TURING 图灵程序设计丛书 .NET系列

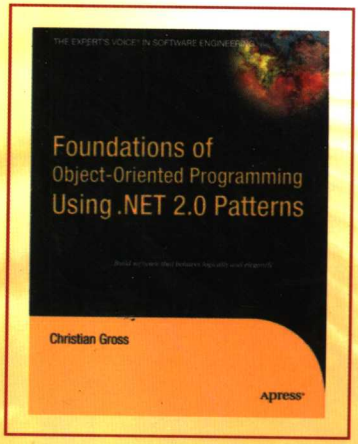
Apress®

Foundations of Object-Oriented Programming
Using .NET 2.0 Patterns

.NET 2.0 模式开发实战

[美] Christian Gross 著
张凯峰 李彦娜 张广亮 译

- 一本与众不同的现代模式图书
- 凝聚程序设计专家的经验与领悟
- 注重实践，提供大量实战代码



人民邮电出版社
POSTS & TELECOM PRESS

TP393.09
123
2007

TURING 图灵程序设计丛书 .NET 系列

.NET 2.0 模式开发实战

Foundations of Object-Oriented Programming Using .NET 2.0 Patterns

[美] Christian Gross 著

张凯峰 李彦娜 张广亮 译

人民邮电出版社
北 京

图书在版编目(CIP)数据

.NET 2.0 模式开发实战/(美)戈洛斯著；张凯峰等译. —北京：人民邮电出版社，2007.5
(图灵程序设计丛书)

ISBN 978-7-115-15385-2/TP

Ⅰ. .N… Ⅱ. ①戈… ②张… Ⅲ. 计算机网络－程序设计 Ⅳ. TP393

中国版本图书馆 CIP 数据核字(2006)第 119846 号

内 容 提 要

本书阐述了应用于.NET 2.0 框架的设计模式，重点以 C#语言来演示应用各种模式。书中的主要内容包括面向对象编程的实质、模块化和异常、测试驱动开发、基本设计模式、应用于架构策略的设计模式、序列化和持久化等，同时针对测试、模式和重构，阐述了相关联的面向对象编程。

本书适合高等学校计算机相关专业师生，以及从事.NET程序设计的程序员阅读。

图灵程序设计丛书

.NET 2.0 模式开发实战

◆ 著 [美] Christian Gross
译 张凯峰 李彦娜 张广亮
责任编辑 傅志红

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京顺义振华印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：800 × 1000 1/16
印张：20
字数：467 千字 2007 年 5 月第 1 版
印数：1－5 000 册 2007 年 5 月北京第 1 次印刷

著作权合同登记号 图字：01-2006-1719 号

ISBN 978-7-115-15385-2/TP

定价：45.00 元

读者服务热线：(010)88593802 印装质量热线：(010)67129223

版权声明

Original English language edition, entitled *Foundations of Object-Oriented Programming Using .NET 2.0 Patterns* by Christian Gross, published by Apress L. P., 2560 Ninth Street, Suite 219, Berkeley, CA 94710 USA.

Copyright © 2006 by Christian Gross. Simplified Chinese-language edition copyright © 2007 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由 Apress L. P. 授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

译 者 序

初看书名中的 Foundation，会让人觉得这不过又是一本讲解面向对象以及设计模式基础知识的图书。市面上流行的模式方面的书籍还不够多吗？

说实话，刚从图灵公司编辑处获得此书时，我的感觉也是这样，甚至有一点点不屑。但随着深入地翻译，我越发感觉这样的书名对于这样的内容也许根本就是不合适的。未曾阅读章节内容，仅从目录即可一瞥作者的别具匠心：作者是在从宏观的角度引导如何构建健壮而具有可维护性和可扩展性的应用程序。

从程序的根基，到架构，到脉络（实现算法），到代码合理性，到数据持久化，到对模式的重构，作者以设计模式为主线穿插其中，深入浅出，融会贯通，为读者展现了一条清晰易懂的构建之路。

