对象包含相同的组成。接着，在客户端代码中，我们实例化一个 RefType 对象（NO.3 处），我们首先执行“赋值”操作，将引用 rt 赋给 rt2（NO.4 处），这时候 rt 和 rt2 指向了堆中同一个实例，紧接着，我们调用 rt 的 Clone 方法，将返回值赋给 rt3（NO.5 处），这时候 rt3 指向了堆中另外一个实例，见下图 3-13：

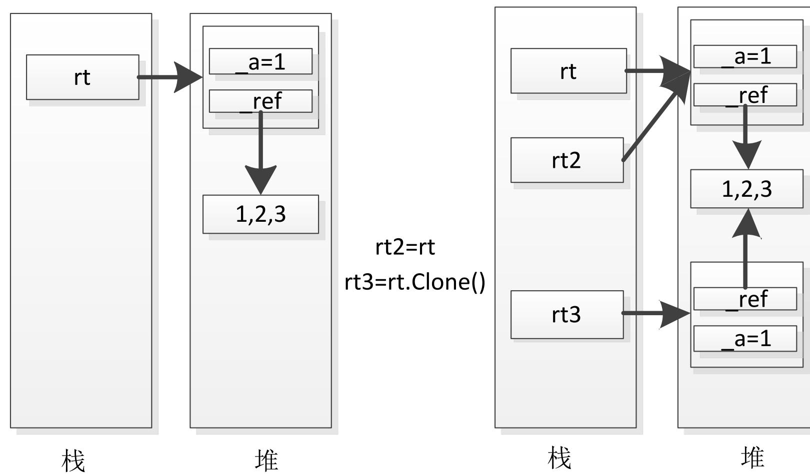


图 3-13 引用类型的浅复制

如上图 3-13 所示，很容易看到，引用类型的赋值和复制之间的区别，“rt2=rt;”这样的赋值语句，导致 rt 和 rt2 指向堆中的同一实例；而“rt3=rt.Clone();”这样的复制语句，不会导致 rt 和 rt3 指向堆中的同一实例，rt3 会指向一个副本，而该副本的内容与原对象中的内容一致，成员之间进行了一一拷贝。我们将图 3-13 与图 3-11 进行比较会发现，引用类型的浅复制与值类型赋值（**前面说过，值类型赋值就是一种浅复制**）非常相似，都是将成员进行逐一拷贝，产生了一个全新的对象，唯一的区别是：值类型的浅复制发生在栈中，而引用类型的浅复制发生在堆中。对象浅复制的过程见下图 3-14：

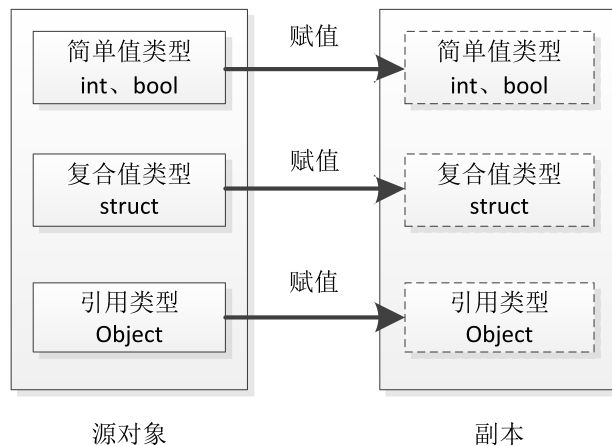


图 3-14 对象浅复制过程

如上图 3-14 所示，任何一个对象包含三种成员：简单值类型、复合值类型以及引用类型。浅

复制发生时，简单值类型成员直接一一赋值，引用类型成员直接一一赋值，复合值类型成员由于本身可以再包含其它的成员，所以需要递归赋值。

注：INT、BOOL 本质上也是 STRUCT 类型，本书中只是将这些.NET 内置类型归纳为“简单值类型”，简单值类型与复合值类型的定义并不是官方的。另外浅复制也称为“浅拷贝”。

不管是值类型还是引用类型，它们的浅复制都只是将对象内部成员进行一一赋值，然后产生一个新对象。如果成员是引用类型，那么新对象与源对象中该引用类型成员会指向堆中同一实例，换句话说，复制产生的副本与源对象仍有关联（见图 3-13 中，rt 与 rt3 中的_ref 指向同一个整型数组），操作其中一个很有可能影响到另外一个。要想复制出来的副本与源对象彻底断绝关联，那么需要将源对象成员（包括所有直接成员和间接成员）中所有的引用类型成员全部进行浅复制，如果引用类型成员本身还包括自己的引用类型成员，那么必须依次递归进行浅复制，由上向下递归进行浅复制的过程叫“深复制”，详细过程请参见下一小节。

注：对于值类型来讲，浅复制出来的副本与源对象是相等的，原因很简单，副本与源对象中包含的成员一一相等；但是对于引用类型来讲，浅复制出来的副本与源对象是不相等的，原因也很简单，副本和源对象在堆中占有不同的内存地址。

3.3.5 深复制

“浅复制”仅仅是将对象成员进行一一赋值，而无论成员是简单值类型、复合值类型还是引用类型，这就造成了一个问题：当一个对象包含有引用类型成员（包括直接成员和间接成员）时，浅复制出来的副本内部与源对象内部都包含一个指向堆中同一实例的引用。要想避免此问题，对象在进行浅复制时，如果存在引用类型成员，不能直接赋值，必须对该引用类型成员再进行浅复制，如果该引用类型成员本身还包含引用类型成员，必须依次递归进行浅复制。

注：直接成员指对象包含的第一级成员，间接成员指对象包含的一级成员（比如引用类型成员和复合值类型成员）本身包含的成员。

下图 3-15 显示了引用类型赋值、浅复制、深复制的区别：

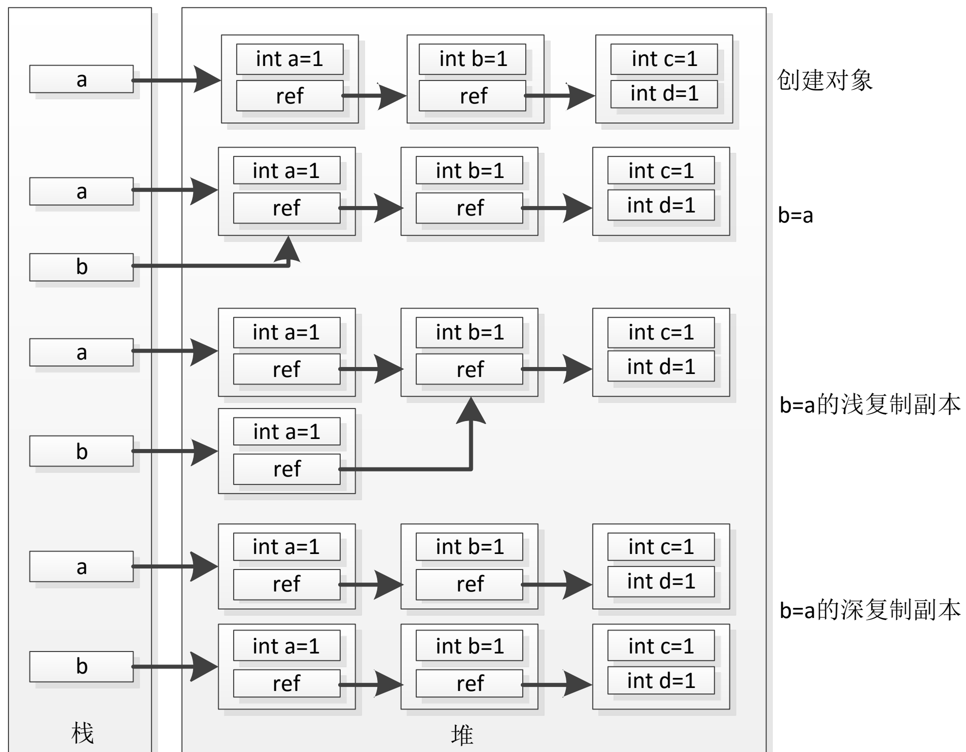


图 3-15 引用类型的深复制

如上图 3-15 所示，浅复制只是将对象的直接成员一一赋值，包括引用类型成员；而深复制指将对象的所有（直接或间接的）引用类型成员依次进行浅复制。值类型的赋值、浅复制、深复制的区别见下图 3-16：

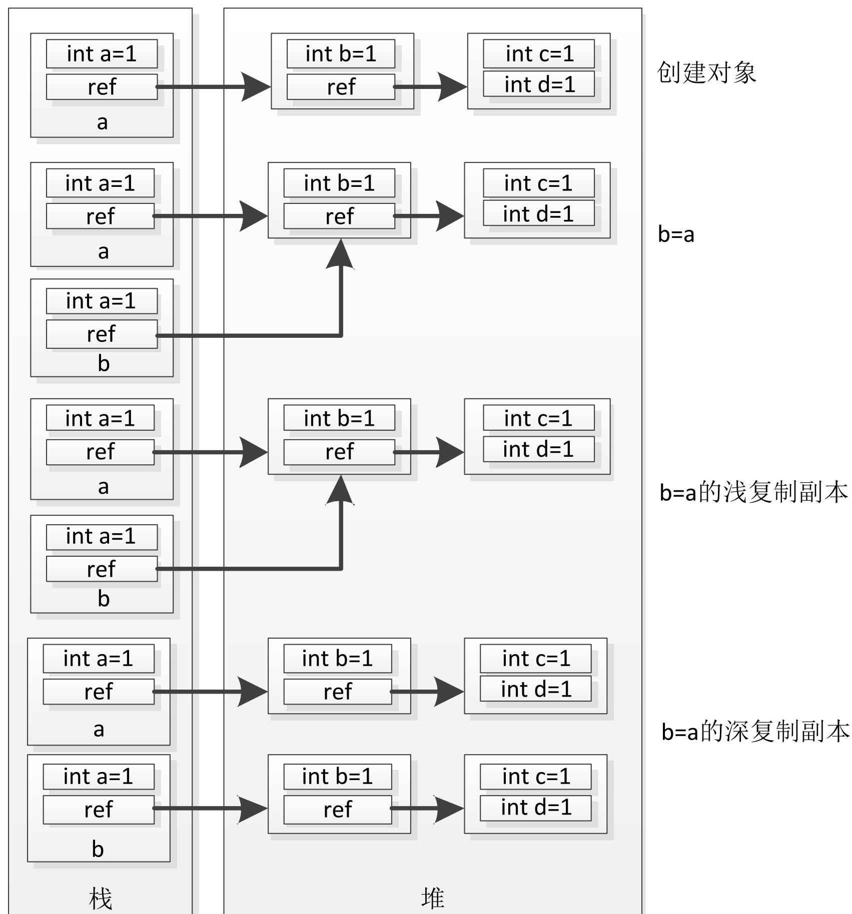


图 3-16 值类型的深复制

如上图 3-16 所示，值类型的赋值与浅复制的效果是一样的，值类型的深复制指将对象的所有（直接或间接的）引用类型成员依次进行浅复制。

显而易见，不管是值类型还是引用类型的深复制，均是一个递归的过程，它要求对象的所有引用类型成员均能够进行浅复制。下图 3-17 显示了对象深复制的过程：

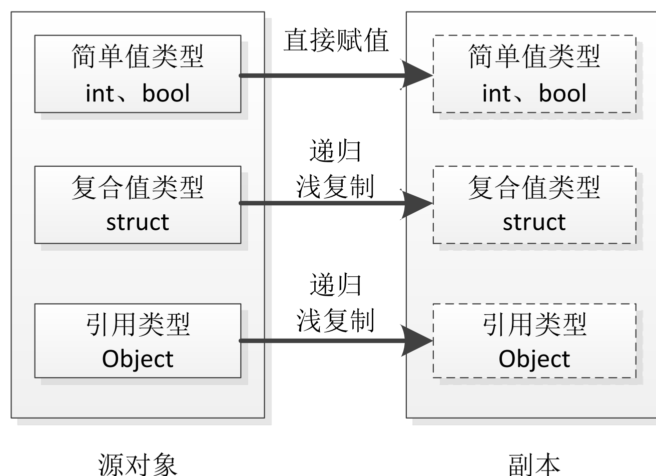


图 3-17 对象深复制过程

如上图 3-17 所示，对象深复制发生时，简单值类型成员直接一一赋值；复合值类型成员由于本身可以包含其它引用类型成员，所以它需要递归浅复制；引用类型成员不再直接一一赋值，而是需要进行浅复制，如果引用类型成员中又包含其它引用类型成员，那么依次递归浅复制。

下面代码 Code 3-11 显示了一个值类型的深复制：

```
//Code 3-11
struct ValType    //NO.1
{
    int _a;
    RefType _ref;
    public ValType(int a,RefType ref)
    {
        _a = a;
        _ref = ref;
    }
    public ValType Clone()
    {
        return new ValType(_a,_ref.Clone());    //NO.2
    }
}
class RefType    //NO.3
{
    int _b;
    bool _c;
    public RefType(int b,bool c)
    {
        _b = b;
        _c = c;
    }
    public RefType Clone()
    {
        return new RefType(_b,_c);    //NO.4
    }
}
class Program
{
    static void Main()
    {
        ValType vt = new ValType(1,new RefType(2,true));    //NO.5
        ValType vt2 = vt;      //NO.6
        ValType vt3 = vt.Clone();    //NO.7
    }
}
```

如上代码 Code 3-11 所示，先定义了一个复合值类型 ValType（NO.1 处），它包含一个简单值类型成员和一个引用类型成员，然后给该类型定义了一个 Clone()方法，注意该方法中，并不是像浅复制那样逐个成员一一赋值，而是当遇到引用类型成员时，对引用类型成员同样调用了 Clone 方法进行浅复制（_ref.Clone()）。在定义的引用类型 RefType 中（NO.3 处），同样给出了一个 Clone 方法，实现该类型的浅复制（NO.4 处）。最后在客户端代码中，先定义了一个值类型 vt（NO.5 处），然后将 vt 的赋值给 vt2（相当于浅复制，NO.6 处），将 vt 深复制出来的返回

值赋给 vt3（NO.7 处），最后浅复制出来的副本 vt2 和深复制出来的副本 vt3 的区别见下图 3-18：

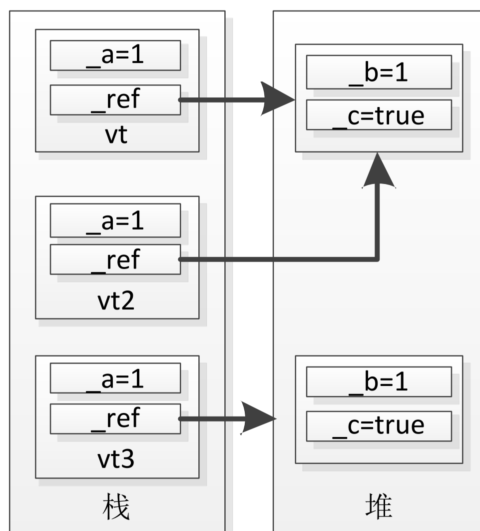


图 3-18 浅复制与深复制区别

如上图 3-18 所示，vt2 是 vt 浅复制出来的副本，由于源对象 vt 中包含有引用类型，很显然，副本 vt2 与 源对象 vt 仍有瓜葛；相反，vt3 是 vt 深复制出来的副本，我们可以看见，vt3 与 vt 毫无关联，对 vt3 的任何操作都不会影响到 vt。

Code 3-11 中的 ValType 类型中只包含一个引用类型成员，如果它还包含有其它的复合值类型成员，那么该成员必须也要提供浅复制的方法，另外引用类型 RefType 中仅仅包含两个简单值类型，如果还包含其它的引用类型或者复合值类型成员，那么这些成员都必须提供能够浅复制的方法，这样要求的原因很简单：对象深复制是一个递归的过程，每个引用类型、复合值类型成员都必须能够浅复制自己。正因为这种限制的存在，所以并不是每种类型都能够进行深复制操作，一种类型能够进行深复制操作的前提是，它所有的引用类型成员（包括直接和间接的）都必须提供深复制的方法。

注：.NET 中可以使用"序列化和反序列化"的技术实现对象的深复制，只要一个类型以及该类型中所有的成员类型都标示为"可序列化"，那么我们就可以先序列化该类型对象到字节流，然后再将字节流反序列化成源对象的副本，这样一来，源对象与副本之间没有任何关联，达到深复制的效果。

3.4 对象的不可改变性

3.4.1 不可改变性定义

一个类型对象创建后，它的状态不能再改变，直到它死亡，它的状态一直维持着跟它创建时一样。这时候称该对象具有不可改变性，称这样的类型为不可改变类型（Immutable Type）。

注：有的地方称这样的对象具备"常量性"。

不可改变对象在创建的时候，必须完全初始化，因为对象的状态之后再没机会发生改变。如果想要在不可改变对象的身上进行操作，试图想让它"变成"另一个全新的对象，多数时候，该

操作只会返回一个全新的对象，如 String 类型就是一种不可改变类型，对它的所有操作，String.Replace()、String.Trim()等方法都不会影响原有 String 对象，取而代之的是，这些方法都会返回一个全新的 String 对象。另外，.NET 中的委托类型也是不可改变的，我们对一个委托进行的所有操作，均会产生一个全新的委托，而不会改变委托本身，详见本书后面有关委托与事件的章节。下图 3-19 显示在 String 对象上调用 ToUpper()方法，不会影响原有对象：

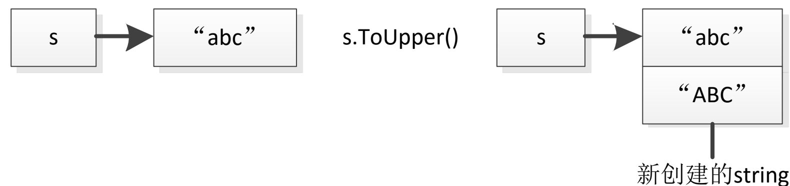


图 3-19 String 类型的不可改变特性

> 注：STRING 类型是一个特殊的类型，前面提到过，它虽然是引用类型，但是它不遵守引用类型判等的标准；另外，它还是不可改变类型，所有的操作均不会改变原有的 STRING 对象。

3.4.2 定义不可改变类型

定义一个不可改变类型时需要注意以下三点：

1）类型的构造方法一定要设计好，能够充分的初始化对象，因为对象创建后，无法再修改，构造方法是唯一能够修改对象状态的地方；

2）涉及到改变对象状态的方法均不能真正地改变对象本身，而都应该返回一个全新的对象，否则操作就没有实际意义；

3）类型所有的公开属性都应该是只读的，并且注意一些引用类型虽然是只读的，但是在类型外部还是可以通过只读的引用去改变堆中的实例，从而能够修改原对象的状态。

下面代码定义了一个不可改变类型：

```
//Code 3-12
class ImmutableType
{
    private int _val;
    private int[] _ref;
    public int Val
    {
        get      //NO.1
        {
            return _val;
        }
    }
    public int[] Ref
    {
        get   //NO.2
        {
            int[] b = new int[_ref.Length];
```

```
                for(int i=0;i<b.Length;++i)
                {
                    b[i] = _ref[i];
                }
                return b;
            }
        }
        public ImmutableType(int val,int[] ref)    //NO.3
        {
            _val = val;
            _ref = ref;
        }
        public ImmutableType UpdateVal(int val)
        {
            return new ImmutableType(this.Val + val,this.Ref);    //NO.4
        }
}
class Program
{
        static void Main()
        {
            ImmutableType a = new ImmutableType(1,new int[]{1,2,3});    //NO.5
            a = a.UpdateVal(2);    //NO.6
        }
}
```

如上代码 Code 3-12 所示，代码创建了一个不可改变类型 ImmutableType，它包含两个成员，一个值类型和一个引用类型，值类型属性是只读的（NO.1 处），引用类型属性也是只读的（NO.2 处），注意该引用类型属性不是简单的将引用对外公开，而是深度复制出来了一个副本，对外公开的是这个副本引用，外部不能使用该副本修改类内部的_ref 数组。类型的构造方法也很完善，能够初始化所有成员（NO.3 处），类型还包含一个“更新”_val 成员的方法 UpdateValue()，它的功能就是将_val 增加一个指定数，但是我们可以看见，操作并没有真正地影响到对象本身，而是新 new 出了另外一个全新对象（NO.4 处），将所有的效果都转嫁给新创建的对象。NO.5 处创建一个不可变对象，NO.6 处调用对象的 UpdateValue()方法，并将返回的新对象赋给 a 引用，NO.5 和 NO.6 处栈和堆中的变化见下图 3-20：

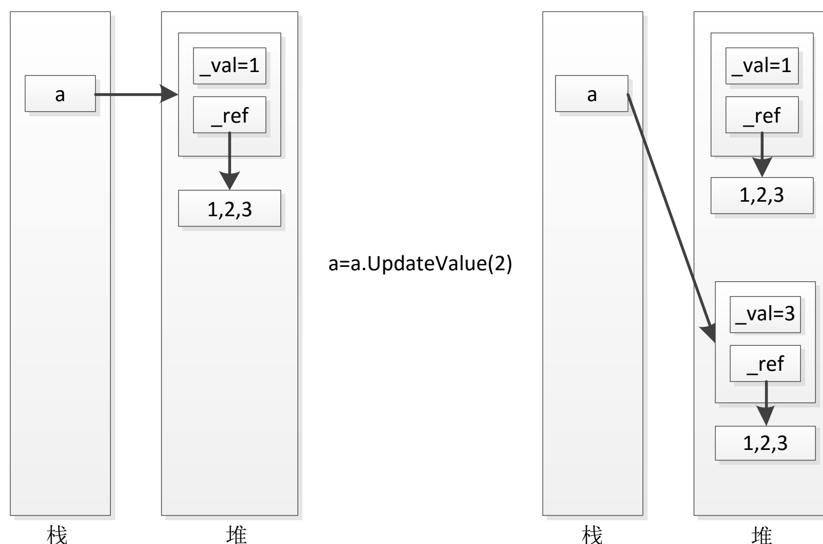


图 3-20 对象的不可改变性

如上图 3-20 所示，左边为执行“a=a.UpdateValue(2);”之前栈和堆中的情况，右边为执行之后栈和堆中的情况，可以看到，对 a 的操作实质上不会改变堆中的对象实例，只会产生一个全新的对象。

3.5 本章回顾

学习“数据类型”是能够学习好编程的前提，熟练掌握各种数据类型的特点有利于提高我们编程的效率、提升开发系统性能以及稳定性。本章介绍了.NET 中的两种数据类型：值类型和引用类型，并分别从对象的内存分配、对象判等、变量赋值以及对象复制等方面一一做出了介绍。章节最后还提到了不可改变类型，熟悉每种数据类型的不同表现差异是我们能够编写出好代码、好程序的第一步。

3.6 本章思考

1.“值类型对象分配在栈中，引用类型对象分配在堆中”这句话是否准确？为什么？

A：严格上讲，不太准确，值类型对象也可以被包含在引用类型对象内部，一起分配在堆中，因此不能一概而论。

2.“值类型对象的赋值就等于对象的浅复制”这句话是否正确？为什么？

A：正确。值类型对象赋值的过程就是浅复制的过程，依次将对象成员一一进行拷贝。

3.下面代码 Code 3-13 运行后会输出“true”，因此可以判断 String 是值类型，因为只有值类型判等时才比较两者所包含的内容是否一一相等，以上陈述是否正确？为什么？

```
//Code 3-13
```

```
class Program
{
	static void Main()
	{
		String a = new String("123");
		String b = new String("123");
		if(a == b)
		{
			Console.WriteLine("true");
		}
		else
		{
			Console.WriteLine("false");
		}
	}
}
```

A：错。String 类型是一个特殊的引用类型，它的判等不同于其它引用类型去比较对象引用是否指向堆中同一实例，而是和值类型判等一致，比较对象内容是否一一相等。除此之外，String 类型还是不可改变类型，对 String 对象的任何操作均不能改变该对象。

4.简要描述深复制与浅复制的区别。

A：对象进行浅复制时，只将对象的直接成员一一进行拷贝，当对象包含有引用类型成员时，源对象与副本之间有关联；对象进行深复制时，会将对象的所有成员（包括直接成员于间接成员）依次进行拷贝，不管对象是否包含引用类型成员，源对象与副本都无任何关联。

第四章 物以类聚：对象也有生命

对象的“生”到“死”一直是.NET 编程中的难点，如果没有正确地理解对象的生命周期，程序很容易就会出现“内存泄露”（Memory Leak）等异常，最终系统崩溃。很多人错误地认为调用了对象的 Dispose 方法之后，对象就“死亡”了，这些错误的观点往往会给系统带来致命的 BUG。本章将从对象生命周期、对象使用资源管理等方面来说明编程过程中我们应该注意哪些陷阱，从而最大程度减少代码异常发生的几率。

4.1 堆和栈

堆（Heap）和栈（Stack）是.NET 中存放数据的两个主要地点，它们并不是首先在.NET 中出现，C++中就有堆和栈的概念，只是 C++中的堆需要人工维护，而.NET 中的堆则由 CLR 管理。在.NET 中，我们也称堆为“托管堆”（Managed Heap），这里的“托管”与本书前面说到的托管代码中的“托管”意思一样，都指需要别人来辅助。

栈主要用来记录程序的运行过程，它有严格的存储和访问顺序，而堆主要存储程序运行期间产生的一些数据，几乎没有顺序的概念。堆跟栈的区别见下图 4-1：

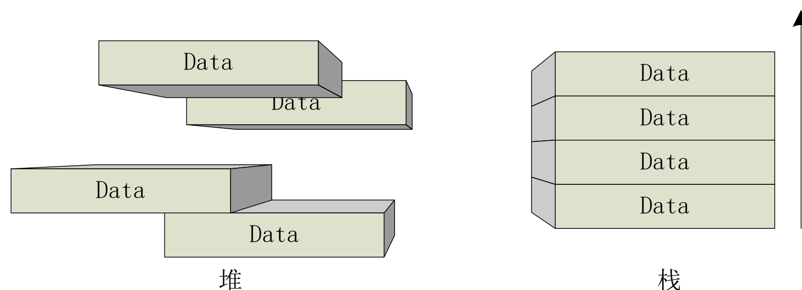


图 4-1 堆与栈存放数据的区别

如上图 4-1，堆中的数据可以随机存取，而栈中的数据必须要按照某一顺序存取。

栈主要存放程序运行期间的一些临时变量，包括值类型对象以及引用类型对象的引用部分，而堆中主要存放引用类型对象的实例部分。注意这里的引用类型对象应该包含两个部分：引用和实例，程序通过前者去访问后者。

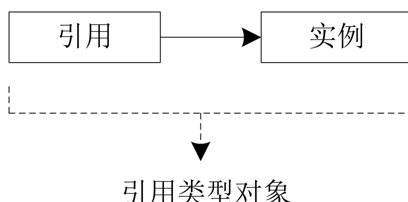


图 4-2 引用类型对象

我们说的一个引用类型对象其实包含两个部分，图 4-2 中的“引用”类似 C++中的指针，存放在栈中，而图中的“实例”则存放在堆中。C#语言中的值类型有类似 bool、byte、int 等等，而引用类型包括一些 class、interface 以及 delegate 等等。

//Code 4-1

```
class A
{
    //...
}
class Program
{
    static void Main()
    {
        int local1 = 0;
        A local2 = new A();
        bool local3 = true;
    }
}
```

上面代码 Code 4-1 中 int 和 bool 为值类型，所以 local1 和 local3 被分配在栈中，而 A 为引用类型（class），所以 A 类对象引用 local2 分配在栈中，A 类对象实例分配在堆中，如下图 4-3：

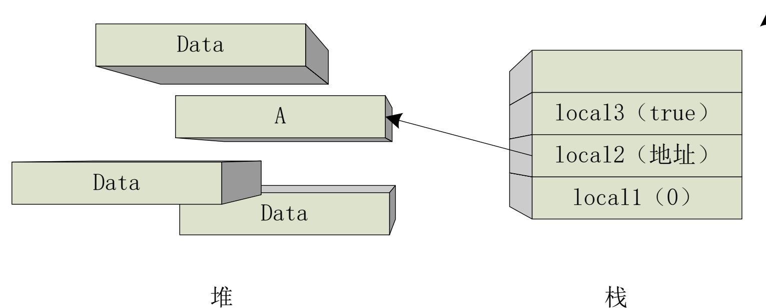


图 4-3 栈和堆中的数据存储

如上图 4-3，程序中如果需要访问 A 类对象时，必须通过它的引用 local2 去访问。

值类型对象与引用类型对象的引用如果属于另一个引用类型的成员，那么它们也是可以分配在堆中，

```
//Code 4-2
class A
{
    //...
}
class B
{
    //...
    A a;
    int aa;
    public B()
    {
        a = new A();
        aa = 1;
    }
    //...
}
```

```
class Program
{
	static void Main()
	{
		int local1 = 0;
		B local2 = new B();
		bool local3 = true;
	}
}
```

上面代码 Code 4-2 中，local1 和 local3 依旧分配在栈中，由于 a 和 aa 属于引用类型 B 的成员，所以它们会随 B 对象实例一起，分配在堆中，下图 4-4 为此时栈和堆中的数据分配情况：

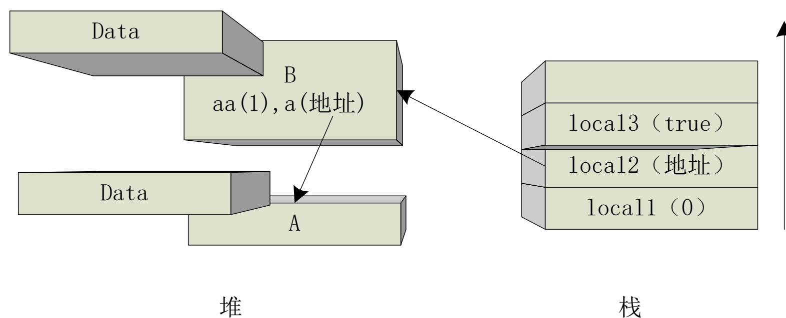


图 4-4 栈和堆中的数据分配情况

如图 4-4 中所示，虽然 aa 是值类型，a 是对象引用，但是它们都是分配在堆中。另外 A 类对象实例也是分配在堆中。

栈中的数据会随着程序的不停执行自动存入和移除，堆中的数据则由 CLR 管理，它们两者不是同步的，也就是说，可能栈中的一个对象引用已经从栈中移除，但是它指向的对象实例却还在堆中。

注：

[1]关于值类型与引用类型，请参见本书第三章介绍的"数据类型"。

[2]引用类型对象包括"对象引用"和"对象实例"两部分，这种叫法可能跟其它地方不一样，本书中统一称栈中的为"对象引用"，称堆中的为"对象实例"。另外，堆跟栈的本质都是一段内存块，本节以上几幅配图中，堆的"杂乱无章"只是为了说明堆中数据存储和访问可以是随机的，突出与栈的区别。

4.2 堆中对象的出生与死亡

栈中的对象由系统负责自动存入和移除，正常情况下，跟我们程序开发的关联并不大。本节主要讨论堆中对象的生命期。

4.2.1 引用和实例

上一节中结尾说到过，对于引用类型对象而言，栈中的引用和堆中的实例并不是同步的。栈中引用已经被移除，堆中的实例可能还存在，我们该怎么判断一个对象的生命周期呢？是依据栈里面的引用还是堆中的实例？

答案当然是堆里面的数据，因为 CLR 管理堆中对象内存时，是根据该对象的当前引用数目（包括栈和堆中），如果引用数目为零，那么 CLR 就会在某一个时间将该对象在堆中的内存回收。换句话说，堆中的对象实例存在的时间永远要比栈中的对象引用存在的时间长。下图 4-5 描述了对象实例与对象引用存在时间：

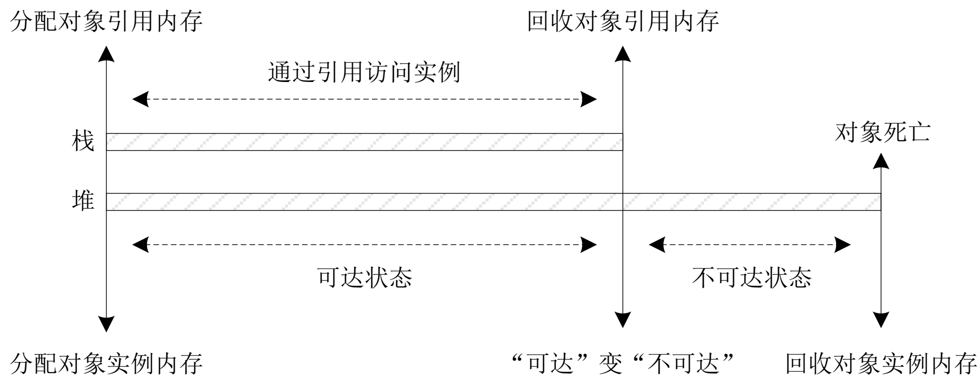


图 4-5 对象引用与对象实例的生存时间对比

图 4-5 中对象实例生存时间要比对象引用生存时间长，由于在代码中我们只能通过引用去访问实例，所以当引用从栈中移除后（图中“可达”变“不可达”处），对象虽然没有死亡，但是也基本没什么用处。

如果再也没有任何一个引用（包括栈和堆中）指向堆中的某个实例，那么 CLR（具体应该是 CLR 中的 GC，下同）就会在某一个时间点对该实例进行内存回收，CLR 的此举动便是第二章中提到的“管理对象托管资源”（堆中内存属于托管资源，由 CLR 管理）。需要注意的是，CLR 回收内存的时间点并不确定，并不是实例的引用数目一为零，CLR 马上就对该实例进行内存回收，因此图 4-5 中堆中对象“不可达”状态持续的时间不确定，有可能很短，也有可能很长。CLR 将对象实例的内存回收后，堆中可能会出现不连续块，这时 CLR 还有类似硬盘的“碎片整理”功能，它能合并堆中不连续部分。

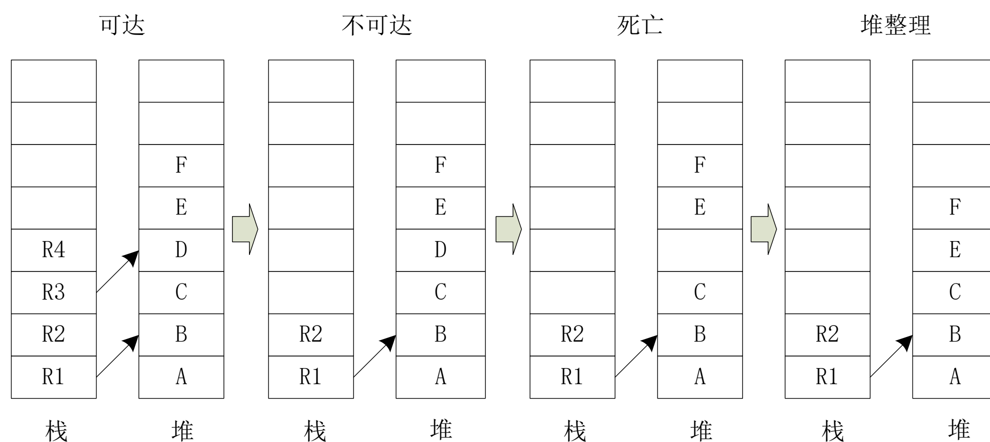


图 4-6 堆中对象的生到死

图 4-6 中对象引用 R3 存放在栈中，指向堆中的 D 对象实例；当栈中 R3 移除后（注意此时 R4 必须移除），堆中的 D 就处于“不可达”状态，D 也成为了 CLR 的回收目标；当 D 被 CLR 回收后，D 的位置就空出来，堆中出现不连续块；CLR 随后进行堆空间整理，合并不连续内存块，这便是对象一次从生到死的全过程。注意上图只是为了说明堆中 D 的生命周期，所以堆中其

它对象实例的引用情况没有完整绘制出来。另外，如果堆中还有其它的引用指向 D，就算 R3 从栈中移除，D 还是不能成为 CLR 的回收目标，如下图 4-7：

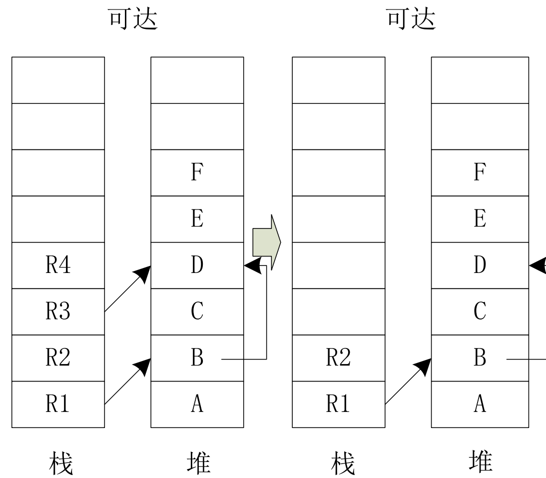


图 4-7 栈中的引用与堆中的引用同时存在

图 4-7 中 R3 和 B 中的某一个成员同时指向 D，就算 R3 从栈中移除，D 还是处于“可达”状态，不会成为 CLR 的回收目标。

很明显，我们程序中如果要操作堆中的某个对象实例，只能在该对象实例处于“可达”状态时，只有这个时候对象引用还在，其它任何时候都不行。换句话说，我们能够控制堆中对象实例的时间是有限制的。

4.2.2 析构方法

CLR 还有另外一个功能，它在准备回收一个“不可达”对象实例在堆中的内存之前，会调用这个对象实例的析构方法，调用时间点如下图 4-8：

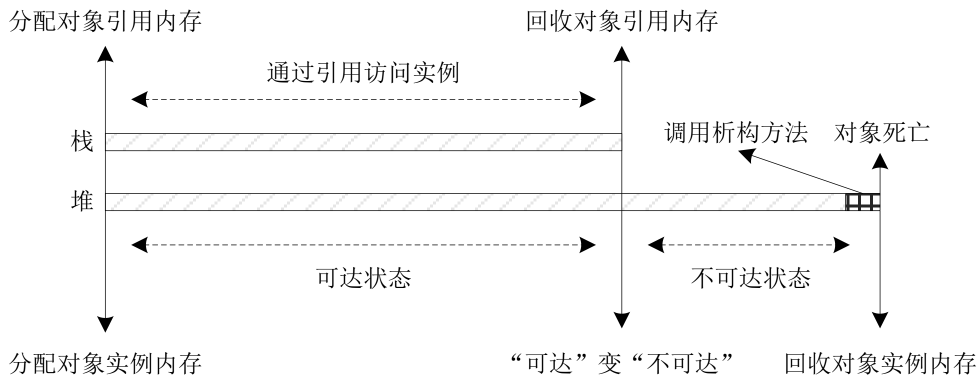


图 4-8 CLR 调用析构方法时间点

图 4-8 中 CLR 调用不可达对象的析构方法是在它回收内存之前，换句话说，对象死亡前一刻，CLR 还会调用它的析构方法，如此一来，我们真正能够操作堆中对象实例的机会又多了一处，之前说到的当对象实例处于“可达”状态时，我们可以通过对象引用访问它，现在，当对象处于“不可达”状态，我们还可以在析构方法中编写代码操作它。由于 CLR 回收内存的时间点不确定，所以它调用析构方法的时间点也不确定。“不可达”对象的析构方法迟早要被调用，但是调用时间我们不可控。

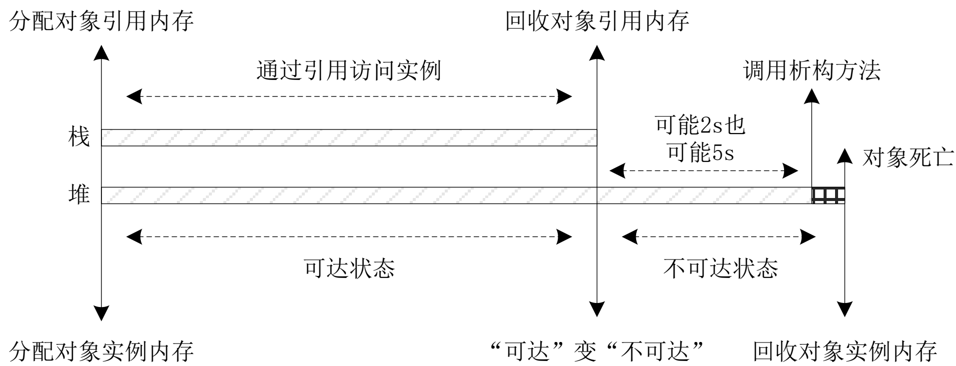


图 4-9 CLR 调用析构方法时间不确定

图 4-9 中显示 CLR 调用析构方法的时间点不确定。

虽然 CLR 调用析构方法的时间不确定，但是还是为我们提供了一个额外操作堆中对象实例的机会。下图 4-10 中网格部分表示我们可以在代码中操作堆中对象实例的时间段：

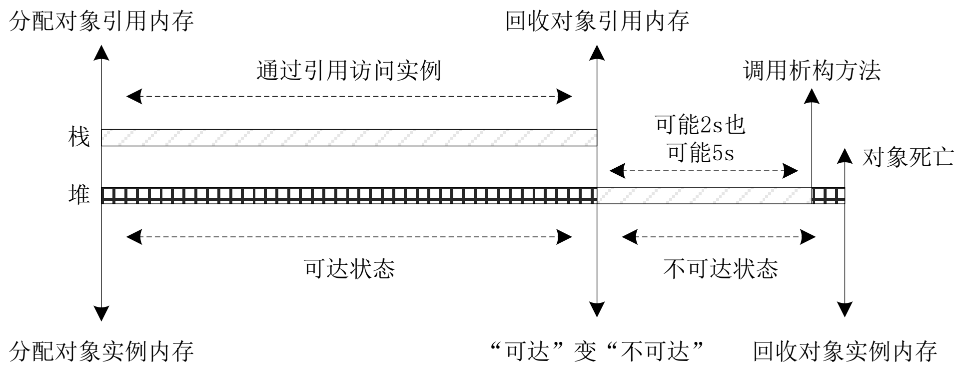


图 4-10 可操作堆中对象的时间段

由于 CLR 是根据“可达”还是“不可达”来判断是否应该回收对象实例的内存，因此，如果我们在析构方法中又将某个引用指向了对象实例，那么等析构方法执行完毕后，这个对象实例又从“不可达”状态变成了“可达”状态，CLR 会改变原来的计划，不再把该对象实例当成回收的目标，这个就是所谓的“重生”。“重生”是指堆中对象实例在将要被 CLR 回收前一刻，状态由“不可达”变成了“可达”。下图 4-11 显示了一个堆中对象实例的重生过程：

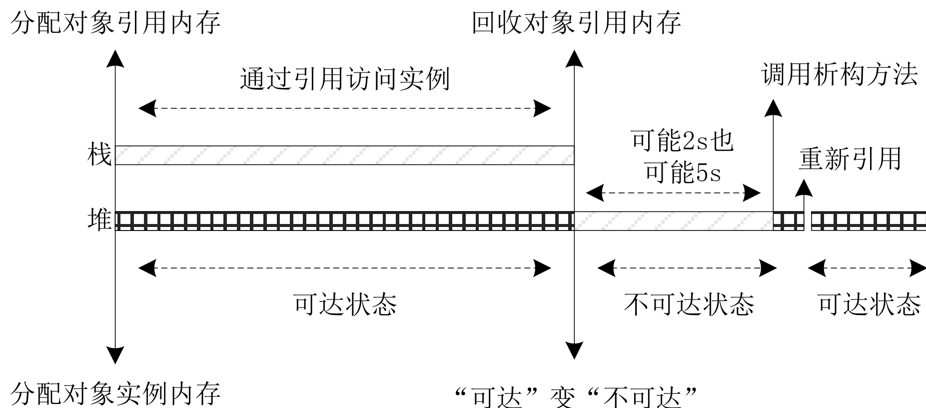


图 4-11 对象重生

理论上，一个对象可以无限次重生，但是为了避免编程的复杂性，“对象重生”在编程过

程中是不提倡的，也就是说，我们应该谨慎编写析构方法中的代码，不提倡在析构方法中再次引用堆中对象实例。

> 注：后面我们可以知道，正常情况下，析构方法除了用来释放自己使用过的非托管资源之外，其余什么都不应该负责。

4.2.3 正确使用对象

实际编程过程中，引用类型对象使用频率居高，导致堆中对象实例数量巨大，如果不正确地使用对象，很容易造成堆中对象实例占用内存不能及时被 CLR 回收，引起内存不足。编程中我们应该遵循以下两条规则：

（1）能使用局部变量尽量使用局部变量，也就是将引用类型对象定义成方法执行过程中的临时变量，不要把它定义成一个类型的成员变量（全局变量）；

局部变量存储在栈中，方法执行完毕后，会直接从栈中移除。如果把一个引用类型对象定义成局部变量，方法执行完毕后，对象引用从栈中移除，堆中的对象实例很快就会由“可达”状态变为“不可达”状态，从而成为 CLR 的回收目标。

```
//Code 4-3
class A
{
    //...
}
class B
{
    A _a;
    public B()
    {
        _a = new A();
    }
    //...
}
class C
{
    B _b;
    A _a;
    public C()
    {
        _a = new A();
    }
    public void DoSomething()
    {
        _b = new B();
        //deal with _b

        //or use local member
        //B b = new B();      NO.3
```

```
            //deal with b
        }
        //...
}
class Program
{
        //...
        static void Main()
        {
            int local1 = 0;
            C c = new C();
            c.DoSomething();    //NO.1
            int local2 = 1;    //NO.2
            // do something else
            //END
        }
}
```

上述代码 Code 4-3 中，C 的 DoSomething 方法中默认使用的是成员变量（全局变量）_b，当 DoSomething 方法返回之后，_b 指向的对象实例依旧处于“可达”状态。执行 DoSomething 方法前和执行 DoSomething 方法之后，栈和堆中的情况分别如下图 4-12：

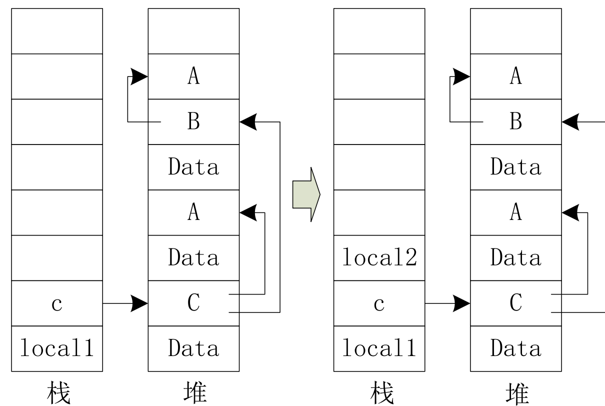


图 4-12 栈和堆中的数据分配

上图 4-12 左边为执行 c.DoSomething 方法时，也就是代码中 NO.1 处，栈跟堆中的数据情况；图中右边为 c.DoSomething 方法执行完毕返回后，也就是代码中 NO.2 处，栈跟堆中的数据情况。可以看出方法执行前后，堆中的 B 对象实例都是处于“可达”状态，根本原因便是指向 B 对象实例的引用是 C 的成员变量，只有 C 被 CLR 回收了之后，B 才由“可达”状态变为“不可达”状态。如果我们将 DoSomething 方法中的_b 改为局部变量（代码中注释部分 NO.3 处），情况则完全不一样：

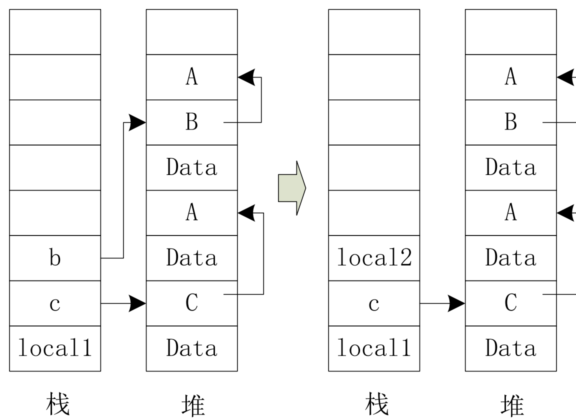


图 4-13 栈跟堆中的数据分配

上图 4-13 左边为执行 c.DoSomething 方法时，也就是代码中 NO.1 处，栈跟堆中的数据情况，可以看出，指向 B 对象实例的引用 b 保存在栈中；图中右边为 c.DoSomething 方法执行完毕返回后，也就是代码中 NO.2 处，栈跟堆中的数据情况，方法执行完毕返回后，栈中的 b 被移除，B 对象实例处于“不可达”状态，立刻成为 CLR 回收的目标。

（2）谨慎使用对象的析构方法。析构方法由 CLR 调用，不受我们程序控制，而且容易造成对象重生，除了下一节中介绍的管理对象非托管资源外，析构方法几乎不能用作其它用途。

4.3 管理非托管资源

非托管资源和托管资源一样，都是对象在使用过程中需要的必备条件。在.NET 中，对象使用到的托管资源主要指它占用堆中的内存，而非托管资源则指它使用到的 CLR 之外的资源（详见第二章关于托管资源和非托管资源的介绍）。非托管资源不受 CLR 管理，因此管理好对象使用的非托管资源是每个程序开发人员的职责。

当一个对象不再使用时，我们就应该将它使用的非托管资源释放掉，归还给系统，不然等到 CLR 将它在堆中的内存回收之后，这些非托管资源只能等到整个应用程序运行结束之后才能归还给系统。那么什么时候是我们释放对象非托管资源的最佳时机呢？

4.3.1 释放非托管资源的最佳时间

前面讲到过，我们能够操作堆中对象实例的机会有两个，一个是该对象实例处于“可达”状态时，即有对象引用指向它；第二个是在析构方法中。因此，我们可以在这两处释放对象的非托管资源。

由于析构方法调用时间不确定，所以我们最好不要完全依赖于析构方法，也就是说，只要我们不再使用某个对象，就应该在程序中马上释放掉它的非托管资源。为了避免忘记此操作而导致的非托管资源泄露，我们可以在析构方法中同样也写好释放非托管资源的代码（作为释放非托管资源的备选方案）。

```
//Code 4-4
class A
{
    //…
    public A()
    {
```

```
            //…
        }
        public void DoSomething()
        {
            //do something here
        }
        public void ReleaseUnManagedResource()
        {
            DoRelease();
            GC.SuppressFinalize(this);    //NO.1
        }
        private void DoRelease()
        {
            //release unmanaged resource here
        }
        ~A()
        {
            DoRelease();
        }
}
class Program
{
        static void Main()
        {
            A a = new A();
            a.DoSomething();    //NO.2
            a.ReleaseUnManagedResource();
        }
}
```

代码 Code 4-4 中 A 类型使用了非托管资源，提供了一个公开 ReleaseUnmanagedResource 方法，程序中使用完 A 类型对象后，立刻调用 ReleaseUnmanagedResource 方法释放它的非托管资源，同时，为了防止程序中没有调用 ReleaseUnmanagedResource 方法而导致的非托管资源泄露，我们在析构方法中调用了 DoRelease 方法。注意 GC.SuppressFinalize 方法（NO.1 处），它请求 CLR 不要再调用本对象的析构方法，原因很简单，既然非托管资源已经释放完成，那么 CLR 就没必要再继续调用析构方法。

> 注：CLR 调用对象的析构方法是一个复杂的过程，需要消耗非常大的性能，这也是尽量避免在析构方法中释放非托管资源的一个重要原因，最好是彻底地不调用析构方法。

如果调用 a.DoSomething（NO.2 处）抛出异常，那么后面的 a.ReleaseUnManagedResource 就不能执行，因此可以改进代码：

```
//Code 4-5
class Program
{
```

```
    static void Main()
    {
        A a = new A();
        try
        {
            a.DoSomething();
        }
        finally
        {
            a.ReleaseUnManagedResource();
        }
    }
}
```

将对 a 对象的操作放入 try/finally 块中，确保 a.ReleaseUnManagedResource 一定执行。

4.3.2 继承与组合中的非托管资源

面向对象编程（OOP）中有两种扩展类型的方法，一种是继承，另外一种便是组合。二者都可以以原有的类型为基础创建一个新的类型，这就产生了一个问题，如果是继承，派生类中使用了非托管资源，基类中也使用了非托管资源，这两种怎么统一管理？如果是组合，类型本身使用了非托管资源，类型中的成员对象也使用了非托管资源，这两种又怎么统一管理？如果继承与组合两者结合起来，又该怎么去管理它们的非托管资源呢？

在继承模式中，我们可以将释放非托管资源的方法定义为虚方法，派生类中只需要重写该虚方法，在方法里面添加释放派生类的非托管资源代码，再调用基类中释放非托管资源的方法即可。上一小节中类型 A 的代码改为：

```
//Code 4-6
class ABase
{
    //...
    public ABase()
    {
        //...
    }
    public void ReleaseUnManagedResource()
    {
        DoRelease();
        GC.SuppressFinalize(this);    //NO.1
    }
    protected virtual void DoRelease()
    {
        //release ABase's unmanaged resource here
    }
    ~ABase()
    {
        DoRelease();
    }
```

```
}
class A:Abase
{
    public A()
    {
        //…
    }
    protected override void DoRelease()
    {
        // release A's unmanaged resource here
        base.DoRelease();    //NO.2
    }
}
```

代码 Code 4-6 中 Abase 和 A 类型都使用到了非托管资源，A 类型重写了父类 Abase 的 DoRelease 虚方法，在其中释放 A 类型的非托管资源，然后再调用父类的 DoRelease 方法去释放父类的非托管资源（NO.2 处）。

> 注：虚方法 DORELEASE 必须声明为 PROTECTED，因为派生类需要调用基类的该方法。

基类和派生类非托管资源关系如下图 4-14：

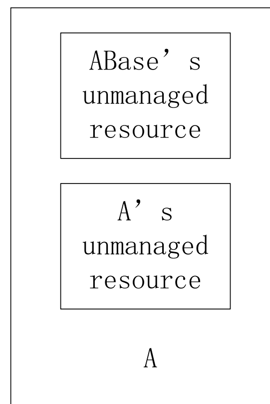


图 4-14 基类与派生类非托管资源关系

在组合模式中，一个类型可能有许多成员对象，这些成员对象也可能使用到了非托管资源。如果该类型对象释放非托管资源，那么其成员对象也应该释放它们各自的非托管资源，因为它们是一个整体，

```
//Code 4-7
class A
{
    //…
    public A()
    {
        //…
    }
    public void DoSomething()
    {
```

```
        //do something here
    }
    public void ReleaseUnManagedResource()
    {
        DoRelease();
        GC.SuppressFinalize(this);    //NO.1
    }
    private void DoRelease()
    {
        //release A's unmanaged resource here
    }
    ~A()
    {
        DoRelease();
    }
}
class B
{
    //…
    A _a1;
    A _a2;
    public B()
    {
        //…
        _a1 = new A();
        _a2 = new A();
    }
    public void DoSomething()
    {
        //do something here
    }
    public void ReleaseUnManagedResource()
    {
        DoRelease();
        GC.SuppressFinalize(this);    //NO.2
    }
    private void DoRelease()
    {
        //release B's unmanaged resource here
        //then release children unmanaged resource
        _a1.ReleaseUnManagedResource();      //NO.3
        _a2.ReleaseUnManagedResource();      //NO.4
    }
    ~B()
    {
        DoRelease();
```

```
    }
}
```

代码 Code 4-7 中 B 类型中包含两个 A 类型成员_a1 和_a2，_a1 和_a2 都需要释放非托管资源，由于它们两个跟 B 类型是一个整体，所以在 B 类型释放非托管资源的时候，我们也应该编写释放_a1 和_a2 的非托管资源代码（NO.3 和 NO.4 处）。

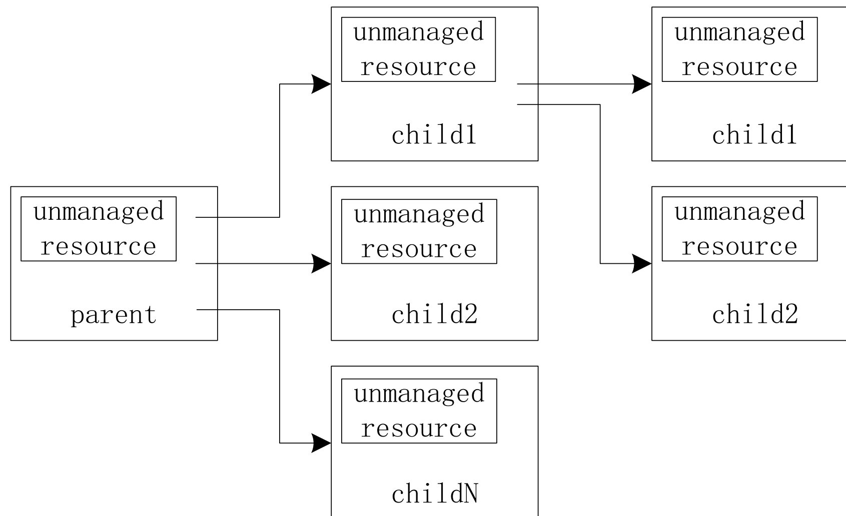


图 4-15 组合模式中的非托管资源

上图 4-15 中一个对象可以包含许多成员对象，这些成员对象又可以包含更多的成员对象，图中每个对象都有可能使用了非托管资源，我们的职责就是在 parent 释放非托管资源的时候，将它下级以及下下级（甚至更多）的所有成员对象的非托管资源全部释放。

> 注：图 4-15 中的 PARENT 是相对的，也就是说，PARENT 也有可能成为另外一个对象的成员对象。纵观程序中各个对象之间的关系，几乎都是这种结构，我们列举出来的只是其中的一小部分，图中 CHILDN 也可能成为另外一个 PARENT。

继承与组合同时存在的情况就很简单了，将两种释放非托管资源的方法合并，代码如下（不完整）：

```
//Code 4-8
class A:ABase
{
    //...
    B _b1;    //member
    B _b2;     //member
    public A()
    {
        //...
        _b1 = new B();
        _b2 = new B();
    }
    public override void DoRelease()
    {
        // 1.release A's unmanaged resource
        // 2.release member's unmanaged resource
```

```
            // 3.release ABase's unmanaged resource

            ReleaseMyUnManagedResource();    //NO.1
            _b1.ReleaseUnManagedResource();    //NO.2
            _b2.ReleaseUnManagedResource();
            base.DoRelease();    //NO.3
        }
        private void ReleaseMyUnManagedResource()
        {
            //...
            //release A's unmanaged resource here
        }
}
```

代码 Code 4-8 中，A 类型派生自 ABase 类型，同时 A 类型中包含_b1 和_b2 两个成员对象，在 A 类型内部，我们重写 DoRelease 虚方法，首先释放 A 中的非托管资源（NO.1），然后释放成员对象的非托管资源（NO.2），最后释放基类的非托管资源（NO.3）。

> 注意，类型 ABASE 的内部结构跟 A 类型一样（只要它们的最顶层基类中有 RELEASEUNMANAGEDRESOURCE 公开方法和对应的析构方法），类型 B 的内部结构也跟 A 类型一样，也就是说，每个类型的内部结构都与 A 类型一样。另外，代码中 NO1. NO2 以及 NO3 的顺序是可以改变的，换句话说，非托管资源的释放顺序也是可以改变的。

实例代码对应的非托管资源释放顺序如下图 4-16：

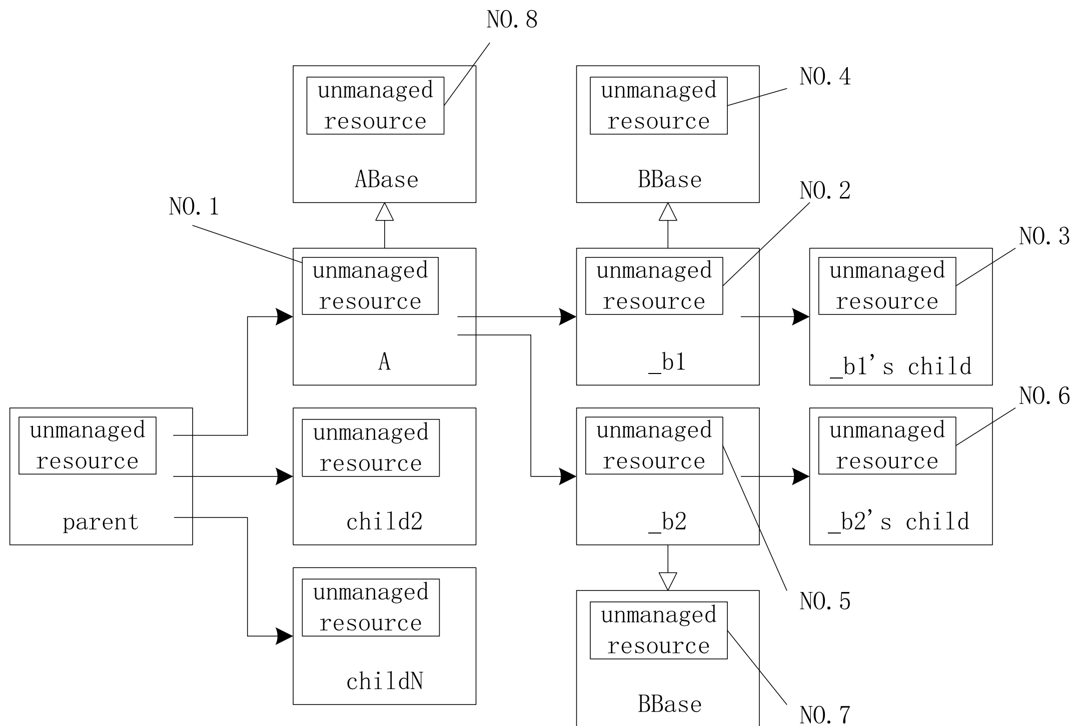


图 4-16 非托管资源的释放顺序

图 4-16 中顺序号为非托管资源的释放顺序，对于每一个单独的对象而言，都是遵循“自己-成员对象-基类”这样的一个顺序。图 4-16 中非托管资源的释放顺序并不是固定的。

4.4 正确使用 IDisposable 接口

上一节讲到了管理对象非托管资源的方法。如果一个类型需要使用非托管资源，那么我们可以这样去做：

（1）在类型中提供类似 ReleaseUnManagedResource 这样的公开方法，当对象不再使用时，开发者可在程序中人工显式释放非托管资源；

（2）编写类型的析构方法，在析构方法中编写释放非托管资源的代码，防止开发者没有人工显式释放非托管资源而造成资源泄露异常。

既然这些都是总结出来管理非托管资源的有效方法，那么我们在编程过程中就应该把它当做一个规则去遵守。.NET 类库中有一个 IDisposable 接口，几乎每个使用非托管资源的类型都应该实现该接口。

4.4.1 Dispose 模式

“Dispose 模式”便是管理对象非托管资源的一种原则，微软官方发布类库中所有使用了非托管资源的类型都遵循了这一原则。该模式很简单，定义一个 IDisposable 接口，该接口包含一个 Dispose 方法，所有使用了非托管资源的类型均应该实现该接口，类似代码如下：

```
//Code 4-9
interface IDisposable
{
    void Dispose();
}
class ABase:IDisposable
{
    //…
    bool _disposed = false;    //check if released or not
    public bool Disposed
    {
        get
        {
            return _disposed;
        }
    }
    public ABase()
    {
        //…
    }
    public void Dispose()    //NO.1
    {
        if(!_disposed)
        {
```

```
                Dispose(true);
                GC.SuppressFinalize(this);
                _disposed = true;
            }
        }
        protected virtual void Dispose(bool disposing)
        {
            if(disposing)
            {
                //release member's unmanaged resource
            }
            //release ABase's unmanaged resource
            //no base class
        }
        ~ABase
        {
            Dispose(false);    //call the virtual Dispose method,maybe override in derived class
        }
    }
    class A:ABase
    {
        //...
        public A()
        {
            //...
        }
        protected override void Dispose(bool disposing)
        {
            if(disposing)
            {
                //release member's unmanaged resource
            }
            //release A's unmanaged resource
            base.Dispose(disposing);    //NO.2
        }
    }
    class B:A
    {
        //...
        public B()
        {
            //...
        }
        public void DoSomething()
        {
            if(Disposed)    //if released, throw exception
```

```
        {
            throw new ObjectDisposedException(...);
        }
        // do something here
    }
    protected override void Dispose(bool disposing)
    {
        if(disposing)
        {
            //release member's unmanaged resource
        }
        //release B's unmanaged resource
        base.Dispose(disposing);    //NO.3
    }
}
```

