华章程序员书库

Pearson

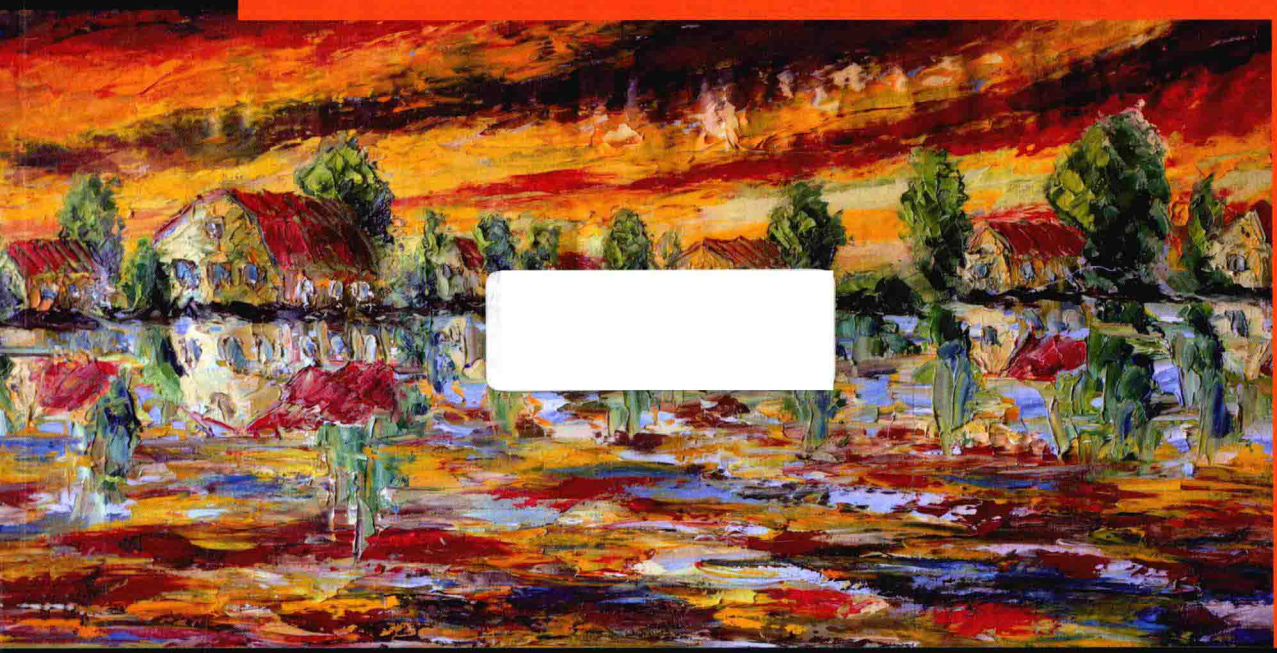
HZ BOOKS 华章IT

面向对象的思考过程

（原书第4版）

The Object-Oriented Thought Process

Fourth Edition

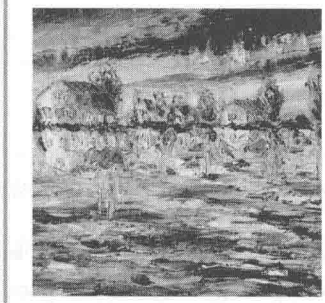


[美] 马特·魏斯费尔德（Matt Weisfeld） 著

黄博文 译

机械工业出版社
China Machine Press

华章程序员书库



The Object-Oriented Thought Process

Fourth Edition

面向对象的思考过程

（原书第4版）

[美] 马特·魏斯费尔德（Matt Weisfeld） 著

黄博文 译

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

面向对象的思考过程（原书第 4 版）/（美）马特·魏斯费尔德（Matt Weisfeld）著；黄博文译．—北京：机械工业出版社，2016.11
（华章程序员书库）
书名原文：The Object-Oriented Thought Process, Fourth Edition

ISBN 978-7-111-55308-3

I. 面… II. ①马… ②黄… III. 面向对象语言－程序设计 IV. TP312.8

中国版本图书馆 CIP 数据核字（2016）第 262345 号

本书版权登记号：图字：01-2013-5977

面向对象的思考过程（原书第 4 版）

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：陈佳媛　　责任校对：殷　虹
印　　刷：北京诚信伟业印刷有限公司　　版　　次：2016 年 11 月第 1 版第 1 次印刷
开　　本：186mm×240mm　1/16　　印　　张：15.5
书　　号：ISBN 978-7-111-55308-3　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

The Translator's Words 译 者 序

很多IT从业人员进入这个行业都是从学习一门编程语言开始的。对于编程，我们往往过于关注语言的语法细节，反而忽略了其背后的设计理念。面向对象的思考过程就是一个非常优秀的设计理念。它可以独立于语言存在。如果你熟练掌握了面向对象的思考过程，那么就可以轻松地在不同的面向对象的语言之间切换。

本书透彻地阐述了面向对象这一概念。作者Matt在书中反复强调学习面向对象的思考过程优于学习任何编程语言或工具。事实上，他也是这么做的。Matt阐述了面向对象的三要素：继承、封装、多态，并且自己加上了第四个要素：组合。关于组合，Matt不惜篇幅做了大量的讲解，并且列举了很多通俗易懂的例子，这也是本书的一大特色。

Matt也纠正了人们的一些普遍误解，比如面向对象的范式与面向过程的范式并不是完全对立的关系。而且在应用面向对象的设计和开发时，Matt也讲解了不少如何与遗留系统集成的技巧。同时，Matt也简要介绍了UML这个建模利器。为了不混淆重点，他把介绍UML的章节放置在很靠后的位置。因为他明白，先了解面向对象的各项概念是最重要的。

我虽然拥有多年的从业经验，但是再看本书时仍然有不少收获。其实自从我接触了函数式编程，就渐渐成为函数式编程的拥趸。我会时不时地“鼓吹”函数式编程范式的好处，顺便“贬低”一下面向对象编程。但同时我也有个疑问，既然函数式编程这么好，为什么这几年的发展只能算是波澜不惊，而没有掀起大风浪呢？读了本书之后，我似乎找到了答案。首先面向对象的思考过程更加符合大家对世界的直观感受，毕竟不是每个人都是数学家。函数式编程可以简化很多问题，但它并不能简化所有问题。其次是面向对象的编程范式和函数式编程的范式并不是完全对立的，正如作者讲过，面向过程的编程范式和面向对象的编程范式也不是完全对立的。比如目前流行的一些语言（Scala、Go等）都具备函数式的特点，也兼具面向对象的特点（只不过它们的面向对象的机制与传统的方式有所不同）。所以无论你喜欢哪种编程范式，了解彼此的不同之处是至关重要的。而本书则是了解面向对象范式的优秀

书籍。

本书已经更新到了第 4 版。从本书长达 10 多年的跨度来看，面向对象范式经久不衰。Matt 也适时地在新版中加入了一些新的主题，比如可移植数据、分布式系统、Web 服务等。Matt 不仅阐述了这些技术，还讲述了它们的前世今生。这样可以帮助读者更加充分地了解技术的演化之路。

无论你是否有面向对象编程的经验，本书都适合你作为面向对象思考的旅程开端。最后，希望本书能给大家带来超凡的阅读体验。

About the Author 作者简介

Matt Weisfeld 居住于美国俄亥俄州的克利夫兰市。他既是大学教授、软件开发工程师，也是作家。他在信息技术领域拥有 20 年的经验，之后进入大学任教。他先后当过软件开发工程师、企业家以及兼职教授。Matt 拥有计算机科学硕士学位以及工商管理硕士学位。Matt 除了本书之外，Matt 还撰写了其他两本关于软件开发的书籍，并在杂志和期刊上发表了多篇文章。这些杂志和期刊包括《developer.com》《Dr.Dobb's》《C/C++ Users》《Software Development》《Java Report》和国际期刊《Project Management》等。

前　言 Preface

本书内容概要

正如书名所述，本书讲述了面向对象思考的过程。选择一本书的主题和书名是个很重要的决定，但如果主题概念性很强，决定就没那么容易了。大部分书籍都只涉及了编程及面向对象设计的某个方面。一些主流的书阐述了诸如面向对象分析、面向对象设计、面向对象编程、设计模式、面向对象的数据结构（XML）、统一建模语言（UML）、面向对象 Web 开发、面向对象移动开发、进阶面向对象编程语言等主题，当然也包括了其他与面向对象编程相关的主题。

然而，许多人仔细研究这些书后，都未曾注意到这些主题都建立在同一个基础之上，即如何以面向对象的方式进行思考。从学生到软件开发专业人员，往往虽然阅读了这些书，但没有花费充分的时间和精力来真正理解代码背后的设计理念。

我认为仅学习一种特定的开发方法、一种编程语言或者一组设计工具并不能说明学会了面向对象这一概念。简单来说，以面向对象方式编程就是一种思考方式。本书就讨论这种面向对象的思考过程。

把面向对象的思考过程从语言、开发实践以及工具中剥离出来并不是一个简单的任务。在学习面向对象这一理念时，往往要求先深入学习一门编程语言。例如，很多年以前，大量的 C 语言程序员在没有直接接触面向对象概念之前，就开始通过 C++ 语言来了解面向对象。其他软件专家第一次接触面向对象则是在演示文稿中使用 UML 创建对象模型。他们也没有直接学习面向对象的概念。即使到现在，互联网作为商业平台的几十年后，编程书籍以及专业的培训材料并没有先介绍面向对象这一概念。

学习面向对象的概念与学习使用面向对象语言进行编程有着巨大差异，理解这点很重要。我在编写本书第 1 版前就意识到了这点。当我阅读 Craig Larman 的文章《What the UML

Is-and Isn't》时，他指出：

但是，在软件开发工程和UML绘图语言领域，读写UML标记的能力有时候好像等同于面向对象的分析和设计能力。事实当然并非如此，后者比前者更加重要。因此我推荐先学习面向对象分析和设计的相关教学资料，它优先于学习使用UML标记的相关工具。

因此，尽管学习一门建模语言是非常重要的步骤，但先学习面向对象的技能更加重要。如果未完全理解面向对象概念前就学习UML，这就像还未了解任何与电路相关的知识就开始学习电路图一样。

学习编程语言也有相同的问题。如前所述，很多C语言程序员还未直接了解任何面向对象的概念，就想通过使用C++语言来达到面向对象的境界。在面试中经常会出现这样的情况，很多自诩C++程序员的开发人员只是会使用C++编译器的C程序员。甚至现在，诸如C#.NET、VB.NET、Objective-C以及Java等语言已经相当普及了，工作面试中的一些关键问题可以迅速暴露出这些程序员缺乏面向对象的思想。

Visual Basic的早期版本并不是面向对象的。C语言也不是面向对象的。而C++在设计时就向后兼容C语言。因此，使用C++编译器编写只含C语言语法的程序，而放弃使用C++的面向对象功能是完全可能的。Objective-C是标准ANSI C语言的一个扩展。更糟糕的是，程序员可能使用勉强够用的面向对象功能把程序写成了既不是面向对象的也不是面向过程的四不像产品。

因此，在学习使用面向对象的开发环境之前，先学习基本的面向对象概念至关重要。与其直接学习一门编程语言（比如Objective-C、VB .NET、C++、C# .NET或Java）或建模语言（如UML），还不如把时间花在学习面向对象的思考过程上。

我在使用C语言编程很多年后，于20世纪80年代后期开始参加Smalltalk语言的学习课程。当时我所在的公司认为公司的软件开发人员应该学习这个极具前途的技术。老师授课时说面向对象的范式是全新的思维方式（事实上它从20世纪60年代就已经萌芽了）。他接着说，虽然几乎我们所有人都是很优秀的程序员，但还有10% ~ 20%的人从来没有按照面向对象的方式做事。如果该说法确实正确，那么很可能是因为很多优秀的程序员从没有花时间进行编程范式的转变，没有深入学习面向对象概念。

第4版中的新增内容

正如在前言中经常提及的一样，第1版中我的愿景仅仅围绕概念本身，而不是具体的新兴技术。尽管我在第2版、第3版以及第4版仍然坚持该目标，但也引入了几个章节讲述关于应用程序的主题，这些主题与面向对象概念契合度很高。第1 ~ 10章涵盖了基本的面向

对象概念，第 11 ~ 15 章将这些概念应用到了一些常用的面向对象技术中。例如，第 1 ~ 10 章提供了面向对象的基础课程（比如封装、多态、继承等），而第 11 ~ 15 章则介绍了一些实际应用。

第 4 版相对于之前的版本拓展了很多主题。以下列出了改进及更新的主题：

- 移动设备开发，包括手机应用、移动应用以及混合开发等。
- iOS 环境下的 Objective-C 代码示例。
- 使用 XML 及 JSON 实现可读性强的数据交换。
- 使用 CSS、XSLT 等技术实现数据渲染与转换。
- Web 服务，包括简单对象访问协议（SOAP）、RESTful Web 服务等。
- 客户端 / 服务器端技术以及封送对象。
- 持久化数据和序列化对象。
- 很多章节都扩充了代码示例，在 Pearson 的网站可以在线查看用 Java、C# .NET、VB .NET 及 Objective-C 编写的代码示例。

目标读者

本书介绍了面向对象的基本概念，并使用代码示例进一步解释概念。最难平衡的一点在于保证代码既具有表现力又能充分阐述概念。本书的目标是让读者即使不用编译器也能理解面向对象的概念和技术。当然，如果你想使用编译器，我们也提供了代码供执行和探索。

本书目标读者包括业务经理、设计师、开发工程师、程序员、项目经理，以及其他想了解面向对象的人士。阅读本书会为你建立一个稳固的基础，有助于阅读其他更与面向对象概念相关的高端书籍。

在这些更高端的书中，我最喜欢的一本是《Java 面向对象设计》(Object-Oriented Design in Java)，作者为 Stephen Gilbert 及 Bill McCarty。我很喜欢这本书的讲述方式，并且将其作为我教授的面向对象概念课程的教材。我在本书中多次引用了《Java 面向对象设计》，所以我推荐你看完本书后可以继续阅读《Java 面向对象设计》。

我认为很优秀的其他书还包括《Effective C++》，作者为 Scott Meyers；《Classical and Object-Oriented Software Engineering》，作者为 Stephen R. Schach；《Thinking in C++》，作者为 Bruce Eckel；《UML Distilled》，作者为 Martin Fowler；以及《Java Design》，作者为 Peter Coad 和 Mark Mayfield。

当我在技术社区以及大学中教授编程和 Web 开发的入门课程时，我很快发现大多数程序员容易学会语言语法，但很难掌握该语言面向对象的本质。

本书讲述方式

显然，我坚定地认为熟悉面向对象的思考过程优先于学习编程语言或建模语言。本书有相应的代码示例及 UML 图，但阅读本书并不需要掌握一门具体的编程语言或 UML。我已经说过了要先学习概念本身，那为什么本书有如此多的 Java、C# .NET、VB .NET 以及 Objective-C 代码和 UML 图？首先，它们可以很好地展示面向对象概念。其次，它们对于面向对象的使用方式是极其重要的，有助于介绍和展示如何使用面向对象。关键在于不要过分关注 Java、C# .NET、VB .NET 及 Objective-C 或 UML，而应当使用它们来帮助理解面向对象的深层理念。

注意，在理解类及其自身的属性和方法时，我非常喜欢使用 UML 类图实现可视化。事实上，本书中只用到了 UML 组件中的类图。我认为 UML 类图提供了一种强大的方式来模拟对象模型的概念本质。我依旧使用对象模型作为教学工具来展示类设计以及类之间的关系。

本书中的代码示例阐述了诸如循环和函数之类的概念。然而，理解代码本身并不是理解面向对象概念的先决条件。如果你想了解语言本身的更多细节，那么有一本讲解语言语法的书放在手边则非常有用。

我不能过分强调本书不讲述 Java、C# .NET、VB .NET、Objective-C 或 UML。你可以自学这些知识。我希望本书能激发你对其他面向对象概念的兴趣，比如面向对象分析、面向对象设计以及面向对象编程。

本书中的源码

本书中提到的示例代码可以在华章网站（www.hzbook.com）下载。

目　录 *Contents*

第 1 章 Chapter 1

面向对象的概念简介

很多程序员其实并不知道在 20 世纪 60 年代就已出现面向对象的软件开发方式。尽管受欢迎的面向对象的编程语言（例如 Smalltalk 和 C++）已被广泛使用，但直到 90 年代中后期面向对象范式才开始快速发展。

面向对象方法论的兴起恰逢互联网作为商业及娱乐平台之际。总之，对象借助网络能良好协作。后来显然互联网存活下来，而面向对象的技术已经在开发新的基于 Web 的技术中占据了重要位置。

本章标题是“面向对象的概念简介”。标题中关键词是“概念”而非“技术”。在软件行业，技术变迁非常快，而概念则是逐步演进。我使用单词“演进”是因为尽管它们保持相对稳定，但也在变化。这正是需要关注这些概念的原因。尽管它们相对稳定，但经常被重新反思，也会导致一些很有意思的讨论。

从 90 年代中后期的最原始的浏览器到如今移动、手机、Web 应用占据主导地位，通过这些多样的行业技术的发展很容易追溯过去 20 年间的演化。甚至如今我们正在探索混合软件，而新的开发技术就在下个拐角。在整个旅程中，每一步都存在面向对象的概念。这也是为什么本章主题如此重要，因为从 20 年前到现在仍在使用这些概念。

1.1 基本概念

本书主要目标是让你学会思考如何将面向对象概念应用于面向对象的系统设计中。历史上定义面向对象的语言拥有以下特点：*封装*（encapsulation）、*继承*（inheritance）和*多态*（polymorphism）。因此，如果设计一门语言时没有完全实现以上特性，那么通常我们认为该

语言不是完全面向对象的。即使实现了这三点，我也往往会加入组合特性。因此，我认为面向对象的概念如下：

- 封装
- 继承
- 多态
- 组合

本书接下来会详述这些特性。

从本书第 1 版开始，我一直在努力解决的问题是这些概念如何直接与当前的设计实践关联起来，因为设计实践始终在变化当中。例如，在面向对象设计中使用继承总是饱受争议。继承是否实际上破坏了封装？（稍后的章节会讨论这一主题。）甚至现在，很多开发人员都试图尽量避免使用继承。

我的方式是始终围绕概念来学习。无论你是否使用继承，你至少需要理解什么是继承，这会保证你的设计决策是有据可依的。正如在介绍中提及的那样，本书目标读者是希望学习基本的面向对象概念的总体介绍的人。请记住这一点，本章中我会展现基本的面向对象概念，希望读者能在做重要的设计决策前打下一个坚实的基础。本章涵盖了这些概念的基本知识，随后的章节中的主题也会讨论这些概念的细节。

1.2 对象及遗留系统

在面向对象成为主流之前，开发人员面临的问题一直就是如何将新的面向对象技术与现有的系统集成起来。面向对象与结构化（或命令式）编程之间拥有明显的界限，而结构化编程当时则是主流的开发范式。而我始终反对面向对象和结构化编程是不兼容的这一观点，因为我认为面向对象与结构化编程绝不是互斥的。它们是互补的，因为对象可以与结构化代码很好地集成。直到现在我还经常听到这样的问题：你是一个面向过程的程序员还是面向对象的程序员？我会毫不犹豫地说："我都是。"

同样，面向对象的代码并不意味着完全替代结构式代码。很多非面向对象的遗留系统（即现存的较老的系统）仍在正常工作，所以为什么要冒着潜在的风险来替换它们？大多数情况下，你不应当替换它们，至少不应当使用这样的理由。使用非面向对象的代码编程并不存在本质上的错误。然而，新一代开发者则会倾向考虑使用面向对象技术（有些情况下也只能这样做）。

过去的 20 年间，面向对象的开发领域一直在平稳显著地增长，全球社会对网络（比如互联网及移动基础设施）的依赖有力地帮助了面向对象技术飞速发展，甚至成为业界主流。浏览器与移动应用之间的海量交易开辟了全新的市场。在这个市场中很多软件都是全新的，几乎未受到遗留系统的影响。即使需要与遗留系统打交道，也可以使用包装对象将遗留系统包装起来。

包装对象

包装对象是指在面向对象的类中包含其他代码。例如，你可以将结构化代码（比如循环和条件）包装到一个对象内部，使其行为就像对象一样。你也可以使用包装对象来封装特性，比如安全特性、不可移植的硬件特性等。第 6 章讲述了如何包装结构化代码。

如今，软件开发最有趣的领域之一是遗留代码与移动及 Web 系统的集成。大部分情况下，一个移动 Web 前端最终会连接到大型机中的数据。市场对既拥有大型机开发经验又同时拥有移动 Web 开发经验的开发人员的需求量非常大。

我们在日常生活中有大量使用对象的经历，甚至有时候自己都没意识到在使用对象。当你打电话、使用你的家庭财务系统、玩电脑游戏（以及很多其他情况）时，这都是使用对象的经历。电子高速公路本质上已经成为基于对象的高速公路。随着业务越来越多地使用移动网络，这些业务也越来越多地使用对象，因为电子商务所用技术大多都是面向对象的。

移动网络

无需置疑，互联网的出现为面向对象技术的推广起到了极大的推动作用。这是因为在网络中很适合使用对象。尽管互联网是这场变革的领导者，而移动网络也加入进来并扮演着重要的角色。本书中，术语移动网络（mobile web）适用于移动应用开发和 Web 开发的上下文中。术语混合应用（hybrid app）则用于呈现在 Web 设备和移动设备的浏览器中的应用程序。

1.3　过程式编程与面向对象编程

在我们深入了解面向对象开发的优势之前，先考虑一个更基本的问题：究竟什么是对象？这既是一个复杂的问题，也是一个简单的问题。它复杂是因为学习任何一种软件开发方法论都非易事。它简单是因为人们已经在按对象的方式进行思考。

例如，当你看到一个人，你会把他看作一个对象。一个对象由两部分组成：属性及行为。一个人具有属性，比如眼睛颜色、年龄、身高等。一个人也有行为，比如行走、讲话、呼吸等。对象的基本定义是一个包含了数据和行为的实体。

单词和（both）是面向对象编程与其他编程范式的核心区别。例如在过程式编程中，代码放置在完全不同的函数或程序中。如图 1-1 所示的理想情况下，程序即为“黑盒”（black box），接收输入，然后输出。数据放置在单独的结构中，通过函数或程序进行操作。

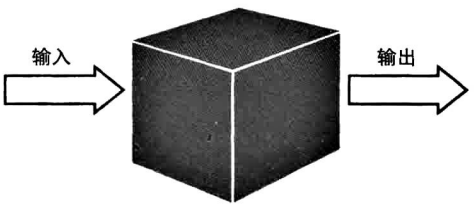


图 1-1　黑盒

面向对象与过程式编程的不同之处

在面向对象的设计中，属性及行为包含在单个对象中，而在过程式或结构式设计中，属性和行为通常是分开的。

虽然面向对象设计越来越流行，但有个现实问题最初阻碍了面向对象设计的推广，那就是很多非面向对象的系统还在正常工作。因此，没有任何商业动力来将其改造为面向对象的系统。任何熟悉计算机系统的人都知道不管对系统的修改有多小，都极可能引发灾难性的后果。

同样的情形也阻碍了采纳面向对象的数据库。随着面向对象开发方式的发展，貌似面向对象的数据库将替代关系型数据库。然而，这绝不会发生。关系型数据库得到了大量的商业投资，而现有的关系型数据库可以正常工作，这一重要因素阻碍了向面向对象的数据库的转换。将关系型数据库转换为面向对象的数据库花销巨大又充满风险，但却没有一个足够令人信服的理由来支持转换。

事实上，企业已经找到了一个折中方案。当今很多软件开发实践糅和了几种开发方法论，比如面向对象和面向过程方法论。

如图 1-2 所示，结构化编程中数据往往与程序分离，而且数据是全局的，所以在你的代码作用域之外依然可以很容易修改数据。这意味着对数据的访问是失控的，并且不可预期（因为很多功能都可以访问全局数据）。而且，由于你无法控制谁能访问数据，那么测试和调试将变得更加困难。对象通过将数据和行为组合到一个完整的包中从而解决了这些问题。

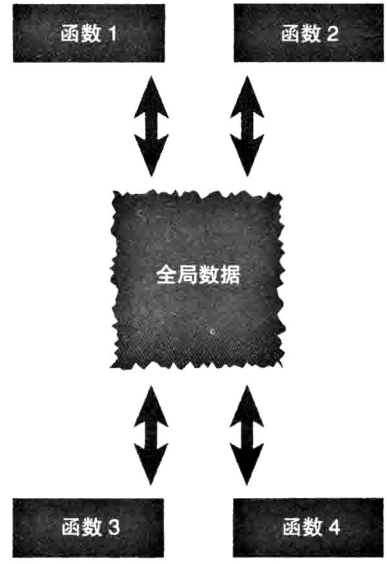


图 1-2 使用全局数据

恰当的设计

如果设计是恰当的，那么在面向对象模型中则不会有诸如全局数据的元素。事实上，在面向对象系统中具有很高的数据完整性。

对象并不会完全替代其他的软件开发范式，它是一种进化。结构化的程序有复杂的数据结构，比如数组等。C++ 有结构体，结构体具有对象（类）的很多特性。

然而，对象比数据结构体及原始数据类型（比如整型和字符串）更丰富。对象包含整型和字符串之类的实体，用于表示属性，对象也包含方法，用于表示行为。在对象中，方法用于操作数据及其他行为。更重要的是，你可以控制对对象中成员（包括属性及方法）的访问。这意味着某些成员（属性和方法）可以对其他对象隐藏起来。例如，名为 `Math` 的对象包含两个整数，叫作 `myInt1` 和 `myInt2`。同时，`Math` 对象也包括了必要的方法存取 `myInt1` 和 `myInt2` 的值。它也包括一个叫作 `sum()` 的方法对这两个整数求和。

数据隐藏

在面向对象的术语中，数据表现为属性，行为表现为方法。限制访问具体属性和（或）方法的行为叫作数据隐藏。

将属性及方法合并到同一个实体中，在面向对象中将这种方式叫作封装（encapsulation）。我们可以控制对 Math 对象的数据的访问。比如将这些整数定义为禁止对外访问，其他函数无法操作整数 myInt1 和 myInt2，只有 Math 对象才行。

合理的类设计指导

请注意完全有可能创建一个设计差劲的面向对象的类，该类没有对自身属性的访问进行任何限制。即使你使用面向对象设计也可能会设计出烂代码，使用其他的编程方法论都会有这样的可能性。请坚持合理的类设计指导（详见第 5 章）。

另一个名为 myObject 的对象如何获取 myInt1 和 myInt2 的总和？它会询问 Math 对象，即 myObject 会给 Math 对象发送一个消息。图 1-3 展示了这两个对象如何通过方法进行通信。该消息其实就是调用 Math 对象的 sum 方法，sum 方法把值返回给了 myObject。巧妙之处在于，myObject 无需知道总和是如何计算出来的（尽管我认为它能猜到）。使用这种设计方法，你可以修改 Math 对象计算总和的方式而无需对 myObject 做任何修改（这意味着获取总和的方式无需改变）。你需要的是总和，而不用关心它是如何计算出来的。

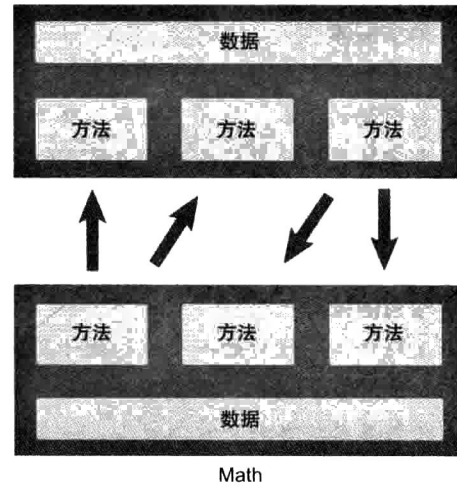


图 1-3　对象之间的通信

我们可以通过一个简单的计算器示例来演示该概念。当使用计算器求和时，你只需使用计算器提供的接口，即键盘和 LED 显示器。计算器有一个求和方法，当你按下正确的按键序列就会调用它。你会得到正确的返回结果，然而你无需知道结果是如何计算出来的，你既不用关心电路控制，也不用关心算法。

求和不是 myObject 的责任，它是 Math 对象的责任。由于 myObject 可以访问 Math 对象，它可以发送相应的信息并获取正确的结果。通常，对象不应当操作其他对象的内部数据（即 myObject 不应当直接修改 myInt1 和 myInt2 的值）。而且随着持续开发，最好多构建一些具有明确任务的小对象，而不应该构建包含很多方法的大对象。

1.4　由面向过程开发过渡到面向对象开发

现在我们对面向过程与面向对象开发技术有了一定的了解。接下来我们继续深入学习。

1.4.1　过程式编程

过程式编程通常会将系统的数据与对数据的操作分离开来。例如，如果你想通过网络发

送信息，只发送相关数据（见图 1-4），而期望网络管道另一端的程序知道如何处理该数据。换句话说，客户端和服务器端要对数据传输建立起一种握手约定。在该模型中，通过网线传输的只有数据，没有代码。

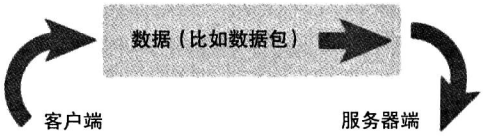


图 1-4 通过网线传输数据

1.4.2 面向对象编程

面向对象编程的最大优势是数据和对数据的操作（代码）都被封装在一个对象中。例如，当通过网络传输对象时，整个对象（包括里面的数据和行为）都会一起被传输。

单个实体

尽管理论上认为传输了单个实体，但大多数情况下行为本身不会被发送，因为两端都有行为代码的副本。然而，理解以整个对象作为单个实体通过网络传输是相当重要的。

在图 1-5 中，通过网络传输 Employee 对象。

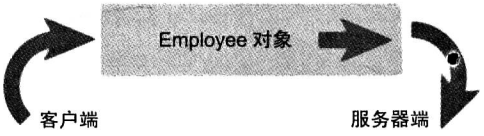


图 1-5 通过网线传输对象

恰当的设计

反应恰当的设计的一个好的例子是通过浏览器加载对象。通常，浏览器并不会提前知道对象是干什么的，因为浏览器并未拥有该对象的任何代码。当对象被加载后，浏览器执行对象中的代码，并使用对象中包含的数据。

1.5 究竟什么是对象

对象是面向对象程序中的基础材料。使用面向对象技术的程序本质上是对象的集合。举例来说，一个公司系统包含了代表公司雇员的对象。接下来的小节描述了这些对象的数据和行为。

1.5.1 对象数据

存放在对象中的数据代表了该对象的状态。在面向对象术语中，数据被称为属性。在图 1-6 的例子中，雇员的属性可以是社会安全号码、生日、性别、电话号码等。属性包含的信息将不同对象区分开来，在本例中是把不同的雇员区分出来。本章中讨论类时会讲述属性的细节。

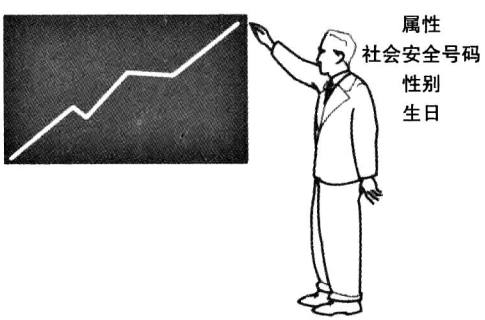


图 1-6 雇员属性

1.5.2　对象行为

对象的行为表示对象可以做什么。在过程式语言中，可以通过过程、函数及子程序定义行为。在面向对象程序设计术语中，这些行为包含在方法中，你可以通过发送消息的方式来调用方法。在雇员例子中，考虑到雇员对象需要的行为之一是对属性值进行存取，所以每个属性会有相应的方法，比如 `setGender()` 和 `getGender()`。当另一个对象需要该信息时，可以给雇员对象发送一个消息来询问其性别。

毫不奇怪，在本书第 1 版发布时，应用程序中已经出现了取值方法（getter）和赋值方法（setter），当然还有很多其他的面向对象技术。而且很多技术都与数据相关。通过第 11 章及第 12 章，我们可以看到如何使用面向对象的方式组织数据。使用对象最强大、最有趣的一点在于，数据与行为是一个整体，而不是与行为代码割裂。

XML 的出现并不全为了以可移植的方式表示数据。它也有一种替代方式来让代码轻松访问数据。在 .NET 技术中，取值方法和赋值方法都被认为是数据自身的属性。

例如，Java 中定义一个名为 Name 的属性如下所示：

```
public String Name;
```

对应的取值方法和赋值方法如下所示：

```
public void setName (String n) {name = n;};
public String getName() {return name;};
```

现在，使用 C#.NET 定义一个名为 Name 的 XML 属性，代码如下所示：

```
Private string strName;
public String Name
{
    get
    {
        return this.strName;
    }
    set
    {
        if (value == null) return;
        this.strName = value;
    }
}
```

在这种方式中，取值方法和赋值方法实际上是属性 Name 的属性。

不论采用什么方式，目的都是一致的，即控制对属性的访问。本章中，我想先聚焦于取值方法和赋值方法的概念本质。而在第 11 章及以后的章节中讲述面向对象数据时会有更多关于属性的知识。

取值方法和赋值方法

取值方法和赋值方法的理念就是数据隐藏。因为其他对象不应该直接操作另一个对象中的

数据，而取值方法和赋值方法提供了对对象数据的访问控制。取值方法和赋值方法有时分别被称为访问方法（accessor method）和设值方法（mutator method）。

注意，我们只展示了这些方法的接口，而不是实现。以下几点列出了用户需要知道如何实际使用这些方法：

- 方法名
- 传给方法的参数
- 方法的返回值

请看图 1-7 演示的行为。

图 1-7 中，Payroll 对象包含了一个名为 CalculatePay() 的方法，用于计算具体雇员的薪水。Payroll 对象需要获取该雇员的社会安全号码。为了获取该信息，Payroll 对象需要给 Employee 对象发送一个消息（本例中即调用 getSocialSecurityNumber() 方法）。基本上，这意味着 Payroll 对象调用 Employee 对象的 getSocialSecurityNumber() 方法。Employee 对象接收到该消息并返回请求的信息。

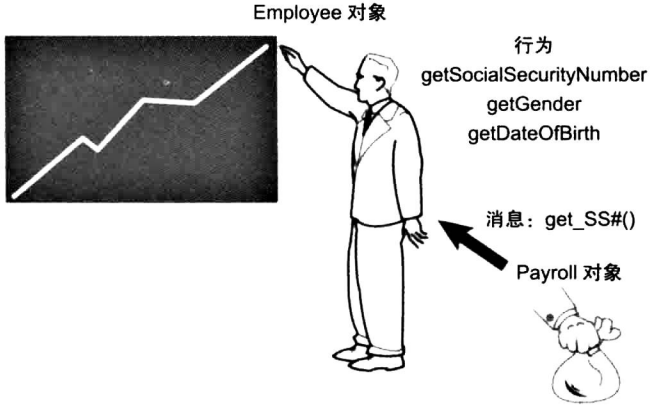


图 1-7 Employee 的行为

图 1-8 进一步展示了以上我们讨论的关于 Employee/Payroll 系统的类图。

UML 类图

由于这是我们看到的第一个类图，所以进行了一些简化，缺乏一些类应该具有的结构（例如构造函数）。不过没关系，第 3 章中我们会详细讨论类图及构造函数。

每个类图由三部分组成：类名、数据（属性）、行为（方法）。图 1-8 中，Employee 类图的属性区域包含 SocialSecurityNumber、Gender 和 DateofBirth。方法区域包含了操作这些属性的方法。你可以使用 UML 建模工具来创建和维护类图以对应真实的代码。

建模工具

可视化建模工具提供了一种方式来使用统一建模语言（unified modeling language，UML）创建和操作类图。对于类图的讨论贯穿本书，第 10 章会指导如何使用该标记语言。UML 类图可以作为一个工具来可视化类以及类与类之间的关系。本书中对 UML 的使用仅限于类图。

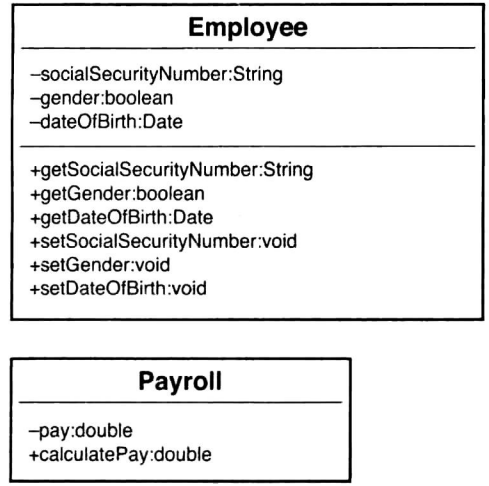


图 1-8　Employee 类和 Payroll 类的类图

本章稍后会介绍类与对象的关系。但现在你可以认为类是创建对象的模板。当对象被创建时，我们看到对象被实例化。因此，如果我们创建三个雇员，实际上是创建了三个 Employee 类的直接实例。每个对象都包含了属性和方法的副本。例如，图 1-9 中，一个叫作 John 的对象（John 是该对象的标识）自身拥有 Employee 类中所有的属性和方法的副本。叫作 Mary 的雇员类也拥有自己独立的属性和方法的副本。这两个对象各自拥有 DataOfBirth 属性和 getDateOfBrith 方法的副本。

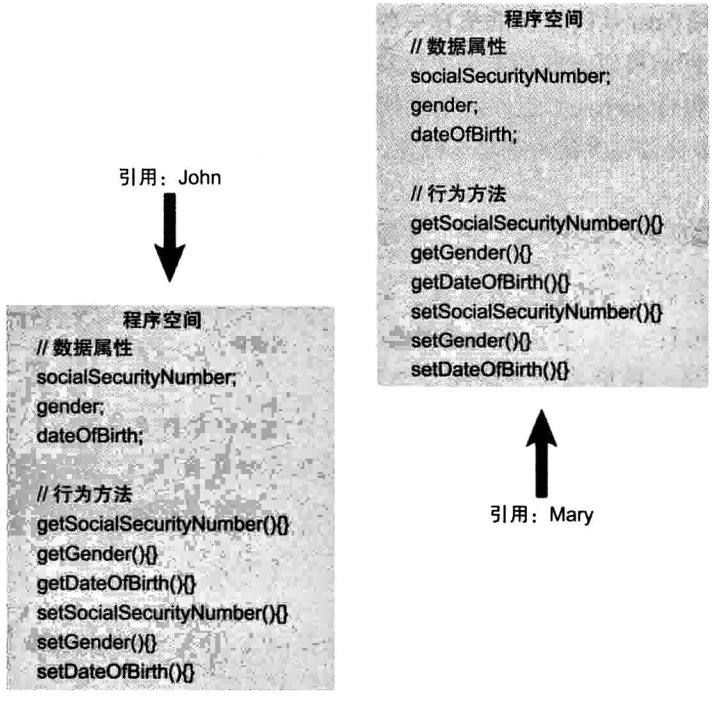


图 1-9　程序空间

实现问题

请注意没必要为每个对象的每个方法都实现一个物理副本。其实，每个对象指向了同一个实现。不过这个问题可以交由编译器 / 操作系统平台来解决。从概念上来说，你可以认为对象是完全独立的，各自拥有独立的属性和方法。

1.6 究竟什么是类

简单地说，类是对象的蓝图。当你实例化一个对象时，你基于类来构建这个对象。事实上，尝试解释类和对象实际上是一个先有鸡还是先有蛋的问题。描述类时不使用术语对象是非常困难的，反之亦然。例如，个人自行车是一个对象。然而，有人已经创建了蓝图（就是类）来建造这个自行车。在面向对象的软件中，对比先有鸡还是先有蛋的问题的不同之处在于，我们知道对象和类哪个先出现（类先出现）。如果没有类，对象无法被实例化。因此本节很多概念与本章之前的很多内容都很相似，特别是我们讨论属性和方法时。

为了解释类和方法，我们可以拿关系型数据库来举例。在数据表中，表自身的定义（即字段、描述和数据类型）是类（即元数据），而对象则是表中的行（即数据）。

本书关注于面向对象的软件开发，但不局限于具体的语言实现（比如 Java、C#、Visual Basic .NET、Objective C 或者 C++）。但经常使用代码示例可以帮助大家理解，所以必要时我会在本书中使用 Java 代码来帮助解释一些概念。而且每章的结尾会包含本章代码示例的 C# 版。在出版社网站可以获取到本书中出现的大部分代码。我们也在网上提供了 Java、C# .NET、VB .NET 和 Objective C 版本的示例代码。

接下来的小节会描述类的一些基本概念，以及它们之间的关系。

1.6.1 创建对象

类可以认为是对象的模板或者模具，正如图 1-10 中的饼干模具一样。类用来创建对象。

类可以认为是一种更高级别的数据类型。例如，你创建了一个整数或浮点数：

```
int x;
float y;
```

你也可以使用预先定义的类创建一个对象：

```
myClass myObject;
```

本例中，通过名称可以明显看出 myClass 是类，而 myObject 是对象。

请记住每个对象有自己的属性（数据）和行

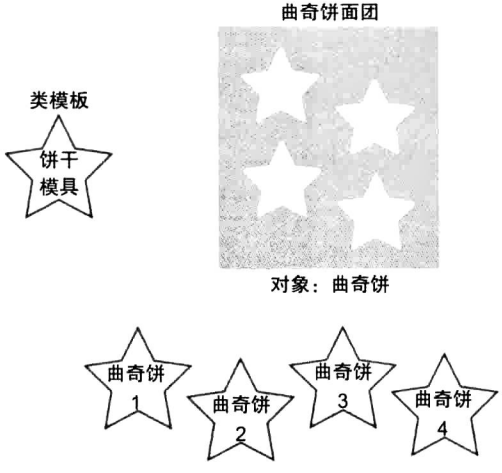


图 1-10 类模板

为（函数或惯例）。类定义了使用该类创建的所有对象具有的属性和行为。类是一块代码。可以单独分发使用类实例化的对象，也可以将类作为程序库的一部分进行分发。因为对象从类中创建，所以类必须定义对象的基础材料（即属性、行为和消息）。总之，在创建对象之前必须先设计一个类。

例如，以下是对 Person 类的定义：

```
public class Person{

    //属性
    private String name;
    private String address;

    //方法
    public String getName(){
        return name;
    }
    public void setName(String n){
        name = n;
    }

    public String getAddress(){
        return address;
    }
    public void setAddress(String adr){
        address = adr;
    }

}
```

1.6.2　属性

正如你看到的一样，类的数据通过属性来表示。每个类必须定义属性，用来存放该类实例化的每个对象的状态。在上一小节中的 Person 类的例子中，Person 类定义了 name 和 address 两个属性。

> **访问控制**
>
> 当数据类型或方法被定义为 public 时，其他对象可以直接访问它。当数据类型或方法被定义为 private 时，只有自身对象可以访问它。另一个访问修饰符 protected 只允许相关对象访问，第 3 章会讲述该点。

1.6.3　方法

正如你本章之前学到的一样，方法实现了一个类需要的行为。从某个类实例化的每个对象都拥有该类中定义的方法。有些方法可以调用其他对象（或信息）的行为，而有些方法则提供该类底层的内部行为。内部行为是私有方法，不允许其他对象访问。在 Person 类中，

行为有 getName()、setName()、getAddress()、setAddress()。其他对象可以通过这些方法检查或修改该对象属性值。这是面向对象的系统中常用的技术。在任何情况下，对对象中属性的访问应该由该对象自身控制，任何一个对象都不应该直接修改其他对象的属性。

1.6.4 消息

消息是对象之间的通信机制。例如，当对象 A 调用了对象 B 的一个方法，对象 A 向对象 B 发送了一个消息。对象 B 的响应由其返回值定义。其他对象只能调用一个对象的公共方法，而无法调用其私有方法。以下代码验证了该点：

```
public class Payroll{

    String name;

    Person p = new Person();

    String = p.setName("Joe");

    ... code

    String = p.getName();

}
```

在该例子中（假设 Payroll 对象已经被实例化），Payroll 对象向 Person 对象发送了一个消息，目的是通过 getName() 方法获取 name。注意不要过度关心实际的代码，这里只是为了演示。本书随后会展示详细的代码。

1.7 使用类图作为可视化工具

这些年来，开发出了很多工具和模型技术帮助我们设计软件系统。开头已经说过，我使用 UML 类图来帮助大家学习本书。讨论 UML 细节已经超出了本书的范围，我们只使用 UML 类图来演示我们构建的类。其实本章中我们已经使用了类图。图 1-11 显示了本章之前讨论过的 Person 类的类图。

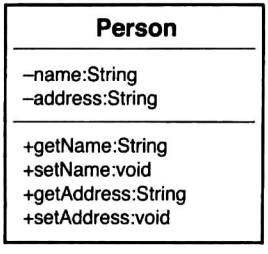


图 1-11 Person 类图

正如我们之前看到的那样，请注意属性和方法是分开的（属性在上面，方法在下面）。随着我们逐渐深入研究面向对象设计，类图会越来越精密，并且会表达不同类之间的交互关系。

1.8 封装和数据隐藏

使用对象的一个显著好处是对象无需暴露它的所有属性和行为。在出色的面向对象设计

（至少通常认为是好的设计）中，对象仅暴露必要的接口来和其他对象进行交互。除了如何使用该对象，其他细节都应当对其他对象隐藏起来。

封装是基于对象既包含属性也包含行为这一事实。数据隐藏是封装的主要部分。

例如，一个计算数字的平方值的对象必须提供一个接口来展示结果。然而，调用方无需知道该对象计算平方值的内部属性和算法。健壮的类会一直拥有良好的封装。下一小节中，我们会讲解接口和实现的概念，这是封装的本质。

1.8.1　接口

我们可以看到接口定义了对象之间通信的基本手段。每个类设计接口规格来保证对象能被正确实例化和操作。必须向对象提供的接口发送消息来使用对象暴露的任何行为。接口需要完整描述类与类之间的交互。在大多数面向对象的语言中，访问修饰符指定为 public 的方法属于接口。

私有数据

为了实现数据隐藏，必须将所有属性声明为 private。属性绝不是接口的一部分。只有 Public 方法是类接口的一部分。将属性描述为 public 破坏了数据隐藏这一概念。

我们回过头看一个使用的例子；计算一个数字的平方值。在这个例子中，包括两个接口：

- 如何实例化一个 Square 对象。
- 如何传递给该对象一个值，并获取返回的平方值结果。

正如在本章前面讨论的一样，如果用户需要访问一个属性，我们会创建一个方法（即 getter，又称为取值方法）来返回该属性值。然后用户想要获取该属性值，只需调用取值方法即可。在这种方式中，拥有该属性的对象可以控制访问该属性。这种控制非常重要，特别对安全、测试和维护而言。如果你可以控制对属性的访问，当出现了问题，无需追踪所有可能修改了该属性的代码片段，我们只需要修改一个地方（即 setter，又称为赋值方法）。

基于安全的原因，你需要防止不受控的代码对诸如密码和个人信息等数据进行修改和获取。

类接口与方法签名

请注意类有接口，方法也有接口。不要混淆这两个接口的概念。类的接口是公共方法。你使用方法的签名来调用这些公共方法。方法的签名主要由方法名和它的参数列表组成。稍后会详细介绍方法签名这一概念。

1.8.2　实现

只有公共属性和方法被认为是接口。用户通过类接口与对象交互时，不应该看到该对象任何内部实现的部分。任何被定义为私有的东西对用户都是不可访问的，这些私有资源属于

该类的内部实现。

在之前的例子中，只隐藏了 Employee 类的实例的属性。很多情况下，也应该隐藏某些方法，不作为接口的一部分。上节的求平方值的例子中，用户不应该关心平方值是如何被计算出来的，只要答案正确即可。我们可以修改实现，但不应该影响用户的代码。例如，提供该计算器的公司可以改变算法（很可能因为该算法更高效）而不会影响结果。

1.8.3 接口 / 实现范式的一个真实示例

图 1-12 使用了真实的对象演示了接口 / 实现范式，这里没有使用代码。烤面包机需要电。为了得到电，烤面包机的插头必须插到电源插座，电源插座就是接口。所有需要获取电力的烤面包机需要实现一个插头，要与电源插座规格适配。这是烤面包机和电力公司（事实上是能源行业）之间的接口。电力设备会关心实际的实现，而烤面包机则无需担心。事实上，对于所有面包机而言，电力实现可能是原子能设备或本地发电机。基于这样的模型，一个装置只要满足接口规格，就可以获取电力，如图 1-12 所示。

1.8.4 接口 / 实现范式的模型

我们再拿 Square 类来举例。假设你正在编写一个计算整数平方值的类。你必须提供接口和实现，也就是说必须定义一种方式使用户能调用并获取平方值，也要实现计算平方值。然而，用户不应该知道任何详细的实现。图 1-13 展示了一种实现方式。注意在该类图中，加号（+）表示 public 访问修饰符，而减号（–）表示 private 访问修饰符。即你可以通过给方法指定加号签名的方式来定义接口。

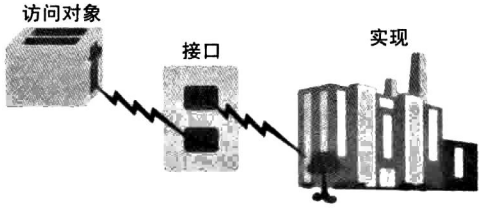


图 1-12 发电厂示例

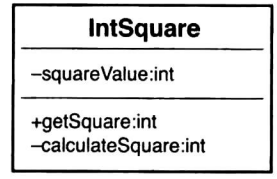


图 1-13 平方类

该类图对应的代码如下：

```
public class IntSquare {

    // 私有属性
    private int squareValue;

    // 公共接口
    public int getSquare (int value) {

        SquareValue =calculateSquare(value);
```

```
        return squareValue;

    }

    // 私有实现
    private int calculateSquare (int value) {

        return value*value;

    }
}
```