虽然书中的程序代码是以 C#编写的，但是简洁易懂，对于稍具 OOP 经验的开发者来说完全不足以成为阅读的障碍，毕竟，书中的精髓在于思想而不是代码的堆砌。在领悟了作者的构建之道后，聪明的开发者一定能够举一反三。

我的同事李彦娜、张广亮也参与了本书的翻译工作，在此对他们表示由衷的感谢。牺牲了那么多私人生活时间，个中的辛苦只有他们自己了解。同时感谢他们能够容忍我一遍一遍的催促和埋怨。

感谢所有热心的读者，希望本书能在你通往技术殿堂的道路上提供许多的帮助。翻译中有不尽如人意的地方在所难免，期待读者的批评指正。

感谢傅志红编辑和人民邮电出版社给予我们这样的机会，在翻译的同时我们又学到了很多东西。

最后，感谢所有为此书的翻译、制作、出版、发行做出贡献的人们。

译 者

2006 年 8 月

此为试读，需要完整PDF请访问：www.ertongbook.com

前　言

模式是一个已经被前人充分讨论过的有趣话题。最早谈及模式的书是 Erich Gamma 等编写的《设计模式》（Boston：Addison-Wesley，1995）。该书出版时，模式还是一种新的思想和概念；而现在该书已经成为设计模式的必备参考书，其中阐述的设计模式成为了各种应用程序的基础。

本书阐述了应用于.NET 2.0 框架的设计模式。其中的一些设计模式来自于该书，一些则来自其他来源。本书的重点并不是要定义各种模式，而是用一种编程语言（例如 C#）来演示各种模式的应用。因为最初的设计模式是利用C++ 进行阐述的，与.NET以及 C#之间存在着很大的语言差异。

为什么使用模式

你也许会问："为什么要写这本书，为什么要使用模式?"的确很奇怪，这个想法是在我培训学生如何使用设计模式的时候冒出来的。当时学生们正在做练习，我看见一位学生编写代码时使用一个类而不是接口作为基类。我问他为什么这样编写代码，他的回答是："因为我一直都是这样做的。"这位同学的方法是错误的，但是他的回答揭示了一个很有趣的观点：当一个基类足够好的时候为什么还要使用接口呢？就在那时，我突然想到：先教授模式再教授面向对象编程（OOP），理解就会更为容易。

当学习传统的 OOP 技术时，老师会讲到有关形状、矩形以及其他的抽象主题。你会学到一个类如何负责它自己的数据以及如何实现这些责任。这种学习方式的问题是，它解释了面向对象编程，但是并没有解释如何利用具体的方案来解决问题。例如，如何实例化一个类型，将它的引用传递给另一个类，并将信息持久化到媒介中？传统面向对象编程的问题是，在实现解决方案时只给出了模糊的指导。

模式就不同了，它给出了一个预定义的形式，告诉你在什么时候该做什么。考虑下面的情况：学习如何烘焙蛋糕。你知道，蛋糕是由面粉、鸡蛋、牛奶、发酵粉以及一些其他原料做成的，但并不是所有蛋糕都是一样的。如果加入过多的发酵粉，蛋糕会过分地发酵胀起；如果加入过多的鸡蛋，蛋糕吃起来会像是煎蛋卷。面向对象编程就像是烘焙蛋糕一样，你知道所需的原料，也知道这些原料的作用，但是不知道这些原料的比例以及制作蛋糕的步骤。就像好的食谱能够帮助你把这些原料变成美味的蛋糕一样，模式能够帮助你将代码转换成高效的程序。

在不知道原料的作用以及它们之间是如何相互作用的情况下，也可能只根据食谱制作出蛋糕。但这种方法的问题是它无法奏效。例如，假设你正在准备一顿饭，开始是豌豆汤，主菜是胡椒薄荷沙司调味的鲑鱼、南瓜以及黄豆面条，最后是餐后甜点冰淇淋。尽管可能每道菜的味道都很好，但这个菜谱听起来并不令人胃口大开。其中的问题是模式，就像是菜谱，之间是相互作用和配合的，因此需要协调。协调模式的前提就是对面向对象编程技术有一个基本的理解。