代码 Code 4-9 中 Abase 类实现了 IDisposable 接口，并提供了两个 Dispose 方法，一个不带参数的普通方法和一个带有一个 bool 类型参数的虚方法。如果程序中人工显式调用 Dispose()方法去释放非托管资源，那么同时会释放所有成员对象的非托管资源（disposing 参数为 true）；如果不是人工显式调用 Dispose()方法释放非托管资源而是交给析构方法去负责，那么就不会释放成员对象的非托管资源（disposing 参数为 false），这样一来，所有成员对象都得由自己的析构方法去释放各自的非托管资源。

ABase 类型的所有派生类，如果使用到了非托管资源，只需要重写 Dispose(bool disposing)虚方法，在其中编写释放自己使用的非托管资源代码。如果有必要（disposing 为 true），则释放自己成员对象的非托管资源，最后再调用基类的 Dispose（bool disposing）虚方法（NO.2 和 NO.3）。另外对象中每一个方法执行之前需要判断自己是否已经 Disposed，如果已经 Disposed，说明对象已经释放非托管资源，大多数时候该对象不会再正常工作，因此抛出异常。

> 注：析构方法只需要在 ABASE 类中编写一次，其派生类中不需要再有析构方法。如果派生类中有析构方法，也一定不能再调用 DISPOSE(BOOL)虚方法，因为析构方法默认是按照"底层-顶层"这样的顺序依次调用，多个析构方法多次调用 DISPOSE(BOOL)，会重复释放非托管资源，引起不可预料异常。

前面提到过，为了确保程序中人工显式释放非托管资源的代码在任何情况中一定执行，需要把代码放在 try/finally 块中。C#中还有一种更为简洁的写法，只要我们的类型实现了 IDisposable 接口，那么就可以这样编写代码：

```
//Code 4-10
using(A a = new A())
{
    a.DoSomething();
}
```

这段代码 Code 4-10 编译之后相当于：

```
//Code 4-11
A a = new A();
try
```

```
{
    a.DoSomething();
}
finally
{
    a.Dispose();
}
```

这样就能确保 a 对象使用完毕后，a.Dispose 方法总能够被调用。在使用 FileStream、SqlDataConnection 等实现了 IDisposable 接口类型的对象时，我们几乎都可以这么使用：

```
//Code 4-12
using(FileStream fs = new FileStream(…))
{
    //deal with fs
}
```

在实际开发过程中，我们经常能遇见应用了“Dispose 模式”的场合，比如 Winform 开发中，新建一个窗体类 Form1，Form1.Designer.cs 中默认生成的代码如下：

```
//Code 4-13
protected override void Dispose(bool disposing)
{
    if (disposing && (components != null))
    {
        components.Dispose();    //NO.1
    }
    // add your code to release Form1's unmanaged resource    //NO.2
    base.Dispose(disposing);    //NO.3
}
```

因为 Form1 派生自 Form，Form 又间接派生自 Control，Control 派生自 Component，最后 Component 实现了 IDisposable 接口，换句话说，Form1 间接实现了 IDisposable 接口，遵循“Dispose 模式”，那么它就应该重写 Dispose（bool disposing）虚方法，并在 Dispose（bool disposing）虚方法中释放成员对象的非托管资源（NO.1 处）、释放本身使用的非托管资源（NO.2 处）以及调用基类的 Dispose（bool disposing）虚方法（NO.3 处）。

> 注：FORM1 中使用了非托管资源的成员对象几乎都派生自 COMPONENT 类型，并存放在 COMPONENTS 容器中，这些代码基本都由窗体设计器（FORM DESIGNER）自动生成，本书第七章中有讲到。

总之，记住如果一个类型使用了非托管资源，或者它包含使用了非托管资源的成员，那么我们就应该应用“Dispose 模式”：正确地实现（间接或直接）IDisposable 接口，正确的重写 Dispose（bool disposing）虚方法。

4.4.2 对象的 Dispose()和 Close()

很多实现了 IDisposable 接口的类型同时包含 Dispose()和 Close()方法，那么它们究竟有什么区别呢？都是用来释放非托管资源的吗？

事实上，它两没有绝对的确定关系，有的时候 Close 的字面意思更形象。比如一个大门

Gate 类，使用 Close 作为方法名称比使用 Dispose 更直白，因此有时候把 Dispose()方法屏蔽，用 Close()方法代替 Dispose()方法，作用跟 Dispose 方法完全一样。

```
//Code 4-14
class Gate:IDisposable
{
    //…
    public Gate()
    {
        //…
    }
    void IDisposable.Dispose()    //implement IDisposable explicitly
    {
        Close();
    }
    public void Close()
    {
        if(!_disposed)
        {
            Dispose(true);
            GC.SuppressFinalize(this);
            _disposed = true;
        }
    }
    //other methods
}
```

上面代码 Code 4-14 中 Close 方法就起到与 Dispose 方法一样的作用，意思便是释放非托管资源。另外还有一些情况 Close()与 Dispose()方法同时存在，但是作用却并不一样。

总之，凡是谈到 Dispose()，与它的字面意思一样，意思是释放非托管资源，Dispose()方法调用后对象就不能再使用。而谈到 Close()，它有时候与 Dispose()的功能一样，比如 FileStream 类型，而有的时候却跟 Dispose()不一样，它仅仅表示“关闭”的意思，说不定还会有一个 Open()方法与它对应，比如 SqlConnection。

不管是 Dispose 还是 Close，我们需要注意一点，如果释放了对象的非托管资源，那么这个对象就不能再使用，否则就会抛出异常。我们调试代码的时候偶尔会碰到类似“无法使用已经释放的实例”的错误信息，意思便是不能再使用已经释放非托管资源的对象，原因很简单，非托管资源是对象正常工作时不可缺少的一部分，释放了它，对象肯定不能正常工作。下图 4-17 表示人工显式释放对象非托管资源在对象整个生命周期的时间点位置：

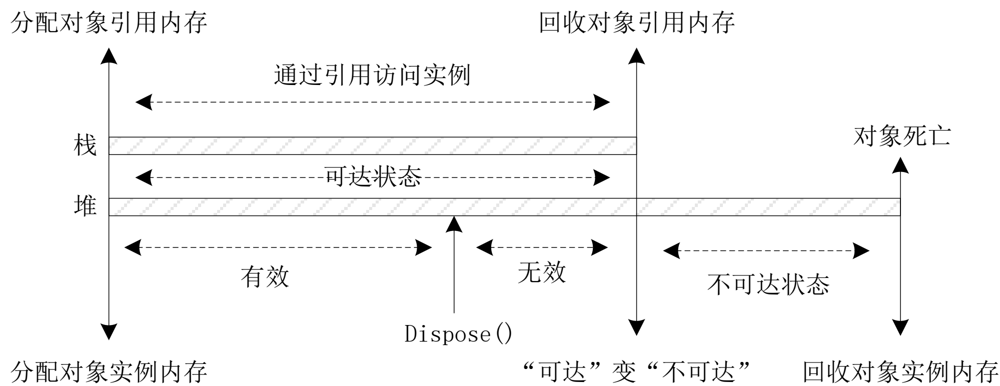


图 4-17 释放非托管资源时间点

图 4-17 中调用对象 Dispose()方法是在对象处于“可达”状态时。Dispose()方法调用之前，对象可以正常使用，调用之后，虽然仍然有引用指向它，但是对象已经不能再使用。在图 4-17 中“无效”区域操作对象时，大部分都会抛出异常。

> 注：对象释放非托管资源后，也就是调用了 DISPOSE()或者 CLOSE()方法之后，不代表该对象死亡，这时候还是可以有引用指向它，继续访问它，虽然此时所有的访问几乎都已无效。

4.5 本章回顾

本章开头介绍了编程中“堆”和“栈”的概念，它们是两种不同的数据结构，程序运行期间起着非同一般的作用；之后着重介绍了存放在堆中对象以及它的生命周期，该部分的配图比较多也比较详细，读者可以仔细阅读配图，相信能更清楚地理解对象在堆中的“从生到死”；之后介绍了.NET 对象使用到的“非托管资源”以及怎样去正确地管理好这些资源，章节最后提到了“Dispose 模式”，它是管理对象非托管资源的有效方式，同时还解释了为什么某些类型同时具备 Close()和 Dispose()方法。对象生命期是.NET 编程中的重点，清楚了解对象的“生”到“死”是能够编写出稳定程序的前提。

4.6 本章思考

1.“当栈中没有引用指向堆中对象实例时，GC 会对堆中实例进行内存回收”这句话是否准确？为什么？

A：不准确。因为 GC 不但会检查栈中的引用，还会检查堆中是否有引用。因此，只有当没有任何引用指向堆中对象实例时，GC 才会考虑回收对象实例所占用的内存。

2.如果一个类型正确地实现了 IDisposable 接口，那么调用该类型对象的 Dispose()方法后，是否意味着该对象已经死亡？为什么？

A：调用一个对象的 Dispose()方法后，并不表明该对象死亡，只有 GC 将对象实例占用的内存

回收后，才可以说对象死亡。但是通常情况下，调用对象的 Dispose()方法后，由于释放了该对象的非托管资源，因此该对象几乎就处于“无用”状态，“等待死亡”是它正确的选择。

3.如果一个类型使用了非托管资源，那么释放非托管资源的最佳时机是什么时候？

A：当对象使用完毕后，就应该及时释放它的非托管资源，比如调用它的 Dispose()方法（如果有），对象的非托管资源释放后，对象基本上就处于“无用”状态，因此一般不能再继续使用该对象。为了防止遗忘手动释放对象的非托管资源，我们应该在对象的析构方法中编写释放非托管资源的代码，这样一来，假如我们没有手动释放对象的非托管资源，GC 也会在适当时机调用析构方法，对象的非托管资源总能正确被释放。

第五章 重中之重：委托与事件

委托是.NET 编程中的重点之一，委托的作用简单概括起来就是“调用方法”。使用委托，我们可以异步（同步）调用方法、一次调用多个方法甚至可以将方法作为参数传递给别人供别人回调。程序的运行过程便是方法之间的调用过程，所以委托是.NET 开发者必须掌握的知识点之一。.NET 编程中的事件建立在委托的基础之上，要掌握事件的用法必须先了解委托。

5.1 什么是.NET 中的委托

委托的字面意思为“把什么什么东西托付给某某人去做”，偏向于一个动作，但是在.NET 中，委托却是一个名词，表示“代理”或者“中间人”的意思。A 本来要找 B 办事，但是它没有直接找 B，而是托付给 C，让 C 去找 B 把事儿给办了，如果按照“委托”字面意思去理解，“A 找 C”的这个行为叫“委托”，但是在.NET 中，C 这个人叫“委托”。

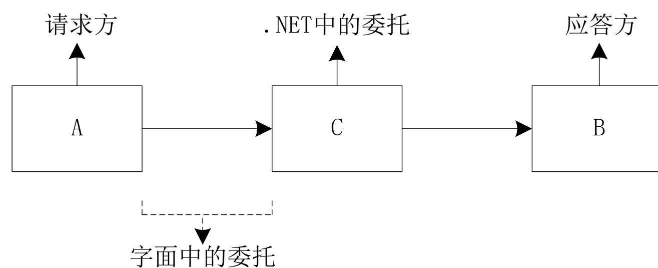


图 5-1 .NET 中委托含义

图 5-1 中 A 表示请求办事情的人（请求方），B 是最终处理事情的人（应答方），C 表示.NET 中的委托。既然 C 是中间人，那么它肯定包含有 B 的一些信息，不然怎么去找 B 办事情？

> 注：本书中之后出现的所有与“委托”有关的词汇均指.NET 中的委托，也就是图 5-1 中的 C 部分。另外，按照第二章中所讲的内容，A 可以称为 CLIENT，B 则称为 SERVER。

5.1.1 委托的结构

委托的职责就是代替请求方去找应答方办事情，在程序中，体现为调用应答方的方法，换句话说，委托其实就是起到“调用方法”的作用。

程序中调用一个方法的必备条件是：知道要调用的方法，知道这个方法的所有者（如果该方法为实例方法）。因此一个委托中至少要包含图 5-2 中的信息：

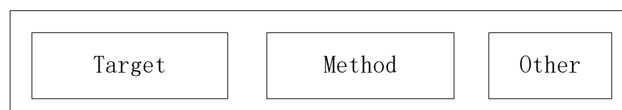


图 5-2 委托组成

图 5-2 中显示一个委托的结构组成，它至少包含要调用的方法 Method 和方法的所有者 Target（如果方法为静态方法，Target 为 null）。也就是说，委托是一种数据结构，我们可以把它看作是一种类型，类型里面包含一些成员。事实上，.NET 中的委托就是一种类型，有着共同的基类 Delegate，我们程序中定义的各种各样的委托都是从该类派生而来。

注：我们使用到的委托类型都派生自 MULTICASTDELEGATE 类，后者再派生自 DELEGATE 类型。系统不允许我们像定义普通类型的方式显式从这两个类型派生出新的委托，只能使用一种特殊定义类型的方法（后面有讲到）。另外，我们平时常说的"委托"是指一个委托类型的对象，本书中可以根据上下文判断"委托"是指委托类型还是委托类型的对象。

由于每个方法的签名不一样，因此一种委托只能负责调用一种类型的方法，也就是说，我们在定义委托类型的时候，必须提供它能够调用方法的签名，因此，.NET 中规定，以如下形式去定义一个委托类型：

```
//Code 5-1
public delegate void DelegateName(object[] arg1);
```

像普通声明一个方法一样，提供方法名称、参数、访问修饰符以及返回值，然后在前面加上 delegate 关键字，这样就定义了一个委托类型，委托类型名称为 DelegateName，它能够调用返回值为 void，带有一个 object[]类型参数的所有方法（包括实例方法和静态方法）。换句话说，就是所有符合该签名的方法都可以由 DelegateName 委托调用。注意我们不能显式在代码中这样去定义一个委托类型：

```
//Code 5-2
public class DelegateName:MulticastDelegate
{
    //…
}
```

编译器不允许以上代码 Code 5-2 通过编译。

注："方法签名"指方法的参数个数、参数类型以及返回值等，具有相同签名的两个方法参数列表一致，返回值一致（名称可以不一样），INT FUN1(STRING A,INT B)与 INT FUN2(STRING B,INT A)两个方法的签名相同。

委托类型定义完成后，怎么去实例化一个委托对象呢？其实很简单，跟实例化其它类型对象一样，我们可以通过 new 关键字，

```
//Code 5-3
class Calculate
{
    public Calculate()
    {
        //…
    }
    public int DoDivision(int first,int second)    //NO.1
    {
        return first/second;
```

```
    }
}
private delegate int DivisionDelegate(int arg1,int arg2);    //NO.2
class Program
{
    static void Main()
    {
        Calculate c = new Calculate();
        DivisionDelegate d = new DivisionDelegate(c.DoDivision);    //NO.3
        int result = d(10,5);    // int result = c.DoDivision(10,5);        NO.4
        Console.WriteLine("the result is " + result);
    }
}
```

代码 Code 5-3 中我们定义了一个 Calculate 类型，专门负责除法运算（NO.1 处），定义了一个 DivisionDelegate 委托（NO.2 处）。在实际计算的时候，我们并没有直接调用 Calculate 类的 DoDivision 方法，而是先新建了一个委托对象 d（NO.3 处），给 d 的构造方法传递一个参数 c.DoDivision。之后，我们通过这个委托 d 来计算 10 除以 5 的值（NO.4 处）。整个过程中，我们没有直接使用对象 c，而是通过委托 d，这就像本节刚开始所说的：委托的职责就是代替请求方（Program 类）去找应答方（c 对象）办事情（除法运算）。代码中委托对象 d 的结构如下图 5-3：

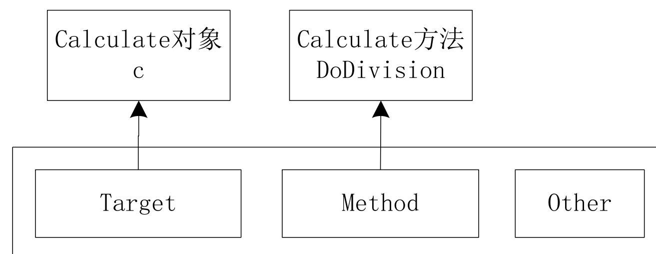


图 5-3 委托对象 d 内部结构

图 5-3 中显示，委托中的 Target 指向 c 对象，Method 指向 c 对象的 DoDivision 方法，委托对象 d 就是对 c.DoDivision(int,int)的一个封装。

另外，在我们使用 new 关键字创建委托实例时，会给它的构造方法传递了一个参数，该参数为一个方法名称。如果是实例方法，就应该使用“对象.方法名称”这样的格式（注意如果在同一个类中，对象默认为 this，可以省略），如果是静态方法，就应该使用“类名称.方法名称”这样的格式（如果在同一个类中，类名称可以省略）。给构造方法传递的这个参数其实就是用来初始化委托内部的 Target 和 Method 两个成员。使用委托调用方法时，我们直接使用“委托对象（参数列表）;”这样的格式即可，它等效于“委托对象.Invoke(参数列表)”。

> 注：给委托赋值的另外一种方式是：委托对象=方法。代码 CODE 5-3 中赋值部分可以换成 DIVISIONDELEGATE D = C.DODIVISION;，含义跟用 NEW 关键字一样。另外，每一个自定义委托类型都包含一个 INVOKE 方法，它的作用就是调用方法（与 BEGININVOKE 方法对应，详见本书第六章），"委托对象(参数列表)"只是调用方法的一种简写方式。

委托内部的 Target 为 Object 类型，表示方法的所有者，Method 为 MethodInfo 类型，表

示一个方法。通过委托调用方法“int result = d(10,5);”，委托内部相当于：

```
//Code 5-4
int result = (int)Method.Invoke(Target,new Object[]{10,5});
```

意思就是在指定的对象（Target）上调用指定的方法（Method）。

5.1.2 委托链表

上一小节中提到的委托都是单委托，它只对一个方法进行封装，也就是说，使用单委托只能调用一个方法。

之前提到过，一个委托应该可以调用多个方法，只要这些方法的签名与该委托一致，那么怎样让一个委托同时调用两个或者两个以上的方法呢？ 我们代码中很好实现，直接使用加法赋值运算符（+=）将多个方法附加到委托对象上，

```
//Code 5-5
class Program
{
    static void Fun1(object sender,EventArgs e)
    {
        //…
        Console.WriteLine(“Call Fun1”);
    }
    static void Fun2(object sender,EventArgs e)
    {
        //…
        Console.WriteLine(“Call Fun2”);
    }
    static void Fun3(object sender,EventArgs e)
    {
        //…
        Console.WriteLine(“Call Fun3”);
    }
    static void Main()
    {
        EventHandler eh = new EventHandler(Fun1);    //NO.1
        eh += Fun2;    //NO.2
        eh += new EventHandler(Fun3);    //NO.3
        eh -= Fun2;    //NO.4
        eh(null,null);    //NO.5
        // print out:
        // Call Fun1
        // Call Fun3
    }
}
```

代码 Code 5-5 中定义了一个 EventHandler 委托对象 eh（NO.1 处），按照先后顺序依次使用加法赋值运算符（+=）给它附加 Fun2 和 Fun3 方法（NO.2 和 NO.3 处），然后使用减法赋值运算符(-=)移除 Fun2 方法(NO.4 处)，最后通过委托调用方法，依次输出“Call Fun1”和“Call Fun3”。

由此可以得出三个结论：

（1）一个委托对象确实可以调用多个方法；

（2）这些方法可以按照附加顺序先后依次调用；

（3）可以从委托对象上移除一个方法，不影响其它方法。

注：确切的说，应该是将委托附加到委托对象上，另外代码中使用的都是静态方法，这时候委托内部 TARGET 为 NULL。+=和-=运算符相当于 DELEGATE 类的静态方法 DELEGATE.COMBINE 和 DELEGATE.REMOVE，专门负责附加或移除委托操作。

根据以上三个结论，我们很有必要了解一下委托内部到底是怎样管理附加到它上面的方法，换句话说，委托内部到底有怎样的数据结构来组织和调用这些方法？

在学习数据结构中的“链表”时我们知道，每一个链表节点（Node）的结构都是相同的。链表表头、链表表尾以及中间的节点本质上是没有任何区别，我们可以将任意一个（或一串）节点附加到已有的一个（或一串）节点后面，从而形成一个更长的节点串。我们还能通过链表表头访问整个链表中的每一个节点（通过 Next 成员）。总之，只要知道了任意一个节点，我们就能访问该节点后面的所有节点（注意这里指的是单向链表）。单向链表结构类似如下图 5-4：

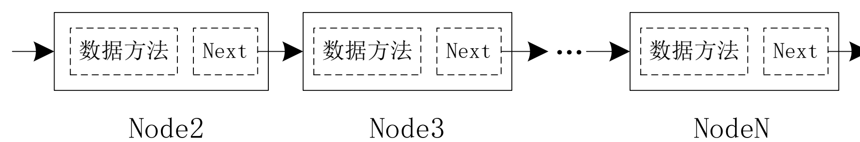


图 5-4 单向链表结构

图 5-4 中实线矩形方框表示单向链表中的一个节点，所有节点都属于同一类型对象，因此结构相同。节点类 Node 代码类似如下：

```
//Code 5-6
class Node
{
    private string _name;    //node's name
    private Node _next;    // the next node
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }
    public Node Next
    {
        get
        {
            return _next;
```

```
        }
        set
        {
            _next = value;
        }
    }
    public Node(string name)
    {
        _name = name;
    }
    public int GetNodesCount()    //get the nodes' count from this to the end
    {
        int count = 0;
        Node tmp = this;
        do
        {
            count++;
            tmp = tmp.Next;
        }
        while(tmp != null)
        return count;
    }
    public Node[] GetNodesList()    //get all nodes from this to the end
    {
        Node[] nodes = new Node[GetNodes()];
        int index = -1;
        Node tmp = this;
        do
        {
            index++;
            nodes[index] = tmp;
            tmp = tmp.Next;
        }
        while(tmp != null)
        return nodes;
    }
    public void ShowMyInfo()    //show node's info
    {
        Console.WriteLine("My name is " + _name);
    }
    public void ShowInfo()    //show the all nodes' info from this to the end
    {
        ShowMyInfo();
        if(Next != null)
        {
            Next.ShowInfo();
```

```
			}
		}
}
class Program
{
	static void Main()
	{
		Node node1 = new Node(“node1”);
		Node node2 = new Node(“node2”);
		Node node3 = new Node(“node3”);
		node1.Next = node2;    //NO.1
		node2.Next = node3;    //NO.2
		Console.WriteLine(“the count of the nodes from node1 to the end:” + 
node1.GetNodesCount());    //NO.3
		Console.WriteLine(“the count of the nodes from node2 to the end:” + 
node2.GetNodesCount());
		Console.WriteLine(“the count of the nodes from node3 to the end:” + 
node3.GetNodesCount());
		Node[] nodes = node1.GetNodesList();    //NO.4
		foreach(Node n in nodes)
		{
			n.ShowMyInfo();    //NO.5
		}
		node1.ShowInfo();    //NO.6
		Console.Read();
	}
}
```

代码 Code 5-6 中我们可以通过一个节点访问该节点以及该节点所有的后续节点（NO3、NO.4、NO.5 以及 NO.6 处），之所以能够这样，是因为每个节点中都保存有下一个节点的引用（Next 引用）。代码中的 node1. node2 以及 node3 组成的单向链表在堆中的存储结构如下图 5-5：

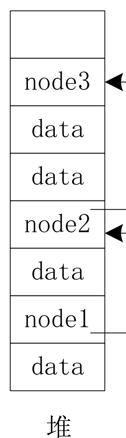


图 5-5 单向链表在堆中的结构

通过一个单向链表中的节点对象，我们能够访问附加到它后面的所有其它节点，委托对象也能够管理和访问附加到它上面的其它委托，也能管理一个“链表”，那么，我们是否可以按照单

向链表的结构去理解委托的内部结构呢？答案虽是肯定的，但是委托内部的“链表”结构跟单向链表的实现原理却不相同，它并不是通过 Next 引用与后续委托建立关联，而是将所有委托存放在一个数组中，类似如下图 5-6：

> 注：准确来讲，委托内部结构不应该称为“链表”。

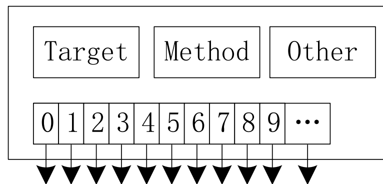


图 5-6 委托结构

图 5-6 中显示委托内部不仅仅有 Target 和 Method 成员，还有一个数组成员，用来存储附加到该委托对象中的其它委托。委托链在堆中的结构如下图 5-7：

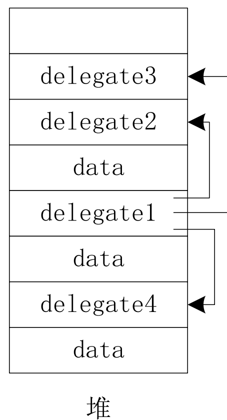


图 5-7 委托链表在堆中的结构

图 5-7 中显示 delegate1 中包含 delegate2、delegate3 以及 delegate4 的引用，注意 delegate2、delegate3 以及 delegate4 中的数组列表不可能再包含有其它的委托引用，也就是说包含关系最多只有两层，具体原因请参见下一小节有关委托的“不可改变”特性。

> 注：每一个委托类型都有一个公开的 GetInvocationList()的方法，可以返回已附加到委托对象上的所有委托，也就是图 5-6 中数组列表。另外，我们平时不区分委托对象和委托链表，提到委托对象，它很有可能就表示一个委托链表，这跟单向链表只包含一个节点时道理类似。

既然现在委托可以调用多个方法，那么它的 Invoke 方法内部是怎样实现的呢？假如是一个简单的单委托，Invoke()方法内部直接调用 Method.Invoke 方法，但如果包含其它委托，那么它就需要遍历整个数组列表。代码类似如下（假设委托的签名为：返回值为 null，含一个 int 类型参数）：

```
//Code 5-7
public void Invoke(int a)
{
    Delegate[] ds = GetInvocationList();    //get all delegates in array
```

```
        if(ds!=null)    //contain a delegate chain
        {
            foreach(Delegate d in ds)    // call each delegate
            {
                DelegateName dn = d as DelegateName;
                dn(a);
            }
        }
        else    //don't contain a delegate chain
        {
            Method.Invoke(Target,new Object[]{a});    //call the Method on Target with argument 'a'
        }
}
```

代码 Code 5-7 中委托的 Invoke 方法先判断该委托中是否包含其它委托，如果是，依次遍历列表调用这些委托；否则，说明当前委托是一个单委托，直接调用 Method.Invoke()方法。

5.1.3 委托的"不可改变"特性

所谓"不可改变"（Immutable），就是指一个对象创建之后，它的内容不能再改变。比如常见的 String 类型，我们创建的一个 String 对象之后，之后在该对象上的所有操作都不会影响对象原来的值，

```
//Code 5-8
Class Program
{
    static void Main()
    {
        string a = "test";    // equal String a = new String("test");
        a.ToUpper();    //NO.1
        Console.WriteLine("a is " + a);
        // print out:
        // a is test
    }
}
```

代码 Code 5-8 中 a 的值并没有因为调用了 a.ToUpper()方法而改变，如果想要让 a 字符串都变为大写格式，必须使用"a = a.ToUpper();"这样的代码，a.ToUpper()方法会返回一个全新的 String 对象，a 重新指向该新对象。注意这里的"不可改变"指的是对象实例，而不是对象引用，也就是说我们还是可以将 a 指向其它对象。如下图 5-8：

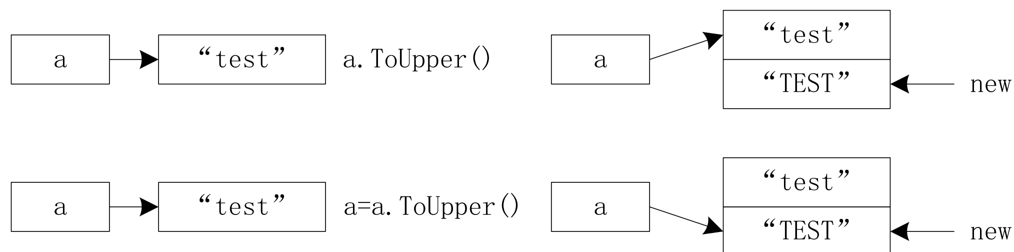


图 5-8 String 类型的不可变性

委托跟 String 类型一样，也是不可改变的。换句话说，一旦委托对象创建完成后，这个对象就不能再被更改，那么我们前面讲到的将一个委托附加到另外一个委托对象上形成一个委托链表又是怎么做到的呢？其实这个跟 String.ToUpper()过程类似，我们对委托进行附加、移除等操作都会产生一个全新的委托，这些操作并不会改变原有委托对象。

```
//Code 5-9
EventHandler eh = new EventHandler(Fun1);    //NO.1
EventHandler tmp = eh;    //tmp and eh point at the same delegate      NO.2
EventHandler eh2 = new EventHandler(Fun2);    //NO.3
eh += eh2;    //NO.4
// equal eh = Delegate.Combine(eh, eh2) as EventHandler;
EventHandler tmp2 = eh;    //tmp2 and eh point at the same delegate    //NO.5
EventHandler eh3 = new EventHandler(Fun3);    //NO.6
eh += eh3;    //NO.7
//equal eh = Delegate.Combine(eh,eh3) as EventHandler;
```

上面代码 Code 5-9 最终会在堆中产生 5 个委托对象，NO.1 处创建第一个,让 eh 指向它，NO.2 处让 tmp 与 eh 指向同一个委托，NO.3 处创建第二个，让 eh2 指向它，NO.4 处合并了 eh 和 eh2，但并没有改变原来的 eh 和 eh2，而是新创建了第三个，并且让 eh 重新指向了新创建的第三个，NO.5 处让 tmp2 与 eh 指向同一个委托，NO.6 处创建第四个，让 eh3 指向它，NO.7 处合并了 eh 和 eh3，但并没有改变原来的 eh 和 eh3，而是新创建了第五个，并且让 eh 重新指向了新创建的第五个。

我们对委托进行的每一个附加（+=或者 Delegate.Combine）操作，都会创建一个全新的委托，该新创建委托的数组列表中包含原来两个委托数组列表内容的总和，这个过程并不会影响原来的委托，移除（-=或者 Delegate.Remove）操作类似。附加或移除委托过程，见下图 5-9：

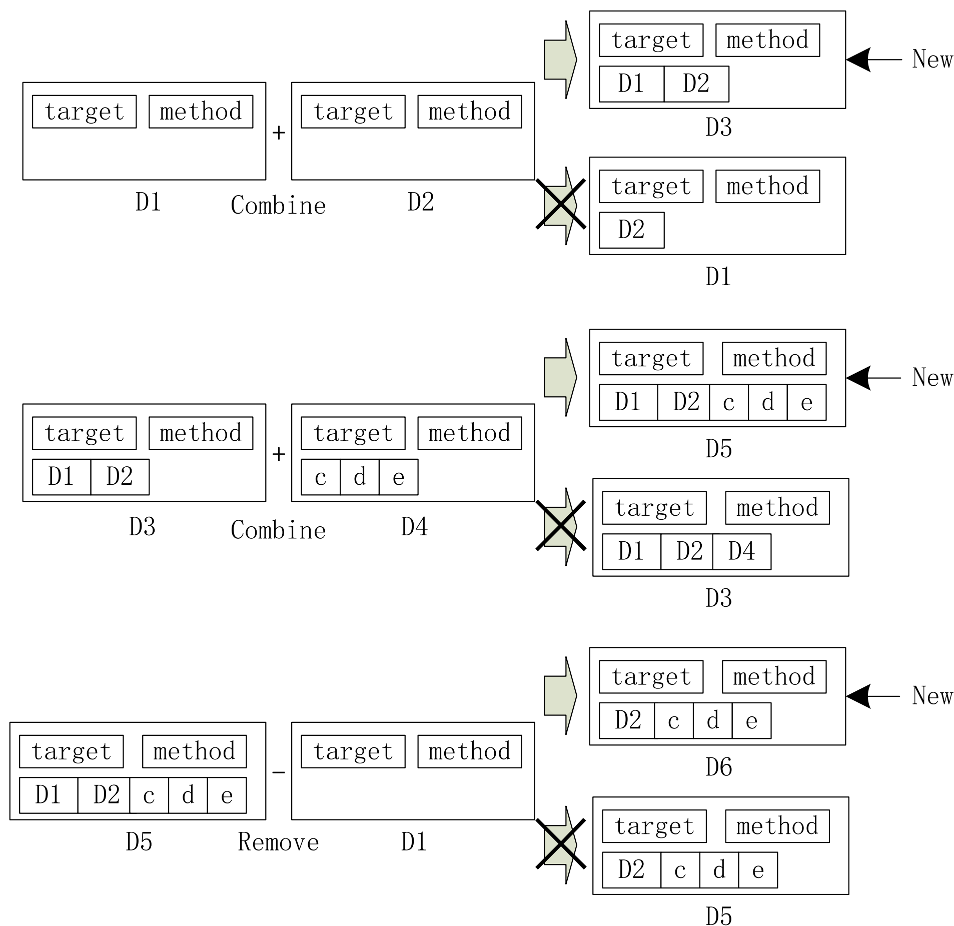


图 5-9 附加或移除委托过程

图 5-9 中 D1、D2、D3、D4、D5、D6 以及 c、d、e 均为委托对象引用。Delegate.Combine(D1,D2)产生了 D3，D1 并没改变；Delegate.Combine(D3,D4)产生了 D5，D5 包含 D3 和 D4 中的数组列表内容之和，D3 并没有改变；Delegate.Remove(D5,D1)产生了 D6，D5 并没有改变。由图 5-9 可以看出，委托包含关系最多只有两层，数组列表中的委托都属于单委托，单委托不再包含其它委托。

> 注：文中的委托对象、单委托、委托链表都是指一个委托类型的对象。

5.1.4 委托的作用

委托是一种数据结构，专门用来管理和组织方法，并负责调用这些方法。那么为什么需要委托来调用方法呢？原因有以下三点：

（1）编程中无时无刻都存在着“方法调用”，委托可以更方便更有组织的管理我们需要调用的方法，理论上没有数量限制，只要是符合某一个委托签名的方法都可以由该委托管理。我们可以使用委托一次性（有先后顺序）地调用这些方法。在使用委托之前，我们调用方法是这样：

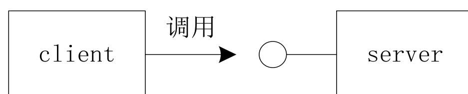


图 5-10 不使用委托调用方法

图 5-10 中为不使用委托直接调用方法的过程，我们每次只能调用一个方法。使用委托之后，

我们可以调用一系列方法，如下图 5-11：

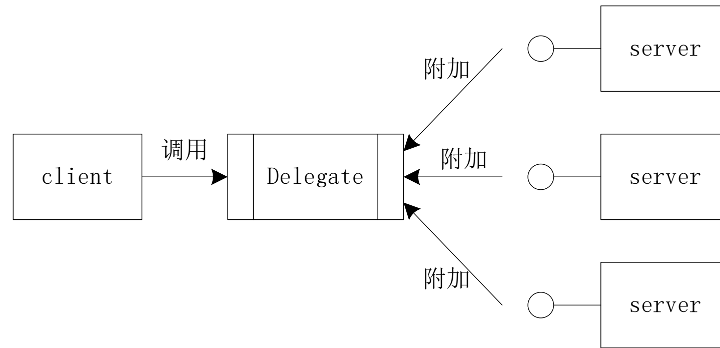


图 5-11 使用委托调用方法

上图 5-11 为使用委托调用方法的过程，使用一个委托我们可以管理多个方法，并且一次性调用这些方法。能够统一管理和组织被调用的方法，在编程中起到一个非常重要的作用，如后面讲到的“事件编程”。

（2）使用普通方式调用方法只能是同步的（特殊方法除外），也就是说，被调用方法返回之前，调用线程一直处于等待状态。使用委托调用方法时，有两种方式可供选择，既可以同步调用也可以异步调用，前者和普通调用方式一样，而后者遵循“异步编程模型”的规律：方法的调用不会阻塞调用线程。

> 注：委托的异步调用关键在于它的BEGININVOKE方法，该方法是INVOKE方法的异步版本，详见第六章关于异步编程的介绍。

（3）有了委托，方法可以作为一种参数在代码中进行传递，这个类似于 C++中的函数指针。委托的这种功能在框架中是非常有用的，框架一般由专业技术团队编写开发，由于框架的开发者并不知道框架使用者的具体代码，那么框架又是怎样调用使用者编写的代码呢？

框架有两种方式调用框架使用者编写的代码，一种便是面向抽象编程。框架中尽量不出现某个具体类型的引用，而是使用抽象化的基类引用或者接口引用代替。只要框架使用者编写的类型派生自抽象化的基类或实现了接口，框架均可以正确地调用它们。我们常见的使用 using 代码块来释放对象非托管资源就是一个例子：

```
//Code 5-10
using(FileStream fs = new FileStream(…))
{
    //use fs
}
```

代码 Code 5-10 中要求 FileStream 类必须实现了 IDisposable 接口（事实上确实如此）。代码 Code 5-10 经过编译后，与下面代码 Code 5-11 类似：

```
//Code 5-11
IDisposable dispose_target = new FileStream(…);
try
{
    //use filestream
}
finally
{
    dispose_target.Dispose();
```

```
}
```

如上代码 Code 5-11 所示，无论何时，FileStream 对象都能正确地释放非托管资源。框架认为所有使用 using 来释放非托管资源的类型都已实现了 IDisposable 接口，因为只有这样，它才能够提前编写释放非托管资源的代码（如 finally 中的 dispose_target.Dispose()）。没有实现 IDisposable 接口的类型不能使用 using 关键字来释放非托管资源。

> 注：关于框架调用框架使用者代码的过程，可以参见第二章中关于对"协议"的介绍，如图 2-14。

框架调用框架使用者代码的另外一种方式就是使用委托，将委托作为参数（变量）传递给框架，框架通过委托调用方法。异步编程中的一些方法往往带有委托类型的参数，比如 FileStream.BeginRead、Socket.BeginReceive 等等（后续章节有讲到）。这些方法都会带有一个 AsyncCallBack 委托类型的参数，我们在使用这些方法时，如果给它传递一个委托对象，当异步操作执行完毕后，框架自动会调用我们传递给它的委托。还有下一节中讲到的"事件"，框架可以通过事件来调用框架使用者编写的代码，如事件发布者激发事件，调用事件注册者的事件处理程序。

> 注：我们使用.NET 中预定义的一些类型、方法均可以当作框架中的一部分。

5.2 事件与委托的关系

委托的附加、移除以及调用，是没有范围限制的。如果一个类型包含一个委托成员，那么在类外部既可以给它附加或者移除委托，还可以调用这个委托。如下面代码：

```
//Code 5-12
public delegate void DelegateName(int a,int b);    //define a delegate type
class A
{
    public DelegateName MyDelegate;    //define a delegate member
    Public A()
    {
        //…
    }
    public void DoSomething()
    {
        //…
        if(MyDelegate != null)
        {
            //… if something happen or if something is OK
            int arg1 = 1; int arg2 = 2;
            MyDelegate(arg1,arg2);    //then call the delegate
        }
    }
    //…
}
```

```
class Program
{
    static void Fun1(int a,int b)
    {
        Console.WriteLine("the result is " + (a + b).ToString());
    }
    static void Main()
    {
        A a = new A();
        a.MyDelegate += new DelegateName(Fun1);    //NO.1
        a.DoSomething();    //NO.2
        a.MyDelegate(1,2);    //NO.3
    }
}
```

代码 Code 5-12 中，我们给 a 对象的 MyDelegate 附加一个方法后（NO.1 处），a 对象内部可以调用这个委托（NO.2 处），a 对象外部也可以调用这个委托（NO.3 处）。也就是说，对 MyDelegate 委托成员的访问是没有限制的，从某种意义上讲，这违背了“面向对象”思想，因为类里面的有些功能不应该对外公开，比如这里的“委托调用”，该操作应该只能发生在类型内部。如果我们把 MyDelegate 定义为 private 私有变量，那么我们在类外部就不能给它附加和移除方法，为了解决这个问题，.NET 中提出了一种介于 public 和 private 之间的另外一种访问级别：在定义委托成员的时候给出 event 关键字进行修饰，前面加了 event 关键字修饰的 public 委托成员，只能在类外部进行附加和移除操作，而调用操作只能发生在类型内部。如果把代码 Code 5-12 中 A 类声明 MyDelegate 成员的代码改为：

```
//Code 5-13
public event DelegateName MyDelegate;
```

按照 Code 5-13 中的方式定义的委托只能在 A 类内部调用，之前代码 Code 5-12 中的 NO.3 处编译通不过。

我们把类中设置了 event 关键字的委托叫作“事件”，“事件”本质上就是委托对象。事件的出现，限制了委托调用只能发生在一个类型的内部，如下图 5-12：

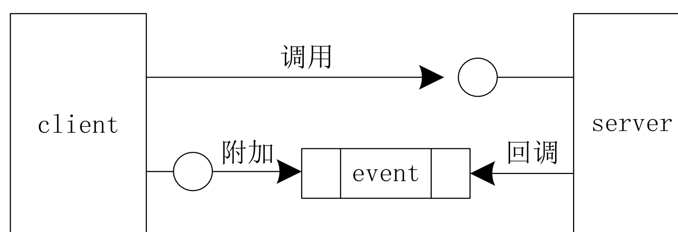


图 5-12 事件在程序调用中的位置

图 5-12 中 server 中的委托使用了 event 关键字修饰，只能在 server 内部调用，外部只能进行附加和移除方法操作。当符合某一条件时，server 内部会调用委托，这个时间不由我们（Client）控制，而是由系统（Server）决定。因此大部分时候，事件在程序中起到了回调作用（关于调用与回调的区别，参见第二章）。

调用加了 event 关键字修饰的委托也称为“激发事件”，调用方（图 5-12 中的 server）称为“事件发布者”，被调用方（图 5-12 中的 client）称为“事件注册者”（或“事件观察者”、“事件订阅者”等，本书中统一称之为“事件注册者”），附加委托的过程称之为“注册事件”（或“绑定事件”、“监听事件”、“订阅事件”等，本书中统一称之为“注册事件”），移除委托的过程称之为“注销事件”。通过委托调用的方法称为“事件处理程序”。

注：将只能在类型内部调用的委托称之为"事件"，主要是因为这些委托一般是当SERVER 中发生某件事件（或符合某个条件）时才被 SERVER 调用。我们所熟知的BUTTON.CLICK、TEXTBOX.TEXTCHANGED、FORM.FORMCLOSING 等事件，都属于这种情况。

“事件”在.NET 中起到了重要作用，它为框架与框架使用者编写代码之间的交互做出了重大贡献。

5.3 使用事件编程

5.3.1 注销跟注册事件同样重要

前面在讲到委托结构组成的时候就知道，委托内部包含了要调用的方法（Method 成员），以及该方法所属的对象（Target 成员）。当我们注册事件时，其实就是附加委托的过程，将一个新委托附加到委托链表中。事件注册者向事件发布者注册事件后，发布者就会保存一个注册者的引用（委托中的 Target 成员），发布者激发事件，其实就是通过该引用调用注册者的事件处理程序。当我们注销事件时，其实就是移除发布者对注册者的引用。

第四章讲到，堆中的对象实例如果存在引用指向它，那么 CLR 就不会回收它在堆中占用的内存，哪怕这个对象已经没有使用价值。注册事件使一个新的引用指向了事件注册者，如果我们不及时注销事件，那么这个引用将会一直存在。

5.3.2 多线程中使用事件

在通常编程中，我们激发一个事件之前需要先判断该事件是否为空，如果不为空，我们就可以激发该事件（调用委托），类似代码如下：

```
//Code 5-14
public event MyDelegate SomeEvent;
if(SomeEvent != null)    //NO.1
{
    //do something
    SomeEvent(arg1,arg2);    //NO.2    call the delegate
}
```

代码 Code 5-14 中 NO.1 处先检查 SomeEvent 是否为空，如果为空，说明没有人注册过该事件，就不会执行 if 块中的语句；如果不为空，说明已经有人注册过该事件，就执行 if 块中的语句，调用委托（图中 NO.2 处）。在单线程中，上面代码没有任何问题，但是如果在多线程中，以上代码就有可能抛出异常：如果在 NO.1 处 if 判断为 true，在 NO.2 执行之前，其它线程将 SomeEvent 改变为 null，这时候再回头执行 NO.2 时，就会抛出 NullReferenceException 的异常。

注：本章前面讲到的"委托不可改变特性"指的是委托实例不可改变，类似 STRING 类型，委托引用仍然可以改变，所以 SOMEEVENT 可以指向其它实例，甚至指向 NULL。

为了解决多线程中事件编程容易引发的异常，我们需要利用“委托不可改变”这一特点。

由于我们对一个委托的任何操作都不会改变该委托本身，只会产生新的委托，那么我们完全可以在 if 判断语句之前，使用一个局部临时变量来指向委托实例，之后所有的操作都针对该局部临时变量。由于局部变量不可能被其它人修改，所以它永远都不会指向 null。

```
//Code 5-15
MyDelegate tmp = SomeEvent;
if(tmp != null)    //NO.1
{
    //do something
    tmp(arg1,arg2);    //NO.2
}
```

上述代码 Code 5-15 中，先让 tmp 和 SomeEvent 指向同一委托实例，NO.1 处 if 判断为 true，if 块中的 tmp 在任何时候都不会被其它线程修改为 null，因为其它线程只能修改 SomeEvent，并且我们对 SomeEvent 的任何操作都不会改变它所指向的委托实例。这种解决方法其实跟我们在做一个除法运算时检测除数是否为零的原理一样，如果在多线程中，我们检查完除数不为零后，直接进行除法运算，有可能抛出异常，如下代码：

```
//Code 5-16
class A
{
    //…
    public int x;
    public A()
    {
        //…
    }
    public int DoSomething(int y)
    {
        if(x != 0)    //NO.1
        {
            return y/x;    //NO.2
        }
        else
        {
            return 0;
        }
    }
}
```

上述代码 Code 5-16 中，如果 NO.1 处 if 判断为 true 后，在 NO.2 执行之前 x 的值被其它线程改变为 0，那么代码执行到 NO.2 处时就会抛出异常。正确的做法是，使用一个临时变量存储 x 的值，之后所有的操作都是针对该临时变量。Code 5-16 中类 A 的 DoSomething 方法可以修改为：

```
//Code 5-17
public int DoSomething(int y)
{
    int tmp = x;
    if(tmp != 0)    //NO.1
```

```
    {
        return y/tmp;    //NO.2
    }
    else
    {
        return 0;
    }
}
```

上述代码 Code 5-17 中，NO.1 处 if 判断为 true 后，tmp 的值就永远不会为零，其它线程对 x 的所有操作都不会影响到 tmp 的值，因此 NO.2 处不可能再有异常抛出。这个原理跟我们刚学习编程的时候碰到的形参和实参的关系一样，在值传递过程中，形参和实参是相互独立的，形参改变不会影响到实参。

> 注：.NET 中值类型赋值都是值传递，也就是说赋值后会产生一个一模一样的拷贝，两者之间是相互独立互不影响的。引用类型赋值也是值传递，因为它传递的是对象引用，赋值后两个引用指向堆中同一个实例，关于值类型与引用类型赋值请参见第三章。

5.3.3 委托链表的分步调用

调用任何方法都有可能出现异常，因此，通过委托调用方法时，我们最好把调用代码放在 try/catch 块中，类似如下：

```
//Code 5-18
class A
{
    //...
    public event MyDelegate SomeEvent;
    public A()
    {
        //...
    }
    public void DoSomething()
    {
        //...
        MyDelegate tmp = SomeEvent;    //NO.1
        if(tmp != null)
        {
            //...
            try    //NO.2
            {
                tmp(arg1,arg2);    //NO.3
            }
            catch
            {
```

```
            }
        }
    }
}
```

上述代码 Code 5-18 中，激发事件的代码（NO.3 处）放在了 try/catch 块中，这样以来，万一事件注册者中的事件处理程序抛出了没有被处理的异常，try/catch 便会捕获该异常，程序不会异常终止。

调用委托链时，如果某一个委托对应的方法抛出了异常，那么剩下的其它委托将不再调用。这个很容易理解，本来是按先后顺序依次调用方法，如果其中某一个抛出异常，剩下的肯定被跳过。为了解决这个问题，单单是将激发事件的代码放在 try/catch 块中是不够的，我们需要分步调用每个委托，将每一步的调用代码均放在 try/catch 块中。类 A 的 DoSomething 方法修改为：

```
//Code 5-19
public void DoSomething()
{
    //…
    MyDelegate tmp = SomeEvent;    //NO.1
    if(tmp != null)
    {
        //…
        Delegate[] delegates = tmp.GetInvocationList();    //NO.2
        foreach(Delegate d in delegates)
        {
            MyDelegate del = d as MyDelegate;
            try    //NO.3
            {
                del(arg1,arg2);    //NO.4
            }
            catch
            {

            }
        }
    }
}
```

上述代码 Code 5-19 中，我们没有直接使用 tmp 来调用委托链表，而是先通过 tmp.GetInvocationList 方法来获取委托链表中的委托集合（NO.2 处），然后再使用 foreach 循环遍历集合，分步调用每个委托（NO.4 处），分步调用过程均放在了 try/catch 块中，这样一来，任何一个方法抛出异常都不会影响到其它委托的调用。

> 注：在单线程中使用事件时，激发事件之前不需要使用一个临时委托变量，本小节所有代码为了与前一小节一致，都使用了临时委托。现实编程中，要看我们定义的类型是否在多线程环境中使用。WINFORM 编程中的 CONTROL 类（及其派生类）在设计

之初就只让它们运行在UI线程中，因此它们激发事件时，都没有考虑多线程的情况。

5.3.4 正确定义一个使用了事件的类

前面说到过，.NET 中的“事件”在框架与客户端代码交互过程中起到了关键作用。那么平常开发过程中，应该怎样去定义一个使用了事件的类型，既能够让该类型的使用者更容易地去使用它，也能够让该类型的开发者更方便地去维护它呢？其实定义一个使用了事件的类型有一套标准方法。下面从命名、激发事件以及组织事件三个方面详细说明：

（1）命名；

前面讲到过，通常情况下，当某件事情发生时，对象内部就会激发事件，通知事件注册者，调用对应的事件处理程序，因此代码中事件的命名最好跟这个发生的事情有关系。比如有一个负责收发 Email 的类，当接收到新的邮件时，应该会激发一个类似叫“NewEmailReceived”的事件，去通知注册了这个事件的其他人，我们最好不要将这个事件定义为“NewEmailReceive”。除了事件本身的命名，事件所属委托类型的命名也同样有标准格式，一般以“事件名+EventHandler”这种格式来给委托命名，前面提到的 NewEmailReceived 事件对应的委托类型名称应该是“NewEmailReceivedEventHandler”。激发事件时会传递一些参数，这些参数一般继承自 EventArgs 类型（后者为.NET 框架预定义类型），以“事件名+EventArgs”来命名，比如前面提到的 NewEmailReceived 事件在激发时传递的参数类型名称应该是“NewEmailReceivedEventArgs”。下面为示例代码：

```
//Code 5-20
private delegate void NewEmailReceivedEventHandler(object sender,NewEmailReceivedEventArgs e);
//define a delegate    NO.1
class EmailManager
{
    //...
    public event NewEmailReceivedEventHandler NewEmailReceived; //define e event member
NO.2
    public EmailManager()
    {
        //...
    }
}
class NewEmailReceivedEventArgs:EventArgs    //define event argument class derived from
EventArgs    NO.3
{
    //...
    public NewEmailReceivedEventArgs()
    {
        //...
    }
}
```

上述代码 Code 5-20 中 NO.1 处定义一个委托，NO.2 处使用该委托定义一个事件，NO.3 处定义一个事件参数类，它派生自 EventArgs 类（通常情况下，EventArgs 为所有事件参数类的基类，如果激发一个事件不带任何参数，那么可以直接使用 EventArgs）。

注：事件的委托签名一般包含两个参数，一个 OBJECT 类型，表示事件发布者（自己），一个为从 EVENTARGS 派生出来的子类型，包含激发事件时所带的参数。

（2）激发事件；

当一个类内部发生某件事情（或者说某个条件成立时），类内部就会激发事件，通知事件的所有注册者。为了便于类型的使用者能够扩展这个类型，比如改变激发事件的逻辑，我们通常使用虚方法去激发事件，比如前面说到的邮件类 EmailManager 中激发 NewEmailReceived 事件应该是这样编写代码：

```
//Code 5-21
private delegate void NewEmailReceivedEventHandler(object sender,NewEmailReceivedEventArgs e);
//define a delegate    NO.1
class EmailManager
{
    //…
    public event NewEmailReceivedEventHandler NewEmailReceived; //define e event member
NO.2
    public EmailManager()
    {
        //…
    }
    private void DoSomething()
    {
        //…
        if(/*…*/)    //NO.4
        {
            NewEmailReceivedEventArgs e = new NewEmailReceivedEventArgs();
            OnNewEmailReceived(e);    //NO.5
        }
    }
    protected void virtual OnNewEmailReceived(NewEmailReceivedEventArgs e)    //NO.6
    {
        if(NewEmailReceived != null)
        {
            NewEmailReceived(this,e);    //NO.7
        }
    }
}
class NewEmailReceivedEventArgs:EventArgs    //define event argument class derived from
EventArgs    NO.3
{
    //…
    public NewEmailReceivedEventArgs()
    {
        //…
    }
```

```
}
```

上述代码 Code 5-21 中，NO.1、NO.2 以及 NO.3 处含义与之前解释相同，NO.4 处当类中某个条件成立时，并没有马上激发事件，而是调用了预先定义的一个虚方法 OnNewEmailReceived（NO.6 处），在该虚方法内部激发事件（NO.7 处），之所以要把激发事件的代码放在一个单独的虚方法中，这是为了让从该类型（EmailManager）派生出来的子类能够重写虚方法，从而改变激发事件的逻辑。下面代码 Code 5-22 定义一个 EmailManager 的派生类 EmailManagerEx：

```
//Code 5-22
class EmailManagerEx:EmailManager
{
    //...
    protected override void OnNewEmailReceived(NewEmailReceivedEventArgs e)
    {
        //...do something here
        if(/*...*/)    //NO.1
        {
            base.OnNewEmailReceived(e);    //NO.2
        }
        else
        {
            // NO.3
        }
    }
}
```

如上代码 Code 5-22 所述，派生类中重写 OnNewEmailReceived 虚方法后，可以重新定义激发事件的逻辑。如果 NO.1 处 if 判断为 true，则正常激发事件（NO.2 处）；否则，不激发事件（NO.3 处）。我们能够在派生类 EmailManagerEx 的 OnNewEmailReceived 虚方法中做许许多多其它的事情，包括示例代码中“取消激发事件”。

虚方法的命名一般为“On+事件名”，另外该虚方法必须定义为 protected，因为派生类中很可能要调用基类的虚方法。

（3）组织事件。

事件类似属性，仅仅只是类型对外公开的一个中介，通过它可以访问类型内部的数据。换句话说，无论事件还是属性，真正存储数据的成员并没有对外公开，比如属性基本都对应有相应的私有字段，每个事件也对应有相应的私有委托成员。我们通过 event 关键字声明的公开事件，经过编译器编译之后，生成的代码类似如下：

```
//Code 5-23
class EmailManager
{
    //...
    private NewEmailReceivedEventHandler _newEmailReceived;    //NO.1
    public event NewEmailReceivedEventHandler NewEmailReceived
    {
        [MethodImpl(MethodImplOptions.Synchronized)]    //NO.2
        add    //NO.3
        {
            _newEmailReceived = Delegate.Combine(_newEmailReceived,value) as
```

```
NewEmailReceivedEventHandler;
            }
            [MethodImpl(MethodImplOptions.Synchronized)]
            remove    //NO.4
            {
                  _newEmailReceived = Delegate.Remove(_newEmailReceived,value) as
NewEmailReceivedEventHandler;
            }
      }
}
```

如上代码 Code 5-23 所示，编译器编译之后，将一个事件分成了两部分，一个私有委托变量_newEmailReceived（NO.1 处）和一个事件访问器 add/remove（NO3 和 NO.4 处），前者类似一个字段，后者类似属性访问器 set/get。可以看出，真正存储事件数据的是私有委托成员_newEmailReceived。

> 注：代码 CODE 5-23 中 NO.2 处[METHODIMPL(METHODIMPLOPTIONS.SYNCHRONIZED)]的作用类似 LOCK(THIS);,为了解决多线程中访问同步问题，这个是官方给出的默认方法，该方法存在缺陷，因为使用 LOCK 加锁时，锁对象不应该是对外公开的，THIS 显然是对外公开的，很有可能出现对 THIS 重复加锁的情况，从而造成死锁。我们可以自己实现事件访问器 ADD/REMOVE，在其中添加自己的 LOCK 块，从而避免使用默认的 LOCK(THIS)。

下图 5-13 为一个类中属性和事件的作用：

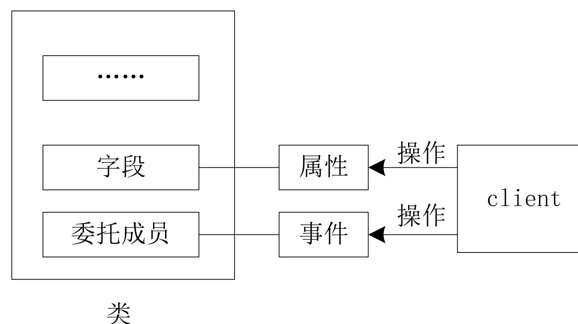


图 5-13 属性和事件的作用

有些类型包含的事件非常多，比如.NET3.5 中 System.Windows.Forms.Control 就包含有 69 个公开事件。一个 Control 类（或其派生类）对象编译后，对象内部就会产生几十个类似代码 Code 5-23 中_newEmailReceived 这样的私有委托成员，这无疑会增加内存消耗，为了解决这个问题，我们一般需要自己定义事件访问器 add/remove，并且自己定义数据结构去存储组织事件数据，不再使用编译器默认生成的私有委托成员。微软在.NET 中的标准做法是：定义一个类似 Dictionary 功能的容器类型 EventHandlerList，专门用来存放委托。一个类型自定义事件访问器 add/remove 后的代码类似如下：

```
//Code 5-24
class EmailManager
```

```
{
    private static readonly object _newEmailReceived;    //NO.1
    private EventHandlerList _handlers = new EventHandlerList();    //NO.2
    public event NewEmailReceivedEventHandler NewEmailReceived
    {
        add
        {
            _handlers.AddHandler(_newEmailReceived,value);    //NO.3
        }
        remove
        {
            _handlers.RemoveHandler(_newEmailReceived,value);    //NO.4
        }
    }
    protected virtual void OnNewEmailReceived(NewEmailReceivedEventArgs e)
    {
        NewEmailReceivedEventHandler newEmailReceived = _handlers[_newEmailReceived] as
NewEmailReceivedEventHandler;    //NO.5
        if(newEmailReceived != null)
        {
            newEmailReceived(this,e);
        }
    }
}
```

如上代码 Code 5-24 所述，自定义事件访问器 add/remove 后，使用 EventHandlerList 来存储事件数据，编译器不再生成默认的私有委托成员，所有的事件数据均存放在_handlers 容器中（NO.2 处），NO.1 处定义了访问容器的 key，NO.3 以及 NO.4 处访问容器，NO.5 处在激发事件之前，先判断容器_handlers 中是否有人注册了该事件。

> 注：自己定义事件访问器还有其它很多作用，比如自己实现线程同步锁、给事件标注[NonSerializable]属性（编译器生成的私有委托成员默认都是 Serializable）等。

上面提到的命名规范、激发事件以及组织事件的方式，这三个是微软给出官方代码中的标准，所有官方源码资料中都遵守了这三个规范。我们平时开发过程中，也应该遵守这些原则，编写出更高质量的代码。

5.4 弱委托

5.4.1 强引用与弱引用

前面章节提到过，一个引用类型对象包括“引用”和“实例”两部分。如果堆中实例至少有一个引用指向它（不管该引用存在于栈中还是堆中），CLR 就不能对其进行内存回收，同时我们一定能够通过引用访问到堆中实例。换句话说，引用与实例是一种“强关联”关系，我

们称这种引用为“强引用”(Strong Reference),堆中对象实例能否被访问完全掌握在程序手中。

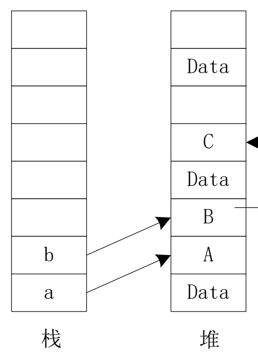


图 5-14 强引用

图 5-14 中 a 是 A 的强引用，b 是 B 的强引用，B 中又存在一个 C 的强引用，只要栈中 a 和 b 存在，堆中 A、B 以及 C 就会一直存在。我们平时编程过程中使用 new 关键字创建一个对象时返回的引用便是强引用，比如“A a=new A();”中，a 就是强引用。

强引用的优点是程序中只要有强引用的存在，就一定能够访问到堆中的对象实例。由于只要有一个强引用存在，CLR 就不会回收堆中的对象实例，这就会出现一个问题：如果我们程序中没有合理地管理好强引用，在该移除强引用的时候没有移除它们，这便会导致堆中的对象实例大量累积，时间一长，就会出现内存不足的情况，尤其当这些对象占用内存比较大的时候。管理好强引用并不是一件容易的事情，通常情况下，强引用在程序运行过程中不断的传递，到最后有些几乎发现不了它们的存在。虽然有时候开发者认为对象已经使用完毕，但是程序中还是会保存这些对象的强引用直到很长一段时间，甚至会一直到程序运行结束。在事件编程中，委托的 Target 成员，就是对事件注册者的强引用，如果事件注册者没有注销事件，这个 Target 强引用便会一直存在，堆中的事件注册者内存就一直不会被 CLR 回收，这对开发人员来讲，几乎是很难发觉的。

> 注：像“A A = NEW A();”中的 A 称为“显式强引用（EXPLICIT STRONG REFERENCE）”，类似委托中包含的不明显的强引用，我们称之为“隐式强引用（IMPLICIT STRONG REFERENCE）”。

对于“强引用”，有一个概念与之对应，即“弱引用”。弱引用与对象实例之间属于一种“弱关联”关系，跟强引用与对象实例的关系不一样，就算程序中有弱引用指向堆中对象实例，CLR 还是会把该对象实例当做回收目标。程序中使用弱引用访问对象实例之前必须先检查 CLR 有没有回收该对象内存。换句话说，当堆中一个对象实例只有弱引用指向它时，CLR 可以回收它的内存。使用弱引用，堆中对象能否被访问同时掌握在程序和 CLR 手中。

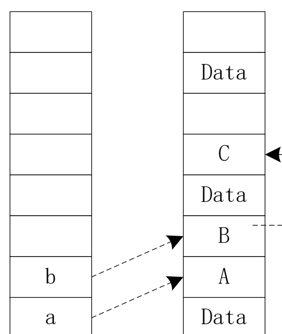


图 5-15 弱引用

图 5-15 中 a 是 A 的弱引用，b 是 B 的弱引用，B 中又包含一个 C 的弱引用，不管 a 和 b 是否存在，堆中 A、B 以及 C 都有可能成为 CLR 的回收目标。

创建一个弱引用很简单，使用 WeakReference 类型，给它的构造方法传递一个强引用作为参数，代码如下：

```
//Code 5-25
class A
{
    public A()
    {
        //…
    }
    public void DoSomething()
    {
        Console.WriteLine("I am OK");
    }
    //…
}
class Program
{
    static void Main()
    {
        A a = new A();
        WeakReference wr = new WeakReference(a);    //NO.1
        a = null;    //NO.2
        //do something else
        A tmp = wr.Target;    //NO.3
        if(wr.IsAlive)    //NO.4
        {
            tmp.DoSomething();    //NO.5
            tmp = null;
        }
        else
        {
            Console.WriteLine("A is dead");
        }
    }
}
```

代码 Code 5-25 中创建了一个 A 对象的弱引用（NO.1 处），然后马上将它的临时强引用 a 指向 null（NO.2 处），此时只有一个弱引用指向 A 对象。程序运行一段时间后（代码中 do something 处），当需要通过弱引用 wr 访问 A 对象的时候，我们必须先检查 CLR 有没有回收它的内存（NO.4 处），如果没有，我们正常访问 A 对象；否则，我们不能再访问 A 对象。

在编程过程中，我们很难管理好强引用，从而造成不必要的内存开销。尤其前面讲到的“隐式强引用”，在使用过程中不易发觉它们的存在。使用弱引用，CLR 回收堆中对象内存不再根据程序中是否有弱引用指向它，因此程序中有没有多余的弱引用指向某个对象对 CLR 回收