注意该类中用户唯一能访问的元素就是公共方法 getSquare。该公共方法就是接口。求平方值算法的实现在私有方法 calculateSquare 中。请注意属性 SquareValue 是私有的，因为用户无需知道该属性的存在。因此对象隐藏了一部分实现，只暴露了用户需要与之交互的接口。与使用该对象无关的细节则对其他对象是隐藏的。

如果要修改实现，比如你想使用语言内置的平方值函数，你不需要修改接口。下面代码使用了 Java 库的方法 Math.pow，其提供了同样的功能，但接口依然是 calculateSquare：

```
// 私有实现
private int calculateSquare (int value) {

    return = Math.pow(value,2);

}
```

用户使用相同的接口会得到相同的功能，但具体实现却变了。如果你编写处理数据的代码时可以利用这一点。例如，你可以把数据存储从文件转移到数据库中，而不必强制用户改变任何程序代码。

1.9 继承

面向对象程序设计的最强大的功能之一就是代码重用。结构化设计提供的代码重用非常受限。你可以编写一个功能块，然后多次重用它。但是面向对象的设计更进一步，允许你定义类之间的关系，通过组织和识别不同类之间的共性，不仅可以实现代码重用，也可以指导设计。继承是实现该功能的主要手段。

继承允许一个类继承另一个类的属性和方法。我们可以通过抽象公共属性和行为来创建新类。

面向对象程序设计中的一个主要设计问题就是识别多个类的共性。例如，假设你有一个 Dog（狗）类和一个 Cat（猫）类，这两个类都有一个属性来表示眼睛颜色。在过程化模型中，Dog 和 Cat 的代码中都会包含这个属性。在面向对象的设计中，可以将颜色属性上移到一个名为 Mammal（哺乳动物）的类中。该类也会包含一些其他的公共属性和方法。在本例中，

Dog 类和 Cat 类都继承自 Mammal 类，如图 1-14 所示。

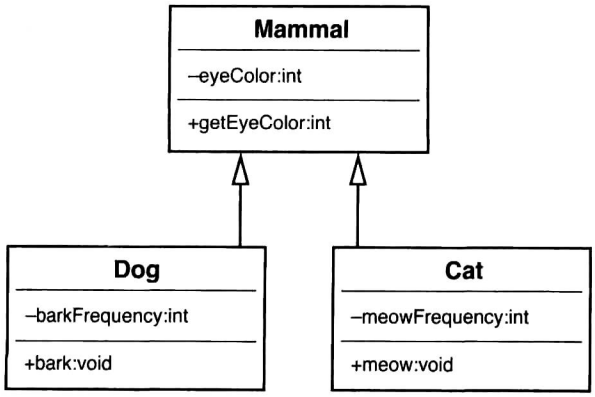


图 1-14　Mammal 继承体系图

Dog 和 Cat 类都继承自 Mammal 类。这意味着 Dog 类有以下属性：

```
eyeColor          // 继承自 Mammal
barkFrequency     // 只在 Dog 类中
```

同样，Dog 对象有以下方法：

```
getEyeColor     // 继承自 Mammal
bark            // 只在 Dog 类中
```

当 Dog 或 Cat 对象被实例化时，它包含了自身类的所有东西，也包含了从父类获取到的东西。即 Dog 拥有自身类定义的所有属性，也包含了从 Mammal 类继承过来的属性。

1.9.1　超类和子类

超类，也称为父类（有时候也叫作基类），包含了继承自它的所有类的公共属性和行为。例如，在 Mammal 类的例子中，所有的哺乳动物拥有相似的属性，比如 eyeColor 和 hairColor，也有相似的行为，比如 generateInternalHeat 和 growHair。所有的哺乳动物都有这样的属性和行为，所以没必要在哺乳动物继承树中为每个动物中都重复一遍。重复不仅加重工作量，而且会带来更多问题，因为会引入错误和矛盾。

子类，也称为孩子类（有时被叫作衍生类），是超类的扩展。比如，Dog 和 Cat 类从 Mammal 类继承了所有的公共属性和行为。Mammal 类是 Dog 和 Cat 子类的超类。

继承提供了丰富的设计优势。当设计 Cat 类时，Mammal 类提供了很多需要的功能。通过继承自 Mammal 对象，Cat 已经拥有了成为真正的哺乳动物的所有属性和行为。为了让猫这种哺乳动物更加具体，Cat 类必须拥有猫自身独一无二的属性或行为。

1.9.2　抽象

继承树可能会增长的非常庞大。当 Mammal 和 Cat 类完成后，可以快速添加其他哺乳

动物，比如狗（或者狮子、老虎和熊）。Cat 类也可以作为其他类的父类，比如作为塔罗猫等猫的父类。而 Dog 类也可以作为 GermanShepherd（德国牧羊犬）和 Poodle（卷毛小狗）的父类（见图 1-15）。继承的力量在于它的抽象和组织技术。

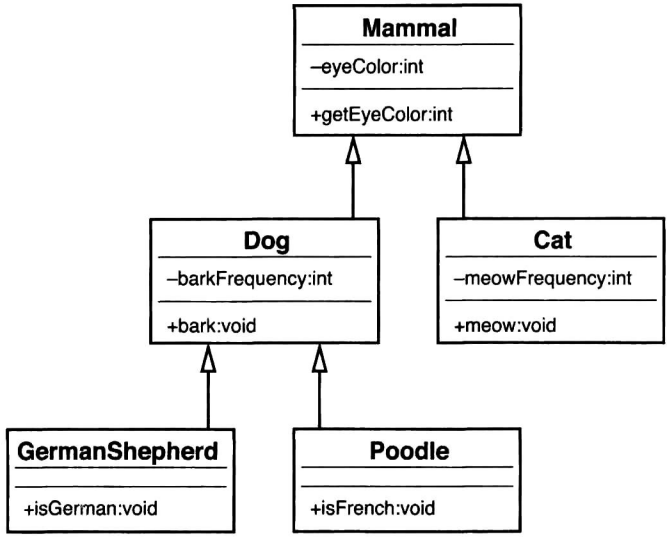


图 1-15　Mammal UML 图

在大多数最新的面向对象的语言中（比如 Java、.NET 和 Objective C），一个类只能有一个父类，然而一个类可以有多个子类。另一些语言中，比如 C++，可以拥有多个父类。前一种情况称为*单继承*，后一种情况称为*多重继承*。

请注意 GermanShepherd 类和 Poodle 类都继承自 Dog 类。每个类只有一个方法。然而由于它们继承自 Dog，所以它们也继承自 Mammal。即 GermanShepherd 和 Poodle 类获取了 Dog 和 Mammal 中所有的属性和方法，同时也拥有它们自身的属性和方法（如图 1-16 所示）。

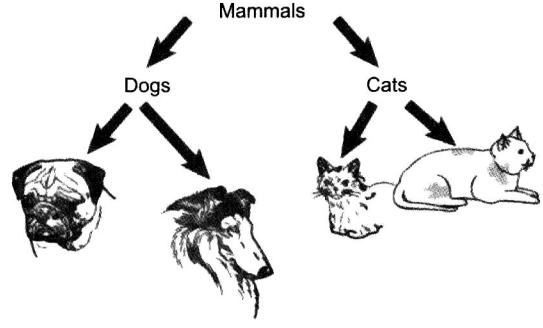


图 1-16　Mammal 继承体系图

1.9.3　is-a 关系

在 Shape（形状）例子中，Circle（圆形）、Square（矩形）和 Star（星形）都直接继承自 Shape。这种关系通常被称为 is-a 关系，因为圆是一个形状，而矩形也是形状。当子类继承自父类时，任何父类能做的事情子类都可以做。即 Circle、Square 和 Star 都是 Shape 的扩展。

在图 1-17 中，每个对象上的名字 Draw 分别代表了 Circle、Star 和 Square 对象的 Draw 方法。当我们设计 Shape 系统时，采用这种方式对各种各样的形状进行标准化。这种方式非常有用，如果我们约定绘制形状时只需调用 Draw 方法即可，而不用管具体是什么形状。那么只要遵守这一约定，无论什么时候绘制形状，只需调用 Draw 方法即可。这正好展示了多态的基本概念，即绘制 Circle、Star 或 Square 是各自对象自身的职责。在如今很多软件应用（比如在绘图和文字处理应用）中都会使用多态概念。

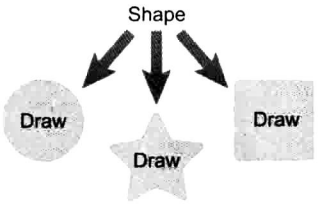


图 1-17 Shape 继承体系图

1.10 多态

多态是一个希腊词，字面上理解为许多形状。尽管多态与继承是紧耦合的关系，但它通常单独作为面向对象技术中最强大的优点之一。当向一个对象发送一个消息时，该对象必须定义一个方法来响应这个消息。在继承体系图中，所有的子类从它们的超类中继承接口。然而，由于每个子类是单独的实体，每个子类需要对同一个消息有单独的应答。

例如，Shape 类有名为 Draw 的行为。当你告诉某人画一个形状时，你被问到的第一个问题是“什么形状?”。没人能绘制一个形状，因为形状是一个抽象概念（事实上 Shape 代码中的 Draw() 方法并不包含实现）。你必须指定一个具体的形状。比如你需要为 Circle 提供具体的实现。虽然 Shape 有一个 Draw 方法，但 Circle 继承了该方法并提供了对该方法的实现。重载（overriding）的基本释义是子类覆盖父类中的一个实现。

例如，假设你有 Circle、Square 和 Star 三个形状组成的数组。尽管你把它们都当作 Shape 对象，并为每个 Shape 对象都发送一个 Draw 消息，但最终结果是每个图形都是不一样的，因为 Circle、Square 和 Star 提供了各自的具体实现。总之，每个类能够对同一个 Draw 方法返回不同的响应来绘制自己。这正是多态的意义。

Shape 类的代码如下所示：

```
public abstract class Shape{

    private double area;

    public abstract double getArea();

}
```

Shape 类有一个名为 area 的属性，它保存了形状的面积。getArea() 方法具有一个 abstract 修饰符。如果方法被定义为 abstract，子类必须提供该方法的实现。在本例中，Shape 要求子类提供 getArea() 的实现。现在创建一个名为 Circle 的类并继承自 Shape（extends 关键字指定了 Circle 继承自 Shape）：

```
public class Circle extends Shape{

    double radius;

    public Circle(double r) {

        radius = r;

    }

    public double getArea() {

        area = 3.14*(radius*radius);
        return (area);

    }
}
```

我们引入了一个名为构造函数（constructor）的新概念。Circle 类有一个同名的方法 Circle。如果方法名与类名相同，并且没有返回值，这个方法就是一个特殊的方法，称为构造函数。可以认为构造函数是类的入口，对象在这里被构造。构造函数里可以进行初始化操作和执行一些启动任务。

Circle 构造函数接收单个参数。这个参数代表半径，并且赋值给 Circle 类的 radius 属性。

Circle 类也提供了 getArea 方法的实现，而在 Shape 类中 getArea 是个抽象方法。

我们创建了一个名为 Rectangle 的相似的类：

```
public class Rectangle extends Shape{

    double length;
    double width;

    public Rectangle(double l, double w){
        length = l;
        width = w;
    }

    public double getArea() {
        area = length*width;
        return (area);
    }

}
```

现在我们可以创建任意数量的矩形和圆，只需调用它们的 getArea() 方法即可。这是因为我们知道所有的矩形和圆都继承自 Shape，所有的 Shape 类都有 getArea() 方法。

如果子类继承了父类的一个抽象方法，它必须提供该方法的具体实现，否则它自身也必须是个抽象类（请看图 1-18 的 UML 图）。这种方式提供了快速创建其他新类的机制。

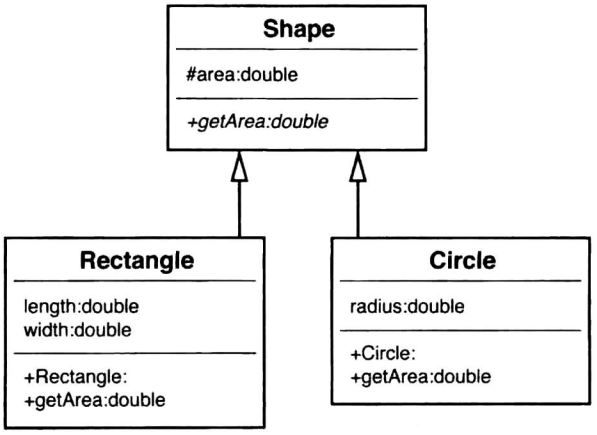


图 1-18　Shape UML 图

我们可以使用以下方式来实例化 Shape 类：

```
Circle circle = new Circle(5);
Rectangle rectangle = new Rectangle(4,5);
```

然后使用结构体栈（stack），把这些 Shape 类放到栈中：

```
stack.push(circle);
stack.push(rectangle);
```

什么是栈

栈是一种数据结构，表示一个后进先出的系统。它像一个自动换币器，你把硬币插入圆柱的最上头，当你需要硬币时，则从最上头拿，即拿的是最后一次放入的硬币。将一个东西推入栈意味着你把该东西加入了最顶端（比如将一个硬币插入自动换币器中）。从栈中取一个东西意味着你拿走了最近一次放入栈的东西（比如从最顶处取硬币）。

接下来是最好玩的部分。我们可以清空栈，而不用担心栈中存入了哪种 Shape 类（我只需知道它们都是形状）：

```
while (  !stack.empty()) {
    Shape shape = (Shape) stack.pop();
    System.out.println ("Area = " + shape.getArea());
}
```

实际上，我们对所有形状发送了相同的信息：

```
shape.getArea()
```

然而产生的实际行为取决于形状类型。例如，Circle 计算的是圆的面积，而 Rectangle

计算的是矩形的面积。实际上，我们给 Shape 类发送了一个消息，而期望根据使用的 Shape 子类类型来返回不同的行为。这体现了多态的关键概念。

这种方式就是为类之间（以及应用程序之间）提供标准化的接口。比如办公套件应用程序包含一个文字处理应用和一个电子表格应用。假设这两个都有一个名为 Office 的类，包含一个名为 print() 的接口。办公套件中的所有类都要实现 print() 接口。有意思的是尽管文字处理和电子表格都调用 print() 接口，但它们会做不同的事情，一个打印文字处理文档，另一个打印电子表格文档。

1.11　组合

对象中包含其他对象非常自然。比如电视机包含开关和显示屏。计算机包含显卡、键盘和光驱。可以将计算机自身看成一个对象，而光驱也是一个有效的对象。

事实上，你可以打开计算机把光驱取下来放到你手上。计算机和光驱都可以看作对象。只是计算机包含其他的对象（譬如光驱）。

使用其他对象来构建或结合成新的对象，这种方式就是组合。

1.11.1　抽象

和继承一样，组合也是一种构建对象的机制。事实上，我想说只有两种方式来使用其他类构建新类，这两种方式就是继承和组合。之前已经看到了，继承允许类继承另一个类。我们可以把属性和行为抽象到通用类中。例如，狗和猫都是哺乳动物，因为狗是（is-a）哺乳动物，猫也是（is-a）哺乳动物。而使用组合，我们也可以把类嵌入其他类中来构造新类。

我们可以拿车和引擎之间的关系来举例。把引擎从车中分离出来是明显可行的。我们可以单独构建引擎，把它用到各种车上，更别提还有其他优势。但我们不能说引擎是（is-a）一辆车。顺口说出来我们就觉得不对（因为我们在为真实世界的系统建模，这正是我们想要的效果）。我们最好用 has-a 术语来描述组合关系。车有（has-a）引擎。

1.11.2　has-a 关系

我们之前已经讨论过了继承关系是 is-a 关系的原因，而组合关系可以称为 has-a 关系。我们可以使用上一小节中的例子，电视有（has-a）开关和显示屏。电视显而易见不是一个开关，所以两者没有继承关系。同样，计算机有（has-a）显卡，有（has-a）键盘，有（has-a）光驱。第 7 章深入讲解继承和组合这两个主题以及相互之间的关系。

1.12　结语

关于面向对象技术实在有太多的内容可供讨论。不过学完本章后你应该对以下主题具有

深入的理解：

- **封装**。把数据和行为封装到单个对象中是面向对象开发中的重中之重。单个对象既包含自身的数据，也包含自身的行为，并且可以向其他对象隐藏自身的某些东西。
- **继承**。类可以继承自另一个类，并且可以使用父类中定义的属性和方法。
- **多态**。多态意味着相似的对象对相同的消息有着不同的响应。例如，你可能拥有一个有很多形状的系统。然而圆、正方形和星形的绘制方式不同。使用多态你可以给这些形状发送相同的消息（例如 Draw 方法），每个形状可以响应自身的绘制。
- **组合**。组合意味着可以使用其他对象来构建新对象。

本章涵盖了面向对象的基本概念，希望你尽可能地领悟这些知识。

1.13 本章中使用的示例代码

以下是 C#.NET 版本的代码。其他语言（比如 VB.NET 和 Objective-C）的代码在出版社网站上有电子版。本章已经展示了这些例子对应的 Java 代码。

1.13.1 C#.NET 版本的 TestPerson 类

```
using System;

namespace ConsoleApplication1
{
    class TestPerson
    {
        static void Main(string[] args)
        {

            Person joe = new Person();

            joe.Name = "joe";

            Console.WriteLine(joe.Name);

            Console.ReadLine();
        }
    }

    public class Person
    {

        //属性
        private String strName;
        private String strAddress;
```

```
        //方法
        public String Name

        {
            get { return strName; }
            set { strName = value; }
        }

        public String Address
        {
            get { return strAddress; }
            set { strAddress = value; }
        }

    }
}
```

1.13.2　C#.NET 版本的 TestShape 类

```
using System;

namespace TestShape
{
    class TestShape
    {
        public static void Main()
        {

            Circle circle = new Circle(5);
            Console.WriteLine(circle.calcArea());

            Rectangle rectangle = new Rectangle(4, 5);
            Console.WriteLine(rectangle.calcArea());

            Console.ReadLine();

        }
    }
    public abstract class Shape
    {

        protected double area;

        public abstract double calcArea();

    }
    public class Circle : Shape
    {
```

```
        private double radius;

        public Circle(double r)
        {

            radius = r;

        }

        public override double calcArea()
        {

            area = 3.14 * (radius * radius);
            return (area);

        }
    }
    public class Rectangle : Shape
    {

        private double length;
        private double width;

        public Rectangle(double l, double w)
        {
            length = l;
            width = w;
        }

        public override double calcArea()
        {
            area = length * width;
            return (area);
        }
    }
}
```

第 2 章 Chapter 2

如何以面向对象的方式进行思考

在第 1 章中，我们学习了面向对象（OO）的基本概念。本书的其他部分会深入讲解这些概念，也会介绍其他的一些知识。不管是否是面向对象的设计，出色的设计都需要考量很多因素。面向对象设计的基本单位是类。面向对象设计渴望的结果是一个健壮实用的对象模型，换句话说，就是一个完整的系统。

与生活中很多事情一样，在解决问题时，不管对错，并不只有一种方式。同一件事通常有多种处理方式。所以试图用面向对象的方案来进行设计时，不要执迷于一开始就尝试做出一个完美的设计（始终存在提升的空间）。我们真正需要做的是进行头脑风暴，让我们的思想延伸到不同的方向。当解决一个问题时不要尝试遵从任何标准或约定，因为整个想法需要有创造性。

事实上，刚开始时甚至不要先考虑一种具体的编程语言。业务上的首要事情是定位和解决业务问题。要先进行概念分析和设计。只有业务问题需要技术来解决时才需要考虑具体的技术。例如，你不能脱离无线技术来设计一个无线网络。而且通常的现实情况是你有多种软件方案可供选择。

因此，在设计一个系统甚至一个类之前，先思考问题本身。本章探索了面向对象思考这一艺术和科学。

关于思考的任何微小改变都是不平凡的。比如我们反复提及的从结构化开发方式变为面向对象开发的思维。之前已经讲过，该思维的一个误解是结构化开发过程和面向对象的开发过程是互斥的。其实并不是这样。众所周知，之前讨论过，通过包装对象，结构化开发和面向对象开发是可以共存的。事实上，当编写面向对象的应用程序时，到处在用结构化的结构体。我还从没看到过一个程序（无论是否采用了面向对象）没有使用循环、条件判断等。转

换到面向对象设计需要不同类型的投资。

从 FORTRAN 语言（甚至 C 语言）切换到 COBOL 语言相当于学习新的语言。然而，从 COBOL 语言切换到 C++、C# .NET、Visual Basic .NET、Objective-C 或者 Java，则要求学习一种新的思考过程。这正是滥用术语面向对象范式造成的不良后果。当学习面向对象语言时，你必须首先花精力来学习面向对象的概念以及相应的思考过程。如果没有进行这种范式的转变，以下两件事情必发生其一。首先是项目本质上不是真正的面向对象（例如，使用了 C++ 语言但没有使用面向对象特性），其次是项目完全由混乱的对象组成。

培养面向对象思考过程的良好习惯需要注意三个方面，本章会进行详细讲解：

- 清楚接口和实现之间的区别。
- 深入理解抽象。
- 给用户提供尽可能少的接口。

第 1 章中我们已经接触到了部分概念，接下来我们会深入细节。

2.1 清楚接口和实现之间的区别

正如第 1 章所示，构建健壮的面向对象设计的关键之一是理解接口和实现之间的不同。因此，当设计类时，应该向用户暴露什么、隐藏什么是非常重要的。而封装与生俱来的数据隐藏机制可以对用户隐藏不必要的数据。

> **小心**
>
> 不要混淆接口与图形化用户接口（graphical user interface，GUI）这两个概念。虽然 GUI 名称本身包含了接口这个单词，但是我们所说的接口是一种更通用的术语，它并不局限于图形化接口。

还记得第 1 章中的烤面包机例子吗？烤面包机（或相似功能的设备）需要插入一个接口，即电源插座，如图 2-1 所示。所有需要电的设备都需要符合正确的接口，即电源插座。烤面包机不需要知道插座的任何实现，或者电是如何产生的。对所有烤面包机而言，它不关心电是燃煤电厂还是核工厂生产的，只关心具体接口可以正确、安全的工作就行。

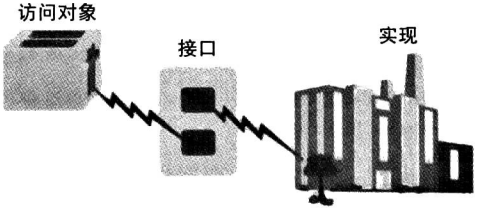


图 2-1 再看发电厂

还有一个汽车的例子。司机和汽车之间具有很多接口，比如方向盘、油门踏板、刹车和点火开关。先抛开美观问题，大多数人开车时的主要关注点是启动、加速、停止、转向等。大部分司机中极少关心那些眼睛看不到的组件（实现）。事实上，大多数人根本就无法识别

出某些组件，比如催化器和垫圈。然而，任何司机必须清楚如何使用油门踏板，因为这是一个通用接口。制造厂商为车安装一个标准的油门踏板，确保市场上的目标客户能够使用这个系统。

然而，如果制造厂商决定安装一个操纵杆来代替油门踏板，大多数司机会不习惯这点，那么这个车型销量不会很广（只能博得一些喜欢打破常规的人的喜爱）。而如果制造厂商替换了汽车的引擎（改变了部分实现），只要没有改变性能和外观，大多数司机都不会注意到这点。

只要接口不变，可替换的引擎必须严格遵守接口。把四缸发动机替换为八缸发动机可能会改变接口规则，导致需要使用该引擎接口的其他组件不能正常工作。而在发电厂例子中从交流电（AC）改成直流电（DC）也会影响接口规则。

引擎属于实现，方向盘属于接口。改变实现不会对司机造成影响，而改变接口则会。司机会注意到方向盘的外观变化，即使改变可能很微小。必须强调的是，对引擎的改变不应让司机注意到，否则就会破坏接口。例如，改变引擎可能会丧失动力，这会引起驾驶者的注意，事实上是改变了接口。

> **用户能看到什么**
>
> 接口与类直接相关。终端用户通常看不到任何类，只会看到 GUI 或者命令行。然而，程序员会接触类接口。重用类的前提是有人已经写了一个类。因此，程序员必须知道如何正确使用这个类。程序员需要将很多类组合成一个系统，所以需要理解类的接口。因此，本章中讨论用户时，我们主要指设计人员和开发人员，没必要引入终端用户。因此，当我们在此上下文中讨论接口时，我们在讨论类接口，而不是 GUI。

正确地设计类时要注意两部分，即接口和实现。

2.1.1　接口

呈现给终端用户的服务暴露了接口。最佳实践中，只呈现给用户需要的服务。当然，不同的人对用户需要什么服务可能持有不同看法。如果你把 10 个人放到一个屋子让他们每个人进行独立设计，你可能会得到 10 份完全不同的设计，而且这些设计都没什么错。然而，作为一个通用的规则，一个类的接口应该只包含需要用户知道的东西。在烤面包机例子中，用户只需要知道烤面包机必须插到接口上（这个例子中接口就是电源插座）以及如何操作烤面包机本身。

> **识别用户**
>
> 当设计类时最重要的考虑就是识别类的读者（或用户）。

2.1.2　实现

实现细节对于用户是隐藏的。我们必须时刻牢记关于实现的一个目标，那就是修改实现

不需要变动用户代码。看起来可能有些困惑，但该目标是设计问题的核心所在。如果对接口的设计是恰当的，那么即使调整了实现，用户调用代码也无需任何改变。请记住，接口包含了调用方法及返回值的语法。如果没有改变接口，用户无需关心是否修改了实现。程序员只关心使用相同的语法能够获得相同的值即可。

我们可以拿手机来举例。打电话的接口很简单，我们只需拨一个号码或者从地址簿中选取一个联系人。如果供应商更新了软件，它不会改变你打电话的方式。无论如何修改实现，打电话的接口始终保持不变。然而，如果我的电话区号变了，供应商也有可能会修改接口。基础接口变了（比如电话区号变了），需要用户改变行为。商家希望保持这样的修改最小化，因为有些用户不喜欢这种改变，或者不想这样的麻烦。

再说烤面包机的例子。只要接口一直是电源插座，具体实现就可能会从一个燃煤电厂切换为核电站，但这不会影响烤面包机。这里有一个非常重要的规则，即煤电厂和核电厂都必须要遵循接口规格。如果煤电厂提供交流电（AC），而核电厂提供直流电（DC），就会出问题。用户和实现都必须要遵循接口规格。

2.1.3 一个接口 / 实现示例

我们来创建一个简单读取数据库的类。我们将编写一些 Java 代码，这些代码会从数据库中获取记录。正如之前讨论的一样，进行任何设计时，识别终端用户一直是最重要的问题。你可能需要做一些场景分析，一起对终端用户做一些引导性访谈，然后会列出这个项目的需求。接下来是我们对这个数据库阅读器的需求：

- 必须能打开数据库的连接。
- 必须能关闭数据库的连接。
- 必须能将游标指向数据库中的第一条记录。
- 必须能把游标指向数据库中的最后一条记录。
- 必须能得到数据库中的记录条数。
- 必须能知道当前数据库是否仍有记录（即我们当前是否指向的是最后一条记录）。
- 必须能够根据键值把游标指向特定的记录。
- 必须能够获取指定键值的记录。
- 必须能基于当前游标的位置获取下一条记录。

根据以上需求，可以开始尝试设计一个读取数据库的类，为终端用户设计可能的接口。

在本例中，读取数据库的类仅提供给想使用数据库的程序员。因此接口本质上是程序员想要使用的应用程序编程接口（application-programming interface，API）。这些方法其实包装了数据库系统暴露的功能。为什么要这么做？本章后面会详细讨论该问题。简短的回答是我们需要定制数据库功能。例如，我们必须处理对象从而可以将它们写入关系型数据库中。编写这样的中间件对于设计和编码而言可能过于简单，但这是封装特性的真实示例。最重要的是，如果我们想替换数据库引擎，则无需修改大量代码。

图 2-2 展示了一个类图，表示了 DataBaseReader 类的潜在接口。

请注意，该类中的方法都是公共方法（请记住，靠近方法名的加号表示该方法是一个公共接口）。而且这里只展示了接口，没有展示任何实现。请花一分钟来确定这个类图是否能大体满足上面列出的项目需求。如果你之后发现该类图没有满足所有需求也没关系。因为面向对象设计是一个迭代的过程，所以你无须一开始就保证它绝对正确。

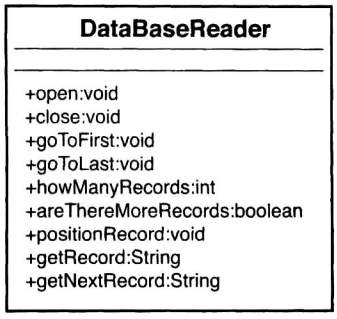


图 2-2　DataBaseReader 类的 UML 类图

公共接口

请记住，如果一个方法是公共方法，那么程序员就可以访问它，因此可以认为它是类的接口。请不要混淆术语“接口”与 Java 和 .NET 中的关键字 interface。稍后的章节会讨论关键字 interface。

对于我们列出的每个需求，需要有对应的方法来提供对应的功能。现在需要考虑一些问题：

- 实际使用此类时，作为程序员的你需要了解与其有关的其他事情吗？
- 需要知道内部数据库代码是如何打开数据库的吗？
- 需要知道内部数据库代码如何对应一条具体记录的物理位置吗？
- 需要知道内部数据库代码如何确定是否还有剩余记录吗？

回答是都不需要！你不需要知道任何信息。只需要关心能获取到正确的值并且操作没有出错。事实上，程序员更喜欢对具体实现再做一层抽象。应用程序将使用你自定义的类来操作数据库，而这些自定义的类则负责调用相应的数据库 API。

最小接口

在极限情况下，保证最小接口是刚开始不给用户提供任何公共接口。当然，这样的类是无用的。然而，这强制用户主动找你说：“我需要这个功能。”然后你们可以协商。这样保证你只在需要的情况下增加接口，绝不要假设用户需要什么东西。

创建包装对象看起来有些小题大做，但编写这样的类有很多好处。比如，当今市场上有很多中间件产品使用了包装对象的技术。考虑把对象映射到关系型数据库的问题。一些面向对象的数据库可能非常适合面向对象的应用程序。然而，一个现存的问题是大多数公司有数年的遗留数据存放在关系型数据库中。如果公司既需要保留关系型数据库中的数据，又要拥抱面向对象技术，那么如何处理这个断层？

首先，可以把所有遗留的关系型数据转换到一个全新的面向对象的数据库中。然而，任何遭受过严重的（也是长期的）数据转换之痛的人都知道不能这样做。这种转换往往会耗费大量的时间和精力，到头来系统还是不能正常工作。

其次，可以使用中间件产品把应用程序代码中的对象平滑地映射到关系型模型中。只要关系型数据库依旧盛行，这种方案相比之前就要更好些。有些人可能会认为面向对象的数据库比关系型数据库更方便持久化对象。事实上，很多开发系统都能提供这样平滑转换的服务。

对象持久化

对象持久化，意思即保存对象的状态以便稍后可以恢复和使用，因为没有持久化的对象在其生命周期之外就会被销毁掉。例如，对象的状态可以保存在数据库中。

在当前的业务环境下，关系型数据库和对象建立映射关系是一个非常好的方案。很多公司集成了这样的中间件技术。比如一个公司拥有一个网站作为前端接口，而数据存放在大型机中，这很常见。

如果创建一个完全面向对象的系统，使用面向对象的数据库是个可行的选项（也拥有更好的性能）。不过面向对象的数据库的发展历程与面向对象语言的发展历程比起来差远了。

独立应用程序

甚至从头创建一个全新的面向对象的应用程序，也很难完全避免遗留数据。新创建的面向对象的应用程序也不会是独立的应用程序，因为它很可能需要获取存储在关系型数据库（或者其他的数据存储设备）中的数据。

让我们回到数据库例子中。图 2-2 只展示了该类的公共接口，除此之外别无其他。当完成该类后，它可能会包含更多的方法，当然也会包含一些属性。不过作为使用该类的程序员无须知道任何私有方法和属性的相关信息。你肯定无需了解公共方法中的具体代码，只需简单知道如何与这些接口交互即可。

公共接口的实现代码会是什么样子呢（假设我们使用的是 Oracle 数据库）？我们来看看 open() 方法：

```
public void open(String Name){

        /* 一些具体的应用程序处理代码 */

        /* 调用 Oracle API 来打开数据库 */

        /* 一些具体的应用程序处理代码 */

};
```

在这个例子中，如果你是程序员，会发现 open 方法需要 String 类型作为参数。Name 代表了需要传入的数据库文件，但我们并不关心它如何映射到一个具体的数据库。使用接口只需知道如何使用这个方法，不用关心方法细节，这正是接口的美好之处！

为了迷惑用户，我们修改数据库的实现。昨晚我们把 Oracle 数据库中的全部数据迁移到了一个 SQLAnywhere 数据库中（我们忍耐了这巨大而漫长的痛苦）。虽然总共花费了好几个

小时，但最终我们做到了。

现在代码如下所示：

```
public void open(String Name){

        /* 一些具体的应用程序处理代码 */

        /* 调用 SQLAnywhere API 来打开数据库 */

        /* 一些具体的应用程序处理代码 */

};
```

今天早上竟然没有听到任何用户的抱怨。这是因为虽然改变了实现，但并未改变接口！用户关心的调用接口仍然是相同的。修改实现代码可能需要大量的工作（即使只改了一行代码，整个模块都需要重新编译），但使用了 DataBaseReader 类的应用程序代码则无需任何修改。

> **代码重新编译**
>
> 动态加载的类是在运行时加载的，并不是静态链接到一个可执行文件。当使用动态加载的类（比如 Java 和 .NET）时，无需重新编译使用者的类。在静态链接语言（比如 C++）中，引入新类需要一个链接。

分离用户接口与实现，能省却大量头痛的事。在图 2-3 中，数据库的具体实现对终端用户来说是透明的，终端用户只能看到接口。

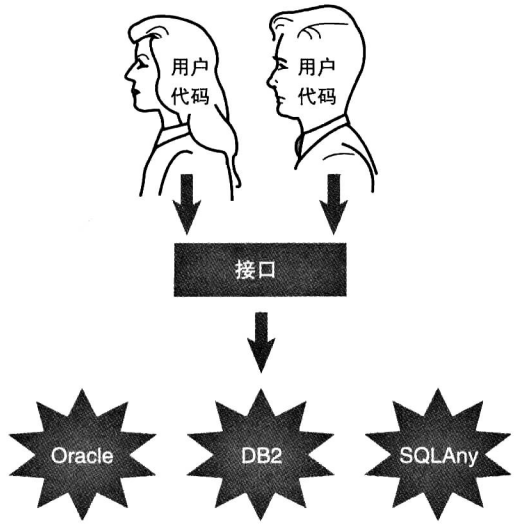


图 2-3　接口

2.2　使用抽象思维设计接口

面向对象编程的主要优势之一是可以重用类。通常可以重用的类比具体的类的接口更加抽象。具体的接口可以是非常明确的，而抽象接口则更通用。简单来说，高层次的抽象接口比高度具体的接口更有用，大部分情况下如此，当然并非适用于所有情况。

完全可以编写一个非常有用的具体的类，而且这个类无法重用。这种情况经常出现，在某些情况下它也没什么错。然而，我们正在进行业务设计并且希望能得到面向对象提供的优势。所以我们的目标是设计抽象的可高度重用的类。为了做到这点，需要设计高度抽象的用户接口。我们可以创建一个出租车对象来演示抽象接口与具体接口的区别。该对象拥有一个诸如“送我去机场”接口可能比拥有一些独立的诸如“左转”、“右转”、“启动”、“停止”之

类的接口更有用。因为正如图 2-4 所示，用户只想去机场。

当你从酒店出来时，把行李放到出租车后备厢，并坐上出租车，出租车司机会问你："你想去哪里？"你回答："请送我到机场。"（当然，这里假设这个城市只有一个机场。在芝加哥，你可能会说："请送我到米德韦机场。"或者会说："请送我到奥黑尔机场。"）。你可能自己都不知道如何到机场，甚至即使你知道，你也无须告诉司机什么时候转弯，转到哪个方向，如图 2-5 所示。作为乘客的你无需关心出租车司机的实际驾驶路径。（然而，如果司机故意绕路去机场，费用可能会有点问题。）

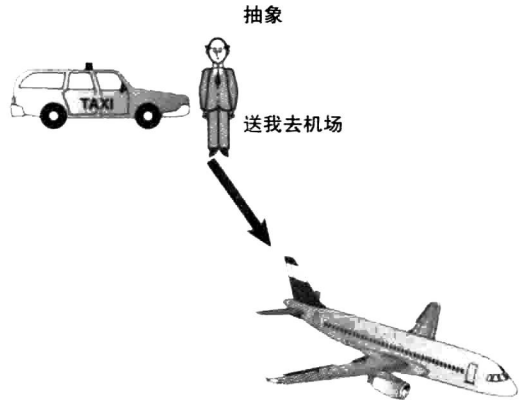


图 2-4　抽象接口

现在，抽象和重用的联系在哪里？请你自问这两个场景哪个更具重用性，是抽象的还是非抽象的？我们来更进一步简化问题，以下哪个短语更具重用性？是"送我去机场"还是"右转，右转，然后再左转，左转，左转"。显而易见，第一个短语更具重用性。你可以在任何城市中使用它，只要你钻进一个出租车并且想去机场。第二个短语只在特定的情况下可行。因此，抽象接口"送我去机场"更具通用性。可重用的面向对象设计针对芝加哥、纽约或克利夫兰市的实现是不同的。

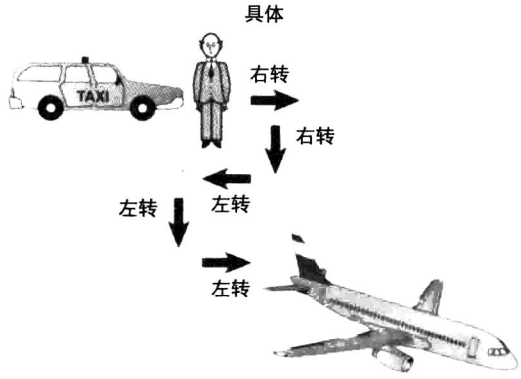


图 2-5　一个具体的接口

2.3　尽可能提供最小化的用户接口

当设计类时，通用规则是尽量不要让用户知道类内部的工作原理。为了达到这点，请遵守以下简单的规则：

- 只提供给用户绝对需要的东西。实际上，这意味着类的接口要尽可能少。当你开始设计一个类时，先从最小化的接口开始。类的设计是迭代式的，所以随后即使你发现最小化的接口可能不合适，也没关系。

最好只有用户真正需要时才添加接口，不要提供超出用户需求的接口。用户拥有某些接口可能会出问题。比如，你不该提供给所有用户关于薪水信息的接口，而应该只提供给需要的用户。

这次我们用硬件示例来阐述软件示例。想象一个用户有一台个人计算机主机，但没有显

示器和键盘。显然，这个个人计算机主机用处不大。你只提供给了该用户关于该个人计算机最小化的接口。然而，该最小化的接口是不合适的，有必要立即添加接口。

- 公共接口定义了用户可以访问什么。你可以刚开始设置所有接口为私有的，这样向用户隐藏了整个类，当程序员开始使用这个类时，再把某些方法公开出来，这些方法变成了公共接口。
- 从用户角度定义类至关重要，而不是从信息系统的角度定义类。通常类（而不是软件）的设计者设计类时会使其适用一个具体的技术模型。即使该设计者站在用户的视角，它仍然可能是一个技术用户的视角。而这个类的设计往往只考虑了技术因素，而没有基于用户的视角。
- 确保设计类时你向真正的用户了解了需求和设计。这些人并不局限于开发人员。当构建系统原型时，类通常都会需要更新和演化。

2.3.1　确定用户

我们再看出租车示例。我们已经确定用户是实际使用这个系统的人。那么，谁是用户？

第一反应是客户。答案只对了一半。客户确实是用户，出租车司机必须为客户提供成功的服务。换句话说，提供一个诸如“免费送我到机场”的接口毋容置疑会取悦客户，但对出租车司机则不是这样。因此，实际上，为了构建一个真实可用的接口，必须考虑客户和出租车司机都是用户。

同样在软件领域，用户可能想要程序员提供某个功能。然而，如果程序员发现该需求在技术上是不可行的，不管程序员多想提供帮助，这样的需求是无法满足的。

总之，任何给出租车对象发送消息的对象都可以认为是用户（是的，用户也是对象）。图 2-6 展示了出租车司机如何提供服务。

展望

出租车司机也是一个对象。

图 2-6　提供服务

2.3.2　对象行为

识别用户只是实践的一部分。识别用户之后，你必须确定对象行为。需要从每个用户的视角来开始识别每个对象的目的以及需要做的事情。注意，很多初始的决策无法保留到最终

的公共接口中。可以通过使用各种方法（比如 UML 用例）聚合需求来进行决策。

2.3.3 环境约束

在《Java 面向对象设计》一书中，Gilbert 和 McCarty 指出环境通常会限制对象。事实上，环境限制往往都是影响因子。计算机硬件可能会限制软件功能。比如，一个系统可能没有连接到网络，或者一个公司可能会使用特定类型的打印机。在出租车例子中，司机不能通过一座坏了的桥，尽管这条路是去机场的捷径。

2.3.4 识别公共接口

我们已经收集到了用户信息、对象行为和环境信息，接下来需要为每个用户对象指定公共接口。所以，想象一下会如何使用出租车：

- 上车。
- 告诉司机你想去哪里。
- 付车费。
- 享受旅途。
- 下车。

你需要如何使用出租车对象？

- 知道去哪里。
- 租用出租车。
- 付给司机钱。

刚开始，你只需考虑如何使用这个对象，不用思考如何构建这个对象。稍后你可能会发现对象需要更多的接口，比如“把行李放到车厢中”或者“和司机进行随意交谈”。图 2-7 提供了一个类图来列出了 Cabbie 类的可能的方法。

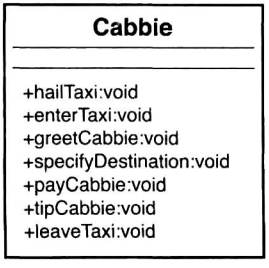


图 2-7 Cabbie 类中的方法

得到最终的接口始终是一个迭代的过程。你必须确定每个接口是否有利于对象的操作。如果不利于，那么它可能没必要存在。很多面向对象的文章推荐每个接口模型只包含一个行为。这带给我们的问题是我们的设计究竟要抽象到哪种层次。如果有一个叫作 enterTaxi() 的接口，我们肯定不想 enterTaxi() 包含付费的逻辑。这样做不但设计上毫无道理，而且事实上该类的使用者会不知道到底如何付费。

2.3.5 识别实现

明确了公共接口后，接下来需要识别实现。设计完类，该类已经包括所有需要操作的方法，接下来需要具体考虑如何保证该类正常工作。

从技术角度来讲，任何非公共接口可以视为实现。这意味着用户不会看到具体的实现方

法，包括方法签名（方法签名包括方法名和参数列表）和方法中的实际代码。

类可能有一些私有方法仅供内部使用。任何私有方法都可以视为实现的一部分，用户绝不会看到它，从而也不能访问它。例如，一个类有一个 `changePassword()` 方法，然而，这个类可以有一个私有方法来对密码做加密处理。这个方法应该对用户是隐藏的，并且只能通过 `changePassword()` 方法调用它。

应该对用户完全隐藏实现细节。公共方法中的代码是实现的一部分，因为用户不能看到它。（用户只能看到接口的调用结构，而看不到里面的代码。）

理论上来说，任何对实现的修改都不应该影响用户通过接口与类的交互方式。当然，前提假设是实现提供的结果是用户期望的结果。

接口代表了用户如何看待对象，实现则是对象的具体细节。实现包含了代表对象状态的代码。

2.4 结语

在本章中，我们探索了可以开始以面向对象的方式进行思考的三个方面。请记住，面向对象思考过程中遇到的问题并不是固定的。以面向对象的方式做事情更像是一门艺术，而不是科学。尝试使用自己的方式来描述面向对象的思考过程。

第 3 章会讨论对象生命周期，包括对象的出生、存活和死亡。在它存活期间，它很可能变换很多状态。例如，当数据库打开时，`DataBaseReader` 对象是一种状态，而当数据库关闭时则会进入另一种状态。这种表现取决于类的设计。

2.5 引用

- Meyers, Scott. 2005. *Effective C++*, 3rd edition. Boston, MA: Addison-Wesley Professional.
- Fowler, Martin. 2003. *UML Distilled*, 3rd edition. Boston, MA: Addison-Wesley Professional.
- Gilbert, Stephen, and Bill McCarty. 1998. *Object-Oriented Design in Java*. Berkeley, CA: The Waite Group Press (Pearson Education).