什么是模式

模式就像菜谱，只不过用于创建模式的原料是面向对象的原理。例如，考虑下面的源代码：

```
interface IBaseInterface { }

class Derived : IBaseInterface { }
```

类 Derived 实现了接口 IBaseInterface，这就是典型的 Bridge 模式的实现。一定会有人问，为什么使用接口而不是类呢？从面向对象的角度看，它们的结果很相似。这就引发了一个问题，为什么在编程语言中要用接口呢？答案是：因为模式定义了最佳实践，而该实践已经被证明在应用程序开发中是有益的。从大量的编程经验来看，在使用基类型以及实现 Bridge 模式时，接口比类更有用。

模式不同于最佳实践，就像理论不同于猜想一样。理论基于已经被科学方法无数次证明的想法，科学方法是一种其他人也能够重新生成结果的实验方法。模式就像是一本食谱，它定义了一系列的步骤以及原料，当按照定义的步骤操作原料时，就能够做出相同的蛋糕。猜想是一种想法，它很可能是正确的，但是并没有被不同的人用必要的重复的实验证明。猜想是理论的先驱，意味着最佳实践是模式的先驱。

《设计模式》一书中定义的模式已经被证明是有效的，并且已经应用于多种情况。例如，利用 Factory 模式定义的结构，可以在不同的编程环境中，利用不同的编程语言产生相同的结果。

你能从本书获得什么

除了模式以及面向对象原理，通过本书将学习像音乐家阅读乐谱那样阅读代码。思考一下，音乐家利用只有音乐家才懂的符号来表达他们的思想。乐谱，代表了要演奏的音乐，对于不懂的人来说就像是一堆胡乱画出的符号。然而对于音乐家，它是巴赫、莫扎特、AC/DC 乐队或者说唱乐手 Eminem，意味深长。当阅读乐谱时，音乐家知道何时演奏他们的乐器，以及当时产生的是一种什么样的情绪。没有对乐谱的简化和抽象，音乐家阅读乐谱后只知道那是一段怎样的音乐。

本书的目的就是使你能够阅读一段代码并且明白其中的意图。在很多开源项目中，经常需要阅读其他人编写的代码。对于其他类型的项目情况就不是这样，每位编程者都使用自己喜欢的风格和技术编写代码。例如，就像是将大括号放在哪里这样的小问题也会引发激烈的争论。会有这种争论的原因是，每位编程者都在实践中或者参加会议的幻灯片中或者别人的注释中学到了一些风格和技术。阅读一段代码并不只是识别出一些关键字，而是理解在一种环境中使用一段关键字解决一个特定的问题。简言之，阅读一段代码就是在模式的环境中诠释 OOP。

从哲学的层面考虑一下，一种编程语言为什么会存在？问问自己，为什么一种编程语言包含了一个特定的关键字、概念或者策略？答案是，因为那个关键字、概念或者策略在一个特定的环境中解决了一个特定的问题。但并没有解释何时才是使用这种特征的最佳时机。例如，在 C#中，可以使用关键字 struct、class、interface、abstract 以及 sealed 来定义一个用于程序的

类型。通常编程者知道上述关键字的技术理由以及效果。然而，编程者通常不知道每个关键字在什么情况下应该或者不应该使用。这就回到了起点，不管在什么地方使用，编程者会依赖于他们自己的习惯或风格。这并不是正确的编程方法，因为它有可能依赖于试错（trial and error）。