该对象内存没有任何影响。弱引用特别适合用于那些对程序依赖程度不高的对象，也就是那些对象生命期不是主要由程序控制的对象。比如事件编程中，事件发布者对事件注册者的存在与否不是很关心，如果注册者在，那就激发事件并通知注册者；如果注册者已经被 CLR 回收内存，那么就不通知它，这完全不会影响程序的运行。

5.4.2 弱委托定义

前面讲到过，委托包含两个部分：一个 Object 类型 Target 成员，代表被调用方法的所有者，如果方法为静态方法，Target 为 null；另一个是 MethodInfo 类型的 Method 成员，代表被调用方法。由于 Target 成员是一个强引用，所以只要委托存在，那么方法的所有者就会一直在堆中存在而不能被 CLR 回收。如果我们将委托中的 Target 强引用换成弱引用的话，那么不管委托存在与否，都不会影响方法的所有者在堆中内存的回收。这样一来，我们在使用委托调用方法之前需要先判断方法的所有者是否已经被 CLR 回收。我们称将 Target 成员换成弱引用之后的委托为“弱委托”，弱委托定义如下：

```
//Code 5-26
class WeakDelegate
{
    WeakReference _weakRef;    //NO.1
    MethodInfo _method;    //NO.2
    public WeakDelegate(Delegate d)
    {
        _weakRef = new WeakReference(d.Target);
        _methodInfo = d.Method;
    }
    public object Invoke(param object[] args)
    {
        object obj = _weakRef.Target;
        if(_weakRef.IsAlive)    //NO.3
        {
            return _method.Invoke(obj,args);    //NO.4
        }
        else
        {
            return null;
        }
    }
}
```

如上代码 Code 5-26 所示，我们定义了一个 WeakDelegate 弱委托类型，它包含一个 WeakReference 类型_weakRef 成员（NO.1 处），它是一个弱引用，指向被调用方法的所有者，还包含一个 MethodInfo 类型_method 成员（NO.2 处），它表示委托要调用的方法。我们在弱委托的 Invoke 成员方法中，先判断被调用方法的所有者是否还在堆中（NO.3 处），如果在，我们调用方法，否则返回 null。

弱委托将委托与被调用方法的所有者之间的关系由“强关联”转换成了“弱关联”，方法的所有者在堆中的生命期不再受委托的控制，下图 5-16 显示弱委托的结构：

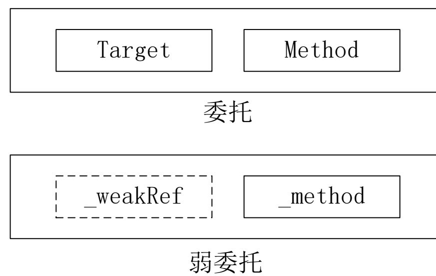


图 5-16 弱委托结构

如上图 5-16 所示，图中上部分表示一个普通委托的结构，下部分表示一个弱委托的结构，虚线框表示弱引用，堆中实例的内存不再受该弱引用影响。

注：本小节示例代码中的 WeakDelegate 类型并没有提供类似 Delegate.Combine 以及 Delegate.Remove 这样操作委托链表的方法，当然也没有弱委托链表的功能，这些功能可以仿照单向链表的结构去实现，把每个弱委托都当作链表中的一个节点。请参照 5.1.2 小节中讲到的单向链表。

5.4.3 弱委托使用场合

我们在使用事件编程时，如果一个事件注册者向事件发布者注册了一个事件，那么发布者就会对注册者保存一个强引用。如果事件注册者未正确地注销事件，那么发布者的委托链表中就一直包含一个对该注册者的强引用，这样一来，注册者在堆中的内存永远都不会被 CLR 回收，如果这样的注册者属于大对象或者数目众多，很轻易就会造成堆中内存不足。弱委托就恰好能够解决这个问题，我们可以将事件编程中用到的委托替换为弱委托，那么事件发布者与事件注册者的关系如下图 5-17：

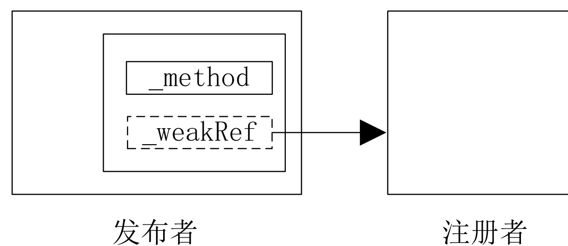


图 5-17 弱委托在事件编程中的应用

如上图 5-17 所示，事件发布者中不再保留对事件注册者的强引用。当发布者激发事件时，先判断注册者是否存在（堆中内存是否被 CLR 回收），如果存在，就通知注册者；否则将对应弱委托从链表中删除。

注：弱委托链表请读者自己去实现。

5.5 本章回顾

委托与事件几乎出现在.NET 编程的每一个地方，它们是.NET 中最重要的知识点之一。程

序的运行就是一个个调用与被调用的过程，而委托的主要作用就是“调用方法”，它是衔接调用者与被调用者的桥梁。本章开头介绍了.NET 中委托的概念和组成结构，同时介绍了委托链表以及它的“不可改变”特性；之后介绍了委托与事件的关系，我们明白了事件是一种特殊的委托对象；紧接着讲到了.NET 中使用事件编程时需要关注的几条注意事项，它们是在事件编程过程中常遇到的陷阱；章节最后还提到了“弱引用”和“弱委托”的概念以及它们的实现原理，“弱委托”是解决内存泄露的一种有效方法。

本章提到了委托的三个作用：第一，它允许把方法作为参数，传递给其它的模块；第二，它允许我们同时调用多个具有相同签名的方法；第三，它允许我们异步调用任何方法。这三个作用奠定了委托在.NET 编程中的绝对重要地位。

5.6 本章思考

1.简述委托包含哪两个重要部分。

A：委托包含两个重要组成：Method 和 Target，分别代表委托要调用的方法和该方法所属的对象（如果为静态方法，则 Target 为 null）。

2.怎样简单地说明委托的不可改变特性？

A：对委托的所有操作，均需要将操作后的结果在进行赋值，比如使用“+=”、“-=”将操作后的结果赋值给原委托变量。这说明对委托的操作均不能改变委托本身。

3.“事件是委托对象”是否准确？

A：准确。.NET 中的事件是一种特殊的委托对象，即在定义委托对象时，在声明语句前增加了“event”关键字。事件的出现确保委托的调用只能发生在类型内部。

4.为什么说委托是.NET 中的“重中之重”？

A：因为程序的运行过程就是方法的不断调用过程，而委托的作用就是“调用方法”，它不仅能够将方法作为参数传递，还能同时（同步或异步）调用多个具有相同签名的方法。

5.弱委托的关键是什么？

A：弱委托的关键是弱引用，弱委托是通过弱引用实现的。

第六章 线程的升级：异步编程模型

所谓“异步编程”，言下之意就是方法的调用与当前执行流程同时进行。这可以使用我们熟悉的“多线程”去实现，但.NET 中另外提供一种技术，即“异步编程模型”，它定义了一种在其它线程中调用方法的规范。本章主要介绍异步编程模型的必要性、怎样使用委托实现异步编程以及在异步编程过程中需要注意的一些常见陷阱等。

6.1 异步编程的必要性

6.1.1 同步调用与异步调用

前面章节讲到过，程序执行的过程就是一系列方法被调用的过程。理论上讲，我们在程序中编写的所有方法均由别人（直接或间接）调用。通常情况下，调用一个方法都符合这样一个规律：调用线程开始调用方法 A 后，在 A 返回之前，调用线程得不到程序执行的控制权，也就是说，方法 A 后面的代码是不可能执行的，直到 A 返回为止，这种调用方式称之为“同步调用”（Synchronous Call），相反，如果调用在返回之前，调用线程依旧保留控制权，能够继续执行后面的代码，那么称这种调用方式为“异步调用”（Asynchronous Call），如下图 6-1 所示：

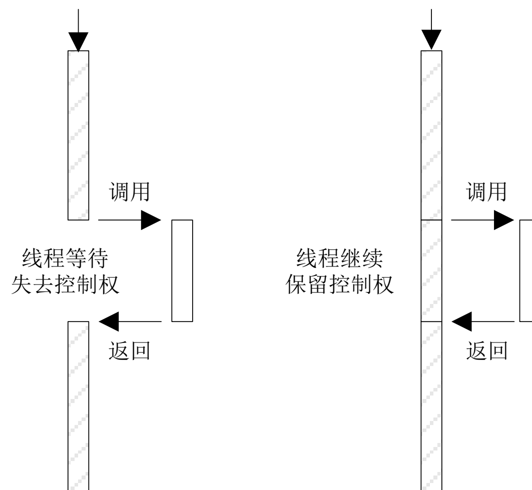


图 6-1 同步调用与异步调用

如上图 6-1 所示，左边为同步调用，调用线程被阻塞，右边为异步调用，调用线程没有被阻塞。正常情况下，方法的调用都是同步的，然而有一些特殊方法内部使用了特殊机制，比如使用了多线程去处理任务，那么这些方法的调用就可以说是异步的，如以下代码所示：

```
//Code 6-1
class A
{
    //...
    public void DoSomething()
```

```
    {
        //do something here
        Threading.Thread.Sleep(10000);    //NO.1
    }
    public void DoSomething2()
    {
        Thread th = new Thread(new ThreadStart(th_proc));
        th.Start();    //NO.2
    }
    private void th_proc()
    {
        //do something here
        Threading.Thread.Sleep(10000); //NO.3
    }
}
class Program
{
    static void Main()
    {
        A a = new A();
        a.DoSomething(); //NO.4
        a.DoSomething2(); //NO.5
        string b = "break";
    }
}
```

如上代码 Code 6-1 所示，A 类中的 DoSomething 方法为一个普通方法，而 Domething2 方法中使用了线程处理耗时操作（NO.2 处）。在客户端使用 A 类对象时，NO.4 处属于同步调用，a.DoSomething 返回之后才能继续执行下面的代码，而 NO.5 处属于异步调用，a.DoSomething2 返回之前，就能执行后面的代码。注意 A 类中的 DoSomething2 方法的实现，前提是 A 类必须是线程安全的（Thread-Safety），因为我们每调用一次 a.DoSomething2 时，就会开辟一个新线程去执行 a 对象的 th_proc，这就出现了多个线程同时操作一个对象的可能。对象的“线程安全”是异步编程中的重点，后面也会讲到。

6.1.2 异步调用的优点

很明显，异步调用不会阻塞调用线程，单位时间内可以处理更多的任务，在 Winform 中异步调用的作用尤其明显。如果我们在 UI 线程中同步调用一个耗时方法，那么窗体界面会出现反应迟钝、卡以及不刷新等现象，主要原因就是同步调用的方法阻塞了 UI 线程，Windows 消息不能及时被处理（详见第八章）。

异步调用过程中，方法的执行不在调用线程中。也就是说，真正执行任务的线程不再是当前调用线程，那么这就会出现一系列问题，比如当前调用线程如何知道任务什么时候执行结束，任务执行的结果如何？当前调用线程怎样给执行任务的线程提供一些额外的参数？再或者当前调用线程怎样捕获执行任务时可能抛出的异常？这些问题本章后面都将会给出答案。

注：其它有些地方称本书中的“同步调用”为“阻塞调用”，称本书中的“异步调用”为

“非阻塞调用”，其实意思是一样的，命名不同而已。

6.2 委托的异步调用

6.1.1 小节中给出的示例代码中，a.DoSomething2 属于特殊方法中的一种，方法内部实现采用了多线程处理任务。在实际编程中，大部分方法并没有采用某些特殊技术去刻意具备“异步调用”的功能，也就是说，大部分方法还是在当前调用线程中执行、在当前调用线程中处理任务。本书中第五章讲委托的时候说到过，委托的一个作用就是可以异步调用与它关联的方法，没错，我们既可以选择同步调用一个委托，也可以选择异步调用一个委托。理论上讲，任何一个方法，通过委托包装后，都可以实现异步调用。本节将详细说明怎样使用委托去异步调用一个普通方法。

6.2.1 BeginInvoke 与 EndInvoke

.NET 编译器为我们定义的每个委托类型自动生成了两个方法：BeginInvoke 和 EndInvoke，这两个方法专门用来负责异步调用委托。如果一个委托定义如下：

```
//Code 6-2
public delegate int DelegateName(int arg1,string arg2);
```

那么编译器为它生成的 BeginInvoke 和 EndInvoke 方法签名如下：

```
//Code 6-3
public IAsyncResult BeginInvoke(int arg1,string arg2,AsyncCallBack callback,object asyncState);
public int EndInvoke(IAsyncResult ar);
```

如上代码 Code 6-3 所示，BeginInvoke 表示开始异步调用委托，它带有四个参数（不同的委托有不同的参数），前面两个跟委托签名中的一致，后面的 callback 表示异步调用完毕后需要调用的回调方法（该参数可以为空，后面详细讲到），最后的 asyncState 表示给异步调用过程传递的一个附加参数（可以为空）。BeginInvoke 返回一个 IAsyncResult 接口类型，它可以唯一区分一个异步调用过程，BeginInvoke 一执行马上返回，不会阻塞调用线程。EndInvoke 表示结束对委托的异步调用，但这并不意味着它可以中断异步调用过程，如果异步调用还未结束，EndInvoke 只能等待，直到异步调用过程结束。另外，如果委托带有返回值，我们必须通过 EndInvoke 获得这个返回结果，所以实例代码 Code 6-3 中 EndInvoke 返回值为 int 类型。

下面示例代码演示怎样通过委托来异步调用方法：

```
//Code 6-4
class A
{
    //…
    public string CombineString(string first,string second)
    {
        //Threading.Thread.Sleep(10000);
        return first + “-” + second;
    }
}
class Program
{
```

```
    delegate string CombineStringDelegate(string first,string second);
    static void Main()
    {
        A a = new A();
        CombineStringDelegate del = a.CombineString;    //NO.1
        //synchronous call below
        string result = del("hello","world");    //NO.2

        //Asynchronous call below
        IAsyncResult ar1= del.BeginInvoke("hello","world",null,null);    //NO.3
        IAsyncResult ar2 = del.BeginInvoke("my name is",".net",null,null);    //NO.4
        //…do something else here
        string result2 = del.EndInvoke(ar2);    //NO.5
        string result1 = del.EndInvoke(ar1);    //NO.6
        Console.WriteLine(result1);
        Console.WriteLine(result2);
    }
}
```

如上代码 Code 6-4 所示，NO.1 处定义一个委托，将它与 a.CombineString 方法关联起来，NO.2 处同步调用委托，a.ComebineString 方法运行在当前调用线程中，NO.3 处开始第一次异步调用委托，BeginInvoke 立刻返回后，继续开始第二次异步调用委托（NO.4 处），ar1 与 ar2 分别区分两次异步调用过程，最后通过 EndInvoke 方法分别结束两次异步调用过程（NO.5. NO.6 处）。委托异步调用开始后，系统会在线程池中找到一个空闲的线程去执行委托，所以两次连续异步调用委托实质上相当于开启了两个线程去执行 a.CombineString 方法，由于线程之间的调度等因素，先开始的异步调用过程不一定先结束，同理，后开启的异步调用过程不一定后结束。NO.5 和 NO.6 处的 EndInvoke 方法与 Thread.Join()方法功能类似，它会阻塞当前调用线程，直到对应的异步调用过程结束。

注意本小节示例代码中的 A 类对象 a 必须是线程安全的，因为通过委托可以同时开启多次异步调用，每次 a.CombineString 都相当于运行在一个单独的线程中，这就相当于多个线程会同时访问 a 对象。另外，当前调用线程为了获得委托执行结果（a.CombineString 方法的返回值），所以不得不在当前线程中调用 EndInvoke 去获取委托异步调用的执行结果，而 EndInvoke 还是会阻塞当前调用线程，其实可以解决这一问题，关键在于怎么去利用 BeginInvoke 方法的 callback 参数，下一小节介绍委托的 AsyncCallback 类型参数。下图 6-2 显示在同一线程中调用 BeginInvoke 方法和 EndInvoke 方法时，线程的运行流程：

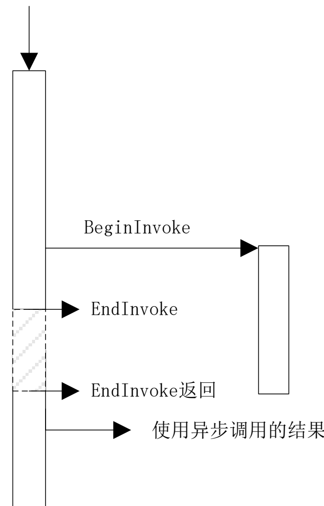


图 6-2 在同一线程中调用 BeginInvoke 和 EndInvoke

如上图 6-2 所示，图中白色部分代表线程拥有控制权，虚线阴影部分代表线程失去控制权。调用 BeginInvoke 后，调用线程还拥有控制权，调用 EndInvoke 后，如果异步调用还未结束，那么当前线程失去控制权，也就是 EndInvoke 会阻塞当前调用线程（图中正是这种情况），但是如果异步调用已经结束，那么 EndInvoke 会马上返回，当前调用线程依旧保留控制权，即此时 EndInvoke 不会阻塞当前调用线程。

> 注：并不是说 ENDINVOKE 方法不能与 BEGININVOKE 方法在同一个线程中调用，否则失去了意义，原因很简单，比如我们要调用两次 A.COMBINESTRING 方法，每次耗时 10 秒钟，如果直接同步调用的话，总耗时两者相加需要 20 秒钟，但是如果采用本小节示例代码中的做法，总耗时（取两者最大值）只需要 10 秒钟。另外，BEGININVOKE 方法与 ENDINVOKE 方法要用在同一个委托上，也就是我们不能将第一个委托的 BEGININVOKE 方法返回的 IASYNCRESULT 值传给第二个委托的 ENDINVOKE 方法作为参数，否则会抛出异常。

6.2.2 AsyncCallback

委托的 BeginInvoke 方法包含一个 AsyncCallback 委托类型参数，如果该参数不为空，当异步调用执行完毕后，系统会自动调用该委托，通知调用线程异步调用已结束，

```
//Code 6-5
class A
{
    //…
    public string CombineString(string first,string second)
    {
        //Threading.Thread.Sleep(10000);
        return first + “-” + second;
    }
}
class Program
```

```
{
    static delegate string CombineStringDelegate(string first,string second);
    static void Main()
    {
        A a = new A();
        CombineStringDelegate del = a.CombineString;    //NO.1

        //Asynchronous call below
        de.BeginInvoke(“hello”,”world”,WhenComleted,del);    //NO.2
    }
    static void WhenCompleted(IAsyncResult ar)
    {
        CombineStringDelegate del = ar.AsyncState as CombineStringDelegate; //NO.3
        string result = del.EndInvoke(ar);    //NO.4
        Console.WriteLine(result);
    }
}
```

如上代码 Code 6-5 所示，在 NO.2 处调用 BeginInvoke 时，给它的第三个参数 callback 传递了一个委托 WhenCompleted，当委托异步调用完成时，系统会自动调用该委托，给它的第四个参数 asyncState 参数传递了一个委托 del，这个参数就是前面说到的附加参数。BeginInvoke 调用后，调用线程不再需要使用 IAsyncResult 类型变量去跟踪它的状态，因为所有的一切信息我们在回调方法 WhenCompleted 中都可以知道，NO.3 处将附加参数还原成 CombineStringDelegate 类型，然后在 WhenCompleted 中调用 EndInvoke 结束异步调用过程，返回委托异步调用的执行结果 result，最终输出。我们可以看到之前在调用线程中所做的工作全部转移到了 WhenCompleted 这个回调方法中，调用线程只负责调用 BeginInvoke 方法去开启一个异步调用过程，不再去关心其余的工作，真正地实现了零阻塞，没错，我们可以在 BeginInvoke 方法的 callback 回调方法中去获取异步调用的执行结果，而这跟调用线程再没有任何关系。下图 6-3 显示 EndInvoke 方法不在当前线程中调用时，线程的运行流程：

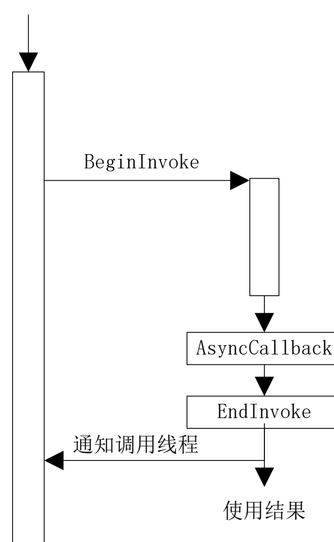


图 6-3 EndInvoke 不在当前调用线程中执行

如上图 6-3 所示，在回调方法 AsyncCallback 中执行 EndInvoke 方法，获得委托异步调用的结果，我们既可以直接使用该结果，也可以通知调用线程。显而易见，图 6-3 中调用线程从来都不会

失去执行控制权。

WhenCompleted 回调方法可以同时作为多个 BeginInvoke 方法的 callback 参数，这就像一个方法可以同时被多个线程调用：

```
//Code 6-6
class A
{
    //..
    public void DoSomething()
    {
        Thread th1 = new Thread(new ParameterizedThreadStart(th_proc)); //NO.1
        Thread th2 = new Thread(new ParameterizedThreadStart(th_proc)); //NO.2
        th1.Start(1);
        th2.Start(2);
    }
    private void th_proc(object threaded)
    {
        if((int)threaded == 1) //NO.3
        {
            //do something …
        }
        if((int)threadid == 2) //NO.4
        {
            //do something …
        }
    }
}
```

如上代码 Code 6-6 所示，th_proc 方法可以同时在多个线程中执行（NO.1 和 NO.2 处），我们可以使用 threadid 参数来区分当前是哪个线程（NO.3 和 NO.4 处），同理 WhenCompleted 方法中的 IAsyncResult 类型参数也可以区分当前是哪个异步调用过程。

> 注：BEGININVOKE 方法的 CALLBACK 参数所代表的回调方法，并不是在当前线程中调用，因此在编写回调方法 WHENCOMPLETED 时需要注意一些资源的线程安全问题。

6.2.3 处理异步调用时的异常

同步调用委托时，方法抛出来的异常直接可以由当前调用线程捕获，异步调用委托时，由于方法实际运行在其它线程中（线程池中的某一线程，非当前调用线程），因此当前线程捕获不了异常，那么我们怎样知道异步调用过程是否有异常呢？答案就在 EndInvoke 方法上，如果异步调用过程有异常，那么该异常会在我们在调用 EndInvoke 方法时抛出，所以我们在调用 EndInvoke 方法时，一定要把它放在 try/catch 块中，6.2.2 小节中的 WhenCompleted 方法应该这样写：

```
//Code 6-7
static void WhenCompleted(IAsyncResult ar)
{
    CombineStringDelegate del = ar.AsyncState as CombineStringDelegate; //NO.3
```

```
        string result = "";
        try
        {
            result =  del.EndInvoke(ar);  //NO.4
        }
        catch
        {
            Console.WriteLine("exception throw");
        }
        Console.WriteLine(result);
}
```

如上代码 Code 6-7 所示，EndInvoke 方法的调用放在 try/catch 块中后，异步调用过程中抛出的异常均由该 try/catch 块捕获。

> 注：由于 WHENCOMPLETED 回调方法由线程池中的线程调用，如果在它里面抛出异常而没有被处理，将很可能会造成系统崩溃。

6.2.4 异步调用的应用

现在假设一个服务器每隔 2 秒钟就会接收到一个用户请求，服务器需要 10 秒钟去处理这个请求，然后将处理结果返回给用户需要 1 秒钟，那么，使用同步调用编写代码是这样的：

```
//Code 6-8
class Server
{
    //...
    public string ReceiveRequest(ref int clientID)  //receive client's request
    {
        //...
    }
    public string DealRequest(string askCode)  //deal client's request
    {
        //...
    }
    public void SendResponse(string response, int clientID)  //send response to client
    {
        //...
    }
}
class Program
{
    static void Main()
    {
        string askCode;
        int clientID;
        string response;
```

```
        Server s = new Server();
        while(flag)
        {
            askCode = s.ReceiveRequest(ref clientID); //NO.1      2 seconds
            response = s.DealRequest(askCode); //NO.2      10 seconds
            s.SendResponse(response,clientID); //NO.3      1 second
        }
    }
}
```

如上代码 Code 6-8 所示，Server 类表示服务器逻辑处理类，采用同步调用的方式分别调用 ReceiveRequest 接收用户请求（NO.1 处）、调用 DealRequest 处理用户请求（NO.2 处）、调用 SendResponse 发送处理结果（NO.3 处）。如果要处理 5 个用户请求，共需要耗时 5*（2+10+1）秒钟，如果把处理用户请求的方法由同步调用改为异步调用，Program 类的代码如下：

```
//Code 6-9
class Program
{
    delegate string DealRequestDelegate(string askCode); //NO.1
    static void Main()
    {
        string askCode;
        int clientID;
        string response;
        Server s = new Server();
        DealRequestDelegate d = s.DealRequest;
        while(flag)
        {
            askCode = s.ReceiveRequest(ref clientID); //NO.2      2 seconds
            d.BeginInvoke(askCode,DealCompleted,new object[]{d,clientID,s}); //NO.3
        }
    }
    static void DealCompleted(IAsyncResult ar)
    {
        object[] objs = ar.AsyncState as object[];
        DealRequestDelegate d = objs[0] as DealRequestDelegate; //NO.4
        int clientID = (int)(objs[1]);    //NO.5
        Server s = objs[2] as Server;    //NO.6
        string response;
        try
        {
            response = d.EndInvoke(ar);    //NO.7
        }
        catch
        {
            //log here
        }
```

```
            s.SendResponse(response,clientID);    //NO.8
        }
}
```

如上代码 Code 6-9 所示，NO.1 处定义了一个与 DealRequest 方法签名相同的委托，用作异步调用，每次接收到一个用户请求后（NO.2 处），马上开启一个异步过程（NO.3 处）去处理请求，调用时给 BeginInvoke 方法传递了三个参数，第一个是委托自带的参数，第二个是回调方法，第三个是附加参数，我们将委托 d、客户端 ID 以及服务类对象 s 作为一个整体（object[]）传给了附加参数。每次 BeginInvoke 方法调用后马上返回，不会阻塞 while 循环。当异步调用过程结束后，系统自动调用 DealCompleted 方法，我们在回调方法 DealCompleted 中还原附加参数（object[]类型，NO.4. NO.5 以及 NO.6 处），然后通过 EndInvoke 方法获得异步处理请求的结果 response（NO.7 处），最后将处理结果发送给对应客户端（NO.8 处）。如果不考虑线程调度等因素，这种方法处理 5 个用户请求共耗时 2*5+11 秒钟。下图 6-4 显示异步调用与同步调用的时间消耗情况：

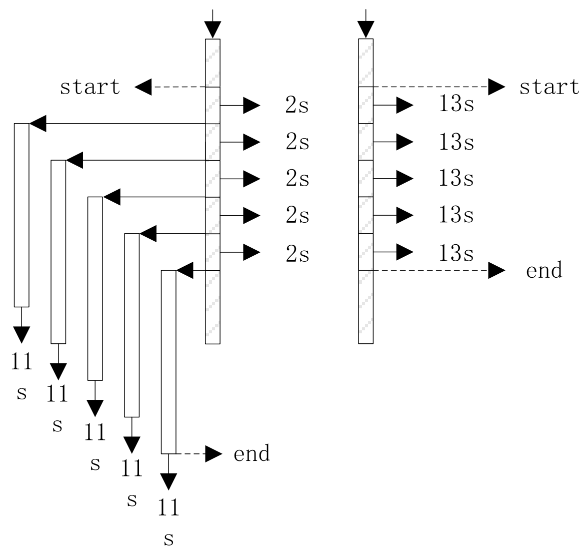


图 6-4 异步调用与同步调用总耗时情况

如上图 6-4 所示，左边显示异步调用处理 5 个客户请求的总耗时，右边显示同步调用处理 5 个客户请求的总耗时。

> 注：本小节示例代码中的 SERVER 类必须是线程安全的，因为多个异步调用时，多个线程会同时调用 SERVER.DEALREQUEST 方法，也就是多个线程同时访问 SERVER 类对象。另外本小节中计算耗时都不考虑线程调用等因素。

6.3 非委托的异步调用

6.3.1 异步方法

6.2 节中介绍了怎样使用委托去异步调用一个方法，首先我们需要定义一个与方法签名相同的委托类型，然后将委托与方法关联起来，使用委托的 BeginInvoke 和 EndInvoke 方法分别开启异步调用和结束调用，使用委托几乎可以异步调用任何一个方法。有一些类似硬盘访问、Socket 访问以及数据库访问等过程大多数情况下耗时都是非常长的，因此在.NET 中有关这些

操作的类型，基本上每个操作方法都会提供两个版本：一种同步方法，即前面讲到过的普通方法，它们很有可能阻塞调用线程；另一种就是异步方法，方法内部采用了特殊机制去实现（类似 6.1.1 小节中 A.DoSomething2），异步方法不再需要我们通过委托去实现异步调用，因为它本身的调用就是异步的，异步方法不会阻塞调用线程。

.NET 中提供异步方法的类型有诸如 Stream（或其派生类）、Socket（或其派生类）以及访问数据库的比如 SqlConnection 类型等等。它们的使用方式跟委托的 BeginInvoke 和 EndInvoke 方法类似，只是命名有所差别，基本上都是“Begin+操作”和“End+操作”这种格式，比如 FileStream.BeginRead 表示开始一个异步读操作，FileStream.EndRead 表示结束异步读操作。下面代码演示 FileStream 类使用异步方法读取文件：

```
//Code 6-10
class Program
{
	static void Main()
	{
		FileStream fs = new FileStream(…);
		byte[] buffer = new byte[1024];
		fs.BeginRead(buffer,0,1024,ReadCompleted,new object[]{fs,buffer});	//NO.1
		//…do something else
	}
	static void ReadCompleted(IAsyncResult ar)
	{
		object[] objs = ar.AsyncState as object[];
		FileStream fs = objs[0] as FileStream;
		byte[] b = objs[1] as byte[];
		int realRead;
		try
		{
			realRead = fs.EndRead(ar);	//NO.2
		}
		catch
		{
		}
		//use b or notify the calling thread
		//..
	}
}
```

如上代码 Code 6-10 所示，NO.1 处 FileStream 对象调用 BeginRead 方法开始异步读取数据，用法类似委托的 BeginInvoke，前面三个参数跟 FileStream.Read 方法（同步版本）相同，第四个参数是一个回调方法，当读取数据的异步过程完成后，系统自动调用该回调方法，第五个参数是附加参数，我们在回调方法中可以使用它。在 ReadCompleted 回调方法中，我们调用 FIleStream 对象的 EndRead 方法结束异步读取数据的过程（NO.2 处）。

注意一个类型如果提供了异步方法，那么它都会提供一个与之对应的同步版本方法，比如 FileStream.BeginRead 对应有 FileStream.Read 方法，Socket.BeginReceive 对应有 Socket.Receive 方法。如果把一个类型的异步版本方法类比为委托的 BeginInvoke/EndInvoke，那么可以把同步版本的方法类比为委托的 Invoke 方法。异步调用过程中，所有的任务处理都发生在另外的线

程之中（非当前调用线程），包括 AsyncCallback 所代表的的回调方法。

6.3.2 FileStream.BeginRead/FileStream.BeginWrite

本小节将以 FileStream.BeginRead 和 FileStream.BeginWrite 为例，说明异步方法的使用。下面代码分别演示了使用同步方法和异步方法两种方式将一个流数据拷贝到另外一个流中：

```
//Code 6-11
class Program
{
    //use FileStream.Read and FileStream.Write
    static void Main()
    {
        FileStream fSrc = new FileStream(…); //NO.1
        FileStream fDes = new FileStream(…);    //NO.2
        byte[] b = new byte[1024];
        int realRead = 0;
        while(true)
        {
            realRead = fSrc.Read(b,0,1024);
            fDes.Write(b,0,realRead);
            if(realRead != 1024)
                break;
        }
        fSrc.Close();
        fDes.Close();
    }
    //use FileStream.BeginRead and FileStream.BeginWrite
    static void Main()
    {
        FileStream fSrc = new FileStream(…); //NO.3
        FileStream fDes = new FileStream(…);    //NO.4
        byte[] b1 = new byte[1024];
        byte[] b2 = new byte[1024];
        byte[] readed = b1;
        int realRead = 0;
        IAsyncResult ar1 = fSrc.BeginRead(readed,0,1024,null,null);
        IAsyncResult ar2;
        while(true)
        {
            realRead = fSrc.EndRead(ar1);
            ar2 = fDes.BeginWrite(readed,0,realRead,null,null); //NO.5
            if(realRead == 1024)
            {
                readed = (readed == b1)?b2:b1;
                ar1 = fSrc.BeginRead(readed,0,1024,null,null); //NO.6
            }
```

```
                fDes.EndWrite(ar2);
                if(realRead != 1024)    //NO.7
                    break;
            }
            fSrc.Close();
            fDes.Close();
        }
}
```

如上代码 Code 6-11 所示，使用 Read 和 Write 方法循环将数据从一个流中拷贝到另外一个流中，如果每次读操作耗时 1 秒钟，每次写操作耗时 2 秒钟，总共需要读写 10 次的话，那么总共需要耗时(1+2)*10 秒钟。使用 BeginRead/EndRead 和 BeginWrite/EndWrite 循环将数据从一个流中拷贝到另外一个流中时，由于后 9 次读操作和前 9 次写操作同时进行，因此这 9 次读写操作共耗时 Max(1,2)*9 秒钟，再加上第一次读操作耗时 1 秒、最后一次写操作耗时 2 秒，所以整个拷贝过程共耗时 Max(1,2)*9+1+2 秒钟。两种不同拷贝方式的耗时情况见下图 6-5（图中只显示了其中三次读写操作）：

图 6-5 同步拷贝与异步拷贝耗时情况

如上图 6-5 所示，图中上半部分表示调用 FileStream.Read 和 FileStream.Write 进行三次数据拷贝时的耗时情况，读操作和写操作是串行进行的，三次数据拷贝共耗时 3*3 秒，图中下半部分表示调用 FileStream.BeginRead 和 FileStream.BeginWrite 进行三次数据拷贝时的耗时情况，除了第一次读操作和最后一次写操作以外，其余读写操作都是成对并列进行的，三次数据拷贝共耗时 Max(1,2)*2+1+2 秒。

> 注：本小节中计算拷贝耗时均不考虑线程调度等因素，也就是说，实际中可能异步拷贝耗时更长。

6.4 异步编程时的注意事项

虽然在程序中适当采用异步编程模式，能够提高系统性能、增加系统吞吐量，但是异步编程使程序的运行流程不再呈“直线流水”型，它打乱了代码的调用关系，代码调用逻辑也变得复杂难以理解。在异步编程过程中，需要注意以下三点：

（1）异步编程中，除了调用线程外，几乎看不见其它任何线程，但是，我们编写的绝大部分代码却都是运行在其它线程中，比如使用委托异步调用方法时，这个方法就运行在线程池中的某个线程中，还有如果我们调用 BeginInvoke 或者 FileStream.BeginRead 方法时给定了 AsyncCallback 类型参数，那么这个参数代表的回调方法也会运行在其它线程中。只要涉及到多线程同时访问同一个资源时，我们就应该注意“线程安全”问题。在 Winform 开发中，上述这些方法中都不应该包含直接访问界面的代码。

（2）系统维护的线程池是有大小限制的，我们不能无限制地开启异步调用过程，这很有可能耗尽线程池资源，从而达不到想要的效果。因此，在有些场合，数量众多、耗时太长的操作最好不要使用异步编程，而改成直接使用多线程的方法（Worker-Thread）。

（3）异步调用过程中，如果我们需要使用异步执行的结果，最好等在 EndInvoke 方法返回之后，因为该方法返回之后，才能说明异步调用已结束，否则异步调用就有可能还没有完成。EndInvoke 方法不仅能够返回异步执行的结果，它的返回还是异步执行过程完成的标志。

6.5 本章回顾

本章开头介绍了“异步编程”的必要性，它能够提高系统数据处理的性能、同时有着比多线程更低机器损耗的优势；随后介绍了两种异步编程的方式：一种借助委托，可以异步调用任何方法，另外一些方法本身就使用了某些机制具备异步调用的功能；之后还提到了异步编程时常用到的两个方法和一个委托类型：BeginInvoke、EndInvoke 方法以及 AsyncCallback 委托类型；本章最后提到了使用异步编程的缺点：它打破了程序常规直线流水型的运行方式，使原本清晰易懂的代码运行流程变得复杂难以理解，这也就给编程造成了一定困难。

注： 这里说的BEGININVOKE和ENDINVOKE也包含了非委托异步编程中以”BEGIN”和”END”开头的这一类方法，如 FILESTREAM.BEGINREAD、FILESTREAM.ENDREAD 等。

6.6 本章思考

1.异步调用开始后，什么时候才能使用异步执行的结果？

A：最好在 EndInvoke()方法返回之后才能使用异步执行的结果，其它时候不能保证异步调用已完成。

2.委托的异步调用开始后（即调用 BeginInvoke 方法后），EndInvoke 方法是否可以在同一线程中调用？

A：可以。EndInvoke()方法既可以在同一线程中调用，也可以在其它线程中调用，EndInvoke()方法会阻塞调用线程，直到异步调用执行结束。

3.异步调用时，AsyncCallback 委托在什么线程中调用？

A：会在系统线程池中的某一个线程中调用 AsyncCallback 委托。

4.简述异步编程与多线程编程的区别与联系。

A：异步编程与多线程编程的效果类似，都是为了能够并行执行代码，达到同时处理任务的目的。异步编程时，系统自己通过线程池来分配线程，不需要人工干预，异步编程逻辑复杂

不易理解，而多线程编程时，完全需要人为去控制，相对较灵活。

第七章 可复用代码：组件的来龙去脉

平时常说的“组件”包含的范围比较广泛，一个程序集、一个链接库甚至代码中的一个类都可以称为“组件”，而本章讲到的“组件”仅指.NET 编程过程中实现了 System.ComponentModel.IComponent 接口的类型，包含范围相对比较狭隘。本章介绍了组件的定义及其作用、组件的两种状态（设计时与运行时），还讲到了“容器-组件-服务模型”以及应用了该模型的“窗体设计器”（Form Designer），最后提到了组件中的一个分支：控件以及自定义控件的分类。

7.1 .NET 中的组件

7.1.1 组件的定义

本章讨论的组件与传统意义中的组件不同。传统上讲，任何一个模块均可以称为“组件”，大到一个程序集或者一个动态链接库，小到代码层面上的一个类型，这些都能称之为组件，它们被当作一个整体，能对外提供特定的功能。

本章讨论的组件范围相对来说更狭隘。在.NET 编程中，我们把实现（直接或者间接）System.ComponentModel.IComponent 接口的类型称为“组件”，其余都不是本章将要讨论的范畴。IComponent 接口的定义如下：

```
//Code 7-1
public interface IComponent : IDisposable
{
    event EventHandler Disposed;
    ISite Site { get; set; }
}
```

如上代码 Code 7-1 所示，IComponent 接口实现了 IDisposable 接口，说明它具有 IDisposable 接口的特性（言下之意便是，我们必须按照第四章中讲 IDisposable 接口那样去定义组件）。另外它还包含一个 Disposed 事件，顾名思义，该事件会在组件 Dispose 时激发。IComponent 接口还包含一个 ISite 类型的属性，从该属性的名称基本上可以确定，该属性跟定位有关，没错，它就是用来帮助组件定位的，后面将会介绍到该属性。

.NET 框架中有一个 IComponent 接口的默认实现：System.ComponentModel.Component 类型。该类型默认实现了接口中的方法、事件以及属性，框架中包含的其它预定义组件大部分均派生自该 Component 类。

7.1.2 Windows Forms 中的组件

前面讲到过，在.NET 中只要实现了 IComponent 接口的类型均叫组件。整个.NET 框架中有许许多多的类型实现了 IComponent 接口，因此它们均称为组件。下图 7-1 显示了 Windows Forms 框架中包含常见组件的关系：

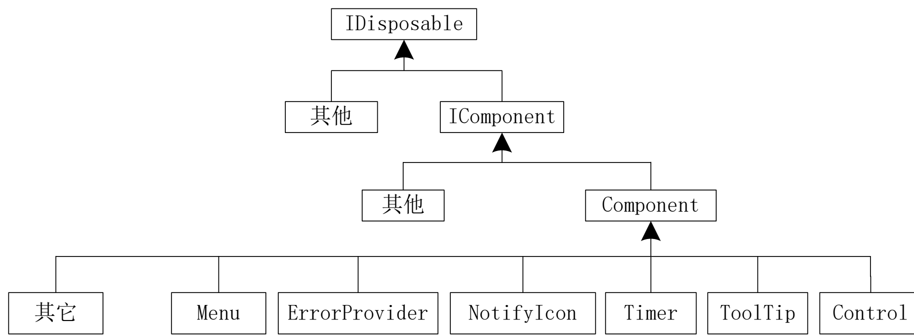


图 7-1 Windows Forms 中组件之间的关系

如上图 7-1 所示，在 System.Windows.Forms 命名空间中，有许多组件均派生自 Component 类，我们需要注意到，Control 类型也派生自 Component 类型，因此，Control 类（及其派生类）均属于组件。

这里需要强调一下，组件和控件不是相等的，组件包含控件，控件只是组件中的一个分类。另外我们常见的 System.Windows.Forms.Timer 以及 System.Windows.Forms.ToolTip 等等严格上讲不能称之为"控件"（虽然我们已经习惯这样称呼）。

7.1.3 Windows Forms 中的控件

控件的种类也有很多，本章中我们只讨论 Windows Forms 中的控件，我们把派生自 System.Windows.Forms.Control 类的类型称为"控件"。控件中包含有处理 Windows 消息的功能（比如窗口过程，详见第八章），因此控件都能以某一种方式在屏幕上呈现出来。在 Winform 编程中，窗体也属于控件，我们可以看到窗体类（Form）也派生自 Control 类：

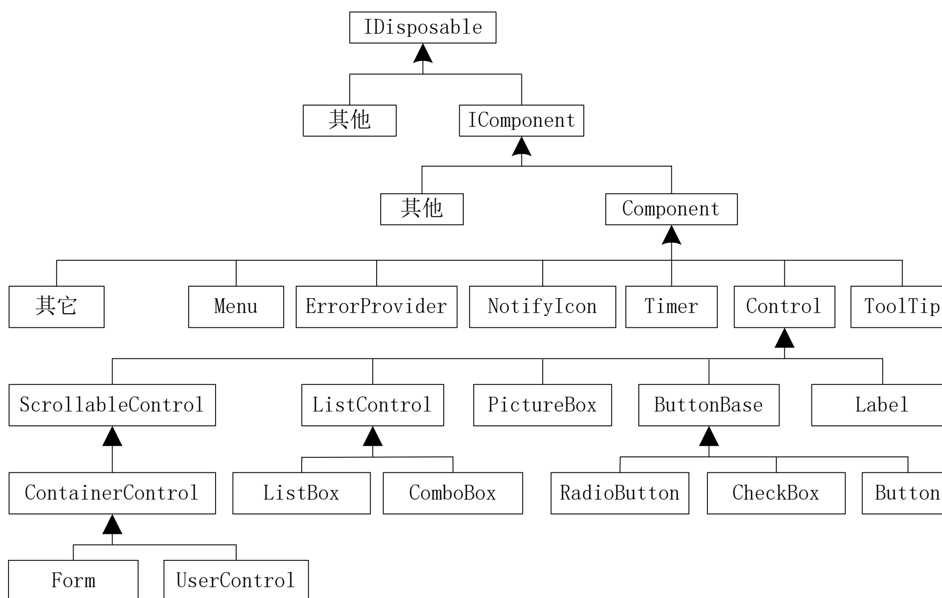


图 7-2 Windows Forms 中控件之间的关系

如上图 7-2 所示，所有的控件均派生自 Control 类，Control 类又属于组件，因此所有控件均具备组件的特性。

不管组件还是控件，它们都是可以重复使用的代码集合，都实现了 IDisposable 接口，都需要遵循第四章中讲到的 Dispose 模式。如果一个类型使用了非托管资源，它实现 IDisposable 接口就可以，那为什么.NET 编程中又要提出组件的概念呢？对于这个问题，并没有官方答案，

不过从.NET 框架的结构和微软为开发者提供的可视化开发环境（比如 Visual Studio）来看，这样做可以说完全是为了实现程序的“可视化开发”，也就是我们常说的“所见即所得”。在类似 Visual Studio 这样的开发环境中，一切“组件”均可被可视化设计，换句话说，只要我们定义的类型实现了 IComponent 接口，那么在开发阶段，该类型就可以出现在窗体设计器中，我们可以使用窗体设计器编辑它的属性、给它注册事件，它还能被窗体设计器中别的组件识别等等。具体介绍请参见本章接下来的几节。

> 注：控件是.NET 编程中的一个重点，它能够提高程序开发效率。ASP.NET 中的所有服务器控件均继承自 SYSTEM.WEB.UI.CONTROL 类，它们之间的关系与 WINDOWS FORMS 中的控件结构类似。

7.2 容器-组件-服务模型

7.2.1 容器的另类定义

这里讨论的容器与我们所熟知的传统容器不同。谈到容器，我们或许会想到 ArraList、Queue 又或者 HashTable 等类，它们的共同特征就是可以存放数据，更具体一点的描述就是：我们能将一个元素放入到容器内部。但本章讨论的容器跟这些都不相同，它们没有物理包含的意思，只是逻辑上包含元素。如果称传统容器为物理容器，那么称这种只是逻辑上包含元素的容器为逻辑容器。两种容器的差别如下图 7-3：

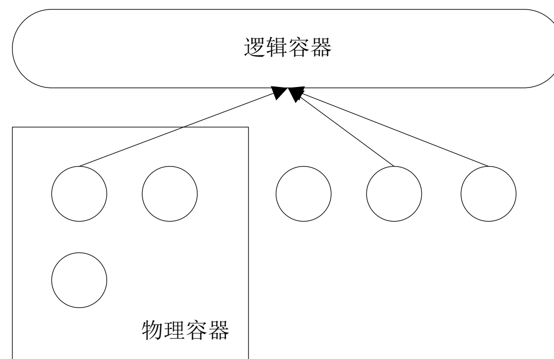


图 7-3 物理容器和逻辑容器的区别

如上图 7-3 所示，物理容器中可以存放元素，但是逻辑容器却没有空间上的限制，物理容器中的元素可以属于另外一个逻辑容器。

> 注：物理容器强调空间上的包含与被包含。对于像 QUEUE、STATCK 或者 ARRAYLIST 这样的物理容器，如果是引用类型对象，存放在这些容器内部的大部分都是对象的引用，而对象实例本身则在容器外部。

.NET 编程中，把所有实现（直接或间接）System.ComponentModel.IContainer 接口的类型称为逻辑容器（以下简称“容器”）。其余都不在本章讨论的范畴之内，IContainer 接口定义如下：

```
//Code 7-2
public interface IContainer : IDisposable
```

```
{
    void Add(IComponent component);
    void Add(IComponent component, string name);
    void Remove(IComponent component);
    ComponentCollection Components { get; }
}
```

如上代码 Code 7-2 所示，IContainer 接口也实现了 IDisposable 接口，说明它具有 IDisposable 接口的特性（言下之意就是，我们必须按照第四章中讲 IDisposable 接口那样去定义容器）。另外它还包含几个添加、移除元素的方法，以及一个组件集合成员。我们可以发现，不管是方法的参数还是集合元素类型，都是组件类型，可以猜测到，容器当然是为组件服务的。没错，本章讨论到的容器只能逻辑包含组件，只有组件才能成为它的逻辑元素。

.NET 框架中有一个 IContainer 接口的默认实现：System.ComponentModel.Container 类型，该类型默认实现了 IContainer 接口中的方法以及属性。

现在我们已经知道，容器包含的逻辑元素是组件，容器也实现了 IDisposable 接口，很显然，我们可以通过容器统一管理各个逻辑元素的非托管资源。但容器对于组件来讲，仅仅是为了更方便地管理组件的非托管资源吗？如果是这样，完全没必要存在本章讨论的这种容器，看来，容器和组件相结合起来，还有其它的作用。

7.2.2 容器与组件的合作

前面讲到过，传统容器中可以物理包含元素，但是这些元素之间是相互独立、不能互相通讯的。也就是说，如果 ArrayList 中包含 A 和 B 两个元素，那么 A 是不知道 B 的存在，B 也不知道 A 的存在，B 也更不可能告诉 A：我今年 25 岁。传统容器仅仅是在空间上简单地将数据组织在一起，并不能为数据之间的交互提供支持。而本章讨论的逻辑容器，在某种意义上讲，更高级，它能为组件（逻辑元素）之间的通讯提供支持，组件与组件之间不再是独立存在的，此外，它还能直接给组件提供某些服务。下图 7-4 显示物理容器和逻辑容器分别与元素之间的关系：

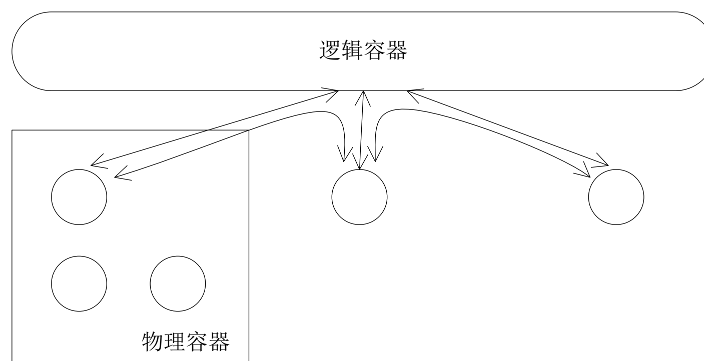


图 7-4 容器与元素之间的关系

如上图 7-4 所示，物理容器中的元素之间不能相互通讯，物理容器也不可能为内部元素提供服务，逻辑容器中的组件之间可以通过逻辑容器作为桥梁，进行数据交换，同时，逻辑容器还能给各个组件提供服务。

上面提到过“服务”，所谓服务，就是指逻辑容器能够给组件提供一些访问支持，比如某个组件需要知道它所属容器共包含有多少个组件，那么它就可以向容器请求，容器为它返回一个获取组件总数的接口。类似下图 7-5：

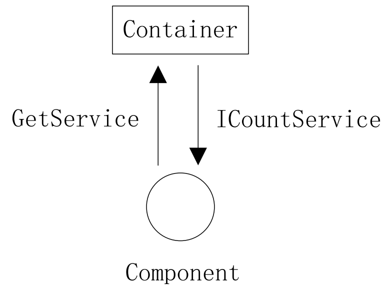


图 7-5 组件向容器请求服务

如上图 7-5 所示，组件向容器请求“计数服务“，容器给组件返回一个 ICountService 的接口，该接口专门负责计数相关的操作,组件可以使用该接口获取当前容器中所有组件的总数。同理，组件还可以向容器请求其它类型的服务，只要容器可以提供。

到此，我们已经大概知道了逻辑容器存在的真正目的，那就是为所有属于该容器的组件提供服务，使组件与组件之间能够自由交互。那么，容器和组件内部到底有着怎样的实现，才会达到如此效果？在本章 7.1.1 小节中就提到过 IComponent 接口中有一个 ISite 类型的属性，当时说是起到一个“定位”的作用，现在看来，组件与容器之间的纽带就是它，组件通过该属性可以与它所属容器取得联系。

我们看一下 ISite 接口的默认实现 Site 类型内部结构：

```
//Code 7-3
private class Site : ISite, IServiceProvider
{
    private IComponent component;
    private Container container;
private string name;

    internal Site(IComponent component, Container container, string name);
    public object GetService(Type service);    //NO.1

    public IComponent Component { get; }    //NO.2
    public IContainer Container { get; }    //NO.3
    public bool DesignMode { get; }
    public string Name { get; set; }
}
```

如上代码 Code 7-3 所示，Site 类型中包含容器 Container 以及组件 Component 属性（NO.3 和 NO.2 处），它两正是分别代表当前组件以及该组件所属容器。Site 类型中还包含一个 GetService 方法（NO.1 处），该方法是组件向容器请求服务的关键。

我们再来看一下 IComponent 接口的默认实现 Component 类型内部结构（不全）：

```
//Code 7-4
public class Component : MarshalByRefObject, IComponent, IDisposable
{
    private static readonly object EventDisposed;
    private EventHandlerList events;
    private ISite site;

    [Browsable(false), EditorBrowsable(EditorBrowsableState.Advanced)]
```

```
    public event EventHandler Disposed;

    static Component();
    public Component();
    public void Dispose();
    protected virtual void Dispose(bool disposing);
    protected override void Finalize();
    protected virtual object GetService(Type service);    //NO.1
    public override string ToString();

    //…
}
```

如上代码 Code 7-4 所示，Component 类中包含一个 GetService 方法（NO.1 处），该方法的作用就是向它所在容器请求服务（见图 7-5），所有 Component 的派生类均可以使用 GetService 请求服务。

最后再看一下 IContainer 接口的默认实现 Container 类型内部结构（不全）：

```
//Code 7-5
public class Container : IContainer, IDisposable
{
    private ComponentCollection components;
    private int siteCount;
    private ISite[] sites;
    private object syncObj;

    public Container();
    public virtual void Add(IComponent component);
    public virtual void Add(IComponent component, string name);
    protected virtual ISite CreateSite(IComponent component, string name);
    public void Dispose();
    protected virtual void Dispose(bool disposing);
    protected override void Finalize();
    protected virtual object GetService(Type service);    //NO.1
    public virtual void Remove(IComponent component);
    private void Remove(IComponent component, bool preserveSite);

    //…
}
```

如上代码 Code 7-5 所示，Container 类中也包含一个 GetService 方法，它专门为组件提供服务，注意它是一个虚方法，也就是说，如果我们从它派生出来一个新的容器，我们完全可以在新容器中重写该虚方法，增加新的服务（Container 容器默认不提供任何服务）。下面代码创建一个新容器，为组件提供计数服务：

```
//Code 7-6
public class MyContainer:Container
{
    protected override object GetService(Type service)    //NO.1
```

```
    {
        if(service == typeof(ICountService))
        {
            return new CountService(this);    //NO.2
        }
        return base.GetService(service);
    }
}
public interface ICountService    //NO.3
{
    //...
    int GetAllComponentsCount();
}
class CountService:ICountService    //NO.4
{
    MyContainer _container;
    public CountService(MyContainer container)
    {
        _container = container;
    }
    public int GetAllComponentsCount()    //NO.5
    {
        //...
    }
}
```

如上代码 Code 7-6 所示，新容器 MyContainer 重写了 GetService 方法（NO.1 处），当组件请求计数服务时，GetService 方法为组件返回一个 ICountService 接口（NO.2 处），组件之后就可以通过该接口获取当前容器中组件的总数目（NO.5 处）。

上面可以看到，Component、Site 以及 Container 三个类型均包含有获取服务的方法 GetService，现在我们可以整理一下，组件向容器请求服务的流程：

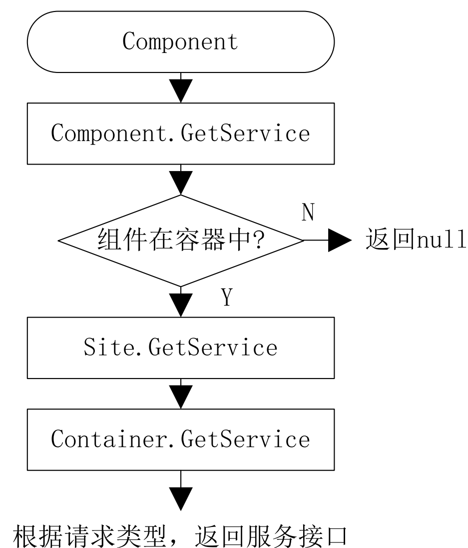


图 7-6 组件请求服务流程

如上图 7-6 所示，组件内部在请求服务的时候，先要判断自己是否在某一个容器中，如果不在容器中，那么返回 null；如果在容器中，则以 Site 作为桥梁（Site.GetService），去获取容器中的服务（Container.GetService）。

注：容器将组件添加进来的时候（执行 CONTAINER.ADD），会初始化该组件的 SITE 属性，让该组件与容器产生关联，只有当这一过程发生之后，组件才能获取容器的服务。有关 COMPONENT、SITE 以及 CONTAINER 类型的详细信息请使用反编译工具查看.NET 源码。

在编程中，使用“容器-组件-服务”模型时，只要容器不变（意味着提供服务的规则不变），我们就可以根据需要开发出各种各样的组件，组件像一个 U 盘，随时可以插在容器上工作。最重要的是，它能通过容器访问到容器中其它组件，容器这时候更像一个总线（Bus），如下图 7-7：

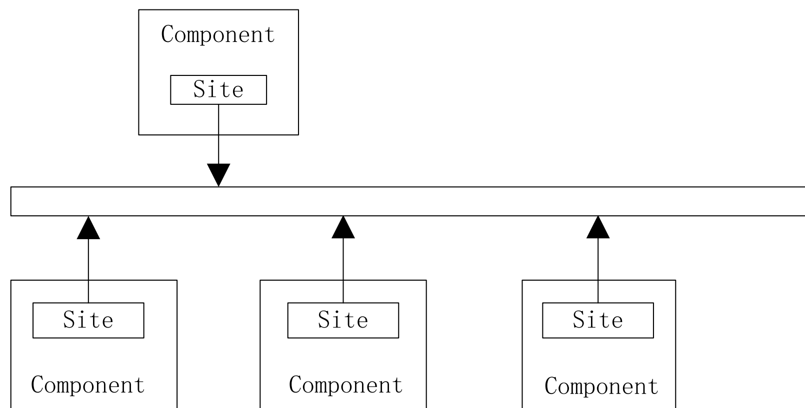


图 7-7 容器充当总线功能

如上图 7-7 所示，在总线不变的情况下，意味着通讯协议不变（容器不变，那么提供服务的规则也不变），各种扩展组件（Component 派生类）均可以加入这条总线。

7.2.3 窗体设计器

首先我们应该搞清楚一件事情，只有源代码经过编译器编译之后才会生成可执行文件。在这个过程中，所有其它的开发工具都只是起到一个辅助作用，无论我们的源代码是怎样来的，是用记事本手动编写还是通过某些工具自动生成，最后本质上都一样，编译器只认结果，并不关心源代码是怎样产生的。现在流行的一些集成开发环境（IDE）中基本都会带有可视化设计界面，通常称作“窗体设计器”，该窗体设计器的功能就是帮助我们自动生成源代码。没错，我们点点鼠标、拖拖控件，窗体设计器就会为我们生成对应的源程序，窗体设计器的作用如下图 7-8：

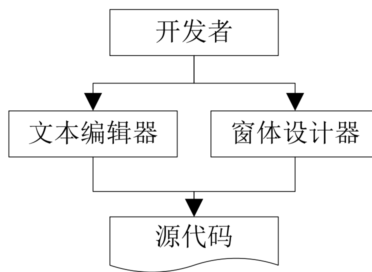


图 7-8 窗体设计器在开发中的作用

如上图 7-8 所示，窗体设计器最终也是为了生成源代码，本质上跟代码编辑器的作用一样。

其次，我们同时也要清楚，我们向窗体设计器中添加的各个组件（比如 Timer、BackgroundWorker、ImageList 等等）以及各种控件（比如 Button、Label 以及 Form 等），它们到底是个什么东西？我们向窗体设计器中拖进去的 Button 控件和程序跑起来之后显示在窗体上的 Button 按钮是一样的吗？答案是肯定的，也就是说，我们向窗体设计器中拖进去的控件（组件）都是内存中存在的对象实例。如果我们使用 Spy++等工具，可以找到窗体设计中控件的句柄，同时我们还可以通过属性窗体（PropertyGrid 控件）修改窗体设计器中控件（组件）的属性，这些修改在设计器中立即就会产生效果，这些都足以说明，在我们向窗体设计器中拖动控件的时候，是会执行类似“new Button();”这样的代码，在内存中实例化一个组件实例。那么，是所有的类型均可以拖放到窗体设计器中吗？答案是否定的，不难发现，能够被窗体设计器设计的像 Imagelist、Timer 等以及所有的控件都属于“组件”，它们都派生自 System.

ComponentModel.Component 类型。我们好像发现了什么，没错，前面说到过，容器（System.