Chapter 3 第 3 章

高级的面向对象概念

第 1 章和第 2 章讲述了面向对象的基本概念。在开始学习关于构建面向对象系统的一些具体设计问题之前，我们需要更进一步了解面向对象的一些概念，比如构造函数、操作符重载以及多重继承。我们也会讲述错误处理技术以及面向对象的设计中作用域的重要性。

其中一些概念可能对深入理解面向对象设计并不是必需的，但设计和实现整个面向对象系统的人有必要了解。

3.1 构造函数

*构造函数*对于结构化编程的程序员来说是个新概念。非面向对象的语言（比如 COBOL、C 和 Basic）通常不会用到构造函数。C/C++ 中的结构体（struct）具有构造函数。前两章提及过这个用于构造对象的特殊方法。在诸如 Java 和 C# 之类的面向对象的语言中，构造函数名称与类名相同。而 Visual Basic .NET 使用关键字 New，Objective-C 使用 init 关键字。这里我们只关注于构造函数的概念，而不会介绍所有语言的特殊语法。接下来用 Java 代码来实现一个构造函数。

例如，第 2 章中 `Cabbie` 类的构造函数如下所示：

```
public Cabbie(){
    /* 用于构造对象的代码 */
}
```

编译器会意识到这个方法名与类名完全相同，所以认为该方法是个构造函数。

小心

注意，Java 代码（以及 C# 和 C++）中，构造函数没有返回值。如果有返回值，编译器就不认为该方法是构造函数。

例如，如果类中有以下代码，那么编译器不会认为该方法是构造函数，因为它有返回值，这个返回值是一个整数：

```
public int Cabbie(){
    /* 用于构造对象的代码 */
}
```

该语法会导致问题，因为虽然这份代码可以通过编译但得不到期望的行为。

3.1.1　什么是构造函数调用

当创建新对象时，首要事情之一是调用构造函数。请看以下代码：

```
Cabbie myCabbie = new Cabbie();
```

new 关键字创建了 Cabbie 类的一个新实例，这会按需分配内存。然后会调用构造函数自身，并且可以通过参数列表传递参数。开发人员可以在构造函数内进行相应的初始化工作。

因此，new　Cabbie() 代码将实例化一个 Cabbie 对象，并调用 Cabbie 方法，即该类的构造函数。

3.1.2　构造函数中包含什么

构造函数最重要的功能大概是当遇到 new 关键字时初始化内存分配。总之，构造函数中的代码会把新创建的对象初始化到稳定、安全的状态。

例如，如果有一个计数器（counter）对象，里面有个属性叫 count，你需要在构造函数中将 count 设置为 0：

```
count = 0;
```

初始化属性

在结构化编程中，名为 housekeeping（管家）或 initialization（初始化）的例程往往用于初始化目的。初始化属性是构造函数经常执行的功能。

3.1.3　默认构造函数

如果编写了一个不包含构造函数的类，这个类仍然可以通过编译，你也可以使用它。如果没有为类提供一个显式的构造函数，那么类会有一个默认构造函数。请记住，无论你是否自定义了构造函数，类始终至少有一个构造函数。如果你没有提供构造函数，系统会为你提

供一个默认的构造函数。

除了创建对象本身之外，默认构造函数的另一个行为是调用父类的构造函数。大多数情况下，父类是语言框架的一部分，比如 Java 中的 Object 类。例如，如果没有为 Cabbie 类提供构造函数，系统会提供下面默认的构造函数：

```
public Cabbie(){
    super();
}
```

如果反编译编译器生成的字节码，你会看到这段代码。这段代码实际上是由编译器插入的。

在本例中，如果 Cabbie 没有显式继承自其他类，Object 类将会是它的父类。默认构造函数在有些场景下是适用的。然而，大多数场景下，需要自定义初始化一系列内存。不管在什么情况下，在类中始终包含至少一个构造函数是一个优秀的实践。如果类有属性，最好始终在构造函数中初始化这些属性。延伸开来，无论是否在编写面向对象的代码，初始化变量总是一个优秀的实践。

提供构造函数

通用规则是即使并不需要在构造函数中做任何事情，也应当始终提供一个构造函数。你可以提供一个不包含任何代码的构造函数，稍后再按需添加代码。尽管使用编译器默认提供的构造函数在技术上没有任何问题，但基于文档化和维护目的，这样更容易看懂你的代码。

这里考虑维护问题并不奇怪。如果你使用的是默认的构造函数，后续操作添加了另一个构造函数，那么系统不会再创建默认的构造函数。总之，只有类中没有包含任何构造函数时，系统才会添加默认的构造函数。一旦你提供了一个构造函数，系统就不再提供默认的构造函数。

3.1.4 使用多个构造函数

大多数情况下，可以用多种方式创建对象。这需要提供多个构造函数。例如，请看 Count 类：

```
public class Count {

    int count;

    public Count(){
        count = 0;
    }
}
```

一方面，可以初始化属性 count 为 0，实现这一点很简单，可以在一个构造函数中初始化 count 为 0，如下所示：

```
public Count(){
    count = 0;
}
```

另一方面，可以传递一个初始化参数，从而可以设置 count 为其他数字：

```
public Count (int number){
    count = number;
}
```

这叫作重载方法（重载适用于所有方法，不止是构造函数）。大部分的面向对象语言都提供了重载方法的功能。

1. 重载方法

重载可以让程序员重复使用相同的方法名，只要每次方法签名不同即可。方法签名包含了方法名以及参数列表（如图 3-1 所示）。

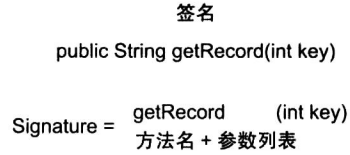


图 3-1　签名的组成

所以，以下所有方法拥有不同的签名：

```
public void getCab();

// 不同的参数列表
public void getCab (String cabbieName);

// 不同的参数列表
public void getCab (int numberOfPassengers);
```

签名

方法签名可能包含返回值类型，也可能不包含返回值类型，这取决于不同的语言。在 Java 和 C# 中，返回值类型并不属于签名的一部分。例如，以下代码即使返回值类型不同，也不能通过编译：

```
public void getCab (String cabbieName);
public int getCab (String cabbieName);
```

了解签名最好的方式是编写一些代码然后进行编译。

通过使用不同的签名，你可以根据不同的构造函数来构造对象。如果你不能保证每次都能掌握足够的信息，那么很适合这种方式。例如，当创建一个购物车时，顾客可能已经登录了自己的账号（你会得到顾客所有的信息）。而一个全新的顾客可能会向购物车中放入产品，但没有任何账号信息，这样的情况下构造函数初始化方式是不同的。

2. 使用 UML 对类建模

我们再回头看第 2 章中用到的数据库阅读器例子。构造数据库阅读器有两种方式：

- 传入数据库名称以及设置游标在数据库中的起始位置。

- 传入数据库名称以及设置游标在数据库中的期望位置。

图 3-2 展示了 DataBaseReader 类的类图。注意，该图列出了此类的两个构造函数。尽管该图显示了两个构造函数，但并未包含参数列表，所以无法区分出这两个构造函数。为了区分这两个构造函数，可以查看下面列出的 DataBaseReader 类的对应代码。

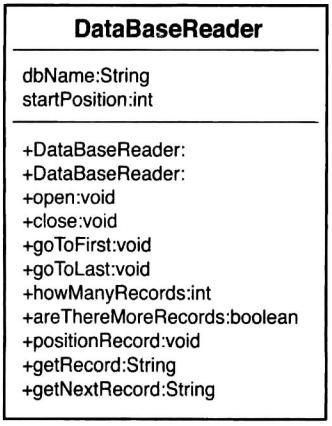


图 3-2 DataBaseReader 类图

无返回值类型

注意，在该类图中，构造函数没有返回值类型。除了构造函数之外，其他所有方法必须要有返回值类型。

以下代码片段展示了该类的构造函数，以及构造函数如何初始化属性（见图 3-3）：

```
public class DataBaseReader {

    String dbName;
    int startPosition;

    // 仅初始化名称
    public DataBaseReader (String name){
        dbName = name;
      startPosition = 0;
    };

    // 初始化名称和位置
    public DataBaseReader (String name, int pos){

        dbName = name;
        startPosition = pos;
    };

    .. // 类中其他部分
}
```

请注意在两个场景中如何初始化 startPosition。如果没有通过参数列表为构造函数提供位置信息，那么 startPostion 会被初始化为默认值，即 0。

3. 如何构造父类

当使用继承时，你必须知道如何构造父类。请记住，当使用继承时，也继承了父类的所有东西。因此必须熟悉父类的数据和行为。任何继承的属性都是完全可见的。然而，对构造函数的继承则是不可见的。如果遇到了 new 关键字，那么会分配对象，并发生以下步骤（见图 3-4）：

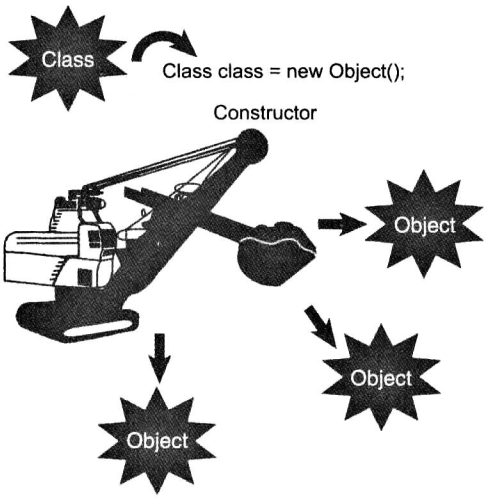


图 3-3　创建新对象

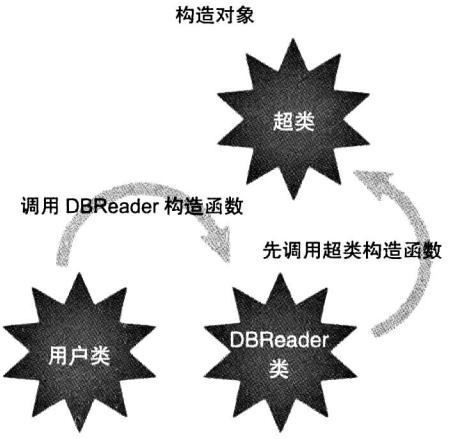


图 3-4　构造对象

1）在构造函数中会调用父类的构造函数。如果没有显式调用父类的构造函数，那么系统会默认自动调用；不过可以在字节码中看到这段代码。

2）对象中的所有属性会被初始化。这些属性是类中定义中的属性（实例变量），不是构造函数或其他方法中的属性（局部变量）。在 `DataBaseReader` 代码中，整数 `startPosition` 是类的实例变量。

3）执行构造函数中的其余代码。

3.1.5　设计构造函数

我们已经看到了，设计类的一个最佳实践是初始化所有属性。有些语言中，编译器会提供一部分初始化工作。与往常一样，不要依赖编译器来初始化属性！在 Java 中，只有属性被初始化后你才能使用它。如果属性在代码中很靠前，请确保你初始化属性为一些有效值，比如设置整数为 0。

构造函数用来确保应用程序处于稳定的状态（我喜欢称之为“安全”的状态）。例如，如果把属性作为除法运算中的分母，那么初始化该属性为 0 会导致应用程序崩溃。你必须考虑除法中用 0 作为除数是非法操作。始终初始化属性为 0 并不总是最好的方式。

在设计时，优秀的实践应该是为所有属性识别一个稳定的状态，然后在构造函数中初始化这些属性为稳定的状态。

3.2　错误处理

首次编写一个类时总想让它完美无缺。但在大多数情况下，事情却往往不是我们想象的

那样。开发人员不预先处理错误，会将程序置于危险之中。

假如想在代码中加入识别错误和处理错误的能力，有好几种方式。在《Java Primer Plus》（9781571690623）一书的第 223 页，Tyma、Torok 和 Downing 指出在程序中有三种基本的解决方案来处理发现的问题：修复问题、通过压制来忽略问题，或以合适的方式退出运行时。《Java 面向对象设计》（9781571691347）一书中的第 139 页，Gilbert 和 McCarty 解释了异常处理的几个选择：

- 忽略该问题。这不是好主意！
- 检查潜在的问题，当发现问题时中止程序。
- 检查潜在的问题，捕获错误并试图修复该问题。
- 抛出异常（通常这是处理异常的最佳方式）。

下面详细讨论了这些策略。

3.2.1 忽略问题

简单地忽略潜在的问题往往会引发灾难。如果你想忽略该问题，那么为什么不在发现时就处理它？显然，不应该忽略任何已知的问题。所有应用程序的基本职责是保证程序绝不崩溃。如果不处理错误，应用程序最终会以丑陋的方式崩溃掉，或者进入一种不稳定的状态。之后继续使用的话你甚至不知道得到结果是否是正确的，这比程序崩溃更糟糕。

3.2.2 检查问题并中止应用程序

如果检查潜在的问题，并且当检测到问题时中止应用程序，应用程序可以显示一条关于该问题的信息。这种情况下应用程序优雅地退出，但留下用户盯着计算机屏幕，想搞清楚刚才发生了什么。显然，这是一个远胜于忽略问题的选项，但它不是最佳选项。不过这可以让系统收拾残局并进入稍稳定的状态，比如关闭文件和强制系统重启。

3.2.3 检查问题并试图恢复

检查潜在问题，捕获错误，并试图恢复的方案远胜于简单地检查问题并中止程序的方案。在这种情况下，通过代码检测发现问题，然后应用程序自身会尝试修复它。这在某些场景下是适用的。例如，请看以下代码：

```
if (a == 0)
    a=1;

c = b/a;
```

显然，如果代码中没有该条件语句，0 会进入除法语句，因为不能除以 0，从而导致系统异常。捕获该异常并设置变量 a 为 1，至少能保证系统不会崩溃。不过设置 a 为 1 可能不是一个正确的方案，因为结果是不正确的。更好的方案是提醒用户重新输入恰当的值。

混合错误处理技术

尽管事实上错误处理类型并不属于面向对象的核心范畴，但我认为在面向对象设计中可以考虑使用。抛出异常（接下来的小节会涉及）会带来昂贵的开销。因此，尽管异常是有效的设计决策，你也应该根据设计及性能要求，适当考虑其他错误处理技术（甚至使用 tried-and-true 结构化技术）。

之前提到的错误检查技术强于什么都不做，但这种技术也有一些问题。比如并不总能在错误第一次发生的地方就能探测到该错误。很可能问题发生一段时间后才会被探测到。不过详细解释异常处理已经超出了本书的范围。然而，一开始设计类时就设计异常处理是非常重要的，而且通常操作系统自身也会对它发现的问题发出警告。

3.2.4　抛出异常

大多数面向对象的语言提供了名为异常的功能。通常来讲，异常是系统内发生的突发事件。异常提供了一种方式来检测问题然后进行处理。在 Java、C#、C++、Objective-C 和 Visual Basic 中，异常是通过关键字 `catch` 和 `throw` 来处理的。这有点像棒球游戏，但这里的关键概念是用特定的代码块用来处理特定的异常。这既解决了尝试找出错误发生的地方的问题，也解决了在正确的地方处理该错误的问题。

以下是 Java 中 `try/catch` 代码块的结构：

```
try {

    // 可能有问题的代码

} catch(Exception e) {

    // 处理异常的代码
}
```

如果在 `try` 代码块中抛出了异常，`catch` 代码块会处理该异常。如果代码块执行过程中抛出了异常，会发生以下事情：

1）`try` 代码块会结束执行。

2）`catch` 从句会检查对应的 `catch` 代码块能否处理这种异常。（一个 `try` 代码块可能会对应多个 `catch` 从句。）

3）如果所有 `catch` 代码块都不能处理抛出的异常，那么该异常会传递给最近的更高一层的 `try` 代码块中（如果代码中没有捕获该异常，系统最终会捕获它，结果是无法预料的，可能导致应用程序崩溃）。

4）如果有一个 `catch` 从句匹配上了（遇到了第一个匹配的从句），会执行 `catch` 从句中的代码。

5）程序会从紧挨着 try 代码块的下面的代码处恢复执行。

以上几点足够说明异常处理是面向对象程序语言中的一个重要优势。以下是 Java 中处理异常的一个示例：

```
try {

    // 可能有问题的代码
    count = 0;
    count = 5/count;

} catch(ArithmeticException e) {

    // 处理异常的代码
    System.out.println(e.getMessage());
    count = 1;


System.out.println("The exception is handled.");
```

异常粒度

你可以在不同层级的粒度上捕获异常。你可以捕获所有的异常或者检查特定的异常（比如算术异常）。如果代码没有捕获异常，Java 运行时会捕获它，那么 Java 运行时可能会崩溃！

在以上代码中，try 代码块中除以 0（因为 count 等于 0）会导致算术异常。如果异常发生（抛出）在 try 代码块之外，程序很可能会中止（崩溃）。然而，因为异常是在 try 代码块中抛出的，catch 代码块会检查是否能处理特定的异常（本例中即算术异常）。因为 catch 代码块包含了对算术异常的检查，所以 catch 代码块中的代码会执行，设置 count 为 1。在 catch 代码块执行之后，try/catch 代码块会退出，并把异常被处理的信息输出到 Java 终端。图 3-5 展示了整个处理过程。

图 3-5 捕获异常

如果没有把 ArithmeticException 放入 catch 代码块，程序很可能会崩溃。可以使用以下代码来捕获所有异常：

```
try {

    // 可能有问题的代码

} catch(Exception e) {

    // 处理异常的代码
}
```

catch 块中的 Exception 参数能够匹配在 try 代码块中产生的任何异常。

稳定的代码

综合使用以上描述的方式可以保证应用程序尽可能保持稳定的状态。

3.3 作用域的重要性

单个类可以实例化出多个对象。每个对象有唯一的标识和状态。这点很关键。会给每个单独构造的对象会分配独立的内存。然而，一个类实例化的多个对象可以共享类中一些属性和方法，从而共享为这些属性和方法分配的内存。

共享方法

构造函数是一个被类的所有实例共享的方法，这是共享方法的一个好例子。

方法代表了对象的行为，属性代表了对象的状态。对象有三种属性：

- 局部属性。
- 对象属性。
- 类属性。

3.3.1 局部属性

局部属性由特定的方法拥有。请看以下代码：

```
public class Number {

    public method1() {
        int count;

    }

    public method2() {

    }

}
```

方法 method1 包含了一个名为 count 的局部变量。只能在 method1 中访问该整数。method2 甚至不知道整数 count 的存在。

基于该点，我们引入了一个非常重要的概念：作用域。属性（和方法）存在于特定的作用域中。在本例中，整数 count 存在于 method1 的作用域中。在 Java、C#、C++ 和 Objective-C 中，使用花括号（{}）来定义作用域。在 Number 类中，通过匹配花括号可以得到几个作用域。

类自身拥有作用域。类的每个实例（即每个对象）也有自己的作用域。method1 和 method2 也拥有各自的作用域。因为 count 存活于 method1 的花括号中，所以当调用 method1 时，会创建 count 的一个副本。当 method1 终结时，会删除 count 的这个副本。

为了让事情更有趣，请看以下代码：

```
public class Number {

    public method1() {
        int count;
    }

    public method2() {
        int count;
    }

}
```

在本例中有两个 count 的副本存在于该类中。请记住，method1 和 method2 拥有各自的作用域。因此编译器可以简单通过分辨 count 所在的方法来确定对 count 的可访问性。你可以这样认为：

```
method1.count;

method2.count;
```

就编译器来说，即使这两个属性的名字是一样的，也很容易区分彼此。这就好比两个人拥有相同的名，但是他们的姓不一样，所以他们是独立的个体。

3.3.2 对象属性

在很多设计场景中，可以在同一个对象中的多个方法中共享属性。例如图 3-6，从单个类中构造出了三个对象。请看以下代码：

```
public class Number {

    int count;     // 对 method1 和 method2 都是可见的

    public method1() {
```

```
        count = 1;
    }

    public method2() {
        count = 2;
    }

}
```

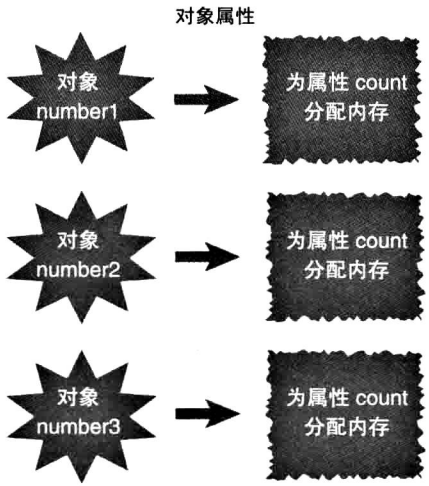


图 3-6　对象属性

注意，该类中对属性 count 的声明放在方法 method1 和 method2 的作用域之外。然而，它在该类的作用域之内。因此，count 对于 method1 和 method2 都是可见的。(基本上，该类中的所有方法都可以访问这个属性)。请注意，两个方法中的代码都指定了 count 的值。整个对象只有一份 count 副本，所以两个赋值行为操作的是内存中的同一份副本。然而，不同的对象并不共享 count 的副本。

我们创建了 Number 类的三个实例副本来进行演示：

```
Number number1 = new Number();
Number number2 = new Number();
Number number3 = new Number();
```

number1、number2、number3 这三个对象都是单独构造的，而且都单独分配了自己的资源。整数 count 有三个独立的实例。当 number1 修改了自身的属性 count 时，不会影响对象 number2 和 number3 中的 count 副本。因此整数 count 是对象属性。

可以通过以下代码来理解作用域：

```
public class Number {

    int count;

    public method1() {
        int count;
    }

    public method2() {
        int count;
    }

}
```

每个对象对于 count 有三个完全独立的内存位置。对象 Number 拥有一份副本，method1() 和 method2() 也各自拥有一份副本。

为了从其中一个方法（比如 method1()）访问对象属性，可以使用一个基于 C 语言的名为 this 的指针：

```
public method1() {
        int count;

        this.count = 1;
}
```

请注意，一些代码有些奇怪：

```
this.count = 1;
```

以上代码需要使用 this 关键字。我们必须使用它。这里使用 this 关键字指导编译器访问对象属性 count，而不是方法体中的局部变量 count。

注意

关键字 this 是对当前对象的一个引用。

3.3.3 类属性

正如之前提到的一样，两个或多个对象之间可以共享属性。在 Java、C#、C++ 和 Objective-C 中，可以把属性设置为 static：

```
public class Number {

    static int count;

    public method1() {
    }

}
```

声明 count 为 static 类型，那么从该类中实例化的所有对象只会为该属性分配一块单独的内存。即该类的所有对象对 count 使用同一块内存位置。事实上，每个类只有一个副本，该类的所有对象共享该副本（如图 3-7 所示）。这与我们在面向对象设计中所说的全局数据很相似。

类属性具有很多实用的用途；然而，你必须要了解潜在的同步问题。我们来初始化两个 Count 对象：

```
Count Count1 = new Count();
Count Count2 = new Count();
```

为便于讨论，Count1 对象使用 count 来记录计算机屏幕上的像素。如果 Count2 对象决定使用属性 count 来记录绵羊个数就会出问题。Count2 记下了第一只羊的瞬间，Count1 保存的数据会丢失掉。

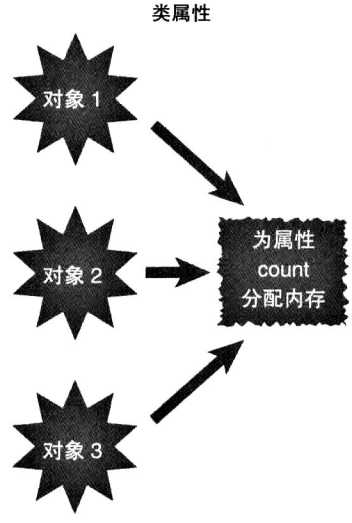


图 3-7 类属性

3.4　操作符重载

有些面向对象的语言允许你重载操作符。C++ 是其中的一个例子。操作符重载允许你修改一个操作符的含义。例如，当大多数人看到一个加号时，会假设它代表加法运算。如果你看到以下恒等式

```
x = 5 + 6;
```

你会认为 x 的值是 11。在这种情况下你是对的。

然而，有时加号也会代表其他含义。例如下面的代码：

```
String firstName = "Joe", lastName = "Smith";

String Name = firstName + " " + lastName;
```

你可能期望 Name 等于 Joe Smith。这里重载的加号用来进行字符串连接操作。

字符串连接

将两个独立的字符串合并为一个新的单个字符串，称为字符串连接。

在字符串上下文中，加号并不表示整数或浮点数的加法操作，而是指字符串的连接。

那么矩阵加法呢？比如有以下代码：

```
Matrix a, b, c;

c = a + b;
```

这里的加号会进行矩阵加法，而不是整数或浮点数的加法。

重载是一种非常强大的机制。然而，它可以完全迷惑阅读和维护代码的人。事实上，开发人员也可能会迷惑自己。如果极端点，可以把加法操作修改为执行减法。操作符重载给予了你修改操作符含义的能力。因此，如果加号被修改为执行减法，以下的代码会导致 x 的值为 -1：

```
x = 5 + 6;
```

近代面向对象的语言（比如 Java、.NET 和 Objective-C）不允许重载操作符。

尽管这些语言没有提供重载操作符的选项，但这些语言自身重载了加号用于字符串连接。Java 的设计者很可能认为操作符重载弊大于利。如果你在 C++ 中必须使用操作符重载，请认真地加入正确的文档和注释，以免迷惑使用该类的人。

3.5　多重继承

第 7 章会讲述有关继承的更多细节。然而，这里很适合讨论多重继承。多重继承是类设计中最强大也最具挑战的一个方面。

顾名思义，多重继承允许一个类继承自多个类。实际看起来这是个好主意。对象都应该模拟现实世界，不是吗？现实世界中存在很多多重继承的例子。父母是多重继承的一个好例子。每个孩子有一对父母，这是非常自然的事。所以设计类时使用多种继承是有道理的。在一些面向对象的语言中（比如 C++ 中）你可以这样做。

然而，这种情况会陷入与操作符重载相似的情况。多重继承是非常强大的技术，事实上没有它一些问题会变得难以解决。多重继承甚至可以完美解决一些问题。然而，多重继承无论对程序员还是编译器作者而言，都会显著增加系统的复杂度。

与操作符重载一样，Java、.NET 和 Objective-C 的设计者认为多重继承带来的系统的复杂度超过了带来的好处，因此从语言层面消除了多重继承。Java、.NET 和 Objective-C 语言提供的接口构造能在某些方面能弥补这一点。但 Java、.NET 和 Objective-C 不允许传统的多重继承。

行为继承和实现继承

接口是行为继承的一种机制，抽象类则用于实现继承。编程语言中的接口类型提供不同行为的接口，但不提供实现，而抽象类既提供接口，也能提供实现。第 8 章详细阐述了该主题。

3.6 对象操作

当处理复杂的数据结构和对象时，编程中的很多最基本的操作会变得越来越复杂。例如，当你想复制或比较原始数据类型时，过程非常简单。然而，复制和比较对象则并不简单。《Effective C++》第 34 页中，Scott Meyers 专门花了整个小节来讲述复制和分配对象。

类和引用

复杂的数据结构和对象的问题在于它们可能会包含引用。简单对引用的复制不能复制它引用的数据结构或对象。同样，当比较对象时，简单地比较两个指针只是比较了引用，而并未比较指针所指的对象。

当对对象进行比较和复制时你就会意识到该问题。具体来说，问题归结于你是否使用了指针。不管怎样，总有一种方式能复制对象。不过这种方式看上去简单，但实际并不简单。因为对象可以包含引用，而必须对整个引用树做有效的复制（如果你要创建一份深拷贝）。

深拷贝与浅拷贝

追踪所有的引用，并对所有引用对象都创建拷贝，这种方式称为深拷贝。深拷贝会拥有很多层级。一个对象引用了很多对象，而这些被引用的对象也可能引用其他对象。拷贝本身开销巨大。浅拷贝只会简单地拷贝引用，而不会深入层级。在《Java 面向对象设计》一书第 265 页的“只见树木不见森林”一节中，Gilbert 和 McCarty 对深拷贝和浅拷贝进行了精彩的讨论。

如图 3-8 所示，如果只是简单地拷贝对象（称为逐位复制），只会拷贝引用，而不会拷贝实际的对象。因此，两个对象（原始对象和副本）会引用（指向）同一个对象。为了完全复制（即需要复制每个引用对象），你必须编写代码来创建所有的子对象。

当比较对象时也有类似的问题。正如拷贝功能一样，比较功能并不像看起来那么简单。因为对象包含引用，必须根据引用树进行有效的对象比较。大多数情况下，语言提供了一种默认的机制来比较对象。当设计类时，你应当在类中提供一个比较功能，从而保证类的行为是预期的。

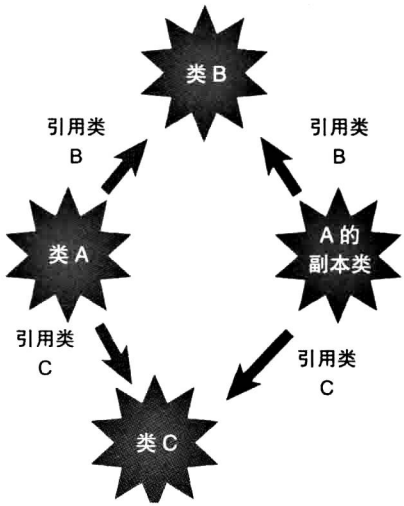


图 3-8　深入对象引用

3.7　结语

本章涵盖了一些面向对象的高级概念。虽然对于大致理解面向对象，可能没必要学习这些概念，但对于面向对象的更高一级任务（比如设计类）则是非常必要的。在第 4 章中，我们开始具体来看如何设计和构造类。

3.8　引用

- Meyers, Scott. 2005. *Effective C++*, 3rd edition. Boston, MA: Addison-Wesley Professional.
- Gilbert, Stephen, and Bill McCarty. 1998. *Object-Oriented Design in Java*. Berkeley, CA: The Waite Group Press.
- Tyma, Paul, Gabriel Torok, and Troy Downing. 1996. *Java Primer Plus*. Berkeley, CA: The Waite Group.

3.9　本章中使用的示例代码

以下是 C# .NET 版本的代码。其他语言（比如 VB .NET 和 Objective-C）的代码在出版社网站上有电子版。本章已经展示了这些例子对应的 Java 代码。

C# .NET 版本的 `TestNumber` 类

```
using System;

namespace TestNumber
{
```

```
    class Program
    {
        public static void Main()
        {

            Number number1 = new Number();
            Number number2 = new Number();
            Number number3 = new Number();
        }
    }

    public class Number
    {

        int count = 0;      // 对 method1 和 method2 是可见的

        public void method1()
        {
            count = 1;
        }

        public void method2()
        {
            count = 2;
        }

    }
}
```

第 4 章 Chapter 4

类的剖析

之前的章节涵盖了面向对象的基本概念，并且讲述了接口和实现的区别。不管你对接口与实现的区别理解有多深入，至少你在设计一个类时要思考该类是否好用，以及它如何与其他类交互。绝不能凭空设计一个类，我曾经说过任何一个类都不可能是个孤岛。因为当对象被实例化后，它始终要与其他对象交互。而且对象也可以作为其他对象的一部分，或者作为继承体系中的一部分。

本章实现了一个简单的类，并逐段查看代码来指导设计类时应该考虑的准则。我们将继续使用第 2 章中出租车司机的例子。

以下每个小节都包含了类的某个特定方面。虽然有些部分不会出现在每个类中，但有助于理解类的设计和构造原理。

> **注意**
>
> 该类只用于演示目的。其中一些方法并不充实（即没有任何实现），只简单地呈现了接口，这样做主要为了强调接口是刚开始进行设计时的关注点。

4.1 类名

类名之所以重要有好几个原因。最明显的原因是类名用来识别类本身。除了简单的识别作用之外，类名必须是描述性的。选择一个合适的名称相当重要，因为类名提供了这个类的用途以及在大系统中的交互方式等信息。

语言自身的一些限制也造就了类名的重要性。例如，Java 中，公共类的类名必须与文件

名一致。如果两者不匹配，应用程序就无法工作。

图 4-1 展示了我们需要详细解释的类。在这个例子中紧跟着关键字 class 之后的 Cabbie 就是类名，它清晰又简单。

```
public class Cabbie {

}
```

注释

```
/*
 该类定义了一个出租车司机并分配了一辆出租车
*/
public class Cabbie{
```

类名

属性

```
  // 公司名称
  private static String companyName = "Blue Cab Company";
  // 出租车司机姓名
  private String Name;
  // 分配给出租车司机的车
  private Cab myCab;
```

构造函数

```
  //Cabbie 类的默认构造函数
  public Cabbie() {

    name = null;
    myCab = null;

  }
  //Cabbie 类的可初始化姓名的构造函数
  public Cabbie(String iName, String serialNumber){

    Name = iName;
    myCab = new Cab(serialNumber);
  }
```

访问方法（公共接口）

```
  // 设置出租车司机姓名
  public void setName(String iName) {
    Name = iName;
  }
  // 获取公司名称
  public static string getName(){
    return Name;
  }
  // 获取出租车司机姓名
  public static String getCompanyName(){
    return companyName;
  }
```

公共接口

```
  public void giveDestination(){
  }
```

私有实现

```
  private void turnRight(){
  }
  private void turnLeft(){
  }
}
```

图 4-1 示例类

使用 Java 语法

请记住，本书约定使用 Java 语法。Java 语法比较简单，但某些语法与 C# .NET、VB .NET、Objective-C 或 C++ 相比可能有些不同，也可能与其他面向对象的语言（比如 Smalltalk）完全不同。

当实例化 Cabbie 类时会使用到类名。

4.2 注释

不管注释采用什么语法，它都能帮助我们理解类的功能。在 Java、C# .NET、Objective-C 和 C++ 中，有两种注释。

Java 和 C# 中特别的注释风格

Java 和 C# 中的注释有三种类型。在 Java 中，第三种注释类型（/** */）表示 Java 提供的格式文档。本书不会涉及这种注释。C# 也提供了类似的语法来创建 XML 文档。

第一种注释是传统 C 语言风格的注释，它使用 /*（斜杠和星号）来开始注释，并使用 */（星号和斜杠）来结束注释。这种注释可以从一行扩展到多行，请注意，不要忘记为每个注释使用配对的开始注释和结束注释符号。如果忘记了结束注释符号（*/），一部分代码则会被标记为注释从而被编译器忽略。以下是一段在 Cabbie 类中使用的这种风格的注释：

```
/*

  本类定义了一个出租车司机并分配了一辆出租车

*/
```

第二种注释是 //（两个斜杠），它会把紧随其后直到行尾的所有代码都作为注释。这种类型的注释能应用到一行代码，所以你无须使用结束注释符号，但请注意每次只能注释一行，并且在注释后面不能跟随任何需要运行的代码。以下是 Cabbie 类中使用的这种类型的注释：

```
// 司机姓名
```

4.3 属性

属性代表了对象的状态，因为它们存储了该对象的相关信息。在本例中，Cabbie 类用属性存储了公司名称、司机姓名以及分配给该司机的出租车。例如，第一个属性存储了公司名称：

```
private static String companyName = "Blue Cab Company";
```

请注意，这里有两个关键字 private 和 static。关键字 private 表示方法或变量只

能被声明的对象访问。

尽可能隐藏数据

本例中的所有属性都是私有的。这是保持接口设计尽量最小化的设计准则。访问属性的唯一方式是使用类提供的方法接口（本章稍后会讲述）。

static 关键字表示从该类实例化的所有对象只存在该属性的一份副本。可以说这是个类属性（请看第 3 章关于类属性的更多讨论）。所以即使从 Cabbie 类实例化 500 个对象，而属性 companyName 在内存中只存在一个副本（见图 4-2）。

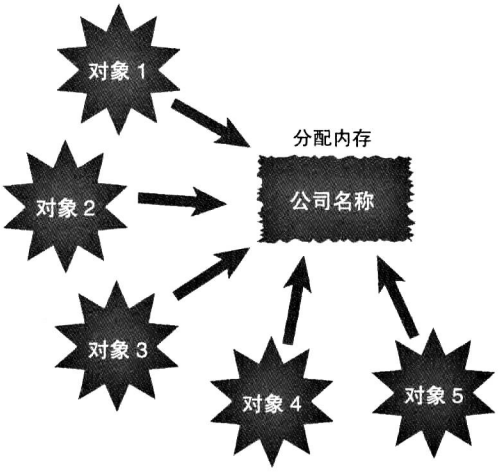


图 4-2 对象内存分配

第二个属性 name 是一个存储了司机姓名的字符串：

```
private String name;
```

这个属性也是私有的，所以其他对象无法直接访问它。其他对象必须使用接口方法来访问该属性。

myCab 属性是对另一个对象的引用。名为 Cab 的类存储了出租车的相关信息，比如它的车牌号以及维修记录：

```
private Cab myCab;
```

传递引用

Cab 对象可能由其他对象创建。所以该对象引用会传递给 Cabbie 对象。不过在这个例子中，Cab 对象是在 Cabbie 对象中创建出来的。因此我们实际并不关心 Cab 对象的内部结构。

请注意，此时只是创建了对 Cab 对象的引用；该定义并不会导致内存分配。

4.4 构造函数

Cabbie 类包含两个构造函数。之所以能识别某些方法是构造函数是因为方法名与类名是相同的，都是 Cabbie。第一个构造函数是默认构造函数：

```
public Cabbie() {

    name = null;
    myCab = null;

}
```

技术上来说，这并不是系统提供的默认构造函数。如果你没有为类指定任何构造函数，调用编译器时，编译器会提供一个默认构造函数。我之所以称这个方法为默认构造函数是因为它是不带参数的构造函数，其实际上重载了编译器提供的默认构造函数。

如果提供了带参数的构造函数，系统就不会提供默认构造函数。这个规则是只有在代码中没有为类提供构造函数时，编译器才会在编译时提供默认构造函数。

在该构造函数中，属性 Name 和 myCab 被设置为 null：

```
name = null;
myCab = null;
```

null 的不存在性

很多编程语言中，值 null 表示一个不存在的值。这似乎是一个深奥的概念，但把属性设置为不存在的值是非常有用的编程技术。检查变量是否为 null 可以识别该值是否正确初始化。例如，你可能想声明一个属性，随后要求用户输入该值。因此，在用户有机会输入数据之前你可以初始化该属性为 null。先初始化一个属性为 null（这是一个有效的条件），稍后可以检查该属性是否正确设置。注意，一些语言中，字符串类型不允许为 null。例如在 .NET 中要求使用 name=string.empty;。

我们知道始终在构造函数中初始化属性是个优秀的实践。同样，检查属性值是否为 null 也是一个优秀的编程实践。如果事先能发现没有正确设置属性或者对象，可以减少很多不必要的麻烦。例如，如果将一个真实对象赋予 myCab 引用以前你就使用了该引用，那么很可能会遇到问题。如果在构造函数中设置 myCab 引用为 null，稍后当你想使用它时可以检查 myCab 是否仍然为 null。如果使用了一个未初始化的引用会产生一个异常。

还有另一个例子：如果一个 Employee（雇员）类包含了一个 spouse（配偶）属性（只是为了举例），你最好考虑到雇员还没结婚这种情况。你可以初始化该属性为 null，然后稍后检查它的状态。

第二个构造函数为类的用户提供了一种初始化 Name 和 myCab 属性的方式。

```
public Cabbie(String iName, String serialNumber) {

    name = iName;
    myCab = new Cab(serialNumber);

}
```

用户可以提供两个字符串给该构造函数的参数列表，用于正确地初始化属性。注意，myCab 对象是在该构造函数中被实例化的。

```
myCab = new Cab(serialNumber);
```

执行这行代码结果会分配 Cab 对象的内存

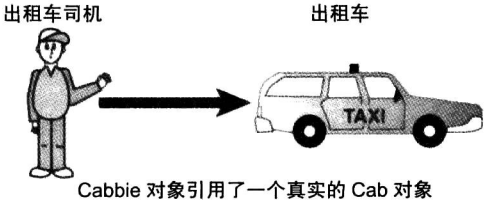


图 4-3 Cabbie 对象引用了一个 Cab 对象

存储。图 4-3 演示了 myCab 属性如何引用 Cab 对象的新实例。本例中的两个构造函数展示了方法重载的这一常用方式。注意，构造函数都定义为 public，这是因为构造函数很显然是类接口的成员。如果构造函数是私有的，其他对象就不能访问它们，从而无法实例化 Cab 对象。

4.5 访问器

本书中的绝大部分例子中，属性都被定义为私有的，所以其他对象无法直接访问属性。创建一个不会与其他对象交互的孤立的对象是荒谬的，因为我们需要在对象间分享恰当的信息。有时候是否有必要检查或修改另一个类中的属性？答案当然是需要。大多数时候一个对象需要访问其他对象的属性。不过它不需要直接访问。

类应该很注意保护自身属性。例如，我们不会让对象 A 在不受控的情况下检查或修改对象 B 中的属性。这样做有好几个理由。最重要的原因是保证数据完整性以及高效调试。

假设 Cab 类存在一个缺陷。你已经追踪到问题出在 Name 属性上。不知什么原因它被覆盖了，查询名称会返回垃圾信息。如果 Name 是 public 的，任何类可以修改它，你需要搜索所有可能的代码，尝试找到引用或修改了 Name 的地方。然而，如果你只允许 Cabbie 对象修改 Name，你只需要查看 Cabbie 类。一种名为*访问器*的类型方法可以提供访问。有时访问器又被称为取值方法（getter）和赋值方法（setter），有时又被简单称为 get() 和 set()。在本书中我们约定方法名加上 set 和 get 前缀，如下所示：

```
// 设置司机姓名
public void setName(String iName) {
    name = iName;
}

// 获取司机姓名

public String getName() {
    return name;
}
```

在该代码片段中，Supervisor 对象会要求 Cabbie 对象返回其名称（见图 4-4）。关键在于 Supervisor 对象自身无法直接获取该信息，它必须询问 Cabbie 对象来获得信息。这种理念可应用于很多地方。比如，你可能有一个 setAge() 方法检查输入的年龄是 0 还是小于 0。如果年龄小于 0，setAge() 方法可以拒绝设置这个不正确的值。通常，赋值方法可以在某种程度确保数据的完整性。

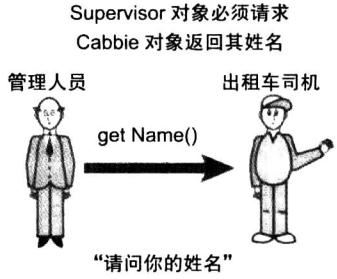


图 4-4 询问信息

这也可以解决安全问题。你可能有一些敏感数据，比如

你想控制对密码或者支付信息的访问。因此通过取值方法和赋值方法来访问数据可以提供一种机制用于密码检查或其他验证技术。这极大地增加了数据的完整性。

注意类方法 getCompanyName 被声明为 static。第 3 章详细描述了类方法。请注意属性 companyName 也被声明为 static。方法和属性一样可以被声明为 static，用于表示整个类只存在该方法的一个副本。

对象

实际上不会为每个对象的非静态方法都建立一个物理副本。每个对象会指向同一段物理代码。然而从概念层面来说，你可以认为对象是完全独立的，拥有各自的属性和方法。

下面的代码演示了如何定义一个静态方法，图 4-5 展示了多个对象如何指向相同的代码。

静态属性

如果属性是静态的，而且类为该属性提供了一个赋值方法，其他对象调用该赋值方法只会修改同一个副本。因此，对该属性值的修改会应用于所有对象。

```
// 获取出租车司机姓名
public static String getCompanyName() {
    return companyName;
}
```

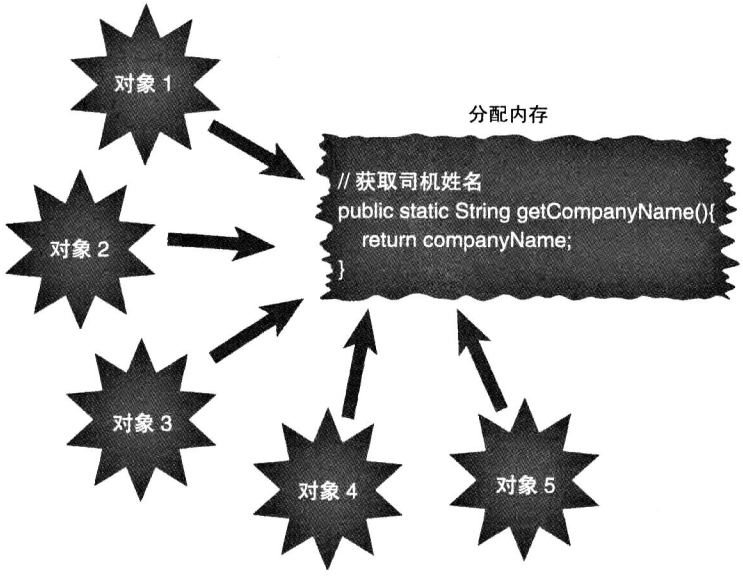


图 4-5 为方法分配内存

4.6 公共接口方法

构造函数和访问器都被定义为公共的，并且属于公共接口的一部分。对外暴露它们是因为它们是使用该类的重要方式。其实很多实际工作都在其他方法中。正如第 2 章中所提及的一样，公共接口方法倾向于非常抽象化，而实现倾向于更具体化。在该类中，我们提供了一个名为 giveDestination 的方法作为公共接口，让用户来描述目的地：

```
public void giveDestination (){

}
```

该方法中包含什么目前来说并不重要。关键在于它是个公共方法，而且是该类的公共接口的一部分。

4.7 私有实现方法

尽管本章中目前为止讨论的所有方法都是公共方法，但并不能说明类中的所有方法都属于公共接口。类中的某些方法对其他类是隐藏的，这种情况很常见。下面的方法被声明为 private：

```
private void turnRight(){
}

private void turnLeft() {
}
```

这些私有方法属于实现，而不属于公共接口。你可能会问如果其他类不能访问它们，那么谁会调用这些方法。答案很简单，你可能已经猜到了，类自身内部会调用这些方法。例如，可以在方法 giveDestination 中调用这些方法：

```
public void giveDestination (){

    .. 一些代码

    turnRight();
    turnLeft();

    .. 另一些代码 }

}
```

再举一个例子，假设有一个内部方法用于数据加密，那么只能在类自身中使用该方法。总之，除了实例化的对象自身能调用该方法，其他对象无法调用该方法。

类中的私有方法是实现的一部分，其他类不能访问它。

4.8 结语

在本章中，我们深入研究了一个类，描述了构建类时必要的基本概念。本章通过实践的角度讨论类，第5章会从设计角度来讲述类。

4.9 引用

- Fowler, Martin. 2003. *UML Distilled,* 3rd edition. Boston, MA: Addison-Wesley Professional.
- Gilbert, Stephen, and Bill McCarty. 1998. *Object-Oriented Design in Java*. Berkeley, CA: The Waite Group Press.
- Tyma, Paul, Gabriel Torok, and Troy Downing. 1996. *Java Primer Plus*. Berkeley, CA: The Waite Group.