本书内容

本书的每一章在知识上都是循序渐进的，并且各有明确的目标。

- 第 1 章揭示了面向对象编程的本质，利用 C#定义和阐述了模块化和异常等概念。阅读这一章有助于更容易地理解各种 OOP 词汇。
- 第 2 章的主要目的是定义和阐述测试驱动开发，它是一种通过一致地创建测试和源代码来开发软件的方法。这种方法的优点是保持源代码的稳定和一致。
- 第 3 章将会接触到作为其他设计模式基础的基本设计模式——Factory（工厂）模式和 Bridge（桥接）模式。
- 第 4 章演示了将基本模式应用于架构策略的创建。上一章介绍的设计模式是通用的，但并不能用于**整体架构**。架构策略能够很容易地扩展和维护，阐述了如何解决应用问题。利用定义好的模式，演示了如何开始创建具有正确开端的应用程序。
- 当实现应用程序时，主要问题经常是利用良好开发的代码使应用程序能够正常工作。接下来的需求和要求会增加新的类或者修改现有的类，而这些增加或修改可能会引发问题。第 5 章探讨了这个问题，提出了一些用于更容易实现应用程序的模式。
- 在编写代码时，通常试图走一些捷径来解决问题。这个捷径可能很小但却可能有重大的影响。在第 6 章中给出了一些模式，定义如何处理想要走捷径的情况，其中解释的模式都很详细，似乎有些大题小作。但是，这些模式的主要目标是简化对代码的扩展和维护。
- 在编写代码时，目标是使代码尽可能的高效。第 7 章教你如何编写在扩展以及维护的同时又不增加理解难度的高效代码。
- 序列化和持久化是很多关于模式的图书回避的话题，然而它们非常重要，第 8 章中讨论了这个话题。这一章中阐述的模式和策略显著地简化了向其他媒体写数据的过程。
- 最后一章通过介绍重构将所有一切包装起来。这一章的重点并不是介绍重构的所有方面，而是定义并阐述如何利用本书中定义的模式来重构现有的代码。

目　录

此为试读，需要完整PDF请访问：www.ertongbook.com

第1章 面向对象编程的实质

在开始了解模式之前，需要熟悉一些用在面向对象编程（Object-Oriented Programming，OOP）中的普通术语。在本章以及整本书中，将会使用C#来演示一些关键的面向对象编程概念。在使用C#语言编写面向对象的代码时，需要知道四个主要的概念：类型、作用域、继承以及泛型。这四个因素综合起来决定了一个面向对象应用程序是否能够正常工作。这四个主要的概念在C#语言中是用关键字实现的，例如class或者interface，本章将会讨论与这些概念相关的、必须了解的内容。

C#是一种面向对象的编程语言，使用它可以在.NET平台上利用.NET运行时创建应用程序。讨论C#就不能不谈到.NET平台，因为C#依赖于.NET运行时。有些人认为，C#是可以用来与.NET运行时交互的最好的语言。本书只会顺便讨论一下.NET及其API的特定细节，这些细节相对于面向对象编程来说并不是主要的。

1.1 理解类型

类型是封装了数据和方法的一块代码。所有类型中最简单的就是类（class），其定义如下：

```
class SimpleClass { }
```

在C#中，使用关键字、标识符和花括号来定义类型。class是一个关键字，用来定义标识符为SimpleClass的类；而花括号则定义了一个代码块，花括号中间的所有内容都属于SimpleClass类型。class关键字还可以和一些属性（attribute）联合使用，但在上面的例子中并没有额外的属性。其他可以使用的类型还有struct、interface以及泛型（generic）（泛型将会在1.6节中讨论）。

让我们花点时间来看看struct。下面的例子演示了如何定义一个struct类型：

```
struct SimpleStructure { }
```

struct和class之间的区别在于，前者是值类型而后者是引用类型。引用类型和值类型之间的不同在于实现细节，其中也有许多概念性的区别，但它们都是实现的直接结果。从面向对象的概念性角度来看，二者都定义了类型。

之所以需要类型，是因为它定义了一块代码，而这块代码可以被另一块代码所引用。下面的代码示例演示了如何引用类型：

```
struct SimpleStruct {
}
class SimpleClass {
    SimpleStruct _myStruct;
    public SimpleStruct getData() {
        return _myStruct;
    }
}

class MainApp
{
    static void testMethod() {
        SimpleClass cls = new SimpleClass();
        SimpleStruct structure = cls.getData();
    }
}
```

MainApp 类有一个 testMethod 方法，其中引用了 SimpleClass 和 SimpleStruct 两个类型。SimpleClass 类引用了 SimpleStruct 类型。示例代码看起来很简单，是开发者通常使用的经典示例。引用产生了依赖，这导致了代码的复杂化。即使依赖关系发生很小的变化，也可能会导致其他重大的变化。