ComponentModel.Container 或其派生类）只能包含组件，并且能够为组件提供服务，组件之间也能够相互通信，那么，我们是不是也可以把窗体设计器当作这样的一种容器呢？事实证明，窗体设计器确实是我们前面提到过了容器，我们在使用窗体设计器时，各个组件之间确实是可以相互通信的，当我们往窗体设计器中拖放一个 ImageList 组件时，设计器中所有其它包含有 ImageList 类型属性的控件都会知道这个拖放进来的 ImageList 组件，因此在我们编辑其它控件的 ImageList 类型属性时，就会有选项供选择。我们不仅仅知道当前可以用来赋值的 ImageList 组件，还可以看到 ImageList 组件中的 Image 列表，从而设置控件的 ImageList 属性值。这一过程的主要贡献在于窗体设计器，它能够为组件与组件之间建立关联。

微软的 Visual Studio 开发环境中的窗体设计器很好的应用了本节之前讲到“容器-组件-服务模型”，这也印证了本章开始之前的一个疑问：.NET 编程中为什么要提出组件的概念？由于窗体设计器中只能容纳组件，所以如果我们想开发出来一个可以被窗体设计器可视化设计的类型，那么我们必须让该类型正确实现 IComponent 接口（或派生自 Component 类）。

既然组件能够请求窗体设计器的服务，那么我们在编写组件时，怎样去取得这些服务呢？又怎样去使用这些服务呢？注意在一般开发中，我们并不需要在组件中编写与窗体设计器交互的代码，原因有两个：

（1）这些事情是由组件开发者来做的，而我们大多数人只是充当组件的使用者；

（2）与窗体设计器交互的代码一般非常复杂，建议没有特殊需要，不要在组件中编写与窗体设计器（组件所在容器）交互的代码，因为这会影响 IDE 的稳定性。

下面示例代码演示了在组件中怎样请求窗体设计器的服务：

```
//Code 7-7
//…include other namespace
using System.ComponentModel;
using System.ComponentModel.Design;
class MyLabel:Label
{
```

```
    //...
    protected override void OnHandleCreated(EventArgs e)
    {
        ISelectionService iss = GetService(typeof(ISelectionService)) as ISelectionService;
        if(iss != null)    //NO.1
        {
            iss.SelectionChanged += new EventHandler(iss_SelectionChanged); //NO.2
        }
        base.OnHandleCreated(e);
    }
    void iss_SelectionChanged(object sender, EventArgs e)
    {
        ISelectionService iss = GetService(typeof(ISelectionService)) as ISelectionService;
        if(iss != null)    //NO.3
        {
            Text = "窗体设计器中当前选中了" + iss.SelectionCount + "个组件\n"
                + "第一个组件是" + (iss.PrimarySelection as Component).Site.Name; //NO.4
        }
    }
}
```

如上代码 Code 7-7 所示，在 Label 的派生类 MyLabel 中，我们重写了 OnHandleCreated 虚方法，在该虚方法中向窗体设计器（容器）请求“组件选择有关”的服务，窗体设计器返回一个 ISelectionService 的服务接口。注意我们需要先判断返回来的服务接口是否为 null（NO.1 处），如果为 null，说明当前组件不在任何容器中或者容器不提供该服务；如果不为 null，我们使用该服务接口注册窗体设计器的 SelectionChanged 事件（NO.2 处），该事件会在窗体设计器选择发生变化后被激发。之后在 SelectionChanged 的事件处理程序中，我们还要向窗体设计器请求服务，请求完成之后依旧要判断返回来的服务接口是否为 null（NO.3 处），如果不为 null，就使用返回来的 ISelectionService 接口将窗体设计器中选择组件的个数显示在 Label.Text 中（NO.4 处）。示例代码最终的效果是：我们向窗体设计器拖放一个 MyLabel 控件，之后任何时候，只要窗体设计器中选择组件发生了变化，MyLabel.Text 属性就会显示当前选中组件的个数，而这一切都发生在窗体设计器中，也就是程序的开发阶段，下图 7-9 为效果图：

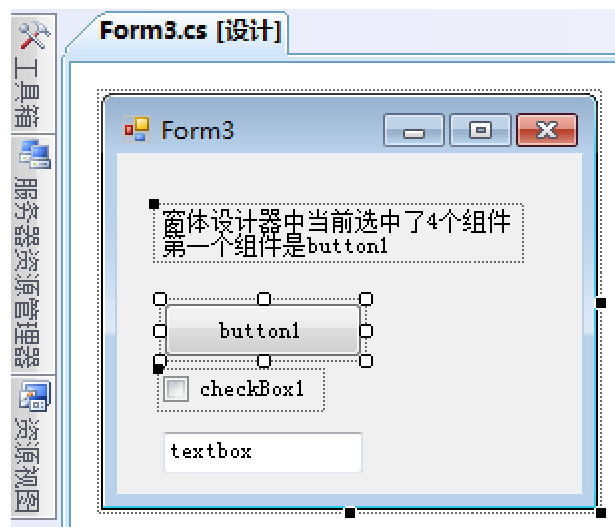


图 7-9 组件与窗体设计器的交互

如上图 7-9 所示，图中窗体最上面的控件是一个 MyLabel 控件，我们在窗体设计器中选中了 4 个组件（注意窗体本身也算一个），MyLabel 的 Text 属性会根据窗体设计器中选择组件的改变而变化。换句话说，组件与窗体设计器之间进行了交互，同理组件与组件之间通过窗体设计器提供的某些服务也可以进行交互。

> 注：VISUAL STUDIO 中窗体设计器默认提供许多与设计有关的服务，服务接口大部分定义在 SYSTEM.COMPONENTMODEL.DESIGN 命名空间中。只要我们知道服务规则，我们就可以在组件中请求这些服务，组件就可以与窗体设计器进行交互。容器永远是服务规则的制定者，组件必须遵守这些规则方可正常工作。

前面说到过，程序开发过程中，源代码才是我们最终所需要的东西。窗体设计器一关闭，设计器中所有的组件全部从内存中销毁，我们用鼠标键盘操作窗体设计器的时候，我们对设计器的任何一个操作，设计器都会帮助我们生成源代码，我们通过属性窗体编辑一个组件的属性，组件能马上产生效果的同时（比如背景色），窗体设计器也能为该操作生成源代码。在 Winform 中，这些代码集中在 InitializeComponent 方法中。只要我们有了源代码文件，窗体设计器中的“假象”就可以随时消失，窗体设计器中的组件和源代码中的组件不是同一个东西，下图 7-10 显示了窗体设计器中的组件和设计器为我们生成的代码：

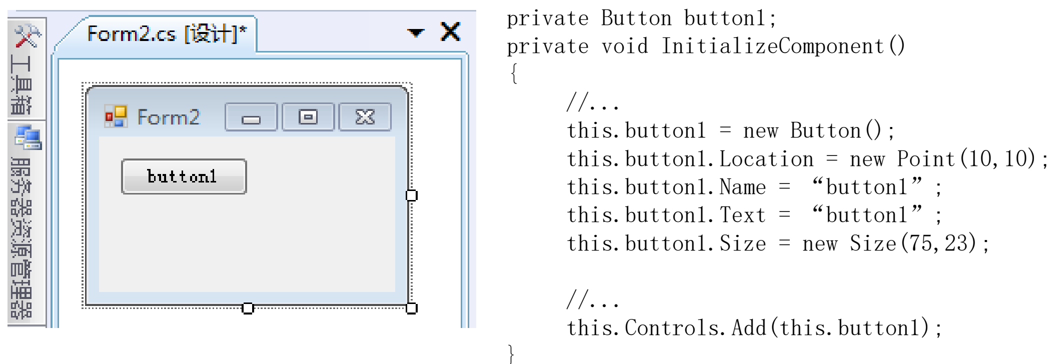


图 7-10 窗体设计器中的组件与生成的源代码

如上图 7-10 所示，图中左边显示我们拖放到设计器中的一个 Button 控件，在这个过程中，窗体设计器除了会实例化一个 Button 控件（图中左边 Form2 中），还会为我们生成图中右边的代码。我们看到，生成的代码会实例化一个 Button 对象，然后将其加入到 Form2.Controls 的子控件集合中。图中左边设计器中的 button1 实例对象与图中右边生成代码中的 button1 变量不是同一个东西。

> 注意：组件不是一定要存在于容器中，它可以和其它对象一样，单独存在。换句话说，组件可以不存在于窗体设计器中，比如程序运行之后，程序中任何组件都不属于窗体设计器，这将是下一节要讨论的话题。

7.3 设计时（Design-Time）与运行时（Run-Time）

7.3.1 组件的设计时与运行时

7.2 节中讲窗体设计器时已经说过，窗体设计器中的各个组件都是真实存在的实例对象。我们向窗体设计器中拖放一个 Button 控件时，必然会调用 Button 类的构造方法实例化一个 Button 对象，Button 对象必然会激发它的 HandleCreated 事件等等，也就是说，无论组件在哪里，它都是以对象的形式真实存在的。

我们把组件处于窗体设计器中的状态称为“设计时”，把组件正常运行时的状态称为“运行时”。设计时的组件和运行时的组件都是以对象的形式存在，因此有很多的相同点，比如都会调用构造方法，都会激发相应事件等等。除了这些相同点以外，还有一些不同点，由于处于设计时的组件存在容器中，因此它可以获取窗体设计器中提供的服务，可以执行一些与窗体设计器交互的代码，而处于运行时状态的组件不存在窗体设计器中，因此它不可以执行与窗体设计器交互有关的代码，见下图 7-11：

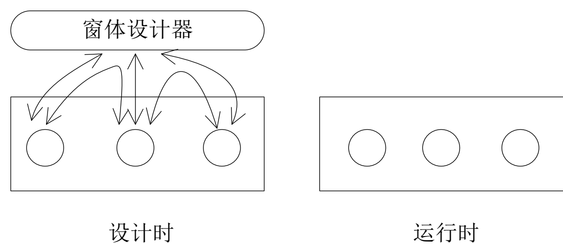


图 7-11 设计时组件与运行时组件

如上图 7-11 所示，圆形代表组件，矩形代表窗体。图中左边显示处于设计器中的组件，这些组件可以与窗体设计器交互，图中右边显示处于运行状态中的组件，它们不能再执行与窗体设计器交互有关的代码，因为它们根本就不在窗体设计器之中。

7.3.2 区分组件的当前状态

任何组件都有两种状态：设计时和运行时。由于在设计时能执行的代码，在运行时可能执行失败，相反，有些代码可能只需要在运行时执行，比如连接数据库的代码，我们在设计时完全没必要让窗体设计器中的一个组件去连接数据库。因此，我们编写组件代码之前，一定要先搞清楚这些代码是在什么状态下去执行。我们可以先检查一下组件的状态，如果组件处于设计时，那么执行代码 A，否则不执行代码 A。有两种方式去判断组件的当前状态：

（1）组件的 DesignMode 属性。每个组件都有一个 Bool 类型的 DesignMode 属性，正如它的字面意思，如果该属性为 true，那么代表组件当前处于设计时状态；否则组件处于运行时状态。我们将任何组件拖进窗体设计器后，设计器就会将组件的 DesignMode 设置为 true（该属性默认为 false）。假如现在要开发一个控件，它具有显示自己版本信息的功能，但是仅仅在开发阶段显示，用于提醒开发者当前所用组件的版本，而当整个程序运行之后，版本信息不再显示，那么该控件代码可以这样写：

```
//Code 7-8
class MyControl:Control
{
    public MyControl()
    {
        //...
    }
    protected override void OnPaint(PaintEventArgs e)
    {
```

```
            e.Graphics.DrawRectangle(Pens.Blue,new Rectangle(0,0,Width-2,Height-2));    //NO.1
            //...
            if(DesignMode)    //NO.2
            {
                string v = "v2.30.109.1302";    //read from somewhere
                using(Font f = new Font("arial",10))
                    e.Graphics.DrawString(v,f,Brushes.Blue,new PointF(5,5));
            }
        }
}
```

如上代码 Code 7-8 所示，在 MyControl 控件的 OnPaint 方法中，我们先绘制了一个蓝色边框（NO.1），该行代码无论组件处在什么状态都会执行，因此我们可以看到拖到窗体设计器中的 MyControl 控件有一个蓝色边框，程序运行之后，窗体中的 MyControl 控件也有一个蓝色边框。接下来需要显示版本信息，因为只有当组件处于设计时状态，才会显示版本信息，因此我们需要先判断组件的当前状态（NO.2 处），如果 DesignMode 属性为 true，那么说明组件处在窗体设计器中，需要显示版本信息；否则，说明组件处在运行时状态，不显示版本信息。任何一个使用 MyControl 控件的开发者，都会在窗体设计器中看到当前 MyControl 控件的版本信息，而当程序运行后，该版本信息不再显示。

（2）随便请求一个服务，看返回来的服务接口是否为 null。前面提到过，当一个组件不属于任何一个容器时，那么它通过 GetService 方法请求的服务肯定返回为 null。因此，我们可以请求一个窗体设计器能够提供的服务（比如前面用到过的 ISelectService 服务），看请求的返回值是否为 null，如果为 null，说明当前组件处于运行时；否则，当前组件处于窗体设计器中。前面的 MyControl 示例代码可以改为：

```
//Code 7-9
class MyControl:Control
{
    public MyControl()
    {
        //...
    }
    protected override void OnPaint(PaintEventArgs e)
    {
        e.Graphics.DrawRectangle(Pens.Blue,new Rectangle(0,0,Width-2,Height-2));    //NO.1
        //...
        ISelectionService iss = GetService(typeof(ISelectionService)) as ISelectionService;
        if(iss != null)    //NO.2
        {
            string v = "v2.30.109.1302";    //read from somewhere
            using(Font f = new Font("arial",10))
                e.Graphics.DrawString(v,f,Brushes.Blue,new PointF(5,5));
        }
    }
}
```

如上代码 Code 7-9 所示，我们先请求一个 ISelectionService 服务，再判断它的返回值是否为 null（NO.2 处），如果不为 null，说明组件当前处于窗体设计器中；否则，组件处于运行时状态。

注：（1）（2）方法均不适合嵌套组件，因为窗体设计器只会将最外层组件的DESIGNMODE 属性值设置为 TRUE，如果这个组件内部还包含其它子组件，那么这些子组件的 DESIGNMODE 属性还是原来的默认值 FALSE，因此（1）对嵌套组件中的子组件无效。我们拖放一个嵌套组件到窗体设计器中，只有最外层组件加入到了窗体设计器中，所以只有最外层组件能够通过 GETSERVICE 方法请求窗体设计器的服务，内部的子组件由于没有加入到容器，因此 GETSEIVICE 方法返回 NULL，因此（2）对嵌套组件中的子组件也无效。有一种可以解决嵌套组件中无法判断其子组件状态的方法，那就是通过 PROCESS 类来检查当前进程的名称，看是否包含"DEVENV"这个字符串，如果是，那么说明组件当前处于 VISUAL STUDIO 开发环境中（即组件处于设计时），IF(PROCESS.GETCURRENTPROCESS().PROCESSNAME.CONTAINS("DEVENV"))为假，说明组件处于运行时。这种方法也有一个弊端，很明显，如果我们使用的不是 VISUAL STUDIO 开发环境（也就是进程名不包含 DEVENV），或者我们自己的程序进程名称就包含 DEVENV 怎么办呢？

7.3.3 组件状态的应用

作为一名普通的开发人员，几乎不需要接触到组件状态这些概念，大部分开发人员只是使用组件，也就只需要编写好组件在运行时需要执行的代码即可，如果你是一个组件开发人员，那么你就可能需要与窗体设计器打交道，控制组件与设计器之间的交互，能让组件的使用者更加方便的去使用组件。

在开发一些需要授权的组件时，就可以用到组件的两种状态，这些需要授权的组件收费对象一般是开发者，因此，在开发者使用这些组件开发系统的时候（处于开发阶段），就应该有授权入口，而当程序运行之后，就不应该出现授权的界面，这时候就可以根据组件的当前状态来判断是否需要显示授权入口。

如果我们编写了与窗体设计器交互的代码，那么一定要谨慎小心，因为访问窗体设计器的代码很容易就会造成开发环境崩溃。

7.4 控件

7.4.1 控件基类

本章 7.1.3 小节中已经介绍了 Windows Forms 中的控件分类及其派生关系。控件作为组件中的一个分支，具有可视化显示的功能，言下之意就是控件内部具备 Windows 消息处理的功能（详见第八章）。System.Windows.Forms.Control 类是所有控件的基类，它内部已经提供了所有控件必须具备的基础结构，比如窗口句柄、窗口过程以及基础的 Windows 消息路由。

Control 类的默认外观显示为一个矩形，Windows Forms 框架中其它所有控件均派生自 Control 类。

7.4.2 用户自定义控件

Windows Forms 中包含有非常强大也非常完善的控件，比如基础控件 Button、CheckBox 等，容器布局控件 Panel、TabControl 等，菜单控件 MenuStrip 以及数据显示控件 DataGridView 控件等。这些控件在一般开发中可以满足我们的需要，但是有些时候对于一些特殊的功能需求，使用系统自带的控件远远不够，因此，我们需要自己开发满足功能要求的控件。有三种方式开发新的控件：

（1）复合控件（Composite Control）；

这种方式很简单，就是将已有控件组合在一起，形成一个整体，将现有控件功能集中起来。我们平时开发的“用户控件”，从 UserControl 类派生而来，将许许多多现有控件集中到一起，这种控件就属于复合控件，见下面代码：

```
//Code 7-10
class MyUserControl:UserControl
{
    public MyUserControl()
    {
        InitializeComponent();   //NO.1
    }
    //…
}
```

如上代码 Code 7-10 所示，在 MyUserControl 类中的 InitializeComponent 方法中（NO.1 处），我们可以将现有的控件组合在一起，形成一个整体。注意 InitializeComponent 方法中的代码一般由窗体设计器生成，我们每次通过窗体设计器向 MyUserControl 中添加一个控件，在 InitializeComponent 方法中都会生成类似“this.Controls.Add(…)”这样的代码，意思就是将新添加的控件加入 MyUserControl.Controls 集合中。

（2）扩展控件（Extended Control）；

从现有控件派生出一个新控件。如果现有控件基本已经满足需求，只是需要稍微增加一些小功能或者稍微修改现有功能，那么我们可以将已有控件作为基类，派生出一个新控件，在派生类中编写增加功能或者修改功能的代码，下面代码演示了一个从 Button 类派生的 MyButton 类，改变了原来 Button 的显示外观，在原来显示外观的基础上绘制了一个蓝色矩形：

```
//Code 7-11
class MyButton : Button
{
    protected override void OnPaint(PaintEventArgs pevent)
    {
        base.OnPaint(pevent);
        pevent.Graphics.DrawRectangle(Pens.Blue, new Rectangle(1, 1, Width - 3, Height - 3));
//NO.1
    }
}
```

如上代码 Code 7-11 所示，我们在 MyButton 类中重写了 OnPaint 虚方法，每次控件需要重绘的时候，我们在控件界面绘制一个蓝色矩形（NO.1 处），除了显示外观的差别，MyButton 类与 Button 类具有完全一样的功能。本示例代码只是在 Button 类的基础上进行一个非常简单的修改。

（3）自定义控件（Custom Control）。

以 Control 为基类，直接派生出一个新控件。这种方式对开发者的技术能力要求较高，因为 Control 类中只是包含了所有控件应该具备的基础结构，它并不提供控件的特定功能以及显

示界面，因此无论从功能的实现还是界面的显示均要求开发者自己去处理，比如控件的外观显示，开发者必须熟悉 GDI 和重写 OnPaint 虚方法，而对于控件的一些功能实现，开发者必须熟悉 Windows 消息和重写控件的窗口过程 WndProc 这个虚方法等。正是因为这种从底层都需要开发者自己去实现的做法，才让我们更灵活地控制控件的行为和外观，下面示例代码演示如何从 Control 类派生出新控件：

```
//Code 7-12
class MyControl:Control
{
    public event EventHandler Event1;    //NO.1
    public event EventHandler Event2;    //NO.2

    public MyControl()
    {
        //…
    }
    protected override void OnPaint(PaintEventArgs e)
    {
        base.OnPaint(e);
        // paint the surface of MyControl
        e.Graphics.DrawRectangle(Pens.Blue, new Rectangle(1, 1, Width - 3, Height - 3));    //NO.3
    }
    protected override void WndProc(ref Message m)
    {
        if(m.Msg == ?)    //NO.4
        {
            //raise Event1 with Message’s arguments
            return;
        }
        if(m.Msg == ?)    //NO.5
        {
            //raise Event2 with Message’s arguments
            return;
        }
        //…
        base.WndProc(ref m);
    }
}
```

如上代码 Code 7-12 所示，我们在 MyControl 类中重写了 OnPaint 虚方法，负责绘制控件的外观显示（NO.3 处），还重写了 WndProc 虚方法，拦截 Windows 消息（NO.4 和 NO.5 处），将消息参数转换成事件参数，最后激发相应事件（这个过程参考第八章），注意重写的虚方法中不要忘记调用基类的虚方法。

> 注：无论是复合控件、扩展控件还是自定义控件，我们均可以重写控件的窗口过程：WNDPROC 虚方法，从根源上接触到 WINDOWS 消息，这个做法并不是自定义控件的专利。

7.5 本章回顾

本章讲到的“组件”是指.NET 编程中的组件，特指直接或间接实现了 IComponent 接口的类型，它跟我们通常意义上谈到的组件有很大的区别。组件在.NET 编程中起到了重要作用，所有可在 IDE 中可视化设计的类型必须是组件，包括我们直接从工具箱中拖到窗体设计器中的 UI 相关组件（如 Button 按钮）和其它功能组件（如 backgroundWorker）。本章还介绍了“容器-组件-服务”模型，介绍了窗体设计器与组件之间的关系，以及为什么一个组件可以放在设计器中进行可视化设计，之后还介绍了组件的两种状态：设计时（Design-Time）和运行时（Run-Time），组件的这两种状态对组件的行为表现起到了重要作用。

7.6 本章思考

1..NET 编程代码中组件的定义是？

A：特指实现（直接或间接）了 System.ComponentModel.IComponent 接口的类型，只有组件才可以在窗体设计器中进行可视化设计。

2.“容器-组件-服务”模型中容器的含义是？

A：特指实现（直接或间接）了 System.ComponentModel.IContainer 接口的类型，不包括 ArrayList、Queue、Stack 以及 Array 等物理容器。

3.组件有哪两种状态？怎样区分组件的当前状态？

A：组件有“设计时（Design-Time）”与“运行时（Run-Time）”两种状态，可以通过组件的 DesignMode 属性去判断它的当前状态，一般情况下，如果该属性为 true，说明当前组件处于设计时，否则处于运行时（参见本章 7.3.2 小节）。在窗体设计器中创建的组件的状态即为设计时，程序运行后组件的状态即为运行时。

第八章 经典重现：桌面 GUI 框架揭秘

Windows 桌面应用程序具有人性化的 GUI 界面一直以来是 Windows 操作系统能够进入平常百姓家的重要原因，本章将以传统 Win32 应用程序和.NET 中 Windows Forms 程序为例，来说明桌面 GUI 框架的整个来龙去脉，让我们不仅能够知道这些流行了前后两个世纪的经典框架的神秘设计结构，还能从这些主流框架中吸取经验，将一些经典的设计结构应用到自己的实际项目中去，如本书第十章中讲到的“泵”结构在 GUI 框架中有明显体现。

受篇幅影响，本章对有关 C++数据类型、Win32 API 以及与 Windows 相关的一些概念比如句柄等并没有做非常详细的介绍，有这方面基础的读者能更好地理解本章内容。

8.1 了解传统 Win32 应用程序的必要性

在类似.NET Frameworks（Windows Forms）、MFC（Microsoft Foundation Class）这样的框架出现之前，开发者只能通过使用 Win32 API（也称 Windows API）来开发 Windows 桌面应用程序（称这样的程序为传统 Win32 应用程序），Win32 API 直接与 Windows 操作系统交互。由于 Win32 API 数量多、参数含义复杂、直接访问操作系统容易造成系统崩溃，再加上开发同样类似的程序需要重复编码，导致 Win32 应用程序开发技术门槛高、开发效率低等现象。1992 年，微软发布了第一版 MFC，MFC 将 C++、面向对象以及 Window 程序开发结合于一体，其中封装了复杂难用的 Win32 API，大大的提高了开发效率，降低了技术开发门槛。慢慢地，传统直接使用 Win32 API 的开发模式渐渐走出了历史舞台，使用框架开发 Windows 应用程序成为一种流行。

既然我们已经摒弃了直接使用 Win32 API 开发应用程序的这种模式，那么我们为什么还要了解传统的 Win32 应用程序呢？原因其实很简单，框架是一把双面刃，它帮助开发者提高了开发效率的同时，也隐藏了整个应用程序真实的运行原理。有了框架，我们往往只专注于程序上层的功能完善，却忽略了程序底层的实现原理。

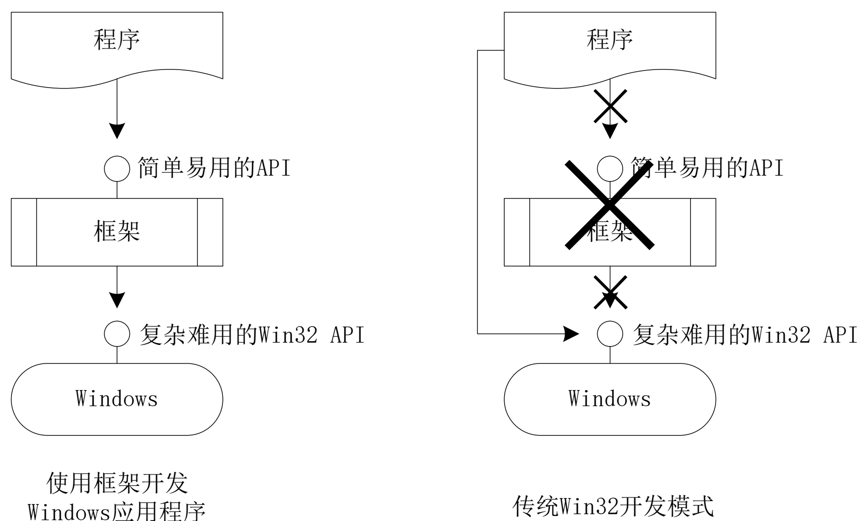


图 8-1 传统 Win32 开发模式与框架开发模式的区别

如上图 8-1，使用框架开发 Windows 应用程序时，我们基本只需要熟悉框架对外公开的接口（API），而一般情况下框架提供的接口的可理解性、易使用性等，都比原始 Win32 API 的高，由于框架内部对 Win32 API 做了一定的封装，所以很多时候一个功能，如果使用 Win32 API 开发需要好几个步骤完成，而使用框架的话，一个步骤就可以达到同样的效果，另外，框架也解决了代码的可重用问题。

注：以上特性适合任何框架。

如果框架是盖完之后的毛胚房，那么框架的开发者就是建筑设计师、建筑工人等，如果使用框架开发出来的 Windows 应用程序是经过装修之后的商品房，那么框架的使用者就是装修设计师、装修工人等，很显然，装修队伍不懂建筑原理，当然现实生活中也不需要懂，但是作为一名框架使用者来讲，了解框架内部原理却是非常有必要的，原因有两个：第一，作为一名技术人员，“刨根问底”应该是我们一直坚持的原则，我们不但要知其然，更要知其所以然；第二，成熟流行的框架一般由专业的技术团队开发完成，他们的技术水平在我们绝大多数人之上，通过研究框架的内部实现，我们可以学习它们的编程思想、设计模式等。

本章先介绍传统 Win32 应用程序的主要结构以及它与.NET 中 Winform 应用程序的关联性，紧接着将揭开.NET 中 Windows Forms 框架的神秘面纱。

8.2 Win32 应用程序结构

8.2.1 运行平台决定程序结构

任何一个程序或者系统，都应该具备“输入”、“处理”以及“输出”这三个部分，不然它就不能正常工作或者没有任何意义。程序都是运行在操作系统之中，操作系统的作用总结下来，就是为程序正常运行提供帮助，如果把操作系统当作一个容器，那么程序就是这个容器中的元素，容器可以为这些元素提供各种各样的服务。

Windows 操作系统是一款基于图形化工作界面的操作系统，它为运行在它里面的程序提供“输入”以及“输出”支持，下图 8-2 描述了 Windows 应用程序与 Window 操作系统之间的关联：

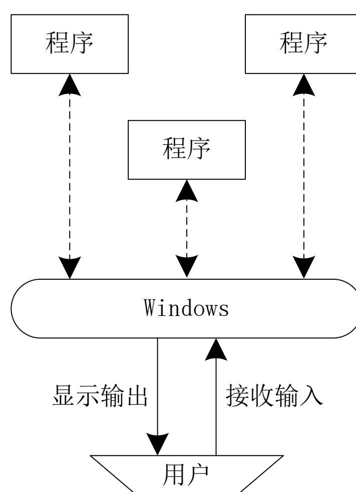


图 8-2 Windows 程序与 Windows 系统之间的关系

如上图 8-2，Windows 操作系统接收用户输入，比如键盘鼠标等输入，由于程序是无法直接识别这些输入信息的，所以 Windows 操作系统要将用户输入转换成固定格式数据，然后供程序

使用，程序则主要负责对固定格式数据进行处理，处理结果由 Windows 操作系统负责显示输出，比如输出到屏幕。Windows 的功能有两个，一个是接收用户输入，第二个就是显示输出，这样一来，程序就只需要把精力放在“处理数据”这一块上。

虽然 Windows 帮我们做了“接收输入”和“显示输出”的工作，但是我们程序要想正常地工作，还是要和 Windows 进行数据交换（图 8-2 中虚线部分），因此每个 Windows 应用程序都应该有从 Windows 中获取数据以及向 Windows 输出数据的模块，也就是说，每个 Windows 应用程序的结构应该类似如下：

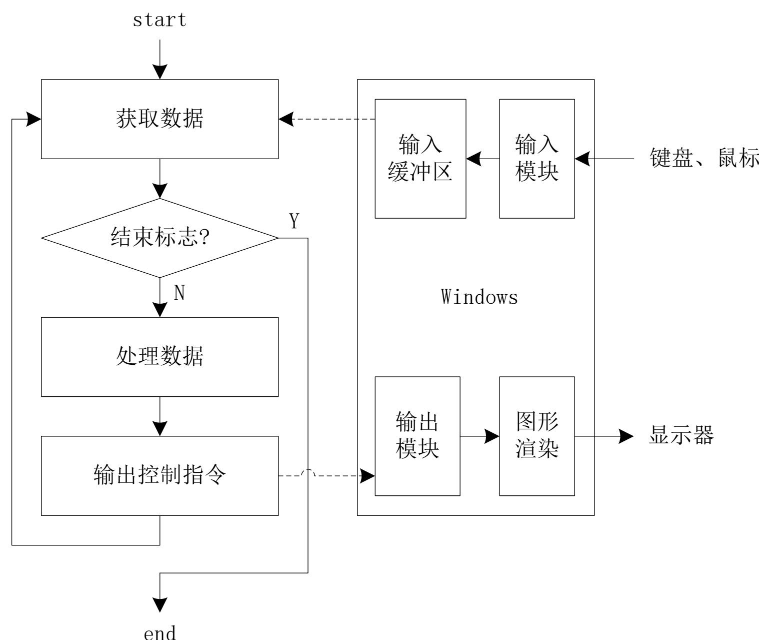


图 8-3 Windows 应用程序结构

上图 8-3 左边部分为一个 Windows 应用程序的主要结构，右边部分为 Windows 中有关输入和输出的结构。图 8-3 中虚线部分对应图 8-2 中虚线部分，可以看到，每个程序都会和 Windows 进行通信，都必须包含有与 Windows 交互的模块，另外，图中左边“处理数据”方框是我们编写程序时的重点部分，后面讲“窗口过程”时会详细说明。

注：本章中出现的“WINDOWS 应用程序”或者“WINDOWS 程序”都指的是 WINDOWS 桌面应用程序，指包含可视化窗口界面的一类程序，本章中也只讨论这一类程序，不包含像 WINDOWS 控制台应用程序等。

8.2.2 Windows 消息循环

上一小节中说到过，程序是无法直接识别用户键盘或者鼠标等设备的输入信息，这些输入必须先由操作系统转换成固定格式数据之后，才能被程序使用，那么程序是怎么获取这些固定格式数据呢？Windows 编程中，我们把由操作系统转换之后的固定格式数据称为 Windows 消息，Windows 消息是一种预定义的数据结构（比如 C++中的 Struct），它包含有消息类型、消息接收者以及消息参数等信息，我们还把程序中获取 Windows 消息的结构称之为 Windows 消息循环，Windows 消息循环在代码中就是一个循环结构（比如 while 循环），它不停地从操作系统中获取 Windows 消息，然后交给程序处理。Windows 消息循环结构类似下图 8-4：

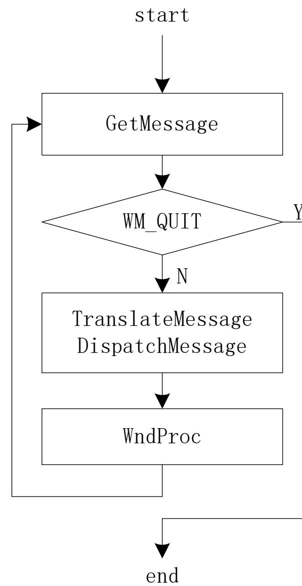


图 8-4 Windows 消息循环

上图 8-4 显示了 Windows 消息循环的结构，GetMessage 从操作系统中的消息队列（一种数据结构）获取 Windows 消息，这些消息包括操作系统将用户输入转换而成的、操作系统本身发送给程序的以及其它程序发送给本程序的，如果获取到的消息不为 WM_QUIT（表示退出含义），那么经过 TranslateMessage 方法进行预处理后（主要进行键盘消息转换），再由 DispatchMessage 将消息投递到指定的窗口过程（WndProc），窗口过程则负责处理消息。

C++中 Windows 消息结构体的定义如下：

```
//Code 8-1
typedef struct tagMSG {
  HWND     hwnd;
  UINT     message;
  WPARAM wParam;
  LPARAM lParam;
  DWORD    time;
  POINT    pt;
} MSG, *PMSG, *LPMSG;
```

如上代码 Code 8-1 所示，消息中包含有消息接收者的窗口句柄 hwnd，消息类型 message（一个数值，代码中常用 WM_PAINT、WM_QUIT 等代表），以及两个消息参数 wParam 和 lParam，还有发送消息时间 time 以及当前光标在屏幕中的坐标位置。消息投递方法 DispatchMessage 就是根据 hwnd 字段，将消息投递给指定窗口的窗口过程，最终由窗口过程处理这一消息。C++中 Windows 消息循环代码类似如下：

```
//Code 8-2
BOOL bRet;
while( (bRet = GetMessage( &msg, hWnd, 0, 0 )) != 0)
{
    if (bRet == -1)
    {
        // handle the error and possibly exit
    }
    else
```

```
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

GetMessage 方法有三种返回值，当取得 WM_QUIT 消息时，返回 0 值，While 循环退出；当有异常发生时，返回-1 值；当获取非 WM_QUIT 的其它消息时，返回非 0 非-1 的值。

如果一个程序需要退出，那么 Windows 消息循环所在的线程必须结束，也就是说，While 循环中的 GetMessage 方法必须返回 0 值，由于 WM_QUIT 是 While 循环退出的条件，因此，消息队列中包含有一个 WM_QUIT 消息是程序退出的前提。

> 注：包含 WINDOWS 消息循环的线程称为"UI 线程"，一般 UI 线程结束意味着程序退出。另外，操作系统为每个 UI 线程维护一个消息队列，一个 UI 线程对应有一个消息队列，UI 线程负责处理对应消息队列中的消息。理论上，一个程序可以包含有多个 UI 线程，但最好不要这样去做。

8.2.3 窗口过程

上一小节中讲到，UI 线程中的消息循环将消息队列中的消息取出来之后，如果不是 WM_QUIT，那么就会通过 DispatchMessage 方法将消息投递（调用）给对应的窗口过程，窗口过程接着处理该消息。那么，这样看来，窗口过程就是我们程序中的重点部分，因为处理消息的逻辑都在窗口过程中。

事实上，窗口过程确实是 Windows 程序开发过程中的重中之重，不管是传统 Win32 应用程序开发还是使用其它任何框架开发，我们开发者所做的大部分工作就是完善（扩展）窗体的窗口过程。窗口过程用 C++代码声明类似如下：

```
//Code 8-3
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    //...
    switch (message)    //NO.1
    {
    case WM_PAINT:
        //draw something here with gdi win32 api
        break;
    case WM_DESTROY:
        PostQuitMessage(0);    //NO.2
        break;
    //...any other message,any other 'case'
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);    //NO.3
        break;
    }
    return 0;
}
```

如上代码 Code 8-3 所述，窗口过程就是一个方法，任何一个窗口过程包含有 4 个参数，跟消息结构体中的前四个相同，分别为接收消息的窗口句柄、消息类型以及两个消息参数。窗口过程中，有一个庞大的 switch/case 块（NO.1 处），我们开发者要做的就是在这个 switch/case 块中拦截我们需要处理的消息，比如示例代码中的 WM_PAINT 消息，当窗口某个部分需要重绘时，系统（或者其它程序）就会向消息队列中发送 WM_PAINT 消息，经过 DispatchMessage 方法后，WM_PAINT 消息被投递到窗口过程，我们在处理 WM_PAINT 消息时，可以将需要显示的内容通过一些 GDI Win32 API 方法输出（显示输出指令）。如果窗口为一个程序的主窗体，那么当这个窗口关闭后，程序应该退出（While 循环结束），又因为当关闭窗体后，系统会有一个 WM_DESTROY 消息发送给窗口过程（注意是直接发送给窗口过程），那我们完全可以在 switch/case 中拦截 WM_DESTROY 消息，然后调用 PostQuitMessage 这个 Win32 API（NO.2 处），它会给消息队列发送一个 WM_QUIT 消息，下一次消息循环时，碰到 WM_QUIT 后，While 消息循环就会结束。除了示例代码中的 WM_PAINT 和 WM_DESTROY 消息外，switch/case 块中还可以处理其它任何消息。窗口过程没有处理的消息，都会由系统的一个预定义 API 方法（DefWindowProc）做默认处理（NO.3 处）。

> 注："将消息投递给窗口过程"或者"将消息发送给窗口过程"意思是指，以消息作为参数，调用窗口过程。

这里要注意一下上面一段文字中，对 WM_PAINT 和 WM_DESTROY 的描述差异。前者是消息循环从消息队列中取出来的，然后通过 DispatchMessage 方法将其投递给窗口过程，而后者是直接发送给窗口过程，这两个消息的处理流程明显不一样，一个需要经过消息循环，而一个则直接被处理。在 Windows 中，其实将消息分成了两类，一类需要存入消息队列，然后由消息循环取出来之后才能被窗口过程处理的，称之为"队列消息"（Queued Message），这类消息主要包括用户的鼠标键盘输入消息、绘制消息 WM_PAINT、退出消息 WM_QUIT 以及时间消息 WM_TIMER；另一类不需要存入消息队列，也不经过消息循环，它们直接传递给窗口过程，由窗口过程直接处理，称之为"非队列消息"（Nonqueued Message），当操作系统想要告诉窗口发生了某件事时，它会给窗口发送一个非队列消息，比如当我们使用 SetWindowPos API 移动窗口后，系统自动会发送一个 WM_WINDOWPOSCHANGED 消息给该窗口的窗口过程，告诉它位置发生变化了。这样一来，前面的一张 Windows 消息循环结构图可以更新为：

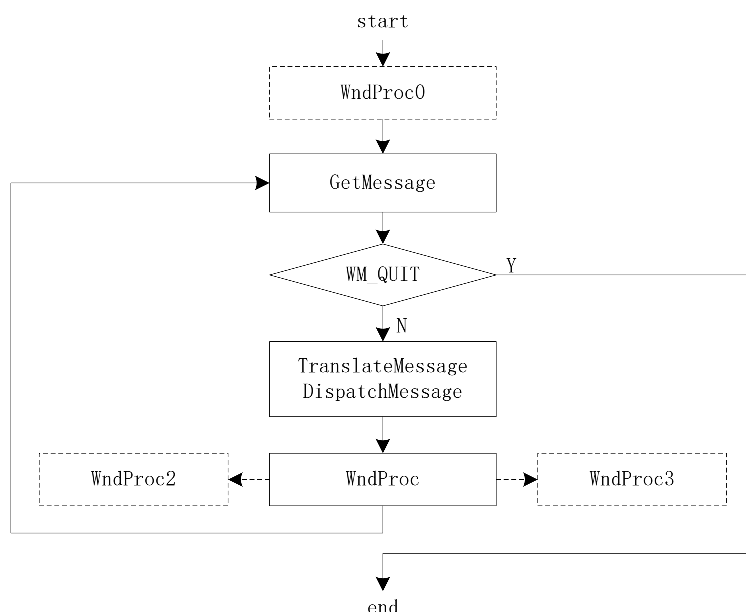


图 8-5 Windows 消息循环

如上图 8-5 所示，虚线框表示窗口过程在处理非队列消息。在消息循环外部，可以发送一个非队列消息给某个窗口的窗口过程（如 WndProc0），在消息循环内部，比如在处理队列消息的 WndProc 中，也可以发送一个非队列消息给某个窗口的窗口过程（如 WndProc2 或 WndProc3，当然也包含自己），这就出现一个有趣的现象，会经常出现嵌套调用（相同的/不同的）窗口过程好几次，但是不管嵌套调用多少次，这个数目一定是有限的，也就是说，单独的某一次 While 循环必须在有限的时间内完成，否则，下一次消息循环不能及时开始，消息队列中的队列消息就不能及时处理。

窗口过程是 Windows 应用程序中的核心部分，所有的消息均由窗口过程处理。

> 注：有些 WIN32 API 在执行过程中（还未返回），会发送非队列消息，直接调用某个窗口的窗口过程，文中所说的"系统会发送非队列消息"包含这些。

8.2.4 创建基于 Win32 的单窗体应用程序

前面讲了一个传统 Win32 应用程序的各个组成部分，本小节将用 C++代码继续说明一个完整的传统 Win32 应用程序应该包含哪些内容，本程序只包含一个主窗体。

（1）添加头文件。添加 include 语句，包含下列头文件：

```
//Code 8-4
#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <tchar.h>
```

（2）添加程序入口方法。像其它每个应用程序一样，都必须有一个程序的入口方法，每个基于 Win32 的应用程序入口方法为 WinMain，声明如下：

```
//Code 8-5
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow);
```

如上代码 Code 8-5 所述，WinMain 方法包含四个参数，表示程序启动时，操作系统传递给它的参数。

（3）声明窗口过程。一个程序至少有一个窗体，那么至少就应该有一个窗口过程，声明如下：

```
//Code 8-6
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
```

（4）注册窗体类。在创建一个窗体之前，必须先向操作系统注册一个窗体类（如果已经注册，那么就不需要），代码如下：

```
//Code 8-7
    WNDCLASSEX wcex;        //NO.1
    wcex.cbSize = sizeof(WNDCLASSEX);
    wcex.style          = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc    = WndProc;    //NO.2
    wcex.cbClsExtra     = 0;
```

```
	wcex.cbWndExtra		= 0;
	wcex.hInstance		= hInstance;
	wcex.hIcon			= LoadIcon(hInstance, MAKEINTRESOURCE(IDI_APPLICATION));
	wcex.hCursor		= LoadCursor(NULL, IDC_ARROW);
	wcex.hbrBackground	= (HBRUSH)(COLOR_WINDOW+1);
	wcex.lpszMenuName	= NULL;
	wcex.lpszClassName	= szWindowClass;
	wcex.hIconSm		= LoadIcon(wcex.hInstance,
MAKEINTRESOURCE(IDI_APPLICATION));
	if (!RegisterClassEx(&wcex))	//NO.3
	{
		MessageBox(NULL,
			_T("Call to RegisterClassEx failed!"),
			_T("Win32 Guided Tour"),
			NULL);
		return 1;
	}
```

如上代码 Code 8-7 所述，注册窗体类之前，我们需要填充一个 WNDCLASSEX 结构体（NO.1 处），它代表我们向操作系统注册的窗体类所包含的信息，比如这类窗体的共同样式、图标以及窗口过程（NO.2 处）等，然后使用 RegisterClassEx 这个 API 方法向系统注册窗体类。

> 注：这里的窗体类跟我们平时所用到的类（CLASS）不是同一个东西，前者并没有"面向对象"的含义，但是它们的作用可以类比，第一，都可以创建实例；第二，实例与实例之间的表现行为都相似；第三，创建实例都是从"抽象"到"具体"的过程。

（5）创建主窗体。为应用程序创建一个主窗体，需要用到 CreateWindow 这个 API 方法，代码如下：

```
//Code 8-8
HWND hWnd = CreateWindow(	//NO.1
	szWindowClass,
	szTitle,
	WS_OVERLAPPEDWINDOW,
	CW_USEDEFAULT, CW_USEDEFAULT,
	500, 300,
	NULL,
	NULL,
	hInstance,
	NULL
);
if (!hWnd)
{
	MessageBox(NULL,
		_T("Call to CreateWindow failed!"),
		_T("Win32 Guided Tour"),
		NULL);
```

```
    return 1;
}
```

如上代码 Code 8-8 所述，使用 CreateWindow 方法创建一个窗体（NO.1 处），需要给定窗体类 szWindowsClass，以及其它一些窗体属性，方法返回新创建窗体的句柄。

这里的创建一个窗体，就是用上一步注册的窗体类实例化出来一个窗体实例，虽然它跟面向对象中的类对象不同，但是仍可以这样去理解，使用同一个窗体类创建出来的窗体表现行为相似，并且具有相同的窗口过程。

（6）显示窗体。窗体创建完成后，我们需要调用 ShowWindow 这个 API 方法将它显示到屏幕，然后调用 UpdateWindow 这个 API 方法更新窗体界面。代码如下：

```
//Code 8-9
ShowWindow(hWnd,
        nCmdShow);
UpdateWindow(hWnd);
```

（7）Windows 消息循环。窗体显示出来之后，我们需要处理用户输入、刷新等消息，因此，需要有一个消息循环，消息循环结构如下：

```
//Code 8-10
BOOL bRet;
MSG msg;
while( (bRet = GetMessage( &msg, hWnd, 0, 0 )) != 0)
{
    if (bRet == -1)
    {
        // handle the error and possibly exit
    }
    else
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

（8）编写窗口过程。最开始我们已经声明过了窗口过程 WndProc，现在我们需要编写它的具体实现，我们在 WndProc 中编写代码，让窗体界面显示“Hello，World！”文本，代码如下：

```
//Code 8-11
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    PAINTSTRUCT ps;
    HDC hdc;
    TCHAR greeting[] = _T("Hello, World!");
    switch (message)
    {
    case WM_PAINT:
        hdc = BeginPaint(hWnd, &ps);
        TextOut(hdc,
```

```
                5, 5,
                greeting, _tcslen(greeting));    //NO.1
            EndPaint(hWnd, &ps);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);    //NO.2
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);    //NO.3
            break;
        }
        return 0;
}
```

如上代码 Code 8-11 所述，NO.1 处调用系统 API 在窗体界面输出文本“Hello World！”，当窗体关闭后，向消息队列发送 WM_QUIT 消息（NO.2 处），窗口过程没有处理的其它消息，都交给系统预定义 API（DefWindowProc）进行处理（NO.3 处）。

由于同一窗体类的不同窗体公用同一个窗口过程，所以不同窗体的表现行为相似，比如都会显示“Hello，World！”字样。我们开发 Windows 应用程序时，理论上就是根据需要注册不同的窗体类，为每个窗体类编写窗口过程，然后根据需要创建许许多多的窗体。

> 注：控件也属于窗体的一种，包含有窗口过程等等。

下面为完整代码：

```
//Code 8-12
#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <tchar.h>

static TCHAR szWindowClass[] = _T("win32app");
static TCHAR szTitle[] = _T("Win32 Guided Tour Application");
HINSTANCE hInst;
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow)    //NO.1
{
        WNDCLASSEX wcex;
        wcex.cbSize = sizeof(WNDCLASSEX);
        wcex.style          = CS_HREDRAW | CS_VREDRAW;
        wcex.lpfnWndProc    = WndProc;
        wcex.cbClsExtra     = 0;
        wcex.cbWndExtra     = 0;
        wcex.hInstance      = hInstance;
```

```
	wcex.hIcon			= LoadIcon(hInstance, MAKEINTRESOURCE(IDI_APPLICATION));
	wcex.hCursor		= LoadCursor(NULL, IDC_ARROW);
	wcex.hbrBackground	= (HBRUSH)(COLOR_WINDOW+1);
	wcex.lpszMenuName	= NULL;
	wcex.lpszClassName	= szWindowClass;
	wcex.hIconSm		= LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_APPLICATION));
	if (!RegisterClassEx(&wcex))	//NO.2
	{
		MessageBox(NULL,
			_T("Call to RegisterClassEx failed!"),
			_T("Win32 Guided Tour"),
			NULL);
		return 1;
	}
	hInst = hInstance; // Store instance handle in our global variable
	HWND hWnd = CreateWindow(	//NO.3
		szWindowClass,
		szTitle,
		WS_OVERLAPPEDWINDOW,
		CW_USEDEFAULT, CW_USEDEFAULT,
		500, 300,
		NULL,
		NULL,
		hInstance,
		NULL
	);
	if (!hWnd)
	{
		MessageBox(NULL,
			_T("Call to CreateWindow failed!"),
			_T("Win32 Guided Tour"),
			NULL);
		return 1;
	}
	ShowWindow(hWnd,	//NO.4
		nCmdShow);
	UpdateWindow(hWnd);	//NO.5
	MSG msg;
	while (GetMessage(&msg, NULL, 0, 0))	//NO.6
	{
		TranslateMessage(&msg);
		DispatchMessage(&msg);
	}
	return (int) msg.wParam;
}
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
```

```
//NO.7
{
	PAINTSTRUCT ps;
	HDC hdc;
	TCHAR greeting[] = _T("Hello, World!");
	switch (message)
	{
	case WM_PAINT:	//NO.8
		hdc = BeginPaint(hWnd, &ps);
		TextOut(hdc,
			5, 5,
			greeting, _tcslen(greeting));
		EndPaint(hWnd, &ps);
		break;
	case WM_DESTROY:	//NO.9
		PostQuitMessage(0);
		break;
	default:
		return DefWindowProc(hWnd, message, wParam, lParam);	//NO.10
		break;
	}
	return 0;
}
```

如上代码 Code 8-12 所述，一个完整的 Win32 程序，应该有一个程序入口方法 WinMain（NO.1 处），创建一个窗体之前，我们需要先注册窗体类（如果还没有注册，NO.2 处），指定了该窗体类的窗口过程为 WndProc，窗体类注册成功后，我们使用新注册的窗体类创建了一个窗体（NO.3 处），之后将窗体显示出来（NO.4. NO.5 处）到屏幕，到此还没有结束，因为我们还不能处理消息队列中的消息，因此我们还得添加一个消息循环（NO.6 处）。在窗口过程中，我们处理 WM_PAINT 消息时，将字符串显示到窗体界面（NO.8 处），窗体关闭时，系统会发送一个非队列消息 WM_DESTROY，在处理这个 WM_DESTROY 时，我们向消息队列发送一个 WM_QUIT 消息（NO.9 处），下次消息循环时，碰到消息队列中的 WM_QUIT，循环退出。每次没有处理的消息均交给系统默认 API 处理（NO.10 处）。

注意窗口过程的调用不是显示的，也就是说，我们在代码中根本看不见调用窗口过程的代码，在需要处理消息（无论是队列消息还是非队列消息）时，系统会自动以消息作为参数，调用对应的窗口过程，而这些是不受开发者控制的，正因为如此，整个程序的运行流程不是特别明了，需要仔细揣摩，下面一张图解释了本小节实例代码的完整运行流程：

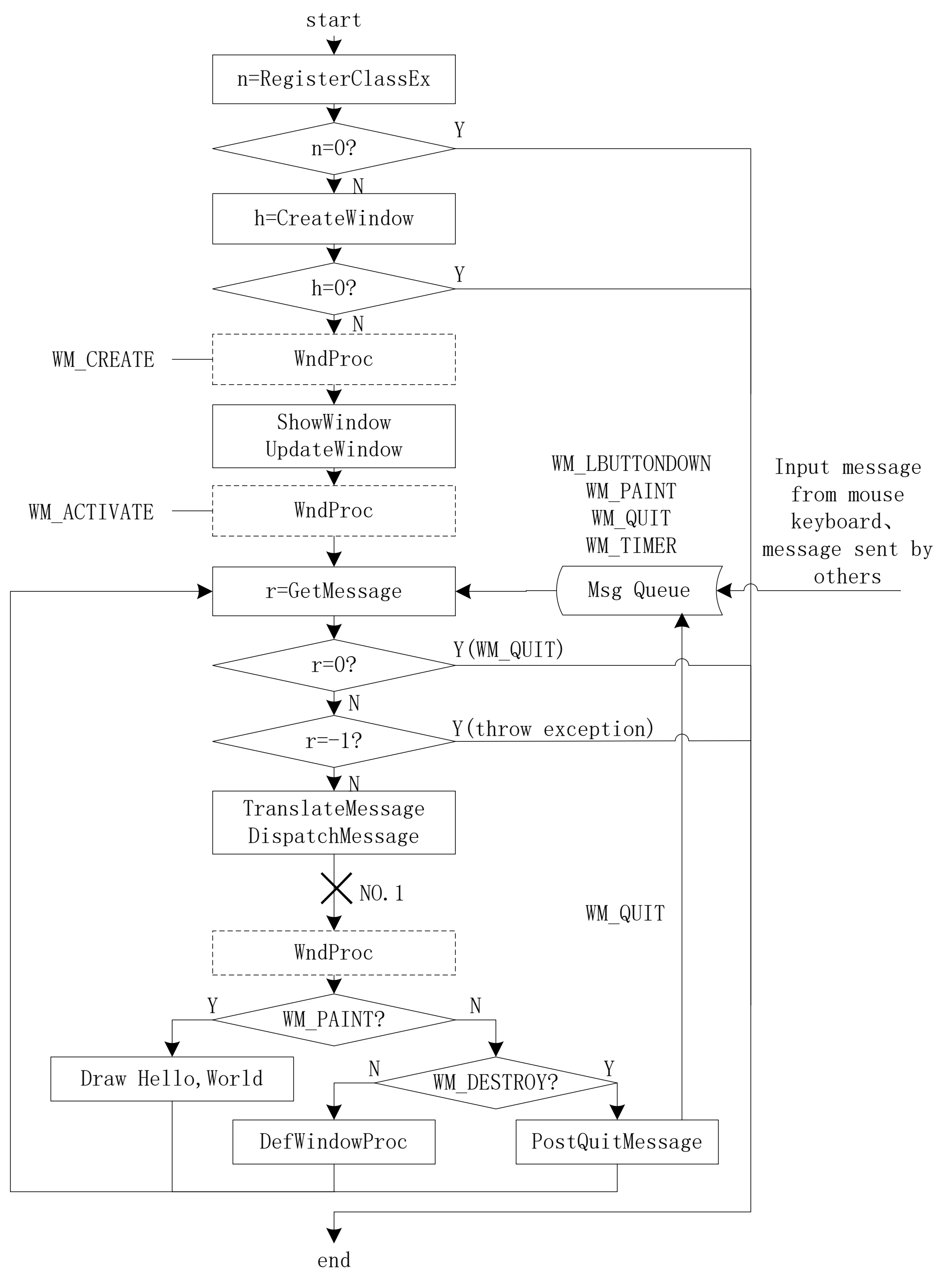


图 8-6 8.2.4 小节示例代码运行流程

如上图 8-6 所示，我们在创建窗体时（CreateWindow 还未返回），会发送类似 WM_CREATE 这样的非队列消息给新创建出来的窗体，告诉窗体：我们已经创建好了，同理，显示窗体等其它操作，也有可能给窗体发送类似 WM_ACTIVATE 这样的非队列消息，告诉窗体：我们已经激活了，这些过程都会调用对应窗体的窗口过程。进入消息循环后，消息循环不断地从消息队列中获取队列消息，然后由 DispatchMessage 方法投递给对应的窗口过程，在窗口过程中，我们处理了 WM_PAINT 消息，将字符串输出到窗体界面。注意图中 NO.1 处的箭头打了一个“叉”，因为下面的 WndProc 不一定只在 DispatchMessage 后面才调用，在其它的地方也有可能被调用，比如处理非队列消息的时候。程序运行效果图如下：

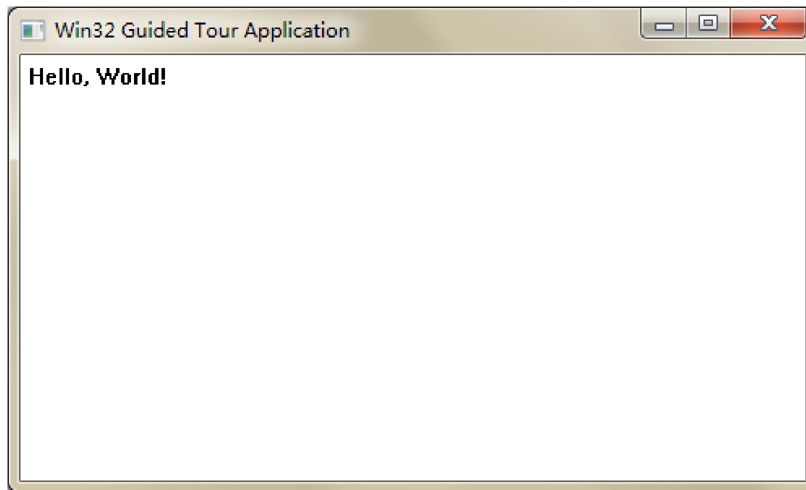


图 8-7 8.2.4 小节示例代码运行效果图

8.2.5 创建基于 Win32 的多窗体应用程序

上一小节中演示了怎么创建只包含一个主窗体的 Win32 应用程序，但是现实生活中，我们的程序不可能仅仅包含一个窗体。我们点击一个按钮，就有可能打开一个新的窗体，本小节将演示在 8.2.4 小节的基础上，怎么在程序中创建多个窗体。

上一小节中，我们是在消息循环（While 循环）之前创建的主窗体，其实可以在任何地方使用 CreateWindow 创建窗体，下面示例代码演示怎么在主窗体的鼠标右键按下消息中创建另外一个新窗体：

```
//Code 8-13
#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <tchar.h>

static TCHAR szWindowClass[] = _T("win32app");
static TCHAR szWindowClass_child[] = _T("win32app_child");
static TCHAR szTitle[] = _T("Win32 Guided Tour Application");
HINSTANCE hInst;
int nCmdS;
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK WndProc2(HWND, UINT, WPARAM, LPARAM);    //NO.1

int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow)    //NO.2
{
    WNDCLASSEX wcex;
    wcex.cbSize = sizeof(WNDCLASSEX);
    wcex.style          = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc    = WndProc;    //NO.3
```

```
        wcex.cbClsExtra         = 0;
        wcex.cbWndExtra         = 0;
        wcex.hInstance          = hInstance;
        wcex.hIcon              = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_APPLICATION));
        wcex.hCursor            = LoadCursor(NULL, IDC_ARROW);
        wcex.hbrBackground      = (HBRUSH)(COLOR_WINDOW+1);
        wcex.lpszMenuName       = NULL;
        wcex.lpszClassName      = szWindowClass;
        wcex.hIconSm            = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_APPLICATION));
        if (!RegisterClassEx(&wcex))    //NO.4
        {
            MessageBox(NULL,
                _T("Call to RegisterClassEx failed!"),
                _T("Win32 Guided Tour"),
                NULL);
            return 1;
    }
        wcex.lpfnWndProc        = WndProc2;    //NO.5
        wcex.lpszClassName      = szWindowClass_child;
        if (!RegisterClassEx(&wcex))    //NO.6
        {
            MessageBox(NULL,
                _T("Call to RegisterClassEx failed!"),
                _T("Win32 Guided Tour"),
                NULL);
            return 1;
    }
    hInst = hInstance; // Store instance handle in our global variable
        nCmdS = nCmdShow;
        HWND hWnd = CreateWindow(    //NO.7    create main window
            szWindowClass,
            szTitle,
            WS_OVERLAPPEDWINDOW,
            CW_USEDEFAULT, CW_USEDEFAULT,
            500, 300,    //main window    width=500,height=300
            NULL,
            NULL,
            hInstance,
            NULL
        );
        if (!hWnd)
        {
            MessageBox(NULL,
                _T("Call to CreateWindow failed!"),
                _T("Win32 Guided Tour"),
                NULL);
```

```
        return 1;
    }
    ShowWindow(hWnd,    //NO.8
        nCmdShow);
    UpdateWindow(hWnd);    //NO.9
    MSG msg;
    while (GetMessage(&msg, NULL, 0, 0))    //NO.10
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return (int) msg.wParam;
}
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
//NO.11
{
    PAINTSTRUCT ps;
    HDC hdc;
TCHAR greeting[] = _T("Hello, World!");
    HWND hWnd2 = 0;
    switch (message)
    {
    case WM_PAINT:    //NO.12
        hdc = BeginPaint(hWnd, &ps);
        TextOut(hdc,
            5, 5,
            greeting, _tcslen(greeting));
        EndPaint(hWnd, &ps);
        break;
    case WM_LBUTTONDOWN:    //NO.13
        hWnd2 = CreateWindow(    //NO.14
        szWindowClass_child,
        szTitle,
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT,
        400, 200,    //child window      width=400,height=200
        NULL,
        NULL,
        hInst,
        NULL
        );
          if (!hWnd2)
        {
            MessageBox(NULL,
                _T("Call to CreateWindow failed!"),
                  _T("Win32 Guided Tour"),
```

```
                NULL);
            return 1;
        }
        ShowWindow(hWnd2,    //NO.15
        nCmdS);
        UpdateWindow(hWnd2);    //NO.16
        break;
    case WM_DESTROY:    //NO.17
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);    //NO.18
        break;
    }
    return 0;
}
LRESULT CALLBACK WndProc2(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
//NO.19
{
    PAINTSTRUCT ps;
    HDC hdc;
    TCHAR greeting[] = _T("Hello, World! I am child window");
    switch(message)
    {
        case WM_PAINT: //NO.20
            hdc = BeginPaint(hWnd, &ps);
            TextOut(hdc,
                5, 5,
                greeting, _tcslen(greeting));
            EndPaint(hWnd, &ps);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);    //NO.21
            break;
    }
}
```

如上代码 Code 8-13 所述，在上一小节的基础上，我们又注册了另一个窗体类（窗体类名称为 szWindowClass_child，窗口过程为 WndProc2，见 NO.5. NO.6 处），我们处理主窗体的 WM_LBUTTONDOWN 消息时，用新注册的窗体类创建了一个子窗体，并将其显示到屏幕（NO.14. NO.15 以及 NO.16 处），这个过程会给子窗体发送非队列消息（调用 WndProc2，直接处理）。无论是主窗体还是子窗体，都共用同一个消息循环，整个示例代码的运行流程如下：

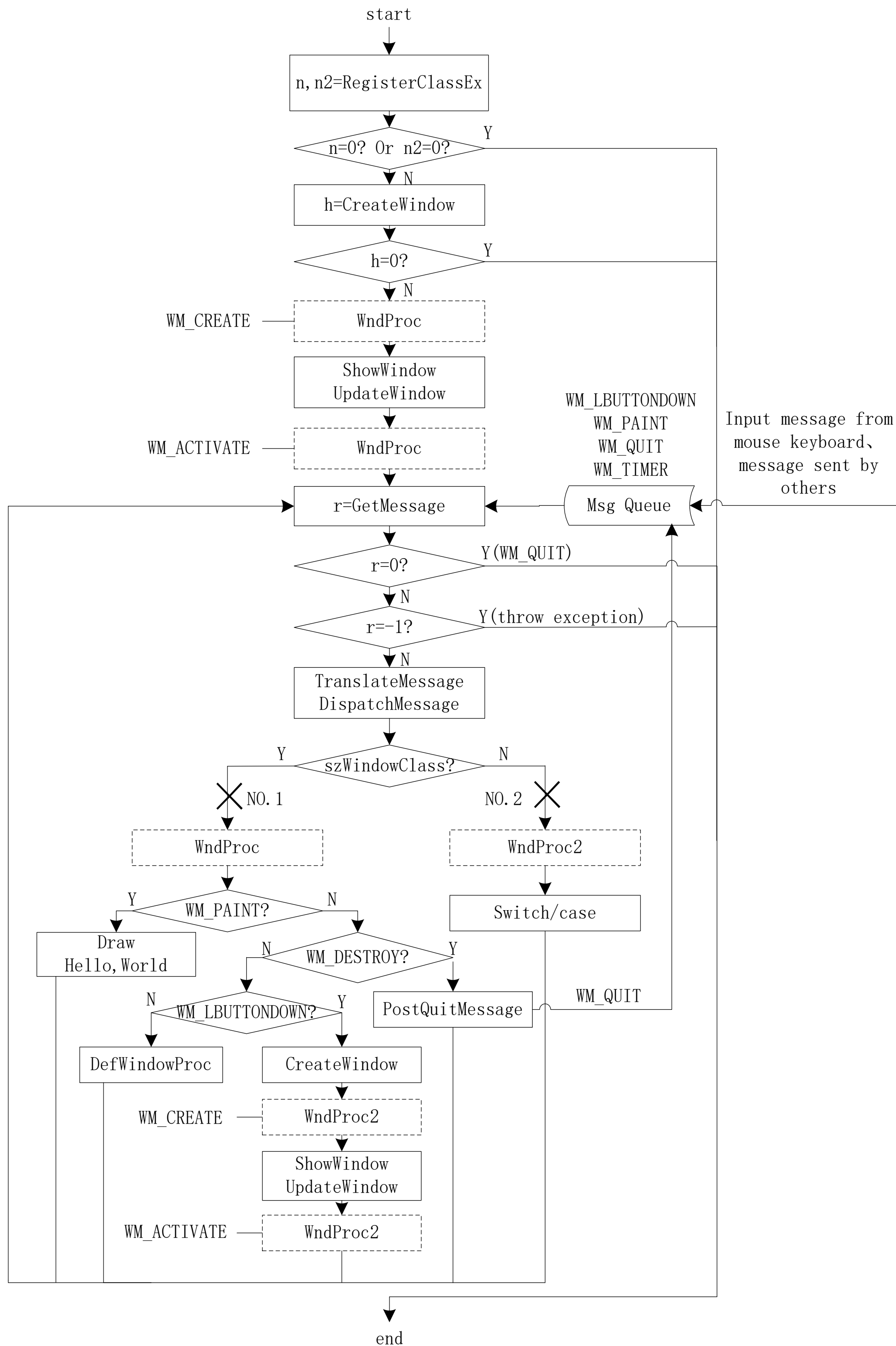


图 8-8 8.2.5 小节示例代码运行流程

如上图 8-8 所示，我们在 switch/case 中处理主窗体的 WM_LBUTTONDOWN 消息时，创建了一个新窗体，在创建新窗体的过程中，会调用好几次（图中有省略）新窗体的窗口过程（WndProc2），可以看出，调用 WndProc2 都是发生在消息循环中，而且是在处理主窗体队列消息的过程中。另外，新创建出来的窗体（无论多少个），都是共用同一个消息循环，DispatchMessage 会根据窗口句柄，将消息投递到对应的窗口过程。运行效果图如下：

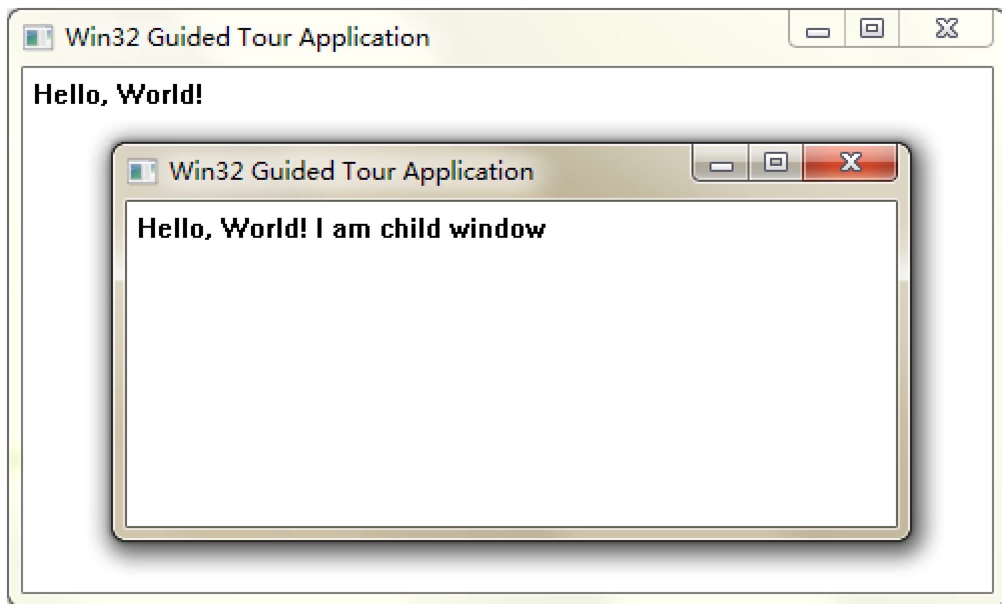


图 8-9 8.2.5 小节示例代码运行效果图

整个程序都是以“处理队列消息”为主线，在处理队列消息的过程中，发送非队列消息给别人（或者自己）。图 8-8 基本上可以表示一个 Windows 桌面应用程序的完整运行流程，无论我们的程序是使用 MFC、Windows Forms 还是其它框架开发完成，内部流程都和直接使用 Win32 API 开发的一样。