4.10 本章中使用的示例代码

以下是C#.NET版本的代码。其他语言（比如VB.NET和Objective-C）的版本在出版社网站上有电子版。本章已经展示了这些例子对应的Java代码。

C# .NET版本的 `TestCab` 类

```
using System;

namespace ConsoleApplication1
{
    class TestPerson
    {
        public static void Main()
        {
            Cabbie joe = new Cabbie("Joe", "1234");

            Console.WriteLine(joe.Name);
            Console.ReadLine();
        }
        }

    public class Cabbie
    {

            private string _Name;
            private Cab _Cab;

            public Cabbie() {
```

```
            _Name = null;
            _Cab = null;

        }

        public Cabbie(string name, string serialNumber) {

        _Name = name;
           _Cab = new Cab(serialNumber);

            }

    //方法
    public String Name
    {
        get { return _Name; }
        set { _Name = value; }
    }

}

public class Cab
{

            public Cab (string serialNumber) {

            SerialNumber = serialNumber;

            }

    //可以从外部访问该属性，但不能修改该属性
    public string SerialNumber { get; private set; }

}
}
```

第 5 章 Chapter 5

类设计指导

之前我们已经讨论过了，面向对象编程可以在创建类时将数据和行为封装到单个实体中。所以，一个类可以代表一个逻辑组件，比如出租车。

本章介绍了设计优秀的类的几项建议。显然这样的建议列表很难称得上是完整的。你完全可以吸收并采纳其他程序员的好建议。

5.1 对现实世界系统建模

面向对象编程的主要目标是按照人们实际所想的相似方式对现实世界系统建模。以面向对象的方式设计类来创建这些模型。使用过程化或自上而下的方式时，数据和行为在逻辑上是分离的实体。而面向对象的方式把数据和行为封装到可以相互交互的对象里。我们处理问题时不会再以一系列事件或者程序来操作独立的数据文件。原因在于类真正代表了现实世界的对象的模型，而且展示了这些对象如何与现实世界中其他的对象交互。

类之间交互的方式与现实世界对象之间的交互方式相似，比如人与其他对象的交互。因此当创建类时，你应该设计这些类来代表对象的真实行为。我们继续使用前一章节中的出租车司机的例子。Cab 类和 Cabbie 类是现实世界中的实体模型。如图 5-1 所示，Cab 和 Cabbie 对象封装了各自的数据和行为，并且通过各自的公共接口进行交互。

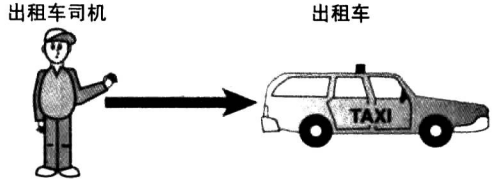


图 5-1 出租车和司机是现实世界中的对象

当第一次使用面向对象方式进行开发时，很多人仍然尝试以结构化的方式来思考问题。这

样导致的主要错误之一是创建只有行为没有数据的类。事实上，他们是在结构化的模型中创建一组函数或子程序。这并不是你想要的，因为这没有利用到封装的强大优势。

注意

我最喜爱的有关类设计指导及建议的其中一本书是《Effective C++：50 Specific Ways to Improve Your Programs and Designs》，由 Scott Meyers 编写。它以简洁明了的方式提供了关于程序设计的重要信息。

Effective C++ 吸引我的原因之一是 C++ 是向后兼容 C 的，编译器允许在 C++ 中不用遵守面向对象的设计准则，而编写结构化的代码。这正是遵循准则（比如存在于 Effective C++ 的准则）如此重要的原因。正如我之前提到的一样，在面试过程中，一些人认为他们是面向对象的程序员，只是因为他们使用 C++ 编程。这指出了对到底什么是面向对象设计的完全误解。因此比起 Java 或 .NET，在使用诸如 C++ 和 Objective-C 之类的语言时要更加注意语言中的面向对象设计问题。

5.2 识别公共接口

显而易见，在设计类时最重要的问题是保持公共接口最小化。构建类的整个目的是提供有用且简便的功能。在《Java 面向对象设计》一书中的第 109 页，Gilbert 和 McCarty 指出，“设计良好的对象接口描述了客户想要实现的服务”。如果类没有为用户提供有用的服务，那么就不应该构建该类。

5.2.1 最小化公共接口

提供最小化的公共接口可以保证类尽可能地简单。我们的目标是提供给用户具体接口来正确地完成工作。如果公共接口是不完整的（即有缺失的行为），那么用户不能够完成整个工作。如果没有正确限制公共接口（即用户可以执行不必要的甚至是危险的行为），那么会出问题，需要调试程序，甚至会陷入系统一致性问题或者遭受安全问题。

创建类是业务决策，保证用户能参与开始的设计阶段到测试阶段的整个设计步骤是非常重要的。这种方式可以保证类的实用性以及接口的正确性。

扩展接口

如果一个类的公共接口对某个应用程序是不够的，对象技术可以轻松扩展和适配该接口。总之，如果设计时使用继承和组合，那么一个新类可以利用现有的类，而且可以使用扩展接口创建新类。

我们继续使用出租车司机例子来做演示。如果系统中的其他对象需要获取出租车司机的

姓名，Cabbie 类必须提供一个公共接口返回它的姓名，即 getName() 方法。因此，如果 Supervisor 对象需要 Cabbie 对象的名字，它必须调用 Cabbie 对象的 getName() 方法。事实上，管理人员会询问出租车司机的姓名（见图 5-2）。

使用你的代码的用户无需知晓其内部工作原理。唯一需要知道的是如何实例化和使用对象。总之，提供给用户使用方式，但是隐藏细节。

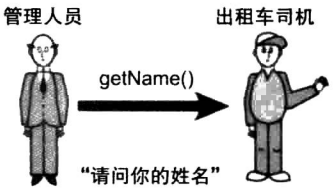


图 5-2　公共接口详细说明了对象如何交互

5.2.2　隐藏实现

隐藏实现的原因已经阐述得非常详细了。但是识别公共接口与类的用户相关，而具体实现则与用户无关。具体实现必须提供用户需要的服务，但这些服务是如何实现的对用户来说是不可见的。改变类的实现不应该影响到用户，这才是设计良好的类。例如，修改了具体实现，那么用户无需修改其应用程序代码。

> **客户与用户**
>
> 有时当描述实际使用软件的人时，我使用术语客户，而不是用户。因为我是一个顾问，系统的用户事实上是客户。同样，用户是你组织的一部分，可以称为内部客户。这似乎是个简单的问题，但我认为把所有终端用户当作实际客户是相当重要的，而且你必须满足他们的要求。

在出租车司机例子中，Cabbie 类可能包含吃早餐等行为。然而，出租车司机的管理人员无需知道出租车司机吃了什么早餐。因此，该行为是 Cabbie 对象实现的一部分，在本系统中对于其他对象应该是不可见的（见图 5-3）。Gilbert 和 McCarty 指出封装的最高指导原则是“所有字段都应该是私有的”。在这种方式下，其他对象无法直接访问类中的任何字段。

图 5-3　对象无需知晓具体实现细节

5.3　设计健壮的构造函数（以及析构函数）

当设计类时，最重要的设计问题之一是如何构造类。第 3 章中我们讨论了构造函数。如果需要温习设计构造函数的准则，可以回顾一下第 3 章。

首先并且最重要的一点是，构造函数应该把对象设置为安全的初始状态。这包括了属性初始化以及内存管理等问题。你也需要确保在默认的条件下能正确地构造对象。通常提供一个构造函数来处理默认情况是个好主意。

在包含析构函数的语言中，析构函数包括了正确的清除功能，这也很重要。在大多数情

况下，清除功能就是释放对象在某些时间点被分配的系统内存。Java 和 .NET 通过垃圾回收机制自动地回收内存。在诸如 C++ 的语言中，开发人员必须在析构函数使用代码来正确地释放对象生存期被分配的内存。如果忽略了该功能，结果会导致内存泄漏。

内存泄漏

当对象在其生命周期中没有正确释放被分配的内存时，一旦创建了该对象的应用程序还在执行，那么这块内存对整个系统来说是不可用的。例如，假设在一些循环中，创建了同一个类的多个对象，并且销毁了这些对象。如果这些对象在生命周期末期没有正确地释放占用的内存，内存泄漏会慢慢地减少系统可用内存池。某一时刻，所有的内存都被使用了，而系统则没有可用的内存供分配。这意味着在该系统中执行的其他应用程序则不能分配到任何内存。这将导致应用程序进入不稳定的状态，甚至会锁定系统。

5.4 在类中设计错误处理

正如设计构造函数一样，设计类如何处理错误也是极其重要的。第 3 章讨论了错误处理的细节。

每个系统一定都会遇到不可预见的问题。因此，忽略潜在的错误不是个好主意。一个优秀的类（或这样的任何代码）的开发者会预判潜在的错误并当错误发生时使用代码处理这些情况。

通用规则是应用程序应当绝不崩溃。当系统遭遇错误时，应当自身修复错误并继续执行，或者在不丢失用户的任何重要数据情况下友好地退出。

5.4.1 使用注释给类加上文档

在每个关于编程的书籍和文章、每次代码审查以及每次对优秀的设计的讨论中都有关于注释和文档的主题。不幸的是，注释和优秀的文档往往不受重视，甚至更糟的是完全被忽略。

大多数开发人员知道应该始终对代码添加文档描述，但通常并不想花时间这样干。然而，没有优秀的文档实践是不可能驱动出优秀的设计的。在类级别，范围可能足够小，开发者可以侥幸远离文档。然而，当类被传递给其他人进行扩展和（或）维护时，它可能会变成一个更大的系统（这种情况很常见），缺失正确的文档和注释会破坏整个系统。

前人已经讲过这些。优秀的设计的最重要的方面之一是，设计类时应该小心地记录过程。譬如 Java 和 .NET 的实现提供了特殊的注释语法来帮助文档化过程。请看第 4 章来查看相应的语法。

太多的文档

注意，注释过多也是个问题。太多的文档和（或）注释会成为背景噪音，而且可能会背离文档本来的目的。就像优秀的类设计一样，应保持文档和注释简单易懂。

5.4.2　构造可以合作的对象

我们可以放心地说，几乎没有完全隔离的类。几乎没有任何原因来构建一个不需要与其他类交互的类。这是基于类生命周期中的一个事实。即类将服务其他类，它也会请求其他类的服务。稍后的章节中我们会讨论几种类交互的方式。

在出租车司机例子中，出租车司机与管理人员并不是独立的实体。它们在多个层面上存在交互（见图 5-4）。

当设计类时，请确保你知道其他对象如何与之交互。

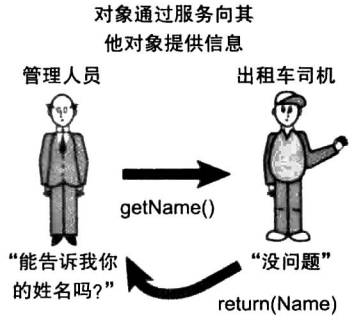


图 5-4　对象应该请求信息

5.5　设计时请考虑重用

可以在不同的系统中重用对象，编写代码时应该考虑重用性。例如，当开发和测试 Cabbie 类时，任何需要出租车司机的系统都可以使用该类。为了能在各种各样的系统中重用类，设计类时必须要考虑重用性。这正是为什么设计过程需要大量思考。试图预料操作 Cabbie 对象的所有可能的场景不是一个简单的任务，事实上它几乎是不可能的。

5.6　设计时请考虑扩展性

给一个类添加新特性应该与扩展一个已有的类、添加一些新方法、修改其他行为一样简单。没必要重写所有事情。这正是继承扮演的功能。如果编写了一个 Person 类，必须考虑事实上稍后你可能会写 Employee 类和 Vendor 类。因此，让 Employee 类继承 Person 可能是最好的策略。这种情况下，Person 类被称为可扩展的类。设计 Person 类时不应该包含一些行为阻止它可以被诸如 Employee 或 Vendor 类继承（假设在你的设计中，确实想让其他类扩展 Person）。例如，你不应该在 Employee 类中加入具体的监管功能。如果你这样做了，那么其他不需要监管功能的类继承自 Employee，就会出现问题。

这点触及了之前讨论过的抽象准则。Person 应该只包含具体某人的数据和行为。其他类可以成为它的子类从而继承相应的数据和行为。

> **什么样的属性和方法可以是静态的**
>
> 确定哪些属性和方法可声明为静态的相当重要。请回顾第 3 章关于使用 static 关键字的使用，理解设计类时如何使用它们。这些属性和方法会被类的所有对象共享。

5.6.1　使用描述性的名称

之前我们讨论了使用正确的文档和注释。为类、属性和方法遵循命名约定也是同一目

的。有很多命名约定，你选择哪种约定并不重要，重要的是选择一个并始终遵守。当选择了一种约定后，确保当你新建类、属性和方法名时，你不仅遵循了约定，而且名称具有含义。当一些人阅读名称时，他可以从名称看出该对象表示什么。很多组织往往会指定命名约定。

> **好的命名**
>
> 确保命名约定很重要。通常人们会创建对自身来说有意义的约定，但其他人则完全不能理解。当强迫别人执行约定时要小心。确保这些约定是有意义的，每个相关的人都能理解背后的意图。

保持命名具有描述性是优秀的开发实践，无论哪种开发范式中的都要执行这项实践。

5.6.2 抽象不可移植的代码

如果设计的系统必须使用不可移植的（原生的）代码（即代码只能运行在特定的硬件平台上），那么你应该把这些代码抽象到类中。通过抽象，可以将这些不可移植的代码隔离到各自的类中，或至少隔离在方法中（一个可以被重载的方法）。例如，如果编写代码来访问特定硬件的串行端口，你应该创建一个包装类来获取该信息，或者提供所需服务。不要在主类中包含依赖系统的代码（见图 5-5）。

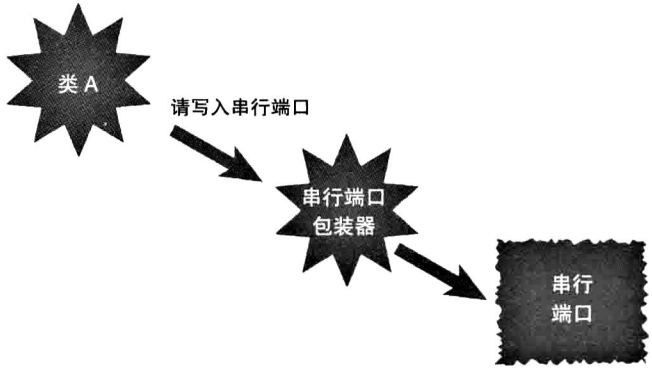


图 5-5 串行端口包装类

例如，考虑程序员需要直接与硬件交互的场景。这种情况下，不同平台的对象代码应该是完全不同的，因为这些代码必须针对每个平台进行编写。然而，如果这个功能是放置在一个“包装”类中，然后该类的用户可以通过接口直接与该类交互而不需要担心各种各样的底层代码。该包装类会处理平台的不同之处并调用相应的代码。

5.6.3 提供一种方式来复制和比较对象

第 3 章讨论了复制和比较对象的问题。理解如何复制和比较对象非常重要。你可能想

（或期望）做一个简单的浅拷贝或比较操作。你必须确定类行为是期望的行为，这意味着你需要花时间来设计如何复制和比较类。

5.6.4　保持尽可能小的作用域

保持类的作用域尽可能小与抽象和隐藏细节息息相关。尽可能的局部化属性和行为。这样维护、测试和扩展类都变得更加容易。

> **作用域和全局数据**
>
> 最小化全局数据是优秀的编程风格，这并不特定于面向对象编程。全局数据在结构化开发中是允许的，但它们是危险的。事实上，在面向对象开发中没有全局数据。静态属性和方法只在同一个类的对象间共享，然而对于其他类的对象则是不可见的。

例如，如果你有一个方法要求一个临时变量，请保持该变量本地化。请看以下代码：

```
public class Math {

    int temp=0;

    public int swap (int a, int b) {

        temp = a;
        a=b;
        b=temp;

        return temp;

    }

}
```

该类有什么问题？问题在于只需在 swap() 方法的作用域内需要 temp 属性。没有理由将 temp 属性放置到类级别。因此，你应该把 temp 的作用域移动到 swap() 方法的作用域中：

```
public class Math {

    public int swap (int a, int b) {

        int temp=0;

        temp = a;
        a=b;
        b=temp;

        return temp;
```

```
    }
}
```

这就是为什么要保持尽可能小的作用域。

5.6.5 类的职责与自身高度相关

在基于《Java Primer Plus》一书的训练课上，Tyma、Torok 和 Downing 认为类设计指导是所有对象的职责与自身应该高度相关。比如尝试画一个圆。

为了演示，让我们采用非面向对象的示例。在该例子中，我们把 Circle 作为参数传递给了打印命令，要求打印自身（见图 5-6）：

```
print(circle);
```

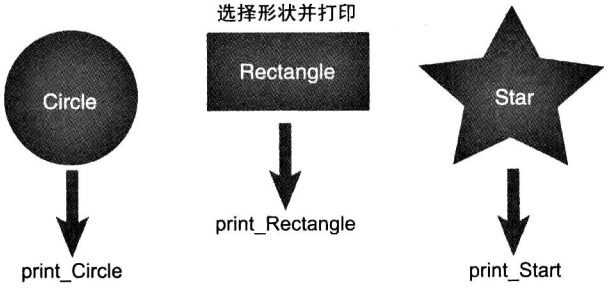


图 5-6 一个非面向对象的打印场景示例

print、draw 以及其他命令需要一个 case 声明（或类似的 if/else 结构）来确定对给定的形状采取什么样的行为。本例中，每个形状会调用单独的打印程序：

```
printCircle(circle);
printSquare(square);
```

每次添加一个新形状，所有方法都需要在各自的 case 声明中添加该形状：

```
switch (shape) {
    case 1:  printCircle(circle); break;
    case 2:  printSquare(square); break;
    case 3:  printTriangle(triangle);  break;
    default: System.out.println("Invalid shape.");break;
}
```

现在我们实现一个面向对象的示例。通过使用多态以及把 Circle 归类到 Shape 中，Shape 会清楚它是一个 Circle，并且知道如何打印自身（见图 5-7）：

```
shape.print(); // 该形状其实是圆形
shape.print(); // 该形状其实是正方形
```

请注意调用是完全一样的，而形状的上下文会指示系统的反应。

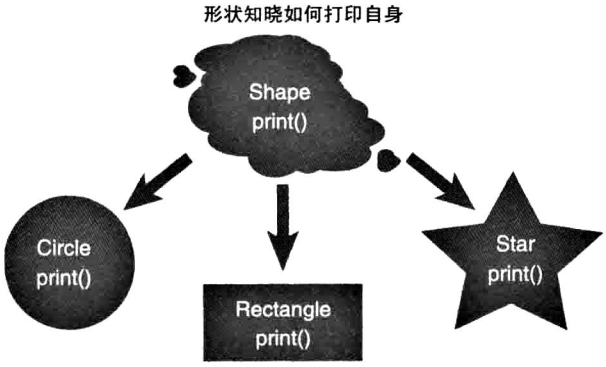


图 5-7　一个面向对象的打印场景示例

5.7　设计时请考虑可维护性

设计有用且简明的类促进了高层次的可维护性。正如设计类时需要考虑扩展性一样，你设计时需要考虑以后的可维护性。

设计类的过程强迫你组织你的代码为（理想情况下）很多可管理的块。把代码分块比大块代码更具维护性（至少应该这么想）。促进可维护性的最好方式之一是减少代码依赖，即修改一个类中的代码不会影响（或使影响最小化）其他类。

> **高度耦合的类**
>
> 高度依赖其他类的行为被称为高度耦合。即如果修改一个类会强迫修改另一个类，那么这两个类则可以说是高度耦合的。没有这样依赖的类则拥有低级别的耦合。可以通过 Scott Ambler 写的《The Object Primer》一书来学习该主题的更多信息。

如果一开始就正确设计类，任何修改系统的行为应当只修改对象的具体实现。应当不惜一切代价避免对公共接口的改变。对公共接口的任何更改将会导致使用该接口的所有系统引起涟漪效应。

例如，如果对 `Cabbie` 类的 `getName()` 方法做了修改，所有系统中使用了该接口的地方都需要进行修改并重新编译。找到所有的方法调用是个艰巨的任务，而且找不全的可能性非常高。

为了促进高度的可维护性，保持类的耦合度越低越好。

5.7.1　在开发过程中使用迭代

在大多数设计和编程过程中，一般都推荐使用迭代过程。这与提供最小接口的概念很吻合。基本上，这意味着不要一次性写完所有代码！用小步增长的方式来编写代码，每步都进行构建和测试。如果提供了不充足的接口，那么很快会无法覆盖期望的行为。通过这种方

式，我们可以迭代地设计类，使其拥有适合的接口。测试过程并不简单地仅限于编码。测试设计走查以及其他的设计回顾技术非常有用。测试人员更喜欢使用迭代过程，因为他们可以在早期就参与进来，而不是在开发过程的结尾才接触程序。

5.7.2 测试接口

接口的最小实现通常称为桩（stub）。（在《Java 面向对象设计》一书中 Gilbert 和 McCarty 对桩有精彩的讨论）。使用桩可以在没有编写任何真实代码的情况下测试接口。接下来的例子中，在验证接口是否正常工作时（从用户的角度来看，请记住接口视为用户服务），桩并没有连接到一个实际数据库。这时具体实现并不是必要的。事实上，这时完成实现会耗费宝贵的时间和精力，因为接口的设计会影响实现，而这时接口还未完成。

在图 5-8 中，注意一个 user 类向 DataBaseReader 类发送一条消息，返回给 user 类的信息是通过代码桩实现的，而不是实际的数据库。（事实上数据库甚至还不存在）。当接口完成后而实现处于开发阶段，这时可以连接数据库而不用连接桩。

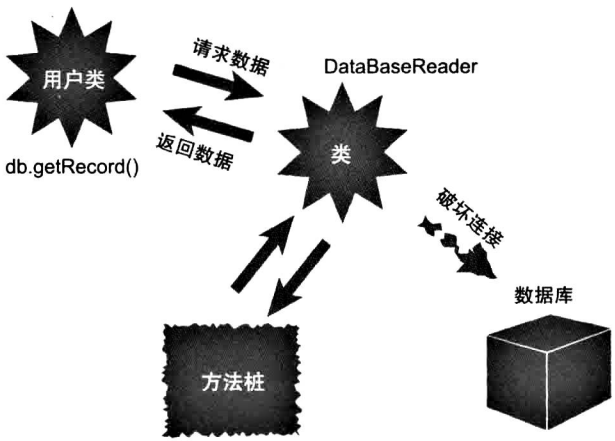


图 5-8 使用桩

以下的代码示例使用内部数组来模拟可以工作的数据库（虽然非常简单）：

```
public class DataBaseReader {

    private String db[] = { "Record1",
        "Record2",
        "Record3",
        "Record4",
        "Record5"};

    private boolean DBOpen = false;
    private int pos;
```

```
    public void open(String Name){
        DBOpen = true;
    }
    public void close(){
        DBOpen = false;
    }
    public void goToFirst(){
        pos = 0;
    }
    public void goToLast(){
        pos = 4;
    }

    public int howManyRecords(){
        int numOfRecords = 5;

        return numOfRecords;
    }
    public String getRecord(int key){

        /* DB Specific Implementation */
        return db[key];
    }
    public String getNextRecord(){

        /* DB Specific Implementation */
        return db[pos++];
    }

}
```

注意这些方法是如何模拟数据库调用的。数组中的字符串代表了将要写入数据库的记录。当成功地把数据库集成到系统中后，它将替换该数组。

保留桩

当使用完桩后，不要删除它。保留桩以便后续使用。确保用户不能看到它们。事实上，一个设计良好的程序中，设计时应该集成测试桩，保留在程序中以便后续使用。总之，设计类时要进行测试。

如果你发现接口设计有问题，可以进行修改并重复这样的过程直到得到满意的结果。

5.8 使用对象持久化

对象持久化是面向对象的系统必须解决的另一个问题。持久化是维护对象状态的概念。当运行程序时，如果没有使用某种方式保存对象，对象消亡以后将不会被恢复。在一些系统

中可以使用转瞬即逝的对象，但大多数业务系统中，必须要保存对象状态，以便后续使用。

对象持久化

尽管对象持久化主题以及下一小节的主题并不是真正的设计指导原则，但我认为设计类时必须要解决这些问题。我在这里介绍它们只是想指出在设计类的早期阶段就需要解决它们。

最简单的形式是通过序列化将一个对象写到平面文件中。而先进的技术则是基于 XML。尽管理论上对象可以持久化到内存中，只要它不被销毁即可，但我们仍需要将持久化的对象存储到一些存储设备中。有三种主要的存储设备可以考虑：

- **平面文件系统**。你可以通过序列化的方式把对象存储到平面文件中。这种使用很受限制。
- **关系型数据库**。可以使用一些把对象转换为关系模型的中间件。
- **面向对象的数据库**。这可能是持久化对象更高效的方式。但大多数公司此时都把数据保存在遗留系统中，而且并不想把关系型数据库转换为面向对象的数据库。

序列化以及封送对象

我们已经讨论过了在结构化程序设计的环境中使用对象的问题。我们编写对象保存到关系型数据库中时使用的中间件例子是个好例子。我们也涉及了把对象写入平面文件并通过网络传输的问题。

通过网线传送对象（比如通过网络传输文件），系统必须解构对象（压平它），通过网线发送，然后在网线另一端重新构造。这个过程被称为*序列化对象*。通过网线发送对象的艺术称为*封送对象*。理论上一个被序列化的对象可以写入平面文件中并稍后发送，状态与被系列化时保持一样。

主要问题是序列化和反序列化必须使用相同的规格。这有点像一个编码算法。如果对象加密了字符串，想解密它必须使用同一个编码算法。Java 提供了一个名为 `Serializable` 的接口，提供了这种转换。

C# .NET 和 Visual Basic .NET 提供了 `ISerializable` 接口，微软官方文档是这样描述它的：*允许一个对象控制自身的序列化和反序列化*。所有想要序列化的类都必须实现这个接口。以下列出了 C# .NET 和 Visual Basic .NET 的语法：

```
' Visual Basic .NET
Public Interface ISerializable

// C# .NET
public interface ISerializable
```

序列化的问题之一是它往往是私有的。而使用 XML（稍后会详细讨论）则不是私有的。

5.9 结语

本章提供了设计类的很多指导方针。这并不是一个完整的指导方针列表。毫无疑问在使用面向对象设计的过程中你可以加入新的指导方针。

本章讨论了对单个类的设计问题。我们已经看到没有类能完全与其他类隔离，设计类时必须考虑与其他类进行交互。一组相互交互的类组成了系统。最终，系统为终端用户提供价值。第 6 章涵盖了设计一个完整系统的主题。

5.10 引用

- Meyers, Scott. 2005. *Effective C++*, 3rd edition. Boston, MA: Addison-Wesley Professional.
- Ambler, Scott. 2004. *The Object Primer*, 3rd edition. Cambridge, United Kingdom: Cambridge University Press.
- Jaworski, Jamie. 1999. *Java 2 Platform Unleashed*. Indianapolis, IN: Sams Publishing.
- Gilbert, Stephen, and Bill McCarty. 1998. *Object-Oriented Design in Java*. Berkeley, CA: The Waite Group Press.
- Tyma, Paul, Gabriel Torok, and Troy Downing. 1996. *Java Primer Plus*. Berkeley, CA: The Waite Group.
- Jaworski, Jamie. 1997. *Java 1.1 Developers Guide*. Indianapolis, IN: Sams Publishing.

5.11 本章中使用的示例代码

以下是 C#.NET 版本的代码。其他语言（比如 VB.NET 和 Objective-C）的代码在出版社网站上有电子版。本章已经展示了这些例子对应的 Java 代码。

C# .NET 版本的 `TestMath` 示例代码

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace TestMath
{
    public class Math
    {

        public int swap(int a, int b)
        {

            int temp = 0;
```

```
            temp = a;
            a = b;
            b = temp;

            return temp;

        }

    }
    class TestMath
    {
        public static void Main()
        {
            Math myMath = new Math();
            myMath.swap(2, 3);
        }
    }
}
```

第 6 章 Chapter 6

使用对象进行设计

当使用软件产品时，你期望它和宣传的一样好。不幸的是，并不是所有的产品都符合期望。问题在于开发大量产品时，大部分时间和精力都花在了开发阶段，而不是设计阶段。

面向对象（OO）的设计已经被吹捧为健壮和灵活的软件开发方式。实际情况是你可以创建优秀的或者糟糕的面向对象设计，正如之前你可以创建优秀的或者糟糕的非面向对象设计一样。不要因为你正在使用一个先进的设计方法就认为你的设计就是正确的。你必须注意整体设计，投入适量的时间和精力来创造出尽可能最好的产品。

第 5 章中我们关注如何设计优秀的类。本章关注设计优秀的*系统*。*系统*可以被定义为相互交互的类。在软件开发的历史中，优秀的设计实践贯穿始终。无论是否使用了面向对象的技术，我们没有任何原因放弃使用充满热血的、甜蜜和泪水的历史软件。

利用之前的成就并不限于设计实践。你甚至可以把已存在的遗留代码包含到面向对象的设计中。很多情况下，可以把已经正常工作数年的代码通过迭代的方式包装到对象中。本章稍后会讨论*包装技术*。

6.1 设计指导

有个误区是只有一种正确的设计方法。事情绝不是这样。没有什么正确或错误的方式来创建设计。当今存在很多设计方法，各自都有支持者。然而，问题在于不是使用哪种设计方法，而是一直坚持使用一种方法。这点可以从设计扩展到整个软件开发过程。很多组织并没有遵循一个标准的软件开发过程，或者他们没有坚持使用。创建优秀的设计最重要的因素是找到一个你和你的组织都感到舒服的方式并且坚持使用它。实现一个没人愿意遵循的设计没

有任何意义。

与面向对象技术有关的大部分书籍都提供了相似的策略来设计系统。事实上，除了一些特定于面向对象技术的问题，大多数策略也同样适用于非面向对象的系统。

通常一个稳固的面向对象的设计过程包含以下步骤：

（1）进行正确的分析。

（2）编写工作陈述文档来描述该系统。

（3）通过规格说明收集需求。

（4）开发用户接口的原型。

（5）识别类。

（6）确定每个类的职责。

（7）确定类与类之间如何交互。

（8）创建一个高层次的模型来描述系统的构建。

在面向对象开发中，高层次的系统模型特别有趣。系统（或者对象模型）是由类和类之间的互动组成。该模型应该忠实地代表系统，而且容易理解和修改。我们也需要对该模型建立一种标记符号。这正是统一建模语言（UML）的由来。众所周知，UML 并不是一种设计过程，而是一个建模工具。本书中仅关注 UML 中的类图。我喜欢使用类图作为可视化工具来帮助设计，也可以作为文档。

持续的设计过程

想法和规划再好，大多数情况下设计是一个持续的过程。甚至产品已经处于测试阶段，仍然会出现设计上的变更。这取决于项目经理决定何时停止更改产品和添加功能。

了解存在很多设计方法论是很重要的。有个较早的方法论称为瀑布模型，提倡在不同的阶段建立严格的边界。在该模型中，设计阶段先于实现阶段，而实现阶段先于测试阶段，依次类推。实践中我们发现瀑布模型是不切实际的。当前其他的设计模型（比如快速原型、极限编程、敏捷、Scrum 等）倡导真正的迭代过程。在这些模型中，一些实现试图先完成设计阶段，作为一种概念验证。尽管厌恶瀑布模型，但该模型背后的目标是可以理解的。在开始编码之前先彻底完成设计只是传说中的实践。你肯定不想当处于产品的发布阶段时，又决定从设计阶段开始迭代。跨阶段边界迭代是不可避免的。然而，你应当保持这些迭代最小化（见图 6-1）。

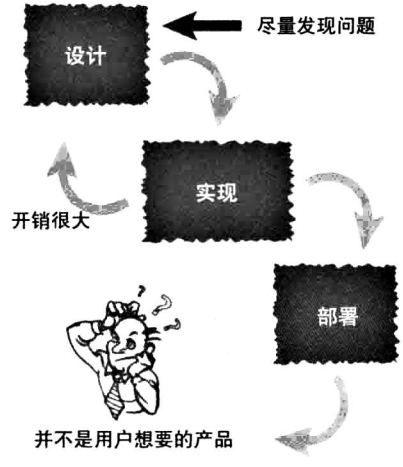


图 6-1 瀑布方法

简言之，早点识别需求并保持设计变更最小化的原因如下：

- 在设计阶段的需求 / 设计变更开销相对较小。
- 实现阶段的设计变更开销显著更高。
- 在开发阶段之后的设计变更开销与第一条相比非常庞大。

同样，在架构设计完成之前不会开始动工构造你的梦想之屋。如果我说开始建造金门大桥或纽约州大楼时并没有考虑设计问题，你可能会认为这太疯狂了。如果我告诉你你正使用的软件可能包含一些设计缺陷，可能没有经过完全测试，你很可能并不当回事。

事实上，彻底测试软件确保绝对没有任何缺陷存在是不可能的。然而，理论上彻底测试软件一直是目标。我们应当始终试图清除尽可能多的缺陷。桥和软件不能直接比较。然而，软件应该争取与“更难”的工程规范（比如桥梁建造）保持一样的工程卓越性。低质量的软件是致命的，它并不仅是薪水支票上的错误数字。例如，医疗设备上低劣的软件会害死或致残人。当然，你觉得现在以及以后每天重启计算器也能活得好好的。但决不能对桥梁损坏抱着同样的态度。

安全与经济

你想通过一座还没有被检查和测试的桥吗？不幸的是，很多用户都被赋予了对软件包进行大量测试的责任。对于用户和软件提供商来说成本都很大。不幸的是，在做项目决策时短期的经济因素常常是主要考虑因素。

因为顾客似乎愿意付出有限的价格来忍受低质量的软件，一些软件提供商发现让广大客户测试自己的软件比自己测试要划算。短期来看可能是这样，但长期来看成本远超软件提供商的预期。最终会损害软件提供商的信誉。

一些计算机软件公司乐意在试用版测试阶段让客户来做测试。理论上来说，在测试版交付客户之前就应当完成测试。很多客户乐意承担使用预发布软件的风险，因为他们渴望使用产品许诺的功能。

在软件发布后，修复在发布之前发现的问题将变得更加费事。比如考虑汽车公司遭遇召回的情形。在发货前就能定位汽车的缺陷并进行修复（理想情况下在制造时就能发现），那么耗费比把所有已经交付的汽车一次性召回并修复要小得多。不仅是因为这种方式代价很高，它也会损害公司的声誉。在竞争日趋激烈的市场中，高质量的软件、支持服务以及声誉是竞争优势（见图 6-2）。

以下小节列出了对上述内容的简短总结条目作为设计过程的一部分。本章稍后会通过一个示例来充分解释以下每个条目的细节。

6.1.1　提供正确的分析

在设计和创造软件产品时会引入很多变量。用户必须在所有阶段与开发人员紧密合作。在分析阶段，用户必须和开发人员进行正确的研究和分析来确定工作陈述文档、项目需求，甚至是否真的需要实施该项目。你看到最后一点可能有点惊讶，但它是非常重要的。在分析

阶段，如果没有有效的理由来做该项目，那么可以毫不犹豫地中止该项目。很多时候抱着实验心态或者一些惯性保持项目前进，而不关心项目会被取消的警告信号。假设项目是可行的，分析阶段主要关注点是让每个人学习这些系统（包括旧系统以及新系统原型）来确定系统需求。

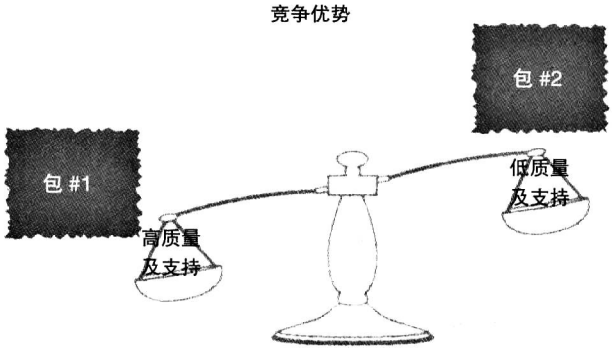


图 6-2 竞争优势

通用软件准则

大多数情况下，这些实践并不特定于面向对象领域。它们通常都适用于软件开发过程。

6.1.2 编写工作陈述文档

工作陈述（SOW）是描述系统的文档。尽管确定需求是分析阶段的终极目标，但分析阶段需求还未成为最终格式。任何人通过 SOW 能够完全理解整个系统。不管 SOW 是如何编写的，它必须代表完整的系统，而且很清楚这个系统的外观和用法。

SOW 包含了与该系统相关的所有知识。很多客户创建了提案申请（request for proposal，RFP），这与工作陈述文档比较类似。客户可以创建一个 RFP，其完整描述了客户想要构建的系统，然后发布给多个供应商。这些供应商然后使用该文档，一起分析出他们需要做的事情，从而决定是否应当投标该项目，以及如果投标需要的花费。

6.1.3 收集需求

需求文档描述了用户期望的系统。尽管需求文档的详细级别不需要具备高度的技术细节，但需求必须是详尽的，能充分代表用户对最终产品所需的本质。需求文档必须为用户对系统的完整性做出明智判断提供足够的细节。它也必须为设计团队使用该文档在设计阶段进行设计提供具体细节。

而 SOW 是由段落的形式（甚至是叙述形式）编写的文档，需求通常被表示为汇总表或者分项条目。每个单独的条目代表了该系统一个特定的需求。需求是从工作陈述文档中蒸馏

出来的。本章稍后会展示该过程。

需求是系统最重要的部分。SOW 可能会包含不相干的材料，然而需求是系统必须要实现的最终表现。软件开发过程中的所有未来的文档都要基于需求。

6.1.4　开发用户接口的原型

确保用户和开发人员理解系统的最好方式之一就是创建一个原型。原型可以是任何东西，但大多数人认为原型应该模拟用户接口。通过创建实际的界面和界面流程，让人们更容易理解他们将要使用的系统，以及使用该系统的感觉。大多数情况下，原型不会包含最终系统的所有功能。

我们会通过集成开发环境（IDE）来创建大多数产品原型。然而，在白板上甚至白纸上绘制界面有时就足够了。传统情况下尽管有其他语言，但 Visual Basic .NET 是一个创建原型的非常棒的环境。请记住当创建原型时你无须实现业务逻辑（接口背后的逻辑 / 代码实际在处理业务逻辑），尽管你可以这样做。当前最大的问题是用户接口的外观和使用感受。优秀的原型可以帮助识别类。

6.1.5　识别类

在需求文档后，就可以开始识别类了。从需求中识别类的一种简单的方式就是将名词高亮出来。这些名词可以代表对象，譬如人、地方及其他东西。不要大惊小怪一开始就识别出所有类。你可能惧怕整个设计过程中的诸多阶段会消除类、添加类以及修改类。先识别类是非常重要的。事实上设计是一个迭代的过程。这有点像另一种方式的头脑风暴，先初始化一些事情，当完全理解整个结果后，很可能已经与最初的样子完全不同。

6.1.6　确定每个类的职责

你需要确定识别出来的每个类的职责。这包括该类必须存储的数据以及该类必须执行的操作。例如，`Employee` 对象要有计算薪水并转账到相应账号的职责。它也许拥有存储多种薪水标准以及多个银行账号的职责。

6.1.7　确定类之间如何协作

大多数类都不是孤立的。尽管类必须满足某些职责，大多数情况下它不得不与其他类交互以获取其需要的东西。所以类之间需要发送消息。当一个类需要另一个类提供的信息时，会给这个类发送一个消息。或者其他类也会给它发消息来获取信息。

6.1.8　创建类模型来描述系统

当识别出了所有类，并且这些类的职责和协作都已清晰，那么可以构造一个类模型来表

示整个系统。类模型展示了各种各样的类如何与系统交互。

本书中，我们使用 UML 来模型化系统。一些商业工具使用 UML 并且提供了友好的环境来创建和维护 UML 类模型。通过以下小节编写的代码，可以看到类图如何与整体结合起来。如果没有一些好用的建模注解和模型化工具，模型化大型系统则是一个几乎不可能的任务。

6.1.9 建立用户接口原型

在设计过程中，我们必须创建用户接口的原型。该原型提供的有价值的信息可以指导迭代整个设计过程。Gilbert 和 McCarty 在《Java 面向对象设计》一书中说过，“对于系统用户来说，用户接口就是这个系统本身”。创建用户接口原型有多种方式。你可以在纸或者白板上描绘用户接口。也可以使用特定的原型工具，甚至使用 Visual Basic 之类的编程环境。Visual Basic 经常被用于快速创建原型。或者你也可以选择你最喜爱的 IDE 开发工具来创建原型。

然而当开发用户接口原型时，确保用户看到和感受到的是最终的产品形态。

6.2 对象包装

在之前章节中有好几次我都声明本书的主要目的之一是反驳面向对象编程与面向过程编程是完全不同的范式这一谬论，甚至它是面向过程的反面。事实上我已经说过了，我经常问以下问题：“*你是一个面向对象的程序员还是面向过程的程序员？*”答案永远是一致的：我都是！

我认为不使用结构化的代码无法写出程序。因此当你使用面向对象的编程语言以及所谓的面向对象的设计技术来编写程序时，你也可以使用结构化的编程技术。我们无法避免这一点。

例如，当创建一个包含属性和方法的新对象时，这些方法中会包含结构化代码。事实上，我甚至会说这些方法中大部分都是结构化代码。这种方法和我们之前章节中遇到的容器概念是相吻合的。事实上，当编写方法中的代码时，编程过程与使用结构化的语言（比如 COBOL、C 等）时没什么大的不同。当然并不是完全一样，因为我显然不得不调整到一些面向对象的结构中。然而，在方法层面编写代码的根本方式实际上没怎么变过。

现在我再问这个问题：“*你是面向对象的程序员还是面向过程的程序员？*”我经常说*编程就是编程*。我认为作为一名优秀的程序员意味着理解基本的编程逻辑，并且对写代码充满激情。你常常会看到一个程序员拥有特定的一组技能，那我们就谈谈一种特定的语言，Java。

尽管我完全理解组织在紧要时刻需要一名有经验的 Java 程序员的迫切之心，但长远来看我更欣赏组织能专注于雇用一名拥有大量编程经验的程序员，而且当有新技术兴起时他能快

速学习和调整。我相信当我雇用人时，我更看重他的学习潜力，而不是他当前知道多少。激情也是非常重要的，因为它确保了雇员会保持探索新技术以及新的开发方法。

6.2.1　结构化代码

尽管程序逻辑的根本可能充满争议，我强调根本的面向对象结构是封装、继承、多态和组合。在我看过的很多书中，结构化编程的基本结构是序列、条件判断和迭代。

序列部分是给定的，因为它看来合乎逻辑。从顶部开始，然后有逻辑地运行到底部。我认为结构化编程由条件判断和迭代构成，分别称为 if 语句以及循环。

请看以下代码，从 0 开始循环 10 次，如果等于 5 则打印出一个值：

```
class MainApplication {

    public static void main(String args[]) {

        int x = 0;

        while (x <= 10) {

            if (x==5) System.out.println("x = " + x);

            x++;
        }

    }

}
```

如果用面向对象的语言来编写该段代码，那么主函数中的代码是结构化的代码。这里呈现了结构化编程的三个基本要素：序列、条件判断和迭代。

序列部分很容易识别，因为以下是执行的第一行：

```
int x = 0;
```

当执行完毕后，下一行会被接着执行：

```
while (x <= 10) {
```

依次类推。总的来说，这是一个真实的自顶而下的编程方式。从第一行开始执行，然后以此类推。

这里也有一个 if 语句：

```
if (x==5)
```

最后，有一个循环来完成结构化三重奏：

```
while (x <= 10) {
    }
```

实际上整个循环也包含了一个判断条件：

```
(x <= 10)
```

可以只使用这三种结构就可以编写很多代码。事实上，包装的概念与结构化编程很类似，只不过它用于面向对象编程中。在结构化设计中，我们将代码包装到函数（比如本例中的 main 方法）中，而在面向对象的设计中，我们将代码包装到对象和方法中。

6.2.2 包装结构化代码

尽管定义属性也属于编码（例如，创建一个整型），但对象的行为主要存在于方法中。并且这些方法中包含了大部分代码逻辑。

在图 6-3 中，一个对象包含方法，这些方法包含了代码，这些代码既可以是变量声明，也可以是条件判断或者循环。

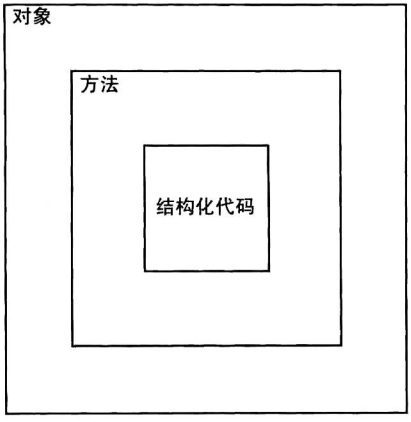


图 6-3 包装结构化代码

例如，我们编写一个功能来包装加法。我们创建了一个名为 add 的方法，其接受两个整型参数，并返回这两个参数的和：

```
class SomeMath {

        public int add(int a, int b) {

                return a + b;

        }

}
```

正如你所见，用于执行加法（$a+b$）的结构化的代码被包装在 add 方法中。这是个极其简单的例子，但它包装了结构化的代码。因此，当用户想使用该方法时，只需使用该方法的签名即可，如下所示：

```
public class TestMath {

        public static void main(String[] args) {

                int x = 0;

                SomeMath math = new SomeMath();

                x = math.add(1,2);

                System.out.println("x = " + x);
```

```
        }
}
```

最后，让我们来看一个更加有趣和复杂的示例。假设我们想在一个方法中计算某个数的斐波那契的值，我们可以添加一个方法，如下所示：

```
public static int fib(int n) {
    if (n < 2) {

        return n;
    } else {
        return fib(n-1)+fib(n-2);
    }
}
```

这里展示了面向对象的方法中包含（包装）了结构化代码。因为 fib 方法包含了条件判断、递归等。正如在简介中提及的一样，是可以在包装器中包含已有的遗留代码。

6.2.3　包装不可移植的代码

对象包装的另一用途是隐藏不可移植的（或原生的）代码。这里是将只能在一个平台（或一些平台）执行的代码包装到一个方法中，为使用该代码的程序员提供一个简单的接口。

比如有一个让计算机发出声音的任务，让计算机发出嘟嘟的声音。在 Windows 平台中我们可以执行以下代码让计算机发出嘟嘟声：

```
System.out.println("\007");
```

为了免除程序员记住这行代码（或者查找它），你可以提供一个名为 Sound 的类，该类包含一个名为 beep 的方法，如下所示：

```
class Sound {

        public void beep() {

                System.out.println("\007");

        }

}
```

注意程序员无需知道发声的代码是什么，他只需使用该类调用 beep 方法即可：

```
public class TestBeep {

        public static void main(String[] args) {

                Sound mySound = new Sound();