在示例代码中，SimpleStruct 类型没有引用（也不需要知道）SimpleClass 的细节。但是 SimpleClass 必须对 SimpleStruct 有所了解。SimpleClass 类型和 SimpleStruct 类型同属于一个模块，使用这个模块对外所暴露的功能的源代码称为*使用者*（consumer）。testMethod 方法就代表使用者，它必须知道 SimpleClass 类型和 SimpleStruct 类型。假设要修改 SimpleStruct，那么两处代码，即使用者代码和 SimpleClass 的实现，都需要进行检查并有可能需要修改。检查模块内部实现的改变是比较直接的，因为模块是自包含的。但是令人头疼的是使用者代码，因为 SimpleStruct 的变化需要使用者代码做相应的变化，这可能会比模块（内部变化）更加复杂。

以上恰恰是对可重用面向对象代码的讽刺。因为有越多的使用者引用和使用同一个模块的代码，就可能出现越多的潜在问题。这是因为模块并不能规定使用者如何使用自己。使用者可能本来在正常运行，但一旦模块更新，这些变化就有可能会引起应用程序中的逻辑错误。下面将讨论可能会遇到的各种错误。

有两种类型的错误：语法上的和逻辑上的。在语法的错误中，类型的细节发生了变化，信息可能变得多余或者缺少，编译器就会捕捉到这些变化。大部分情况下，可能是参数错误导致了这些问题。语法错误并不会引起太大的问题，因为程序员一般能够很快地修正错误，尽管众多这样的错误修改起来的确很乏味。

逻辑上的错误则更成问题。逻辑错误是指在类型的细节发生变化时可能并不会引起语法上的错误。因为编译器不会报错，从而使得这种类型的错误更具有挑战性。在程序运行时，防御

性代码（内建错误机制的代码）能够捕捉到这类错误。逻辑错误通常称为 bug。导致逻辑错误的原因通常可能是代码设计得不好，实现存在问题，或者缺乏测试。逻辑错误的发生是由于使用者与模块之间存在误解。也许有人会争辩说逻辑错误不是错误，而是由必要的变化导致的结果。这让我想起了我的工程静力学教授说过的："桥要么矗立要么倒塌，绝没有中间状态。"逻辑错误叫什么并不要紧，重要的是代码不能正常工作，需要修改。不管逻辑错误产生的原因是什么，由其导致的经济上的损失有可能会非常惨重。

代码中包含对其他类型的不可控制或者不可管理的引用可能是导致逻辑错误的一个原因。在前面看到的简单的使用者代码的例子中，只引用了两个类型。想像一下，假设有四个分别只引用两个或者三个类型的简单的使用者代码，如果其中任何一个使用者的实现代码在一个错误的时机进行了方法调用，或者做出了某种错误的假设，如果按照排列组合的方式计算，逻辑错误的数量将会是惊人的。如果修复一个逻辑错误时又导致了其他逻辑错误的产生，情况会变得更糟，这是致命逻辑错误的一种形式，也就是所说的特征交互作用（feature-interaction）。

特征交互作用错误改起来很困难并且枯燥，而且经常有可能导致代码需要重写，所以应该尽量避免。模式有助于解决这个问题。测试驱动的开发使得特征交互作用错误变得可控，因为任何可能引起问题的变化都能被一下子识别出来。尽早发现就能够尽早修正，这就保证了能够将特征交互作用错误消灭在萌芽状态。

考虑一下这样的情形：一个房间中有很多人，每个人是一个类型，就像在一个聚会中每个人都可以和任何其他人交流一样，一个类型也可以引用任何其他的类型。如果交流是在有组织的情况下进行的，同一时刻只有几个人在交谈，那只会产生轻微的低语声。如果事情失控了，那么声音可能会变得很大，甚至上升至刺耳的噪声。这是因为如果一个人想和另一个人说话，她就要想办法使自己的声音大过房间中的噪声。另一个人也可能在做着同样的事情，很快你会发现所有人都在叫嚷。如果声音能被保持在一个较低的程度，那么一切都还是可以控制的。再联想到类型与引用，如果类型之间的引用不遵守规则，那么很快代码就会因为有太多的类型引用而变得难以理解。