注：

[1]本章示例代码参考 MSDN。网址：

HTTP://MSDN.MICROSOFT.COM/ZH-CN/LIBRARY/BB384843(V=VS.100).ASPX

[2]一些 WIN32 API 在调用过程中，会给它操作的窗体发送非队列消息，比如 CREATEWINDOW 和 SHOWWINDOW 等 API。调用 CREATEWINDOW 创建窗体时会向新创建的窗体发送以下非队列消息：WM_GETMINMAXINFO、WM_NCCREATE、WM_NCCALCSIZE、WM_CREATE，而 SHOWWINDOW 显示窗体时，它会向显示的窗体发送以下非队列消息：WM_SHOWWINDOW、WM_WINDOWPOSCHANGING、WM_WINDOWPOSCHANGING、WM_ACTIVATEAPP、WM_NCACTIVATE、WM_GETTEXT、WM_ACTIVATE、WM_IME_SETCONTEXT、WM_IME_NOTIFY、WM_SETFOCUS、WM_NCPAINT、WM_GETTEXT、WM_ERASEBKGND、WM_WINDOWPOSCHANGED、WM_SIZE、WM_MOVE，由此可见，窗体的窗口过程几乎无时无刻都在被调用。

8.3 .NET Winform 程序与传统 Win32 程序的关联

Winform 应用程序是.NET 平台中使用 Windows Forms 框架开发出来的 Windows 桌面应用

程序，它属于一种托管程序，必须运行在.NET 平台中。类似 MFC，Windows Forms 框架内部也是通过原始 Win32 API 实现的，那么，Winform 应用程序和传统 Win32 应用程序有哪些相同点和不同点呢？

相同点：

（1）两者都是运行在 Windows 平台中的 Windows 桌面应用程序；

（2）两者都是通过（直接/间接）调用 Win32 API 来实现完成。

不同点：

（1）前者属于托管程序，需要.NET 平台支持，为托管时代的产物，后者属于非托管程序；

（2）前者使用.NET Frameworks（Windows Forms 部分）框架开发完成，间接地调用 Win32 API，后者直接使用 Win32 API 开发完成。

在传统 Win32 应用程序中，窗体数据和方法都是分开的，我们只能通过 Win32 API 去操作窗体，去更新窗体的属性，窗体的窗口过程也是作为一个全局方法独立存在的，在传统 Win32 开发模式中，没有用到“面向对象”思想，并且消息循环、Windows 消息处理（编写 switch/case 块）都得开发人员自己动手。Windows Forms 框架将.NET、面向对象以及 Windows 应用程序开发三者结合一起，帮我们完成了消息循环、Windows 消息处理（将处理消息转换成激发事件），让我们把精力放在业务逻辑处理上，它还将窗体数据和操作窗体的一些 API 方法封装到类中，数据成了属性，API 方法成了成员方法，并且充分利用了面向对象编程的一些优势。Windows Forms 框架的具体介绍请参见本章后续内容。

8.4 Windows Forms 框架

Windows Forms 是.NET 框架中的一个分支，主要用于桌面应用程序的开发，编程中常用类型包含在 System.Windows.Forms.dll 程序集中的 System.Windows.Form 命名空间中。8.3 节末尾讲到，Windows Forms 框架底层是通过 Win32 API 实现的，并且充分利用了面向对象编程特性，比如继承、多态等。

在传统 Win32 开发模式中，窗体数据、操作窗体的方法以及窗体的窗口过程等都是相互独立的，没有面向对象的概念。如果一个应用程序只考虑有一个 UI 线程，那么传统 Win32 开发模式中，程序的数据结构类似如下图 8-10：

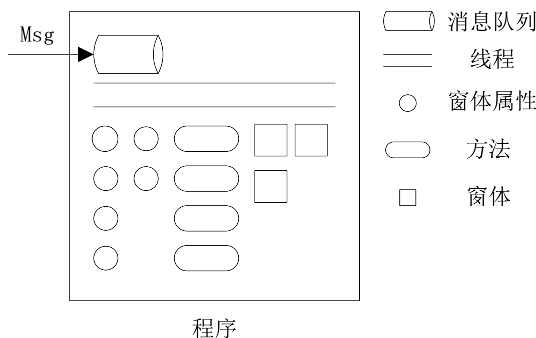


图 8-10 传统 Win32 程序结构

而在使用 Windows Forms 框架开发的 Winform 程序中，窗体（控件）的数据、方法等都是以一个整体出现的（类对象），如果一个 Winform 应用程序只考虑有一个 UI 线程，Winform 程序的数据结构如下：

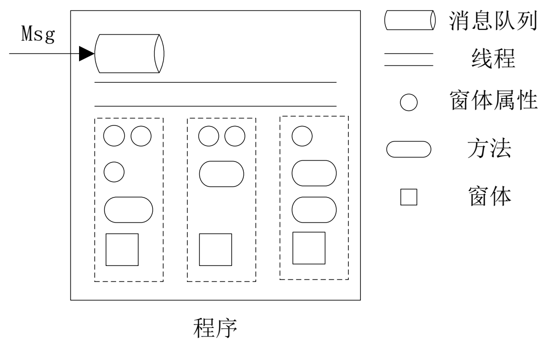


图 8-11 Winform 程序结构

如上图 8-11，图中虚线框代表一个整体，代码中常用对象来表示，窗体的属性以及与窗体有关的方法都封装在了一起，窗体的窗口过程也成了成员方法，在 Windows Forms 框架中，以 Control 为基类，其它所有与窗体显示有关的控件几乎都派生自它，Control 类中的 WndProc 虚方法就是我们在 Win32 开发模式中所熟悉的窗口过程。另外，前面也讲到过，窗体和控件本质上是一个东西，只是它们有着不同的属性，所以我们可以看到，窗体类 Form 间接派生自 Control 类。Win32 开发中，向操作系统注册的窗体类，类似 Windows Forms 框架中的 Control（或其派生类，下同）类，Win32 开发中通过 CreateWindow API 创建一个窗体，类似在 Winform 程序中实例化一个 Control 类对象，两者都是从“抽象”到“具体”的过程。

> 注：我们可以把一个 WINDOWS 桌面应用程序分成五个部分：1）消息队列，负责缓存即将被当前线程处理的队列消息；2）UI 线程，包含消息循环结构，是应用程序中处理所有与界面有关逻辑的线程；3）窗体，这里的窗体可以看作"窗体句柄"，系统中区分窗体的唯一标示；4）窗体属性，与窗体相关的属性，比如背景色、字体等等；5）方法，操作窗体的方法，比如移动、创建窗体等，当然也包括窗口过程。而在 WINDOWS FORMS 框架中，窗体唯一标示句柄、属性以及方法都封装在了一起（称为控件，CONTROL），因此我们可以认为，WINFORM 程序中包含三个部分：消息队列、UI 线程以及控件。

本章前面也讲到过，Windows Forms 框架已经帮我们完成了消息循环，在我们开发过程中，不再需要我们自己去编写消息循环，事实上，我们在 Winform 程序中也确实都没见过消息循环的影子，我们只需要把精力集中在消息处理上，也就是对窗口过程（switch/case）的完善，但是在大多数情况下，我们并不需要去接触 Control（或其派生类，下同）类中的 WndProc 这个方法，原因是什么呢？原因就是在 Control 类的 WndProc 方法中已经做了默认处理，它将 Windows 消息以及消息携带的参数直接转换成了.NET 中的事件和事件参数，当消息到达时，Control 类对象会激发与该消息对应的事件，从而通知事件注册者。当然必要时，我们完全可以在 Control 类的派生类中重写 WndProc，它包含一个 Message 结构体类型的参数，与 Win32 开发中的 Msg 结构体类似，表示一个 Windows 消息，重写 WndProc 虚方法时，我们可以从源头接触到 Windows 消息。

使用 Windows Forms 框架开发程序时，开发者主要负责的部分如下图 8-12：

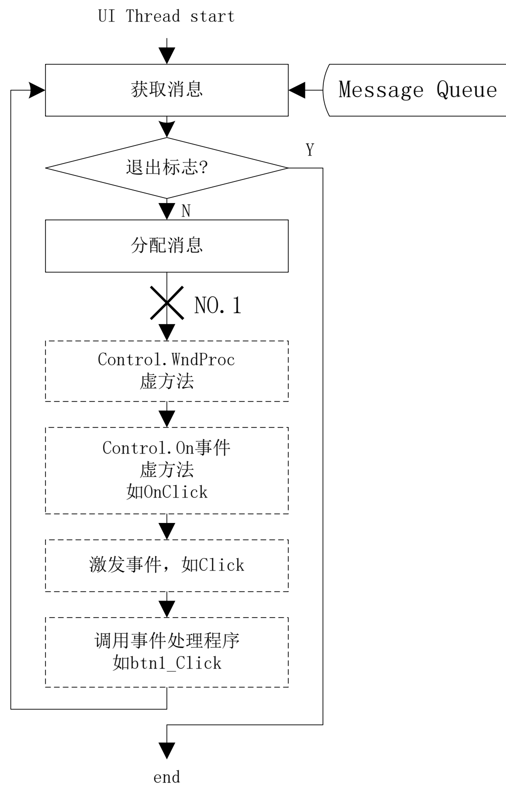


图 8-12 开发者在 Winform 程序中负责部分

如上图 8-12 所示，虚线框为开发者负责部分，可以看出，我们编写的代码，都是被窗口过程 WndProc 回调。需要注意图中 NO.1 处的箭头有一个“叉”，因为控件的窗口过程并不止在控件处理队列消息的时候调用，还可能在处理非队列消息的时候调用（参见本章前面，窗口过程并不都是在 DispatchMessage 之后调用）。

System.Windows.Forms 命名空间中的类型主要分以下几类：

1）控件，以 Control 类为基类，主要负责界面显示；

2）委托，控件激发事件的委托类型；

3）事件参数，控件激发事件时传递的参数；

4）枚举，跟控件显示有关的属性值；

其它功能类，比如 Application 类，控制整个 Winform 程序。

详细信息参见下表 8-1：

表 8-1　Windows Forms 中类型的分类

类别	举例	说明
控件	Control、Button、Form	主要负责界面显示，与 Win32 开发中，向操作系统注册的各种各样的窗体类相似
委托	PaintEventHandler、MouseEventHandler FormClosingEventHandler	控件会将 Windows 消息转换成事件，这些事件所属的委托类型（关于命名规则，参见第五章）
事件参数	PaintEventArgs、MouseEventArgs FormClosingEventArgs	控件激发事件时，传递的参数（关于命名规则，参见第五章）
枚举	AnchorStyles、BorderStyle	一些控件属性的枚举值

	FormWindowState	
功能类	Application	负责 UI 线程中消息循环的启动等与应用程序有关的操作

8.5 Winform 程序结构

在.NET 中，使用 Windows Forms 框架开发的 Windows 桌面应用程序称为 Winform 应用程序，本章前面讲到过，Winform 应用程序本质上跟传统 Win32 应用程序类似，也具有 UI 线程、消息循环、窗体（控件）以及窗口过程等元素。本节将从四个方面介绍我们曾以为比较熟知的 Winform 程序。

8.5.1 UI 线程

在每个 Winform 程序的 Program.cs 文件中，都有一个 Main 方法，该 Main 方法就是程序的入口方法，每个程序启动时都会以 Main 方法为入口，创建一个线程，这个线程就是 UI 线程，可能你会问 UI 线程怎么没有消息循环呢？那是因为 Main 方法中总是会出现一个类似 Application.Run 的方法，而消息循环隐藏在了该方法内部，具体参见下一小节。一个程序理论上可以有多个 UI 线程，每个线程有自己的消息队列（由操作系统维护）、消息循环、窗体等，下图 8-13 显示了包含多个 UI 线程的应用程序：

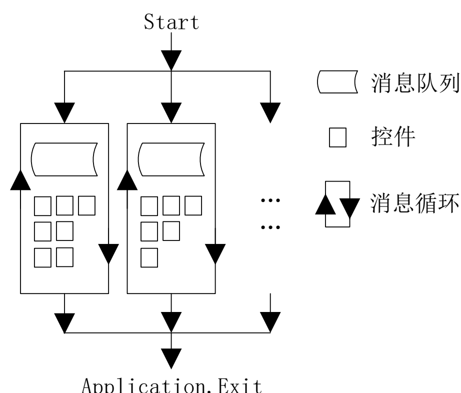


图 8-13 包含多个 UI 线程的 Winform 程序

如上图 8-13 所示，每个 UI 线程都有自己的消息队列、消息循环以及自己创建的控件，相互之间不干扰，只有程序中所有的 UI 线程结束以后，整个程序才会结束（Exit）。由于 UI 线程之间的数据交换比较复杂，因此在实际开发中，在没有特殊需求下，一个程序一般只包含有一个 UI 线程。

Winform 程序中，Application 类代表一个程序，该类几乎只包含静态成员，类似全局成员的作用，给整个应用程序提供帮助。Application 类中包含一个内嵌（Nested Type）类 ThreadContext，该类没有对外公开，因此在 MSDN 上找不到与它相关的资料，ThreadContext 类代表一个 UI 线程，因为一个程序可以包含多个 UI 线程，所以一个程序中可以有多个 ThreadContext 类对象。Windows Forms 框架中通过使用一个容器（哈希表）来存储程序中的 ThreadContext 对象，但每次需要用到容器中某个 ThreadContext 对象时，并不是像我们平时开发过程中通过一个 key 去获取 value 值的方式，而是通过 ThreadContext 类的一个静态方法 ThreadContext.FromCurrent()来获取与当前线程关联的 ThreadContext 对象，类似以下代码，我们在 DoSomething 方法中需要用到与当前线程关联的 ThreadContext 对象时：

```
//Code 8-14
```

```
// Application class is Defined below
public sealed class Application
{
    //...
    // use ThreadContext in Application class
    public static void DoSomething()
    {
        //...
        Application.ThreadContext tc = Application.ThreadContext.FromCurrent();    //NO.1
        //use tc
    }

    //define ThreadContext
    internal sealed class ThreadContext    //NO.2
    {
        //...
    }
}
class Program
{
    static void Main()
    {
        Application.DoSomething(); //NO.3
    }
}
```

如上代码 Code 8-14 所示，它说明了 Windows Forms 框架中 Application 类和 ThreadContext 类之间的定义关系，声明为 internal 的 ThreadContext 类（NO.2 处）不能在外部使用，我们只需要使用 Application 类就可以，在示例代码中的 DoSomething 方法里，通过 ThreadContext.FromCurrent()方法可以获取与当前线程关联的 ThreadContext 对象（NO.1 处）。在客户端代码中，我们按照 NO.3 处那样使用 Application 类，那么 ThreadContext.FromCurrent() 就代表由 Main 方法为入口的 UI 线程。

> 注：THREADCONTEXT.FROMCURRENT()静态方法返回的 THREADCONTEXT 对象总是与当前线程相关联的，而不管该静态方法运行在哪个线程中，请参见第二章中关于"线程"的解释。

8.5.2 消息循环

上一小节中提到过，Program.cs 文件里 Main 方法中 Application.Run 方法中包含了消息循环，那么具体 Application.Run 内部是怎样实现的呢？要搞清楚这个问题，我们需要先回忆一下本章前面介绍的传统 Win32 应用程序的流程：1）创建主窗体并显示；2）启动 While 循环。没错，Application.Run 方法也是这样实现的：

```
//Code 8-15
public sealed class Application
{
```

```
//…
public static void Run(Form mainForm)    //NO.1    start message loop with mainform
{
        Application.ThreadContext.FromCurrent().RunMessageLoop(mainForm); //NO.2
}
public static void Run()    //NO.3    start message loop without mainform
{
        Application.ThreadContext.FromCurrent().RunMessageLoop(null);
}

//nested type
internal sealed class ThreadContext
{
        //use native methods
        [DllImport("user32.dll", CharSet=CharSet.Unicode, ExactSpelling=true)]
public static extern bool GetMessageW([In, Out] ref NativeMethods.MSG msg, HandleRef hWnd, int uMsgFilterMin, int uMsgFilterMax);
        [DllImport("user32.dll", CharSet=CharSet.Auto, ExactSpelling=true)]
public static extern bool TranslateMessage([In, Out] ref NativeMethods.MSG msg);
        [DllImport("user32.dll", CharSet=CharSet.Ansi, ExactSpelling=true)]
public static extern IntPtr DispatchMessageA([In] ref NativeMethods.MSG msg);

        private static int totalMessageLoopCount; //NO.4
        private int messageLoopCount; //NO.5
        internal void RunMessageLoop(Form mainForm)
        {
                totalMessageLoopCount++; //NO.6
                messageLoopCount++; //NO.7
                if(mainForm != null)
                        mainForm.Show(); //NO.8
                While(GetMessage())    //NO.9
                {
                        TranslateMessage();
                        DispatchMessage();//dispatch message to Control.WndProc virtual method
                }
                messageLoopCount--;    //NO.10
                totalMessageLoopCount--;    //NO.11
                if(messageLoopCount == 0)
                {
                        Application.RaiseThreadExit();//NO.12 raise Application.ThreadExit event
                }
                if(totalMessageLoopCount == 0) //NO.13 raise Application.Exit event
                {
                        Application.RaiseExit();
                }
        }
```

```
    }
}
```

如上代码 Code 8-15 所示，Application 类公开了两个进入消息循环的方法，一个带 Form 类型参数（NO.1 处），一个不带参数（NO.3 处），两者方法体内都只有一行代码（NO.2 处），该行代码的作用是：获取与当前线程关联的 ThreadContext 对象，然后通过获取到的 ThreadContext 对象启动消息循环。

由此看来，UI 线程中的主要逻辑都由 ThreadContext 负责，这前面也已经说过。在 ThreadContext 类中，首先声明了要用到的"本地方法"（Native Methods），这也印证了前面说到的：Windows Forms 底层是通过调用 Win32 API 来实现的。

ThreadContext 类中还定义了两个 int 类型变量，一个静态的 totalMessageLoopCount（NO.4 处），它能记录整个程序的消息循环总数，一个非静态的 messageLoopCount（NO.5 处），它只记录当前线程（当前 ThreadContext 对象）的消息循环总数。

进入 RunMessageLoop 方法后，totalMessageLoopCount 加一（NO.6 处），messageLoopCount 加一（NO.7 处），表示将要创建一个新的消息循环。

NO.9 处消息循环开始，我们可以看到，它跟传统 Win32 应用程序中的结构几乎一样。当消息循环退出后，totalMessageLoopCount 减一（NO.10 处），messageLoopCount 减一（NO.11 处），表示有一个消息循环已经结束。最后，如果 messageLoopCount 为 0，表示当前 UI 线程将要结束，因此需要激发 Application.ThreadExit 事件（NO.12 处），如果 totalMessageLoopCount 为 0，表示整个程序将要结束，因此需要激发 Application.Exit 事件（NO.13 处）。

我们在客户端代码中使用 Application.Run 方法时，直接可以这样：

```
//Code 8-16
class Program
{
    static void Main()
    {
        Application.Run(new Form());
    }
}
```

当然，我们还可以这样：

```
//Code 8-17
class Program
{
    static void Main()
    {
        new Form().Show();
        Application.Run();
    }
}
```

以上两种方式启动一个消息循环都是对的。

> 注：本小节有关 APPLICATION 类以及 THREADCONTEXT 类的源代码只是一种说明性代码，并不是 WINDOWS FORMS 中的源代码，与官方代码有一定差距，不过道理相似，可以类比。

通过对 Application 类以及 ThreadContext 类的部分代码分析，我们可以进一步明确：1）

一个程序可以包含多个 UI 线程，我们完全可以通过 System.Threading.Thread 类新创建一个普通线程，然后在该线程中调用 Application.Run 方法来开启消息循环；2）一个 UI 线程结束（该线程中的消息循环个数为 0）后，将会激发 Application.ThreadExit 事件，告知有 UI 线程结束，只有当所有 UI 线程都结束（程序中消息循环总数为 0），才会激发 Application.Exit 事件，告知应用程序退出。再看一张图，说明 Windows Forms 中消息循环的结构：

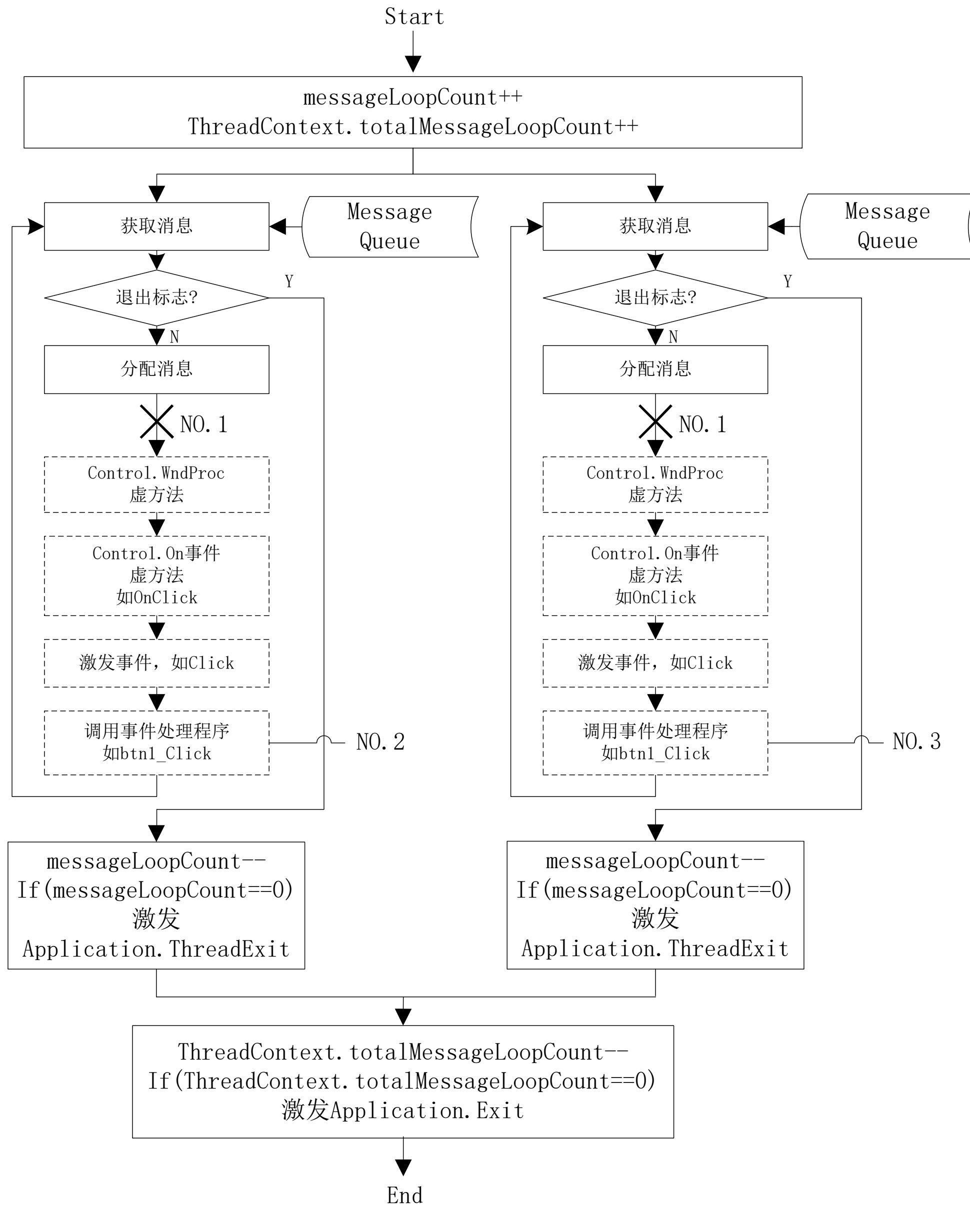


图 8-14 Windows Forms 中消息循环结构

如上图 8-14 所示，图中显示了包含两个 UI 线程的 Winform 应用程序，虚线框仍然代表开发者负责的部分。可以看到，两个 UI 线程中的消息循环相互独立，互不影响，只有两个 UI 线程都结束了以后，整个应用程序才会退出。

有心的读者可能已经看出，消息循环开始之前，messageLoopCount 加一，消息循环结束之后，messageLoopCount 减一，这样 messageLoopCount 最后肯定为 0，为什么还要去判断一下呢？这是个非常重要的问题，就目前本书中讲到了有关消息循环，都是单层循环，也就是说 while 中再没有嵌套循环，但实际情况是，在 Windows Forms 框架的这个单层消息循环内部，经常会出现嵌套循环，比如图中 NO.2 或者 NO.3 处，开发人员完全可以在 btn_Click 事件处理程序中再开启一个消息循环，嵌套消息循环多数出现在“模式对话框”中（后面有讲到）。如

果出现了嵌套消息循环，内外（或者更多个）两个消息循环是公用同一个消息队列的，这符合一个 UI 线程只有一个消息队列的规则，如下图 8-15：

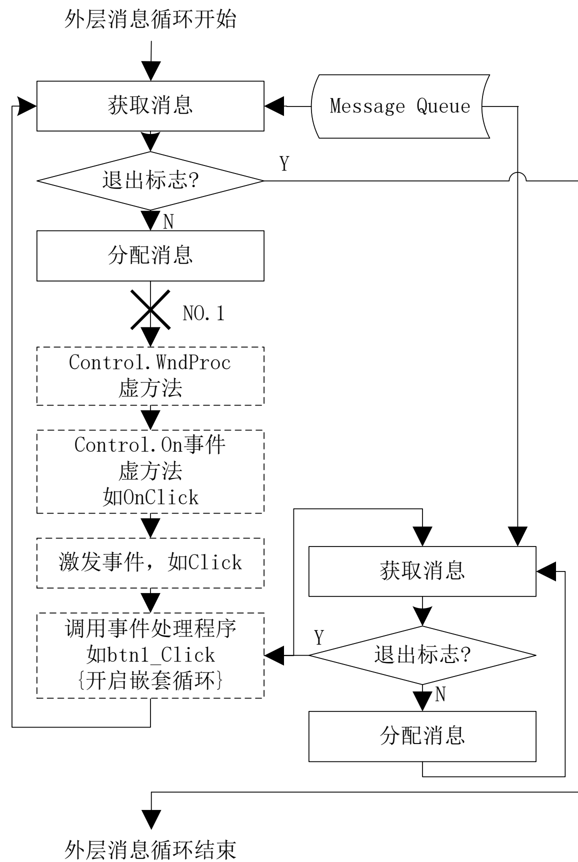


图 8-15 嵌套消息循环

如上图 8-15 所示，外层消息循环（一般 Program.cs 文件里 Main 方法中的 Application.Run 进入的消息循环都为外层消息循环）执行到 btn_Click 事件处理程序时，开启了一个消息循环，那么这个新开启的消息循环就属于内层消息循环，内层消息循环还是从同一个消息队列中获取队列消息，内层消息循环退出后，返回到 btn_Click 中，外层消息循环继续执行。注意嵌套循环理论上可以无限多次，每个循环都使用同一个消息队列。由于每个内层消息循环开始之前都会将 messageLoopCount 和 ThreadContext.totalMessageLoopCount 加一，结束之后都会将 messageLoopCount 和 ThreadContext.totalMessageLoopCount 减一，因此出现了前面所说的情况：在激发 Application.ThreadExit 和 Application.Exit 事件之前，需要通过检查 messageLoopCount 和 ThreadContext.totalMessageLoopCount 是否为 0 来决定。

嵌套消息循环一般用在模式对话框中，也就是 Form.ShowDialog()中会用到嵌套消息循环，后面有专门介绍，其它情况一般不要轻易使用。

8.5.3 创建窗体

本章前面讲到过，传统 Win32 应用程序中，给定一个窗体类，然后通过 CreateWindow 这样的 API 去创建窗体，再通过 ShowWindow 这个 API 将创建出来的窗体显示到屏幕。创建窗体的过程可以在程序的任何地方，可以出现在消息循环之前，也可以出现在消息循环之中，那么 Winform 中，我们是怎样去创建一个窗体呢？原理跟 Win32 开发中类似：先创建，后显示。在消息循环之前，我们这样创建窗体：

```
//Code 8-18
class Program
```

```
{
    static void Main()
    {
        Form f = new Form(); //NO.1
        f.Show();    //NO.2
        Application.Run(); //NO3
    }
}
```

如上代码 Code 8-18 所示，注意窗体 f 显示之后（NO.2 处），需要开启一个消息循环（NO.3 处）。在消息循环之中（窗口过程调用时），我们这样创建窗体：

```
//Code 8-19
class Form1:Form
{
    public Form1()
    {
        InitializeComponent();
        btn1.Click += new EventHandler(btn1_Click); //NO.1
    }
    private void btn_Click(object sender,EventArgs e)
    {
        Form f = new Form();
        f.Show(); //NO.2
    }
}
```

如上代码 Code 8-19 所示，我们注册了 btn1 的 Click 事件（NO.1 处），然后在 btn1_Click 事件处理程序中实例化一个 Form 对象，并将其显示出来（NO.2 处）。注意这里 f 显示出来之后，并不需要消息循环，因为这个过程已经发生在消息循环内部了：GetMessage->DispatchMessage->WndProc->OnClick->激发 Click 事件->btn1_Click，已经有消息循环了之后，之后所有窗体都可以接受该消息循环的服务。

本章前面曾讲到过，通过 CreateWindow 创建窗体时，CreateWindow 会发送一系列非队列消息给新创建的窗体，这些消息不经过消息队列，而是直接发送给窗口过程（调用窗口过程），那么 Winform 中与之类似，我们在创建窗体并将其显示出来这一过程也会发送许许多多的非队列消息给窗体的窗口过程，也就是说，在下面代码 Code 8-21 中：

```
//Code 8-20
private void btn_Click(object sender,EventArgs e)
{
    Form f = new Form(); //NO.1
    f.Show(); //NO.2
    string b = “break”; //NO.3
}
```

NO.2 处代码到 NO.3 处代码中间，程序需要调用好几次 f 的窗口过程 WndProc，下面一张图显示了创建窗体前后，程序的运行流程：

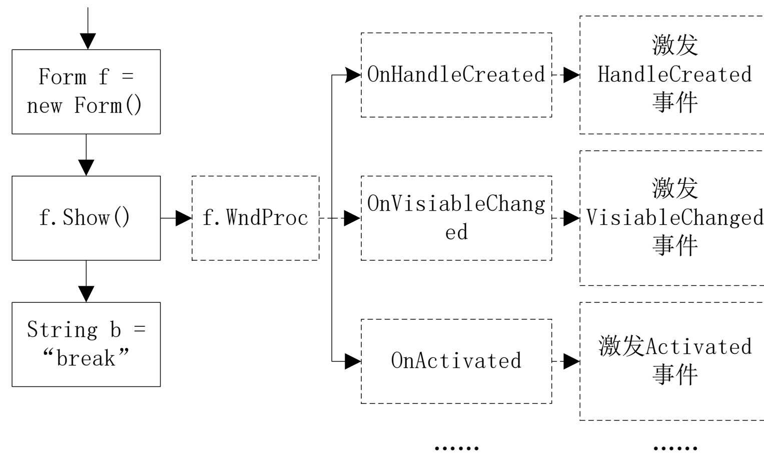


图 8-16 创建窗体的过程

如上图 8-16 所示，虚线框表示创建窗体并显示窗体这个过程中需要调用的窗口过程，图中只是列举了需要调用三次 f.WndProc，实际上远远不止这些。实质上，平时开发时，有很多看似简单的操作都会给窗体（控件）发送非队列消息，直接调用它的窗口过程，最后激发事件，比如修改 TextBox 的 Text 属性，会给 TextBox 发送一系列非队列消息，直接调用 TextBox 的窗口过程，最后激发类似 TextChanged 这样的事件，也就是说，一行修改 TextBox.Text 属性的简单代码内部，会有很多事情发生。

窗体中的其它控件，创建过程与窗体类似。到目前为止，我们讨论的窗体都是非模式对话框，模式对话框的创建与非模式对话框有一些区别，详见 8.6 节中的介绍。

8.5.4 窗口过程

前面提到过，Windows Forms 框架将窗体和窗口过程封装在了一起，Control（或其派生类，下同）类中的 WndProc 虚方法就是控件的窗口过程，之所以将窗口过程声明为虚方法，这是因为 Windows Forms 框架是面向对象的，充分地利用了面向对象编程中的“多态”特性，所有 Control 类的派生类均可以重写它的窗口过程，从而从源头上拦截到 Windows 消息，处理自己想要处理的 Windows 消息，Control 类中的 WndProc 虚方法类似如下：

```
//Code 8-21
public class Control
{
    //…
    public event EventHandler HandleCreated;    //NO.1
    public event EventHandler GotFocus;    //NO.2
    protected virtual void WndProc(ref Message m)
    {
        switch(m.Msg)    //NO.3
        {
            case 1:    //WM_CREATE
            {
                OnHandleCreated(new EventArgs());    //NO.4
                return;
            }
            //…
```

```
                case 7:    //WM_SETFOCUS
                {
                    OnGotFocus(new EventArgs());    //NO.5
                    return;
                }
                //...
                default:    //other windows messages
                {
                    this.DefWndProc(ref m);    //NO.6
                }
            }
        }
        protected virtual void OnHandleCreated(EventArgs e)    //NO.7
        {
            if(HandleCreated != null)
            {
                HandleCreated(this,e);
            }
        }
        protected virtual void OnGotFocus(EventArgs e)    //NO.8
        {
            if(GotFocus != null)
            {
                GotFocus(this,e);
            }
        }
    }
```

如上代码 Code 8-21 所示，Control 类一开始，定义了两个事件 HandleCreated 和 GotFocus（NO.1 和 NO.2 处），意思很明显，前者应该是当控件创建好了激发，后者是当控件获得焦点后激发。在 WndProc 虚方法一开始，就有我们尤其熟知的 switch/case 块（NO.3 处），没错，它跟传统 Win32 开发中几乎是一模一样的，唯一一个不同点就是：switch/case 块中并没有自己默认处理 Windows 消息，而是根据不同的消息去激发不同的事件（NO.4 和 NO.5 处），只要我们注册了 HandleCreated 和 GotFocus 事件，我们就能知道控件在什么时候创建完成、在什么时候获得了焦点。窗口过程做了一个非常重要的事：将 Window 消息转换成了.NET 中的事件，如下图 8-17 所示：

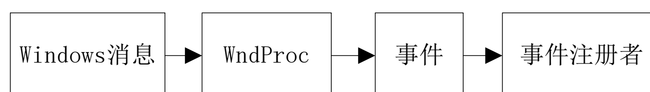


图 8-17 Windows 消息转换成事件

如上图 8-17 所示，Windows 消息经由窗口过程 WndProc 后，以事件的形式告知事件注册者。这很符合框架中的“好莱坞原则”——“do not call us,we will call you”，Windows Forms 框架就是通过这种方法，去调用开发者编写的事件处理程序。

Control 类中的 WndProc 虚方法只处理了一部分 Windows 消息（示例代码不全），其余的都交给了默认窗口过程 DefWndProc 去处理，如果 Control 类的派生类需要自己处理 Control 类中没被处理的 Windows 消息的话，那么派生类就需要重写 WndProc 虚方法，这正是面向对象

的好处之一，看下面代码：

```
//Code 8-22
public class MyControl:Control
{
    //…
    public event EventHandler Click;    //NO.1
    protected override void WndProc(ref Message m)
    {
        if(m.Msg == 0xf5)    // mouse click
        {
            OnClick(new EventArgs());    //NO.2
            return;
        }
        base.WndProc(ref m);    //NO.3
    }
    protected virtual void OnClick(EventArgs e)
    {
        if(Click != null)
        {
            Click(this,e);
        }
    }
}
```

如上代码 Code 8-22 所述，MyControl 类派生自 Control 类，由于 Control 类中没有处理鼠标点击的 Windows 消息，那么我们可以在 MyControl 类中重写窗口过程 WndProc，提前拦截 Windows 消息，判断 m.Msg 是否为 0xf5（消息对应整数值），如果是，激发 Click 事件（NO.2 处），注意，MyControl 类中其余没处理的消息需要调用基类 Control 类的 WndProc 虚方法处理（NO.3 处），否则会有很多 Windows 消息没被处理造成异常。Control 类及其派生类处理 Windows 消息的顺序如下图 8-18：

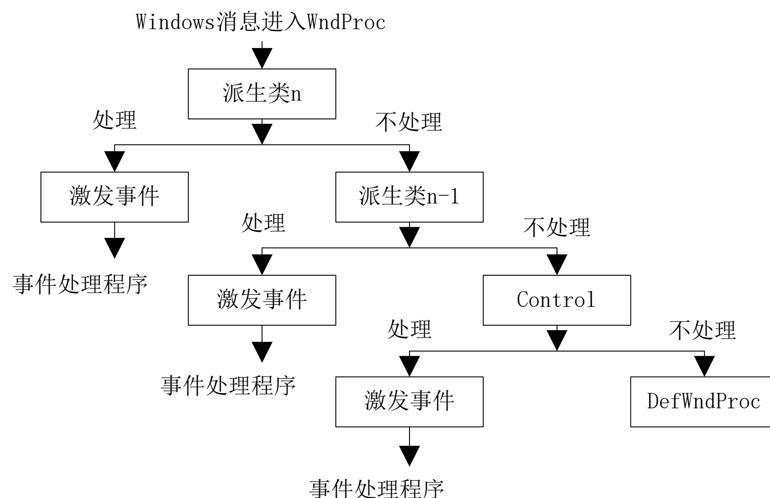


图 8-18 Windows 消息的处理顺序

如上图 8-18 所示，每个派生类不处理的消息，都需要交给上一层基类去处理，所以重写 WndProc 时，一定不要忘记调用基类的 WndProc（base.WndProc），这也是虚方法必须声明为 protected

的原因。

我们可以看到，窗口过程在激发事件的时候，并没有直接去激发，而是通过类似“On+事件名称”这样命名的虚方法去激发事件，具体原因请参见第五章关于事件的说明。

> 注：本小节中关于 CONTROL 类的源代码只是一种说明性代码，并不是 WINDOWS FORMS 中的源代码，它与官方代码还有一定差距，不过道理相似，可以类比。

8.6 模式窗体与非模式窗体

有关“模式”窗体和“非模式”窗体的概念想必每一个从事 Winform 应用程序开发的技术人员都有所了解，模式窗体显示之后，在它关闭之前，用户不能操作应用程序中的其它窗体，而非模式窗体则没有这样的限制，用户可以在非模式窗体之间来回切换，在这之前我们讨论的都是非模式窗体。模式窗体与非模式窗体在代码中的区别就是，使用 Form.ShowDialog()去显示一个模式窗体，使用 Form.Show()去显示一个非模式窗体：

```
//Code 8-23
class Form1:Form
{
    //…
    public Form1()
    {
        InitializeComponent();
        btn1.Click += new EventHandler(btn1_Click);
    }
    private void btn1_Click(object sender,EventArgs e)
    {
        Form f1 = new Form();
        f1.Show();      //NO.1
        string b = “break1”;    //NO.2

        Form f2 = new Form();
        f2.ShowDialog();    //NO.3
        b = “break2”;    //NO.4
    }
}
```

如上代码 Code 8-23 所示，点击 btn1 按钮，在 btn1_Click 事件处理程序中，我们先显示一个非模式窗体 f1（NO.1 处），f1 显示之后，NO.2 处代码马上就能执行，紧接着，我们显示一个模式窗体 f2（NO.3 处），f2 显示之后，NO.4 处代码却不能马上执行，只有等 f2 关闭后，NO.4 处代码才能继续执行，造成这个问题的原因就是前面提到过的：使用 Form.ShowDialog()显示一个模式窗体时，内部会创建一个嵌套消息循环，只有等内层消息循环退出之后，外层消息循环才能继续运行，见下图 8-19：

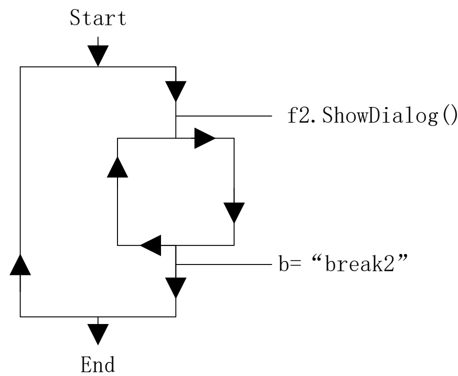


图 8-19 模式窗体中的消息循环

如上图 8-19 所示，在外层消息循环执行期间（比如执行 btn1_Click 事件处理程序时），如果我们调用 f2.ShowDialog()显示一个模式窗体，那么它会开启一个新的内层消息循环，前面讲到过，内层消息循环与外层消息循环共用同一个消息队列，能为当前线程中所有的控件提供服务。从图中可以看出，只有当内层循环退出之后，外层循环才能继续执行，那么，内层消息循环退出的条件是什么呢？还是当消息队列中有 WM_QUIT 消息时退出吗？显然不是，因为 WM_QUIT 消息是主消息循环（最外层循环）退出的条件，当一个线程的消息队列中有 WM_QUIT 消息时，代表这个 UI 线程将要结束。

我们来看一下 Form.ShowDialog()方法代码：

```
//Code 8-24
class Form
{
    //…
    public DialogResult ShowDialog()
    {
        Application.RunDialog(this);    //NO.1
        return this.DialogResult;    //NO.2
    }
}
public sealed class Application
{
    //…
    public static void RunDialog(Form f)
    {
        Application.ThreadContext.FromCurrent().RunModalMessageLoop(f); //NO.3
    }
    //…
    //nested type
    internal sealed class ThreadContext
    {
        //use native methods
        [DllImport("user32.dll", CharSet=CharSet.Unicode, ExactSpelling=true)]
        public static extern bool GetMessageW([In, Out] ref NativeMethods.MSG msg, HandleRef
hWnd, int uMsgFilterMin, int uMsgFilterMax);
        [DllImport("user32.dll", CharSet=CharSet.Auto, ExactSpelling=true)]
```

```
        public static extern bool TranslateMessage([In, Out] ref NativeMethods.MSG msg);
        [DllImport("user32.dll", CharSet=CharSet.Ansi, ExactSpelling=true)]
        public static extern IntPtr DispatchMessageA([In] ref NativeMethods.MSG msg);

        private static int totalMessageLoopCount; //NO.4
        private int messageLoopCount; //NO.5
        internal void RunModalMessageLoop(Form mainForm)
        {
            totalMessageLoopCount++; //NO.6
            messageLoopCount++; //NO.7
            if(mainForm != null)
                mainForm.Show(); //NO.8
            //disable other forms in this UI thread;    NO.9
            While(mainForm.DialogResult == DialogResult.None )    //NO.10
            {
                GetMessage();
                TranslateMessage();
                DispatchMessage();//dispatch message to Control.WndProc virtual method
            }
            if(mainForm != null)
                mainForm.Hide();
            //enable other forms in this UI thread    NO.11
            messageLoopCount--;    //NO.12
            totalMessageLoopCount--;    //NO.13
            if(messageLoopCount == 0)
            {
                Application.RaiseThreadExit();//NO.14 raise Application.ThreadExit event
            }
            if(totalMessageLoopCount == 0) //NO.15 raise Application.Exit event
            {
                Application.RaiseExit();
            }
        }
    }
}
```

如上代码Code 8-24所示，Form中的ShowDialog方法会调用Applicatin.RunDialog静态方法（NO.1处），Application.RunDialog 方法中通过 ThreadContext.FromCurrent()获取与当前 UI 线程关联的 ThreadContext 对象（NO.3 处），紧接着调用 ThreadContext 对象的 RunModalMessageLoop 方法开启一个消息循环，ThreadContext 类中的 RunModalMessageLoop 和前面讲到的 RunMessageLoop 结构类似，唯一不同的是前者中 while 消息循环退出条件改为了 DialogResult 是否为 DialogResult.None（不再是判断 Windows 消息是否为 WM_QUIT，NO.10 处），并且在循环开始，将当前线程中其它所有窗体禁用（NO.9 处，具体代码省略），循环结束后，将当前线程中其它所有窗体重新启用（NO.11 处，具体代码省略），这样一来，就是我们看到的情况：当模式窗体显示时，其它窗体不可用，当模式窗体的 DialogResult 属性为非 DialogResult.None 时，内层循环退出，模式窗体消失，其它不可用窗体继续可以使用。

现在我们应该能理解，为什么 Form.ShowDialog 方法不能立刻返回，就是因为它内部又开启了一个消息循环，内层消息循环退出条件是 Form.DialogResult 为非 DialogResult.None，所以当我们需要关闭模式窗体时，只需要将它的 DialogResult 属性设置为非 DialogResult.None，比如以下代码：

```
//Code 8-25
class Form1:Form
{
    //…
    public Form1()
    {
        InitializeComponent();
        btn1.Click += new EventHandler(btn1_Click);
    }
    private void btn1_Click(object sender,EventArgs e)
    {
        DialogResult = DialogResult.OK    //NO.1
        //or DialogResult = DialogResult.Cancel;
    }
}
//client code
Form1 f = new Form1();
f.ShowDialog();    //NO.2
```

如上代码 Code 8-25 所示，我们在 btn1_Click 事件处理程序中将 Form1 的 DialogResult 属性设置为 DialogResult.OK（非 DialogResult.None，NO.1 处），之后当我们调用 f.ShowDialog 显示一个模式窗体后（NO.2 处），点击 f 界面上的 btn1 按钮，模式窗体就会关闭，原因就是“非 DialogResult.None”是内层消息循环退出的条件。下图 8-20 显示调用 ShowDialog 方法产生的嵌套消息循环：

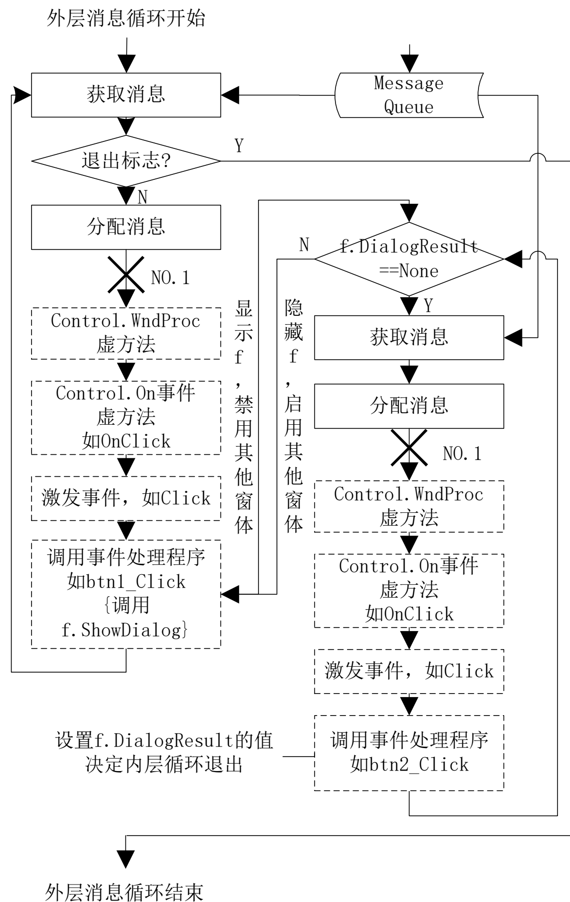


图 8-20 ShowDialog 产生的嵌套消息循环

如上图 8-20 所示，在外层循环中（btn1_Click 调用时），调用 f.ShowDialog()方法创建一个模式窗体，开启了一个内层循环，内层循环退出的条件由 f.DialogResult 控制。

通过将模式窗体的 DialogResult 属性值设置为非 DialogResult.None 之后，内层消息循环退出，模式窗体隐藏，但并没有 Close，因此之后我们还可以继续将该模式窗体显示出来，如果之后不再使用，我们一般需要将模式窗体放在 using 块中，如下代码：

```
//Code 8-26
using(Form f = new Form())
{
    f.ShowDialog();
}
```

如上代码 Code 8-26 所示，我们将创建模式窗体的代码放在 using 块中，当 f.ShowDialog()返回后，结束 using 块时，会自动调用 f 的 Dispose 方法去释放非托管资源（详见第四章非托管资源的释放）。

> 注：
>
> [1]本小节中关于 APPLICATION 类、THREADCONTEXT 类以及 FORM 类的源代码只是一种说明性代码，并不是 WINDOWS FORMS 中的源代码，它与官方代码还有一定差距，不过道理相似，可以类比。
>
> [2]由于 WINDOWS FORMS 框架的影响，原本比较清晰的代码运行流程变得不再明

了，方法与方法之间的调用关系尤其复杂，不过如果只考虑对框架的应用，WINDOWS FORMS 却恰恰隐藏了我们不需要知道的东西，大大简化了程序的开发工作流程，本章大部分配图中的虚线框代表 WINDOWS FORMS 框架对外公开的接口，也就是开发人员需要负责的部分。

8.7 本章回顾

本章开始介绍了传统 Win32 应用程序的“从生到死”，紧接着介绍了.NET 中的 Windows Forms 框架，目的是想让读者能从两者之间找到共同点，我们发现，无论是传统的 Win32 应用程序还是.NET 平台中的 Winform 应用程序，均包含“Windows 消息”、“消息循环结构”以及“窗口过程”等组成部分。在讲 Windows Forms 框架部分时，首先介绍了.NET 中 Windows Forms 框架包含的主要类型，然后分别详细介绍了框架内部隐藏的“消息循环泵”以及“窗口过程”等桌面应用程序必备结构，本章最后还介绍了我们常见的“模式窗体”和“非模式窗体”的实现原理。本章中出现的“消息泵”在第十章中有详细说明，“泵”是框架的必备结构。

8.8 本章思考

1.一个 Windows 桌面应用程序主要包括哪几个部分？

A：存放数据的消息队列、为程序提供动力的消息循环以及处理数据的窗口过程。

2.GUI 界面能够持续响应用户输入并能维持界面显示的根本原因是什么？

A：消息泵的存在，它能够为程序持续运转提供动力。

3.“嵌套消息循环”指的是什么？

A：在一次消息循环过程中，再次开启一个消息循环，内部循环退出之前，外部循环一直处于等待状态，但是由于两个循环均负责处理同一个消息队列中的消息，因此不会出现 Windows 消息不能及时被处理的情况。

第九章 沟通无碍：网络编程

没有网络，主机跟主机之间就没有资源共享，每台主机就像一座孤岛，与外世隔绝。.NET类库中包含有丰富的有关网络编程的类型，如 System.Net 命名空间（包括子命名空间）中的所有类型。本章主要介绍了两种 Socket 网络编程方式：TCP 编程以及 UDP 编程，并说明了这两种网络编程的区别以及各自适用的场合。

9.1 两种 Socket 通信方式

Socket 通信编程主要分为两类，一类为基于连接的 TCP 通信，一类为基于无连接的 UDP 通信，两者各有优缺点，适合不同场合。

9.1.1 IP 与端口

理论上讲，IP 是世界互联网中某台主机的唯一标示，我们可以通过指定 IP 访问到互联网中任何一台主机。IP 只能帮我们找到主机，那么怎样才能找到具体通信的某个应用程序呢？这时候就需要引入另外一个概念：端口（Port），端口的作用就是在找到远程通信主机的基础上，再确定远程通信应用程序。

在网络编程中，IP 和端口几乎都是同时出现的，IP 确定哪栋楼，而端口确定该进哪扇门。下图 9-1 显示 IP 和端口在通信中的作用：

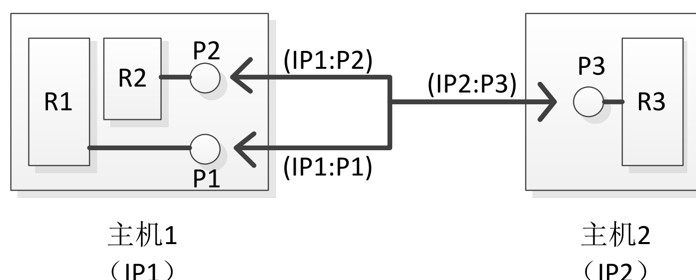


图 9-1 IP 与端口在通信中的作用

如上图 9-1 所示，R1、R2 以及 R3 分别代表运行在两个不同主机上的应用程序，本地程序以“目标 IP：目标端口”为地址去访问远程程序，端口就是传输网络数据的“进出口”。理论上，一台主机可以分配 2 的 16 次方个端口供程序使用。

> 注：通信程序与端口不是一对一的关系，也就是说，一个程序可以绑定（占有）多个端口，多个端口同时与外界进行数据交换。

理想状态下，通过“IP：端口”就能找到互联网中任意一台主机，但是现实往往跟理想相差甚远。随着世界计算机数量的增多，如果每台主机占用一个独立 IP，那么可分配的 IP 迟早要被消耗殆尽（IP4 由 4 个字节表示），为了解救 IP 资源稀缺的现状，人们发明了一种以网络

地址转换（NAT，Network Address Transfer）的形式去共享一个 IP 地址，也就是说，可以将一小范围的主机先组成一个子网，该子网内的主机有自己的一套 IP 规则，称为私有 IP，整个子网与外界通信时，都使用同一个 IP，称为公有 IP，子网主机与外界通信时，NAT 会自动将主机的私有 IP 转换成公有 IP，外界只能看到子网中所有主机共用的一个公有 IP。私有 IP 与公有 IP 可以重复，因为它们存在于不同的网络范围中。NAT 的出现缓解了 IP 资源稀缺危机，但是给网络编程带来了困扰，因为互联网从此变得不再“直截了当”，我们不再能够轻易地去使用一个公有 IP 访问到某个子网中的某台主机，网络寻址从此不再简单。

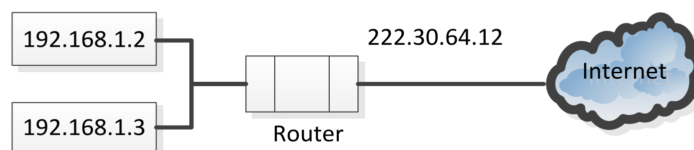


图 9-2 网络地址转换

如上图 9-2 所示，子网中两台主机私有 IP 分别为“192.168.1.2”和“192.168.1.3”，通过一个路由器组成一个子网，路由器对外的公有 IP 为“222.30.64.12”，子网中每一台主机与外界通信时，对外界来讲，源 IP 均为“222.30.64.12”。正因为图中 Router 的隔绝，外界几乎从来都不能主动与子网某台主机取得联系，NAT 的出现将通信过程的建立变得复杂，因为通信双方中间新增加了障碍，需要一些穿透技术，有关“网络穿透”的内容本章暂不介绍，本章介绍的网络编程均以“两台主机均在同一子网中”为前提。

9.1.2 基于连接的 TCP 通信

所谓“基于连接”，就是指通信双方在进行数据交换之前，先要建立连接，连接建立后，通信双方之间相当于有一条隧道，数据按顺序在该隧道中传输，数据传输完毕后，双方可以选择关闭隧道，连接结束。TCP 通信犹如打电话，只有双方均就位，通话才能正常进行，而且在通话期间，任何一方退出，均会导致通话终止。TCP 通信具有以下特点：

1）**可靠性**；

通信双方均就位，一方发送数据，另一方收到后会做出回应，如果超时未发送成功，会自动重发，数据不会丢失。

2）**顺序性**；

既然数据是按顺序走在建立的一条隧道中，那么数据遵循“先走先达到”的规则，并且隧道中的数据以“流”的形式传输，发送方发送的前后两次数据之间没有边界，需要接收方自己根据事先规定好的“协议”去判断数据边界。

3）**高损耗。**

“高损耗”包括机器性能损耗高、宽带流量损耗高。因为通信双方时刻需要维持着连接的存在，这必然会损耗通信双方主机性能，要想维持隧道的通畅，通信双方必须不断地发送检测包和应答包，同时，它还支持数据重发等数据纠错功能，这些都将导致网络流量的增加。

TCP 通信编程中，“请求方”主动连接“被请求方”，该过程称为“连接”（Connect），被请求方收到“连接”的请求后，接受（Accept）并创建一个 Socket 代理与请求方进行数据交互，该过程如下图 9-3：

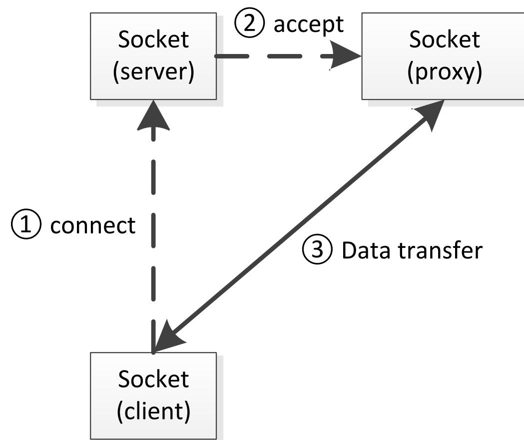


图 9-3 TCP 通信过程

如上图 9-3 所示，通信双方中主动请求方称为“客户端（Client)”，被请求方称为“服务端（Server)”，图中经过 connect、accept 之后，服务端与客户端之间的数据交换发生在客户端 Socket 和服务端代理 Socket 之间。

由于 TCP 通信中，数据以“流”的形式传输，数据之间没有边界，如果数据发送方发送两次数据，一次为：“0x01 0x02 0x03 0x04”，下一次为：“0x05 0x06 0x07 0x08”，那么数据接收方接收到这两次数据的形式是不确定的，可能一次全部接收，也可能第一次接收“0x01”，第二次接收“0x02 0x03 0x04 0x05 0x06 0x07 0x08”。

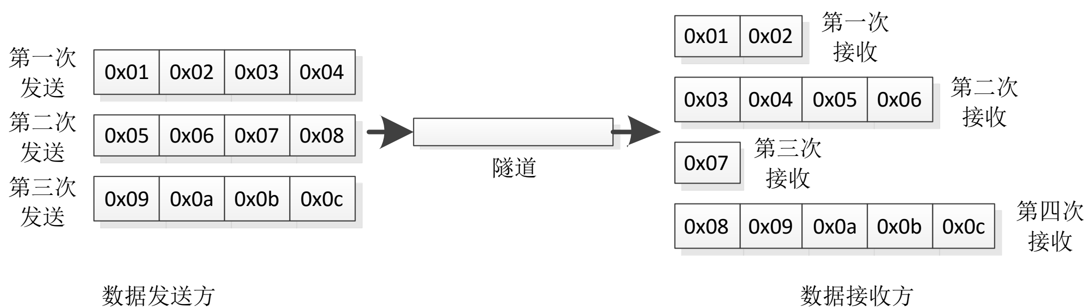


图 9-4 TCP 通信传输数据无边界

如上图 9-4 所示，数据发送方前后共发送三次数据，每次发送 4 个字节，数据接收方可能分四次才接收完毕所有的数据，并且每次接收数据字节数不确定，正因为数据接收方无法自动判断每次发送数据的边界，这就需要我们在编程中使用某些技术进行人工判断（详见 9.2 和 9.3 节）。

> 注：我们可以从图 9-4 中看出，虽然接收到的数据无法确认边界，但是将每次接收到的数据拼接在一起，它们的字节顺序跟发送方发送数据的顺序一致，这就是 TCP 通信中数据的“顺序性”。

TCP 通信模式适合一般对数据准确度要求比较高的场合，在这些场合中，数据的一次出错、丢失均能够对通信双发造成很大的负面影响。如应用层的 Http、FTP 等协议均是基于 TCP 通信的，确保数据能够可靠、正确传输。

9.1.3 基于无连接的 UDP 通信

与 TCP 通信不同的是，UDP 通信之前不需要建立连接。如果将 TCP 通信比如成打电话，那么 UDP 通信就是发送短消息，发送短消息前，不需要去关心对方是否就位，甚至都不用去

管对方手机是否已开机，它仅仅是单方面的一个操作。UDP 通信具有以下特点：

1）**不可靠性；**

既然无连接，发送方只管发送数据，而不管对方是否能够正确地接收到数据，更不负责数据超时重发等功能。

2）**无序性；**

数据以“数据报”的形式发送，可以把“数据报”看成是一个“包”。如果把 TCP 传输数据比如成“河里的流水”，那么 UDP 传输数据就是‘邮局寄信’。发送方先发送的数据可能后到达，后发送的数据可能先到达，这个跟短消息类似。

3）**低损耗。**

“低损耗”包括机器性能损耗低、宽带流量损耗低。UDP 通信不需要维持一个连接的存在，所以它不需要消耗额外的机器性能。同时它也没有像 TCP 通信那样为了保持隧道的通畅，而必须不停地发送检测包和应答包，更不会进行一些数据检测纠错、重发等行为。

UDP 通信编程中，没有 TCP 通信中所谓的“服务端”，只存在“客户端”，每个客户端之间是平等的，发送数据之前不需要进行“连接”请求。

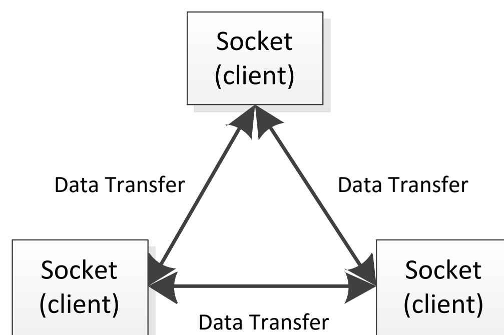


图 9-5 UDP 通信过程

如上图 9-5 所示，客户端跟客户端发送数据时，直接进行。

由于 UDP 通信中，数据以“数据报”的形式传输，数据以一个整体发送、以一个整体接收，存在数据边界。但是接收到的数据是无序的，先发送的可能后接收，后发送的可能先接收，甚至有的接收不到，见下图 9-6：

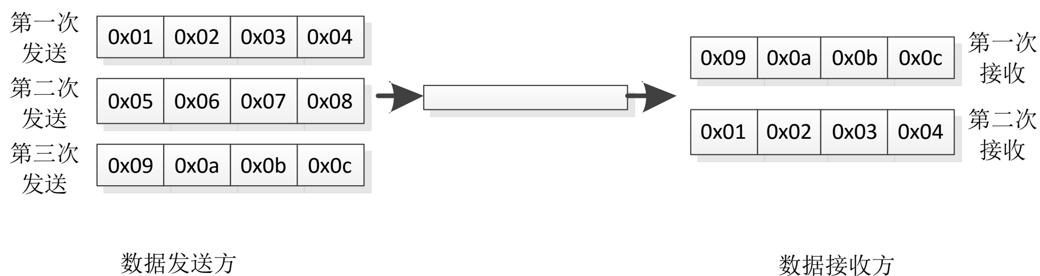


图 9-6 UDP 通信传输数据无序性

如上图 9-6 所示，数据发送方依次发送了三次数据，数据接收方只接收到了第一次和第三次发送的数据，而且先收到第三次发送的数据，而后收到第一次发送的数据，一次接收到的数据长度等于数据发送时的长度，第二次发送的数据丢失。

> 注：UDP 通信中虽然数据报的传输是无序的，但是对于每一次发送的数据而言，接收方接收到的数据顺序跟发送时的顺序相同。

UDP 通信模式一般适合单次数据传输量比较小、但是需要高频率传输，并且对数据准确度要求不高的场合，在这些场合中，数据偶尔异常、丢失对通信双方影响不大，如实时监控、在线视频等需要不断的接收数据，后一次接收到的数据马上就可以覆盖前一次接收到的数据，偶尔数据异常并不会对程序功能产生很大影响。

9.1.4 应用层协议

TCP 通信基于 TCP 协议，UDP 通信基于 UDP 协议，两种协议均属于 OSI（Open System Interconnection）七层网络结构模型中的“传输层”协议（见第二章 2.13 节），而我们开发的通信应用程序运行在七层网络模型中的顶层：应用层，正常情况下，应用层以下的协议对于通信应用程序来讲都是透明的，不需要程序关心其具体实现，通信应用程序只需要关心应用层的应用层协议，比如常见的 HTTP 协议、FTP 协议、Telnet 协议等等。