                mySound.beep();
```

```
        }
    }
```

这种方式不仅仅是让程序员用起来更加方便，还可以扩展该类包含发出其他声音的功能。更重要的是，如果该代码被用于非 Windows 平台，可以保持一致的接口，而构建 Sound 类的团队来负责处理不同平台之间发声的具体细节。对于在应用程序中使用该类的程序员来说，他不用关心这些不同之处，因为调用都是 beep 方法。

6.2.4 包装已有类

尽管包装遗留的结构化代码或者甚至不可移植的代码到一个新的（面向对象）的类中的原因比较明显，但包装已有类的需求则不那么明显。然而有很多原因来为已有（面向对象的）类创建包装对象。

软件开发人员经常使用其他人编写的代码。可能这些代码是从供应商买来的，或者是同一个组织内部其他人写的。很多情况下这些代码不能被修改。比如编写这段代码的人离职了，或者供应商不能提供维护更新等。这里就要显露出包装的力量了。

方式是将一个现有的类包装到一个新类中，以便修改它的实现或接口。这正如我们处理结构化的代码和不可移植的代码一样。不同之处在于，我们不仅建立起了面向对象的体系，而且还修改了其实现或接口。

为什么要这样做呢？答案是实现和接口会说谎。

比如第 2 章中使用的数据库例子。我们的目标是为使用不同数据库的开发人员提供一致的接口。事实上，如果需要支持其他数据库，仍然要保持一致的目标，即对新数据库的转换对用户来说是透明的（见第 2 章图 2-3）。

而且，还记得之前讨论过的关于创建中间件来为对象和关系型数据库提供一个接口吗？作为开发人员，我们想使用对象。因此我们想有将对象持久化到数据库中的功能。而我们不想为每个对象都编写 SQL 语句来存储到关系型数据库中。所以我们才考虑使用一个中间件来作为包装器，而我们可以使用很多支持建立对象 – 关系映射的产品。第 12 章会涉及更多细节。

从概念上来说，接口和实现范式的终极案例是第 2 章中讨论的发电厂例子（图 2-1）。在这个例子中，我们可以相互置换（包装），可以改变插线板来修改接口，亦可以改变电力生产设施来修改实现。

在软件开发过程中使用包装是非常普遍的，不仅开发人员会大量使用，而且供应商也会这样。当开发软件系统时包装器是个很重要的工具。

在本章中，我们探究了多种设计指导。包括编写新代码以便利用以前的代码，无论已有的代码是自己的还是供应商的。某些情况下包装甚至成为一种设计范式。比如很多情况下使用包装作为一种设计模式。稍后我们会看到装饰器模式会包装实现，而适配器模式则修改接口。第 15 章会讲述更多关于设计模式的细节。

6.3 结语

本章涵盖了一个完整系统的设计过程。请注意面向对象和结构化的代码并不是互斥的。实际上，不用结构化的代码你无法创建对象。因此，当构建面向对象的系统时，在设计中我们依然使用了结构化技术。

包装对象用来包装各种类型的功能，这包括传统的结构化（遗留的）代码和面向对象的代码（类）以及不可移植的（原生）代码。对象包装的主要目的是为使用代码的程序员提供一致的接口。

接下来的几章中，我们将探索类之间的关系。第 7 章覆盖了继承和组合这两个概念，以及相互之间的关系。

6.4 引用

- Ambler, Scott. 2004. *The Object Primer,* 3rd edition. Cambridge, United Kingdom: Cambridge University Press.
- McConnell, Steve. 2004. *Code Complete: A Practical Handbook of Software Construction,* 2nd edition. Redmond WA: Microsoft Press.
- Gilbert, Stephen, and Bill McCarty. 1998. *Object-Oriented Design in Java*. Berkeley, CA: The Waite Group Press.
- Jaworski, Jamie. 1999. *Java 2 Platform Unleashed*. Indianapolis, IN: Sams Publishing.
- Jaworski, Jamie. 1997. *Java 1.1 Developers Guide*. Indianapolis, IN: Sams Publishing.
- Wirfs-Brock, R., B. Wilkerson, and L. Weiner. 1990. *Designing Object-Oriented Software*. Upper Saddle River, NJ: Prentice-Hall.
- Weisfeld, Matt and John Ciccozzi. September, 1999. "Software by Committee," *Project Management Journal* v5, number 1: 30–36.

Chapter 7 第 7 章

精通继承和组合

继承和组合在面向对象（OO）系统中扮演着重要的角色。事实上，最困难及最有趣的设计决策就是决定使用继承还是组合。

当近几年发明了面向对象设计以后，这种决定变得越来越有意思。继承大概是最有意思的争议之一。尽管继承是面向对象开发（支持继承的语言可以认为就是面向对象的语言）的基本结构之一，但越来越多的程序员倾向于使用其他设计策略来替代继承。

其实继承和组合都是一种重用机制。继承正如其名字所示，继承其他类的属性和行为。这是一种父子关系。子（子类）直接继承父（或超类）。

组合，也正如其名称所示，即使用其他对象来构建新对象。在本章中，我们将探索继承和组合之间的显著差异和细微不同。首先，我们应当考虑在适合的场景下使用彼此。

7.1 重用对象

继承和组合的主要目的大概就是对象重用。简单来说，通过继承和组合来利用其他类就可以构建新的类（其最终变为对象）。实际上这是重用以前构建好的类的唯一方式。

继承代表了 is-a 关系。第 1 章中讲述过该关系。比如狗是哺乳动物。

组合则是使用其他类来构建出更加复杂的类，即其是一个装配集合。这种情况下没有父子关系。基本上复杂的对象都是由其他对象组合而成。组合代表了 has-a 关系。例如，汽车拥有一个引擎。引擎和汽车是分离的、独立的对象。然而，汽车是一个复杂的对象，包含（拥有）了引擎对象。事实上，子对象本身也有可能由其他对象组成。比如，引擎包括气缸。即引擎拥有一个（实际上有多个）气缸。

当面向对象的技术刚成为主流时，继承往往是如何设计一个面向对象系统的第一个示例。你会先设计一个类，然后继承它的功能，这是使用面向对象技术的最突出的优势之一。重用是目的，而继承则是重用的终极展示。

但是随着时间的推移继承的光环逐渐褪去。事实上，之前有些讨论中使用继承本身都充满疑问。在 Peter Coad 和 Mark Mayfield 撰写的《 Java Design 》一书中有一整章，标题为“使用组合而不是继承来进行设计”。很多早期的基于对象的平台甚至并不真正支持继承。当 Visual Basic 演化到 Visual Basic .NET 时，早期的基于对象的实现并不具有严格的继承能力。诸如 MS COM 模型等平台都是基于接口继承。第 8 章讲述了接口继承的细节。

如今对继承的使用仍然是充满争议的主题。抽象类是一种继承方式，很多语言并不直接支持此特性，比如 Objective-C。即使接口提供不了抽象类的全部功能，我们还是使用接口。

好消息是关于是使用继承还是组合的争论逐渐变得理性。正如所有明智的质疑一样，两边都有充满激情的观点。幸运的是，往往这些讨论导致我们更加清晰地理解如何优化该技术。

稍后章节中我们会看到人们为何相信应当避免使用继承，而采用组合作为设计方式。该观点既复杂又微妙。实际上，继承和组合都是有效的类设计技术，在面向对象开发者的工具库中都占有一席之地。至少你需要了解彼此，从而做出正确的设计决策。

实际上，继承很容易被误用和滥用。缺乏对继承的理解，结果导致完全使用继承作为设计策略，应用程序底层就会存在缺陷。

构建面向对象的系统中继承和组合都是重要的技术。设计者和开发人员需要花时间理解两者的优点和缺点，并且学习如何在正确的上下文中使用彼此。

7.2　继承

第 1 章已经对继承做了定义，即子类可以从父类中继承属性和行为。然而，这里我们会进一步讨论继承，探索有关继承的更多细节。

第 1 章指出一个简单的规则来确定继承关系。即如果你说类 B 是类 A，那么该关系很可能就是继承关系。

> **is-a**
>
> *面向对象设计的主要规则之一就是公共继承代表了 is-a 关系。*

我们再回顾一下第 1 章中的哺乳动物示例。有一个 Dog 类。狗拥有一些行为来确定它是一只狗，从而与猫区别开来。本例中指定了两个行为，狗会汪汪叫，也会喘粗气。所以我们创建了一个名为 Dog 的类，并且包含了这两个行为，而且具有两个属性（见图 7-1）。

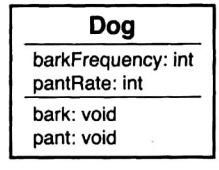


图 7-1　Dog 类的类图

接下来我们想创建一个 GoldenRetriever（金毛寻回犬）类。我们可以创建一个新类，具有 Dog 类的相同行为。然而，我们可以得出一个合理的结论，金毛寻回犬是一只狗。基于这样的关系，我们可以从 Dog 类中继承属性和行为，从而在新的 GoldenRetriever 类中使用它们（见图 7-2）。

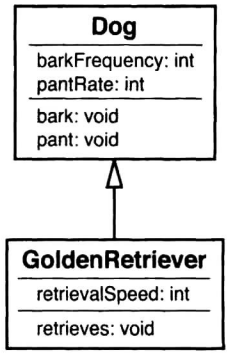


图 7-2 GoldenRetriever 类继承自 Dog 类

GoldenRetriver 类既包含了普通狗的行为，也拥有自己的特有行为。这给我们提供了很多便利。首先当编写 GoldenRetriever 类时，无需重复编写 bark 和 pant 方法。这不仅节省了设计和编码时间，而且节省了测试及维护时间。只需编写一次 bark 和 pant 方法，而且假设当编写 Dog 类时我们已经进行了正确测试，那么就无需再次测试。但是对于新接口还是需要重新测试。

现在来了解继承结构的全部优势。我们来创建基于 Dog 类的第二个类，名为 LhasaApso（拉萨阿普索犬）的类。金毛寻回犬善于寻回，而拉萨阿普索犬是一种警戒犬。这些狗并不是攻击性的狗。它们反应非常灵敏，当感觉到异常的事情时会吠叫。所以正如 Golden-Retriever 类一样，可以创建 Lhasa-Apso 类并继承自 Dog 类（见图 7-3）。

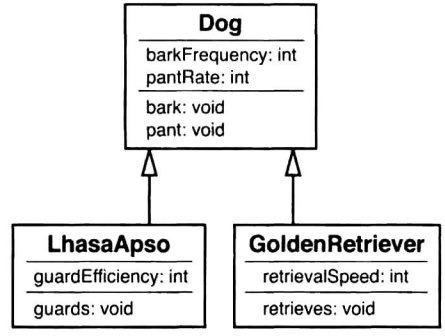


图 7-3 LhasaApso 类继承自 Dog 类

测试新代码

GoldenRetriever 类的例子中，当编写 Dog 类时，我们就应该编写 bark 和 pant 方法，并完成测试和调试。理论上来说，这些代码是健壮的，并且能在其他情况下重用。然而事实上你无须重写这些代码并不意味着你可以不用再做测试。比如对于寻回犬而言可能有一些特殊特征不符合这些代码。正确做法是应该始终测试新代码。每个新的继承关系使用继承而来的方法时会创建新的上下文。完整的测试策略是基于这些上下文做测试。

继承的另一个主要优势是只存在一份 bark() 和 pant() 的代码。如果有需求修改 bark() 方法中的代码，我们只需在 Dog 类中修改即可，无须修改 LhasaApso 类和 GoldenRetriever 类。

你能发现什么问题吗？目前来看继承模型没啥问题。然而你能确定所有的狗都有 Dog 类中包含的行为吗？

在《Effetcive C++》一书中，Scott Meyers 举了一个好例子来说明使用继承有多么左右为难。比如创建一个关于鸟的类，对于鸟来说最广为人知的特征就是鸟会飞。所以我

们可以创建一个名为 Bird 的类，拥有一个 fly 方法。你可能会立即意识到一个问题，企鹅和鸵鸟呢？它们是鸟，但它们不会飞。你可以局部重载该行为，但该方法仍叫 fly。而且对于一个不会飞，只会摇摆着走、跑或者游泳的鸟，拥有一个名为 fly 的方法确实不太合适。

这会导致一些潜在的重大问题。例如，如果企鹅拥有 fly 方法，那么企鹅则会尝试测试该方法。然而，如果 fly 方法实际已经被重载，并不真实存在飞行的行为，企鹅在跳下悬崖后调用 fly 方法会得到一个大大的“惊喜”。想象一下，企鹅调用 fly 方法后结果不是飞行，而是蹒跚步行后它有多么气恼。这种情况下，蹒跚学步不会杀了它。但假设这样的代码存在于航天飞机的导向系统中会带来什么后果？

在狗的例子中，我们设计所有的狗都有吠叫的能力。然而一些狗是不会叫的。巴辛吉品种的狗是哑巴狗。虽然这些狗不会叫，但它们会用假嗓音叫。所以我们是否应该重新评估我们的设计？新的设计应该是什么样的？图 7-4 展示了 Dog 类的更正确的模型继承体系图。

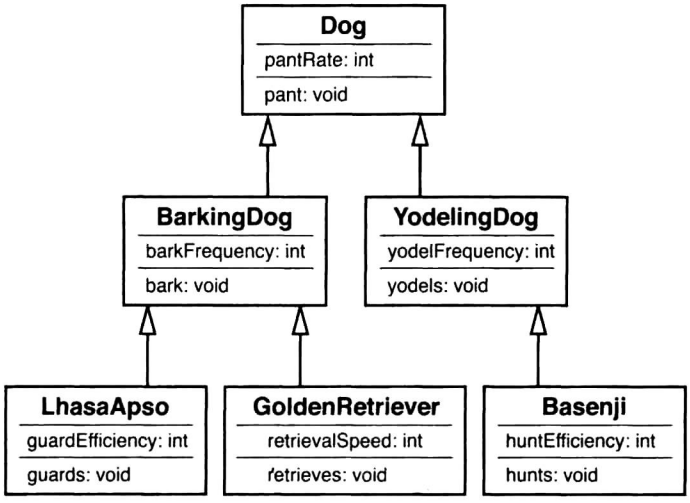


图 7-4　Dog 类继承体系图

7.2.1　通用和特例

我们来设计 Dog 类模型继承体系图。先创建一个名为 Dog 的类，我们找出了各种各样品种的狗的共同之处。该概念有时被称为通用到特例，这是使用继承时的又一个重要的考虑因素。即在继承树中越往下，事情变得越特殊。最通用的事情位于树的顶端。在 Dog 继承树中，Dog 类是树的顶端，并且是最通用的分类。其他品种的类（GoldenRetriever 类、LhasaApso 类和 Basenji 类）更具体。继承就是根据通用因素从通用到具体的过程。

在 Dog 继承模型中，我们刚开始通过找出共同的行为来建模。比如尽管黄金寻回犬和拉萨阿普索犬拥有不同的行为，但这些品种的狗有一些共同的行为，比如它们都会吠叫和喘

气。然后我们意识到不是所有的狗都会吠叫，有些会用假嗓音叫。因此，我们不得不把吠叫行为放置到单独的 BarkingDog 类中，而将用假嗓音叫的行为放置到 YodelingDog 类中。然而，我们意识到吠叫的狗和不会吠叫的狗仍然共享一些通用的行为，比如它们都会喘气。因此我们保留 Dog 类，让 BarkingDog 和 YodelingDog 类继承自它。现在巴辛吉狗可以继承自 YodelingDog，而 LhasaApso 和黄金寻回犬可以继承自 BarkingDog。

我们也可以决定不为 BarkingDog 和 YodelingDog 创建两个截然不同的类。我们可以为每个独立品种的类实现各自的吠叫和假嗓音叫，因为每种狗的叫声是不同的。这只是众多设计决策的其中一个示例。大概最好的方案是将吠叫和喘气作为接口实现。第 8 章会讨论这一点。

7.2.2 设计决策

理论上来说，找到越多的共同点越好。然而有时过于精确也会存在设计问题。尽管找出尽可能多的共同点更加靠近真实生活，然而并不表示更靠近模型。找出的点越多，系统会越复杂。所以你会陷入一个难题，到底想要更加精确的模型还是复杂度低的系统？你需要根据情况进行决策，有一些简单的指导性建议可以帮助你做决策。

> **计算机不擅长的事**
>
> 显而易见，计算机模型只能近似描述真实世界的情况。计算机善于捣鼓数字，但不善于抽象操作。

例如，将 Dog 类分解为 BarkingDog 和 YodelingDog 比假设所有的狗都会吠叫更接近现实世界模型，但同时也增加了复杂度。

> **模型复杂度**
>
> 本例中多加两个类不会复杂到无法驾驭模型。但在大型的系统中，如果不停地做这样的决策，那么复杂度会急剧上升。在大型系统中，尽可能保持简单往往是最佳实践。

可以通过实例来讲解过于精确的模型无法维持较低的复杂度。假设你作为动物饲养员想构造一个系统来记录所有的狗。该系统模型包括了会叫的狗和不会叫的狗，看起来还可以。然而假设你没有饲养任何不会叫的狗，以前没养过，以后也不会养。那么你可能就无需引入复杂度来区别会叫的狗和不会叫的狗。这会减少系统复杂度，并且依然能满足功能需求。

决定在设计时引入更小的复杂度或者更多的功能是一项平衡艺术。我们的主要目标始终是创建一个系统，该系统保持一定的弹性，引入复杂度时不至于被自身压垮。

当前还有未来的开销因素也是决策的主要因素。新加功能尽管让系统更加完整和灵活，但可能勉强带来了一点好处，投资回报率却很低。例如，假设需要扩展 Dog 系统，引入其他的犬科动物，比如鬣狗和狐狸会怎么样（见图 7-5）？

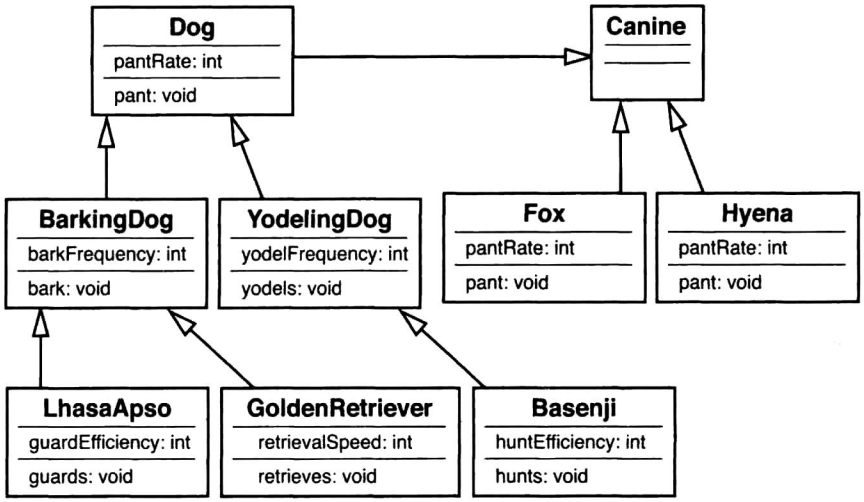


图 7-5　扩展后的犬科模型

如果你是动物园主那么这样的设计是精巧的，但如果你只是饲养或售卖宠物狗，那么这样 Canine 类扩展是不必要的。

所以可以看到，进行设计时始终需要找到一个平衡点。

> **做设计决策时请考虑未来**
>
> 你可能会说："永远不要说永远。"尽管你现在不饲养不会叫的狗，未来某个时刻你可能会干。如果你当前不考虑设计不会叫的狗的可能性，那么将来再对系统进行修改时引入这个功能带来的开销会更大。这是你需要进行设计决策的问题之一。你可能会重载 bark() 方法来使其不会叫。然而这违背直觉，因为人们调用 bark() 方法时期望它实际会进行吠叫。

7.3　组合

一个对象包含其他的对象是很自然的。电视机包括按钮和显示器。计算机包含显卡、键盘以及驱动。计算机可以看作一个代表自身的对象，而闪存驱动也是一个有效的对象。你可以拆开计算机把硬盘移除下来并拿到手上。实际上你可以把这块硬盘安装到其他计算机上。硬盘是一个独立的对象，因为它可以在不同的计算机上工作。

对象组合的经典例子是汽车。很多书、培训课程以及文章中都使用汽车作为对象组合的示例。除了步枪制作流水线外，大多数人认为 Henry Ford 创造的汽车装配流水线是可替换性的经典例子。因此汽车自然成为设计面向对象软件系统时的主要引用示例。

大多数人认为汽车包含引擎是很自然的事情。然而汽车除了引擎还包括其他对象，比如

轮子、方向盘以及立体音响系统。一个特定的对象由其他对象组合而成，而且这些对象被作为对象属性，那么这个新的对象被称为混合（或者聚合，或者综合）对象（见图 7-6）。

聚合、联合以及组合

在我看来重用类只有两种方式，继承和组合。第 9 章中我们讨论了组合的更多细节，具体来说就是聚合和联合。尽管大家对聚合和联合有多种观点，但在本书中我认为聚合和联合是组合的方式。

使用 UML 表示组合

UML 使用图 7-7 中的标记来模型化展示汽车对象包含一个方向盘对象这一事实。

聚合、联合以及 UML

本书中，UML 中的聚合以带线的菱形表示，比如引擎是汽车的一部分。联合则只由一根线（没有菱形）表示，比如独立的键盘服务于一个单独的计算机。

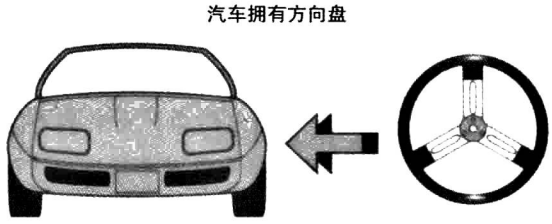


图 7-6 组合示例

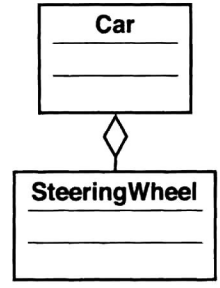


图 7-7 使用 UML 表示组合

注意，连接 Car 类和 SteeringWheel 类的线在 Car 类这一端带有一个菱形。这表示了汽车包含（has-a）方向盘。

接下来扩展该示例。假设在该设计中我们不对任何对象使用任何方式的继承。所有的对象关系都是严格的组合关系，并且有多层组合。当然这是一个极其简单的例子，设计汽车时会有越来越多的对象及对象关系。然而这个设计只为了简单演示什么是组合。

这里我们认为汽车是由一个引擎、一个立体音响系统和一个门组成。

有多少门和立体音响系统

注意车通常有多个门。有些有两个门，有些有 4 个门。你甚至会认为带后备厢的车有第五个门。同样，并不是所有的车都有立体音响系统。一个车可以没有音响系统，也可以有一个。我甚至看过一个车拥有两个独立的立体音响系统。第 9 章我们讨论了这些细节。所以在这个例子中，我们假装一个车只有一个门（比如它是一辆特殊的赛车），有一个立体音响系统。

很容易理解汽车是由引擎、音响系统及门组成的，因为大多数人都这么认为。然而，重要的是请记住当设计软件系统（比如汽车）时，对象是由其他对象组合而成。事实上，可以引入到类的树状结构的节点数目和分支是无限的。

图 7-8 展示了汽车的对象模型，其由引擎、音响系统和门组成。

注意组成汽车的这三个对象本身又由其他对象组成。引擎包含活塞和火花塞。音响系统包括收音机和 CD 播放器。门包括一个门把手。请注意还有另一个层级。收音机包括一个旋钮。事实上门把手包括一个锁。CD 播放器包含一个调谐器。另外，我们也可以在旋钮上创建一个拨号盘对象。由设计者控制层级以及对象模型复杂度。

模型复杂度

正如对会叫的狗和不会叫的狗使用继承导致的问题一样，使用过多的组合也会导致更高的复杂度。创建一个包含足够的粒度从而具有充分的表现力的对象模型和粒度过细导致难以理解和维护的模型只在一线之间。

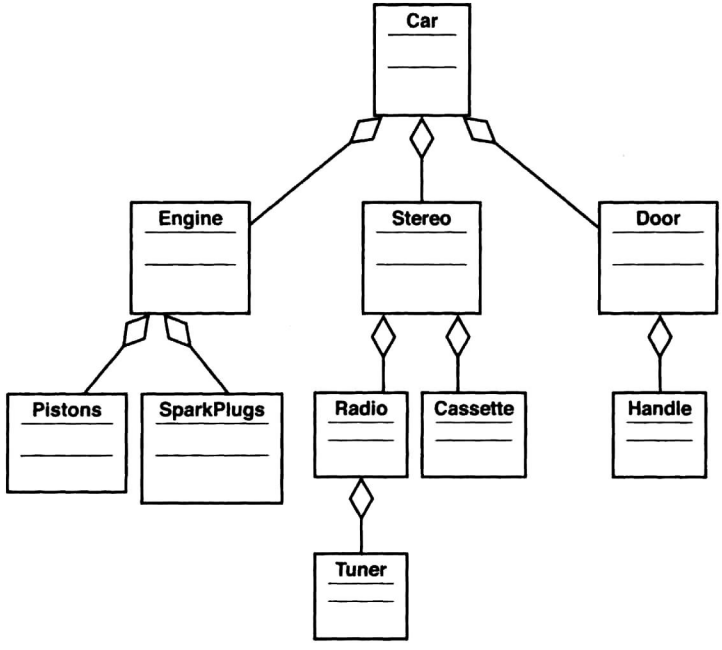


图 7-8　Car 类体系图

7.4　为什么封装是面向对象的本质

封装是面向对象的本质概念。在谈接口 / 实现范式时其实都在谈封装。根本问题是对于一个类而言，它应该暴露什么信息，不暴露什么信息。封装等同于数据和行为。当讨论类

时，主要的设计决策是解决把数据和行为封装到出色组织的类中。

Stephen Gilbert 和 Bill McCarty 将封装定义为："打包程序的过程，将其中的类分为两个不同的部分：接口和实现。"在这本书中我们不断在传达这样的信息。

但是封装和继承是什么关系，以及和本章又有什么关系？这看起来是面向对象的一个悖论。封装是面向对象的本质，所以它是面向对象设计的基本原则之一。继承也是三个主要的面向对象概念之一。然而，继承在某种方式上实际上破坏了封装！怎么会这样呢？是不是三个面向对象中的主要概念中其中两个可能是不兼容的呢？让我们探讨这种可能性。

7.4.1 继承如何减弱封装

之前已经提过，封装就是把类打包为公共接口和私有实现的过程。本质来说，一个类应该隐藏其他类无需了解的任何事情。

Peter Coad 和 Mark Mayfield 在使用继承时举了一个例子，在类的体系结构中继承弱化了封装。他们讨论了一个具体的风险：继承意味着对其他类的强封装，但是弱化了父类和其子类之间的封装。

如果你从父类中继承一个实现，然后父类修改了这个实现，父类的修改会波及整个类的体系结构。这种波及潜在影响所有的子类。貌似这看起来不是一个大问题。但是，正如我们看到的一样，这种波及影响会导致难以预料的问题。例如，测试会变成噩梦。第 6 章中我们讨论了使用封装会更加容易测试系统。理论上来说，如果创建了一个名为 Cabbie 的类（见图 7-9），使用了相应的公共接口，那么任何对于 Cabbie 类的实现修改对于其他类来说都是透明的。然而，任何设计中，父类的修改对子类来说都是不透明的。这是一个难题。

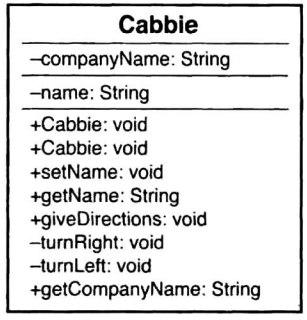


图 7-9 Cabbie 类的 UML 图

如果其他类直接依赖于 Cabbie 类的实现，测试则会变得更加困难。

持续测试

即使使用封装，你依然想重测使用了 Cabbie 类的类来验证本次修改没有引入任何问题。

如果创建了一个 Cabbie 类的子类，叫作 PartTimeCabbie，PartTimeCabbie 继承了 Cabbie 的实现，那么修改 Cabbie 的实现会直接影响到 PartTimeCabbie 类。

请看图 7-10 中的示例，PartTimeCabbie 是 Cabbie 的子类。因此，PartTimeCabbie 继承了 Cabbie 的公共实现，包括方法 giveDirections()。如果 Cabbie 中的 giveDirections() 发生了改变，它将会直接影响 PartTimeCabbie 类，以及其他可能成为 Cabbie 子类的类。在这种微妙的情形下，对 Cabbie 实现的修改没必要封装到 Cabbie 类中。

为了减少这种左右为难的风险，使用继承时严格考量 is-a 条件是非常重要的。如果子类真的是父类的特殊类型，对父类的修改会影响到子类，这是自然和期望的行为。例如，如果 Circle 类继承自 Shape 类，如果修改 Shape 的实现破坏了 Circle，那么 Circle 并不真的需要继承自 Shape。

继承是如何被错误地使用的？比如以下情形，你想创建一个窗口来提供图形化的用户接口（GUI）。你会冲动地创建一个窗口类，其是 Rectangle 类的子类：

```
public class Rectangle {

}

public class Window extends Rectangle {

}
```

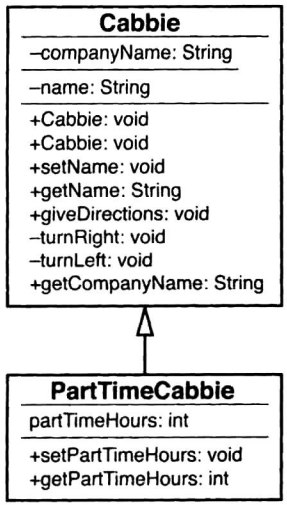


图 7-10　Cabbie/PartTimeCabbie 类的 UML 图

事实上，GUI 窗口包含的内容比矩形多得多。它并不是矩形的一个特例（比如正方形）。真正的窗口包含一个矩形（事实上包含很多矩形）。然而，它不是一个真正的矩形。本例中 Window 类不应该继承自 Rectangle，而应该包含 Rectangle 类：

```
public class Window {

    Rectangle menubar;
    Rectangle statusbar;
    Rectangle mainview;

}
```

7.4.2　关于多态的一个具体例子

很多人认为多态是面向对象设计的基石。面向对象设计类目的就是创建完全独立的对象。在设计良好的系统中，一个对象需要了解所有与其实现有关的问题。对象必须对自身负责，这是一条规则。独立是代码重用的重要机制之一。

正如第 1 章所述，多态从字面上理解就是很多形状。当对象收到消息后，该对象必须定义一个方法来响应该消息。在继承体系结构中，所有的子类会继承其父类的接口。然而，由于每个子类是单独的实体，每个子类需要对同一个消息有不同的响应。

回顾第 1 章中的示例，有个类叫作 Shape。该类有个行为叫作 Draw。然而，如果你告诉别人绘制一个形状，第一个问题会是“什么形状？”简单地让一个人绘制一个形状是非常抽象的（事实上，Shape 类中的 Draw 方法并不包含实现）。你必须指定你想要哪种形状。因此你要为 Circle 类及其他子类提供实际的实现。尽管 Shape 类有个 Draw 方法，但

Circle 重载了该方法并提供了自己的 Draw 方法。重载意味着将父类中的实现替换为自己的实现。

7.4.3 对象职责

我们来重新回顾第 1 章中的 Shape 例子，见图 7-11。

多态是对继承的最优雅的使用之一。请记住 Shape 类是不能被实例化的。由于它有个抽象方法 getArea()，所以它是一个抽象类。第 8 章解释了抽象类的细节。

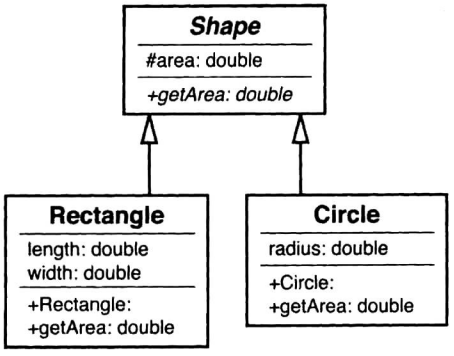


图 7-11 Shape 类继承体系图

不过 Rectangle 和 Circle 可以被实例化，因为它们是实实在在的类。尽管矩形和圆都是形状，但它们有不同之处。它们作为形状，可以计算各自的面积。而各自计算面积的公式是不同的。因此，计算面积的公式不能放置在 Shape 类中。

这就是多态的由来。多态的前提是你可以给不同的对象发送消息，而对象则会根据各自的类型进行响应。例如，如果你向 Circle 类发送 getArea() 消息，调用的计算方式与向 Rectangle 类发送相同的 getArea() 消息的计算方式是不同的。这是因为 Circle 和 Rectangle 拥有各自的响应方式。如果你让 Circle 返回其面积，它知道怎么做。如果你让圆绘制自身，它也知道怎么办。即使 Shape 对象可以被实例化，它也做不到这些，因为它缺乏对自身足够的信息。注意图 7-11 中的 UML 图，Shape 类中的 getArea() 方法是斜体的。这表示了该方法是抽象方法。

假设如下一个简单的示例，我们有 4 个类：抽象类 Shape，实体类 Circle、Rectangle 和 Star。以下是代码：

```
public abstract class Shape{

    public abstract void draw();

}

public class Circle extends Shape{

    public void draw() {

        System.out.println("I am drawing a Circle");

    }
}
```

```
public class Rectangle extends Shape{

    public void draw() {

        System.out.println("I am drawing a Rectangle");

    }
}

public class Star extends Shape{

    public void draw() {

        System.out.println("I am drawing a Star");

    }
}
```

注意每个类中只存在一个方法：draw()。以下是对象自身职责与多态相关的重要一点：实体类自身负责实现绘制功能。Shape 类并不提供绘制的代码，Circle、Rectangle 和 Star 类需要各自实现绘制代码。以下是实现绘制的一些代码：

```
public class TestShape {

    public static void main(String args[]) {

        Circle circle = new Circle();
        Rectangle rectangle = new Rectangle();
        Star star = new Star();

        circle.draw();
        rectangle.draw();
        star.draw();

   }

}
```

测试程序 TestShape 创建了以下类：Circle、Rectangle 和 Star。TestShape 调用各自类的方法来绘制自身：

```
circle.draw();
rectangle.draw();
star.draw();
```

当执行 TestShape 时，你会得到以下结果：

```
C:\>java TestShape
I am drawing a Circle
I am drawing a Rectangle
I am drawing a Star
```

这就是多态的工作原理。如果你想创建一个新形状（比如三角形）应该如何做呢？只需按照如下所示编写这个类，编译并测试和使用。无需修改基类 Shape，也无需编写其他代码：

```
public class Triangle extends Shape{

    public void draw() {

        System.out.println("I am drawing a Triangle");

    }
}
```

可以给 Triangle 发消息了。虽然 Shape 类不知道如何绘制三角形，但 Triangle 类知道：

```
public class TestShape {

    public static void main(String args[]) {

        Circle circle = new Circle();
        Rectangle rectangle = new Rectangle();
        Star star = new Star();
        Triangle triangle = new Triangle ();

        circle.draw();
        rectangle.draw();
        star.draw();
        triangle.draw();

   }

}
```

```
C:\>java TestShape
I am drawing a Circle
I am drawing a Rectangle
I am drawing a Star
I am drawing a Triangle
```

为了见证多态的真实力量，你可以将形状传递给一个方法，该方法绝对不知道形状对象是从哪里来的。请看以下代码，这里包含了具体的形状作为参数。

```
public class TestShape {

   public static void main(String args[]) {

     Circle circle = new Circle();
```

```
    Rectangle rectangle = new Rectangle();
    Star star = new Star();

    drawMe(circle);
    drawMe(rectangle);
    drawMe(star);

  }

  static void drawMe(Shape s) {
    s.draw();
  }

}
```

在本例中，Shape 对象可以传递给 drawMe() 方法。drawMe() 方法可以处理任何有效的形状，甚至是后来实现的形状。你可以用之前的方式运行当前版本的 TestShape。

7.4.4　抽象类、虚方法和协议

Java 以及 .NET 和 C++ 都可以直接实现抽象类。C# .NET 的代码与 Java 代码非常相似，如下所示：

```
public abstract class Shape
{

    is a construct similar to a Java-type interface, called a protocol (discussed
further public abstract void draw();

}
```

Visual Basic .NET 代码如下所示：

```
Public MustInherit Class Shape

    Public MustOverride Function draw()

End Class
```

在 C++ 中使用虚方法也可以实现相同的功能，如下所示：

```
class Shape
{
    public:
        virtual void draw() = 0;
}
```

正如之前章节中提及的一样，Objective-C 不完全支持抽象类功能。

例如，以下 Shape 类的 Java 接口代码我们已经见过多次：

```
public abstract class Shape{

    public abstract void draw();

}
```

对应的 Objective-C 协议如下所示。注意在 Java 代码及 Objective-C 代码中，draw() 方法都没有对应的实现：

```
@protocol Shape

@required
- (void) draw;

@end // Shape
```

从此可以看出，抽象类和协议的功能是等同的。然而，Java 类型的接口与 Objective-C 协议存在分歧。请看以下 Java 代码：

```
public abstract class Shape{

    public abstract void draw();
    public void print() {
        System.out.println("I am printing");
    };

}
```

在上面的 Java 代码中，print() 方法可以被子类继承，C# .NET、VB.NET 以及 C++ 中情况也类似。但在 Objective-C 协议中，如下所示：

```
@protocol Shape

@required
- (void) draw;
- (void) print;

@end // Shape
```

在协议中，提供了 print() 方法签名，因此必须由子类实现，然而这里没有包含任何代码。简单来说，子类不能从协议继承任何代码。因此协议用法与抽象类不是完全相同的，所以设计对象模型时要考虑这一点。

7.5 结语

本章对继承和多态做了基本的综述，并且阐述了它们的不同之处。很多受尊敬的面向对象设计大师指出应该尽量使用组合，只在必要的时候使用继承。

然而，这又过于简单化了。我认为如果组合能尽量地隐藏真正的问题，组合比继承更适合，而不是尽可能地使用组合。事实上，在大多数情况下更适合使用组合并不意味着继承是邪恶的。要在正确的上下文中使用继承和组合。

在之前的章节中，抽象类以及 Java 接口的概念出现过多次。第 8 章中我们会探索开发契约的概念以及抽象类和 Java 接口如何满足契约。

7.6 引用

- Holzner, Steven. 2010. *Visual Quickstart Guide, Objective-C*. Berkeley, CA: Peachpit Press.
- Booch, Grady, Robert A. Maksimchuk, Michael W. Engel, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. 2007. *Object-Oriented Analysis and Design with Applications*, 3rd edition. Boston, MA: Addison-Wesley.
- Meyers, Scott. 2005. *Effective C++*, 3rd edition. Boston, MA: Addison-Wesley Professional.
- Coad, Peter, and Mark Mayfield. 1997. *Java Design*. Upper Saddle River, NJ: Prentice-Hall.
- Gilbert, Stephen, and Bill McCarty. 1998. *Object-Oriented Design in Java*. Berkeley CA: The Waite Group Press.

7.7 本章中使用的示例代码

以下是 C#.NET 版本的代码。其他语言（比如 VB.NET 和 Objective-C）在出版社网站上有电子版。本章已经展示了这些例子对应的 Java 代码。

```
using System;

namespace TestShape
{
    public class TestShape
    {

        public static void Main()
        {

            Circle circle = new Circle();

        Rectangle rectangle = new Rectangle();

        circle.draw();
        rectangle.draw();

    }

}
```

```
public abstract class Shape
{

    public abstract void draw();

}

public class Circle : Shape
{

    public override void draw()
    {

        Console.WriteLine("I am drawing a Circle");

    }
}

public class Rectangle : Shape
{

    public override void draw()
    {

        Console.WriteLine("I am drawing a Rectangle");

    }
}

public class Star : Shape
{

    public override void draw()
    {

        Console.WriteLine("I am drawing a Star");

    }
```

第 8 章 Chapter 8

框架和重用：使用接口和抽象类进行设计

第 7 章中解释了继承和组合在设计面向对象系统的过程中扮演了重要的角色。本章介绍了 Java 类型的接口、Objective-C 中的协议以及抽象类的概念。

接口、协议和抽象类是代码重用的重要机制，提供了所谓契约这一功能。本章主要讨论代码重用这一主题，讲述了框架、契约、接口、协议以及抽象类的概念。注意在本章中，除非特别说明，当使用术语接口时包含了 Objective-C 中的协议概念。在本章的最后，我们将通过一个实际的案例来展示如何使用这些概念。

8.1 代码：重用还是不重用

当程序员写下第一行代码的时候，就要面临代码重用的问题。很多软件开发范式把代码重用作为开发过程中的重要部分。自从计算机软件诞生以来，重用代码的概念已经有过多次革新。面向对象的范式也是为了重用代码。面向对象的拥护者鼓吹面向对象的主要优势就是一次编写，多次重用。

从某些层面来说的确如此。在所有的设计方式中，设计和实现代码需要着重考虑代码的利用率和重用性。并不只是面向对象设计能够重用代码。使用非面向对象的语言照样可以编写非常健壮的可重用的代码。确切地说，使用 COBOL、C 和传统的 VB 之类的结构化的语言编写的大量的常用应用都具有很高的质量以及重用性。

因此，采用面向对象的范式显而易见不是开发可重用的代码的唯一方式。不过面向对象技术提供了几种机制来引导开发人员重用代码。创建可重用的代码的方式之一是使用框架。本章中，我们关注于使用接口和抽象类来创建框架，鼓励编写可重用的代码。

8.2 什么是框架

与代码重用这一概念紧密相关的另一个概念是标准化，又被称为插和拔。框架可以实现插拔机制和重用准则。有关框架的一个经典例子是桌面应用程序。我们用 Office 家族中的一款产品作为示例。我当前使用的文档编辑器（Microsoft Word，是 Office 2010 组件中的编辑工具）的条状工具栏包含了很多标签选项。这些选项与我正在使用的演示文档软件（Microsoft PowerPoint 2010）和电子表格软件（Microsoft Excel 2010）中的选项非常相似。事实上前两个菜单选项（主面板、插入）在三个程序中都是一样的。不仅菜单选项是类似的，其他选项（新建、打开、保存等）看起来也极为相似。条状区域下面是文档区域，这些文档包括 Word 文档、演示文稿和电子表格。这种统一的框架能让我们更容易使用 Office 家族中的软件。它也能让开发者最大化地重用代码，而且可以重用界面设计。

事实上所有的菜单项外观都很相似，这一点并不意外。实际上，当使用集成开发环境进行开发时，如果使用某个具体的平台（比如 Microsoft Windows），平台会提供一些功能，而无需自己实现。当你在 Windows 环境下创建一个窗口时，你会发现在右上角自动拥有一个最大化窗口的按钮及关闭按钮。一些操作也是标准化的，当双击标题栏时，屏幕会最大化（或者最小化）。当点击右上角的关闭按钮时，应用程序会结束运行。这就是框架的力量。图 8-1 是一个文字处理程序的屏幕截图。注意菜单栏和工具栏以及其他元素都是框架的一部分。

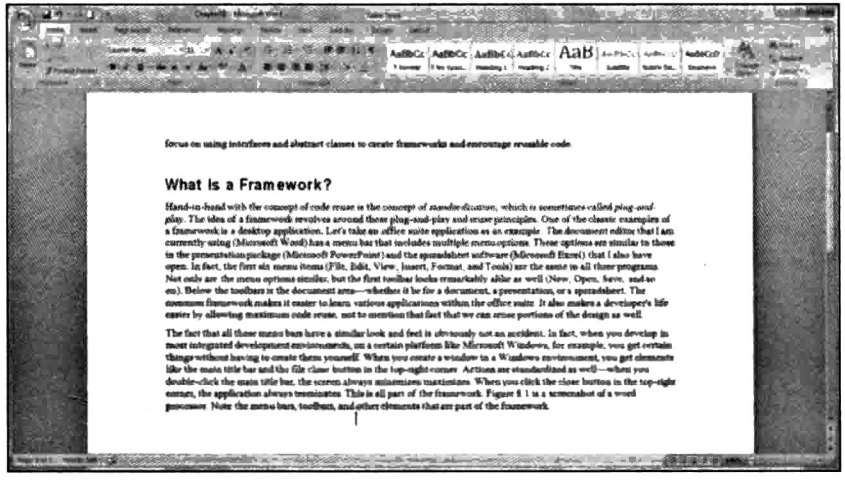


图 8-1 文字处理程序

一个文字处理程序框架通常包括了创建文档、打开文档、保存文档、剪贴文本、复制文本、粘贴文本、搜索文档等操作。开发人员想要使用该框架，必须使用预先定义好的接口来创建应用程序。这些预先定义的接口遵从框架的标准，显而易见非常实用。首先，外观能保持一致，终端用户无需学习新的框架。其次，开发人员可以重用这些编写好的并经过测试的代码（通过测试很重要）。如果能够使用一个充分测试过的窗口，就没必要新建一个。在时间

紧迫的情况下，除非绝对必要，人们并不想学习新的东西。

再谈代码重用

第 7 章我们讨论了继承（即一个类继承另一个类）带来的代码重用方式，本章则是有关框架及重用整个（或部分）系统。

有个问题是，如果需要一个对话框，那么如何使用由框架提供的对话框？答案很简单，只需遵循框架提供的规则即可。那么如何知晓这些规则呢？文档中会记录关于使用框架的规则。编写类或类库的人应该提供文档来介绍如何使用这些类和类库（至少我们希望他这样做）。通常这些文档代表了应用程序编程接口（API）。

例如，在 Java 中创建菜单栏，可以使用 `JMenuBar` 类的 API 文档，查看该文档介绍的公共接口。图 8-2 展示了部分 Java API。使用这些 API 就可以创建一个合法的遵循标准的 Java applet 应用。如果完全遵循标准，这个 applet 应用甚至可以运行在支持 Java 的浏览器中。

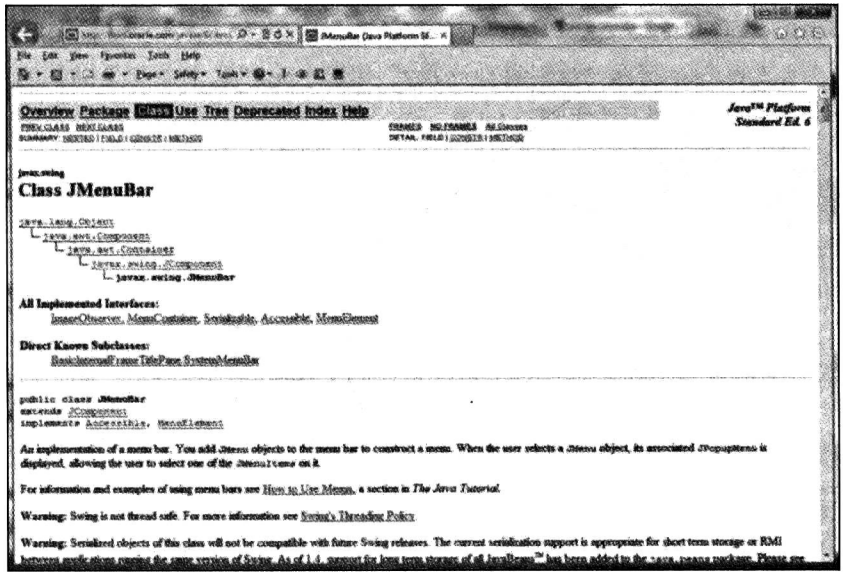


图 8-2　API 文档

8.3　什么是契约

在本章的上下文中，我们定义契约为要求开发人员遵循 API 规格要求的一种机制。通常，API 是指框架。在线字典网站 Dictionary.com（http://www.dictionary.com）定义契约为“两方或多方之间做某事（或不做某事）的约定”，以及“依法必须遵守的约定”。

当开发人员使用 API 时必须强制遵守约定，而项目经理、产品负责人或行业标准提供这些约定。当使用契约时，要求开发人员遵守框架定义的规则，包括方法名、参数数量（比如方法签名）等。总之，这些标准用于倡导良好的开发实践。

术语契约

商业领域广泛使用契约，也包括软件开发领域。不要把这里的契约与其他开发设计概念中的契约混淆了。

强制遵守契约非常重要，因为开发人员总有可能会破坏契约。如果不强制遵守，一些淘气的程序员会决定重新发明轮子来自己实现代码，而不会使用框架提供的规格说明。如果人们通常习惯性漠视或回避标准，那么这个标准就没实际用途。在 Java 和 .NET 语言中，实现契约的两种方式是使用抽象类和接口。

8.3.1 抽象类

实现契约的方式之一是使用抽象类。抽象类包含一个或多个没有提供任何实现的方法。假设有一个名为 Shape 的抽象类。这个类是抽象的，因为你无法实例化它。如果你让某人绘制形状，这个人首先会问你“绘制哪种形状？”因此，形状这个概念是抽象的。然而如果某人让你绘制一个圆，你则不会问这个问题，因为圆是一个实体概念，你知道圆形是什么样子的。你也知道如何绘制诸如矩形等其他形状。

如何应用契约？假设我们想创建一个应用程序来绘制形状。我们的目标是绘制产品设计中包含的所有类型的形状，以后还可以添加新的形状。那么必须遵循两个条件。

首先所有形状必须使用相同的语法来绘制自身。例如，在自定义的系统中每个形状都包含一个名为 draw() 的方法。因此经验丰富的开发人员无需知晓具体的形状，只需调用 draw() 方法就可以绘制形状。从理论上来说可以减少查看 API 文档的时间，而且避免了语法错误。

其次，请记住每个类必须要响应自身的行为。因此一个类被要求提供一个 draw() 的方法，还必须提供该方法的具体实现。例如，Circle 类和 Rectangle 类都有 draw() 方法，而 Circle 类拥有绘制圆形的代码，而 Rectangle 类有绘制矩形的代码。当最终创建 Circle 和 Rectangle 类时，它们都是 Shape 的子类，这些类必须实现各自的 Draw 的版本（见图 8-3）。

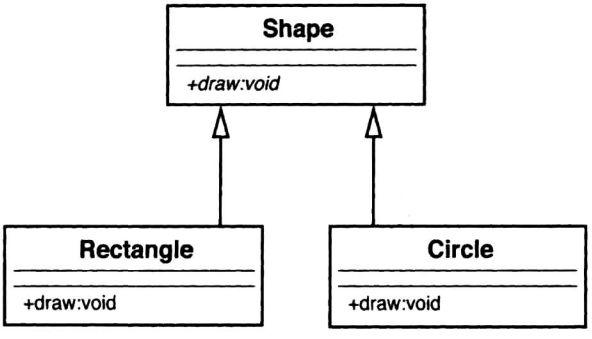


图 8-3 抽象类继承体系图

这样 Shape 框架拥有多态性。系统中的单个对象都可以调用 Draw 方法，

调用不同的形状会返回不同的结果。调用 Circle 对象中的 Draw 方法会绘制一个圆，调用 Rectangle 对象中的 Draw 方法会绘制一个矩形。向对象发送消息，不同的对象会得到不同的响应，这是多态的本质：

```
circle.draw();          // 绘制圆
rectangle.draw();     // 绘制矩形
```

我们通过一些代码来演示 Rectangle 和 Circle 是如何遵循 Shape 契约的。下面是 shape 类的代码：

```
public abstract class Shape {

    public abstract void draw(); // 没有实现

}
```

注意该类并没有提供对 draw() 方法的任何实现。这里几乎没什么代码，所以该方法是抽象方法（该方法包含任何代码，那么它都会立即成为实体方法）。该方法未包含任何代码有两个原因，首先是 Shape 并不知道要绘制什么，即使我们想实现 draw() 方法，也无法实现。

结构类推

这是个有意思的问题。如果我们想让 Shape 类包含所有可能的（当前的以及以后加入的）形状代码，那么需要一些条件语句（比如 Case 语句）。这会非常杂乱，而且难以维护。而使用面向对象设计则可以避免这样的条件语句，展示出了优势。

其次，我们让子类来提供实现。我们来看 Circle 和 Rectangle 类：

```
public class Circle extends Shape {

    public void Draw() {System.out.println ("Draw a Circle")};

}

public class Rectangle extends Shape {

    public void Draw() {System.out.println ("Draw a Rectangle")};

}
```

注意 Circle 和 Rectangle 类都扩展（继承）自 Shape。并且它们都提供了实际实现（本例中实现只是一个示例）。这就是契约的由来。如果 Circle 继承自 Shape 但没有提供 draw() 方法，对 Circle 类的编译会失败。这是因为 Circle 没有满足 Shape 的契约。项目经理会要求在应用程序中创建形状时必须继承自 Shape。这样做可以保证该应用程序中的所有形状都拥有 draw() 方法，并且具有期望的行为。

圆形

如果 Circle 没有实现 draw() 方法，那么可以认为它自身是抽象的。那么另一个子类必须继承自 Circle 并且实现 draw() 方法。该子类则成为 Shape 和 Circle 类的实体实现。

抽象类的概念提供了抽象方法，不过 Shape 类仍可以提供一些实现。(请记住对抽象类的定义是它包含一个或多个抽象方法，这暗示了抽象类也可以提供实体方法。) 例如，尽管 Circle 和 Rectangle 实现了不同的 draw() 方法，但是它们设置形状颜色的机制是相同的。因此 Shape 类可以有一个颜色属性，以及对应的方法来设置颜色。setColor() 方法是一个实体方法，可以被 Circle 和 Rectangle 继承。子类必须要实现的方法就是父类中声明的抽象方法。这些抽象方法就是契约。

小心

请记住本例中的 Shape、Circle、Rectangle 是一个严格的继承关系，而不是接口关系，下章我们会讨论接口。圆是一个形状，矩形是一个形状。这点很关键，因为契约不适用于组合情况（或者 has-a 关系）。

一些语言（比如 C++）只使用抽象类来实现契约。然而 Java 和 .NET 提供了其他机制来实现契约，这种机制叫作接口。而其他语言（比如 Objective-C）本身并不提供抽象类。因此，要在 Objective-C 中实现契约，你需要使用协议，这是 Objective-C 版本的接口。

8.3.2 接口

在定义接口之前，请注意 C++ 没有叫作接口的结构体。当使用 C++ 时，你可以使用抽象类的语法子集创建一个接口。例如，以下的 C++ 代码是一个抽象类。然而由于该类中的唯一一个方法是一个虚方法，所以没有任何实现。结果该抽象类提供了与接口相同的功能。

```
class Shape
{
    public:
        virtual void draw() = 0;
}
```

接口术语

这又一次造成了软件术语的混淆，而且非常混淆。请注意你可以在不同场景下使用术语接口，所以请确保在正确的上下文中使用它。

第一，当涉及用户交互的可视化接口（比如显示器）时，会广泛使用图形化用户接口 (GUI)。

第二，类的接口基本上是指其方法签名。

第三，在 Objective-C 语言中，接口和实现会将代码在物理上分割为不同的模块。

第四，Java 中的接口和 Objective-C 的协议本质上是父类和子类之间的契约。
你还能想出其他关于接口的概念吗？

这有个明显的问题，如果抽象类能够提供与接口一样的功能，那么为什么 Java 和 .NET 还要提供名为接口的结构类型呢？而且为什么 Objective-C 提供协议类型？

另外，C++ 支持多重继承，而 Java、Objective-C 和 .NET 却不支持。尽管 Java、Objective-C 和 .NET 类只能继承自一个父类，但它们可以实现多个接口。使用多个抽象类构成了多重继承。所以 Java 和 .NET 并不支持多重继承。总之，当使用接口时，你无须考虑正式的继承体系，如果设计上是合理的，理论上你可以为任何类添加接口。而抽象类要求你继承自该抽象类，并且延伸至其自身所有可能的父类。

圆

基于这些考虑，接口往往作为缺少多重继承情况下的一种替代方案。不过从技术角度来说并不是这样。接口是单独的设计技术，虽然使用多重继承也可以完成对应用程序的设计，但接口并不是替代或回避使用多重继承。

与抽象类一样，接口可以强有力地强制遵守框架的契约。在介绍任何概念性的定义之前，我们先查看接口的 UML 图以及相应的代码。图 8-4 展示了一个名为 Nameable 的接口。

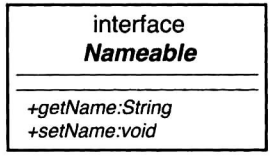


图 8-4　Java 接口的 UML 图

注意 UML 图中的 Nameable 被定义为接口，有别于正常类（抽象类或非抽象类）。还有该接口包含两个方法，getName() 和 setName()。以下是相应的代码：

```
public interface Nameable {

    String getName();
    void setName (String aName);

}
```

以下是对应的 Objective-C 协议的代码用于比较：

```
@protocol Nameable

@required
        - (char *) getName;
        - (void) setName: (char *) n;
@end // Nameable
```

在这些代码中，注意并没有将 Nameable 定义为类，而被定义为接口。所以 getName() 和 setName() 方法是抽象的，并未提供实现。接口不像抽象类，它完全不能提供任何实现。所以任何实现某个接口的类必须提供实现所有方法。例如，在 Java 中，类继承抽象类，而实现接口。

实现继承与定义继承

有时继承被称为实现继承，而接口被称为定义继承。

8.3.3 综合运用

如果抽象类和接口都提供抽象方法，那么两者真正的区别是什么？正如我们之前看到的一样，抽象类可以提供抽象方法，也可以提供实体方法，而接口只能提供抽象方法。为什么要有这样的区别呢？

假设我们想设计一个类代表一只狗，稍后还想添加更多的哺乳动物。那么逻辑上应该创建一个名为 Mammal 的抽象类：

```
public abstract class Mammal {

    public void generateHeat() {System.out.println("Generate heat");}

    public abstract void makeNoise();

}
```

该类拥有一个实体方法叫作 generateHeat()，还有一个抽象方法叫作 makeNoise()。generateHeat() 是个实体方法，因为所有的哺乳动物有体温。而 makeNoise() 方法是抽象的，因为每个动物的叫声是不同的。

我们创建一个名为 Head 的类，用于组合关系中：

```
public class Head {

    String size;

    public String getSize() {

        return size;

    }

    public void setSize(String aSize) { size = aSize;}

}
```

Head 有两个方法 getSize() 和 setSize()。尽管组合不能清晰地阐明抽象类和接口之间的区别，但本例中使用组合来说明在面向对象系统的整体设计中如何一起使用抽象类和接口。这是一个完整的示例。请记住有两种方式来构建对象关系：继承表示 is-a 关系，组合表示 has-a 关系。问题是，接口适合哪种关系？

为了回答该问题，并且引入多种对象关系，我们来创建一个名为 Dog 的类，该类是 Mammal 的子类，实现了 Nameable 接口，并且有一个 Head 对象（见图 8-5）。

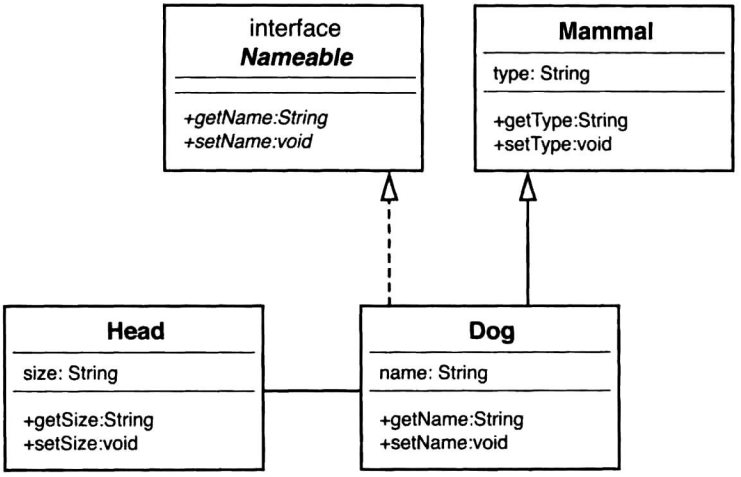


图 8-5　该示例代码的 UML 图

简单来说，Java 和 .NET 中有三种方式构建对象：继承、接口及组合。注意图 8-5 中的虚线代表了接口。该例子演示了何时使用这些方式。什么时候选择抽象类？什么时候选择接口？什么时候选择组合？我们来进一步探索。

你应该熟悉以下概念：

- 狗是哺乳动物，所以 Dog 和 Mammal 之间的关系是继承关系。
- Dog 实现了 Nameable，所以它们之间是接口关系。
- 狗有头，所以 Dog 和 Head 之间是组合关系。

以下代码展示了如何在同一个类中同时使用抽象类和接口：