1.2 模块化与可重用代码

模块化代码是指根据功能分离成块的代码；可重用代码是指能够适用于不同情况的代码。模块化代码不一定是可重用代码，可重用代码也不一定是模块化代码，理解这一点很重要。利用率很高的可重用代码很有可能不是模块化代码。这两点可能听起来有些令人迷惑，因为我们经常把模块化与可重用性混为一谈。

Software Engineering: A Practitioner's Approach[1]一书对模块化做了非常好的解释。这本书中将模块化解释为一个有 40 年历史的概念，概括来说就是"软件被分成独立命名并存储的源代码

1. Roger S. Pressman, *Software Engineering: A Practitioner's Approach, European Adaptation*, 3rd ed. (London: McGraw-Hill, 1994), p. 325。

块，这些源代码块能够集成到一个程序中来解决问题”。

一个模块化程序包含一些特定的代码片段，这些代码片段通过接口相互调用来解决问题。模块化使得在实施解决方案的过程中可以采用分治策略。模块由它的约定（contract）来定义，该约定明确了模块的功能。开发人员知道了约定之后，只需要编写源代码来满足约定的要求即可。之后，可以将不同的模块组合在一起，如果正确定义并实现了约定，那么最终的程序就可以满足要求了。打个比方来说，这个过程就像是同时从两边开始建一座桥。如果正确定义并实现了约定，那么桥的两段就能够在中间的某个位置正确会合。

下面的源代码是一个模块化应用程序的例子：

```
class User1 {
}
class User2 {
}
class User3 {
}
class MainApp
{
    static void testMethod() {
        User1 cls1 = new User1();
        // Do something with cls1
        User2 cls2 = new User2();
        // Do something with cls2
        User3 cls3 = new User3();
        // Do something with cls3
    }
}
```

User1、User2 和 User3 是三个特定的相互独立的类型，相互之间没有影响。testMethod 方法实例化了这三个类型，并且利用它们执行了一些逻辑。这三个类型的使用者独立地使用了这些类型。在实际情况中，可能存在一些相互影响，但将一个应用程序模块化的目标是将代码段相互分离，消除相互之间的影响。

注意　源代码的可重用性与模块化的目标不同，对可重用性最好的诠释是“能够更适用于不止一种上下文中”[1]。

可重用性的思想是定义可供多个使用者使用的代码。可重用的代码应该是经常被使用的，并且理论上能够减少开发者编写代码的需要。在这种情况下会出现一个问题，可重用代码难以修改（脆弱性代码的一种表现），因为对可重用代码做很小的改动也许会引起很多的变化。下面的程序段说明了这一点：

1. Grady Booch, *Object-Oriented Analysis and Design with Applications*, 2nd ed. (Boston: Addison-Wesley, 1993), p. 138。

```
class BaseClass {
}
class User1 {
    BaseClass _inst1;
}
class User2 {
    BaseClass _inst2;
}
class User3 {
    BaseClass _inst3;
}
class MainApp
{
    static void testMethod() {
        User1 cls1 = new User1();
        // Do something with cls1
        User2 cls2 = new User2();
        // Do something with cls2
        User3 cls3 = new User3();
        // Do something with cls3
    }
}
```

本例中 BaseClass 类型被三个类（User1、User2 和 User3）所使用，这说明 BaseClass 是可重用的。从另一方面来讲，如果修改了 BaseClass，那么需要检查所有用到 BaseClass 的地方。这就是导致代码脆弱性问题的原因。要对 BaseClass 做修改，就需要检查 BaseClass 是如何被使用的。编译器可以发现与使用 BaseClass 相关的语法错误，但是与使用 BaseClass 相关的逻辑错误就很难被发现，这就提高了对测试程序的要求。

BaseClass 类型并没有模块化，因为对类型的使用跨越了不同的模块。可能会有争议认为 BaseClass 自身就是一个模块。但问题是这个可重用的模块被使用的频率太高，以至于不能将它看做一个模块。有些书认为可重用类型或者可重用类型的集合应该规模较小[1]。