> 注：本章未提及到 TCP 协议或 UDP 协议格式，笔者认为本章重点是让读者掌握两种 SOCKET 通信方式以及应用层协议在通信过程中起到的作用。

应用层协议也是一种数据结构，数据发送方应用程序需要按照该数据结构规定的格式发送数据，对应的，数据接收方应用程序需要按照该数据结构规定的格式去解析接收到的数据，只有双方通信程序均遵守该协议，通信才能正常进行，下图 9-7 为浏览器某一次 Http 请求后服务器返回数据其中的响应头部分：

```
▼ Response Headers      view parsed
   HTTP/1.1 200 OK
   Cache-Control: private, max-age=10
   Content-Type: text/html; charset=utf-8
   Content-Encoding: gzip
   Expires: Wed, 18 Jun 2014 13:28:32 GMT
   Last-Modified: Wed, 18 Jun 2014 13:28:22 GMT
   Vary: Accept-Encoding
   Server:
   Date: Wed, 18 Jun 2014 13:28:21 GMT
   Content-Length: 18657
```

图 9-7 HTTP 请求响应头

如上图 9-7 所示，我们可以从 HTTP 响应头中按照规定格式读取到 Http 协议版本信息、请求结果、返回内容格式以及内容编码等信息。HTTP 协议是一种应用层的规范，通信双方程序如 Web 服务器、浏览器均要遵守该规范，并按照该规范去发送/解析数据。

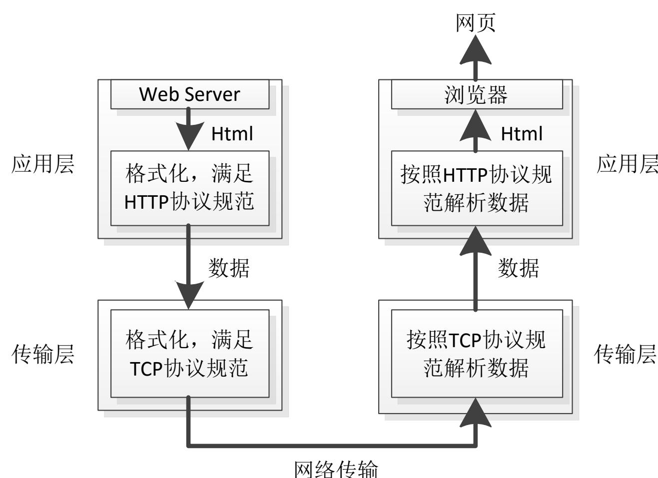


图 9-8 HTTP 请求响应过程

如上图 9-8 所示，Web Server 和浏览器都运行在应用层，都应该遵守应用层中的 Http 协议，原始数据（如 Html 文档）经由应用层程序 Web Server 处理，使其符合 Http 协议规范，然后将符合 Http 协议规范的数据传递给传输层（中间省略其它层），在传输层中数据又被进行一次加工（该步骤对于应用层来说，是透明的），使上一层传递过来的数据符合 TCP 协议规范，接着将符合 TCP 协议规范的数据传递到下一层，最终经由互联网发送至浏览器所在主机，相反地，接收数据的主机依次将数据按照“从下往上”的顺序从物理层传递到传输层，在传输层中，按照 TCP 协议规范解析数据，将解析后的数据依次往上传递，最后至应用层，应用层中的浏览器程序需要对数据按照 Http 协议规范进行解析，读取数据中 HTTP 协议版本信息、内容编码、内容正文等，最后将解析出来的内容正文（如 Html 文档）显示在界面。

注：OSI（Open System Interconnection）网络结构模型将网络分为七个层，从上到下分别为：应用层、表示层、会话层、传输层、网络层、数据链路层以及物理层，详见第二章 2.13 节。

应用层协议跟具体的应用程序有关，它的出现是为了解决某一个（类）具体问题，比如 Http 协议是为了规范 Web 数据传输方式，FTP 协议是为了规范文件数据传输方式，而 Telnet 协议是为了规范远程登录数据传输方式，Http 协议对于文件传输来讲是无效的，同理，Telnet 协议对 Web 数据传输来讲也是无效的。通信双方的应用程序都必须遵守同一个应用层协议，如果我们要开发一款自己的浏览器，那么我们必须遵守 Http 协议，说得直白一点，我们只能按照 Http 协议规定的格式去解析 Web 服务器返回的数据，同时我们也只能按照 Http 协议规定的格式去发送数据给 Web 服务器，否则 Web 服务器无法识别我们上传的数据是什么。

我们前面提到的应用层协议比如 Http、FTP 等都是现在广泛流行的应用层协议，如果我们自己开发一个网络通信程序，完全可以自己定义一套应用层协议，通信双方应用程序必须严格按照该协议来传输数据。为自己的通信应用程序定义自己的应用层协议有以下好处：

1）有利于数据处理；

无论是数据发送方程序还是数据接收方程序，只需要按照协议文档给出的规范去发送数据、解析数据即可，对于 TCP 通信来讲，应用层协议还有助于判断数据边界，详见后面有关“TCP 通信实现”。

2）方便扩展；

对于一个大型的网络通信系统，有了应用层协议的规范，可以更容易地动态增加通信终端模块，只要终端模块收发数据时遵循了协议规范即可。如一个雷达显示监控系统，只要按照协议规范解析服务器传来的雷达数据，可以很方便地扩展出新的雷达显示终端。

3）便于后期程序维护。

数据传输的格式化有利于后期程序的维护，若有新的数据传输需求，只需要更新协议，而更新协议的数据结构对通信双方程序来讲，需要改动的地方不多，也不会复杂。

9.1.5 .NET 中 Socket 编程相关类型

.NET 中为 Socket 两种通信方式提供了非常丰富的类型支持，这些类型主要集中在 System.Net.Sockets 命名空间下，表 9-1 列出了.NET 中 Socket 通信编程需要用到的最重要的几个类型：

表 9-1 Socket 通信编程主要类型

序号	类型	说明

1	System.NET.Sockets.Socket	最重要的类型，windows 中 socket 的托管实现，TCP、UDP 通信实现均基于该类型。
2	System.Net.Sockets.TcpListener	TCP 通信中，用来侦听客户端连接，它是对 Socket 类型的一个定向（TCP）封装，为 TCP 通信提供更简洁更方便的公开接口。
3	System.Net.Sockets.TcpClient	TCP 通信中，用来负责连接、收发数据，它是对 Socket 类型的一个定向（TCP）封装，为 TCP 通信提供更简洁更方便的公开接口。
4	System.Net.Sockets.UdpClient	UDP 通信中，用来负责收发数据，它是对 Socket 类型的一个定向（UDP）封装，为 UDP 通信提供更简洁更方便的公开接口。
5	System.Net.Sockets.NetworkStream	提供网络通信的基础数据流，可以使用该类型对象进行网络数据的读写，类似读写本地文件一样。

如上表 9-1 所示，.NET 中有关 Socket 通信编程的类型主要包含 5 种，其中 Socket 类型负责实现 TCP、UDP 通信中的连接、数据的收发等操作，TcpListener 和 TcpClient 均是对 Socket 类型的一个封装，专门用于 TCP 通信，因此内聚性比 Socket 类型更高，同样的，UdpClient 也是对 Socket 类型的一个封装，专门用于 UDP 通信，内聚性也比 Socket 类型更高。

TcpListener 和 TcpClient：

我们从表中可以看到，TCP 通信中有两个重要类型：TcpListener 和 TcpClient，而 UDP 通信中却只有一个 UdpClient 类型，原因很简单，前面讲到过，TCP 通信中一般分两部分，一个服务端，一个客户端，而 UDP 通信中没有服务端的概念，每个客户端都是平等的。TcpListener 和 TcpClient 的关系见下图 9-9（参见图 9-3）：

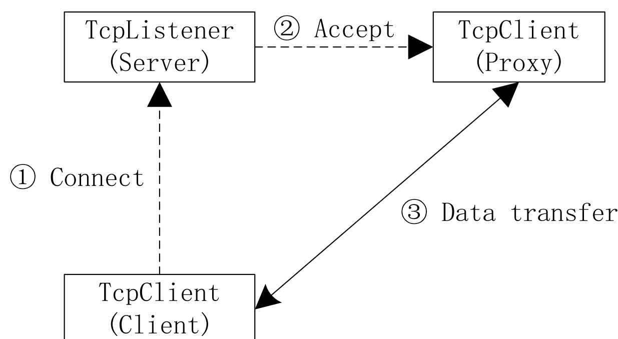


图 9-9 TcpListener 与 TcpClient 的关系

如上图 9-9 所示，TcpListener 侦听来自客户端的“连接”请求，返回一个代理 TcpClient，该代理与客户端的 TcpClient 进行数据交换。

UdpClient：

UdpClient 在 UDP 通信中所处的角色如下图 9-10（参见图 9-5）：

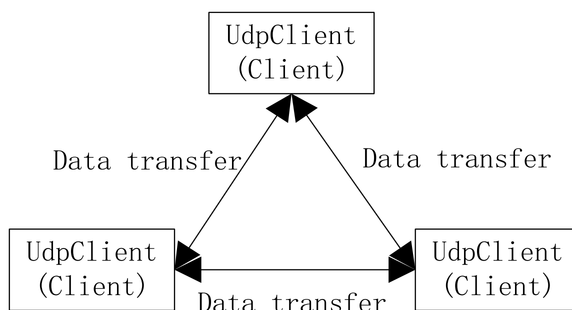


图 9-10 UdpClient 在通信中的角色

如上图 9-10 所示，在 UDP 通信中，每个 UdpClient 代表一个终端，每个终端都是平等的，任何终端不需要进行“连接”请求就可以直接发送数据（不保证能被接受）。

NetworkStream：

NetworkStream 类型是 System.IO.Stream 类的一个派生类，前面讲到过，TCP 通信中数据是以“流”的形式进行传输，NetworkStream 就是专门负责收发（读写）TCP 通信时的网络数据，可以使用 TCPClient.GetStream()方法返回一个 NetworkStream 对象，使用该对象进行网络数据的读写。

> 注：NETWORKSTREAM 只能用于 TCP 通信中。

Socket：

Socket 类型是一个相对较基础的通信类型，它既能实现 TCP 通信也能实现 UDP 通信，可以认为 TcpListener、TcpClient 以及 UdpClient 进一步封装了 Socket 类型。本章后面的实例均以 Socket 类型作为说明对象。

9.2 TCP 通信实现

本节以一个简单的“字符串变换”为例，来说明一个 TCP 通信程序的整个结构，现假设客户端请求服务端，并向服务端发送一个字符串和一个“变换指令”，要求服务端按照“变换指令”去变换该字符串，然后再将变换后的字符串返回给客户端。服务端与客户端的关系为一对多，即同时可以有多个客户端请求服务端。

前面讲到过，服务端需要先侦听客户端的“连接”请求，然后生成一个代理 Socket，让该代理与客户端通信，那么服务端的结构见下图 9-11：

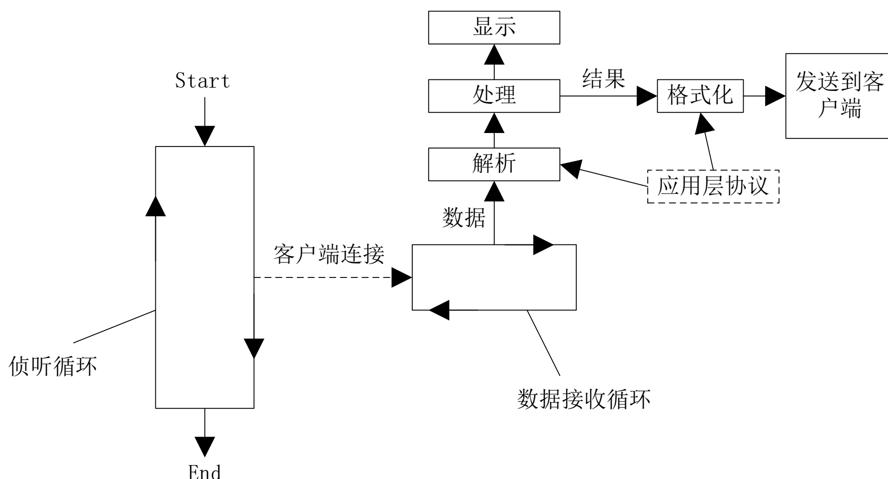


图 9-11 TCP 通信服务端结构

如上图 9-11 所示，由于服务端可以连入多个客户端，所以它需要有一个“侦听泵”，该泵不断地接收客户端的连接请求，并为每个连接创建独立的“数据接收泵”，该泵不断地接收客户端发送的数据，当有数据到达时，需要按照事先定义的“应用层协议”去解析数据，解析完成后才能处理数据（字符串转换），数据处理完毕后，将结果显示到界面，并再将结果按照“应用层协议”格式化，发送给客户端。我们很容易发现，服务端与所有的客户端是同时进行交互的（虽然图 9-11 中只画了一个客户端连接），因此程序中必然需要使用到多线程。

注：可以将"泵"理解为"循环"，这是一个很形象的比喻，现实生活中的泵为我们提供动力，使某个过程能周而复始地进行，本书后面有单独的章节讲述代码中的"泵"结构。

对于客户端来讲，不需要侦听的过程，只需要连接到服务端即可开始数据的接收，详见下图 9-12：

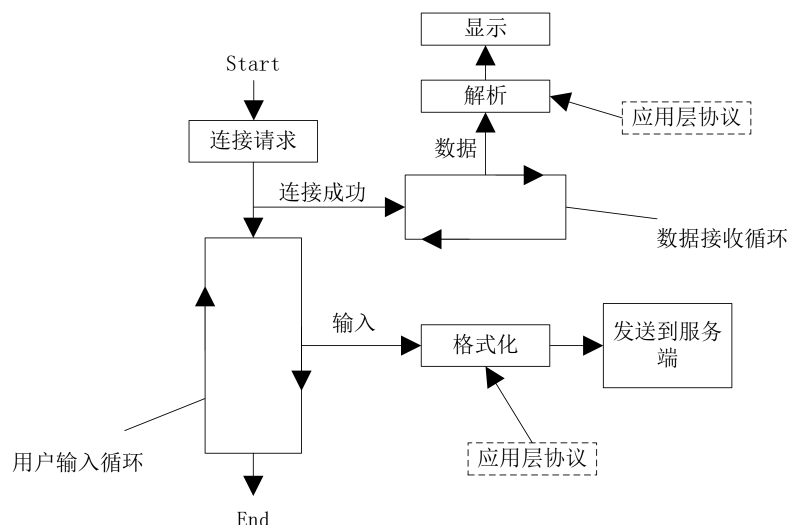


图 9-12 TCP 通信客户端结构

如上图 9-12 所示，客户端向服务端发送"连接"请求后，如果成功，就创建一个"数据接收泵"，该泵不断地接收来自服务端的数据，当有数据到达时，需要先按照事先定义的"应用层协议"去解析数据，解析完成后才能将结果显示出来。客户端还需要一个"用户输入泵"，该泵不断地接收用户输入（包括需要转换的字符串和转换指令），然后将用户的输入按照事先定义好的"应用层协议"格式化，最后发送到服务端。我们很容易发现，接收网络数据和接收用户输入两个过程同时进行，因此程序必须使用到多线程。

不管是服务端收发数据还是客户端收发数据，均应该遵守事先定义好的应用层协议，只有这样，双方通信才能顺利进行。

注：通讯程序是以控制台应用程序的方式实现，因此需要定义一个"数据接收循环（泵）"来循环接受用户输入。

9.2.1 定义应用层协议

客户端发送的字符串"转换指令"有多种，见下表 9-2：

表 9-2 字符串转换指令

序号	指令值（byte）	说明
1	0x01	将字符串中小写字符转换成大写
2	0x02	将字符串中大写字符转换成小写
3	0x03	去掉字符串中的百分号（%）字符
4	0x04	将字符串中的百分号（%）替换为空格

如上表 9-2 所示，假设一共有四种字符串转换请求，那么我们可以按下面图 9-13 设计应用层协议的数据结构：

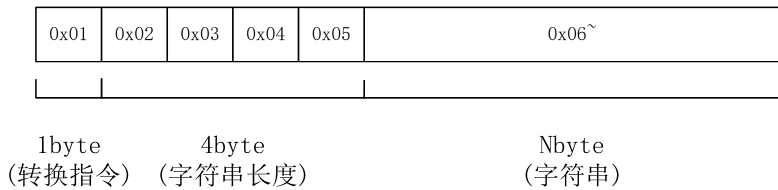


图 9-13 应用层协议数据结构

如上图 9-13 所示，开头一个字节代表字符串转换类型，后续四个字节存放一个 Int32 的整型数据，表示字符串的长度（字符串采用 Unicode 编码），最后 N 个字节表示字符串内容。数据发送方必须按照此协议格式发送数据，数据接收方必须按照此协议格式接收数据。

9.2.2 编写服务端

服务端主要包含五个重要部分：数据发送、数据解析、数据处理（转换字符串）、数据接收泵（循环）和侦听泵（循环）。

数据发送：

给定转换指令和转换之后的字符串，将其发送到对应客户端：

```
//Code 9-1
static void Send(Socket socket, int msg_type, string msg)
{
    byte[] msg_buffer = Encoding.Unicode.GetBytes(msg);    //to bytes      //NO.1
    byte[] bytes_to_send = new byte[5 + msg_buffer.Count()];    //1 + 4 + N        NO.2
    using (BinaryWriter bw = new BinaryWriter(new MemoryStream(bytes_to_send)))
    {
        bw.Write((byte)msg_type);              //NO.3
        bw.Write(msg_buffer.Length);           //NO.4
        bw.Write(msg_buffer);                  //NO.5
    }
    socket.Send(bytes_to_send);    //send to remote
}
```

如上代码 Code 9-1 所示，方法中首先将字符串转变成字节数组（NO.1 处），然后创建一个发送缓冲区（NO.2 处），该缓冲区的长度等于应用层协议数据结构的长度（1+4+N），紧接着依次将转换类型（msg_type）、字符串数组长度（一个 Int32 整型数据，字符串使用 Unicode 编码）、字符串内容写入缓冲区（NO.3、NO.4 以及 NO.5 处），最后将整个缓冲区数据发送给远程客户端。

代码 Code 9-1 中发送数据时，遵循了应用层协议。

数据解析：

当服务端接收到数据时，先将接收到的数据写入一个字节容器，然后再按照应用层协议，从该容器中解析出一条完整的数据：

```
//Code 9-2
static bool ResolveInfos(ref int msg_type, ref string msg, List<byte> bytes_container)
{
    if (bytes_container.Count > 5)    //NO.1
    {
```

```
            using (BinaryReader br = new BinaryReader(new
MemoryStream(bytes_container.ToArray())))
            {
                byte head = br.ReadByte();
                int length = br.ReadInt32();
                if (bytes_container.Count - 5 >= length)    //NO.2
                {
                    msg = Encoding.Unicode.GetString(br.ReadBytes(length));    //NO.3
                    msg_type = (int)head;    //NO.4
                    bytes_container.RemoveRange(0, 5 + length);    //NO.5
                    return true;
                }
                else     //NO.6
                {
                    msg_type = 0;
                    msg = null;
                    return false;
                }
            }
        }
        else    //NO.7
        {
            msg_type = 0;
            msg = null;
            return false;
        }
}
```

如上代码 Code 9-2 所示，方法开始先判断存储已接收数据的字节容器 bytes_container 中的数据长度是否大于等于 5（一条完整数据至少包含 5 个字节，1+4），若是，则开始解析（NO.1 处）；若不是，则说明字节容器中数据不足（NO.7 处）。如果字节容器中的剩余数据长度大于等于指定数据长度（NO.2 处，协议中第 2 到第 5 个字节表示的一个 Int32），则继续解析；否则，字节容器中数据不足（NO.6 处）。最后，如果字节容器中的数据足够一条完整的数据，那么可以依次从中解析出“字符串转换类型”、“字符串长度”（Unicode 编码）以及字符串内容（NO.3 处）。紧接着将已解析出来的数据从容器中移除（NO.5 处）。

代码 Code 9-2 中解析数据时，遵循了应用层协议。

数据处理（字符串转换操作）：

当接收到的数据经过解析完毕后，就需要进行处理（字符串转换）：

```
//Code 9-3
static void DealAsk(Socket proxy_socket, int msg_type, string msg)
{
    string msg_dealed = msg;
    switch (msg_type)
    {
        case 1:    //NO.1
```

```
            {
                msg_dealed = msg_dealed.ToUpper();
                break;
            }
        case 2:   //NO.2
            {
                msg_dealed = msg_dealed.ToLower();
                break;
            }
        case 3:   //NO.3
            {
                msg_dealed = msg_dealed.Replace("%","");
                break;
            }
        case 4:   //NO.4
            {
                msg_dealed = msg_dealed.Replace("%"," ");
                break;
            }
        default:
            {
                break;
            }
    }
    Send(proxy_socket, msg_type, msg_dealed);   //NO.5
}
```

如上代码 Code 9-3 所示，数据处理很简单，按照 msg_type 的值，进行对应的字符串转换（NO.1、NO.2、NO.3 以及 NO.4 处），然后将转换之后的字符串发送给远程客户端（NO.5 处）。

数据接收泵：

每个客户端都有一个数据接收泵与之对应，下面代码显示了数据接收的线程方法：

```
//Code 9-4
static void Receive_th(object obj)     //NO.1
{
    Socket proxy_socket = obj as Socket;
    string remote_ip = (proxy_socket.RemoteEndPoint as IPEndPoint).Address.ToString(); //NO.2
    int remote_port = (proxy_socket.RemoteEndPoint as IPEndPoint).Port;   //NO.3

    List<byte> bytes_already_recv = new List<byte>();

    byte[] bytes_to_recv = new byte[256];
    int length_to_recv = 0;
    do   //NO.4
    {
        try
```

```
        {
            length_to_recv = proxy_socket.Receive(bytes_to_recv);
            byte[] tmp = new byte[length_to_recv];
            Buffer.BlockCopy(bytes_to_recv, 0, tmp, 0, length_to_recv);
            bytes_already_recv.AddRange(tmp);

            int msg_type = 0;
            string msg = null;
            while (ResolveInfos(ref msg_type, ref msg, bytes_already_recv))    //NO.5
            {
                Console.ForegroundColor = ConsoleColor.Green;
                Console.WriteLine("[" + DateTime.Now.ToLongTimeString() + "] 来自(" +
remote_ip + ":" + remote_port + ")的请求");
                Console.ForegroundColor = ConsoleColor.White;
                Console.WriteLine("                请求类型:" + GetMsgType(msg_type));
                Console.WriteLine("                数据内容:" + msg);
                Console.WriteLine();
                DealAsk(proxy_socket, msg_type, msg);     //deal data    NO.6
            }
        }
        catch
        {
            proxy_socket.Close();
        }
    }
    while (length_to_recv != 0);
}
```

如上代码 Code 9-4 所示，Receive_th 方法是一个线程调用方法（通过 Thread 创建），NO.2 和 NO.3 处获取远程客户端的 IP 和端口号，NO.4 处开始一个“数据接收泵”，该泵不断地接收来自客户端的数据，每次有数据到达时，先将接收到的数据写入一个字节容器（bytes_already_recv）中，然后开始解析数据（NO.5 处），当解析成功时，将其显示在屏幕，并做出处理（转换操作，NO.6 处）。

侦听泵：

侦听泵的实现放在主线程中（Main 方法中）：

```
//Code 9-5
static void Main(string[] args)
{
    Console.Title = "tcp    server";
    int local_port = 5600;
    Socket server = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
    server.Bind(new IPEndPoint(IPAddress.Any, local_port));    //NO.1
    server.Listen(10);    //No.2
    Console.WriteLine("Server Start Listening...(Port:" + local_port + ")");
```

```
        while (true)     //accept loop     NO.3
        {
            Socket proxy_socket = server.Accept();       //NO.4
            new Thread(new ParameterizedThreadStart(Receive_th)).Start(proxy_socket); //NO.5
            string remote_ip = (proxy_socket.RemoteEndPoint as IPEndPoint).Address.ToString();
            int remote_port = (proxy_socket.RemoteEndPoint as IPEndPoint).Port;
            Console.ForegroundColor = ConsoleColor.Green;
            Console.WriteLine("[" + DateTime.Now.ToLongTimeString() + "] 远程主机(" + remote_ip +
":" + remote_port + ")连入");    //NO.6
            Console.ForegroundColor = ConsoleColor.White;
            Console.WriteLine();
        }
}
```

如上代码 Code 9-5 所示，Main 方法中，NO.1 和 NO.2 处分别绑定侦听端口、开始侦听，NO.3 处开始一个“侦听泵”，不断的侦听来自客户端的“连接”请求，当有请求到达时，创建一个代理 Socket（NO.4 处），并马上创建一个线程（NO.5 处），该线程负责创建“数据接收泵”（见前面），紧接着，将用户连接信息显示在屏幕（NO.6 处），一次侦听结束，开始下一次侦听。

> **注：由于代码调用关系，笔者颠倒了介绍顺序，按照CODE 9-5->CODE 9-4->CODE 9-3->CODE 9-2->CODE 9-1 的顺序阅读代码，更易理解。**

客户端连入服务端之后通信结构如下图 9-14：

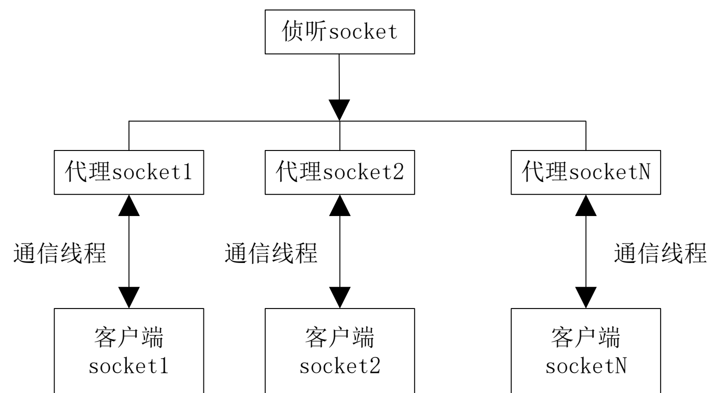


图 9-14 客户端连入服务端之后通信结构

如上图 9-14 所示，每个客户端都对应一个通信线程，每个线程中均有一个“数据接收泵”。本例服务端的效果图见下图 9-15：

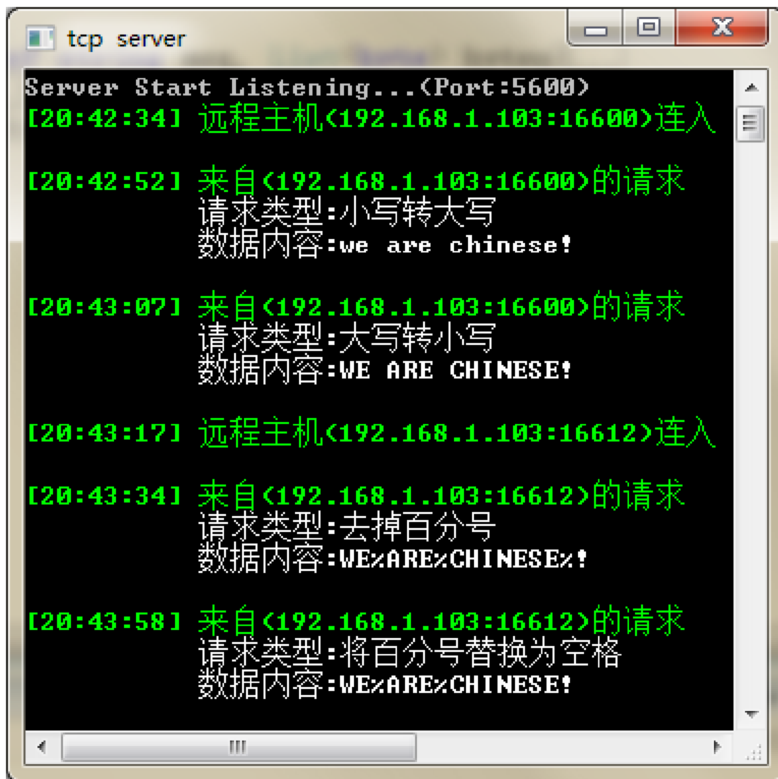


图 9-15 TCP 通信服务端实例效果图

如上图 9-15 所示，先后有两个客户端连入服务端，分别请求“小写转大写”、“大写转小写”和“去掉百分号”、“将百分号替换为空格”。

9.2.3 编写客户端

客户端包含四个重要部分：数据发送、数据解析、数据接收泵（循环）、用户输入泵（循环）。其中数据发送和数据解析跟服务端一致，下面主要介绍数据接收泵和用户输入泵：

数据接收泵：

与服务端一样，客户端的数据接受泵也存在一个独立的线程中：

```
//Code 9-6
static void Receive_th(object obj)    //NO.1
{
    Socket client_socket = obj as Socket;
    string remote_ip = (client_socket.RemoteEndPoint as IPEndPoint).Address.ToString(); //NO.2
    int remote_port = (client_socket.RemoteEndPoint as IPEndPoint).Port;    //NO.3
    List<byte> bytes_already_recv = new List<byte>();
    byte[] bytes_to_recv = new byte[256];
    int length_to_recv = 0;
    do    //NO.4
    {
        length_to_recv = client_socket.Receive(bytes_to_recv);
        byte[] tmp = new byte[length_to_recv];
        Buffer.BlockCopy(bytes_to_recv, 0, tmp, 0, length_to_recv);
        bytes_already_recv.AddRange(tmp);
        int msg_type = 0;
```

```
            string msg = null;
            while (ResolveInfos(ref msg_type, ref msg, bytes_already_recv))    //NO.5
            {
                Console.ForegroundColor = ConsoleColor.Green;
                Console.WriteLine("[" + DateTime.Now.ToLongTimeString() + "] 来自服务器(" + 
remote_ip + ":" + remote_port + ")的处理结果");    //NO.6
                Console.ForegroundColor = ConsoleColor.White;

                Console.WriteLine("           处理类型:" + GetMsgType(msg_type));
                Console.WriteLine("           处理结果:" + msg);
                Console.WriteLine();
                Console.WriteLine("输入字符串发送:");
            }
        }
        while (length_to_recv != 0);
}
```

如上代码 Code 9-6 所示，Receive_th 方法是一个线程调用方法（通过 Thread 创建），NO.2 和 NO.3 处获取远程服务端的 IP 和端口， NO.4 处开始一个“数据接收泵”，该泵不断地接收来自服务端的数据，每次有数据到达时，先将接收到的数据写入一个字节容器（bytes_already_recv）中，然后开始解析数据（NO.5 处），当解析成功时，将其显示在屏幕。客户端的数据接收线程与服务端的数据接收线程类似，唯一不同的是它不需要处理服务端发送的数据。

用户输入泵：

用户输入泵的实现放在主线程中（Main 方法中）：

```
//Code 9-7
static void Main(string[] args)
{
        Console.Title = "tcp client";
        Socket client = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
        client.Connect(IPAddress.Parse("192.168.1.103"), 5600);    //NO.1
        Console.WriteLine("[" + DateTime.Now.ToLongTimeString() + "] 连接服务器成功...");
        new Thread(new ParameterizedThreadStart(Receive_th)).Start(client);    //NO.2
        string input = "";
        string[] heads = {"1","2","3","4" };
        Console.WriteLine("输入字符串发送:");
        while ((input = Console.ReadLine()) != "quit")    //NO.3
        {
            string[] inputs = input.Split(' ');
            if (inputs.Length >= 2)
            {
                if (heads.Contains(inputs[0]))    //NO.4
                {
                    Send(client, int.Parse(inputs[0]), input.Replace(inputs[0] + " ",""));    //NO.5

                }
```

```
                else
                {
                    Console.WriteLine("输入错误!");
                }
            }
            else
            {
                Console.WriteLine("输入错误!");
            }
        }
}
```

如上代码 Code 9-7 所示，客户端首先连接服务端（NO.1 处），连接成功后，马上创建一个通信线程（NO.2 处），该线程中会创建一个“数据接收泵”。之后主线程中开始了一个“用户输入泵”（NO.3），该泵不断地接收用户输入，检查其输入格式，然后将其发送到服务端（NO.5 处）。

> 注：由于代码调用关系，笔者颠倒了介绍顺序，按照 Code 9-7->Code 9-6->Code 9-2->Code 9-1 的顺序阅读代码，更易理解。

用户输入必须按照“转换类型+空格+需要转换的字符串”这种格式，转换类型包括“1”，“2”，“3”，“4”。本例客户端的效果图见下图 9-16：

图 9-16 TCP 通信客户端实例效果图

如上图 9-16 所示，客户端连入服务端后，分别发送了 4 个字符串转换请求，图中每个方框标示出来的两部分，前面表示“转换类型”，后面表示需要转换的字符串。

9.3 UDP 通信实现

本节以一个简单的局域网即时通讯程序为例，来说明一个 UDP 通信程序的整个结构。通讯程序没有服务端，直接“点到点”地进行消息发送，类似“飞鸽传书”软件。由于没有服务端，所以每次新用户上线时，要想获取局域网中已在线用户信息，就必须向局域网广播一条“上线”的消息，局域网中其他已在线用户收到该“上线”的消息后，将新用户保存在在线列表，同时将自己的信息（如昵称）反馈回去，新上线用户得到“反馈”消息后，依次将反馈信息（如昵称）添加到自己的在线列表中。

通讯程序中没有服务端，每个客户端都是平等的，因此只需要编写一个客户端程序即可，见下图 9-17：

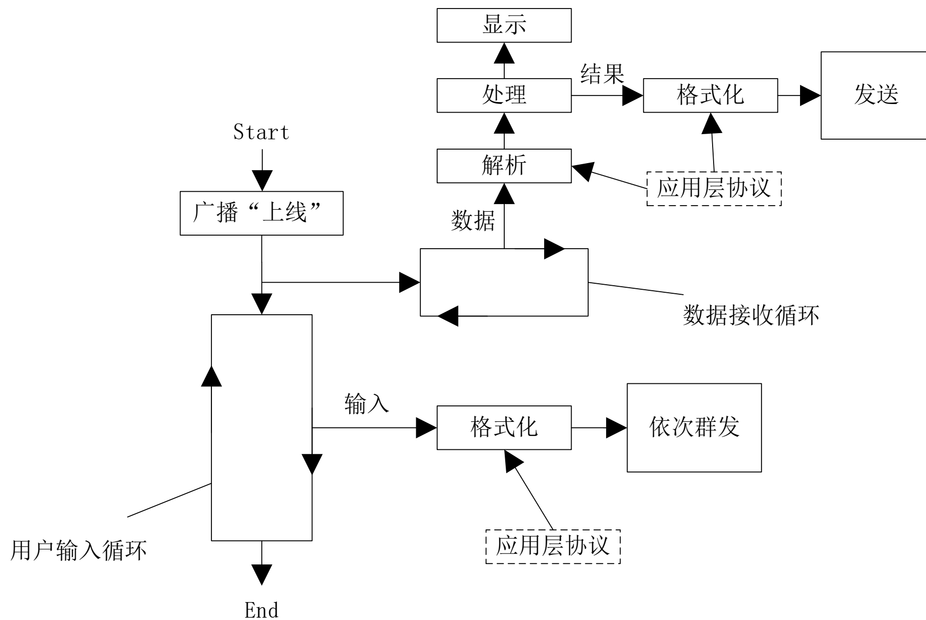


图 9-17 UDP 通信客户端结构

如上图 9-17 所示，客户端开始广播一条“上线”的消息，然后创建一个“数据接收泵”，该泵不断地接收来自远端（未知终端）的数据，当有数据到达时，需要先按照事先定义的“应用层协议”解析数据，数据解析完成后才能处理。程序中还需要一个“数据接收泵”来循环接收用户输入，比如发送内容、退出命令等。我们很容易发现，接收网络数据和接收用户输入两个过程同时进行，因此程序必须使用到多线程。

客户端的收、发数据均应该遵守事先定义好的“应用层协议”，只有这样双方通信才能进行顺利。

> 注：通讯程序是以控制台应用程序的方式实现，因此需要定义一个”数据接收循环（泵）”来循环接受用户输入。

9.3.1 定义应用层协议

通讯程序有四种数据类型，见下表 9-3：

表 9-3 数据类型

序号	数据类型（string）	说明

1	login	新用户上线，附带自己的昵称
2	login_back	新用户上线，已上线用户反馈自己的信息
3	send_smg	在线用户发送文本消息
4	logout	在线用户下线

如上表 9-3 所示，假设一共有四种类型消息，我们可以按照下图 9-18 设计应用层协议的数据结构：

```
type:login;              //数据类型，
                           还可以取值：
                           login_back、send_msg、logout
user_name:xiaozhi_5638;  //数据发送方的昵称
time:12：20：30;         //数据发送时间
content:msg              //文本消息内容（当type==send_msg）
```

图 9-18 应用层协议数据结构

如上图 9-18 所示，应用层协议使用“文本”格式，而非“字节流”格式，共有四对数据，每对数据之间使用分号（;）相隔开来，第一对数据表示数据类型，“type”为标示，冒号后面可以取值：“login”（新用户上线）、“login_back”（新用户上线，已经在线用户反馈自身信息）、“send_msg”（在线用户发送文本消息）以及“logout”（在线用户下线）；第二对数据表示数据发送方的昵称，“user_name”为标示，冒号后面为数据发送方的昵称；第三对数据表示数据发送的时间，“time”为标示，冒号后面为数据发送方发送数据的时间；第四对数据表示文本消息，“content”为标示，冒号后面为文本消息内容（此数据项仅当 type==“send_msg”时有效）。数据发送方必须按照此协议发送数据，数据接收方必须按照此协议接收数据。

> 注：之所以将 TCP 通信中应用层协议数据结构设计成“字节流”的形式，是因为 TCP 通信中数据是以“流”的形式传输，以字节流的形式格式化数据后，更方便程序判断数据边界；UDP 通信中数据以“数据报”的形式传输，每次接收到的数据是完整的，对于当前局域网即时通讯实例来讲，协议使用“文本”的形式更方便程序处理数据，当然，我们完全也可以将 UDP 通信中应用层协议数据结构设计成“字节流”的形式。

9.3.2 编写客户端

由于每个客户端都需要保存一个在线列表，因此需要定义一个在线用户类型 User，存储在线用户的 IP、端口以及昵称，客户端主要包含五个重要部分：在线用户类型、数据发送、数据解析、数据接收泵、用户输入泵：

在线用户类型：

定义一个 User 结构体，存储在线用户的昵称、IP 以及端口：

```
//Code 9-8
struct User
{
    string _user_name;    //nick name
    string _remote_ip;    //remote ip
    int _remote_port;     //remote port
    public string User_Name
```

```
    {
        get
        {
            return _user_name;
        }
        set
        {
            _user_name = value;
        }
    }
    public string Remote_IP
    {
        get
        {
            return _remote_ip;
        }
        set
        {
            _remote_ip = value;
        }
    }
    public int Remote_Port
    {
        get
        {
            return _remote_port;
        }
        set
        {
            _remote_port = value;
        }
    }
    public User(string user_name, string remote_ip, int remote_port)
    {
        _user_name = user_name;
        _remote_ip = remote_ip;
        _remote_port = remote_port;
    }
}
```

如上代码 Code 9-8 所示，结构体 User 仅仅存储了在线用户的昵称、通信 IP 和通信端口。

数据发送：

按照事先定义的应用层协议格式化需要发送的数据，再将数据发送给指定终端（IP 和端口确定）：

```
//Code 9-9
```

```
static void Send(Socket client, string remote_ip, int remote_port,string type,string user_name,string content)
{
    string str_to_send = "";
    str_to_send += "type:" + type + ";";    //NO.1
    str_to_send += "user_name:" + user_name + ";";    //NO.2
    str_to_send += "time:" + DateTime.Now.ToString("hh：mm：ss")+";";    //NO.3
    str_to_send += "content:" + content;    //NO.4

    byte[] bytes_to_send = Encoding.Unicode.GetBytes(str_to_send);
    IPEndPoint remote = new IPEndPoint(IPAddress.Parse(remote_ip), remote_port);
    client.SendTo(bytes_to_send, remote);      //NO.5
}
```

如上代码 Code 9-9 所示，先将字符串拼接成符合协议规定的格式（NO.1、NO.2、NO.3 以及 NO.4 处），然后通过 Socket 发送给指定远程终端（NO.5 处）。

数据发送方必须遵守事先定义好的应用层协议，通信才能顺利进行。

数据解析：

按照事先定义好的应用层协议，解析接收到的数据：

```
//Code 9-10
static bool ResolveInfos(ref string type, ref string user_name, ref string time, ref string content, string data)
{
    string[] items_split = data.Split(';');
    if (items_split.Length == 4)
    {
        type = items_split[0].Split(':')[1];    //NO.1
        user_name = items_split[1].Split(':')[1];    //NO.2
        time = items_split[2].Split(':')[1];    //NO.3
        content = items_split[3].Split(':')[1];    //NO.4
        return true;
    }
    else
    {
        type = null;
        user_name = null;
        time = null;
        content = null;
        return false;
    }
}
```

如上代码 Code 9-10 所示，按照事先定义的应用层协议，将接收到的数据解析成单独可识别的值（NO.1、NO.2、NO.3 以及 NO.4 处）。

数据接收泵：

跟 TCP 通信类似，UDP 通信同样需要一个接收远程数据的“泵”，该泵存在于一个独立的线程中：

```
//Code 9-11
static void Receive_th(object obj)    //NO.1
{
        Socket client_socket = obj as Socket;
        byte[] bytes_to_recv = new byte[256];
        int length_to_recv = 0;
        EndPoint remote = new IPEndPoint(IPAddress.Any, 0);
        do    //NO.2      receive loop
        {
                length_to_recv = client_socket.ReceiveFrom(bytes_to_recv, ref remote);
                string str_recv = Encoding.Unicode.GetString(bytes_to_recv, 0, length_to_recv);
                string remote_ip = (remote as IPEndPoint).Address.ToString();
                int remote_port = (remote as IPEndPoint).Port;
                string type = null;
                string user_name = null;
                string time = null;
                string content = null;
                if (ResolveInfos(ref type, ref user_name, ref time, ref content, str_recv))    //NO.3
                {
                        if (user_name != _my_user_name && _my_user_name != "")
                        {
                                if (type == "login")    //NO.4
                                {
                                        lock (_sync_obj)
                                        {
                                                _online_users.Add(new User(user_name, remote_ip, remote_port));
                                        }
                                        Console.ForegroundColor = ConsoleColor.Green;
                                        Console.WriteLine("[" + time + "]  【" + user_name + "_" + remote_ip + ":" +
remote_port + "】上线");
                                        Console.ForegroundColor = ConsoleColor.White;
                                        Console.WriteLine("输入消息发送:");
                                        Send(client_socket, remote_ip, remote_port, "login_back",
_my_user_name, "");
                                }
                                else if (type == "login_back")    //NO.5
                                {
                                        lock (_sync_obj)
                                        {
                                                _online_users.Add(new User(user_name, remote_ip, remote_port));
                                        }
                                }
                                else if (type == "send_msg")    //NO.6
```

```
                {
                    Console.ForegroundColor = ConsoleColor.Green;
                    Console.WriteLine("[" + time + "] 来自【" + user_name + "_" + remote_ip +
":" + remote_port + "】的消息:");
                    Console.ForegroundColor = ConsoleColor.White;
                    Console.WriteLine(content);
                    Console.WriteLine("输入消息发送:");
                }
                else if (type == "logout")    //NO.7
                {
                    lock (_sync_obj)
                    {
                        _online_users.Remove(new User(user_name, remote_ip,
remote_port));
                    }
                    Console.ForegroundColor = ConsoleColor.Green;
                    Console.WriteLine("[" + time + "] 【" + user_name + "_" + remote_ip + ":" +
remote_port + "】下线");
                    Console.ForegroundColor = ConsoleColor.White;
                    Console.WriteLine("输入消息发送:");
                }
            }
        }
    }
    while (length_to_recv != 0);
}
```

如上代码 Code 9-11 所示，方法 Receive_th 是一个线程方法（使用 Thread 创建），NO.2 处开始了一个“数据接收泵”，该泵不断地接收来自远程终端的数据，当有数据到达时，先按照事先定义好的应用层协议解析数据（NO.3 处），如果解析成功，则处理数据：如果是新用户上线（NO.4 处），将新用户的信息存放在一个在线列表中（_online_users），然后再将自己的信息反馈给新上线用户；如果是上线反馈信息（NO.5 处），将反馈的已上线用户信息存放在在线列表中（_online_users）；如果是发送文本消息（NO.6 处），将文本消息显示到界面；如果是用户下线（NO.7 处），将下线用户从在线用户列表中移除。

用户输入泵：

用户输入泵的实现在主线程中（Main 方法中）：

```
//Code 9-12
static void Main(string[] args)
{
    Console.Title = "udp client";
    Socket client = new Socket(AddressFamily.InterNetwork, SocketType.Dgram, ProtocolType.Udp);
    client.SetSocketOption(SocketOptionLevel.Socket, SocketOptionName.Broadcast, true); //NO.1
    client.Bind(new IPEndPoint(IPAddress.Any, 5600));    //NO.2
    new Thread(new ParameterizedThreadStart(Receive_th)).Start(client);    //NO.3
    Console.WriteLine("输入您的昵称：");
```

```
    _my_user_name = Console.ReadLine();
    Send(client, "255.255.255.255", 5600, "login", _my_user_name, "");    //NO.4
    Console.Title = "udp client [" + _my_user_name +"]";
    string input = "";
    Console.WriteLine("输入消息发送:");
    while ((input = Console.ReadLine()) != "quit")    //NO.5      input loop
    {
        lock (_sync_obj)
        {
            foreach (User u in _online_users)
            {
                Send(client, u.Remote_IP, u.Remote_Port, "send_msg", _my_user_name, input);
            }
        }
        Console.WriteLine("输入消息发送:");
    }
    foreach (User u in _online_users)    //NO.6
    {
        Send(client, u.Remote_IP, u.Remote_Port, "logout", _my_user_name, "");
    }
    Environment.Exit(0);
}
```

如上代码 Code 9-12 所示，首先需要将 Socket 设置为可以收发广播消息（NO.1 处），然后绑定 Socket 到端口（NO.2 处），紧接着创建一个数据接收的线程（NO.3 处），该线程中会开始一个“数据接收泵”，NO.4 处广播“自己上线”的通知（type==”login”），NO.5 处开始了一个“用户输入泵”，该泵不断地接收用户输入，然后向所有在线用户发送输入的文本消息（type==”send_msg”），NO.6 处表示用户下线，需要向所有的在线用户发送下线通知（type==”logout”）。

> 注：由于代码调用关系，笔者颠倒了介绍顺序，按照 CODE 9-12->CODE 9-11->CODE 9-10->CODE 9-9->CODE 9-8 的顺序阅读代码，更易理解。

客户端正常工作后的通信结构如下图 9-19：

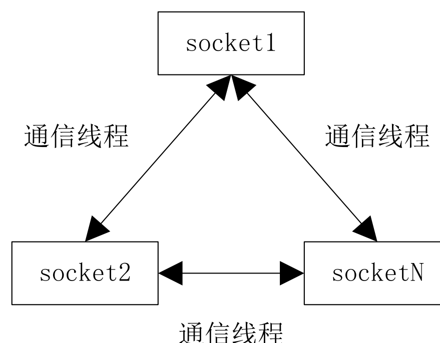


图 9-19 客户端通信结构

如上图 9-19 所示，每个客户端都有一个数据接收的线程，每个线程中均有一个“数据接收泵”。

下图 9-20 为客户端运行效果图：

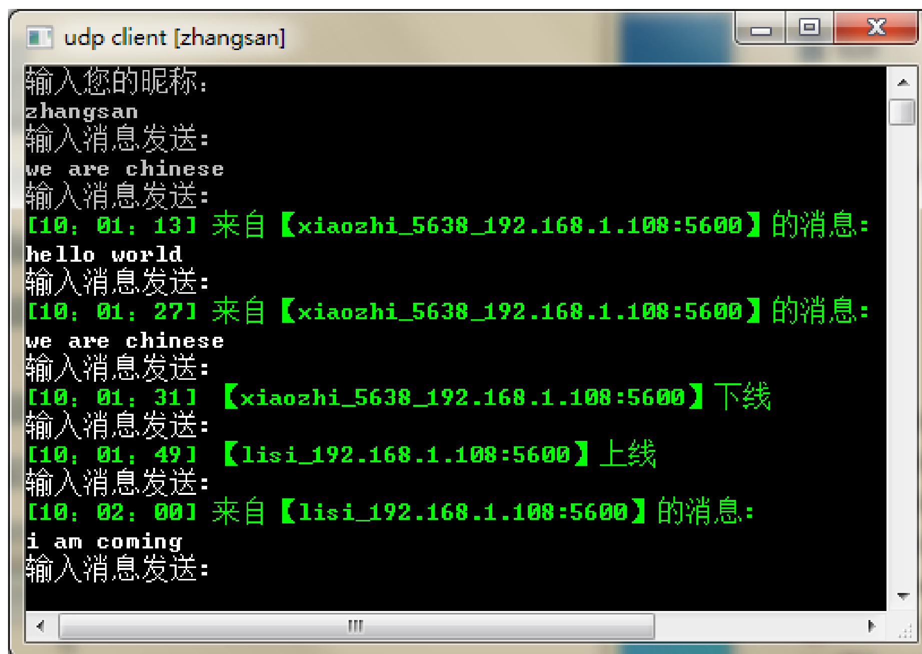


图 9-20 UDP 通信客户端实例效果图

9.4 异步编程在网络编程中的应用

前面介绍的有关 TCP 实现和 UDP 实现中（9.2 与 9.3 节），所有的 Socket 操作均调用的是同步方法，比如 Socket.Connect()、Socket.Accept()、Socket.Send()、Socket.Receive()等等，这些方法都属于“阻塞方法”，换句话说，这些方法“耗时长”的可能性较大、返回时间不确定，会出现阻塞调用线程的情况。

注：有关“阻塞方法”、“调用线程”的概念请参见本书第二章。

其实.NET 中有关 Socket 网络编程类型的同步方法基本上都有对应的异步版本，下表 9-4 列出了 Socket、TcpClient、UdpClient 以及 TcpListener 类型包含的异步方法：

表 9-4 Socket 网络编程类型异步方法

序号	类型	同步方法	异步版本	说明
1	Socket	Socket.Connect()	Socket.BeginConnect() Socket.EndConnect()	异步连接服务端
		Socket.Accept()	Socket.BeginAccept() Socket.EndAccept()	异步侦听客户端连接
		Socket.Send()	Socket.BeginSend() Socket.EndSend()	异步发送数据（包括 SendTo）
		Socket.Receive()	Socket.BeginReceive() Socket.EndReceive()	异步接收数据（包括 ReceiveFrom()）
		Socket.DisConnect()	Socket.BeginDisConnect()	异步断开连接服

			Socket.EndDisConnect()	务端
2	TcpClient	TcpClient.Connect()	TcpClient.BeginConnect() TcpClient.EndConnect()	异步连接服务端
		NetworkStream.Read()	NetwrokStream.BeginRead() NetWrokStream.EndRead()	TcpClient.GetStream()返回一个网络流，使用该流异步接收数据
		NetworkStream.Write()	NetworkStream.BeginWrite() NetworkStream.EndWrite()	TcpClient.GetStream()返回一个网络流，使用该流异步发送数据
3	UdpClient	UdpClient.Send()	UdpClient.BeginSend() UdpClient.EndSend()	异步发送数据
		UdpClient.Receive()	UdpClient.BeginReceive() UdpClient.EndReceive()	异步接收数据
4	TcpListener	TcpListener.AcceptSocket()	TcpListener.BeginAcceptSocket() TcpListener.EndAcceptSocket()	异步侦听来自客户端的连接，返回 Socket 类型
		TcpListener.AcceptTcpClient()	TcpListener.BeginAcceptTcpClient() TcpListener.EndAcceptTcpClient()	异步侦听来自客户端的连接，返回 TcpClient 类型

如上表 9-4 所示，每一个同步方法均对应一个异步方法，无论是“连接”、“侦听”、“发送”还是“接收”均可以使用异步编程去实现。

9.4.1 小节中以“发送数据”为例说明异步编程在 Socket 程序设计中的应用。

9.4.1 异步发送数据

同步发送数据使用 Socket.Send()方法，以 UDP 通信为例：

```
//Code 9-13
class Program
{
    static void Main()
    {
        //… initialize socket here
        Socket client_socket =   new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
ProtocolType.Udp);
        client_socket.Bind(…);
        byte[] bytes = Encoding.Unicode.GetBytes(“Is Anybody in?”);
        IPEndPoint remote = new IPEndPoint(IPAddress.Parse(“…”),5600);
        client_socket.SendTo(bytes,remote);   //NO.1
        Console.WriteLine(“Send OK”);
    }
}
```

如上代码 Code 9-13 所示，NO.1 处将字符串同步发送到指定主机。如果使用异步方式发送数据，

见下面代码：

```
//Code 9-14
class Program
{
    static void Main()
    {
        //… initialize socket here
        Socket client_socket =  new Socket(AddressFamily.InterNetwork, SocketType.Dgram, 
ProtocolType.Udp);
        client_socket.Bind(…);
        byte[] bytes = Encoding.Unicode.GetBytes(“Is Anybody in?”);
        IPEndPoint remote = new IPEndPoint(IPAddress.Parse(“…”),5600);
        client_socket.BeginSendTo(bytes, 0, bytes.Length, SocketFlags.None, remote, new 
AsyncCallback(OnSend), client_socket);    //NO.1
        Console.Read();
    }
    static void OnSend(IAsyncResult ar)
    {
        int length_to_send = (ar.AsyncState as Socket).EndSendTo(ar);   //NO.2
        Console.WriteLine("Send OK");
    }
}
```

如上代码 Code 9-14 所示，NO.1 处使用 Socket.BeginSendTo()方法开始一个异步发送过程，并为该方法提供一个 AsyncCallback 的回调参数，该方法的调用不会阻塞调用线程，我们在回调方法 OnSend 中使用 Socket.EndSendTo()方法结束异步发送过程，该方法返回实际发送的数据长度。

9.4.2 异步实现"泵"结构

通常要想开始一个“泵”循环，需要单独创建一个线程，使用异步编程，不需要显示地创建线程，下面代码显示普通方式实现“泵”结构（假设以 TCP 通信数据接收泵为例）：

```
//Code 9-15
static void Main()   //NO.1    main thread
{
    //…
    new Thread(new ThreadStart(Receive_th)).Start();
    //…
}
static void Receive_th()   //NO.2   other thread
{
    while(…)
    {
        //…
        //call synchronous method
        socket.Receive(…);
        //…
```

```
    }
}
```

如上代码 Code 9-15 所示，while 循环运行在一个独立的线程中（NO.2 处），在 while 循环中调用同步方法。使用异步编程实现“泵”结构，见下代码：

```
//Code 9-16
staic void Main()    //NO.1    main thread
{
    //…
    socket.BeginReceive(…,…,…,…,new AsyncCallback(OnReceive),socket);    //NO.2
    //…
}

static void OnReceive(IAsyncResult ar)    //NO.3
{
    Socket socket = ar.AsyncState as Socket;
    int length_to_recv = socket.EndReceive(ar);
    //…
    DealData(…);    //NO.4
    socket.BeginReceive(…,…,…,…,new AsyncCallback(OnReceive),socket);    //NO.5
}
```

如上代码 Code 9-16 所示，NO.2 处开始了一个异步过程，在回调方法 OnReceive 中，当有数据到达，处理完数据（NO.4）后马上又开始另一个新的异步过程（NO.5 处），继续异步接收数据。我们可以看到，异步编程也能实现循环接收数据，但是我们却看不到显示去创建的线程，也看不到类似 while 这样的循环语句。

> 注：本节只是简单地介绍异步编程在 SOCKET 通信中的使用，如需具体了解异步编程的原理，请参见本书前面介绍有关“异步编程模型”的章节。

再看代码 Code 9-16 中，当调用 socket.EndReceive()接收到数据后，马上调用 DealData()去处理数据，数据处理完后，才开始另外一个异步过程，换句话说，数据处理完之前，是不能接收新的数据。这个“泵”的功能效果跟使用多线程时一样，见下图 9-21：

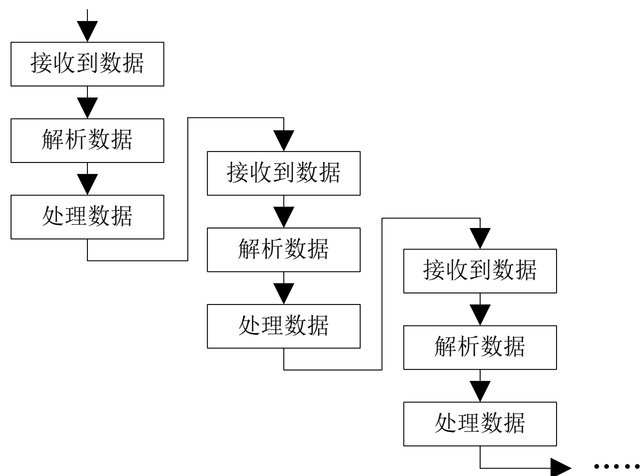


图 9-21 顺序执行的循环

如上图 9-21 所示，一次循环结束后，方可开始下一次循环，这样有一个好处就是，先接收到的数据肯定先处理，而后接收到的数据肯定后处理，特别在 TCP 通信中，一次接收到的数据可能不是完整的（数据边界未知），需要按顺序将每次接收到的数据放进一个缓冲区，然后再从缓冲区中解析出一条完整的数据，使用这种“泵”结构接收数据，保证了数据处理的顺序性。而对于 UDP 通信来讲，它接收到的数据本身是完整的，而且是无序的，先发送的数据可能后接收，而后发送的数据可能先接收，因此在 UDP 通信中，没有必要保证先接收到的数据先处理，而后接收到的数据后处理，所以在 UDP 通信中，我们可以使用这种“泵”结构：

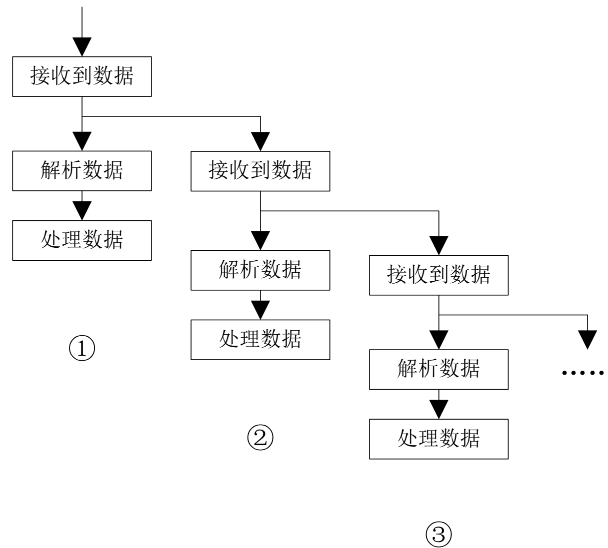


图 9-22 非顺序执行的循环

如上图 9-22 所示，每次循环接收到数据后，马上开始下一次循环，继续接收下一次数据，这样一来，图中第 1、2 和 3 次循环中的数据处理结束顺序不确定（三次数据同时进行处理），可能第 1 次循环接收到的数据后处理完毕，而第 3 次循环接收到的数据先处理完毕。这种“泵”实现很简单，只需要将代码 Code 9-16 中 NO.4 和 NO.5 处代码换一下位置即可：

```
//Code 9-17
staic void Main()    //NO.1    main thread
{
    //…
    socket.BeginReceiveFrom(…,…,…,…,…,new AsyncCallback(OnReceive),socket);    //NO.2
    //…
}

static void OnReceive(IAsyncResult ar)    //NO.3
{
    Socket socket = ar.AsyncState as Socket;
    EndPoint remote = new IPEndPoint(IPAddress.Any,0);
    int length_to_recv = socket.EndReceiveFrom(ar,ref remote);
    socket.BeginReceiveFrom(…,…,…,…,…,new AsyncCallback(OnReceive),socket);    //NO.5
    //…
    DealData(…);    //NO.4
}
```

如上代码 Code 9-17 所示，代码实现了 UDP 通信中的“数据接收泵”，对比代码 Code 9-16 我

们可以发现，每次接收数据后，没有马上进行处理，而是马上开始下一次异步数据接收过程（NO.5 处），之后才进行数据的其它操作（NO.4 处）。

我们可以看见，图 9-21 那种数据接收泵会影响数据处理效率，原因很简单，因为只有数据处理完毕后，下一次数据接收才能开始，数据处理会阻塞“泵”的运转，网络接收的数据会一直累积得不到及时的处理；而图 9-22 那种数据接收泵不会影响数据处理效率，因为数据处理并没有阻塞“泵”的运转，数据的处理发生在另外线程（虽然看不见明显创建线程的痕迹）。

> 注：异步实现循环效果相对来讲，是难以理解的，注意每次循环中的变量都是不一样的。另外，本章中有关 SOCKET、TCPCLIENT、TCPLISTENER 以及 UDPCLIENT 等类型的详细信息并没有提及到，比如它们一些具体方法的使用等，这些需要读者自己阅读其它资料进行了解。

9.5 本章回顾

本章主要介绍了.NET 中 Socket 网络编程的有关知识，主要涉及到了 TCP 通信概念、UDP 通信概念，以及两者的特点和使用场合，之后分别给出了两种通信方式的具体应用实例，虽然实例相对简单，但是它们的程序结构基本上就是一些复杂通信程序的大概结构，在讲解实例时，还提到了“应用层协议”的有关内容。章末结合本书前面介绍的知识，讲到了异步编程在通信程序中的应用。

9.6 本章思考

1.如果两台主机之间需要通信，为何仅仅使用 IP 地址是不够的？

A：因为 IP 只能确定主机，不能定位通信程序，通信过程表面上是发生在两台主机之间，实质上是发生在两个通信程序之间。

2.举例说明有哪些应用层协议。

A：常见应用层协议如下：

（1）域名系统(Domain Name System，DNS)：用于实现网络设备名字到 IP 地址映射的网络服务。

（2）文件传输协议(File Transfer Protocol，FTP)：用于实现交互式文件传输功能。

（3）简单邮件传送协议(Simple Mail Transfer Protocol, SMTP)：用于实现电子邮箱传送功能。

（4）超文本传输协议(HyperText Transfer Protocol，HTTP)：用于实现 WWW 服务。

（5）简单网络管理协议(simple Network Management Protocol，SNMP)：用于管理与监视网络设备。

（6）远程登录协议(Telnet)：用于实现远程登录功能。

3.简述通信协议的本质。

A：所有的通信协议本质上都是一种数据结构，通信双方都必须按照这种数据结构规定的形式去发送或接收（解析）数据。

4.简述 TCP 通信与 UDP 通信的区别。

A：基于 TCP 协议的通信在进行数据交互之前需要先建立连接，类似打电话，这种通信方式保证了数据传输的正确性、可靠性；基于 UDP 协议的通信在进行数据传输之前不需要建立连接，类似发短信，这种通信方式不能保证数据传输的正确性。

5.“泵”结构对通信编程有哪些作用？

A：“泵”结构保证了通信双方程序能够持续正常运转，如持续监听、持续接收数据等，它能为通信程序提供持续运行的动力。

第十章 动力之源：代码中的“泵”

“循环”语句作为一种常见的代码结构，几乎存在于我们写的任何一段程序代码中，它负责实现“代码重复执行”的功能，像.NET 中常见的 While 循环、Do-While 循环、For 循环等等。从微观上看这些循环语句，它们仅仅只是简单地控制代码运行流程，但如果从宏观上去看一些稍微复杂的模块、系统，我们会发现，“循环”原来是整个程序的“动力之源”，我们称这些能够支撑整个模块乃至整个系统长时间重复运作的结构为“泵”。

10.1 “泵”的概念

10.1.1 现实生活中的“泵”

平时生活中提到“泵”这个词，会让我们联想到“水泵”，它主要用于传输类似水这样的液体，下图 10-1 为一种类型的水泵：

图 10-1 水泵

水泵一般包含两个口，一个是液体入口，一个是液体出口，泵能够长时间、不断循环地将液体从一个地方传输到另外一个地方，为液体流动提供动力。现实生活中的泵主要有两个特征：

1）持续性；

泵能够长时间、不间断地干着同一件事情，像汽车发动机一样，启动后会一直重复地做着“转动”运动。

2）动力性。

泵具备传输液体的功能，能为液体流动提供动力支持，尤其是在地势相差很大的场合，泵能够将处于地势低的液体传送到地势高的地方。

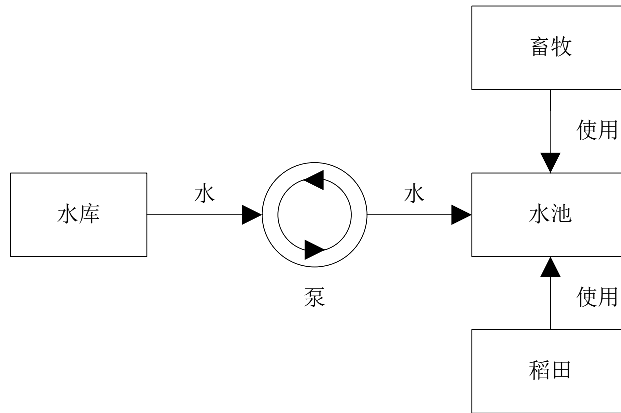


图 10-2 水泵的作用

上图 10-2 显示了水泵的一个简单使用场合，它负责将水从水库传送到水池，供稻田和畜牧等使用。

10.1.2 代码中的"泵"