```
public class Dog extends Mammal implements Nameable {

    String name;

    Head head;

    public void makeNoise(){System.out.println("Bark");}

    public void setName (String aName) {name = aName;}
    public String getName () {return (name);}

}
```

看完 UML 类图后你可能会有一个问题：尽管 Dog 和 Nameable 之间的虚线表示接口，这不也代表的是继承关系吗？乍一看好像是这样，但答案并不简单。尽管接口是继承的特殊类型，但了解特殊之处是非常重要的。理解这特殊之处是设计出强壮的面向对象系统的关键。

虽然继承是严格的 is-a 关系，但接口不是。例如：

- 狗是哺乳动物。

- 爬虫不是哺乳动物。

因此 Reptile(爬虫）类不应该继承 Mammal 类。然而接口可以应用到这些类上。例如：

- 狗有名字。
- 蜥蜴有名字。

在严格的继承关系中，类之间有近似关系。例如，在该设计中，Dog 类与 Mammal 类直接相关。狗是哺乳动物。狗和蜥蜴在哺乳动物层面是不相关的，因为你不能说蜥蜴是哺乳动物。

接口可被应用到不相关的类。你可以给狗命名，也可以给蜥蜴命名。这是使用抽象类和使用接口的关键区别。

抽象类可以提供一些实现。事实上，我们看到 Mammal 提供了一个叫作 generateHeat() 的实体方法。即使我们不知道是哪种哺乳动物，但我们知道所有的哺乳动物都能产生热量。然而接口只模型化行为。接口永远不会提供任何类型的实现，只有行为。接口指定了没有明显联系的类之间的相同行为。不仅狗有名字，而且车、植物等都有名字。

8.3.4 编译器佐证

我们能否证明（或反证）接口具有真实的 is-a 关系吗？在 Java（C# 或 VB）可以使用编译器来证明。请看以下代码：

```
Dog D = new Dog();
Head H = D;
```

当编译器编译这段代码时，会抛出以下错误：

```
Test.java:6: Incompatible type for Identifier. Can't convert Dog to Head. Head H = D;
```

显而易见，狗并不是头。不仅我们知道，而且编译器也同意这一点。然而，以下代码如预期一样工作：

```
Dog D = new Dog();
Mammal M = D;
```

这是一个真正的继承关系，所以不必惊奇这段代码可以通过编译，因为狗就是哺乳动物。

现在我们可以对接口进行测试。接口是不是真正的 is-a 关系呢？编译器是这样认为的：

```
Dog D = new Dog();
Nameable N = D;
```

这段代码工作正常。所以我们可以安全地说狗是有名字的实体。这是简单但有效的证据，继承和接口都构成 is-a 关系。

Nameable 接口

接口指定了某种行为，但不包含实现。通过实现 Nameable 接口，你可以说你通过实现名为 getName() 和 setName() 的方法提供了命名行为。如何实现这些方法决定权在于你。你只需提供这些方法。

8.3.5　创建契约

定义契约的简单规则就是通过抽象类或接口提供一个不含实现的方法。因此，设计子类实现契约时，它必须提供父类（或接口）中对不包含实现的方法的实现。

之前提到过，契约的优势之一是标准化编码的约定。我们可以通过一个示例来展示当没有采用编码标准时的情况。有三个类：Planet、Car 和 Dog。每个类都实现方法来命名该实体。然而，由于它们各自的实现是独立的，每个类有不同的语法来获取名称。Planet 类的代码如下：

```
public class Planet {

    String planetName;

    public void getplanetName() {return planetName;};

}
```

同样，Car 类的代码如下：

```
public class Car {

    String carName;

    public String getCarName() { return carName;};

}
```

Dog 类的代码如下：

```
public class Dog {

    String dogName;

    public String getDogName() { return dogName;};

}
```

显而易见，任何使用这些类的人都需要翻看文档来查明如何获取每个对象的名字（这太糟糕了）。尽管查看文档不算是世界上最糟糕的命运，但如果一个项目（或公司）中的所有类使用相同的命名约定会让生活变得更加容易。这就是为什么会有 Nameable 接口。

所以可以为所有需要使用名字的类制定一个契约。用户在使用各种各样的类时无需查明命名对象的语法。Planet 类、Car 类和 Dog 类拥有相同的命名语法。

为了实现该崇高的目标，我们可以创建一个接口（我们可以使用之前用过的 Nameable 接口）。约定是所有类必须实现 Nameable 接口。因此如果所有类都遵循该命名约定，那么用户只需记住一个接口即可。

```
public interface Nameable {

    public String getName();
    public void setName(String aName);

}
```

新的 Planet、Car 和 Dog 类如下所示：

```
public class Planet implements Nameable {

    String planetName;

    public String getName() {return planetName;}
    public void setName(String myName) { planetName = myName;}

}

public class Car implements Nameable {

    String carName;

    public String getName() {return carName;}
    public void setName(String myName) { carName = myName;}

}

public class Dog implements Nameable {

    String dogName;

    public String getName() {return dogName;}
    public void setName(String myName) { dogName = myName;}

}
```

我们创建了一个标准的接口，并且使用契约保证使用该接口。事实上，使用现代 IDE 的主要好处之一是当实现接口时，IDE 会自动生成需要的方法签名。当使用接口时该功能可以节省大量时间和精力。

这里有个小问题需要考虑。如果人人都遵守规则，那么契约是个好主意，但如果一些不靠谱的个体（比如淘气的程序员）不想遵守规则呢？如果没任何方式阻止人们破坏标准的契约，那么某些情况下会让人们陷入大麻烦。

项目经理可以强调人人必须使用契约，比如团队成员必须使用相同的变量命名约定以及配置管理系统。如果有人没有遵守规则，他会被惩罚，甚至被解雇。

强制规则是保证遵守契约的方式之一，有实例说明如果破坏契约会导致不稳定的代码。比如 Java 中的接口 Runnable。Java applet 实现了 Runnable 接口，它要求实现 Runnable 的类必须实现一个 run() 方法。这很重要，因为浏览器调用 applet 时，会调用 Runnable

的 run() 方法。如果不存在 run() 方法则会出问题。

8.3.6　系统插接点

基本上契约是通过插接点机制引入到代码中的。你想让系统中的任何一部分抽象化，可以使用契约。不用把对象和具体的类建立关联，你可以实现契约来关联所有对象。你需要知道如何在恰当的地方使用契约，不过你可能会滥用契约。你可以识别通用的功能（比如本章中的 Nameable 接口），然而使用契约时要注意权衡。它们可以带来真正的代码重用，但有时也会让事情更加复杂。

8.4　一个电子商务示例

如果决策人没有任何开发背景，那么有时很难说服他重用代码会节省开销。然而当重用代码时，很容易理解这种优势。本小节我们在一个简单的实践示例中，使用继承、抽象类、接口和组合创建一个可以工作的框架。

8.4.1　一个电子商务问题

大概领略重用的力量最好的方式就是通过一个例子来练习如何重用代码。本例中，我们将使用继承（通过接口和抽象类）和组合。我们的目标是创建一个框架，它会真正实现代码重用，减少编码时间以及维护时间。这是典型的软件开发的普遍期望。

让我们开始构建自己的互联网商业。假设我们拥有一个客户，名为 Papa's Pizza 的小比萨店。尽管它非常小，属于家庭生意，但 Papa 认为互联网会带来更多的生意。Papa 想让他的客户访问他的网站，全面了解 Papa 的比萨，而且可以通过自己的浏览器来预定比萨。

在我们开发的网站上，客户可以访问该网站，选择他们想要的商品，并且选择快递的方式和时间。他们可以在餐馆吃东西，预订食物，并且快递过来。例如，客户在 3 点钟决定预订一个比萨作为晚饭（还有沙拉、面包和饮料），并且在 6 点钟送到他家。假设客户在工作中（当然这时在休息），他访问网站并选择比萨，包括尺寸、装饰和外包装、包含调味品的沙拉、面包棒和饮料。然后选择运送选项，要求食物在 6 点钟送到他家。然后他用信用卡付账，得到了确认号码，并且退出。几分钟后，他收到了一份确认邮件。他可以设置账号，所以当打开网站可以记住是哪个用户登录了，该用户最喜欢什么比萨，本周制造了什么新的比萨等。

当该软件系统成功交付后，它取得了巨大的成功。接下来的几周，Papa 的客户很乐意通过网络预订比萨以及其他的食物和饮料。在推广期，Papa 的表兄弟访问了 Papa，这个表兄弟拥有一个名为 Dad's Donuts 的甜甜圈商店。Papa 给 Dad 展示了这个系统，Dad 非常喜欢该系统。第二天，Dad 给我们公司打电话想让我们为他的甜甜圈商店开发一个基于 Web 的系统。这非常棒，正是我们希望的。现在我们如何在甜甜圈店系统中重用为比萨店写的代码呢？

如何让很多更小的商业（包括 Papa's Pizza 和 Dad's Donuts）可以借助框架的优势展示在互联网上？如果我们能够开发一个良好的稳固的框架，我们就能够比以前更加高效地交付基于 Web 的系统，而且开销更小。另外还有附加好处，这些代码之前都已经被测试和实现了，所以再次使用可以减少调试和维护的时间。

8.4.2 非重用方式

基于很多原因，代码重用的概念并不一定符合一些软件开发人员的期望。首先，很多情况下开发系统时甚至并未考虑重用。其次，即使考虑到了重用，计划限制、有限的资源以及经费考虑等问题经常干扰最佳实践。

很多系统中，代码与具体的应用程序高度耦合。这意味着该系统中的代码高度依赖同一个系统中的其他代码。

一些代码重用就是使用剪贴、复制和粘贴操作。在文本编辑器中打开一个程序，复制代码并粘贴到另一个程序中。有时无需任何修改就可以使用某些函数或通用操作。不幸的是，虽然大部分代码无需调整，但仍需要修改小部分代码以适应具体的应用程序。

例如，图 8-6 的 UML 图展示了两个单独的应用程序。

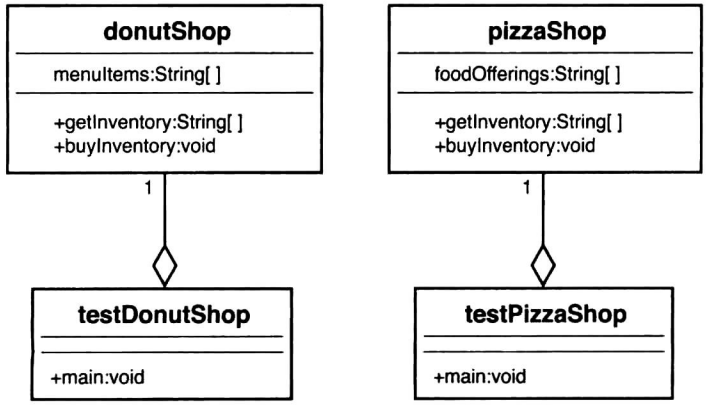


图 8-6 不同的应用程序

本例中应用程序 `testDonutShop` 和 `testPizzaShop` 是完全独立的代码模块。这些代码可以保持独立，模块之间没有任何交互。然而这些应用程序可以使用一些通用的代码。事实上可能从一个程序复制整个代码到另一个程序中。该项目中的某人在某个时刻可能决定为这些在不同应用程序间共享的代码创建一个库。在很多运转良好且遵守纪律的项目中，这种方式工作良好。编码标准、配置管理、修改管理等都会按照标准进行。有些项目则不会严格遵守标准。

任何熟悉软件开发过程的人都知道产生缺陷且时间紧急的情况下，会做一些临时的修复工作。这可能只修复了出问题的应用程序中的问题，但可能在其他应用程序中留下未发现的

潜在问题。因此这样的情况下，刚开始共享的代码会演进，所以需要维护单独的代码库。

例如，有一天 Papa 的网站崩溃了。他心急火燎地通知我们，其中一个开发人员会追踪该问题。该开发人员修复了该问题，知道这个补丁起效了，但不知道为什么有效。该开发人员也不知道该补丁是否会影响该系统的其他地方。所以该开发人员复制了一份代码，只限于用在 Papa 的比萨店系统中。这被亲切地命名为 2.01 papa 版。由于开发人员并未完全理解该问题，而且 Dad 的系统运转良好，补丁代码并未被迁移到甜甜圈商店系统中。

> **追踪缺陷**
>
> 事实上，出现在比萨系统中的缺陷不一定会出现在甜甜圈系统中。尽管该缺陷导致比萨店崩溃了，但甜甜圈店则永远不会遭受这种情况。给比萨店打的补丁代码比起原始缺陷来说可能对甜甜圈店更具危害。

第二周，Dad 惊慌地打了电话，他遇到了一个完全不相关的问题。开发人员修复了它，并且还不知道该补丁是否会影响系统的其他部分，他复制了这份独立的代码，命名为 2.03 dad 版。这种场景发生在我们运营的所有网站中。一切都变得混乱了。我们有很多代码路径，并且无法回溯。我们永远无法合并它们。(可能我们可以，但从商业的角度来说，这开销巨大。)

我们的目标是防止出现前一个例子中的混乱情况。尽管很多系统必须处理遗留问题，而幸运的是比萨程序和甜甜圈程序是全新的。因此，我们可以考虑长远一点，设计系统时尽量考虑重用。通过重用我们不会陷入之前描述的维护噩梦。我们需要做的是尽可能地分解通用部分。在设计中，我们将关注基于 Web 的系统中所有通用的业务功能。我们不会创建多个应用程序类（比如 `testPizzaShop` 和 `testDonutShop`），而是设计一个叫作 `Shop` 的类来供所有的应用程序使用。

注意，`testPizzaShop` 和 `testDonutShop` 具有相似的接口 `getInventory()` 和 `buyInventory()`。我们将提取通用部分并要求所有应用程序遵守 `Shop` 框架来实现 `getInventory()` 和 `buyInventory()` 方法。这种遵循某种标准的要求被称为契约。通过显式的设置服务契约，隔离了代码与实现。在 Java 中可以使用接口或抽象类来实现契约。接下来看如何实现该点。

8.4.3　电子商务解决方案

现在展示如何使用契约来提取该系统的通用部分。我们会创建一个抽象类来抽取实现，使用接口（我们已经非常熟悉的 `Nameable`）来抽取行为。

我们的目标是提供一个定制化版本的 Web 应用程序，其提供以下功能：

- 一个名为 `Nameable` 的接口，是契约的一部分。
- 一个名为 `Shop` 的抽象类，也是契约的一部分。
- 一个名为 `CustList` 的类，在组合中需要使用它。
- 一个 `Shop` 类的新实现，提供给我们服务的每个客户。

8.4.4 UML 对象模型

最新创建的 Shop 类是提取出来的功能。注意图 8-7 中的 getInventory() 和 buyInventory() 方法在体系树中从 DonutShop 和 PizzaShop 中移动到了抽象类 Shop 中。现在无论何时想提供一个新的定制化的 Shop 版本，我们只需加入 Shop 类的新实现（比如杂货店）。Shop 类是实现时必须遵守的契约：

```
public abstract class Shop  {

    CustList customerList;

    public void CalculateSaleTax() {

        System.out.println("Calculate Sales Tax");

    }

    public abstract String[] getInventory();

    public abstract void buyInventory(String item);

}
```

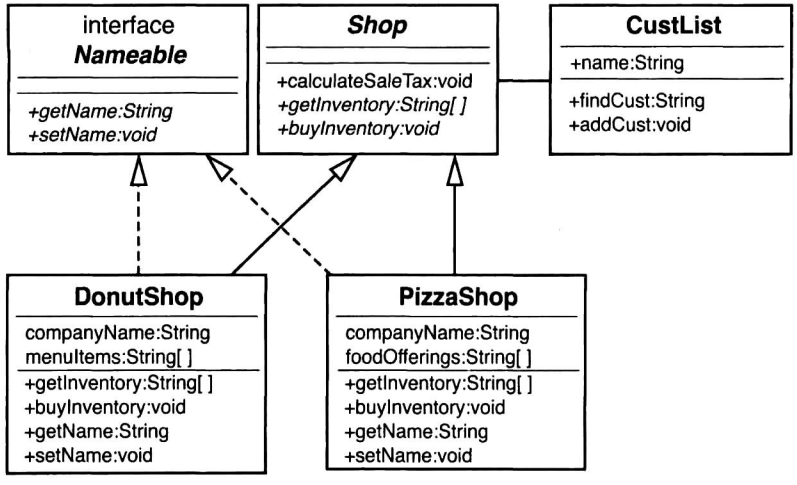


图 8-7　Shop 模型的 UML 图

Shop 类有一个客户列表来展示如何使用组合。因此类 CustList 被包含在 Shop 类中。

```
public class CustList {

    String name;

    public  String findCust() {return name;}
```

```
    public  void addCust(String Name){}

}
```

我们定义了一个名为 Nameable 的接口来演示接口的使用：

```
public interface Nameable {

    public abstract String getName();
    public abstract void setName(String name);

}
```

我们潜在拥有大量不同的实现。但（该程序中）剩余的代码是相同的。在这个小例子中，似乎没节省多少代码。但在大型的现实的应用程序中，节省的代码是巨大的。我们来看看甜甜圈店的实现：

```
public class DonutShop extends Shop implements Nameable {

    String companyName;

    String[] menuItems = {
        "Donuts",
        "Muffins",
        "Danish",
        "Coffee",
        "Tea"
};

    public String[] getInventory() {

        return menuItems;

    }

    public void buyInventory(String item) {

        System.out.println("\nYou have just purchased " + item);

    }

    public String getName(){

        return companyName;
    }

    public void setName(String name){

        companyName = name;
    }
}
```

比萨店的实现看起来是类似的：

```
public class PizzaShop extends Shop implements Nameable {

    String companyName;


    String[] foodOfferings = {
        "Pizza",
        "Spaghetti",
        "Garden Salad",
        "Antipasto",
        "Calzone"
    }

    public String[] getInventory() {

        return foodOfferings;

    }

    public void buyInventory(String item) {

        System.out.println("\nYou have just purchased " + item);

    }

    public String getName(){

        return companyName;
    }

    public void setName(String name){

        companyName = name;
    }

}
```

现在的程序与初始版本不同。初始版本中有大量的定制化的应用程序存在，而我们只有一个主要的类（Shop）和多个定制化的类（PizzaShop 和 DonutShop）。该应用程序以及定制类之间没有耦合。该应用程序中唯一耦合的地方是契约（Shop）。该契约指定了 Shop 的任何实现都必须提供两个方法，getInventory() 和 buyInventory()。它也必须提供 getName() 和 setName() 的实现，这是因为接口 Nameable 的要求。

尽管该程序解决了高度依赖实现的问题，还有个问题在于决定使用什么实现。根据当前的策略，我们不得不分离应用程序。本质上你不得不为每个 Shop 的实现提供一个应用程序。尽管我们使用了 Shop 契约，我们仍然面临使用契约之前相同的情形：

```
DonutShop myShop= new DonutShop();

PizzaShop myShop = new PizzaShop ();
```

那么如何避免这个问题呢？我们可以动态创建对象。在 Java 中可以使用以下代码：

```
String className = args[0];

Shop  myShop;

myShop = (Shop)Class.forName(className).newInstance();
```

上述代码中设置 className 作为参数传给代码。（还有其他的方式来设置 className，比如使用系统属性。）

我们在 Shop 中应用该方式。（注意对象实例化时除了异常处理就没其他代码了）。

```
class TestShop {

   public static void main (String args[]) {

      Shop shop = null;

      String className = args[0];

      System.out.println("Instantiate the class:" + className + "\n");


      try {

       // new pizzaShop();
         shop = (Shop)Class.forName(className).newInstance();

      } catch (Exception e) {

         e.printStackTrace();
      }

      String[] inventory = shop.getInventory();
      // 列出商品清单

      for (int i=0; i<inventory.length; i++) {
         System.out.println("Argument" + i + " = " + inventory[i]);
      }


      // 购买商品

      shop.buyInventory(Inventory[1]);

   }

}
```

通过这种方式，我们可以为 PizzaShop 和 DonutShop 使用相同的应用程序代码。如果我们添加 GroceryShop 应用程序，只需提供实现以及为主应用程序提供相应的字符串，无需修改应用程序代码。

8.5 结语

当设计类和对象模型时，理解对象之间的相互关系是非常重要的。本章主要讨论了构建对象的主题：继承、接口和组合。通过本章可以学习如何通过契约设计来构建可重用的代码。

第 9 章将继续进行面向对象的旅程，并且探索似乎完全不相干的对象是如何交互的。

8.6 引用

- Holzner, Steven. 2010. *Visual Quickstart Guide, Objective-C*. Berkeley, CA: Peachpit Press.
- Booch, Grady, Robert A. Maksimchuk, Michael W. Engel, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. 2007. *Object-Oriented Analysis and Design with Applications*, 3rd edition. Boston, MA: Addison-Wesley.
- Meyers, Scott. 2005. *Effective C++*, 3rd edition. Boston, MA: Addison-Wesley Professional.
- Coad, Peter, and Mark Mayfield. 1997. *Java Design*. Upper Saddle River, NJ: Prentice-Hall.

8.7 本章中使用的示例代码

以下是 C#.NET 版本的代码。其他语言（比如 VB.NET 和 Objective-C）的代码在出版社网站上有电子版。本章已经展示了这些例子对应的 Java 代码。

C# .NET 版本的 TestShop 示例

```
using System;


namespace TestShop
{
    class TestShop
    {

        public static void Main()
        {

            Shop shop = null;

            Console.WriteLine("Instantiate the PizzaShop class:" + "\n");
```

```
            shop = new PizzaShop();

            string[] inventory = shop.getInventory();


            // 列出商品清单

            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine("Argument" + i + " = " + inventory[i]);
            }


            // 购买商品

            shop.buyInventory(inventory[1]);

        }

    }



    public abstract class Shop  {

        public void CalculateSaleTax() {

            Console.WriteLine("Calculate Sales Tax");

        }

    public abstract string[] getInventory();

    public abstract void buyInventory(string item);

}

public interface Nameable {

      string getName();
      void setName(string name);

}

public class PizzaShop : Shop , Nameable
{

    string _CompanyName;
```

```
string[] foodOfferings = {
    "Pizza",
    "Spaghetti",
    "Garden Salad",
    "Antipasto",
    "Calzone"
};

public override string[] getInventory() {

    return foodOfferings;

}

public override void buyInventory(string item) {

    Console.WriteLine("\nYou have just purchased " + item);

}

public string getName(){

    return _CompanyName;
}

public void setName(string name){

    _CompanyName = name;


    }

}

public class DonutShop : Shop , Nameable {

    string _CompanyName;

    string[] menuItems = {
        "Donuts",
        "Muffins",
        "Danish",
        "Coffee",
        "Tea"
    };

    public override string[]  getInventory() {
```

```
            return menuItems;
        }
        public override void  buyInventory(string item) {
            Console.WriteLine(string.format("\nYou have just purchased {0}.", item);
        }
        public string getName(){
            return _CompanyName;
        }
        public void setName(string name){
            _CompanyName = name;
        }
    }
}
```

Chapter 9 第 9 章

创建对象及面向对象设计

前面两章讲述了继承和组合这两个主题。第 7 章中我们学到了继承和组合是创建对象的两种主要方式。而第 8 章中我们深入学习了继承，以及如何一起使用继承、接口、抽象类和组合。

本章会讨论在整体设计中对象之间是如何相互关联的。你可能认为之前已经讲过该问题，的确如此。继承和组合代表了对象交互的方式。然而继承和组合在构建对象时有显著不同。当使用继承时，（至少从概念上来说）最终结果是子类包括了整个继承体系中的所有行为和属性。当使用组合时，会用一些类构建其他类。

继承表示了两个类之间的关系，即子类会包含父类的属性和方法。我们再来看 Person（人）和 Employee（雇员）类的例子（见图 9-1）。

尽管这两个类是相互独立的类，但它们之间的关系并不简单，它们之间是继承关系。通常来说一个雇员是一个人。Employee 对象不会向 Person 对象发送消息，而且 Employee 对象不需要 Person 对象的服务。这是因为 Employee 对象就是 Person 对象。

然而组合是不同的情形。组合代表了不同对象之间的交互。第 8 章主要讲述了继承的多样性，本章会深入了解组合的多样性，以及对象之间如何使用组合进行交互。

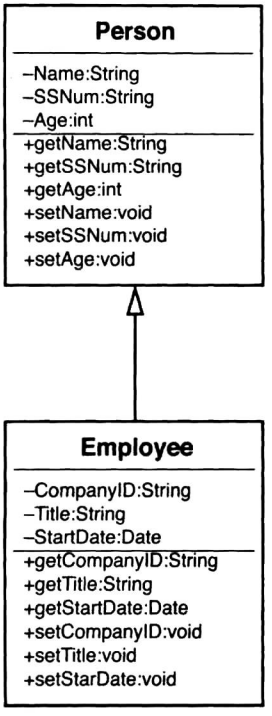


图 9-1　继承体系关系示例

9.1　组合关系

我们已经知道组合代表了整体的一部分。继承关系的术语是 is-a，组合的术语是 has-a。比如，直观来说一个汽车“有”(has-a) 方向盘（见图 9-2）。

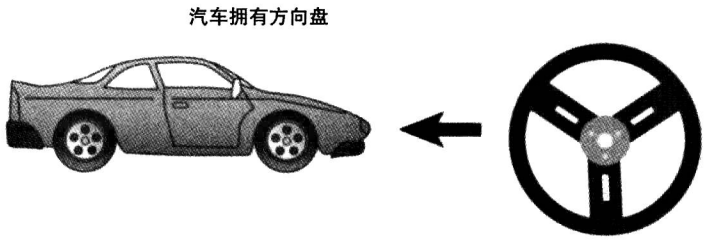


图 9-2　组合关系

> **is-a 和 has-a**
>
> 虽然“has an engine”是正确的语法表达，但是为了一致性，我还是使用“has a engine”。这样做只是为了简单表示规则 is-a 和 has-a。

使用组合的原因是通过组合可以降低构建系统的复杂性。这是人们解决问题的通用方式。研究表明在短期记忆中一次性最多只能记住 7 组数据，所以我们喜欢使用抽象概念。我们不会说拥有一个很大的器件，它包括方向盘、四个车轮、一个引擎等，我们会说拥有一辆车。这有助于相互交流同时保持清晰的头脑。

组合另一方面也保证了组件的可替换性。如果所有的方向盘是一样的，那么无需考虑为具体的车安装特定的方向盘。在软件开发领域，可替换的组件意味着重用。

Stephen Gilbert 和 Bill McCarty 撰写的《Java 面向对象设计》一书中，第 7 章和第 8 章详细列举了联合和组合的很多实例。我强烈推荐阅读该书的相关主题。本章我们关注这些概念的基础部分，并探索一些示例。

9.2　分阶段构建

使用组合的另一个优势是可以分别构建系统及子系统，而且更重要的是这些系统可以被独立测试和维护。

毫无疑问软件系统是相当复杂的。为了构建高质量的软件，必须遵循一种规则来取得成功，这个规则就是尽可能保持简单。为了让大型的软件系统能够正常工作并且易于维护，必须将其分割为更小的更容易管理的单元。如果实现这一点呢？ 1962 年发表的标题为“架构的复杂性”一文中，作者 Herbert simon 作为诺贝尔奖的获得者总结了对稳定的系统的以下思考：

- **稳定的复杂系统通常具有一定的层级结构，每个系统由更简单的子系统构建而成，这些子系统又由更简单的子系统构建而成**。这种方式是软件开发过程中基本的解耦方式，所以你可能已经相当熟悉了。在面向对象的设计中，组合适用于这条准则，即通过简单的对象来构造复杂的对象。
- **稳定的复杂系统是可分解的**。这意味着你可以识别组成系统的各个部分，以及这些部分之间的交互关系。稳定的系统中组成部分之间的交互要少于组成部分内部的交互。比如立体音响系统由更简单的器件组成，即话筒、按钮和扩音器。这种方式比集成系统更稳定，因为集成系统不容易解耦。
- **稳定的复杂系统往往由不同类型的子系统以不同的方式组合而成**。而这些子系统又由更小的部分组合而成。
- **可工作的复杂系统往往是从可工作的简单系统演化而来**。我们往往不会从头建立新系统（即重新发明轮子），而是基于已经经过验证的系统来构建新系统。

在立体音响系统例子（图 9-3）中，假设立体音响系统是一个整体，而不是由组件组成（即立体音响系统是一个巨大的黑盒系统）。在这种情况下，如果 CD 播放器坏掉了而导致系统不稳定应该怎么办呢？你需要把整个系统送去维修。这样做不仅非常复杂和昂贵，而且你无法使用剩余的其他组件。

图 9-3　一步步构建、测试和验证完整系统

在 Java 语言和 .NET 框架中组合概念更加重要。因为对象可以被动态加载，所以解耦设计很重要。例如，如果你发布了一个应用程序，后来由于修复缺陷或者维护的目的，需要重新创建其中一个类文件，那么只需要重新发布这个特定的类文件即可。如果所有的代码都在单个文件中，则需要重新发布整个应用程序。

假设整个系统并不是作为单个单元存在，而是可以分解为多个组件。那么对于立体音响系统来说，如果 CD 播放器坏了，只需要卸下 CD 播放器并送去维修（注意所有的组件通过连接线来关联起来）。这样系统复杂度较低，而且维修也便宜。相比处理整个一体化的系统而言，修复问题花费的时间也显著减少。还有一个附加的好处，那就是你仍然可以使用剩下的组件。你甚至可以买一个新的 CD 播放器，因为它只是整个系统中的组件。而维修工可以将坏了的 CD 播放器插到他的维修系统上来测试和修复它。总之组件方式非常好用。作为软件设计者，组合是用于对抗软件的复杂度的非常重要的策略之一。

使用组件的主要好处就是可以使用其他开发人员（甚至是第三方供应商）构建的组件。不过使用第三方组件要确保充分信任该组件。第三方组件的来源必须可信，并且经过了严格的测试，而且确保其与广告中的宣传表现一致。很多人宁愿自己构建组件，而不会信任别人构建的组件。

9.3　组合类型

通常有两种组合方式：联合和聚合。这些方式代表了对象之间的协作关系。我们拿立体音响系统来解释联合的优势。

> **组合是否是联合的一种形式**
>
> 在面向对象技术领域中，组合是另一个先有鸡还是先有蛋的问题。有些人认为组合是联合的一种方式。在本书中我们认为继承和组合是构建类的两种主要方式。因此本书认为联合是组合的一种方式。

任何组合类型都是 has-a 关系。然而，联合和聚合的微小区别在于部分如何构成整体。在聚合中，通常只看到整体，而在联合中，通常看到的是组成整体的部分。

9.3.1　聚合

最直观的组合方式就是聚合。聚合意味着复杂的对象由其他对象构成。电视机是进行娱乐活动的统一平台。当看电视时，你只看到了一个电视机。大多数时候你不会思考电视机事实上是由一些晶体管、一个屏幕、一个控制器等器件组成。当然你可以看到用于控制电视机的开关，你也确实能看到屏幕。不过这不是通常的人们认识电视机的方式。当进入一个电器商场时，销售人员不会说“请看这个晶体管、屏幕和开关等的聚合体”，他会说“请看这个电视机”。

同样，当你买汽车时，不会选择组成汽车的各个独立的组件。你并不会决定买什么样的火花塞以及什么样的门把手，你会直接买一辆车。当然你可以选择一些配件，但整个汽车对象已经提供了绝大多数的组件。汽车对象又是由很多复杂的或简单的对象构成（见图 9-4）。

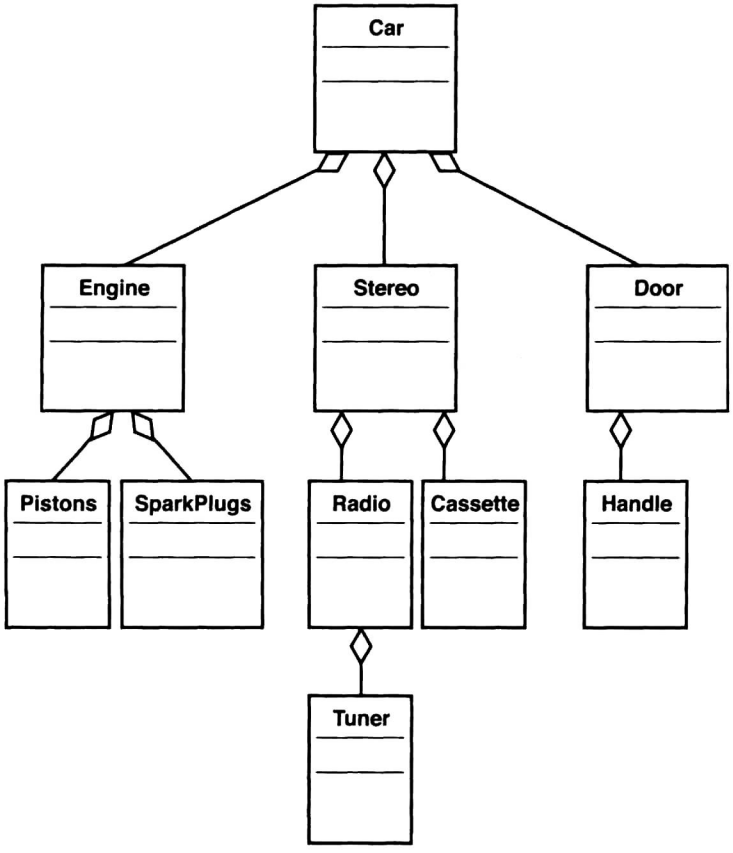


图 9-4　车的聚合体系图

9.3.2　联合

聚合代表你通常只看到了整体，而联合既代表整体，也代表部分。在立体音响系统例子中，各种各样的组件是独立的，通过插接线（连接各种各样组件的线）连接成整体。

我们拿一个计算机系统来举例（见图 9-5）。计算机系统是一个整体，而组成部件为显示器、键盘、鼠标和主机。每个部件都是单独的对象，但它们一起表示了整个计算机系统。主机通过键盘、鼠标以及显示器来分配工作。换句话说，主机需要鼠标的服务，而自己本身无需具备提供这种服务的能力。因此主机通过特定的端口和线将鼠标连接起来，从而可以接受单个的鼠标的服务。

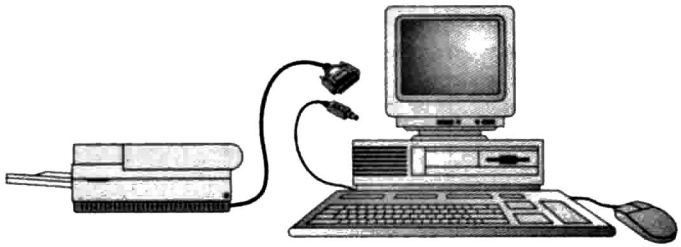


图 9-5　联合作为独立的服务

聚合和联合

聚合是指复杂的对象由其他对象组成。而当一个对象需要其他对象的服务时则使用联合。

9.3.3　同时使用联合和聚合

你可能注意到所有的例子中聚合和联合的区别通常是模糊的。这也说明大多数设计决策都是在考虑是使用聚合还是联合。

例如，计算机系统例子可用来描述联合关系，但也包含了一些聚合关系。尽管计算机主机、显示器和键盘以及鼠标之间的交互是联合关系，但计算机主机本身则代表了聚合。你只看到了计算机主机，但它其实是由其他对象组成的一个复杂系统，它包括了集成电路、主板、显卡等。

一个 Employee 对象可能会由 Address 对象和 Spouse 对象组成。你可能认为 Address 对象是聚合关系（基于它是 Employee 对象的一部分），而 Spouse 对象则是联合关系。为了阐述，假设该雇员和其配偶都是同一个公司的雇员。如果雇员被解雇了，那么其配偶仍然在系统中，但破坏了联合关系。

同样，在立体音响系统中，接收器和扬声器以及 CD 播放器是联合关系。扬声器和 CD 播放器自身则是其他对象（比如电源插头）聚合起来的结果。

在汽车例子中，尽管引擎、火花塞和门代表组合关系，而音响系统则代表了联合关系。

没有绝对答案

事实上，进行设计决策时并不一定有一个绝对正确的答案。设计并不是一门精密的科学。不过我们可以使用一些通用规则，这些规则不难使用，而且能帮助我们快速开始。

9.4　避免依赖

使用组合时要求避免对象之间高度依赖。混合不同领域会导致对象间的高度依赖。最佳实践是一个领域中的对象不应当和另一个领域中的对象混合，除非有非常特殊的情况。我们再次使用立体音响系统例子解释该概念。

将接收器和 CD 播放器划分在不同的领域中，则很容易维护立体音响系统。例如，如果 CD 部件坏了，你可以将 CD 播放器单独送去维修。CD 播放器和 MP3 播放器是单独的领域。这样更具灵活性，比如从不同的制造商那里购买 CD 播放器和 MP3 播放器。因此你可以更换 CD 播放器的品牌。

混合领域在某些方面有一定的便利性。TV/VCR 和 TV/DVD 组合是个好例子。本书第 1 版出版的时期不仅兴起了面向对象的技术，而且也发明了很多面向消费者的技术。VCR 已经不再流行，而 DVD 也会步 VCR 的后尘。消费者短时间内从流媒体技术的 VCR 过渡到 DVD 是个好例子，有力地说明了设计决策的各个层级都要尽量避免依赖。

你需要确定具体场景下什么因素最重要。你是想要便利还是稳定，没有正确的答案，这取决于应用程序和环境。在 TV/VCR 组合例子中，我们认为集成起来的组件比各个独立的组件更便利（见图 9-6）。

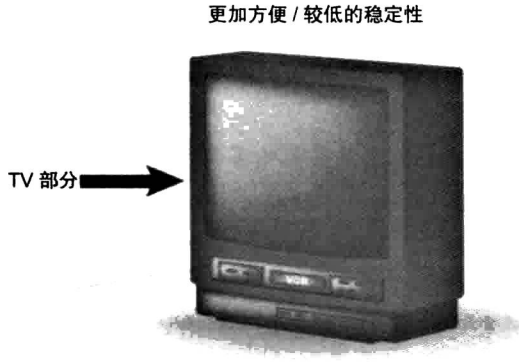


图 9-6 便利和稳定

混合领域

利用混合系统的便利性是一项设计决策。如果 TV/VCR 集成系统的便利性比起单个组件的风险和故障更重要，那么采用混合领域则是首选的设计决策。

9.5 基数

在《Java 面向对象设计》一书中，Gilbert 和 McCarty 将基数表示参与联合的对象个数，可以表示这种联合关系是可选的还是强制的。为了确定基数，Gilbert 和 McCarty 问了以下问题：

- 对象之间是如何协作的？
- 有多少对象参与协作？
- 这种协作是可选的还是强制的？

例如，请看以下例子。我们创建了一个 `Employee` 类，其继承自 `Person` 类，并且和以下类有关系：

- `Division`
- `JobDescription`
- `Spouse`
- `Child`

这些类是用来做什么的？哪些是可选的？哪些是 Employee 类必须使用的？

- Division。
  - 该对象包括了该雇员所在部门的相关信息。
  - 每个雇员都必须有部门，所以它们的关系是强制的。
  - 一个雇员只能隶属于一个部门。
- JobDescription
  - 该对象包括了工作简介，比如薪水等级以及薪水范围等信息。
  - 每个雇员必须有工作简介，所以它们的关系是强制的。
  - 雇员在公司工作期间可以有多个工作。因此一个雇员可以有多个工作简介。如果雇员更改了工作岗位，但之前的工作内容仍需做历史记录，而且一个雇员可以同时拥有两种不同的工作。例如，管理员在某个雇员离职后还未找到替代者之前可以肩负起该雇员的职责。
- Spouse
  - 在这个极其简化的例子中，Spouse 类只包括结婚周年纪念日信息。
  - 雇员可以是已婚也可以是未婚。因此，Spouse 类是可选的。
  - 一个雇员最多只能有一个配偶。
- Child
  - 在这个简单的例子中，Child 类只包含字符串 FavoriteToy。
  - 雇员有可能有孩子，也可能没有孩子。
  - 雇员可能没有孩子，也可能有无数个孩子。你可以决定系统可以处理的孩子数量上限。

表 9-1 总结了这些类的联合基数。

表 9-1　类联合关系的基数

可选 / 联合	基　数	强制性
Employee/Division	1	强制
Employee/JobDescription	1…n	强制
Employee/Spouse	0…1	可选
Employee/Child	0…n	可选

基数标记

0…1 标记意思是雇员可以有 0 个或 1 个配偶。0…*n* 标记意思是雇员可以有从 0 到无限个孩子。*n* 本质上代表了无穷。

图 9-7 展示了该系统的类图。注意在该类图中，基数由连接线决定。请参照表 9-1 来看联合是强制的还是可选的。

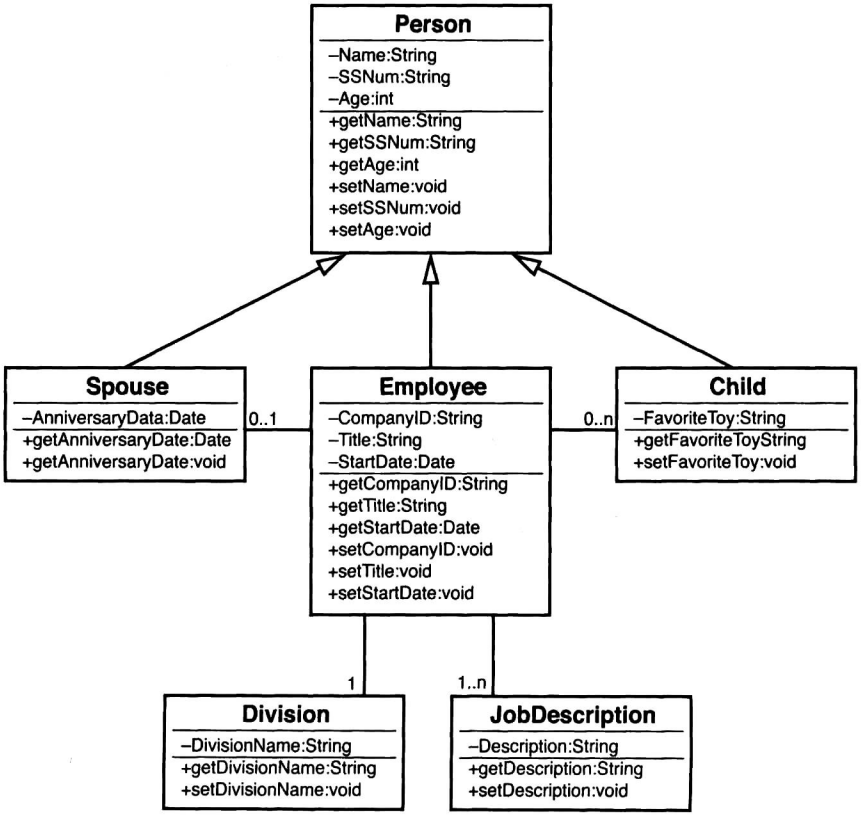


图 9-7 UML 图中的基数

9.5.1 多个对象联合

如何在代码中表示包含多个对象的联合（比如 0 到多个孩子）？这是 Employee 类的代码：

```
import java.util.Date;

public class Employee extends Person{

        private String CompanyID;
        private String Title;
        private Date StartDate;

        private Spouse spouse;
        private Child[] child;
        private Division division;
        private JobDescription[] jobDescriptions;

        public String getCompanyID() {return CompanyID;}
        public String getTitle() {return Title;}
        public Date getStartDate() {return StartDate;}
```

```
        public void setCompanyID(String CompanyID) {}
        public void setTitle(String Title) {}
        public void setStartDate(int StartDate) {}

}
```

注意代码中的类使用数组表示一对多的关系：

```
private Child[] child;
private JobDescription[] jobDescriptions;
```

9.5.2　可选的联合

当处理联合时最重要的问题之一是确保设计应用程序时检查可选的联合。即代码必须检查该联合是否为 null。

假设之前的例子代码中每个雇员有一个配偶。然而，如果雇员没有结婚，代码则会出问题（见图 9-8）。如果代码期望有一个配偶，结果会出问题，导致系统处于不稳定的状态。所以代码必须检查空值情况，并且必须作为有效的情况进行处理。

例如，如果雇员没配偶，代码不应该试图调用配偶的方法，这会导致应用程序崩溃。因此，代码必须能够处理 Employee 对象没有配偶的情况。

图 9-8　检查所有可选的联合关系

9.6　一个综合性示例

我们通过一个简单的例子来把继承、接口、组合、联合和聚合概念糅合到一个简单的系统中。

我们使用第 8 章中的例子，并做了以下扩展。我们添加一个 Owner 类用来遛狗。

Dog 类直接从 Mammal 类继承。图 9-9 中的实体箭头表示了 Dog 类和 Mammal 类之间的关系。Nameable 类是 Dog 实现的接口，通过 Dog 类和 Nameable 接口之间的虚线箭头表示这种关系。

在本章中，我们一直在讨论联合和聚合。Dog 类和 Head 类之间是聚合关系，因为头实际上是狗的一部分。连接这两个类图的线上的基数表示了一只狗只能有一个头。

Dog 类和 Owner 类之间是联合的关系。主人当然不是狗的一部分，反过来也一样，所以肯定不是聚合关系。然而狗需要主人的服务（遛狗）。Dog 类和 Owner 类之间的线基数指定狗可以拥有一个或多个主人（例如，妻子和丈夫都可以认为是狗的主人，都有遛狗的责任）。

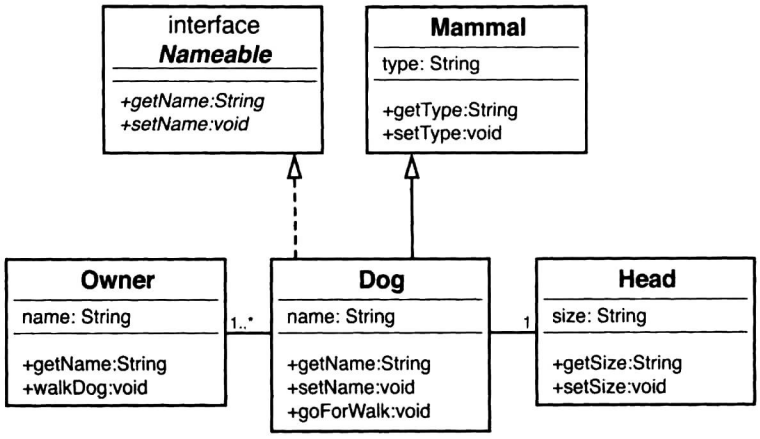


图 9-9 Dog 例子中的 UML 类图

当设计面向对象的系统时，主要就是使用继承、接口、组合、联合和聚合等手段组织对象之间的关系。

> **Head 类在哪里**
>
> 你可能认为将 Head 类附加到 Mammal 类上而不是 Dog 类上更合理，因为所有的哺乳动物一般都只有一个头。在该模型中，我只是想突出 Dog 类，所以才会把 Head 类附加到 Dog 类上。

9.7 结语

本章我们探索了组合关系，以及组合的两种主要形式：聚合和联合。继承表示一种新的已知对象，而组合表示各种各样的对象之间的交互关系。

前面三章涵盖了继承和组合的本质。在软件开发过程中使用这些概念和技能，有助于设计出稳固的类和对象模型。接下来的章节将探索如何借助 UML 类图进行对象建模。

9.8 引用

- Booch, Grady, Robert A. Maksimchu, Michael W. Engel, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. 2007. *Object-Oriented Analysis and Design with Applications*, 3rd edition. Boston, MA: Addison-Wesley.
- Meyers, Scott. 2005. *Effective C++*, 3rd edition. Boston, MA: Addison-Wesley Professional.
- Coad, Peter, and Mark Mayfield. 1997. *Java Design*. Upper Saddle River, NJ: Prentice-Hall.
- Gilbert, Stephen, and Bill McCarty. 1998. *Object-Oriented Design in Java*. Berkeley, CA: The Waite Group Press.

第 10 章 Chapter 10

创建对象模型

我坚持认为在学习具体的建模工具之前，应先学习底层的面向对象概念。因此何时引入本章值得考虑。有很多理由将本章放在前面，因为统一建模语言（UML）图贯穿本书，包括第 1 章。最终它被放置在讲述概念的章节（即第 1 ~ 9 章）之后。剩下的章节都是一些具体的应用程序技术问题和概念。

本章是本书中使用的 UML 类图标记的概览。它并不是 UML 的综合教程，因为这需要一本书来介绍，而且已经有很多这样的书了。本章最后的引用部分列出了一些优秀的资源。由于本书只讲基础，所以只对 UML 的功能做简要介绍。

本书使用 UML 类图对面向对象的系统进行建模，我喜欢称之为*对象建模*。UML 标记语言使用类图来进行系统建模。本书不会涉及很多 UML 的概念，比如状态图和活动图。每个主题都需要一个或多个完整的章节来讲述。而本章的目的只是提供对象模型的快速概览，特别是类图。所以如果你不熟悉类图，学习本章可以让你快速掌握基本知识。通过本章的介绍，你会更容易理解本书中的一些示例。

10.1 什么是 UML

UML，正如其名字所示，是一种建模语言。UML 用户指南将 UML 定义为“一种图形化语言，可以可视化、设计、构造以及文档化软件系统”。UML 提供了一种标准的方式来绘制系统蓝图。简言之，UML 提供了一种方式来图形化表达和操作面向对象的软件系统。它不仅可以展现系统设计，也可以作为工具帮助设计。

UML 是 Grady Booch、James Rumbaugh 和 Ivar Jacobson（被亲切地称为三剑客）分别开

发的不同的建模语言的集合。软件公司 Rational 将这三种建模语言组合在一起，命名为统一建模语言。之前已经说过，对象建模只是 UML 的一部分。

然而不要把 UML 和面向对象开发关联得过于紧密。Craig Larman 在文章《 What the UML Is-and Isn't》中写道：

> 但是，在软件开发工程和 UML 绘图语言领域，读写 UML 标记的能力有时候好像等同于面向对象的分析和设计能力。事实当然并非如此，后者比前者更加重要。因此我推荐先学习面向对象分析和设计的相关教学资料，它优先于学习使用 UML 标记的相关工具。

尽管 UML 非常重要，但先学习面向对象技能更重要。学习面向对象概念之前先学习 UML，就如同不了解任何电路知识之前先学习电路图。

10.2 类图结构

类图由三部分组成：类名、属性和方法（构造函数也是方法）。类图本质上是由三条线分割组成的一个矩形。我们再次拿出出租车司机的例子来阐述类图。图 10-1 展示了表示该类的 UML 类图。

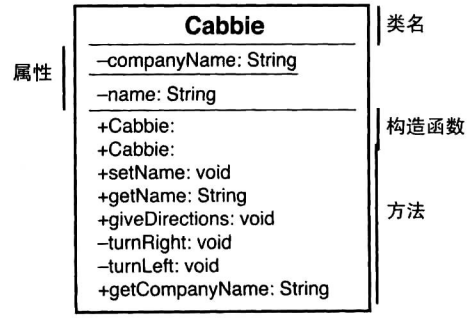


图 10-1 Cabbie 类的 UML 图

这个 UML 图完全对应了以下的 Java 代码：