要记住，模块化与可重用性应该被看做是彼此无关的。模块化与重用性都不是最终目标，而应该根据需要来使用。通常来说，模块化实现分治策略，可重用性是在不同地方对功能的重复利用。它们两个都是非常有用的，本书后面会有详细的论述。

1.3 利用作用域控制访问

回到 1.1 节中所举的房间里人谈话的例子，每一个人都可以接触到另一个人并且和他说话。假设这个房间里有聚会，不同的人之间相互交谈是件好事，而且可以形成一个网络。但反过来，如果这个房间是法庭，这样的交谈就会引起混乱。法庭是需要秩序的，只在特定的时间允许特定的人讲话。C#编程语言既可满足聚会的需要也可满足法庭的要求，这取决于使用什么样的访

1. Steve McConnell, *Rapid Development* (Redmond, WA: Microsoft Press, 1996), p. 533。

问修饰符。

访问修饰符可用于类型声明，也可用于与之相关联的方法与数据。想像在一个聚会中，你可以去和别人谈话，但这并不意味着你有权力翻看别人口袋里面有多少钱。C#中也有同样的概念，可以使用相同的类型，但是不一定可以使用相同的方法与数据成员。访问修饰符使你可以管理对于一个类型的方法和数据的引用。

在解释作用域修饰符的本质前，需要说明一个额外的信息。通常当你创建一个.NET 应用程序时，C#编译器会产生一个源代码文件的集合，它们被编译成一个二进制的.NET 模块[1]。可以将这个.NET 模块看做一个原始的.NET 可执行代码。问题是目前.NET 运行时还不知道如何去执行这个模块。因此，应该将这个模块打包，这样就可以产生一个可以自己执行的程序或者程序集。作用域标识符不但影响被编译在同一个程序集中的代码，而且会影响到类型如何访问其他程序集里面的类型。

.NET 环境中包括下面一些作用域修饰符：public、internal、protected、internal protected 和 private。并不是任何时候都能使用所有的修饰符。当声明类型的时候，只能使用 public 和 internal 修饰符。在声明一个类型的方法或者数据成员的时候则可以使用所有的修饰符。

下面解释一下各种作用域修饰符。

- internal：可用于定义类型、方法或者数据成员，在程序集内拥有公共的作用域。类型、方法或者数据成员不能被该程序集以外的对象访问。
- internal protected：这个保护性的修饰符只能用于方法或者数据成员，它相当于 internal 和 protected 修饰符的结合。意味着被它修饰的方法或者数据成员在父类类型中具有公共的作用域。
- private：定义一个私有的方法或者数据成员，只能在定义它们的类型内部进行访问。
- protected：被该修饰符修饰的方法或者数据成员对于试图直接使用它们的代码来说，其作用域与 private 是一样的。但是如果从父类类型中调用这些方法或者数据成员，它们的作用域又类似于 public。对于快速引用来说，protected 的作用域相当于 public 还是 private 就要根据语境来判断了。下节将给出继承的详细解释。
- public：定义一个类型、方法或者数据成员时，如果使用 public 修饰符，则对于任何程序集或者程序的作用域来说它都是可见的。public 作用域修饰符是所有作用域修饰符中程度最宽松的一个，因此，从可能产生非意愿的引用的角度来说，也是最危险的一个因素。

在本书中讲解的模式都会使用作用域修饰符来规定使用者如何引用并使用类型。作用域修饰符很重要，在使用的时候要仔细考虑。当遇到一个复杂的问题时，开发人员通常会直接使用 public 修饰符。这是个不好的习惯，可能会引起类型的不可控引用问题。还记得那个聚会中噪音失控的例子吗？使用错误的作用域修饰符就可能导致同样的问题。

---

1. Don Box 与 Chris Sells，*Essential .NET Volume 1: The Common Language Runtime*（Boston：Addison-Wesley，2003），pp. 13 - 23。

此为试读，需要完整PDF请访问：www.ertongbook.com