在我们刚学习计算机编程语言时，上课需要写一些实践程序，那时候我们不知道 Web 网站，也不知道桌面程序，更不知道手机 APP，我们只能写一些简单的控制台程序，比如我们测试“冒泡排序”的代码这样写：

```
//Code 10-1
class Program
{
    static List<int> list = new List<int>() { 89, 14, 59, 32, 29, 78, 2, 77, 89, 73 };
    static void Main(string[] args)    //NO.1    entry
    {
        Console.WriteLine("排序前数组为：");
        foreach (var item in list)
        {
            Console.Write(string.Format("{0} ", item));
        }
        int temp = 0;
        for (int i = list.Count; i > 0; i--)
        {
            for (int j = 0; j < i - 1; j++)
            {
                if (list[j] > list[j + 1])
                {
                    temp = list[j];
                    list[j] = list[j + 1];
                    list[j + 1] = temp;
                }
            }
        }
        Console.WriteLine();
```

```
            Console.WriteLine("冒泡排序后数组为：");
            foreach (var item in list)
            {
                Console.Write(string.Format("{0} ", item));
            }
            Console.Read();    //NO.2    stop
        }
}
```

如上代码 Code 10-1 所示，一个简单的控制台程序，Main()方法为程序入口（NO.1 处），它被系统调用。冒泡排序完成后，我们将结果显示在屏幕上，NO.2 处设置一个“等待点”，当时老师告诉我们之所以要在这里增加一行“Console.Read();”是为了让我们能看到排序后的输出结果，不然黑屏显示后马上就会关闭，这种解释没错，至少从功能上达到的就是这种效果。现如今再看这段代码时，我们应该要有更深的理解。

程序的运行是“流水型”的，有起点也有终点，从微观上看，程序是由许许多多的线程组成，每个线程有运行开始也有运行结束，也就是说，一个线程开始执行后，理论上讲，它必须在某一时刻结束。而如果一个程序只有一个线程，那么该线程结束就意味着整个程序退出（程序退出意味着操作系统会清理回收它活动时占用到的资源），要想线程处于持续工作状态而不马上结束，唯一的办法就是在线程中调用阻塞方法或者线程中包含循环，从实用角度上讲，调用阻塞方法没有什么实际作用，因为阻塞方法大多时候只能处理一件事件，阻塞方法返回后线程结束，而对于循环来说，每次循环都能处理一件事情，多次循环就可以持续处理一类事情，见下图 10-3：

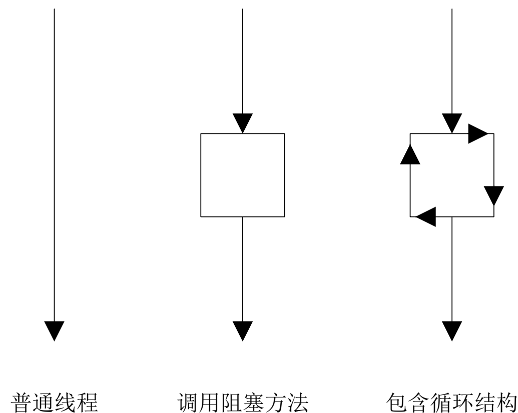


图 10-3 三种线程

如上图 10-3 所示，左边为普通线程，线程开始后，马上结束；中间为调用了阻塞方法的线程，阻塞方法耗时较长，但是当它返回后，线程也会马上结束；右边为包含循环结构的线程，该线程能够持续处理一类问题。

> 注：“阻塞方法”和“非阻塞方法”是一个相对概念，它们之间并没有准确的界线，我们可以认为耗时超过 10s 的方法属于“阻塞方法”，也可以认为耗时超过 100MS 的方法就应该属于“阻塞方法”。图 10-3 中的普通线程就是指只调用非阻塞方法的线程，有关“阻塞”与“非阻塞”的概念请参见本书第二章。

大多数系统或者软件程序不可能一开始运行就马上结束，通常情况下，它们都会持续、不间断地循环处理一类问题，这个时候程序代码中肯定会包含循环结构，我们称能够持续处理一

类问题的循环结构为“泵”，和生活中的“水泵”一样，代码中的“泵”结构能够为程序提供持续运行的动力。.NET 代码中的常见循环结构有：

1）While 循环；

```
//Code 10-2
While(条件)
{
	//do some work
}
```

2）Do-While 循环；

```
//Code 10-3
do
{
	//do some work
}
while(条件);
```

3）For 循环；

```
//Code 10-4
for(int i=0;i<最大值;++i)
{
	//do some work
}
```

4）Foreach 循环。

```
//Code 10-5
foreach(…)
{
	//do some work
}
```

上面代码 Code 10-2、Code 10-3、Code 10-4 以及 Code 10-5 中的四种循环结构中，后两种主要用于遍历容器中的元素，一般很少当作泵来使用，而 While 循环和 Do-While 循环则通常当作泵来使用。

10.1.3 代码中“泵”的作用

经过前面两小节的讨论，我们应该很容易知道代码中“泵”的重要性，和生活中的“水泵”一样，它主要有以下两个作用：

1）持续性；

代码中的泵能够让线程持续运行而不是很快结束，这是程序（线程）持续工作的前提。

2）动力性。

既然水泵能够产生动力，将水等液体从一个地方传送到另外一个地方，代码中泵照样具备“提供动力”的特性，它能够将“数据”从一个地方搬到另外一个地方，供其他人（模块）使用，见下图 10-4：

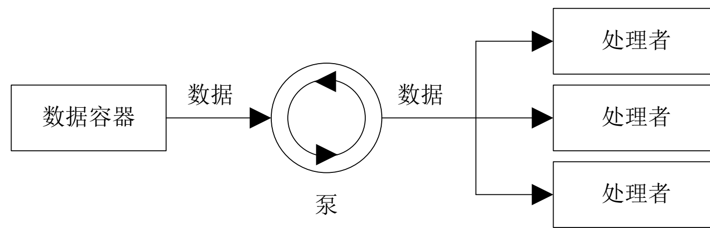


图 10-4 代码中“泵”的动力效果

上图 10-4 显示了代码中泵的动力效果，它能将某个地方的数据源源不断地传送给使用者。

在一个典型的“生产者-消费者”模式系统中，“泵”的作用尤其重要，生产者不停地将数据存入数据容器，消费者需要使用泵源源不断地将数据从容器中取出，进而传送给数据处理者，泵是消费者能够持续工作的核心部件，见下图 10-5：

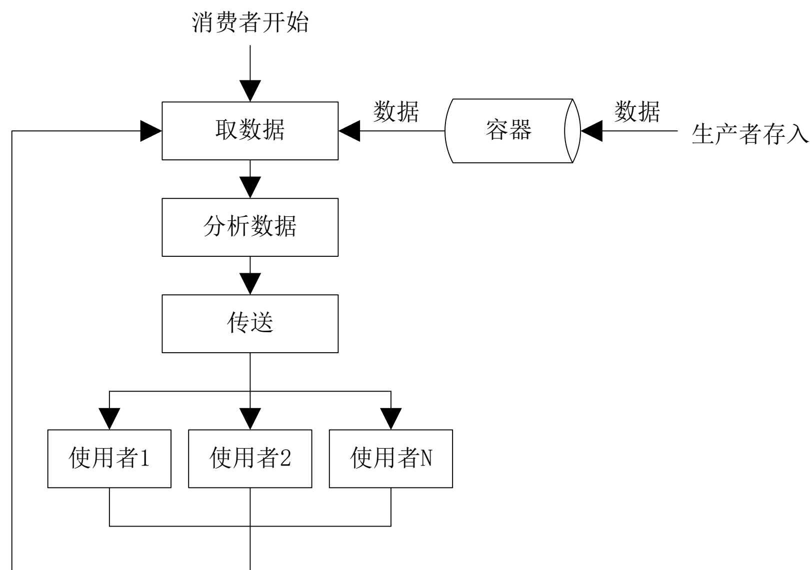


图 10-5 “生产者-消费者”模式中的泵

如上图 10-5 所示，消费者中包含一个泵结构，它是消费者持续稳定工作的支柱。

> 注：“泵”结构在“生产者-消费者”模式中起到了非常关键的作用，“很不幸的是”，任何一个软件系统总会有若干模块属于“生产者-消费者”模式，比如 WINDOWS 操作系统中，用户鼠标键盘等外设的输入可以看成是“生产者”，而操作系统内部肯定会有一个“泵”结构不断地获取用户外设输入，然后传递给其他处理者。更详细的有关常见的“泵”结构请参见 10.2 节。

10.2 常见“泵”结构

本节将介绍几种常见的“泵”结构，我们可以从以下这些成熟的应用实例中获取灵感，进而将“泵”运用在自己的程序代码中。其中“桌面 GUI 框架”中使用到的泵可以参见第八章，“Socket 通信”中使用到的泵可以参见本书第九章。

10.2.1 桌面 GUI 框架

第八章中在讲“桌面 GUI 框架解密”中已经提到过，桌面程序的 UI 线程中包含一个消息循环（确切的说，应该是 While 循环），该循环不断地从消息队列中获取 Windows 消息，最终调用对应的窗口过程，将 Windows 消息传递给窗口过程进行处理。如果按照本章前面的介绍，消息循环就应该是“泵”，消息队列就应该是“数据容器”，Windows 消息就应该是“数据”，而窗口过程就应该是“处理者”，那么整个结构应该是这样的：

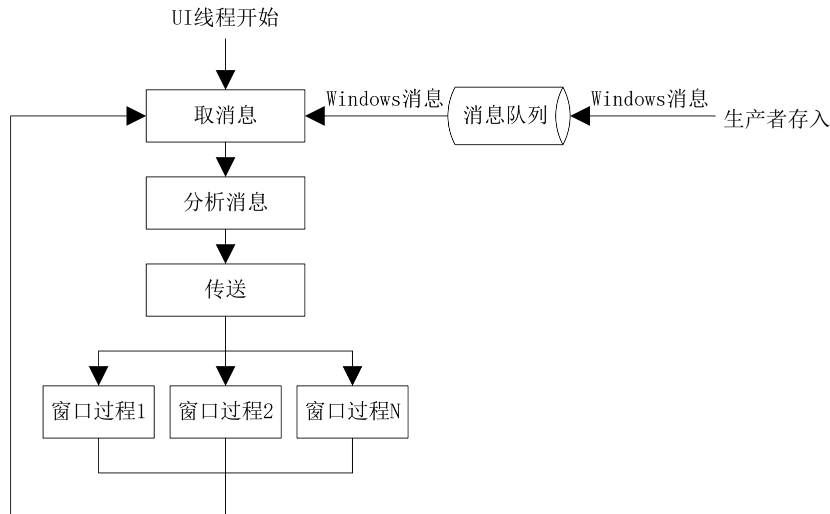


图 10-6 GUI 框架中的“泵”

图 10-5 跟图 10-6 类似，可以说后者是前者的一个具体实例。

到目前为止，我们只是知道 GUI 框架中获取 Windows 消息的结构是一个“泵”结构，它维持着整个桌面 GUI 界面的运转，殊不知，图 10-6 中右侧省略的关于“生产者”的部分也是一种“泵”结构，这部分由操作系统负责，数据的最终源头是计算机鼠标键盘等外设，见下图 10-7：

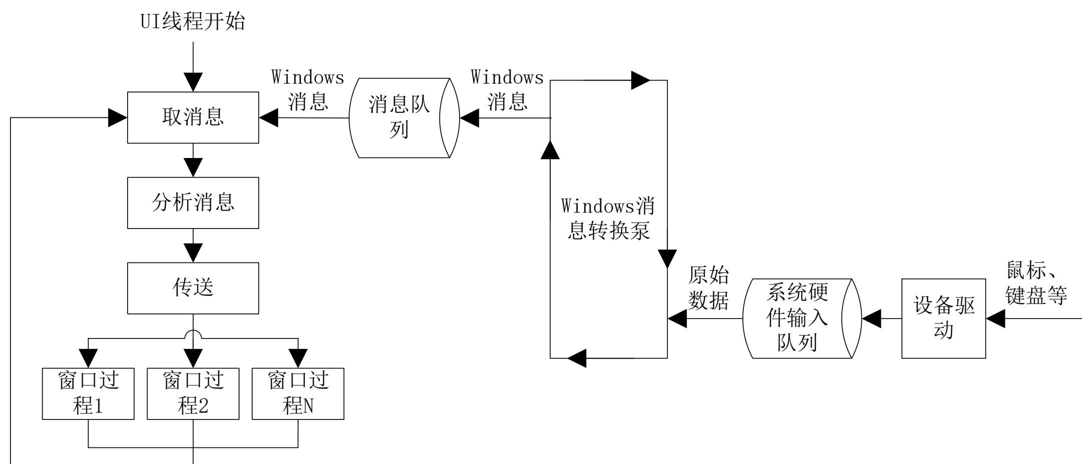


图 10-7 完整的 GUI 框架结构

如上图 10-7，我们可以看到，作为 Windows 消息的生产者，它依旧包含有“泵”结构，源源不断地将用户外设输入信息转换成 Window 消息，进而存入消息队列。正因为有这些“泵”相互配合着工作，才能给整个系统提供持续运转的动力。

注：图 10-7 右侧有关 WINDOWS 消息转换、外设信息采集等结构均属于"示意结构"，并不代表真实情况。

10.2.2 Socket 通信

第九章中讲到 Socket 网络编程时就提到过“泵”的概念，比如“侦听泵”、“数据接收泵”等，如果按照本章前面介绍的“生产者-消费者”模式，数据接收泵如下图 10-8：

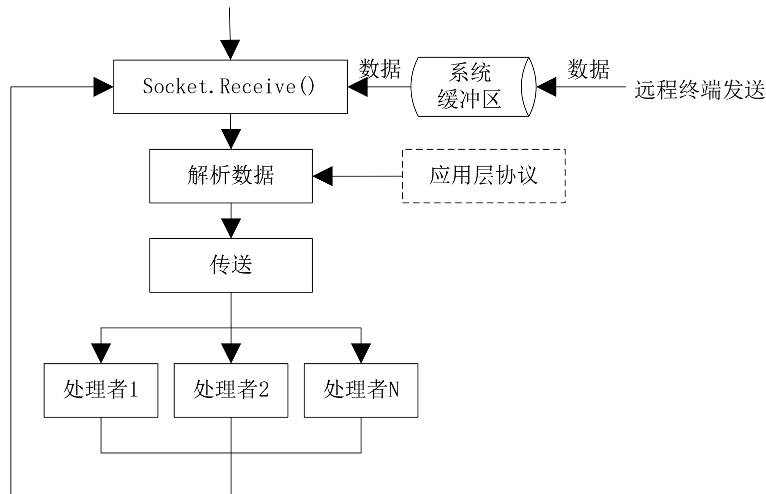


图 10-8 Socket 网络编程中的“泵”

图 10-5 与图 10-8 类似，可以说后者是前者的一个具体实例。

图 10-8 中，如果处理者在处理数据时，耗时太长（即所谓的“阻塞方法”），那么一次循环不能及时完成，系统缓冲区中的数据就会大量累积，得不到及时的处理，这种泵虽然确保了数据的顺序处理（即先接收到的数据先处理完毕，后接收到的数据后处理完毕，前一次处理结束之前，后一次处理不能开始），但是影响了处理效率，如何解决这个问题，请参见下一小节。

注：如何提高"泵"处理数据的效率这个问题，在第九章结尾有所提示。

10.2.3 Web 服务器

本节将详细介绍 Web 服务器的工作原理，并为大家演示“泵”结构是如何在 Web 服务器中担当着重要角色。

在第九章曾提到过，无论是 Web 服务器还是装在普通用户电脑中的浏览器，均要遵守应用层协议：HTTP 协议，而我们一提到 Http 协议时，就会想到它至少有以下两个特点：

1）无连接；

我们常说 Http 协议是一种无连接协议，这可能给人一种误导，这里的“无连接”并不是指遵循 HTTP 协议的 Web 服务器与浏览器之间通信不需要建立连接就可以进行，因为 Http 协议在传输层是使用 TCP 进行传输的，而 TCP 协议是一种面向连接的协议，也就是说，Web 服务器与浏览器通信之前必须建立连接，那么我们常说的“Http 协议是一种无连接的协议”到底是个什么意思呢？

如果我们了解 Web 服务器与浏览器之间的通信过程，我们就能很清楚为什么称 Http 协议是无连接的，

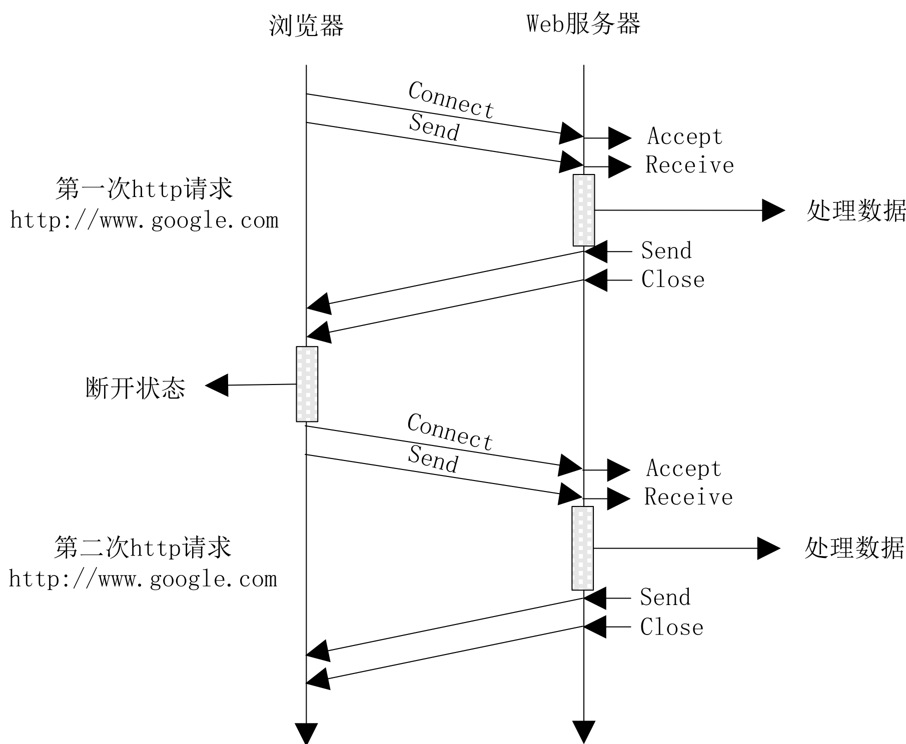


图 10-9 Web 服务器与浏览器通信过程

如上图 10-9 所示，浏览器每次发送 http 请求时，都必须与 Web 服务器建立连接，Web 服务器端请求处理结束后，连接立刻关闭，浏览器下一次发送 http 请求时，必须再一次重新与服务器建立连接。由此我们应该了解，我们所说的 Http 协议是面向无连接的，是指 Web 服务器一次连接只处理一个请求，请求处理完毕后，连接关闭，浏览器在前一次请求结束到下一次请求开始之前这段时间，它是处于"断开"状态的，因此我们称 Http 协议是"无连接"协议。

2）无状态。

Web 服务器除了跟浏览器之间不会保持持久性的连接之外，它也不会保存浏览器的状态，也就是说，同一浏览器先后两次请求同一个 Web 服务器，后者不会保留第一次请求处理的结果到第二次请求阶段，如果第二次请求需要使用第一次请求处理的结果，那么浏览器必须自己将第一次的处理结果回传到服务器端。

如果结合本章前面讲到的"生产者-消费者"模式，我们可以将浏览器端的请求看作是"生产者"，而将 Web 服务器端的请求处理看作成"消费者"，消费者不断地处理来自生产者的"请求"，见下图 10-10：

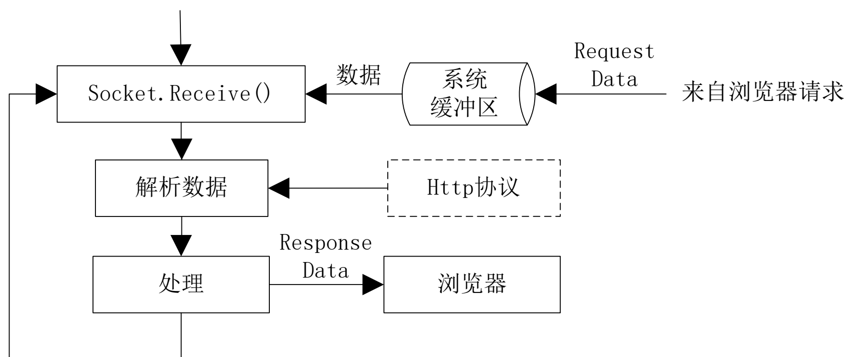


图 10-10 Web 服务器中的"泵"结构

如上图 10-10，Web 服务器中的数据接收泵源源不断地接收来自浏览器的请求数据，然后传递

给其他人（模块）进行处理，处理完毕后，将结果（Reponse Data）发回给浏览器。

> 注：HTTP 协议数据是按照 TCP 协议进行传输的，所以我们完全可以通过 SOCKET 编程来实现一个简单的 WEB 服务器。

我们注意到图 10-10 中，如果 Web 服务器在处理某一次请求时耗时过长，阻塞了泵循环，那么系统缓冲区中就会积累大量请求不能及时被处理，这显然影响了服务器的响应速度，如果我们将处理数据的环节放在泵循环以外，也就是说，数据接收泵只负责接收数据，而不负责处理数据，见下图 10-11：

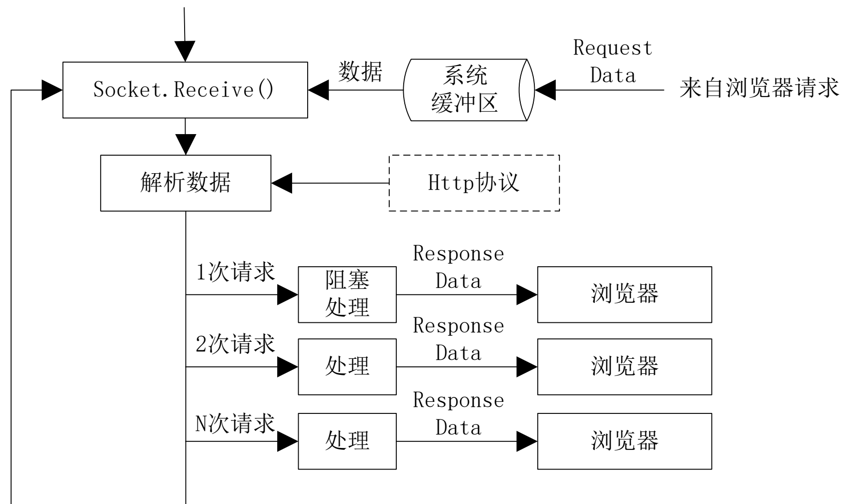


图 10-11 Web 服务器中改进后的“泵”结构

如上图 10-11 所示，改进后的数据接收泵只负责数据的接收，而不负责数据的处理和回复，这样一来，任何阻塞处理数据都不会影响后面的请求处理，因为所有的处理都是“并行”发生的。

使用第九章介绍的 Socket 编程知识，我们可以模拟一个 Web 服务器程序，并对比两种“泵”结构对浏览器请求的影响：

1）主线程；

```
//Code 10-6
class Program
{
    static void Main(string[] args)
    {
        IPAddress localIP = IPAddress.Loopback;
        IPEndPoint endPoint = new IPEndPoint(localIP, 8010);
        Socket server = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
ProtocolType.Tcp);
        server.Bind(endPoint);     //NO.1
        server.Listen(10);
        Console.WriteLine("开始监听，端口号：{0}", endPoint.Port);
        server.BeginAccept(new AsyncCallback(OnAccept), server); //NO.2    asynchronous accept
        Console.Read();    //NO.3
    }
 }
```

如上代码 Code 10-6 所示，创建 Socket 套接字对象，绑定端口 8010（NO.1 处），并开始一个异步侦听过程（NO.2 处），NO.3 处阻塞当前主线程，防止屏幕退出关闭。

2）侦听泵。

```
//Code 10-7
static void OnAccept(IAsyncResult ar)
{
    Socket server = ar.AsyncState as Socket;
    Socket proxy_socket = server.EndAccept(ar);        //NO.1    get proxy socket
    Console.WriteLine(proxy_socket.RemoteEndPoint);
    byte[] bytes_to_recv = new byte[4096];
    int length_to_recv = proxy_socket.Receive(bytes_to_recv);    //NO.2    receive request data
    string received_string = Encoding.UTF8.GetString(bytes_to_recv, 0, length_to_recv);
    Console.WriteLine(received_string);    //NO.3

    string statusLine = "";
    string responseContent = "";
    string responseHeader = "";
    byte[] statusLine_to_bytes;
    byte[] responseContent_to_bytes;
    byte[] responseHeader_to_bytes;

    string[] items = received_string.Split(new string[] { "\r\n" }, StringSplitOptions.None); //items[0]
like "GET / HTTP/1.1"        NO.4      resolve the request string
    if (items[0].Contains("Sleep"))      //NO.5
    {
        Thread.Sleep(1000 * 10);
        statusLine = "HTTP/1.1 200 OK\r\n";
        statusLine_to_bytes = Encoding.UTF8.GetBytes(statusLine);
        responseContent = "<html><head><title>Sleeping Web
Page</title></head><body><h2>Sleeping 10 seconds,Hello Microsoft .NET<h2></body></html>";
        responseContent_to_bytes = Encoding.UTF8.GetBytes(responseContent);
        responseHeader =
string.Format("Content-Type:text/html;charset=UTF-8\r\nContent-Length:{0}\r\n",
responseContent_to_bytes.Length);
        responseHeader_to_bytes = Encoding.UTF8.GetBytes(responseHeader);
    }
    else    //NO.6
    {
        statusLine = "HTTP/1.1 200 OK\r\n";
        statusLine_to_bytes = Encoding.UTF8.GetBytes(statusLine);
        responseContent = "<html><head><title>Normal Web
Page</title></head><body><h2>Hello Microsoft .NET<h2></body></html>";
        responseContent_to_bytes = Encoding.UTF8.GetBytes(responseContent);
        responseHeader =
```

```
string.Format("Content-Type:text/html;charset=UTF-8\r\nContent-Length:{0}\r\n", responseContent_to_bytes.Length);
                responseHeader_to_bytes = Encoding.UTF8.GetBytes(responseHeader);
        }
        proxy_socket.Send(statusLine_to_bytes);    //NO.7
        proxy_socket.Send(responseHeader_to_bytes);    //NO.8
        proxy_socket.Send(new byte[] { (byte)'\r', (byte)'\n' });    //NO.9
        proxy_socket.Send(responseContent_to_bytes);    //NO.10
        proxy_socket.Close();    //NO.11
        server.BeginAccept(new AsyncCallback(OnAccept), server);    //start the next accept    NO.12
}
```

如上代码 Code 10-7 所示，当有浏览器发送 Http 请求时，NO.1 处获得请求连接的代理 Socket，NO.2 处接收浏览器发送的请求数据（Request Data），并将其显示到屏幕（NO.3 处），NO.4 处简单地解析了 Http 请求数据，NO.5 处判断请求 URL 中是否包含“Sleep 字符串”（即 URL 为“http://localhost:8010/Sleep”），如果是，则线程等待 10 秒（模拟耗时操作），最终，按照 Http 协议规定的数据格式，将应答数据（Response Data）发送给浏览器（NO.7、NO.8、NO.9 以及 NO.10 处），数据发送完成后，立即关闭 Socket，意味着服务器与浏览器的连接关闭（NO.11 处），这一切完成后，开始下一次异步侦听过程（NO.12 处）。

注意 Http 请求数据的格式类似如下：

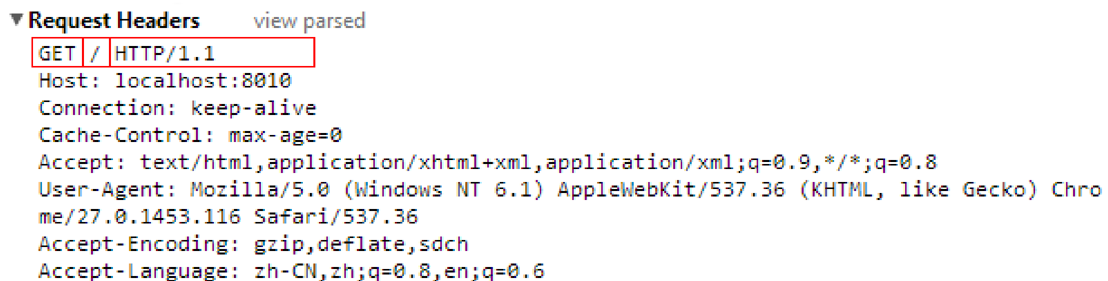
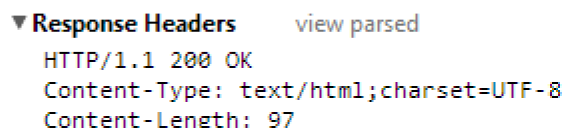


图 10-12 Http 请求数据格式

上图 10-12 表示浏览器向 Web 服务器发送 Http 请求的数据格式，图中方框中的第二格表示请求的路径（图中完整的 URL 应该为：http://localhost:8010/），Web 服务器按照 Http 协议格式解析浏览器发送过来的数据，然后进行处理，将结果按照 Http 协议规定的格式发回浏览器，Http 应答数据格式见下图 10-13：



Html文档内容

图 10-13 Http 应答数据格式

上图 10-13 表示 Web 服务器向浏览器返回数据的格式（方框内表示返回的 Html 文档内容），浏览器按照 Http 协议格式解析服务器发送过来的数据，然后进行网页显示。

串行处理请求的“泵”：

代码 Code10-6 和 Code10-7 最终的效果是：如果浏览器前一次请求 URL 为“http://localhost:8010/Sleep”，服务器端会调用“Thread.Sleep(1000*10);”这行代码，这意味着会阻塞整个“泵”的运转，这时候如果使用 URL 为 http://localhost:8010/请求服务器，Web 服务器不能做出应答，因为前一次请求还未处理完成，也就是说，Http 请求是“串行”处理的，见下图 10-14：

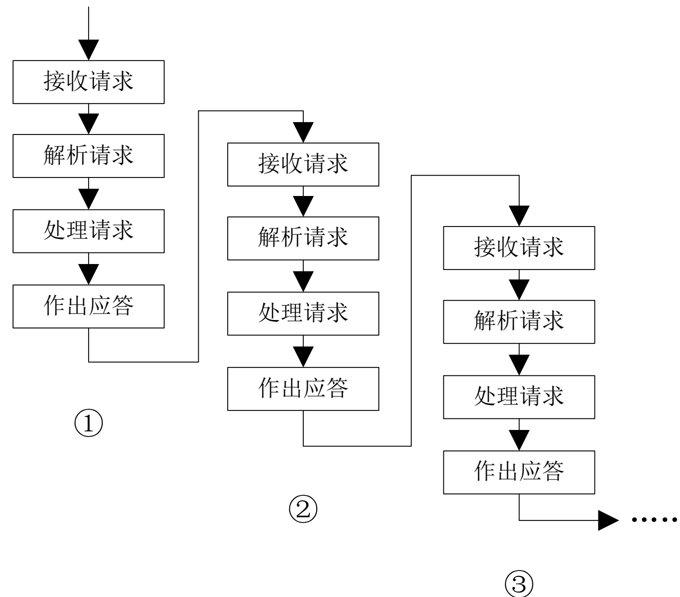


图 10-14 串行处理请求的“泵”

如上图 10-14 所示，第一次请求处理完毕之前，第二次、第三次请求均不可能被处理，也就是说，其余的浏览器请求均处于“等待”状态，见下图 10-15：

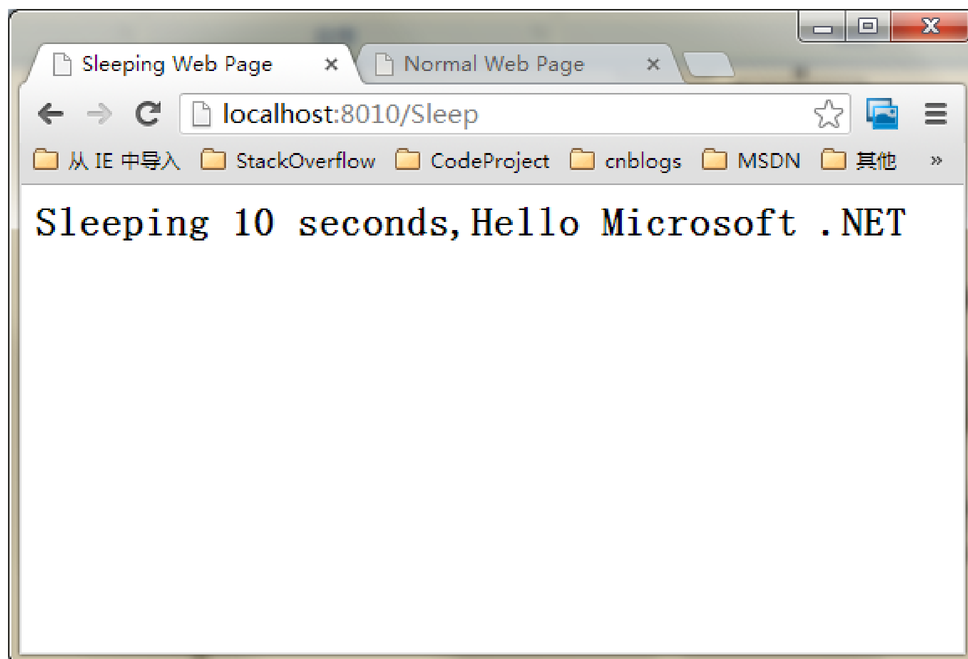


图 10-15 串行处理请求的“泵”效果图

图 10-15 显示，先访问 http://localhost:8010/Sleep 地址，然后马上请求 http://localhost:8010/ 地址，在“Sleeping Web Page”页面返回之前，“Normal Web Page”页面一直处于等待状态，直到“Sleeping Web Page”返回。

并行处理请求的“泵”：

将代码 Code 10-7 中 NO.12 行代码移到 NO.1 下一行，也就是说，侦听到一个浏览器请求后，马上开始另外一个异步侦听过程，这样一来，任何数据处理均不会因为耗时长而影响到后面请求的处理，因为它们都是“并行”处理的：

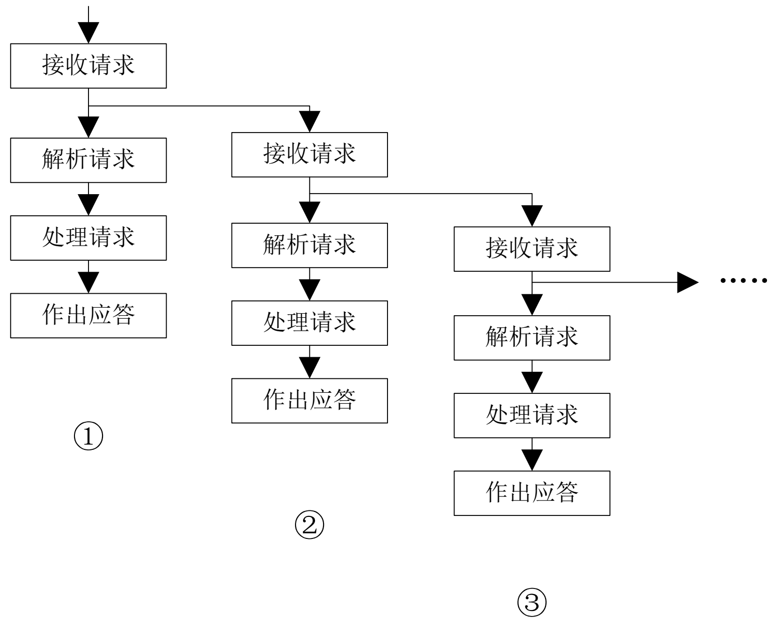


图 10-16 并行处理请求的“泵”

如上图 10-16 所示，第一次请求处理完毕之前，就可以开始第二次甚至第三次请求的处理，不管请求处理是否耗时，其余浏览器请求均能及时返回，见下图 10-17：

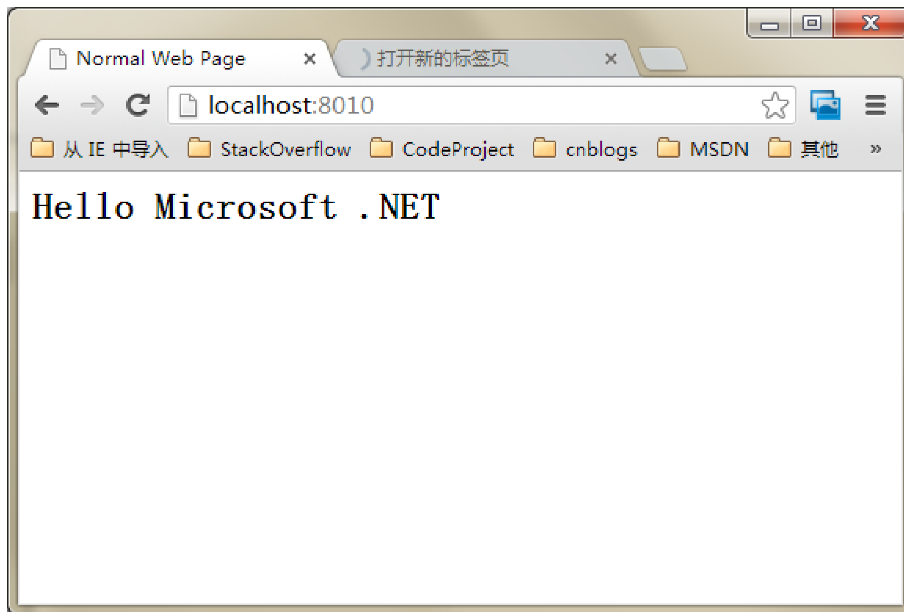


图 10-17 并行处理请求的“泵”效果图

图 10-17 显示，先访问 http://localhost:8010/Sleep 地址，然后马上请求 http://localhost:8010/ 地址，在“Sleeping Web Page”页面返回之前，“Normal Web Page”页面就能立刻返回。

10.3 “泵”对框架的意义

10.3.1 重新回到框架定义

本书第二章中介绍“框架与库”的区别时曾讲到，框架是一个不完整的应用程序，理论上讲，我们不做任何处理，框架就可以正常运行起来，只是这运行起来的框架不具备任何功能或者只具备简单的通用功能。我们在使用框架开发程序时，实际上就是结合实际具体的功能需求，在框架的基础上进行一系列的扩展，最终开发的软件系统能够帮助我们解决某一具体工作。由于主流框架均是由出色的技术团队开发完成，他们无论在技术造诣还是业务了解程度上几乎都比我们要高，因此，借助框架来开发应用程序不仅能够缩短开发周期，还能够保证最后应用程序的稳定性。

既然我们最终的应用程序是在框架的基础之上扩展出来的，这说明应用程序的主要运行逻辑、主要的流程控制均是由框架决定的，框架控制应用程序的启动、决定主要的流程转向，负责调用框架使用者编写的“扩展代码”，总之，框架能够保证最终应用程序的持续正常工作。

> 注：上面提到的“扩展代码”可以理解为开发者在使用框架开发程序时编写的所有代码。

10.3.2 框架离不开“泵”

既然框架能够保证最终应用程序的持续正常工作，按照本章前面的结论，那说明框架内部必然有一种结构能够重复性处理问题，这种结构就是“泵”，泵的“持续性”和“动力性”特性完全满足框架的需求。如果需要将这种抽象关系图形化显示出来，见下图 10-18：

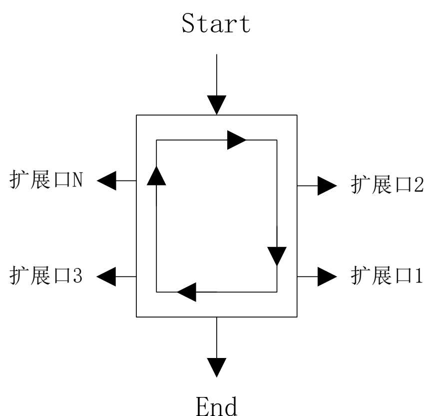


图 10-18 “泵”在框架中的体现

图 10-18 中方框表示框架，循环代表“泵”结构，可以看出，“泵”是框架提供动力的源头，虽然用图 10-18 来轻率地描述框架结构显然是不准确的，但是它足以能够说明“泵”在框架中的重要位置。

> 注：框架控制程序的运行流程称为“控制反转（IoC）”，本书前面章节多次提到过。

10.4 本章回顾

本章主要介绍了代码中“泵”的具体表现形式，以及它对软件系统的重要性。本章可以说是对第八章和第九章的一个补充，第八章中讲“桌面 GUI 框架”时就已经涉及到了“泵”的概念，第九章中讲“Socket 网络编程”时也已经提出了“泵”的定义，本章结合前两章的内容，系统性地对“泵”在编程中的应用做了统一阐述。

10.5 本章思考

1..NET 中循环结构有哪些？分别主要用于什么场合？

A：.NET 中的循环结构有 for 循环、foreach 循环、while 循环以及 do-while 循环。for 循环主要用于重复执行指定次数的操作，foreach 循环主要用于遍历容器元素，while 循环和 do-while 循环主要用于重复执行某项操作直到某一条件满足或不满足为止。代码中的"泵"结构主要由 while 循环来实现。

2.简述代码中“泵”结构的作用。

A：代码中的"泵"结构具备"持续性"和"动力性"两大特点，它能够维持程序的持续运行状态，为程序运转提供动力支持。

3.串行处理数据的泵与并行处理数据的泵之间有什么区别？

A：串行处理数据的泵是按顺序处理数据的，本次数据处理结束之前，下一次处理不能开始；并行处理数据的泵不是按顺序处理数据，所有的数据处理均是同时进行的，没有先后顺序，不能确保先开始处理的数据一定先结束处理，也不能保证后开始处理的数据一定后结束处理。通过异步编程很容易实现两种泵结构。

第十一章 规绳矩墨：模式与原则

软件设计模式（Design Pattern）是一套被反复使用的代码设计经验总结。使用成熟的设计模式可以让代码更容易被他人理解、保证代码可靠性。好的设计，成就好的作品，在软件设计过程中，倘若有一些设计原则（Design Principle）的约束，那我们的软件系统会重构得更好。设计模式和设计原则博大精深，需要我们长时间的大量实践和总结才能真正领悟到其真谛，本章首先以“观察者模式”为例，介绍.NET 中设计模式的应用（其它常用设计模式略），之后详细介绍五大设计原则（简称 Solid 原则）。

11.1 软件的设计模式

在.NET 平台中开发应用程序时，“观察者模式”最为常见，因此本节以“观察者模式”为例对设计模式进行一个简单的介绍。

11.1.1 观察者模式

程序的运行意味着模块与模块之间、对象与对象之间不停地有数据交换，观察者模式强调的就是，当一个目标本身的状态发生改变时（或者满足某一条件），它会主动发出通知，通知对该变化感兴趣的其它对象。我们将通知者称为“Subject”（主体），将被通知者称为“Observer”（观察者），具体结构图如下：

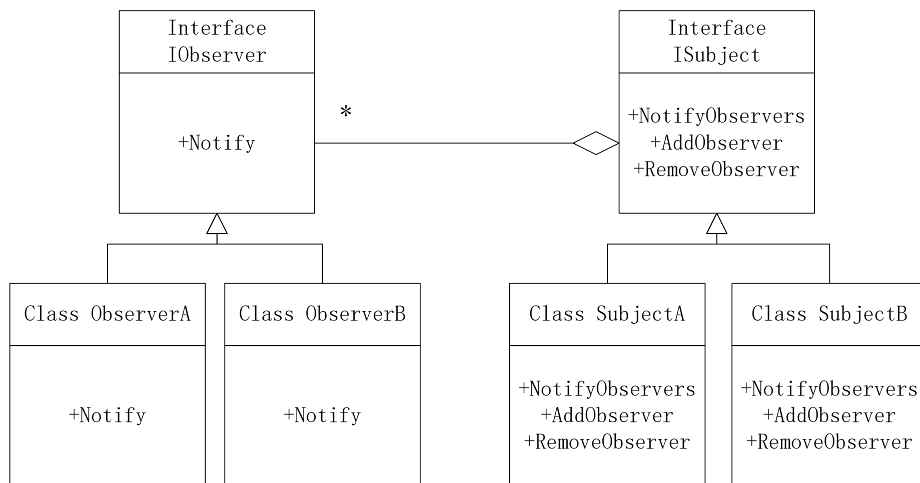


图 11-1 观察者模式中类关系图

如上图 11-1 所示，图中将主体和观察者的逻辑抽象出来两个接口，分别为：ISubject 和 IObserver，ISubject 接口中包含一个通知观察者的 NotifyObservers 方法、一个添加观察者的 AddObserver 方法和一个移除观察者的 RemoveObserver 方法，IObserver 接口中则只包含一个接受通知的 Notify 方法，ISubject 和 IObserver 接口的关系为：一对多，一个主体可以通知多个观察者。

具体代码实现如下：

```
//Code 11-1
```

```
interface ISubject    //NO.1
{
    void NotifyObservers(string msg);
    void AddObserver(IObserver observer);
    void RemoveObserver(IObserver observer);
}
interface IObserver    //NO.2
{
    void Notify(string msg);
}
class MySubject:ISubject
{
    //…
    ArrayList _observers_list = new ArrayList();
    public void AddObserver(IObserver observer)    //NO.3
    {
        if(!_observers_list.Contains(observer))
        {
            _observers_list.Add(observer);
        }
    }
    public void RemoveObserver(IObserver observer)    //NO.4
    {
        if(_observers_list.Contains(observer))
        {
            _observers_list.Remove(observer);
        }
    }
    public void NotifyObservers(string msg)    //NO.5
    {
        foreach(IObserver observer in _observers_list)
        {
            observer.Notify(msg);
        }
    }
    public void DoSomething()
    {
        //…
        if(…)    //NO.6
        {
            NotifyObservers(…);
        }
    }
}
class MyObserver:IObserver
{
```

```
    public void Notify(string msg)
    {
        Console.WriteLine("receive msg :" + msg);    //NO.7
    }
}
class YourObserver:IObserver
{
    public void Notify(string msg)
    {
        //send email to others    NO.8
    }
}
```

如上代码 Code 11-1 中所示，NO.1 和 NO.2 处分别定义了 ISubject 和 IObserver 接口，接着定义了一个具体的主体类 MySubject，该类实现了 ISubject 接口，在 AddObserver、RemoveObserver 方法中分别将观察者加入或者移除集合_observers_list（NO.3 和 NO.4 处），最后在 NotifyObservers 方法中，遍历_observers_list 集合，将通知发送到每个观察者（NO.5 处），注意我们可以在 DoSomething 方法中判断当满足某一条件时，通知观察者（NO.6 处）。

我们使用 IObserver 接口定义了两个具体的观察者 MyObserver 和 YourObserver，在两者的 Notify 方法中分别按照自己的逻辑去处理通知信息（一个直接将 msg 打印出来，一个将 msg 以邮件形式发送给别人）（NO.7 和 NO.8 处）。

现在我们可以将 MySubject 类对象当作一个具体的主体，将 MyObserver 类对象和 YourObserver 类对象当作具体的观察者，那么代码中可以这样去使用：

```
//Code 11-2
ISubject subject = new MySubject();
subject.AddObserver(new MyObserver());    //NO.1
subject.AddObserver(new YourObserver());    //NO.2

subject.NotifyObservers("it's a test!");    //NO.3
(subject as MySubject).DoSomething();    //NO.4
```

如上代码 Code 11-2 所示，我们向主体 subject 中添加两个观察者（NO.1 和 NO.2 处），之后使用 ISubject.NotifyObservers 方法通知观察者（NO.3），另外，我们还可以使用 MySubject.DoSomething 方法去通知观察者（当某一条件满足时），两个观察者分别会做不同的处理，一个直接将“it's a test”字符串打印输出，而另一个则将字符串以邮件的形式发送给别人。

> 注：CODE 11-2 中，我们不能使用 ISUBJECT 接口去调用 DOSOMETHING 方法，而必须先将 ISUBJECT 类型引用转换成 MYSUBJECT 类型引用，因为 DOSOMETHING 不属于 ISUBJECT 接口。

观察者模式中，整个流程见下图 11-2：

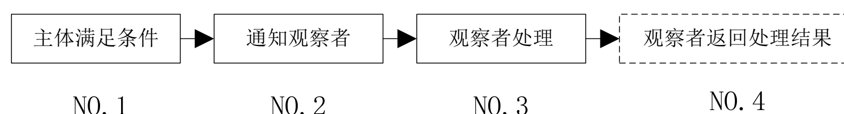


图 11-2 观察者模式中的运行流程

如上图 11-2 所示，在有些情况中，NO.2 处会做一些筛选，换句话说，主体有可能根据条件通知部分观察者，NO.4 处虚线框表示可选，如果主体关心观察者的处理结果，那么观察者就应该将自己的处理结果返回给主体。“观察者模式”是所有框架使用得最频繁的设计模式之一，原因很简单，“观察者模式”分隔开了框架代码和框架使用者编写的代码，它是“好莱坞原则”（Hollywood Principle，don't call us，we will call you）的具体实现手段，而“好莱坞原则”是所有框架都严格遵守的。

Windows Forms 框架中的“观察者模式”主要不是通过“接口-具体”这种方式去实现的，更多的是使用.NET 中的“委托-事件”去实现，详见下一小节。

11.1.2 Windows Forms 中的观察者模式

在 Windows Forms 框架中，可以说“观察者模式”无处不在，在第八章讲 Winform 程序结构时已经有所说明，比如控件处理 Windows 消息时，最终是以“事件”的形式去通知事件注册者的，那么这里的事件注册者就是观察者模式中的“观察者”，控件就是观察者模式中的“主体”。我们回忆一下第八章中有关 System.Windows.Forms.Control 类的代码（部分）：

```
//Code 11-3
class Control:Component
{
    //…
    public event EventHandler Event1;
    public event EventHandler Event2;
    protected virtual void WndProc(ref Message m)
    {
        switch(m.Msg)
        {
            case 1:    //NO.1
            {
                //…
                OnEvent1(…);
                break;
            }
            case 2:    //NO.2
            {
                OnEvent2(…);
                break;
            }
            //…
        }
    }
    protected virtual void OnEvent1(EventArgs e)
    {
        if(Event1 != null)
        {
            Event1(this,e); //NO.3
        }
    }
```

```
    protected virtual void OnEvent2(EventArgs e)
    {
        if(Event2 != null)
        {
            Event2(this,e);    //NO.4
        }
    }
}
```

如上代码 Code 11-3 所示，在 Control 类的 WndProc 窗口过程中的 switch/case 块中，会根据不同的 Windows 消息去激发不同的事件（NO.1 和 NO.2 处），由于 WndProc 是一个虚方法，所有在任何一个 Control 的派生类中，均可以重写 WndProc 虚方法，处理 Windows 消息，然后以“事件”的形式去通知事件注册者。

如果我们在 Form1 中注册了一个 Button 类对象 btn1 的 Click 事件，那么 btn1 就是观察者模式中的“主体”，Form1（的实例）就是观察者模式中的“观察者”，如下代码：

```
//Code 11-4
class Form1:Form
{
    //...
    public Form1()
    {
        InitializeComponent();
        btn1.Click += new EventHandler(btn1_Click);    //NO.1
    }
    private void btn1_Click(object sender,EventArgs e)    //NO.2
    {
        //...
    }
}
```

如上图 Code 11-4 代码所示，我们在 Form1 的构造方法中注册了 btn1 的 Click 事件（NO.1 处），那么 btn1 就是“主体”，Form1（的实例）就是“观察者”，当 btn1 需要处理 Windows 消息时，就会激发事件，通知 Form1（的实例）。

Windows Forms 框架正是使用“观察者模式”实现了框架代码与框架使用者编写代码之间的相互分离。

> 注：我们可以认为，事件的发布者等于观察者模式中的“主体”（SUBJECT），而事件的注册者等于观察者模式中的“观察者”，有关“事件编程”，请参考第五章。

11.1.3 设计模式分类

根据各种设计模式的作用，我们可以将常见的 23 种设计模式分三大类（参见《易学 设计模式》一书），见下表 11-1：

表 11-1 设计模式分类

序号	分类	说明	设计模式
1	创建型模式	对象的创建会消耗掉系统的很多资	工厂方法模式

		源，所以单独对对象的创建进行研究，从而能够高效地创建对象就是创建型模式要探讨的问题。	抽象工厂模式
			创建者模式
			原型模式
			单例模式
2	结构型模式	在解决了对象的创建问题之后，对象的组成以及对象之间的依赖关系就成了开发人员关注的焦点，因为如何设计对象的结构、继承和依赖关系会影响到后续程序的维护性、代码的健壮性、耦合性等。	外观模式
			适配器模式
			代理模式
			装饰模式
			桥接模式
			组合模式
			亨元模式
3	行为型模式	在对象的结构和对象的创建问题都解决了之后，就剩下对象的行为问题了，如果对象的行为设计的好，那么对象的行为就会更清晰，它们之间的协作效率就会提高。	模板方法模式
			观察者模式
			解释器模式
			状态模式
			责任链模式
			策略模式
			命令模式
			访问者模式
			调停者模式
			备忘录模式
			迭代器模式

11.2 软件的设计原则

11.2.1 Solid 原则介绍

“Solid 原则”代表软件设计过程中常见的五大原则，分别为：

（1）S：单一职责原则（Single Responsibility Principle）：

一个类应该只负责一个（种）事情；

（2）O：开闭原则（Open Closed Principle）：

优先选择在已有的类型基础上扩展新的类型，避免修改已有类型（已有代码）；

（3）L：里氏替换原则（Liskov Substitution Principle）：

任何基类出现的地方，派生类一定可以代替基类出现，言下之意就是，派生类一定要具备基类的所有特性；

（4）I：接口隔离原则（Interface Segregation Principle）：

一个类型不应该去实现它不需要的接口，换句话说，接口应该只包含同一类方法或属性等；

（5）D：依赖倒置原则（Dependency Inversion Principle）：

高层模块不应该依赖于低层模块，高层模块和低层模块应该同时依赖于一个抽象层（接口层）。

设计模式相对来讲更具体，每种设计模式几乎都能解决现实生活中某一具体问题，而设计原则相对来讲更抽象，它是我们在软件设计过程中的行为准则，并不能用在某一具体情景之

中。以上五大原则单从字面上理解起来不太直观，下面依次举例说明之。

11.2.2 单一职责原则（SRP）

“一个类应该只负责一个（种）事情”，原因很简单，负责的事情越多，那么这个类型出错或者需要修改的概率越大，假如现在有一个超市购物的会员类 VIP：

```
//Code 11-5
class VIP:IData
{
    public void Read()
    {
        try
        {
            //read db here…
        }
        catch(Exception ex)
        {
            System.IO.File.WriteAllText(@"c:\errorlog.txt", ex.ToString());    //NO.1
        }
    }
}
interface IData    //NO.2
{
    void Read();
}
```

如上代码 Code 11-5 所示，定义了一个访问数据库的 IData 接口（NO.2 处），该接口包含一个 Read 方法，用来读取会员信息，会员类 VIP 实现了 IData 接口，在编写 Read 方法时，我们捕获访问数据库的异常后，直接将错误信息写入到了日志文件（NO.1 处）。这段代码看似没有任何问题，但是后期确会暴露出设计不合理的现象，如果我们现在不想把日志文件输出到本地 C 盘（NO.1 处），而是输出到 D 盘，那我们需要修改 VIP 的源码，没错，本来我们只是想修改日志部分的逻辑，现在却不得不更改 VIP 类的代码。出现这种现象的原因就是 VIP 类干了本不应该它干的事情：记录日志。就像下面这张图描述的：

图 11-3 一个负责了太多事情的工具

如上图 11-3 所示，一把包含太多功能的刀，如果哪天某个功能坏掉，我们不得不将整把刀送去维修。正确解决以上问题的做法就是将日志逻辑与 VIP 类分开，代码如下：

```
//Code 11-6
class Logger    //NO.1
{
    public void WriteLog(string error)
    {
        System.IO.File.WriteAllText(@"c:\errorlog.txt", error);
    }
}
class VIP:IData
{
    private Logger _logger = new Logger();    //NO.2
    public void Read()
    {
        try
        {
            //read db here…
        }
        catch (Exception ex)
        {
            _logger.WriteLog(ex.ToString());    //NO.3
        }
    }
}
```

如上代码 Code 11-6 所示，我们定义了一个类型 Logger 专门负责记录日志（NO.1 处），在 VIP 类中通过 Logger 类型来记录错误信息（NO.2 和 NO.3 处），这样一来，当我们需要修改日志部分的逻辑时，不需要再动 VIP 类的代码。

单一职责原则提倡我们将复杂的功能拆分开来，分配到每个单独的类型当中，至于什么是复杂的功能，到底将功能拆分到什么程度，这个是没有标准的，如果记录日志是一个繁琐的过程（本小节示例代码相对简单），我们还可以将日志类 Logger 的功能再继续拆分。

11.2.3 开闭原则（OCP）

"优先选择在已有的类型基础上扩展新的类型，避免修改已有类型（已有代码）"，修改已有代码就意味着需要重新测试原有的功能，因为任何一次修改都可能影响已有功能。如果在普通 VIP 顾客的基础之上，多了白银会员（silver vip）顾客，这两种顾客在购物时的折扣不一样，如果 VIP 类定义如下（不全）：

```
//Code 11-7
class VIP:IData
{
    private int _viptype;    //vip type    NO.1
    //…
    public virtual void Read()
    {
```

```
        //…
    }
    public double GetDiscount(double totalSales)
    {
        if(_viptype == 1)    //vip
        {
            return totalSales – 10;    //NO.2
        }
        else    //silver vip
        {
            return totalSales – 50;    //NO.3
        }
    }
}
```

如上代码 Code 11-7 所示，我们在定义 VIP 类的时候，使用_viptype 字段来区分当前顾客是普通 VIP 还是白银 VIP（NO.1 处），在打折方法 GetDiscount 中，根据不同的 VIP 种类返回不同打折后的价格（NO.2 和 NO.3 处），这段代码的确也可以运行的很好，但是后期还是会暴露出设计不合理的地方，如果现在不止增加一个白银会员，还增加了一个黄金会员（gold vip），那么我们不得不再去修改 GetDiscount 方法中的 if/else 块，修改意味着原有功能可能会出现 bug，因此我们不得不再去测试之前所有使用到了 VIP 这个类型的代码。出现这个问题的主要原因就是我们从一开始设计 VIP 类的时候就不合理：没有考虑到将来可能会有普通会员的衍生体出现。

如果我们一开始在设计 VIP 类的时候就应用了面向对象思想，我们的 VIP 类可以这样定义：

```
//Code 11-8
interface IDiscount    //NO.1
{
    double GetDiscount(double totalSales);
}
class VIP:IData,IDiscount
{
    //…
    public virtual void Read()
    {
        //…
    }
    public virtual double GetDiscount(double totalSales)    //NO.2
    {
        return totalSales – 10;
    }
}
class SilverVIP:VIP
{
    //…
    public override double GetDiscount(double totalSales)
    {
        return totalSales – 50;    //NO.3
```

```
    }
}
class GoldVIP:SilverVIP
{
    //...
    public override double GetDiscount(double totalSales)
    {
        return totalSales – 100;    //NO.4
    }
}
```

如上代码 Code 11-8 所示，我们定义了一个 IDiscount 的接口（NO.1 处），包含一个打折的 GetDiscount 方法，接下来让 VIP 类实现了 IDiscount 接口，将接口中的 GetDiscount 方法定义为虚方法（NO.2 处），后面的白银会员（SilverVIP）继承自 VIP 类、黄金会员（GoldVIP）继承自 SilverVIP 类，并分别重写 GetDiscount 虚方法，返回相应的打折之后的总价格（NO.3 和 NO.4 处）。这样一来，新增加会员类型不需要去修改 VIP 类，也不影响之前使用了 VIP 类的代码。

下图 11-4 显示了重新设计 VIP 类的前后区别：

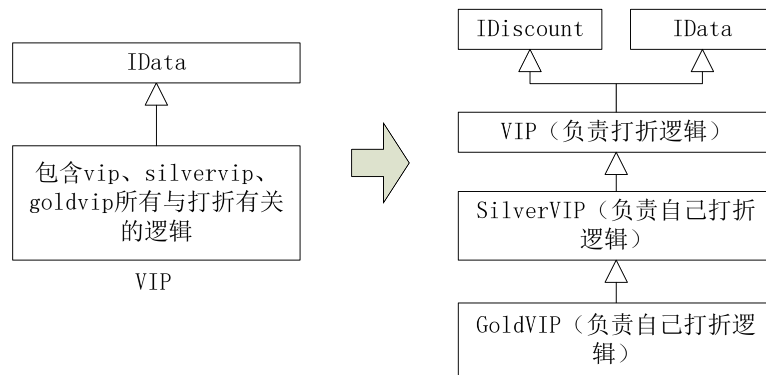


图 11-4 继承发生之后

如上图 11-4 所示，图中左边部分表示不采用继承的方式去实现普通 VIP、白银 VIP 和黄金 VIP 的打折逻辑，可以看出，每次需要增加一种会员时，都必须去修改 VIP 类的代码，图中右边部分表示采用继承方式之后，每种会员均定义成一个类型，每个类型均可以负责自己的打折逻辑，以后不管新增多少种会员，均可以定义新的派生类，在派生类中定义新的打折逻辑。

注：派生类中只需要重写打折的逻辑，不需要重新去定义读取数据库的逻辑，因为这个逻辑在基类和派生类中并没有发生变化。

11.2.4 里氏替换原则（LSP）

“任何基类出现的地方，派生类一定可以代替基类出现，言下之意就是，派生类一定要具备基类的所有特性”，意思就是说，如果 B 是 A 的儿子，那么 B 一定可以代替 A 去做任何事情，否则，B 就不应该是 A 的儿子。我们在设计类型的时候，往往不去注意一个类型是否真的应该去继承另外一个类型，很多时候我们只是为了遵从所谓的“OO”思想。如果现在有一个管理员类 Manager，因为管理员也需要读取数据库，所以我们让它继承自 VIP 类，代码如下：

```
//Code 11-9
class Manager:VIP
{
```

```
    //…
    public override void Read()
    {
        //…
    }
    public override double GetDiscount(double totalSales)
    {
        throw new Exception(“don't have this function!”);    //NO.1
    }
}
```

如上代码 Code 11-9 所示，我们定义 Manager 类，让其继承自 VIP 类，由于 Manager 类并没有“打折扣”的逻辑，因此我们重写 GetDiscount 方法时，抛出“don't have this function!”这样的异常（NO.1 处），接下来我们可能编写出如下这样的代码：

```
//Code 11-10
List<VIP> vips = new List<VIP>();    //NO.1
vips.Add(new VIP());
vips.Add(new SilverVIP());
vips.Add(new GoldVIP());
vips.Add(new Manager());
//…
foreach(VIP v in vips)
{
    /…
    double d = v.GetDiscount(…);    //NO.2
    //…
}
```

如上代码 Code 11-10 所示，我们定义了一个 VIP 类型的容器（NO.1 处），依次将 VIP、SilverVIP、GoldVIP 以及 Manager 类型对象加入容器，最后通过 foreach 遍历该容器，调用容器中每个元素的 GetDiscount 方法（NO.2 处），此段代码一切正常通过编译，因为编译器承认“基类出现的地方，派生类一定能够代替其出现”，但事实上，程序运行之后，在调用 Manager 类对象的 GetDiscount 虚方法时会抛出异常，造成这个现象的主要原因就是，我们根本没搞清楚类的继承关系，Manager 类虽然也要访问数据库，但是它并非属于 VIP 的一种，也就是说，Manager 类不应该是 VIP 类的儿子，如下图 11-5：