```
/*

    该类定义了一个出租车司机并分配了一辆出租车


*/
public class Cabbie {

    // 放置公司名称
    private static String companyName = "Blue Cab Company";

    // 出租车司机姓名
    private String name;

    // 分配给出租车司机的出租车

    // 默认构造函数
    public Cabbie() {

        name = null;
        myCab = null;

    }
```

```
    // 初始化构造函数
    public Cabbie(String iName, String serialNumber) {

        Name = iName;
        myCab = new Cab(serialNumber);

    }

    // 设置司机姓名
    public void setName(String iName) {
        name = iName;
    }

    // 获取司机姓名
    public String getName() {
        return name;
    }

    // 设置出租车方向
    public void giveDirections(){
    }

    // 右转
    private void turnRight(){
    }

    // 左转
    private void turnLeft() {
    }

    // 获取公司名称
    public static String getCompanyName() {
        return companyName;
    }

}
```

我们仔细阅读这段代码并和 UML 类图作对比。注意代码中的类名、属性和方法与类图中符号的关联。事实上，类图与代码结构是完全一致的。而且类图提供了更多的信息。下面的小节会详细讨论。

10.3　属性和方法

类图既可以表示类的结构，也可以表示属性和方法相关信息。

10.3.1 属性

通常属性并没有签名，只有方法具有签名。然而属性有类型，类图中可以体现出该类型。比如 Cabbie 例子中有两个属性：

```
-companyName:String
-name:String
```

这两个属性都被定义为字符串类型。紧跟着属性名的字符串（本例中是 String）代表了该属性的类型。属性也可以被定义为整型和浮点型，如下所示：

```
-companyNumber:float
-companyAge:int
```

通过类图可以知道参数的数据类型。因为属性之前有个减号（-）前缀，声明了这些属性为私有属性。加号（+）则表示这些属性是公共的，而这是不应该的。基于之前章节中的讨论，我们知道应该将所有的属性定义为私有的。虽然有些人会使用公共属性，但本书中始终坚持属性应该为私有的。

10.3.2 方法

UML 图中表示方法的方式与表示属性的方式是相同的。UML 图不仅可以表示参数类型，还可以展示方法的返回类型。

以下的代码片段展示了 Cabbie 类中的方法名、返回值类型以及访问修饰符（例如，public、private）：

```
+Cabbie:
+giveDirections:void
+getCompanyName:String
```

以上的三个方法访问修饰符都是公共方法（使用加号表示）。如果方法是私有的，则会使用减号。每个方法名后会跟着一个冒号，用于分割方法名和返回值类型：

以下方式可以包含一组参数列表：

```
+getCompanyName(parameter-list):String
```

逗号用于分隔参数列表中的参数：

```
+getCompanyName(parameter1, parameter2, parameter3):String
```

我喜欢尽量保持对象模型的简单化。因此通常在类图中我只包括类名、属性和方法。这可以让我们专注于整体设计，而不会局限于细节。在类图引入过多的信息（比如参数）会使对象模型难以阅读。这点由你自己来权衡。

10.4 访问符号

之前说过，属性和方法前面的加号（+）和减号（-）表示这些属性和方法是公共的还是

私有的。如果是减号，则认为属性和方法是私有的。那么其他类就无法访问该属性或方法。只有自身类中的方法可以访问和修改私有属性。

如果属性或方法前面有个加号，那么该属性或方法则是公共的，任何类可以访问和修改它。例如以下代码：

```
-companyNumber:float
+companyAge:int
```

在这个例子中，companyNumber 是私有的，只有自身类中的方法可以使用该属性。然而 companyAge 是公共的，因此任何类可以访问和修改它。

如果代码中没有访问符号，那么系统将设置访问权限为默认值，并且不会使用加号或减号：

```
companyNumber:float
companyAge:int
```

在 Java 中访问修饰符的默认类型是受保护类型。受保护的访问意味着只有同一个包中的类可以访问该属性或方法。Java 包是一组相关类的集合，由开发人员有意识地放置在一起：

而 .NET 中微软的 MSDN 文档定义了以下访问修饰符：

- public。在同一个集合或其他引用其的集合中的所有代码都可以访问该类型或成员。
- private。该类型或成员只能被同一个类或结构体中的代码访问。
- protected。该类型或成员只能被同一个类或结构体或衍生类中的代码访问。
- internal。该类型或成员能被同一个集合中的任何代码访问，但其他集合则不能访问。Objective-C 也提供了公共（@public）和私有（@private）访问修饰符。Objective-C 的默认访问是受保护类型，只允许子类中的成员访问。

10.5　继承

第 7 章中的 Dog 例子讲解了继承的工作原理。在该例子中，GoldenRetriever 类继承自 Dog 类，如图 10-2 所示。UML 图中通过带箭头的线指向父类（或超类）来表示这种关系。

这种标记非常简单，当看到一条带有箭头的线时，那么表示了继承关系。

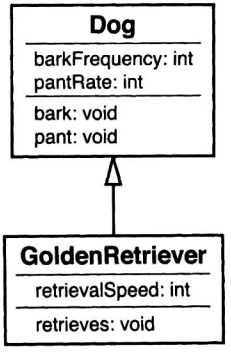


图 10-2　Dog 继承体系的 UML 图

表示接口继承

带箭头的虚线表示接口，下一小节会进行讨论。

由于本书中的代码示例主要使用 Java 语言，所以无需担心多重继承。同样 .NET 和 Objective-C 也无多重继承。而 C++ 实现了多重继承。

不过多个子类可以继承自同一个父类。我们可以使用第 7 章中的 Dog 示例（如图 10-3 所示）。

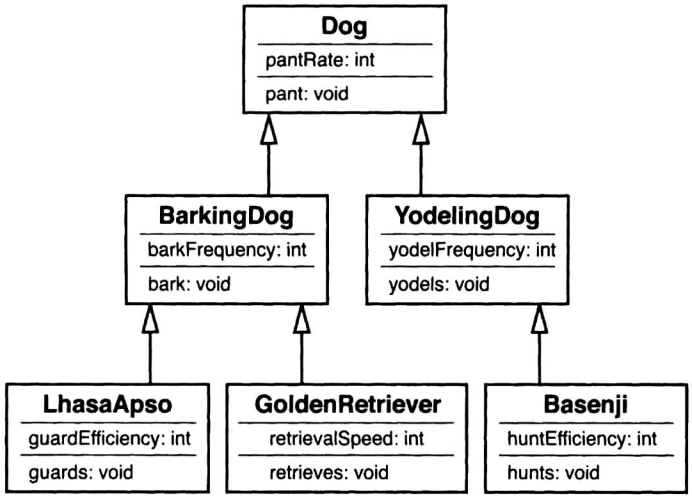


图 10-3　扩展后的 Dog 继承体系的 UML 类图

本例阐明了对继承树进行建模时的两个要点。首先父类可以拥有多个子类。其次，继承树可以扩展为多层。图 10-3 中的例子中有三层。我们可以添加明确的寻回犬（retriever）类型来扩展更多层次，或者创建一个犬科（Canine）类来向上添加一层（如图 10-4 所示）。

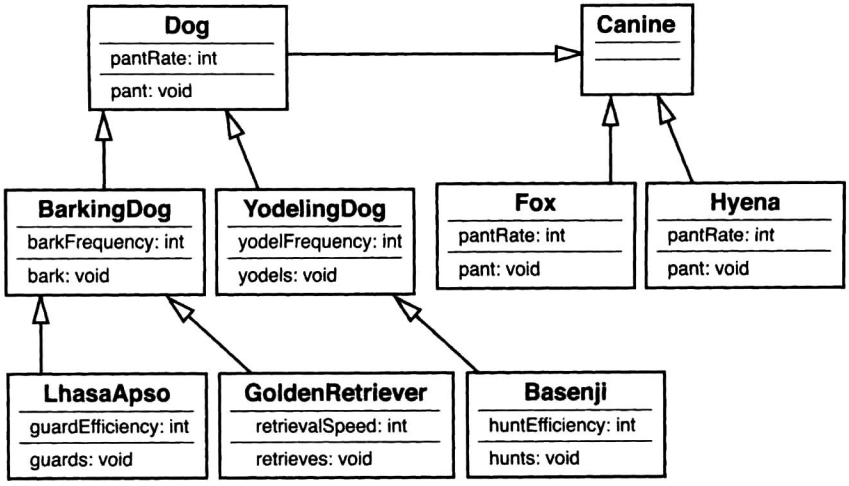


图 10-4　Canine 继承体系的 UML 图

多重继承

如果你使用 Eiffel 和 C++ 之类的语言进行设计，那么这些语言包含了多重继承，在整个设计中可以使用这个特性。

10.6　接口

由于接口是特殊的继承类型，所以两者标记很相似，有时候会引起混淆。之前我们说过带箭头的线表示继承。接口也是由带箭头的线表示，但是箭头连接的是一条虚线。UML 标记将继承和接口区分开来。请看图 10-5，这是第 8 章例子中的一个简化版本。在该图中，Dog 类继承自 Mammal 类并且实现了接口 Nameable。

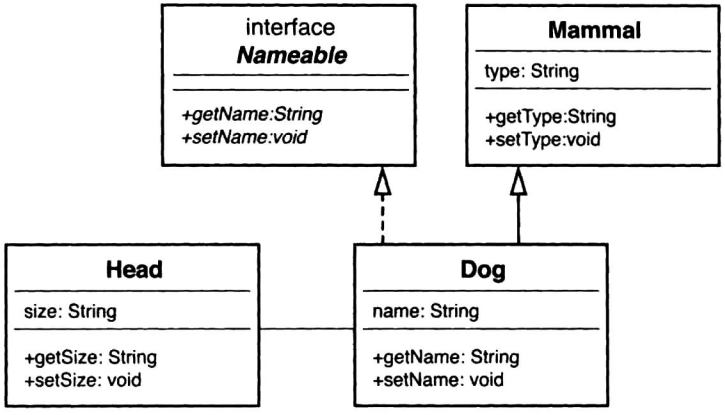


图 10-5　展示了接口关系的 UML 图

10.7　组合

组合表明了 has-a 关系。如果在设计时不适合用继承（因为不适用于 is-a 关系），那么通常可以使用组合。

第 9 章中讨论了两种组合关系：聚合和联合。当需要借助其他类来创建一个类时就是组合关系。当一个类由其他类组成时就是聚合关系（比如轮胎和汽车的关系）。当一个类需要其他类的服务时就是联合的关系（比如客户需要服务器的服务）。

10.7.1　聚合

聚合由一个头部有一个菱形的线表示。第 9 章中的汽车例子中，为了表示方向盘属于车的一部分，请看图 10-6 的标识符。

在继承树中，理论上来说并没有限制聚合层次数量。图 10-7 的例子中有 4 层。注意多种层级可以表示多种聚合。例如尽管立体声系统是汽车的一部分，但收音机是立体声系统的一部分，而调谐器则是收音机的一部分。

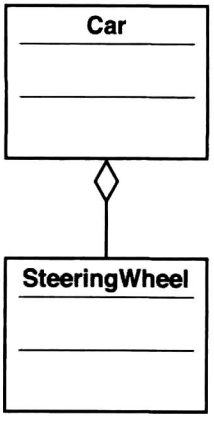


图 10-6　表示组合关系的 UML 图

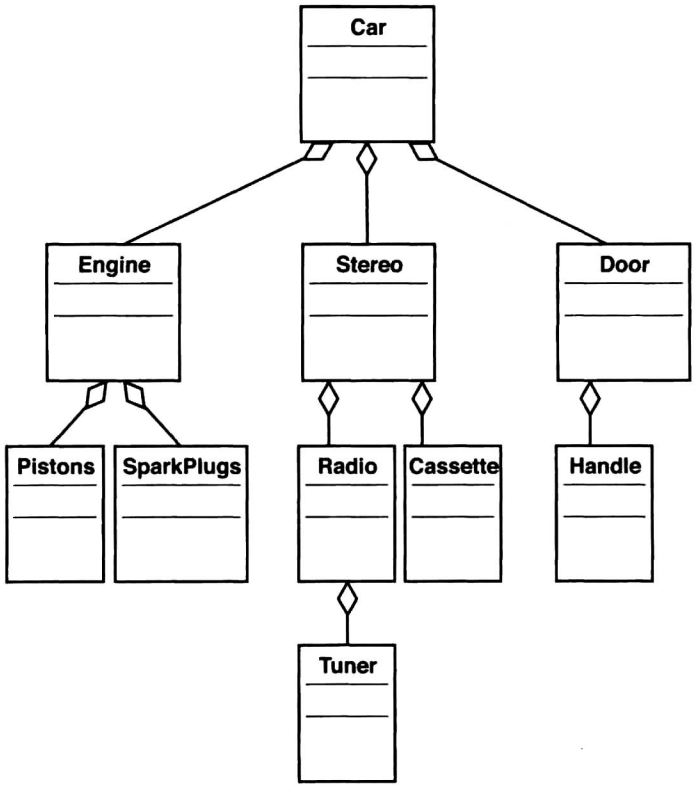


图 10-7 表示组合关系的扩展 UML 图

10.7.2 联合

聚合表示的是整体和部分的关系，意味着一个类由其他类组成，而联合则是两个类间提供服务。

之前提及过，客户端 / 服务器端关系是这种模型。显而易见，客户端不是服务器端的一部分，而且服务器端也不是客户端的一部分，它们相互依赖。大部分情况下是服务器端为客户端提供服务。在 UML 标记中，一条单纯的线表示这种关系，线的两端没有任何形状（见图 10-8）。

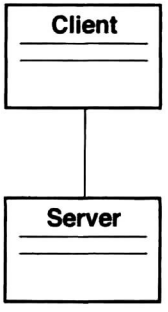


图 10-8 表示联合的 UML 图

注意，由于线的两端没有任何形状，所以看不出服务的流向。这只指出了两个类间存在联合。

我们用第 9 章中的计算机系统例子作为演示。该例子中有很多组件，比如计算机、显示器、扫描仪、键盘和鼠标。每个都是完全独立的组件，和计算机自身在某种程度上进行交互（见图 10-9）。

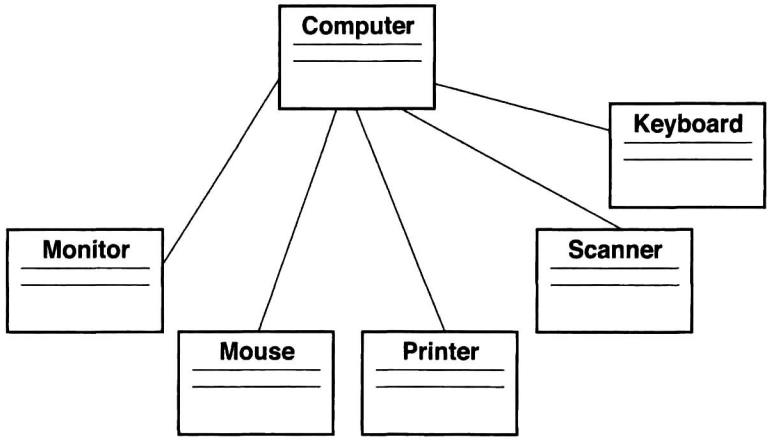


图 10-9　表示联合关系的扩展 UML 图

注意从技术角度来说，显示器是计算机的一部分。如果你想为计算机系统创建一个类，你可以使用聚合表示这种关系。然而计算机自身代表了一种聚合，因为它是由主板、RAM 等组成（见图 10-10）。

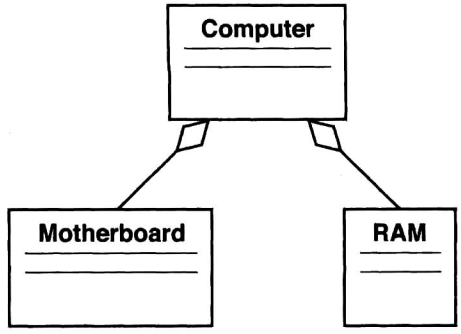


图 10-10　表示聚合的 UML 图

10.8　基数

本章最后一个知识点是基数。基数是类对应的对象的个数。再用前一章节中的例子，我们可以说计算机由一个且仅一个主板构成。基数表示为 1。计算机不包含主板是不行的，而且当今个人计算机也没有两个及两个以上的主板。另外，计算机必须拥有至少一个 RAM 芯片，但它也可以拥有多个芯片。因此我们可以表示基数为 1…*n*，*n*（从一般意义上来说）表示一个不受限的值。

受限的基数值

如果我们知道 RAM 芯片的插槽有 6 个，那么最大的限制数则不是无限的。因此 *n* 就是 6，而基数则是 1…6。

请看第 9 章中的图 9-7。

在本例中我们展示了几种基数。首先，`Employee` 类与 `Spouse` 类是联合关系。基于世俗规则，一个雇员要么没有配偶，要么只能有一个配偶（至少在我们的文化中，一个雇员不能拥有多个配偶）。因此这种联合关系的基数表示为 0…1。

Employee 类与 Child 类之间的联合关系与雇员理论上拥有孩子的数量不受限制这一实际情况略有不同。雇员可以没有小孩，但如果雇员有小孩，那么数量不存在上限。因此此联合的基数为 0…*n*，*n* 意味着系统对孩子数量无上限限制。

Employee 类和 Division 类之间的关系展示了每个雇员隶属于且只能隶属于一个部门。简单的 1 表示了这种联合。虽然有人认为这里设置基数为 1 有些牵强，但只是为了表示基数是对象模型的重要部分。

更多设计问题

在某些情况下，雇员可能属于多个部门。例如，大学可以允许一个人同时在数学学院和计算机科学学院任职。这是你需要考虑的另一个设计问题。

我们将讨论的最后一个基数是 Employee 类和 JobDescription 类之间的联合。在本系统中，雇员可以拥有无限数量的工作简介。但是与 Child 类不同，雇员可以有 0 个孩子，但系统中每个雇员至少要有一个工作简介。因此该联合的基数表示为 1…*n*。这种联合表示了每个雇员至少有一个工作简介，当然也可能有多个工作简介（本例中为无限数量）。

保存历史记录

你也必须考虑雇员可以拥有过往的工作简介，也包括当前的工作简介。这种情况下，需要区分出当前工作简介和历史工作简介。实现方式可以使用继承创建一组 job 对象，其中有个属性标示了这个对象是当前的工作简介。

10.9 结语

本章对本书中使用到的 UML 类图标记法做了简要的介绍。正如介绍中所说，UML 是个非常复杂和重要的话题，需要一本书甚至几本书来完整介绍 UML。

本书中使用类图来演示面向对象的例子。你不一定需要 UML 来设计面向对象的系统，但是 UML 可以协助开发面向对象的系统。我本人非常喜欢使用类图作为可视化工具来创建对象模型。

应该在熟悉了底层的面向对象概念之后再深入学习 UML。这又有点像先有鸡还是先有蛋的问题。使用 UML 来演示本书中的一些示例非常实用。

当解释面向对象的概念时使用建模语言（比如 UML）和编程语言（如 Java）非常必要。当然我们可以使用 C++ 替代 Java，使用除 UML 之外的其他建模系统。但请记住无论选择什么样的工具，都应该始终关注面向对象概念本身。

目前为止，本书已经介绍了大量的面向对象基本概念的材料。本书目的是能从一个全局视角来理解面向对象思考过程中使用到的概念。本书已经讲解了面向对象的基本概念，包括封装、继承、多态以及组合。本书的剩余部分则关注几个面向对象的技术。

10.10 引用

- Schmuller, Joseph. 2006. *Teach Yourself UML in 24 Hours,* 3rd ed., Indianapolis, IN: Sams Publishing.
- Booch, G., I. Jacobson, and J. Rumbagh. 2005. *The UML Users Guide,* 2nd ed., Boston, MA: Addison-Wesley.
- Lee, Richard, and William Tepfenhart. 2003. *Practical Object-Oriented Development with UML and Java.* Upper Saddle River, NJ: Prentice Hall.
- Ambler, Scott. 2003. *The Elements of UML Style.* Cambridge, United Kingdom: Cambridge University Press.
- Fowler, Martin. 2003. *UML Distilled,* 3rd ed. Boston, MA: Addison-Wesley Longman.
- Larman, Craig. May 1999. "What the UML Is—and Isn't." *Java Report,* 4(5): 20–24.

Chapter 11 第 11 章

对象与可移植数据：XML 和 JSON

面向对象的技术在过去的 10 年已经取得了显著的发展。在应用程序开发行业，对象已经成为主要技术之一。对象也在数据定义和传输方面取得了巨大进展。借助于可移植的代码，过去几年中发生了很多激动人心的大事。例如，Java 的成功很大程度上是由于它跨平台高度的可移植性。

Java 生成的字节码可以在多种平台上执行，条件是只要这些平台加载了 Java 虚拟机。.NET 框架提供了另一种非常重要的可移植的类型，即跨语言的移植性。C# .NET 生成的集合可以用于 Visual Basic .NET 应用程序，或其他的 .NET 语言。当然，尽管存在很多技术标准，但主要的编程语言（Java、.NET、Objective-C 等）还是占主导地位。大概未来将会有一种编程语言能够完全且高效地跨语言和平台。

可移植的语言是很强大的工具，但光实现可移植性还不够。使用这些语言编写的应用程序必须处理数据，而数据必须转换为信息。正是这些信息来驱动业务。信息是可移植方式的另一部分。

XML 是一种标准的机制，可以在完全不同的系统之间定义和传输数据（JSON 是另一种机制）。通过使用面向对象的语言（比如 Java、VB 和 C#）并结合面向对象的数据定义语言（比如 XML 和 JSON），在多个目的地之间移动数据变得高效和安全。XML 和 JSON 提供了一种在相互独立的应用程序之间共享数据的机制。

11.1 可移植数据

商业领域的一个历史问题就是数据存储格式的多样性。例如，假设阿尔法公司使用

Oracle 数据库系统来操作它的销售系统。而后来贝塔公司使用 SQL Server 数据库系统来操作其购买的系统。假设阿尔法公司和贝塔公司想要通过网络进行生意往来。构建这样的交易系统需要考虑很多因素，而其中之一就是两种数据库是不兼容的。我们的目标是使用 SQL 数据库为贝塔公司创建一个电子账单，该电子账单可以直接与阿尔法公司的销售系统进行交互，但阿尔法公司的销售系统使用的是 Oracle。

其实很多公司都需要在不用的组织间（或公司间）传输数据。电子商务系统都是基于互联网或本地网络，多种多样的业务促使了电子商务系统的多样化。

XML 提供了以多种方式传输数据的标准。通常可以认为数据能够以垂直和水平两种范式来传输。词条*垂直*意味着数据可以跨行业传输。在会计和财务的行业有自定义的标记语言（FpML，又称为财务产品标记语言）来提供标准的数据定义。垂直的应用程序提供商业模型和术语用于跨行业传输信息。这些标准常被称为*词汇表*。因此行业使用 XML 来建立词汇表。

采用 XML 标准的另一种方式是建立水平应用程序。*水平*的应用程序特定用于某个行业，比如零售业或运输业。在所有的电子商务程序中，首要需要考虑数据共享。图 11-1 展示了数据在各种各样的行业中通过垂直和水平传输。

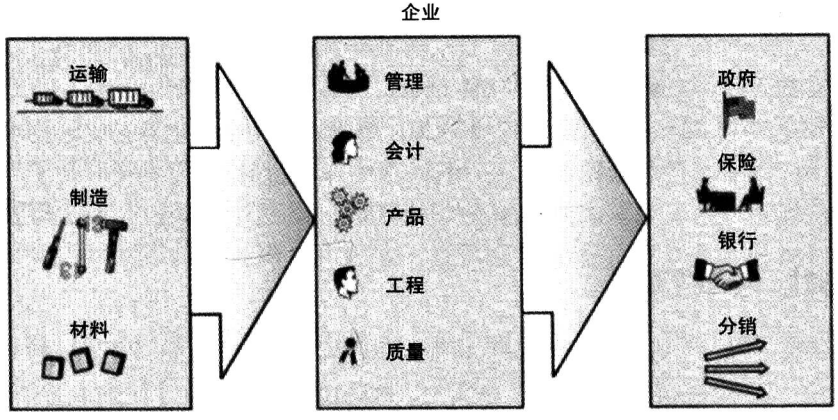


图 11-1　跨行业的 XML

一个有趣的行业 XML 应用程序的例子是食谱标记语言（RecipeML）。RecipeML 是一种 XML 词汇表，用来定义食品行业（比如酒店、餐厅、出版社）的标准。使用 RecipeML 可以让这些行业用标准的可移植的方式互相传输数据。基于 XML 标准的行业有法律、酒店住宿、会计、零售、旅行、财务和教育等。

这就是为什么要考虑数据的可移植性。尽管低级别的数据（在机器级别）确实不具有移植性，但我们需要在信息层面建立更高级别的移植性。所以 Java、.NET 和 Objective-C 在编程语言级别提供了不同层级的移植性，XML 提供的对信息的可移植性恰好符合我们的需求。

11.2 XML

XML 全称为扩展标记语言。你可能早已熟悉另一种标记语言，叫作超文本标记语言（HTML）。XML 和 HTML 是 SGML 的后代，SGML 是标准的通用标记语言。神奇的是，SGML 出现在 20 世纪 70 年代早期，而在 20 世纪 80 年代成为标准。

HTML 的主要功能是在浏览器中展示数据。它用超链接来组织数据，而浏览器则是实现这一目的的理想工具。然而 HTML 主要用来格式化和呈现数据，并不会定义和验证它。HTML 是 SGML 的子集，而且不包括 SGML 技术规格中的数据验证结构。原因是 SGML 非常复杂和精密，完全实现 SGML 标准则成本过高。而 HTML 在早期并不需要担心自身的数据验证问题。

XML 自身也不用担心数据验证问题。1997 年定义的 XML 是 SGML 的子集。XML 比用于呈现数据的 HTML 的格式要求更严格。XML 不是私有格式，世界万维网协会（W3C）这一组织目的在于建立和推广 XML 标准。我讲述很多 Web 技术时喜欢引用 w3schools 网站（http://www.w3schools.com/）。

在接下来的章节中，我们将看到 XML 如何运用在各种各样的面向对象的技术中（比如网站服务、分布式计算、对象持久化等）。本章我们将讲述 XML（和 JSON）的概念和语法。

Java 语言的问题之一是它是私有的（被 Oracle 拥有）。.NET 框架也是私有的（属于微软），而 Objective-C 也一样（属于苹果公司）。而 XML 的美好之处在于它是开放的技术。事实上，它是 IT 行业领导者倡导的几项技术之一。因此 XML 不会过时。

11.3 XML 与 HTML

XML 兴起之后，一些人推测 XML 将替代 HTML。很多人认为 XML 和 HTML 都是 SGML 的后代，而 XML 是升级版本。事实上，HTML 和 XML 的设计目的并不同。HTML 与 CSS（层叠样式表）一起决定了文档的结构和展示效果，而 XML 只用来描述数据。HTML 和 XML 在开发基于 Web 的系统中都是重要的工具。

XML 和 HTML 非常相似。不必惊讶，因为它们的来源相同。然而 XML 提供了两个 HTML 不具备的优势，就是验证文档以及格式化文档。

HTML 的标签是预定义的。诸如 `<HTML>`、`<HEAD>`、`<BODY>` 之类的标签是 HTML 规格说明定义的。你不能添加自定义标签。由于 HTML 用于格式化数据，所以不能自定义标签并不是什么大问题。XML 则用于定义数据。为了定义数据，你需要自定义标签名。所以存在一种叫作文档类型定义（DTD）的文档。DTD 用于定义描述数据的标签。当创建 XML 文档时，只可以使用预定义的标签。可以通过 DTD 验证所有的 XML 文档。XML 处理器读取 DTD 并且确定该文档是否合法。如果文档不合法，会产生语法错误。

合法文档

你不必强制使用 DTD。但使用 DTD 可以验证 XML 文档。对 XML 的唯一验证方式就是检查 XML 的格式是否正确。你需要显式包括一个 DTD 来检查文档的合法性。你可以在 DTD 中定义参数。

例如，如果创建订单系统，那么需要在 DTD 中创建一个名为 <PurchaseOrder> 的标签。如果你将该标签错误拼写为 <PurchasOrder>，那么格式检查会失败，该文档则会被标记为非法。

经过验证的 XML 文档更加健壮，尤其是处理数据时验证变得非常必要。例如，HTML 有很多成对的标签，比如 <FONT> 和 </FONT>。如果你忘记使用 </FONT> 标签来配对，那么浏览器依然能够加载该文档，但结果则不可预测。HTML 会尽力猜测并继续执行。而使用了 DTD 的 XML 则不接受最佳猜测。如果文档结构不正确，那么会产生一个错误，该文档是不合法的。

强制对文档进行验证并且保证文档是规整的，这是确保为行业提供共享信息的重要机制。

11.4　XML 和面向对象的语言

XML 紧密结合面向对象的语言提供“可移植的信息”。通常使用面向对象的语言开发的应用程序可以与 XML 进行交互。我们使用本章前面的例子作为演示。阿尔法公司作为一个百货商店使用 Oracle 数据库，而贝塔公司作为真空吸尘器生产商使用 SQL Server 数据库。阿尔法公司想要从贝塔公司购买一些真空吸尘器。所有的交易必须通过互联网进行。

简单来说，问题在于数据是存储在两个完全不同的数据库中。即使数据库是相同的，数据库中的记录格式也很有可能是不同的。因此需要考虑如何在阿尔法公司和贝塔公司之间共享数据，即在它们的数据库之间共享数据，但并不会在数据库之间直接建立物理连接。我们需要考虑如何处理业务，比如说一个公司发送订单，而接收的公司处理订单。

私有方案

我们可以创建一个私有应用程序建立起阿尔法公司和贝塔公司之间的连接。虽然这种方式是可以工作的，但更好的方式是使用一种更通用的方案（并且是面向对象的方式）。例如，阿尔法公司作为一个商场可能要求所有的供应商遵守它的订单规格说明。使用 XML 再合适不过。阿尔法公司可以创建一个 XML 规格说明并提供给所有的供应商。

为了达到连接这两个公司系统的目的，阿尔法公司可以生成一个 XML 规格说明来描述如何完成对信息的传输和存储。这可以使用面向对象的语言来实现。诸如 Java、VB 或 C# 之

类的语言可被用于从阿尔法公司的 SQL Server 数据库中获取数据，并且基于双方都认同的标准创建 XML 文档。

该 XML 文档然后可以通过互联网发送给贝塔公司。贝塔公司可以使用协商好的 XML 标准来从 XML 文档中抽取信息并保存到 Oracle 数据库中。图 11-2 表示了数据从一个数据库到另一个数据库的流程。在该图中，应用程序\解析器从 SQL 数据库中提取数据并通过网络发送给另一个应用程序\解析器。该解析器然后将数据转换为 Oracle 数据库支持的格式。

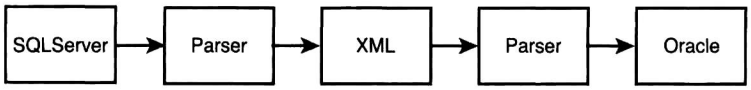


图 11-2　应用程序之间传递数据

解析器

解析器是一个读取文档并提取指定信息的应用程序。例如，编译器会包含一个解析器。解析器读取程序的每一行并使用特定的语法规则来处理代码。解析器可以验证 print 语句的语法正确性。

11.5　在企业间共享数据

作为阿尔法公司和贝塔公司例子的拓展，可以实现两者之间的通信机制。我们会创建一个在两个公司进行交易的简单的 XML 文档。在本例中，我们创建了一个简单的文档，包含了表 11-1 中的信息。该表格定义了在公司间传输的数据格式。

表 11-1　传输数据规格说明

对　象	类　别	字　段
supplier		
	name	
		<companyname>
	address	
		<street>
		<city>
		<state>
		<zip>
	product	
		type
		price
		count

11.6　使用 DTD 验证文档

在本例中，我们将从贝塔公司向阿尔法公司发送 XML 文档。XML 文档表示一条交易，里面包括公司名、公司地址和一些产品信息。注意这些信息是嵌套的。也就是说，整个文档可以被描述为一个对象，即供应商对象，而嵌套的供应商识别信息为公司名、公司地址以及产品信息。注意地址和产品识别也包含嵌套信息。首先定义 DTD（文档类型定义）来指导交易格式。列表 11-1 展示该 DTD。

列表 11-1　用于验证的数据定义文档

```
<!-- DTD for supplier document -->
<!ELEMENT supplier ( name, address)>
<!ELEMENT name ( companyname)>
<!ELEMENT companyname ( #PCDATA)>
<!ELEMENT address ( street+, city, state, zip)>
<!ELEMENT street ( #PCDATA)>
<!ELEMENT city ( #PCDATA)>
<!ELEMENT state ( #PCDATA)>
<!ELEMENT zip ( #PCDATA)>
```

DTD 定义了如何构造 XML 文档。组成该文档的标签与 HTML 标签很相似。第一行是一条 XML 注释：

```
<!-- DTD for supplier document -->
```

XML 的注释与其他编程语言的注释功能相同，即用来文档化代码。XML 使用注释来让文档更加容易阅读和理解。不要在文档中加入过多的注释，否则文档阅读起来会更困难。该文档只包含一条注释。

剩下的行定义了 XML 文档的结构。请看第一行：

```
<!ELEMENT supplier ( name, address, product)>
```

该标签定义了一个名为 supplier 的元素。之前已经说过，一个供应商包括名称、地址和产品。因此当 XML 解析器实际解析 XML 文档时，该文档必须是一个包含了名称、地址和产品的供应商。

接下来的一行可以看到该元素由另一个名为 <companyname> 的元素组成：

```
<!ELEMENT name ( companyname)>
```

<companyname> 元素则是由 #PCDATA 设计的数据元素定义的：

```
<!ELEMENT companyname ( #PCDATA)>
```

该标签结束了元素树的整个层级结构。这个 DTD 文件被命名为 supplier.dtd。你可以使用任何文本编辑器来创建这个 DTD。很多集成工具和环境也可以用来创建这个文档。图 11-3 使用 Notepad 来展示该应用程序的 DTD。

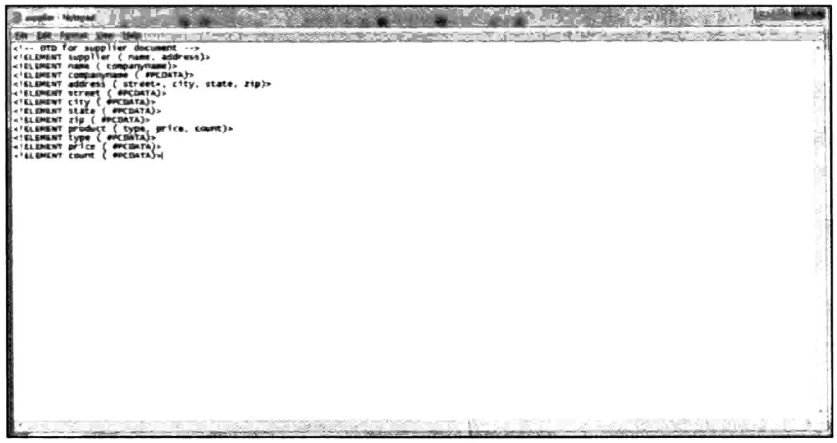


图 11-3　在 Notepad 中创建 DTD

文档验证

指定了 DTD 的文档是否合法取决于该 DTD。如果文档没有指定 DTD，那么无法判断该 XML 文档是合法还是非法的。XML 文档可以将 DTD 内嵌在文档中，也可以指定一个外部的 DTD。由于外部的 DTD 提供了一种更强大的机制，所以本例会使用外部的 DTD。

PCDATA

PCDATA 全称为解析字符数据，是从文本文件中解析字符信息的标准。该解析器会转换任何数字（比如整数）。

11.7　将 DTD 集成到 XML 文档中

我们已经创建了 DTD，接下来可以创建一个真实的 XML 文档。XML 文档不一定都要遵守我们刚编写的供应商 DTD。

表 11-2 中，我们识别了一些需要包含在 XML 文档中的实用信息。而这些数据都包含在根节点元素中，而不是在 address 和 name 之类的聚合元素中。

表 11-2　向表中添加值

对　象	类　别	字　段	值
supplier			
	name		
		<companyname>	The Bata Company
	address		
		<street>	12000 Ontario St
		<city>	Cleveland
		<state>	OH
		<zip>	24388

为了将信息保存到 XML 文档中，我们可以使用和 DTD 一样的文本编辑器。其实已经有一些专门的工具来创建 XML。图 11-4 展示了使用 Notepad 编写的 XML 文档。该文档被称为 beta.xml。

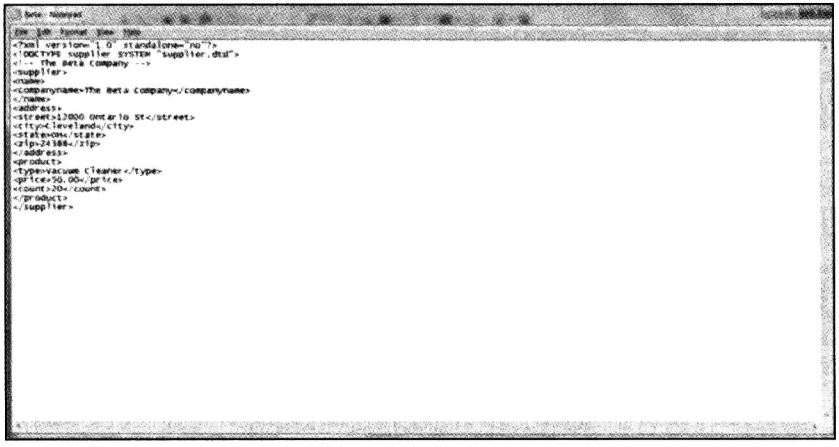


图 11-4　贝塔公司包含了 DTD 的 XML 文档

注意第二行将该文档与之前定义的 supplier DTD 文件绑定到了一起：

```
<!DOCTYPE supplier SYSTEM "supplier.dtd">
```

请看图 11-4，标签结构印证了规格说明。请注意这些标签是嵌套的，而且只有结束标签能包含数据。一些标签是高层级的标签，某种意义上和抽象类的概念很相似。你可以认为 <address> 标签是“抽象的”，因为我们没有真正定义它。然而 <street> 标签是“实在的”，因为我们可以给它实际赋值。换句话说，<street> 标签包含信息，而 <address> 标签则没有：

```
<address>
<street>12000 Ontario St</street>
```

还有更好的方式来检查 XML 文档。之前说过，可以使用很多工具开发 XML 文档。早期的一个工具叫作 XML Notepad，它和微软操作系统提供的 Notepad 很相似。

> **XML Notepad**
>
> 目前你在互联网上简单搜索“XML Notepad”就可以找到 XML Notepad。很多网站提供下载。

XML Notepad 有助于我们理解 XML 文档的结构。当安装 XML Notepad 后，可以打开 beta.xml 文件。图 11-5 展示了使用 XML Notepad 打开 beta.xml 文件后的样子。当打开该文档后，展开所有的加号标记来查看整个文档。

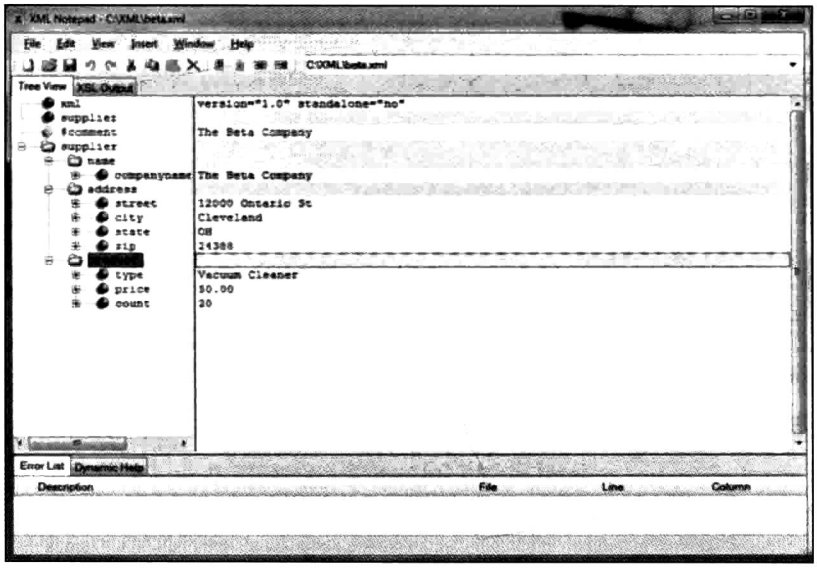


图 11-5 使用 XML Notepad 打开 `beta.xml` 文件

XML Notepad 列出了该文档的所有层级，由 supplier 标签开始。注意我们之前说过，只有结束元素才能包含信息。

开发 DTD 的显著优势是可应用于多个文档，比如应用于多个供应商。假设有一个生产冰鞋的公司，叫作伽马公司，它也想向阿尔法公司供货。伽马公司需要做的事情是创建一个遵守 supplier DTD 的 XML 文档。使用 XML Notepad 打开该文档的截图如图 11-6 所示。

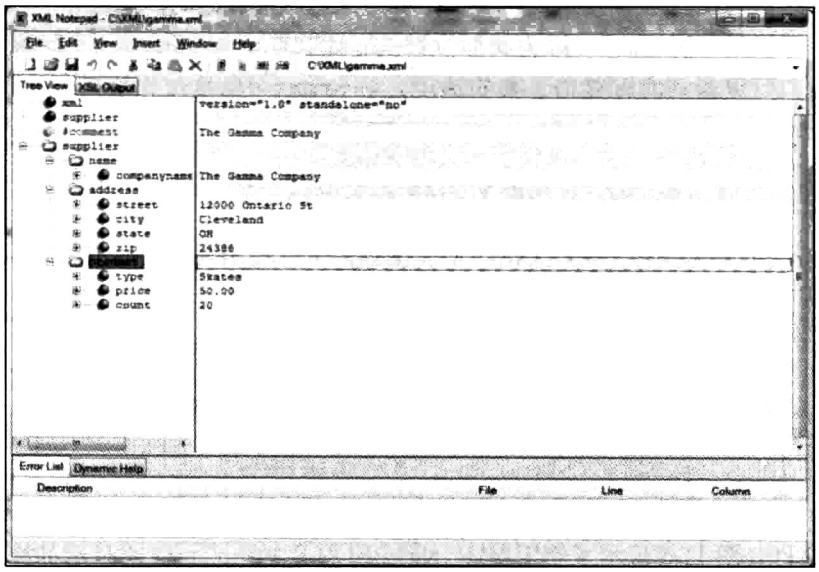


图 11-6 使用 XML Notepad 打开的 `gamma.xml` 文件

注意 beta.xml 和 gamma.xml 都遵守 supplier DTD。那么如果 XML 文档没有遵守 DTD 会怎样？DTD 会起作用。我们来故意在 gamma.xml 中创建一个错误，即移除与名称有关的所有信息：

```
<name>
<companyname>The Gamma Company</companyname>
</name>
```

对于 supplier DTD 来说，我们创建了一个非法的文档。请看图 11-7 中非法的文档。请注意 Notepad 不会发现该文档是非法的，因为 Notepad 不会检查文档是否合法。你需要使用 XML 验证器来检查合法性。

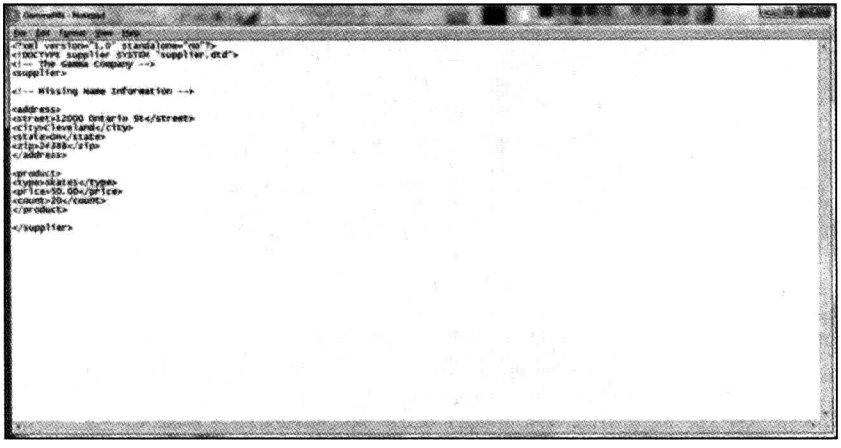


图 11-7　非法的文档（没有名称信息）

现在有了一个基于 supplier DTD 的非法文档。我们如何验证它是否合法？可以使用 XML 验证器来打开该非法的 gamma.xml 文档，比如 w3schools 网站提供的一个（www.w3schools.com/xml/xml_validator.asp）。注意图 11-8 展示的结果。XML 验证器提供了一个对话框指出该文档是非法的。

XML 验证

很多应用程序都可以验证 XML 代码。w3schools 网站的 XML 验证器是其中之一，而且简单易用。

由于 supplier DTD 期望文档遵守它的定义，否则会产生错误。事实上，错误信息非常明确地指出了问题所在，DTD 期望名称信息。因此为该系统创建正确的 XML 文档，必须提供所有对应的信息，而且使用正确的格式。文档中的 <address> 标签也必须是合法的。

需要指出 HTML 不会进行这种类型的检查。事实上，即使 XML 文件结构是非法的，但仍然可以用浏览器打开，请看图 11-9。

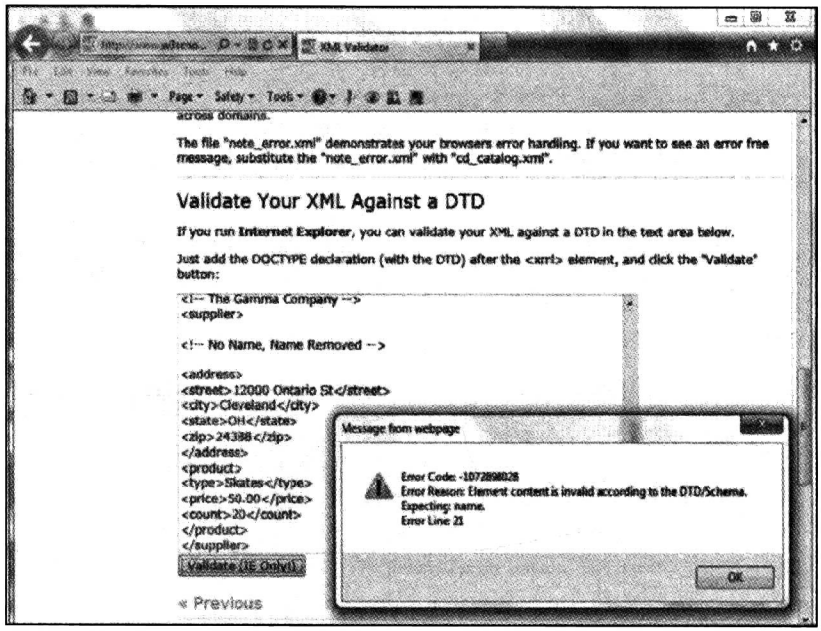


图 11-8　通过 w3schools 网站验证的非法 `gamma.xml` 文档

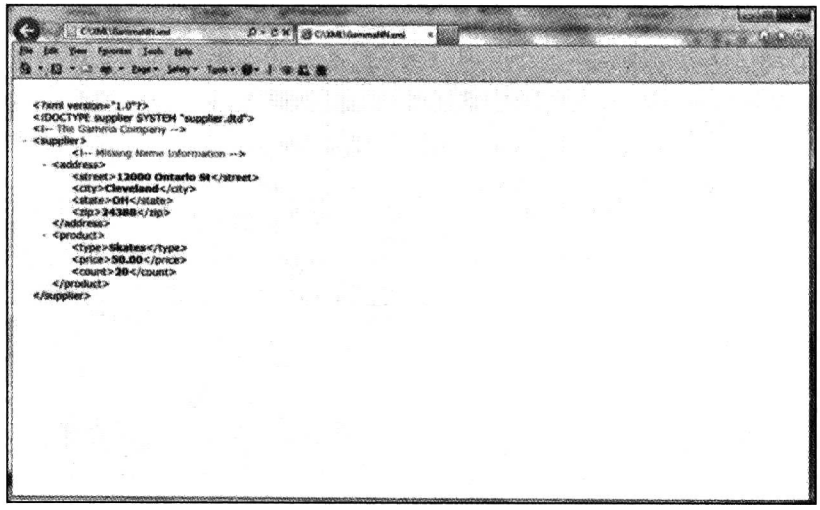


图 11-9　使用 Internet Explorer 打开 `gamma.xml` 文档（不含名称信息）

注意，尽管该文档是非法的，浏览器还是可以打开它，甚至显示它。这是因为浏览器不会检查确保该文档遵守了 DTD，而 XML 验证器则会进行检查。理论上来说，这是在与数据交互时（使用验证器）XML 提供的一个重要优势。HTML 用于显示数据，XML 用于格式化数据，这意味着 XML 更加严格。这是两者之间重要的不同之处。

你可能会问诸如 XML Notepad 之类的工具在整个 supplier 例子中有哪些好处？用处是什

么？第一个问题的答案是，XML Notepad（或类似的编辑器）可以在早期验证文档。第二个问题的答案是，XML Notepad（或类似的编辑器）可用来构建文档。

11.8　使用层叠样式表

从技术角度来说，可移植数据的概念往往关注于在不同终端间传输数据。然而将数据从 A 送到 B 没有任何价值，除非能以恰当的方式展示数据。因此我们必须考虑在 XML 系统中传输的数据如何展现给用户。

请记住 XML 主要用于定义数据，而 HTML 则基本上是一种展示机制。不过 XML 和 HTML 都可以用于在浏览器中展示数据。

虽然 XML 通常不用于展示目的，有方法来格式化 XML。方式之一是使用 CSS。CSS 在 HTML 世界中被广泛使用来格式化内容。从某个角度来说，CSS 也可以用来格式化 XML。在 supplier XML 文档中包含对 `<companyname>`、`<street>`、`<city>`、`<state>`、`<zip>` 的定义。假设我们想格式化这些定义的元素，可以提供一个规格说明来格式化 XML 文档中的元素，如表 11-3 所示。

表 11-3　层叠样式表规格说明

标　签	字　体	大　小	颜　色	显示方式
`<companyname>`	Arial，sans serif	24	Blue	Block
`<street>`	Tims New Roman，serif	12	Red	Block
`<city>`	Courier New，serif	18	Black	Block
`<state>`	Tahoma；serif	16	Gray	Block
`<zip>`	Arila Black，sans serif	6	Green	Block

将以下样式表内容放置到 CSS 中：