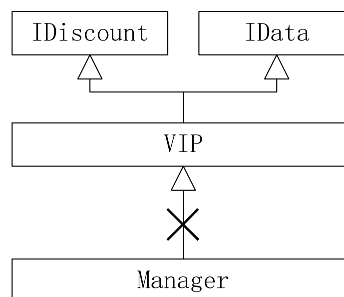


图 11-5 Manager 错误的继承关系

如上图 11-5 所示，Manager 类虽然需要读取数据库，但是它并不需要有与“折扣”相关的操作，而且它根本不属于一种 VIP 的衍生物，正确的做法是让 Manager 类直接实现 IData 接口即

可，如下代码：

```
//Code 11-11
class Manager:IData
{
    //...
    public void Read()
    {
        //...
    }
}
```

如上代码 Code 11-11 所示，Manager 实现了 IData 接口之后，不再跟 VIP 类有关联，这样一来，前面 Code 11-10 代码在编译时，就会通不过，

```
//Code 11-12
List<VIP> vips = new List<VIP>();   //NO.1
vips.Add(new VIP());
vips.Add(new SilverVIP());
vips.Add(new GoldVIP());
vips.Add(new Manager());   //NO.2
```

如上代码 Code 11-12 所示，编译器会在 NO.2 处报错，原因很简单，Manager 既然不是 VIP 的派生类，就不能代替 VIP 出现。

如果两个类从逻辑上就没有衍生的关系，就不应该有相互继承出现，见下图 11-6：

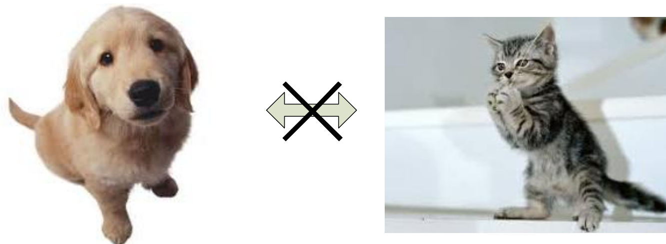


图 11-6 没有衍生关系的两个物体

如上图 11-6 所示，狗跟猫两种动物没有衍生关系，狗类（Dog）不能继承自猫类（Cat），猫类也不能继承自狗类，但是它们都可以同时继承自动物类（Animal）。

11.2.5 接口隔离原则（ISP）

“一个类型不应该去实现它不需要的接口，换句话说，接口应该只包含同一类方法或属性等”，如果把所有的方法都放在一个接口中，那么实现了该接口的类型必须实现接口中的全部方法（即使不需要），同理，在一个已经很稳定的系统中，不应该再去修改已经存在的接口，因为这会影响到之前所有实现该接口的类型。现在如果需要新增加一种 VIP 顾客（SuperVIP），允许它修改数据库，我们可能这样去修改 IData 接口：

```
//Code 11-13
interface IData
{
    void Read();
    void Write();   //NO.1
}
```

如上代码 Code 11-13 所示，我们修改已经存在的 IData 接口，使其包含一个写数据库的 Write 方法（NO.1 处），满足 SuperVIP 类的需要，这个方法看似可以，但是它要求我们修改其它已经实现了 IData 接口的类型，比如前面的 VIP 类，只要涉及到 VIP 类的更改，那么其它所有使用到了 VIP 类的地方都得重新测试，可以看出，这会影响整个已经存在的系统。正确的做法应该是，新增加一个接口 IData2，将数据库的写入方法放在该接口中，让 SuperVIP 类实现该接口，代码如下：

```
//Code 11-14
interface IData2:IData    //NO.1
{
    void Write();
}
class SuperVIP:IData2,IData
{
    public void Read()
    {
        //…
    }
    public void Write()
    {
        //…
    }
}
```

如上代码 Code 11-14 所示，我们定义了一个新的接口 IData2（NO.1 处），该接口包含一个 Write 方法，让 SuperVIP 类实现该接口，这样一来，整个过程不会影响已经存在的 VIP 类。

11.2.6 依赖倒置原则（DIP）

“高层模块不应该依赖于低层模块，高层模块和低层模块应该同时依赖于一个抽象层（接口层）”，本原则目的很明确，就是为了降低模块之间的耦合度，我们观察一下 11.2.2 小节示例代码中的 VIP 类和 Logger 类，很明显，VIP 类直接依赖于 Logger 类，如果我们想换种方式记录日志的话（也就是改变记录日志的逻辑），必须得重新修改 Logger 类中的已有代码，现在如果让 VIP 类依赖于一个抽象接口 ILog，其它所有记录日志的类型同时也依赖于 ILog 接口，那么整个系统就会更加灵活，

```
//Code 11-15
interface ILog    //NO.1
{
    void Log(string error);
}
class FileLogger:ILog    //NO.2
{
    public void Log(string error)
    {
        //write error log to local file
    }
}
```

```
class EmailLogger:ILog    //NO.3
{
        public void Log(string error)
        {
                //send error log as email
        }
}
class NotifyLogger:ILog    //NO.4
{
        public void Log(string error)
        {
                //notify other modules
        }
}
class VIP:IData,IDiscount    //NO.5
{
        //…
        ILog _logger;
        public VIP(ILog logger)    //NO.6
        {
                _logger = logger;
        }
        public virtual void Read()
        {
                try
                {
                        //…read db here
                }
                catch(Exception ex)
                {
                        _logger.Log(ex.ToString());    //NO.7
                }
        }
        public virtual double GetDiscount(double totalSales)
        {
                return totalSales – 10;
        }
}
```

如上代码 Code 11-15 所示，我们定义了一个日志接口 ILog 作为抽象层（NO.1 处），之后定义了各种各样的低层日志模块（NO.2. NO.3 和 NO.4 处），这些记录日志的类均依赖（实现）ILog 这个抽象接口，之后我们在定义 VIP 类时，不再让它依赖于某个具体的日志类，换句话说，不再让高层模块直接依赖低层模块，取而代之的是，让 VIP 类依赖于 ILog 这个抽象接口（NO.6 处），我们在使用 VIP 类的时候，可以根据需要给它传递不同的日志类对象（也可以是除了示例代码中的三个以外其它自定义类型，只要该类型实现了 ILog 接口），程序运行后，会将错误日志记录到相应位置（NO.7 处）。我们可以这样使用 VIP 类：

```
//Code 11-16
IData v = new VIP(new FileLogger());
v.Read();    //NO.1

IData v2 = new VIP(new EMailLogger());
v2.Read();    //NO.2

IData v3 = new VIP(new NotifyLogger());
v3.Read();    //NO.3
```

如上代码 Code 11-16 所示，NO.1 处如果出现异常，错误日志会保存到文件，NO.2 处如果出现异常，错误日志将会通过邮件发送给别人，NO.3 处如果出现异常，VIP 对象会自动把错误信息通知给别的模块。

依赖倒置原则提倡模块与模块之间不应该有直接的依赖关系，见下图 11-7：

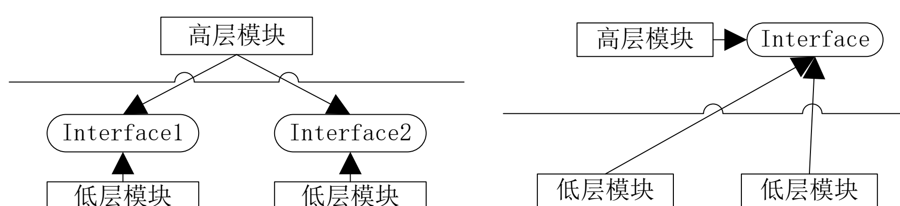


图 11-7 依赖倒置发生前后

如上图 11-7 所示，图中左边部分表示依赖倒置之前高层模块与低层模块之间的依赖关系，图中右边部分表示依赖倒置发生之后，高层模块与低层模块之间的依赖关系，很明显，依赖倒置发生后，高层模块不再直接受低层模块控制，高层模块与低层模块没有具体的对应关系，灵活性增加，耦合度降低。

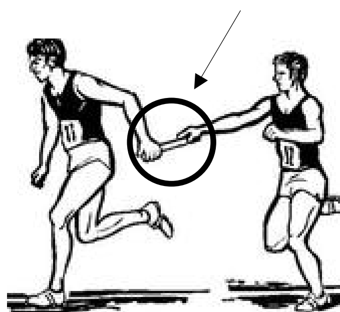


图 11-8 接力赛跑中的接力棒

如上图 11-8 所示，接力过程中前后两人没有具体对应关系。

> 注：依赖倒置原则是每个框架都必须遵循的，框架不可能受框架的使用者控制，换句话说，框架作为“高层模块”，不应该依赖于框架使用者编写的代码（低层模块），而应该均依赖于一个抽象层，所以我们在使用框架编写代码时，大部分时候均以框架库作为基础，从已有的类型或者接口（抽象层）派生出新的类型。

11.3 设计模式与设计原则对框架的意义

“IT 语境中的框架，特指为解决一个开放性问题而设计的具有一定约束性的支撑结构。在此结构上可以根据具体问题扩展、安插更多的组成部分，从而更迅速和方便地构建完整的解决问题的方案。”——摘自互联网

上面是一段摘自互联网上描述“框架”的话，从这段话中我们了解到，首先，每个框架解决问题的范围是有限的，比如 Windows Forms 框架只会帮助我们完成 Windows 桌面应用程序的开发，这就是它的“约束性”；其次，框架本身解决不了什么特定的问题，它只给了解决特定问题的相关模块（或者组件）一个可插接、可组合的底子，这个底子为我们解决实际具体问题提供了支持，这就是框架的“支撑性”，见下图 11-9：

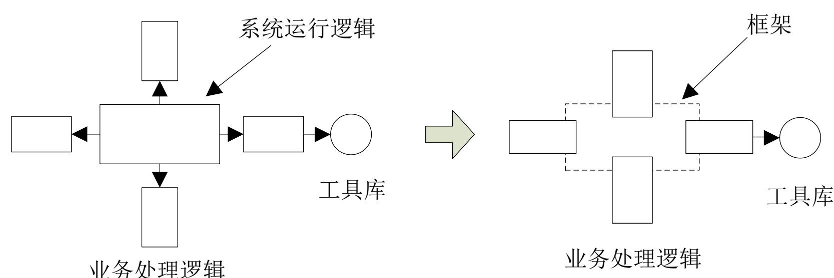


图 11-9 框架使用前后

如上图 11-9 所示，图中左边部分表示使用框架之前，整个系统均由开发者编写代码的结构图，我们可以看见，无论系统的“系统运行逻辑”还是“业务处理逻辑”均由开发者负责，开发者自己调用自己的代码，整个系统的运行流程由开发者控制；图中右边部分表示使用了框架之后，“系统运行逻辑”由框架接管，开发者只需要把精力集中在“业务处理逻辑”之上，除此之外，还有一个非常大而且非常重要的改变：开发者不再（几乎不）自己调用自己的代码，自己编写的代码均由框架调用，系统运行的控制权交给了框架。这就是所有框架所必须满足的“好莱坞原则”（Hollywood Principle，don't call us,we will call you），“好莱坞原则”跟“控制转换原则”（IoC，Inversion of Control）类似，参见前面章节，便可知道框架是怎样反过来控制程序的运行。

我们在使用框架开发应用程序去解决实际具体的问题时，框架避免不了会与我们开发者编写的代码进行交互，这就会产生一个问题，那就是怎样去把握框架代码和框架使用者编写代码两者之间的关联性，也就是我们常说的“高内聚，低耦合”。“高内聚，低耦合”在框架中要求更高，因为框架的使用人群和范围比一般普通系统更大更广泛，优秀的框架要想使用寿命更长口碑更好，就要求框架能在使用后期能够更容易升级、更方便扩展新的功能来满足使用者的各种需要，而这些大部分取决于框架最开始的设计好坏，正确地使用各种“设计模式”以及严格地遵守各种“设计原则”是决定框架后期能否应付各种变更、频繁升级的重要因素。

11.4 本章回顾

正确地运用软件设计模式、严格地遵循软件设计原则能够使开发出来的软件系统更具备扩展性，后期软件系统维护更容易，源代码更易重构。本章开头介绍了比较常见的一种设计模式：观察者模式，.NET 中任何使用到“事件编程”的地方均属于观察者模式，由于软件设计模式种类数量之多，加之又博大精深，受篇幅影响，所以本章开头只是起到一个抛砖引玉的作用，

要想更完整地去学习软件设计模式，读者可以去参阅专门讲解设计模式的书籍；本章随后着重讲述了五大软件设计原则，以首字母组合而成的 Solid 原则，分别为：单一职责原则、开闭原则、里氏替换原则、接口隔离原则以及依赖倒置原则；本章最后提到了软件设计模式和软件设计原则对框架的意义，并指出，任何一个优秀的框架都正确地使用了相应的软件设计模式以及严格地遵循了各种软件设计原则。

11.5 本章思考

1.简述“软件设计模式”与“软件设计原则”的区别。

A：虽然两者都是前人通过大量的实践总结出来、有利于软件系统开发的一些经验，但是”设计模式”更具体，每个设计模式的存在都是为了解决某一个（或某一类）问题，而”设计原则”相比起来更抽象，它只是一个理论思想，并不能应用到某个具体的场景之中，去解决某一个具体的问题。”设计模式”与”设计原则”在软件系统设计过程中均起到了重要作用。

2.“Solid 原则”包含哪五大软件设计原则？英文全称分别是？

A：

（1）单一职责原则（Singleton Responsibility Principle）；

（2）开闭原则（Open Closed Principle）；

（3）里氏替换原则（Liskov Substitution Principle）；

（4）接口隔离原则（Interface Segregation Principle）；

（5）依赖倒置原则（Dependency Inversion Principle）。

详见本章 11.2.1 小节。笔者认为熟记编程中常见的英文术语有利于对源概念的理解。

3.哪些设计原则的主要目的是为了降低代码之间的依赖程度？

A：单一职责原则和依赖倒置原则，前者建议每个类型只应该负责某一个（或某一类）任务，这样与外界发生关联关系的可能性较小；后者建议对象之间发生不可避免的依赖关系时，所有的依赖均应该建立在”抽象”之上，而不应该是”具体”，由于抽象的事物可变，对象之间不会产生绑定的不可变的依赖关系。

第十二章 难免的尴尬：代码依赖

在浩瀚的代码世界中，有着无数的对象，跟人和人之间有社交关系一样，对象跟对象之间也避免不了接触，所谓接触，就是指一个对象要使用到另外对象的属性、方法等成员。现实生活中一个人的社交关系复杂可能并不是什么不好的事情，然而对于代码中的对象而言，复杂的“社交关系”往往是不提倡的，因为对象之间的关联性越大，意味着代码改动一处，影响的范围就会越大，而这完全不利于系统重构和后期维护。所以在现代软件开发过程中，我们应该遵循“尽量降低代码依赖”的原则，所谓尽量，就已经说明代码依赖不可避免。

有时候一味地追求“降低代码依赖”反而会使系统更加复杂，我们必须在“降低代码依赖”和“增加系统设计复杂性”之间找到一个平衡点，而不应该去盲目追求“六人定理”那种设计境界。

> 注：“六人定理”指：任何两个人之间的关系带，基本确定在六个人左右。两个陌生人之间，可以通过六个人来建立联系，此为六人定律，也称作六人法则。

12.1 从面向对象开始

在计算机科技发展历史中，编程的方式一直都是趋向于简单化、人性化，“面向对象编程”正是历史发展某一阶段的产物，它的出现不仅是为了提高软件开发的效率，还符合人们对代码世界和真实世界的统一认识观。当说到“面向对象”，出现在我们脑海中的词无非是：类，抽闲，封装，继承以及多态，本节将从对象基础、对象扩展以及对象行为三个方面对“面向对象”做出解释。

> 注：面向对象中的“面向”二字意指：在代码世界中，我们应该将任何东西都看做成一个封闭的单元，这个单元就是“对象”。对象不仅仅可以代表一个可以看得见摸得着的物体，它还可以代表一个抽象过程，从理论上讲，任何具体的、抽象的事物都可以定义成一个对象。

12.1.1 对象基础：封装

和现实世界一样，无论从微观上还是宏观上看，这个世界均是由许许多多的单个独立物体组成，小到人、器官、细胞，大到国家、星球、宇宙， 每个独立单元都有自己的属性和行为。仿照现实世界，我们将代码中有关联性的数据与操作合并起来形成一个整体，之后在代码中数据和操作均是以一个整体出现，这个过程称为“封装”。封装是面向对象的基础，有了封装，才会有整体的概念。

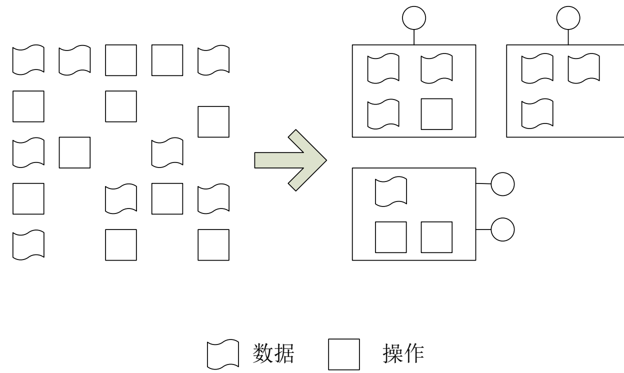


图 12-1 封装前后

如上图 12-1 所示，图中左边部分为封装之前，数据和操作数据的方法没有相互对应关系，方法可以访问到任何一个数据，每个数据没有访问限制，显得杂乱无章；图中右边部分为封装之后，数据与之关联的方法形成了一个整体单元，我们称为“对象”，对象中的方法操作同一对象的数据，数据之间有了“保护”边界。外界可以通过对象暴露在外的接口访问对象，比如给它发送消息。

通常情况下，用于保存对象数据的有字段和属性，字段一般设为私有访问权限，只准对象内部的方法访问，而属性一般设为公开访问权限，供外界访问。方法就是对象的表现行为，分为私有访问权限和公开访问权限两类，前者只准对象内部访问，而后者允许外界访问。

```
//Code 12-1
class Student    //NO.1
{
    private string _name;    //NO.2
    private int _age;
    private string _hobby;
    public string Name      //NO.3
    {
        get
        {
            return _name;
        }
    }
    public int Age
    {
        get
        {
            return _age;
        }
        set
        {
            if(value<=0)
            {
                value=1;
            }
            _age = value;
```

```
        }
    }
    public string Hobby
    {
        get
        {
            return _hobby;
        }
        set
        {
            _hobby = value;
        }
    }
    public Student(string name,int age,string hobby)
    {
        _name = name;
        _age = age;
        _hobby = hobby;
    }
    public void SayHello()    //NO.4
    {
        Console.WriteLine(GetSayHelloWords());
    }
    protected virtual string GetSayHelloWords()    //NO.5
    {
        string s = “”;
        s += ”hello,my name is ” + _name + “,\r\n”,
        s += ”I am ”+_age + “years old,” + “\r\n”;
        s += ”I like ”+_hobby + “,thanks\r\n”;
        return s;
    }
}
```

上面代码 Code 12-1 将学生这个人群定义成了一个 Student 类（NO.1 处），它包含三个字段：分别为保存姓名的_name、保存年龄的_age 以及保存爱好的_hobby 字段，这三个字段都是私有访问权限，为了方便外界访问内部的数据，又分别定义了三个属性：分别为访问姓名的 Name，注意该属性是只读的，因为正常情况下姓名不能再被外界改变；访问年龄的 Age，注意当给年龄赋值小于等于 0 时，代码自动将其设置为 1；访问爱好的 Hobby，外界可以通过该属性对_hobby 字段进行完全访问。同时 Student 类包含两个方法，一个公开的 SyaHello()方法和一个受保护的 GetSayHelloWords()方法，前者负责输出对象自己的“介绍信息”，后者负责格式化“介绍信息”的字符串。Student 类图见图 12-2：

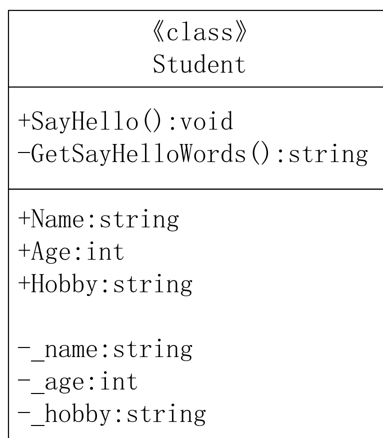


图 12-2 Student 类图

注：上文中将类的成员访问权限只分为两个部分，一个对外界可见，包括 PUBLIC；另一种对外界不可见，包括 PRIVATE、PROTECTED 等。

注意类与对象的区别，如果说对象是代码世界对现实世界中各种事物的一一映射，那么类就是这些映射的模板，通过模板创建具体的映射实例：

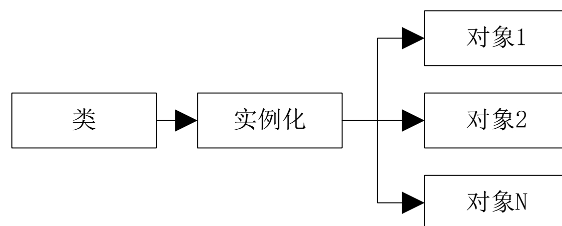


图 12-3 对象实例化

我们可以看到代码 Code 12-1 中的 Student 类既包含私有成员也包含公开成员，私有成员对外界不可见，外界如需访问对象，只能调用给出的公开方法。这样做的目的就是将外界不必要了解的信息隐藏起来，对外只提供简单的、易懂的、稳定的公开接口即可方便外界对该类型的使用，同时也避免了外界对对象内部数据不必要的修改和访问所造成的异常。

封装的准则：

封装是面向对象的第一步，有了封装，才会有类、对象，再才能谈继承、多态等。经过前人丰富的实践和总结，对封装有以下准则，我们在平时实际开发中应该尽量遵循这些准则：

1）一个类型应该尽可能少地暴露自己的内部信息，将细节的部分隐藏起来，只对外公开必要的稳定的接口；同理，一个类型应该尽可能少地了解其它类型，这就是常说的“迪米特法则（Law of Demeter）”，迪米特法则又被称作“最小知识原则”，它强调一个类型应该尽可能少地知道其它类型的内部实现，它是降低代码依赖的一个重要指导思想，详见本章后续介绍；

2）理论上，一个类型的内部代码可以任意改变，而不应该影响对外公开的接口。这就要求我们将“善变”的部分隐藏到类型内部，对外公开的一定是相对稳定的；

3）封装并不单指代码层面上，如类型中的字段、属性以及方法等，更多的时候，我们可以将其应用到系统结构层面上，一个模块乃至系统，也应该只对外提供稳定的、易用的接口，而将具体实现细节隐藏在系统内部。

封装的意义：

封装不仅能够方便对代码对数据的统一管理，它还有以下意义：

1）封装隐藏了类型的具体实现细节，保证了代码安全性和稳定性；
2）封装对外界只提供稳定的、易用的接口，外部使用者不需要过多地了解代码实现原理也不需要掌握复杂难懂的调用逻辑，就能够很好地使用类型；
3）封装保证了代码模块化，提高了代码复用率并确保了系统功能的分离。

12.1.2 对象扩展：继承

封装强调代码合并，封装的结果就是创建一个个独立的包装件：类。那么我们有没有其它的方法去创建新的包装件呢？

在现实生活中，一种物体往往衍生自另外一种物体，所谓衍生，是指衍生体在具备被衍生体的属性基础上，还具备其它额外的特性，被衍生体往往更抽象，而衍生体则更具体，如大学衍生自学校，因为大学具备学校的特点，但大学又比学校具体，人衍生自生物，因为人具备生物的特点，但人又比生物具体。

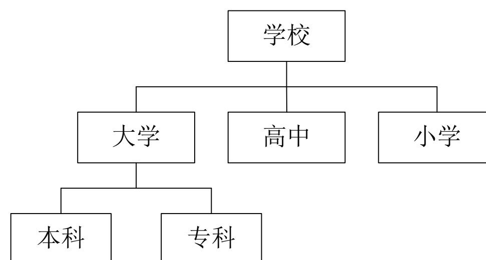


图 12-4 学校衍生图

如上图 12-4，学校相对来讲最抽象，大学、高中以及小学均可以衍生自学校，进一步来看，大学其实也比较抽象，因为大学还可以有具体的本科、专科，因此本科和专科可以衍生自大学，当然，抽象和具体的概念是相对的，如果你觉得本科还不够具体，那么它可以再衍生出来一本、二本以及三本。

在代码世界中，也存在“衍生”这一说，从一个较抽象的类型衍生出一个较具体的类型，我们称“后者派生自前者”，如果 A 类型派生自 B 类型，那么称这个过程为“继承”，A 称之为“派生类”，B 则称之为“基类”。

> 注：派生类又被形象地称为“子类”，基类又被形象地称为“父类”。

在代码 12-1 中的 Student 类基础上，如果我们需要创建一个大学生（College_Student）的类型，那么我们完全可以从 Student 类派生出一个新的大学生类，因为大学生具备学生的特点，但又比学生更具体：

```
//Code 12-2
class College_Student:Student    //NO.1
{
    private string _major;
    public string Major
    {
        get
        {
            return _major;
        }
        set
```

```
            {
                _major = value;
            }
        }
        public College_Student(string name,int age,string hobby,string major)
            :base(name,age,hobby)    //NO.2
        {
            _major = major;
        }
        protected override string GetSayHelloWords()    //NO.3
        {
            string s = "";
            s += "hello,my name is " + Name + ",\r\n",
            s += "I am "+ Age + "years old, and my major is " + _major + ",\r\n";
            s += "I like "+ Hobby + ", thanks\r\n";
            return s;
        }
}
```

如上代码 Code 12-2 所示，College_Student 类继承 Student 类（NO.1 处），College_Student 类具备 Student 类的属性，比如 Name、Age 以及 Hobby，同时 College_Student 类还增加了额外的专业（Major）属性，通过在派生类中重写 GetSyaHelloWords()方法，我们重新格式化“个人信息”字符串，让其包含“专业”的信息（NO.3 处），最后，调用 College_Student 中从基类继承下来的 SayHello()方法，便可以轻松输出自己的个人信息。

我们看到，派生类通过继承获得了基类的全部信息，之外，派生类还可以增加新的内容（如 College_Student 类中新增的 Major 属性），基类到派生类是一个抽象到具体的过程，因此，我们在设计类型的时候，经常将通用部分提取出来，形成一个基类，以后所有与基类有种族关系的类型均可以继承该基类，以基类为基础，增加自己特有的属性。

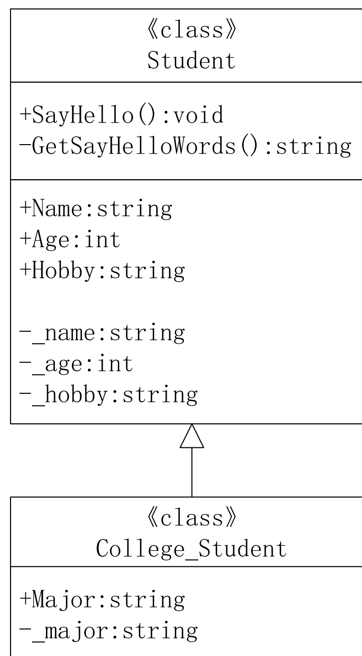


图 12-5 College_Student 类继承图

有的时候，一种类型只用于其它类型派生，从来不需要创建它的某个具体对象实例，这样的类高度抽象化，我们称这种类为“抽象类”，抽象类不负责创建具体的对象实例，它包含了派生类型的共同成分。除了通过继承某个类型来创建新的类型，.NET 中还提供另外一种类似的创建新类型的方式：接口实现。接口定义了一组方法，所有实现了该接口的类型必须实现接口中所有的方法：

```
//Code 12-3
interface IWalkable
{
    void Walk();
}
class People:IWalkable
{
    //…
    public void Walk()
    {
        Console.WriteLine("walk quickly");
    }
}
class Dog:IWalkable
{
    //…
    public void Walk()
    {
        Console.WriteLine("walk slowly");
    }
}
```

如上代码 Code 12-3 所示，People 和 Dog 类型均实现了 IWalkable 接口，那么它们必须都实现 IWalkable 接口中的 Walk()方法，见下图 12-6：

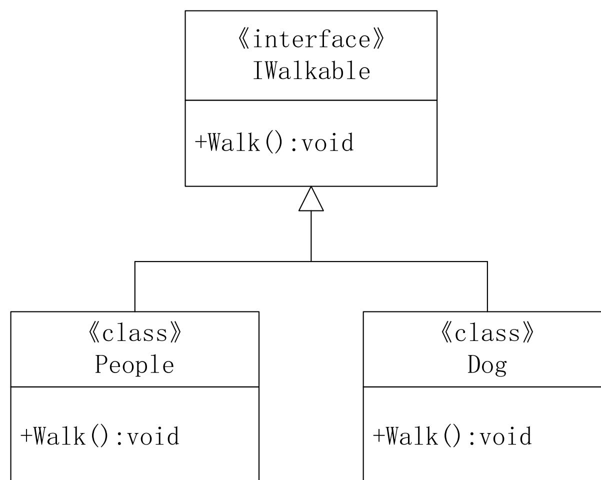


图 12-6 接口继承

继承包括两种方式，一种为“类继承”，一种为“接口继承”，它们的作用类似，都是在现有类型基础上创建出新的类型，但是它们也有区别：

1）类继承强调了族群关系，而接口继承强调通用功能。类继承中的基类和派生类属于祖宗和子孙的关系，而接口继承中的接口和实现了接口的类型并没有这种关系。

2）类继承强调“我是（Is-A）”的关系，派生类“是”基类（注意这里的“是”代表派生类具备基类的特性），而接口继承强调“我能做（Can-Do）”的关系，实现了接口的类型具有接口中规定的行为能力（因此接口在命名时均以“able”作为后缀）。
3）类继承中，基类虽然较抽象，但是它可以有具体的实现，比如方法、属性的实现，而接口继承中，接口不允许有任何的具体实现。

继承的准则：

继承是面向对象编程中创建类型的一种方式，在封装的基础上，它能够减少工作量、提高代码复用率的同时，快速地创建出具有相似性的类型。在使用继承时，请遵循以下准则：
1）严格遵守“里氏替换原则”，即基类出现的地方，派生类一定可以出现，因此，不要盲目地去使用继承，如果两个类没有衍生的关系，那么就不应该有继承关系。如果让猫（Cat）类派生自狗（Dog）类，那么很容易就可以看到，狗类出现的地方，猫类不一定可以代替它出现，因为它两根本就没有抽象和具体的层次关系。
2）由于派生类会继承基类的全部内容，所以要严格控制好类型的继承层次，不然派生类的体积会越来越大。另外，基类的修改必然会影响到派生类，继承层次太多不易管理，继承是增加耦合的最重要因素。
3）继承强调类型之间的通性，而非特性。因此我们一般将类型都具有的部分提取出来，形成一个基类（抽象类）或者接口。

12.1.3 对象行为：多态

“多态”一词来源于生物学，本意是指地球上的所有生物体现出形态和状态的多样性。在面向对象编程中多态是指：同一操作作用于不同类的实例，将产生不同的执行结果，即不同类的对象收到相同的消息时，得到不同的结果。

多态强调面向对象编程中，对象的多种表现行为，见下代码 Code 12-4：

```
//Code 12-4
class Student      //NO.1
{
    public void IntroduceMyself()
    {
        SayHello();
    }
    protected virtual void SayHello()
    {
        Console.WriteLine(“Hello,everyone!”);
    }
}
class College_Student:Student     //NO.2
{
    protected override void SayHello()
    {
        base.SayHello();
        Console.WriteLine(“I am a college student…”);
    }
}
```

```
class Senior_HighSchool_Student:Student    //NO.3
{
    protected override void SayHello()
    {
        base.SayHello();
        Console.WriteLine("I am a senior high school student…");
    }
}
class Program
{
    static void Main()
    {
        Console.Title = "SayHello";
        Student student = new Student();
        student.IntroduceMyself();    //NO.4
        student = new College_Student();
        student.IntroduceMyself();    //NO.5
        student = new Senior_HighSchool_Student();
        student.IntroduceMyself();    //NO.6
        Console.Read();
    }
}
```

如上代码 Code 12-4 所示，分别定义了三个类：Student（NO.1 处）、College_Student（NO.2 处）、Senior_HighSchool_Student（NO.3 处），后面两个类继承自 Student 类，并重写了 SayHello()方法。在客户端代码中，对于同一行代码"student.IntroduceMyself();"而言，三次调用（NO.4、NO.5 以及 NO.6 处），屏幕输出的结果却不相同：

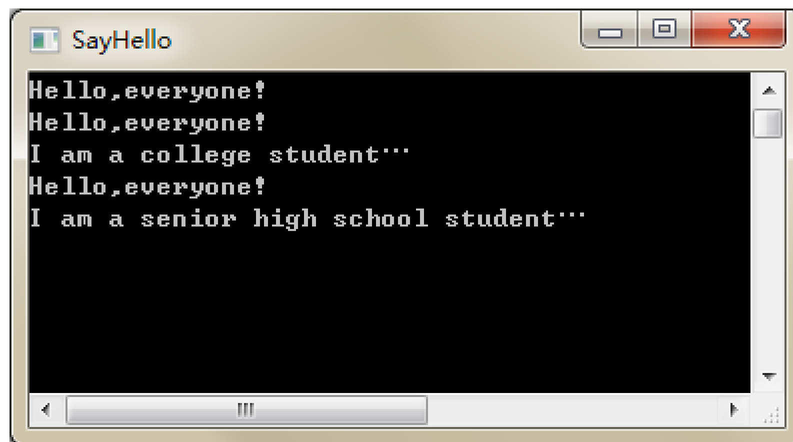


图 12-7 多态效果

如上图 12-7 所示，三次调用同一个方法，不同对象有不同的表现行为，我们称之为"对象的多态性"。从代码 Code 12-4 中可以看出，之所以出现同样的调用会产生不同的表现行为，是因为给基类引用 student 赋值了不同的派生类对象，并且派生类中重写了 SayHello()虚方法。

对象的多态性是以"继承"为前提的，而继承又分为"类继承"和"接口继承"两类，那么多态性也有两种形式：

1）类继承式多态；

类继承式多态需要虚方法的参与，正如代码 Code 12-4 中那样，派生类在必要时，必须重

写基类的虚方法，最后使用基类引用调用各种派生类对象的方法，达到多种表现行为的效果：

2）接口继承式多态。

接口继承式多态不需要虚方法的参与，在代码 Code 12-3 的基础上编写如下代码：

```
//Code 12-5
class Program
{
    static void Main()
    {
        Console.Title = "Walk";
        IWalkable iw = new People();
        iw.Walk();    //NO.1
        iw = new Dog();
        iw.Walk();    //NO.2
        Console.Read();
    }
}
```

如上代码 Code 12-5 所示，对于同一行代码"iw.Walk();"的两次调用（NO.1 和 NO.2 处），有不同的表现行为：

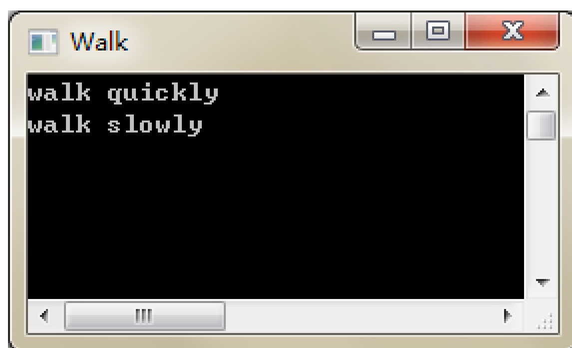


图 12-8 接口继承式多态

在面向对象编程中，多态的前提是继承，而继承的前提是封装，三者缺一不可。多态也是是降低代码依赖的有力保障，详见本章后续有关内容。

12.2 不可避免的代码依赖

本书前面章节曾介绍过，程序的执行过程就是方法的调用过程，有方法调用，必然会促使对象跟对象之间产生依赖，除非一个对象不参与程序的运行，这样的对象就像一座孤岛，与其它对象没有任何交互，但是这样的对象也就没有任何存在价值。因此，在我们的程序代码中，任何一个对象必然会与其它一个甚至更多个对象产生依赖关系。

12.2.1 依赖存在的原因

"方法调用"是最常见产生依赖的原因，一个对象与其它对象必然会通信（除非我们把所有的代码逻辑全部写在了这个对象内部），通信通常情况下就意味着有方法的调用，有方法的

调用就意味着这两个对象之间存在依赖关系（至少要有其它对象的引用才能调用方法），另外常见的一种产生依赖的原因是：继承，没错，继承虽然给我们带来了非常大的好处，却也给我们带来了代码依赖。依赖产生的原因大概可以分以下四类：

1）继承；

派生类继承自基类，获得了基类的全部内容，但同时，派生类也受控于基类，只要基类发生改变，派生类一定发生变化：

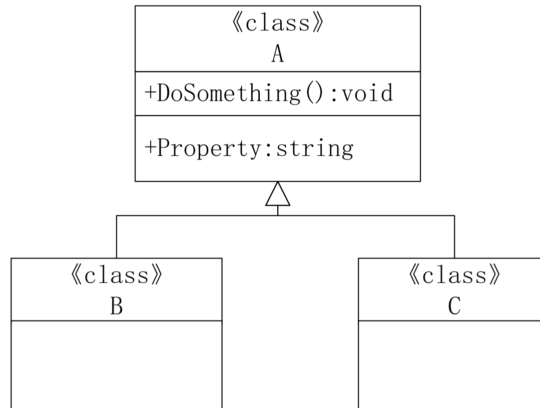


图 12-9 继承依赖

上图 12-9 中，B 和 C 继承自 A，A 类改变必然会影响 B 和 C 的变化。

2）成员对象；

一个类型包含另外一个类型的成员时，前者必然受控于后者，虽然后者的改变不一定会影响到前者：

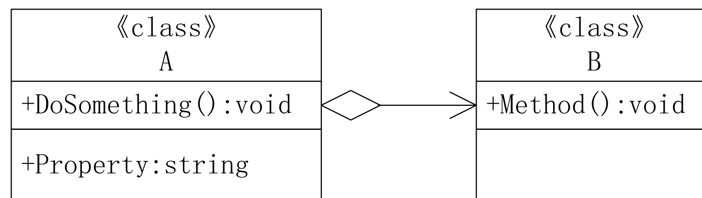


图 12-10 成员对象依赖

如上图 12-10，A 包含 B 类型的成员，那么 A 就受控于 B，B 在 A 内部完全可见。

> 注：成员对象依赖跟组合（聚合）类似。

3）传递参数；

一个类型作为参数传递给另外一个类型的成员方法，那么后者必然会受控于前者，虽然前者的改变不一定会影响到后者：

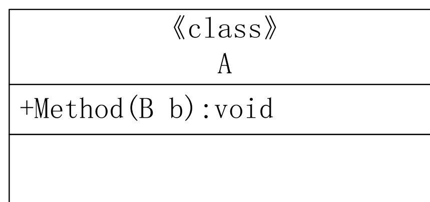


图 12-11 传参依赖

如上图 12-11，A 类型的方法 Method()包含一个 B 类型的参数，那么 A 就受控于 B，B 在 A 的 Method()方法可见。

4）临时变量。

任何时候，一个类型将另外一个类型用作了临时变量时，那么前者就受控于后者，虽然后者的改变不一定会影响到前者：

```
//Code 12-6
class A
{
    public void DoSomething()
    {
        //…
    }
}
class B
{
    public void DoSomething()
    {
        //…
        A a = new A();
        a.DoSomething();
        //…
    }
}
```

如上代码Code 12-6，B的DoSomething()方法中使用了A类型的临时对象，A在B的DoSomething()方法中局部范围可见。

通常情况下，通过被依赖者在依赖者内部可见范围大小来衡量依赖程度的高低，原因很简单，可见范围越大，说明访问它的概率就越大，依赖者受影响的概率也就越大，因此，上述四种依赖产生的原因中，依赖程度按顺序依次降低。

12.2.2 耦合与内聚

为了衡量对象之间依赖程度的高低，我们引进了“耦合”这一概念，耦合度越高，说明对象之间的依赖程度越高；为了衡量对象独立性的高低，我们引进了“内聚”这一概念，内聚性越高，说明对象与外界交互越少、独立性越强。很明显，耦合与内聚是两个相互对立又密切相关的概念。

> 注：从广义上讲，“耦合”与“内聚”不仅适合对象与对象之间的关系，也适合模块与模块、系统与系统之间的关系，这跟前面讲“封装”时强调“封装”不仅仅指代码层面上的道理一样。

“模块功能集中，模块之间界限明确”一直是软件设计追求的目标，软件系统不会因为需求的改变、功能的升级而不得不大范围修改原来已有的源代码，换句话说，我们在软件设计中，应该严格遵循“高内聚、低耦合”的原则。下图12-12显示一个系统遵循该原则前后：

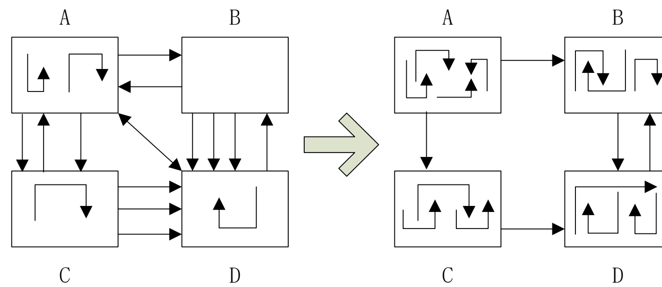


图 12-12 高内聚、低耦合

如上图 12-12 所示，“高内聚、低耦合”强调对象与对象之间（模块与模块之间）尽可能多地降低依赖程度，每个对象（或模块，下同）尽可能提高自己的独立性，这就要求它们各自负责的功能相对集中，代码结构由“开放”转向“收敛”。

“职责单一原则（SRP）”是提高对象内聚性的理论指导思想之一，它建议每个对象只负责某一个（一类）功能。

12.2.3 依赖造成的“尴尬”

如果在软件系统设计初期，没有合理地降低（甚至避免）代码间的耦合，系统开发后期往往会遇到前期不可预料的困难。下面举例说明依赖给我们造成的“尴尬”。

假设一个将要开发的系统中使用到了数据库，系统设计阶段确定使用 SQL Server 数据库，按照“代码模块化可以提高代码复用性”的原则，我们将访问 SQL Server 数据库的代码封装成了一个单独的类，该类只负责访问 SQLServer 数据库这一功能：

```
//Code 12-7
class SQLServerHelper    //NO.1
{
    //…
    public void ExcuteSQL(string sql)
    {
        //…
    }
}
class DBManager    //NO.2
{
    //…
    SQLServerHelper _sqlServerHelper;    //NO.3
    public DBManager(SQLServerHelper sqlServerHelper)
    {
        _sqlServerHelper = sqlServerHelper;
    }
    public void Add()    //NO.4
    {
        string sql = “”;
        //…
        _sqlServerHelper.ExcuteSQL(sql);
    }
```

```
    public void Delete()    //NO.5
    {
        string sql = ””;
        //…
        _sqlServerHelper.ExcuteSQL(sql);
    }
    public void Update()    //NO.6
    {
        string sql = ””;
        //…
        _sqlServerHelper.ExcuteSQL(sql);
    }
    public void Search()    //NO.7
    {
        string sql = “”;
        //…
        _sqlServerHelper.ExcuteSQL(sql);
    }
}
```

如上代码 Code 12-7 所示，定义了一个 SQL Server 数据库访问类 SQLServerHelper（NO.1 处），该类专门负责访问 SQL Server 数据库，如执行 sql 语句（其它功能略），然后定义了一个数据库管理类 DBManager（NO.2 处），该类负责一些数据的增删改查（NO.4、NO.5、NO.6 以及 NO.7 处），同时该类还包含一个 SQLServerHelper 类型成员（NO.3 处），负责具体 SQL Server 数据库的访问。SQLServerHelper 类和 DBManager 类的关系见下图 12-13：

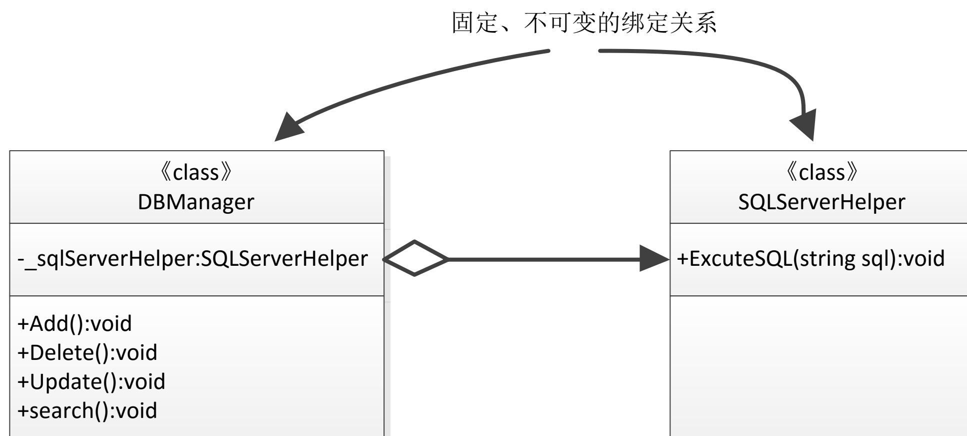


图 12-13 依赖于具体

如上图 12-13 所示，DBManager 类依赖于 SQLServerHelper 类，后者在前者内部完全可见，当 DBManager 需要访问 SQL Server 数据库时，可以交给 SQLServerHelper 类型成员负责，到此为止，这两个类型合作得非常好，但是，现在如果我们对数据库的需求发生变化，不再使用 SQL Server 数据库，而要求更改使用 MySQL 数据库，那么我们需要做些什么工作呢？和之前一样，我们需要定义一个 MySQLHelper 类来负责 MySQL 数据库的访问，代码如下：

```
//Code 12-8
class MySQLHelper
{
```

```
    //...
    public void ExcuteSQL(string sql)
    {
        //...
    }
}
```

如上代码 Code 12-8，定义了一个专门访问 MySQL 数据库的类型 MySQLHelper，它的结构跟 SQLServerHelper 相同，接下来，为了使原来已经工作正常的系统重新适应于 MySQL 数据库，我们还必须依次修改 DBManager 类中所有对 SQLServerHelper 类型的引用，将其全部更新为 MySQLHelper 的引用。如果只是一个 DBManager 类使用到了 SQLServerHelper 的话，整个更新工作量还不算非常多，但如果程序代码中还有其它地方使用到了 SQLServerHelper 类型的话，这个工作量就不可估量，除此之外，我们这样做出的所有操作完全违背了软件设计中的“开闭原则（OCP）”，即“对扩展开放，而对修改关闭”。很明显，我们在增加新的类型 MySQLHelper 时，还修改了系统原有代码。

出现以上所说问题的主要原因是，在系统设计初期，DBManager 这个类型依赖了一个具体类型 SQLServerHelper，“具体”就意味着不可改变，同时也就说明两个类型之间的依赖关系已经到达了“非你不可”的程度。要解决以上问题，需要我们在软件设计初期就做出一定的措施，详见下一小节。

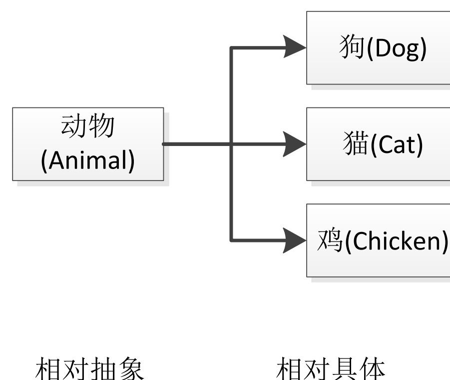
12.3 降低代码依赖

上一节末尾说到了代码依赖给我们工作带来的麻烦，还提到了主要原因是对象与对象之间（模块与模块，下同）依赖关系太过紧密，本节主要说明怎样去降低代码间的依赖程度。

12.3.1 认识“抽象”与“具体”

其实本书之前好些地方已经出现过“具体”和“抽象”的词眼，如“具体的类型”、“依赖于抽象而非具体”等等，到目前为止，本书还并没有系统地介绍这两者的具体含义。

所谓“抽象”，即“不明确、未知、可改变”的意思，而“具体”则是相反的含义，它表示“确定、不可改变”。我们在前面讲“继承”时就说过，派生类继承自基类，就是一个“抽象到具体”的过程，比如基类“动物（Animal）”就是一个抽象的事物，而从基类“动物（Animal）”派生出来的“狗（Dog）”就是一个具体的事物。抽象与具体的关系如下图 12-14：



相对抽象　　　　相对具体

图 12-14 抽象与具体的相对性

注：抽象与具体也是一个相对的概念，并不能说"动物"就一定是一个抽象的事物，它与"生物"进行比较，就是一个相对具体的事物，同理"狗"也不一定就是具体的事物，它跟"哈士奇"进行比较，就是一个相对抽象的概念。

在代码中，“抽象”指接口、以及相对抽象化的类，注意这里相对抽象化的类并不特指“抽象类”（使用 abstract 关键字声明的类），只要一个类型在族群层次中比较靠上，那么它就可以算是抽象的，如上面举的“动物（Animal）”的例子；“具体”则指从接口、相对抽象化的类继承出来的类型，如从“动物（Animal）”继承得到的“狗（Dog）”类型。代码中抽象与具体的举例见下表 12-1：

表 12-1 抽象与具体举例

序号	抽象	具体	说明

1	Interface IWalkable
	{
	    void Walk();
	}
		class Dog:IWalkable
		{
		    public void Walk()
		    {
		        //…
		    }
		}
			IWalkable 接口是“抽象”，实现 IWalkable 接口的 Dog 类是“具体”。

2	class Dog:IWalkable
	{
	    public void Walk()
	    {
	        //…
	    }
	}
		class HaShiQi:Dog
		{
		    //…
		}
			Dog 类是“抽象”，继承自 Dog 类的 HaShiQi 类则是“具体”。

如果一个类型包含一个抽象的成员，比如“动物（Animal）”，那么这个成员可以是很多种类型，不仅可以是“狗（Dog）”，还可以是“猫（Cat）”或者其它从“动物（Animal）”派生的类型，但是如果一个类型包含一个相对具体的成员，比如“狗（Dog）”，那么这个成员就相对固定，不可再改变。很明显，抽象的东西更易改变，“抽象”在降低代码依赖方面起到了重要作用。

12.3.2 再看"依赖倒置原则"

本书前面章节在讲到“依赖倒置原则”时曾建议我们在软件设计时：

1）高层模块不应该直接依赖于低层模块，高层模块和低层模块都应该依赖于抽象；

2）抽象不应该依赖于具体，具体应该依赖于抽象。

抽象的事物不确定，一个类型如果包含一个接口类型成员，那么实现了该接口的所有类型均可以成为该类型的成员，同理，方法传参也一样，如果一个方法包含一个接口类型参数，那么实现了该接口的所有类型均可以作为方法的参数。根据“里氏替换原则（LSP）”介绍的，基类出现的地方，派生类均可以代替其出现。我们再看本章 12.2.3 小节中讲到的“依赖造成的尴尬”，DBManager 类型依赖一个具体的 SQLServerHelper 类型，它内部包含了一个 SQLServerHelper 类型成员，DBManager 和 SQLServerHelper 之间产生了一个不可变的绑定关系，如果我们想将数据库换成 MySQL 数据库，要做的工作不仅仅是增加一个 MySQLHelper 类型。

假设在软件系统设计初期，我们将访问各种数据库的相似操作提取出来，放到一个接口中，之后访问各种具体数据库的类型均实现该接口，并使 DBManager 类型依赖于该接口：

```
//Code 12-9
interface IDB    //NO.1
{
    void ExcuteSQL(string sql);
}
class SQLServerHelper:IDB      //NO.2
{
    //…
    public void ExcuteSQL(string sql)
    {
        //…
    }
}
class MySQLHelper:IDB      //NO.3
{
    //…
    public void ExcuteSQL(string sql)
    {
        //…
    }
}
class DBManager    //NO.4
{
    //…
    IDB _dbHelper;    //NO.5
    public DBManager(IDB dbHelper)
    {
        _dbHelper = dbHelper;
    }
    public void Add()    //NO.6
    {
        string sql = “”;
        //…
        _dbHelper.ExcuteSQL(sql);
    }
    public void Delete()    //NO.7
    {
        string sql = ””;
        //…
        _dbHelper.ExcuteSQL(sql);
    }
    public void Update()    //NO.8
    {
```

```
        string sql = "";
        //…
        _dbHelper.ExcuteSQL(sql);
    }
    public void Search()    //NO.9
    {
        string sql = "";
        //…
        _dbHelper.ExcuteSQL(sql);
    }
}
```

如上代码 Code 12-9 所示，我们将访问数据库的方法放到了 IDB 接口中（NO.1 处），之后所有访问其它具体数据库的类型均需实现该接口（NO.2 和 NO.3 处），同时 DBManager 类中不再包含具体 SQLServerHelper 类型引用，而是依赖于 IDB 接口（NO.5 处），这样一来，我们可以随便地将 SQLServerHelper 或者 MySQLHelper 类型对象作为 DBManager 的构造参数传入，甚至我们还可以新定义其它数据库访问类，只要该类实现了 IDB 接口，

```
//Code 12-10
class OracleHelper:IDB    //NO.1
{
    //…
    public void ExcuteSQL(string sql)
    {
        //…
    }
}
class Program
{
    static void Main()
    {
        DBManager dbManager = new DBManager(new OracleHelper());    //NO.2
    }
}
```

如上代码 Code 12-10，如果系统需要使用 Oracle 数据库，只需新增 OracleHelper 类型即可，使该类型实现 IDB 接口，不用修改系统其它任何代码，新增加的 OracleHelper 能够与已有代码合作得非常好。

修改后的代码中，DBManager 不再依赖于任何一个具体类型，而是依赖于一个抽象接口 IDB，见下图 12-15：

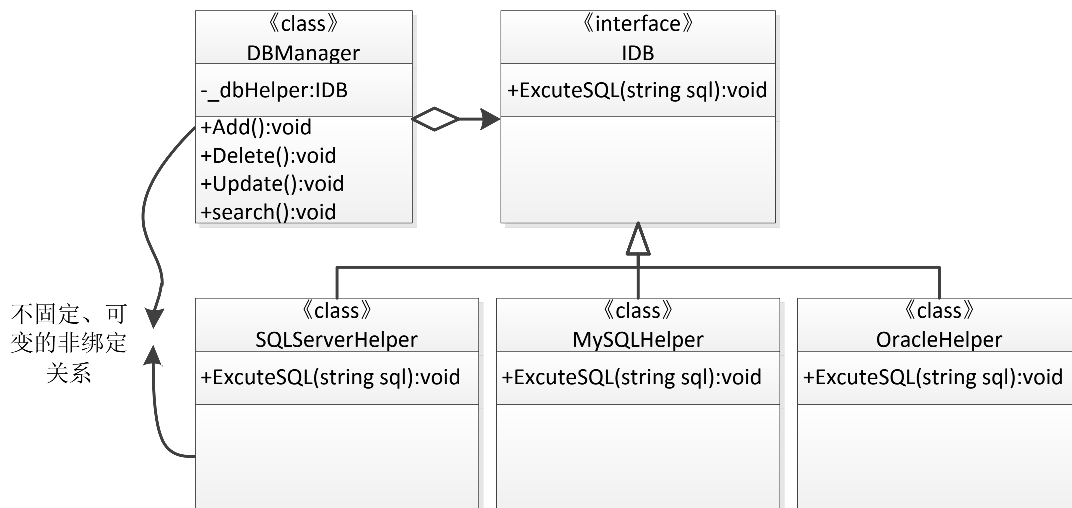


图 12-15 依赖于抽象

如上图 12-15，代码修改之前，DBManager 直接依赖于具体类型 SQLServerHelper，而代码修改后，DBManager 依赖于一个“抽象”，也就是说，被依赖者不确定是谁，可以是 SQLServerHelper，也可以是其它实现了 IDB 的任何类型，DBManager 与 SQLServerHelper 之间的依赖程度降低了。

理论上讲，任何一个类型都不应该包含有具体类型的成员，而只应该包含抽象类型成员；任何一个方法都不应该包含有具体类型参数，而只应该包含抽象类型参数。当然这只是理论情况，软件系统设计初期就已确定不会再改变的依赖关系，就不需要这么去做。

> 注：除了上面说到的将相同部分提取出来放到一个接口中，还有时候需要将相同部分提取出来，生成一个抽象化的基类，如抽象类。接口强调相同的行为，而抽象类一般强调相同的属性，并且用在有族群层次的类型设计当中。

12.3.3 依赖注入（DI）

当两个对象之间必须存在依赖关系时，“依赖倒置”为我们提供了一种降低代码依赖程度的思想，而“依赖注入（Dependency Injection）”为我们提供了一种具体产生依赖的方法，它强调“对象间产生依赖”的具体代码实现，是对象之间能够合作的前提。“依赖注入”分以下三种（本小节代码均以 12.3.2 小节中的代码为前提）：

（1）构造注入（Constructor Injection）；

通过构造方法，让依赖者与被依赖者产生依赖关系，

```
//Code 12-11
class DBManager
{
    //...
    IDB _dbHelper;
    public DBManager(IDB dbHelper)    //NO.1
    {
        _dbHelper = dbHelper;
    }
    public void Add()
    {
```

```
            string sql = ””;
            //...
            _dbHelper.ExcuteSQL(sql);
        }
        //...
}
class Program
{
        static void Main()
        {
            DBManager manager = new DBManager(new SQLServerHelper()); //NO.2
            DBManager manager2 = new DBManager(new MySQLHelper()); //NO.3
            DBManager manager3 = new DBManager(new OracleHelper()); //NO.4
        }
}
```

如上代码 Code 12-11 所示，DBManager 中包含一个 IDB 类型的成员，并通过构造方法初始化该成员（NO.1 处），之后可以在创建 DBManager 对象时分别传递不同的数据库访问对象（NO.2、NO.3 以及 NO.4 处）。

通过构造方法产生的依赖关系，一般在依赖者（manager、manager2 以及 manager3）的整个生命期中都有效。

> 注：虽然不能创建接口、抽象类的实例，但是可以存在它们的引用。

（2）方法注入（Method Injection）；

通过方法，让依赖者与被依赖者产生依赖关系，

```
//Code 12-12
class DBManager
{
        //...
        public void Add(IDB dbHelper)    //NO.1
        {
            string sql = “”;
            //...
            dbHelper.ExcuteSQL(sql);
        }
        //...
}
class Program
{
        static void Main()
        {
            DBManager manager = new DBManager();
            //...
            manager.Add(new SQLServerHelper());    //NO.2
            //...
```

```
            manager.Add(new MySQLHelper());    //NO.3
            //…
            manager.Add(new OracleHelper());    //NO.4
        }
}
```

如上代码 Code 12-12 所示，在 DBManager 的方法中包含 IDB 类型的参数（NO.1 处），我们在调用方法时，需要向它传递一些访问数据库的对象（NO.2、NO.3 以及 NO.4 处）。

通过方法产生的依赖关系，一般在方法体内部有效。

（3）属性注入（Property Injection）。

通过属性，让依赖者与被依赖者产生依赖关系，

```
//Code 12-13
class DBManager
{
        //…
        IDB _dbHelper;
        public IDB DBHelper    //NO.1
        {
                get
                {
                        return _dbHelper;
                }
                set
                {
                        _dbHelper = value;
                }
        }
        public void Add()
        {
                string sql = "";
                //…
                _dbHelper.ExcuteSQL(sql);
        }
        //…
}
class Program
{
        static void Main()
        {
                DBManager manager = new DBManager();
                //…
                manager.DBHelper = new SQLServerHelper();    //NO.2
                //…
                manager.DBHelper = new MySQLHelper();    //NO.3
                //…
```

```
            manager.DBHelper = new OracleHelper();    //NO.4
            //...
        }
}
```

如上代码 Code 12-13 所示，DBManager 中包含一个公开的 IDB 类型属性，在必要的时候，可以设置该属性（NO.2、NO.3 以及 NO.4 处）的值。

通过属性产生的依赖关系比较灵活，它的有效期一般介于“构造注入”和“方法注入”之间。

> 注：在很多场合，三种依赖注入的方式可以组合使用，即我们可以先通过"构造注入"让依赖者与被依赖者产生依赖关系，后期再使用"属性注入"的方式更改它们之间的依赖关系。"依赖注入（DI）"是以"依赖倒置""为前提的。

12.4 框架的"代码依赖"

12.4.1 控制转换（IoC）

“控制转换（Inversion Of Control）”强调程序运行控制权的转移，一般形容在软件系统中，框架主导着整个程序的运行流程，如框架确定了软件系统主要的业务逻辑结构，框架使用者则在框架已有的基础上扩展具体的业务功能，为此编写的代码均由框架在适当的时机进行调用。

“控制转换”改变了我们对程序运行流程的一贯认识，程序不再受开发者控制，

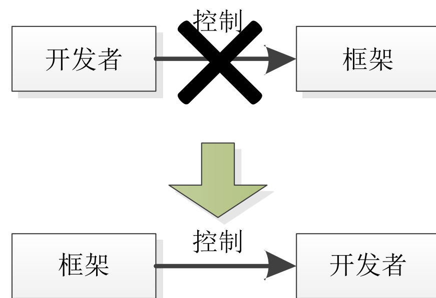


图 12-16 程序控制权的转移

如上图 12-16 所示，框架负责调用开发者编写的代码，框架控制整个程序的运转。

> 注："控制转换（IoC）、依赖倒置（DIP）以及依赖注入（DI）是三个不同性质的概念，"控制转换"强调程序控制权的转移，注重软件运行流程；"依赖倒置"是一种降低代码依赖程度的理论指导思想，它注重软件结构；"依赖注入"是对象之间产生依赖关系的一种具体实现方式，它注重编程实现。笔者认为有的书籍将三者做相等或者相似的比较是不准确的。

通常，又称“控制转换（IoC）”为“好莱坞原则（Hollywood Principle）”，它建议框架与开发者编写代码之间的关系是：“Don’t call us,we will call you.”，即整个程序的主动权在框架手中。

12.4.2 依赖注入（DI）对框架的意义

框架与开发者编写的代码之间有“调用”与“被调用”的关系，所以避免不了依赖的产生，“依赖注入”是框架与开发者编写代码之间相结合的一种方式。任何一个框架的创建者不仅仅要遵循“依赖倒置原则”，使创建出来的框架与框架使用者之间的依赖程度最小，还应该充分考虑两者之间产生依赖的方式。

> 注：“框架创建者”指开发框架的团队，“框架使用者”指使用框架开发应用程序的程序员。

12.5 本章回顾

本章首先介绍了面向对象的三大特征：封装、继承和多态，它们是面向对象的主要内容。之后介绍了面向对象的软件系统开发过程中不可避免的代码依赖，还提到了不合理的代码依赖给我们系统开发带来的负面影响，有问题就要找出解决问题的方法，随后我们从认识“具体”和“抽象”开始，逐渐地了解可以降低代码依赖程度的具体方法，在这个过程中，“依赖倒置（DIP）”是我们前进的理论指导思想，“高内聚、低耦合”是我们追求的目标。

12.6 本章思考

1.简述“面向对象”的三大特征。

A：从对象基础、对象扩展以及对象行为三个方面来讲，“面向对象（OO）”主要包含三大特征，分别是：封装、继承和多态。封装是前提，它强调代码模块化，将数据以及相关的操作组合成为一个整体，对外只公开必要的访问接口；继承是在封装的前提下，创建新类型的一种方式，它建议有族群关系的类型之间可以发生自上而下地衍生关系，处在族群底层的类型具备高层类型的所有特性；多态强调对象的多种表现行为，它是建立在继承的基础之上的，多态同时也是降低代码依赖程度的关键。

2.简述“面向抽象编程”的具体含义。

A：如果说“面向对象编程”教我们将代码世界中的所有事物均看成是一个整体——“对象”，那么“面向抽象编程”教我们将代码中所有的依赖关系都建立在“抽象”之上，一切依赖均是基于抽象的，对象跟对象之间不应该有直接具体类型的引用关系。“面向接口编程”是“面向抽象编程”

的一种。

3.“依赖倒置原则（DIP）”中的“倒置”二字作何解释？

A：正常逻辑思维中，高层模块依赖底层模块是天经地义、理所当然的，而“依赖倒置原则”建议我们所有的高层模块不应该直接依赖于底层模块，而都应该依赖于一个抽象，注意这里的“倒置”二字并不是“反过来”的意思（即底层模块反过来依赖于高层模块），它只是说明正常逻辑思维中的依赖顺序发生了变化，把所有违背了正常思维的东西都称之为“倒置”。

4.在软件设计过程中，为了降低代码之间的依赖程度，我们遵循的设计原则是什么？我们设计的目标是什么？

A：有两大设计原则主要是为了降低代码依赖程度，即：单一职责原则（SRP）和依赖倒置原则（DIP）。我们在软件设计时追求的目标是：高内聚、低耦合。