```
companyname{font-family:Arial, sans-serif;
    font-size:24;
    color:blue;
    display:block;}
street {font-family:"Times New Roman", serif;
    font-size:12;
    color:red;
    display:block;}
city {font-family:"Courier New", serif;
    font-size:18;
    color:black;
    display:block;}
state {font-family:"Tahoma"; serif;
    font-size:16;
    color:gray;
    display:block;}
```

```
zip {font-family:"Arial Black", sans-serif;
    font-size:6;
    color:green;
    display:block;}
```

通过在 XML 文档中添加以下一行代码来应用样式表。

```
<?xml-stylesheet href="supplier.css" type="text/css" ?>
```

例如，ZIP 邮编以前只是简单的文本展示，现在使用了 Arial Black 字体、绿色以及字号 6 来格式化。属性 `display:block` 将会把每个属性呈现到新行中。

请看新加的代码：

```
<?xml version="1.0" standalone="no"?>
<?xml-stylesheet href="supplier.css" type="text/css" ?>
<!DOCTYPE supplier SYSTEM "supplier.dtd">
<!-- The XML data -->
<supplier>
<name>
<companyname>The Beta Company</companyname>
</name>
<address>
<street>12000 Ontario St</street>
<city>Cleveland</city>
<state>OH</state>
<zip>24388</zip>
</address>
</supplier>
```

我们可以在浏览器中打开该包含 CSS 的 XML 文档。图 11-10 演示了它的展现。

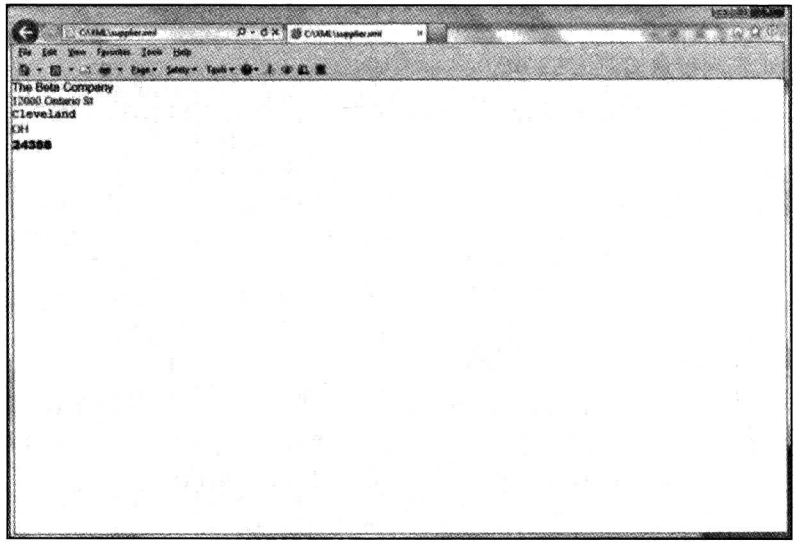


图 11-10　使用了层叠样式表的 XML 文档

请再通过图 11-9 查看未使用 CSS 的文档的展现效果。

11.9　JavaScript 对象标记

可移植性数据这一概念非常强大，而且 XML 在数据交换领域是广泛采纳的标准，但我们仍有其他的标准可供选择。整个行业在创新和改变的过程中很难确保行业标准的一致性。例如，很多开发人员见识了强类型的强大之处，但编译型语言很难忽略类型，而一些诸如 Perl 之类的弱类型的语言也颇受欢迎。在处理数据时也会有同样的困扰。

尽管 XML 更加结构化，特别是结合 DTD 来进行使用，而诸如 JavaScript 对象标记（又称为 JSON）则“更加灵活”。w3schools 网站（http://www.w3schools.com/json/default.asp）对 JSON 提供了以下的描述：

- JSON 是轻量级文本数据交换格式。
- JSON 与语言无关。
- JSON 是“自描述的”，并且容易理解。

*JSON 使用 JavaScript 语法来描述数据对象，但 JSON 依然是语言和平台无关的。很多编程语言都存在 JSON 解析器和 JSON 库。

理解 JSON 的关键点是 JSON 使用与 JavaScript 完全一致的语法来创建对象。因此 w3schools 网站解释道：除了使用解析器，JavaScript 程序可以使用内建的 `eval()` 函数来执行 JSON 数据并创建原生的 JavaScript 对象。事实上，你可以用多种方式解析该数据，一些开发人员不屑于使用 `eval()` 函数。

为了展示该概念，我们将本章之前创建的 XML 对象转换为等同的 JSON 格式。我们来看看通过 s3schools 网站创建的这个简单的 JSON 对象：

```
{
"employees": [
{ "firstName":"John" , "lastName":"Doe" },
{ "firstName":"Anna" , "lastName":"Smith" },
{ "firstName":"Peter" , "lastName":"Jones" }
]
}
```

注意，该 JSON 由花括号包裹的代码来表示。JSON 对象本质上也是一个属性，是一个名称 / 值对。为了整体展示 JSON 对象，请看 w3schools 网站的完整的代码示例。你可以使用网站提供的编辑器，地址是 http://www.w3schools.com/json/json_intro.asp。

```
<!DOCTYPE html>
<html>
 <body>
 <h2>JSON Object Creation in JavaScript</h2>
<p>
 Name: <span id="jname"></span><br />
```

```
Age: <span id="jage"></span><br />
Address: <span id="jstreet"></span><br />
Phone: <span id="jphone"></span><br />
</p>

<script type="text/javascript">
 var JSONObject= {
 "name":"John Johnson",
 "street":"Oslo West 555",
"age":33,
 "phone":"555 1234567"};
document.getElementById("jname").innerHTML=JSONObject.name
document.getElementById("jage").innerHTML=JSONObject.age
document.getElementById("jstreet").innerHTML=JSONObject.street
document.getElementById("jphone").innerHTML=JSONObject.phone
</script>
</body>
</html>
```

w3schools 网站的 JSON 编辑器

注意 w3schools 网站使用的编辑器只能使用单个完整的文件。因此如果想使用单独的文件（比如样式表等），你需要将该文件嵌入到编辑器中。总之，该编辑器只能处理单个文件，主要用来测试一些概念。当创建应用程序时，需要将代码放置在不同逻辑的文件中。

我们将早期创建的 XML 对象转换为 JSON 对象：

```
var address= {
 "street":"23456 Main St",
 "city":"Cleveland",
 "state":"OH",
 "zip":"24388"
 };
```

该 JSON 对象是一个 JavaScript 联合数组，使用了名称 / 值对。图 11-11 展示了如何在 HTML 文档中使用 JSON 对象。

为了加深理解，我们以同样的方式来展示 JSON 代码，可以使用以下样式表：

```
<!DOCTYPE html>
<html>
 <head>
 <style type="text/css">
 companyname{font-family:Arial, sans-serif;
     font-size:24;
     color:blue;
     display:block;}
 street {font-family:"Times New Roman", serif;
```

```
    font-size:12;
    color:red;
    display:block;}
city {font-family:"Courier New", serif;
    font-size:18;
    color:black;
    display:block;}
state {font-family:"Tahoma"; serif;
    font-size:16;
    color:gray;
    display:block;}
zip {font-family:"Arial Black", sans-serif;
    font-size:6;
    color:green;
    display:block;}
 </style>
 </head>

 <body>
 <companyname>The Beta Company</companyname>

<p>
<street> <span id="jstreet"></span><br /> </street>
<city> <span id="jcity"></span><br /> </city>
<state> <span id="jstate"></span><br /> </state>
<zip> <span id="jzip"></span><br /> </zip>
</p>

<script type="text/javascript">
 var address= {
 "street":"23456 Main St",
 "city":"Cleveland",
 "state":"OH",
 "zip":"24388"
 };

document.getElementById("jstreet").innerHTML=address.street
document.getElementById("jcity").innerHTML=address.city
document.getElementById("jstate").innerHTML=address.state
document.getElementById("jzip").innerHTML=address.zip
</script>
</body>
</html>
```

在 w3schools 编辑器中执行这段代码，输出展示在图 11-12 中。

最后来比较 Internet Explorer 浏览器中 XML 例子中的输出。XML 版本的输出请参见图 11-10。而我们在浏览器中打开 JSON（嵌入在 HTML 中）来进行对比，JSON 版本的输出请参见图 11-13。

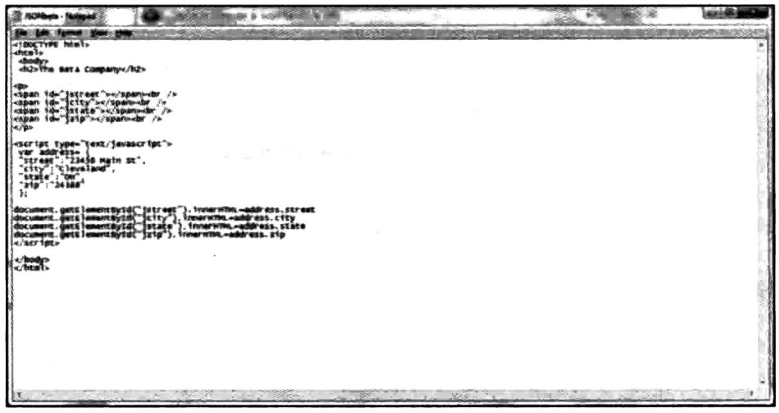


图 11-11　使用了 JSON 对象的 HTML 文档

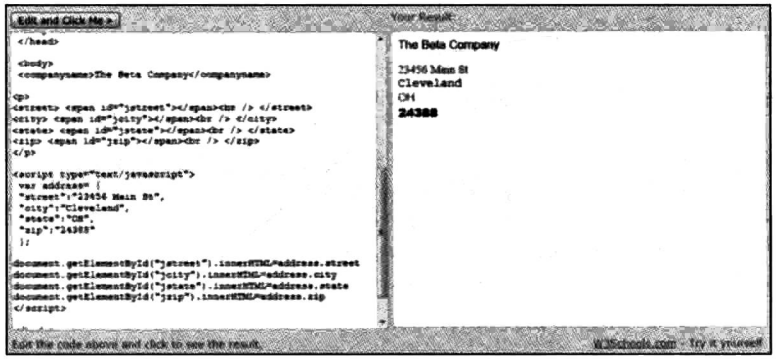


图 11-12　使用 w3schools 编辑器在 HTML/CSS 文档中采用 JSON 对象

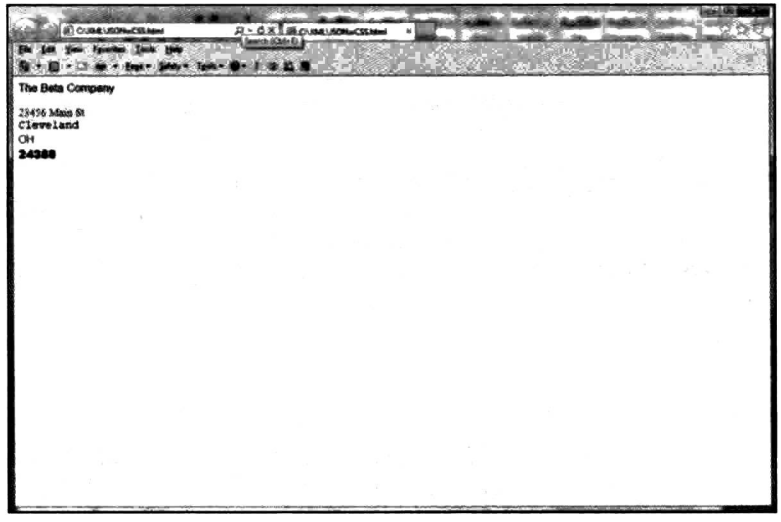


图 11-13　XML 文件的 JSON 实现

XML 和 JSON 实现可移植性数据的方式不同，但目的一致。它们的共同目标是可以轻松解析、分享和使用包含在对象中的传输数据。之前小节说过，很多人喜欢使用 JSON 是因为它的结构比 XML 简单，而且 JSON 处理速度更快。

11.10　结语

在本章中我们全方位讨论了 XML 技术，因为它是 IT 行业中很重要的技术。IT 市场中很多厂商遵从一致的标准这一情形非常罕见，而 XML 则是其中一个。我们也快速浏览了另一个替代技术，JSON。XML 和 JSON 都用来传输所谓的可移植性数据。

从面向对象的角度来说，你应该远离本章，因为学习面向对象的开发过程优先于学习面向对象的编程语言以及数据处理。然而数据是信息系统的根本，所以设计面向对象的系统时要重点关注数据。在当今的业务环境中需要重点考虑如何在不同的系统和平台下传输数据。

XML 和 JSON 还有很多相关知识值得深究。本书只是简单的概念介绍，目的是对 XML 和 JSON 有一个通用了解，以及如何使用一些相关工具。本章还提及了样式表，使用 CSS 以及其他技术可以更好地格式化 XML 文档。

11.11　引用

- Marcotte, Ethan. 2011. *Responsive Web Design*. New York, NY: A Book Apart.
- http://www.w3schools.com/json/default.asp
- Deitel, Harvey and Paul Deitel. 2010. *JavaScript for Programmers*, Upper Saddle River, NJ: Pearson Education, Inc.
- Hughes, Cheryl. 2003. *The Web Wizard's Guide to XML*. Boston, MA: Addison-Wesley.
- Watt, Andrew H. 2003. *Teach Yourself XML in 10 Minutes*. Indianapolis, IN: Sams Publishing.
- McKinnon, Al and Linda. 2003. *XML: Web Warrior Series*. Boston, MA: Course Technology.
- Holtzer, Steven. 2003. *Real World XML*. Indianapolis, IN: New Riders.

Chapter 12 第 12 章

持久化对象：序列化、封送及关系型数据库

不管要创建什么类型的业务系统，几乎都需要使用数据库。事实上，我最喜欢的一句与软件开发相关的话是，“一切关乎数据”。简单来说，当开发软件应用程序时无论使用的是什么样的硬件、操作系统、应用程序软件，而数据需求往往是创建该系统的首要原因。

12.1 持久化对象基础

当应用程序实例化一个对象时，该对象的生命周期依赖于应用程序自身。如果实例化了一个 Employee 对象，其包含 name、ss# 等属性，直到该应用程序终止前该 Employee 对象将一直存在。图 12-1 简单直白地演示了典型的对象生命周期。当应用程序创建了对象后，该对象的存活周期被限制在该应用程序的运行周期之内。当应用程序结束后，该对象会被销毁。为了让对象继续存活，必须将其写入某种持久化存储中。

当实例化和初始化 Employee 后，该对象会拥有一个明确的状态。对象的状态是由其属性值决定的。如果想保存 Employee 对象的状态，必须采取一些措施。保存对象的状态以便以后使用，这一概念被称为持久化。我们使用术语持久化对象来定义一个不依赖于单个应用程序的对象，该对象可以被存储并稍后再次使用。图 12-2 演示了使用了持久化的对象的生命周期。在该图中，应用程序 1 中创建了一个对象，然后将该对象写入一个存储设备（比如数据库中）。由于该对象被持久化存储了，所以其他的应用程序可以访问它。在本图中，应用程序 2 可以实例化该对象并且加载该持久化对象的内容。

有很多种方式来保存对象的状态。以下列出了一些方式：

- 保存到平面文件中。

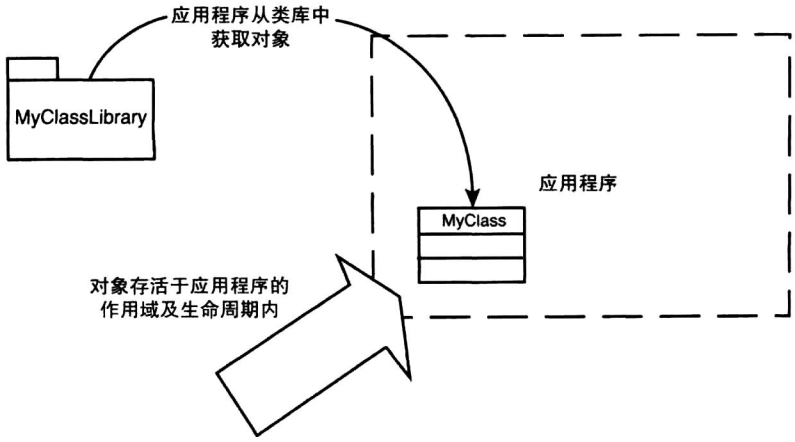


图 12-1　对象生命周期

- 保存到关系型数据库中。
- 保存到对象数据库中。

如何保存对象的最简单的演示方式是编写代码将对象写入到平面文件中，因为很多人通过家用计算机无法访问对象数据库或工业级别的关系型数据库。如果只是出于演示目的，可以将对象写入平面文件中，在商业应用程序中使用平面文件则不是很规范。

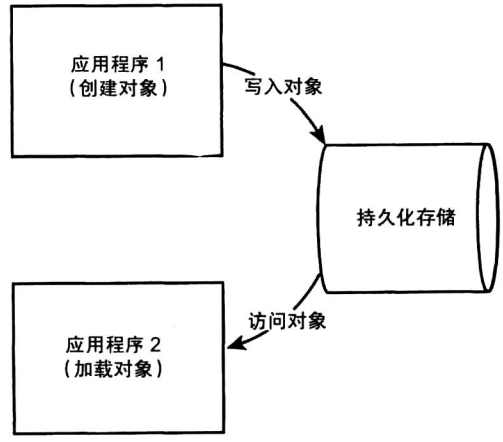


图 12-2　采用了持久化的对象生命周期

12.2　将对象保存到平面文件中

本小节使用平面文件演示对象持久化。我定义了一个简单的由操作系统管理的平面文件。平面文件是个基本概念，无需用很大篇幅来描述。

> **平面文件**
>
> 很多人并不喜欢术语平面文件。单词“平面”暗示对象是平的。从某种方面来说的确如此。对于存储和传输对象来说，不管对象有多复杂，都要将其摊平。

你可能认为对象不能像一个变量一样简单地存放到文件中，事实的确如此。保存对象状态这一课题已经成为软件开发领域的一个分支，本章稍后会详细讨论。通常如果你将一些变量存放在文件中，那么要先确定变量顺序以及变量分隔方式（比如逗号分隔），然后将其写入文件中。该文件可以采用逗号分隔，或者采用其他任何你喜欢的协议。

问题在于对象并不是一组变量的简单集合。对象可以看作由几部分组成的一个独立的单

元。因此必须先解耦对象，然后将其写入存储媒介（比如平面文件）。在该对象被解耦并写入平面文件后，还要考虑如何重新构造该对象，即将各部分再组合到一起。

存储对象还要考虑的另一个问题是对象可以包含其他对象。比如 Car 对象可能包含 Engines 和 Wheels 之类的对象。当将对象保存到平面文件时，必须保存整个对象，Car、Engines 等对象都需要保存起来。

现代语言都有内置的机制实现对象持久化。例如，Java 和其他基于 C 的语言，经常使用流的概念来处理 I/O。Java 通过一个流将对象保存到文件中。对象必须实现 Serializable 或 Externalizable 接口，这样能保证它能被流写入文件中。

本方式的缺点在于这种方案是私有的，你必须使用 Java 来实现。事实上，保存和恢复对象都要使用 Java。另一种解决该问题的更具移植性的方式是创建一个 XML 文档作为中间文件，使用开放的 XML 技术解耦并重新构造对象。

我们在本章中会使用这两种方式。首先用 Java 演示 Java 的序列化技术，然后会使用 XML 策略来完成一个 C#.NET 示例。

12.2.1 序列化文件

作为示例，请看以下一个名为 Person 的类的 Java 代码：

```
package Serialization;
import java.util.*;
import java.io.*;

class Person implements Serializable{

    private String name;

    public Person(){
    }

    public Person(String n){
        System.out.println("Inside Person's Constructor");
        name = n;
    }

    String getName() {
       return name;
    }

}
```

该类非常简单，只包含一个属性，用来表示此人的姓名。

要额外注意下面这行代码，这说明该类实现了 Serializable 接口。如果查看 Java 文档，会发现 Serializable 接口只包含少量代码，事实上它仅仅意味着该对象可以被序列化：

```
class Person implements Serializable
```

该类也包含一个名为getName的方法，它会返回该对象的名称。除了实现了Serializable接口，该类再没有其他特别之处。而Serializable正是关键所在。我们将要编写一个应用程序，该应用程序会将该对象写入一个平面文件中。这个应用程序被称为SavePerson，如下所示：

```
package Serialization;
import java.util.*;
import java.io.*;
public class SavePerson implements Serializable{

    public SavePerson(){

        Person person = new Person("Jack Jones");

        try{
            FileOutputStream fos = new FileOutputStream("Name.txt");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            System.out.print("Person's Name Written: ");
            System.out.println(person.getName());

            oos.writeObject(person);
            oos.flush();
            oos.close();
        } catch(Exception e){
            e.printStackTrace();
        }

    }
}
```

尽管其中一些代码深度使用了 Java 功能，不过我们还是能认识到对象是如何被序列化并写入平面文件的。

Java 代码

这里没有展示出全部代码（比如文件 I/O 部分的代码），你可以参考本章最后引用部分列出的书籍来学习这些代码。

现在这是一个真实的应用程序了。这个代码有一个主方法。该应用程序做了三件事：

- 实例化一个 Person 对象。
- 序列化该对象。
- 将该对象写入文件 Name.txt。

序列化并写入该对象的代码如下所示：

```
oos.writeObject(person);
```

这比挨个保存每个属性值的方式要简单得多。这种将对象直接写入文件的方式非常便利。

12.2.2 再次讨论实现和接口

请注意文件序列化的底层实现比其接口要复杂得多。本书最重要的主题之一是分离接口和实现。通过提供一个易用的接口来隐藏底层的实现，从而保证用户能轻松使用该功能。

序列化文件是阐述接口与实现之间的区别的好例子。程序员只需使用接口就能将对象写入文件中，不用关心具体实现的任何细节。只需关心以下几点：

- 可以将对象作为独立的单元写入。
- 可以严格按照存储时的格式恢复该对象。

这就像使用汽车一样。启动汽车的接口是点火装置上的钥匙，用于启动引擎。大多数人不用知道或关心启动操作的具体技术问题。他们只关心汽车能被启动即可。

SavePerson 程序将对象写入文件 Name.txt 中。以下的代码则恢复该对象：

```
package Serialization;
import java.io.*;
import java.util.*;

public class RestorePerson{

    public RestorePerson(){
        try{
            FileInputStream fis = new FileInputStream("Name.txt");
            ObjectInputStream ois = new ObjectInputStream(fis);

            Person person = (Person )ois.readObject();
            System.out.print("Person's Name Restored: ");
            System.out.println(person.getName());
            ois.close();
        } catch(Exception e){
            e.printStackTrace();
        }

    }
}
```

以下代码将对象从文件中获取出来：

```
Person person = (Person )ois.readObject();
```

注意该对象是从平面文件中构造出来的，这里实例化和初始化了一个新的 Person 对象。该 Person 对象是我们存储在 SavePerson 应用程序中的 Person 对象的真实副本。图 12-3 展示了 SavePerson 和 RestorePerson 应用程序的输出。

图 12-3　序列化对象

注意图 12-3 中当执行应用程序时，该 Person 对象的名称“Jack Jones”存储到了 Name.txt 文件中，当 RestorePerson 被执行时会恢复该对象。当对象被恢复后，我们可以访问该 Person 对象的属性。

12.2.3　为什么不保存方法

我们讨论对象持久化时说，当保存对象时，很容易看到属性是如何被保存的。那么你可能会问那方法呢？

对对象的一种定义是它包含属性和行为，或者换句话说，它包含数据和方法。当存储对象时是否需要存储方法？

在上面 Java 序列化的例子中，方法并没有被显式保存。注意我们已经说明 Java 既用于存储对象，也用于恢复对象。事实上定义类时就限制了存储和恢复时对象的类型都必须是同一个类。

因此在 Person 对象例子中，SavePerson 应用程序和 RestorePerson 应用程序都必须访问 Person 类。动态访问 Person 类是可能的，但应用程序必须使用同一个类。因此方法自身无需保存到数据存储中。

不过一些人可能有些担心这破坏了对象是由属性和行为组成的这一概念。其实它并没有破坏概念，只是物理实现不一定要契合概念模型。

12.3　序列化过程中使用 XML

尽管使用专门的序列化技术比较便利和简洁，但不具有可移植性。XML 是定义数据的标准，所以我们可以为上面的序列化例子创建一个 XML 模型。至少从理论上来说这种模型可以跨多平台和多语言。本小节中创建的 XML 模型中的例子可以使用 C# .NET。实际上，

你使用 Java 程序或其他语言来访问生成的 XML 文件也不会有任何问题。

XML 模型和 Java 序列化模型的主要区别在于使用 XML 模型需要生成一个 XML 文档。该文档表示了 Person 类的属性。这种方式对于 Person 类说可能稍显复杂，但这种语法提供了一种结构化的封装。

我们先来看 C# 代码。Person 类与之前的主要区别在于定义属性的方式不同。尽管很多代码与非 XML 方式的结构很相似（比如构造函数、行为等），但数据则是以 XML 方式定义的。

例如，直接在代码中植入了 XML 根节点的定义、属性以及元素。这些定义如下所示：

```
    [XmlRoot("person")]
    public class Person
...
    [XmlAttribute("name")]
    public String Name
...
    [XmlElement("age")]
    public int Age
```

这种策略的好处在于类属性自身与 XML 的属性能紧密结合。这可能会添加多行代码，增加复杂度，但好处是对类的封装更加紧密。例如，本书中我们经常宣传私有属性的好处，以及应该通过定义的取值方法和赋值方法来访问这些属性。这显然是一个强硬且重要的理念，但事实上请记住取值方法和赋值方法的定义（以及签名）是由程序员决定的。总之程序员可以任意定义一些方法作为取值方法和赋值方法。而在 XML 模型中，取值方法和赋值方法是属性的属性，这是绑定属性的标准方式。

例如，当创建名为 Name 的 XML 属性时，定义如下所示：

```
[XmlAttribute("name")]
public String Name
{
    get
    {
        return this.strName;
    }
    set
    {
        if (value == null) return;
        this.strName = value;
    }
}
```

检查代码可以发现代码量比单纯定义一个属性的代码量大得多：

```
public String Name;
```

这里仍然定义属性为 String 类型，主要添加了 Name 属性，也是 XML 属性这一定义，而相应的取值方法和赋值方法是 Name 属性自身的方法。

数据验证的方式保持不变。不过这样看起来更加直观（前提是你理解了这种机制）。

设置 Name 属性的语法现在变得非常简单，如下所示：

```
this.Name = name;
```

当执行这行代码时，会调用属性的赋值方法。最终是一个操作符重载（在 C 和 C++ 中大量使用了操作符重载）。当赋值运算符（等于标志）处于 Name 属性（在左边）的上下文中，会调用取值方法。这很像一个内联的编译器指令。

使用 Person 类的 XML 版本的方式与 Java 序列化模型很相似。以下是示例代码：

```
public void Serialize()
 {
     Person[] myPeople = new Person[3];
     myPeople[0] = new Person("John Q. Public", 32, 95);
     myPeople[1] = new Person("Jacob M. Smith", 35, 67);
     myPeople[2] = new Person("Joe L. Jones", 65, 77);
     XmlSerializer mySerializer = new XmlSerializer(typeof(Person[]));
     TextWriter myWriter = new StreamWriter("person.xml");
     mySerializer.Serialize(myWriter, myPeople);
     myWriter.Close();
 }
```

主要的不同之处在于，一个序列化为私有的 Java 格式，而另一个产生的是 XML 文件。

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfPerson xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Person name="John Q. Public">
    <age>32</age>
  </Person>
  <Person name="Jacob M. Smith">
    <age>35</age>
  </Person>
  <Person name="Joe L. Jones">
    <age>65</age>
  </Person>
</ArrayOfPerson>
```

可以使用以下代码来恢复对象：

```
public void DeSerialize()
{
Person[] myRestoredPeople;
XmlSerializer mySerializer = newXmlSerializer(typeof(Person[]));
TextReader myReader = new StreamReader("person.xml");
myRestoredPeople = (Person[])mySerializer.Deserialize(myReader);
Console.WriteLine("My People restored:");
foreach (Person listPerson in myRestoredPeople)
{
Console.WriteLine(listPerson.Name + " is " + listPerson.Age
```

```
                                        + " years old.");
    }
    Console.WriteLine("Press any key to continue...");
    Console.ReadKey();
    }
```

注意我们使用 foreach 循环来迭代该数据结构。C# 示例的完整代码放置在本章的最后。

之前已经讲过，这种方式的主要好处之一就是 XML 文件可以跨语言和平台。只要语言或平台实现了 XML 接口，都可以访问 XML 文件。Java 语言当然也可以。Java 例子中的实现是一种私有方式，只是用于演示目的。程序员完全可以在 Java 中使用 XML 的方式。

12.4 写入关系型数据库

自从信息技术领域兴起之后，关系型数据可以算是最成功的工具之一。可能其他人并不这样认为，认为有很多比它伟大的工具，但关系型数据库还是影响了 IT 行业。事实上，关系型数据库依然具有相当大的能量，尽管最终可能会被其他技术替代。

关系型数据库目前兴盛的原因在于大多数商业机构都在使用它。从 Oracle 到 SQL Server 这种大型数据库，还有 Microsoft Access 小型数据库到中型数据库，关系数据库都被人们广泛使用。

尽管关系型数据库是很出色的技术，但在与对象交互时还存在一些问题。正如将由其他对象组成的一个对象写入平面文件中时遇到的问题一样，将这样的对象写入关系型数据库中的方式并不是面向对象的方式，这会产生问题。

关系型数据库建立在表的概念上。图 12-4 展示了典型的微软 Access 表关系。这种关系模型非常广泛，很多人直观认为所有数据模型都是这种方式。然而面向对象的模型不是表驱动的。图 12-4 展示了微软 Access 提供的 Northwind 关系型数据库模型。

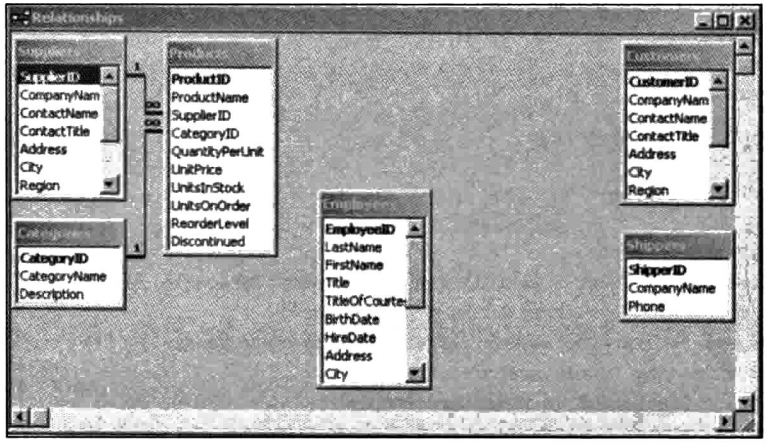


图 12-4 关系模型

由于不能将对象方便地映射为表，20 世纪 90 年代有人开发了面向对象的数据库系统。尽管这些数据库可以使用面向对象模型，而且性能可能更好，但大规模应用却遇到一个大问题，即遗留数据。

遗留数据

遗留数据是存在于各种各样的存储设备中超过 10 年的数据。本章中我们认为遗留数据是存放在关系型数据库中的历史数据。很多人不喜欢术语“遗留”是因为他们认为遗留暗示了被淘汰。事实上，重要的遗留数据并不会被淘汰，而是系统的重要部分。

由于大多数公司使用关系型数据库，如今很多商业数据都存放在关系型数据库中。这意味着对关系型数据库仍有巨大的投资。摒弃这些系统的阻碍之一是它们依然可以工作。尽管使用对象数据库写入数据可能性能更好，但将所有关系型数据转换为对象数据的开销则是不可接受的。总之，公司如果使用对象数据库，则需要将所有数据从关系型数据库中转换到对象数据库中。这有很多缺点。

第一，任何做过数据库间数据迁移的人都知道这是非常痛苦的过程。第二，即使数据转换成功了，也没有任何方式知道数据库工具的改变将如何影响应用程序代码。第三，当发生问题时（这种情况经常发生），很难确定是数据库导致的问题还是应用程序代码导致的问题。这是个噩梦，因此大多数公司都不会选择替换。所以往往在用面向对象的代码编写的全新的系统中才使用对象数据库。

然而，我们依然有以下问题：我们想编写面向对象的应用程序，但我们需要访问关系型数据库中的遗留代码。因此出现了对象 – 关系型映射。

访问关系型数据库

所有的数据库应用程序具有以下的结构：

- 数据库客户端。
- 数据库服务器端。
- 数据库。

数据库客户端是提供连接到数据库系统接口的用户应用程序。它通常是一个 GUI 应用程序，允许用户查询和更新数据库。

SQL

SQL 全称为结构化查询语言（structured query language）。它是数据库客户端与各种各样的供应商数据库系统标准的通信方式。这些数据库系统需要实现这种标准。

数据库客户端与数据库服务器通过 SQL 语句进行通信。图 12-5 显示了数据库客户端 / 服务器端模型的常用方案。

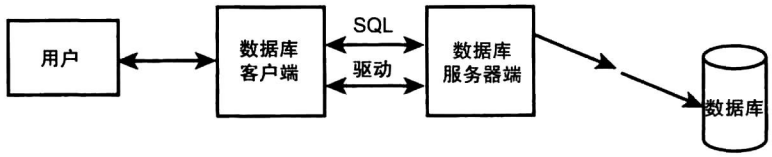


图 12-5 数据库客户端 / 服务器端模型

我们使用 Java 和 Microsoft Access 数据库进行通信来做示例。Microsoft Access 数据库是一个关系型数据库。Java 使用 JDBC（Java 数据库连接池）来与数据库服务器进行通信。

使用数据库驱动有个问题是这些驱动由具体的供应商提供。这是个常见问题。比如当购买一个新的打印机时，打印机会包含一个相应的驱动，甚至还要下载软件更新驱动。软件产品也有相似的问题。每个供应商提供了特定协议与其产品进行通信。如果你持续使用一个特定的供应商的产品，这种方案可能没什么问题。但如果想更换供应商，那么可能会带来问题。

微软开发过一个标准，叫作开放数据库连接（ODBC）。Jamie Jaworski 在《Java 2 Platform Unleashed》中写道，“ODBC 驱动抽象了特定供应商协议，为数据库客户端提供了一种通用的应用程序编程接口。在数据库客户端编写 ODBC API，可以让应用程序访问更多的数据库服务器。”请看图 12-6，该图演示了 ODBC 的工作机制。

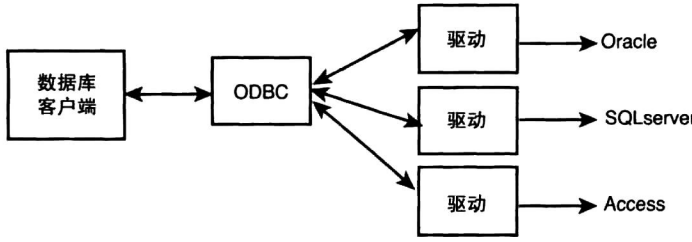


图 12-6 使用了 ODBC 的数据库客户端 / 服务器端模型

我们再来看看软件 API 定义中使用的单词*抽象*和*接口*。通过使用 ODBC，我们可以用特定的标准编写应用程序，不需要知道实现。理论上来说，我们可以针对 ODBC 标准编写代码而不关心数据库实现是 Microsoft Access 数据库还是 Oracle 数据库。至少在理论上来说如此。

在图 12-5 中，客户端使用驱动向数据库服务器发送 SQL 语句。Java 使用 JDBC 和数据库服务器通信。JDBC 有多种工作方式。JDBC 驱动可以直接连接到数据库服务器，也可以使用 ODBC 连接到数据库服务器，如图 12-7 所示。根据编写应用程序的方式，可能需要下载多个服务器和多个驱动。这些内容已经超出了本书的范围，这是通用性需要考虑的问题。你可以阅读其他进阶书籍获取更多细节，比如《Java 2 Platform Unleashed》来了解如何设置数据库并连接到应用程序。

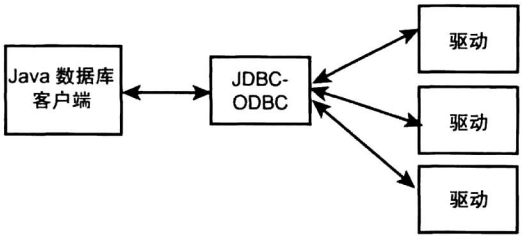


图 12-7　使用 ODBC/JDBC 的数据库客户端 / 服务器端模型

12.5　结语

本章中我们涵盖了对象持久化的概念。之前我们主要关注于面向对象基本概念，把对象当作一个实体，对象存活于创建该对象的应用程序的生命周期中。我们认为对象能够跨应用程序存活，这就需要对象持久化。

例如，应用程序可能需要使用另一个应用程序创建的对象，或者可能创建一个对象稍后被自身或者其他应用程序使用。持久化对象的方式之一是将其序列化到约定的文件中。另一种方式是使用关系型数据库。

12.6　引用

- Savitch, Walter. 2008. *Absolute Java*, 3rd ed. Boston, MA: Addison-Wesley.
- Walther, Stephen. 2008. *ASP.NET 3.5 Unleashed*. Indianapolis, IN: Sams Publishing.
- Skeet, Jon. 2008. *C# in Depth: What You Need to Master C# 2 and 3*. Greenwich, CT: Manning.
- Bob Leasure and James Leasure. 2004. *The Web Warrior Guide to Web Database Technologies*. Boston, MA: Course Technology (Cengage).
- Deitel, et al. 2003. *C# for Experienced Programmers*. Upper Saddle River, NJ: Prentice Hall.
- Deitel, et al. 2003. *Visual Basic .NET for Experienced Programmers*. Upper Saddle River, NJ: Prentice Hall.
- Jaworski, Jamie. 1999. *Java 2 Platform Unleashed*. Indianapolis, IN: Sams Publishing.
- Flanagan, David, et al. 1999.*Java Enterprise in a Nutshell*. Sebastopol, CA: O'Reilly.
- Farley, Jim. 1998. *Java Distributed Computing*. Sebastopol, CA: O'Reilly.
- Oracle: http://www.oracle.com/technetwork/java/index.html

12.7　本章中使用的示例代码

以下是 C#.NET 版本的代码。其他语言（比如 VB.NET 和 Objective-C）在出版社网站上

有电子版。本章已经展示了这些例子对应的 Java 代码。

C# .NET 版的 Person 类示例

```
// Person 类
using System;
using System.Collections;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

namespace CSSerial
{
    [XmlRoot("person")]
    public class Person
    {
        private String strName;
        private int intAge;
        private int intScore;

        public Person()
        {
            this.Name = "John Doe";
            this.Age=25;
            this.Score=50;
        }

        public Person(String name, int age, int score)
        {
            this.Name = name;
            this.Age = age;
            this.Score = score;
        }
        [XmlAttribute("name")]
        public String Name
        {
            get
            {
                return this.strName;
            }
            set
            {
                if (value == null) return;
                this.strName = value;
            }
        }

        [XmlElement("age")]
```

```
        public int Age
        {
            get
            {
                return this.intAge;
            }
            set
            {
                this.intAge = value;
            }
        }

        [XmlIgnore()]
        public int Score
        {
            get
            {
                return intScore;
            }
            set
            {
                this.intScore = value;
            }
        }
    }
}

// CSSerial 类
using System;
using System.Collections;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

namespace CSSerial
{
    class Program
    {
        static void Main(string[] args)
        {
            Program myProgram = new Program();

        }

        public Program()
        {
            Serialize();
            DeSerialize();
        }
```

```
        public void Serialize()
        {
            Person[] myPeople = new Person[3];
            myPeople[0] = new Person("John Q. Public", 32, 95);
            myPeople[1] = new Person("Jacob M. Smith", 35, 67);
            myPeople[2] = new Person("Joe L. Jones", 65, 77);
            XmlSerializer mySerializer = new XmlSerializer(typeof(Person[]));
            TextWriter myWriter = new StreamWriter("person.xml");
            mySerializer.Serialize(myWriter, myPeople);
            myWriter.Close();
        }

        public void DeSerialize()
        {
            Person[] myRestoredPeople;
            XmlSerializer mySerializer = new XmlSerializer(typeof(Person[]));
            TextReader myReader = new StreamReader("person.xml");
            myRestoredPeople = (Person[])mySerializer.Deserialize(myReader);
            Console.WriteLine("My People restored:");
            foreach (Person listPerson in myRestoredPeople)
            {
                Console.WriteLine(listPerson.Name + " is " + listPerson.Age
                + " years old.");
            }
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }    }
}
```

第 13 章 Chapter 13

Web 服务、移动应用及混合应用中的对象

在软件开发领域对象变得如此流行的主要原因可能是互联网的发展。尽管面向对象语言的历史基本上和结构化语言的历史一样长，但直到互联网出现，对象才广泛被接受。如今对象事实上成为各种网络的标准选择。从互联网到移动网络（通过网线和电话线建立的基础设施）都会借助对象来传输娱乐节目和游戏媒体之类的内容。

直到近代（自从 20 世纪 90 年代后期）对象才成为主流，而面向对象的语言 Smalltalk 在 80 年代和 90 年代就已经相当流行，而基于对象的语言 C++ 在 20 世纪 90 年代开始被广泛传播。Smalltalk 广泛支持面向对象语言特性。而 C++ 成为市场上胜出的第一门面向对象的语言。Java 初始主要目标是网络，不过作为面向对象语言在商业上相当成功。如今随着 .NET 和 Objective-C 的发展，面向对象的语言成为主流。本章介绍了一些被广泛用于互联网和其他主要网络的对象技术。

13.1 分布式计算的演进

作为专业开发人员的最酷的现实之一是改变永无止境。尽管改变会使人始终保持兴奋，而且保持整个行业永远充满朝气，但改变也增加了开销，需要维护那些伟大的但过时的技术。总之，当前的开发人员需要了解这些最新最酷的技术，而且还要与大量的遗留技术集成起来。这意味着始终需要维护各种各样的当代的以及遗留的系统。比如数十年前编写的 COBOL 代码当前仍然在很多公司的 IT 技术设施中扮演着不可或缺的角色。关系型数据库是数十年前的技术，但是现在很多系统依然与它紧密相关。分布式计算的整个历程也牵扯了很多遗留技术。

基本上可以说，电子邮件的出现揭示了“分布式计算”这一概念。本章主要关注于处于分布式机器上的应用程序间如何传输对象。分布式计算囊括了很多技术，本章会讨论以下技术：

- HTML
- EDI
- RPC
- CORBA
- DCOM
- XML
- SOAP
- Web services
- ReST

13.2 基于对象的脚本语言

本书主要关注的是面向对象的编程语言。诸如 Java、.NET 和 objective-C 之类的语言可创建完整的能够独立运行的应用程序。然而使用对象进行编程并不是这些语言的特例。我们已经说过 C++ 不是真正的面向对象的编程语言，而是基于对象的编程语言。C++ 之所以被认为是基于对象的编程语言，是因为在使用 C++ 过程中可以不遵守某些面向对象的要求。你可以借助 C++ 编译器编写非面向对象的 C 程序。有一些脚本语言也类似，比如 JavaScript、VBScript、ASP、JSP、PHP、Perl 和 Python。

总体模型

创建网页过程中引入了很多技术。编程语言、脚本语言和标记语言在该模型中都占有一席之地。尽管本书主要关注面向对象的编程语言，但是也有必要了解一下其他语言。

因此，先暂缓了解一些互联网相关技术，尽管这些技术是互联网以及 Web 服务的基本技术。我们先回顾一下客户端 / 服务器端模型。图 13-1 展示了一个经典的客户端 / 服务器端模型。

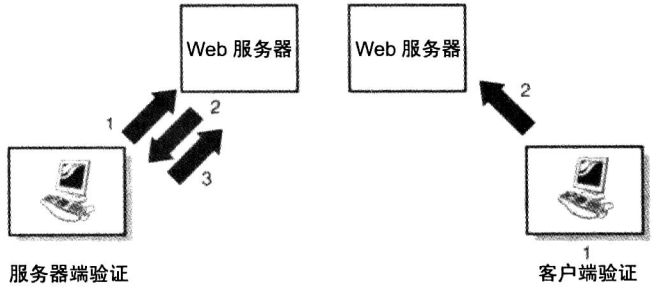


图 13-1 客户端 / 服务器端模型

理解客户端 / 服务器端模型中的两端非常重要。一端是客户端，大部分情况下指浏览器，另一端是服务器端，即物理 Web 服务器。我们可以通过一个简单的电子商务网站来帮助理解。

假设你正在创建一个简单的网页，该网页需要用户提供以下信息：

- 日期
- 名
- 姓
- 年龄

该 HTML 页面会被浏览器渲染，而浏览器可以被认为是客户端，如图 13-2 所示。

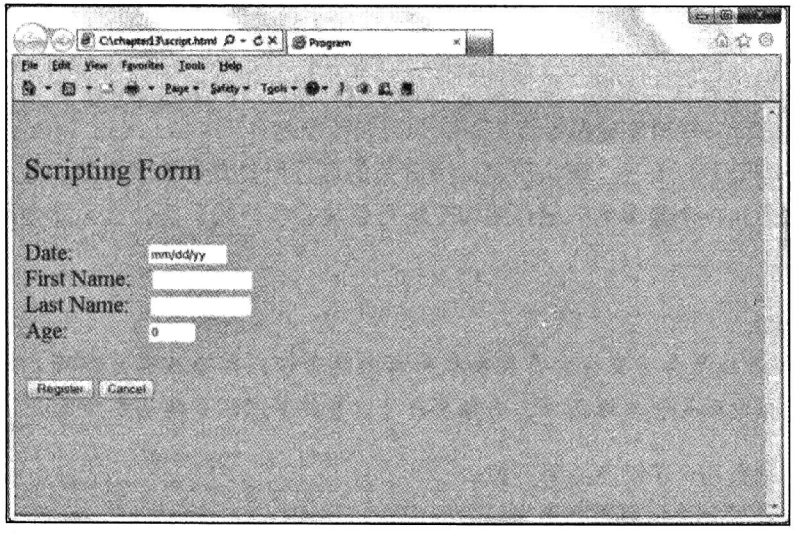


图 13-2　渲染 HTML 文档

这是一个非常简单的 HTML 文档。它也很好地演示了表单验证的概念。我们必须弄清楚的问题之一是当开发一个客户端 / 服务器端系统时，我们是在客户端做验证，还是在服务器端做验证，抑或两端都做。

例如，假设想验证用户输入的日期是否合法。同时也想让年龄处于一个合法的范围。我们当然不希望用户输入年龄为 –5。问题在于验证是放在客户端还是服务器端。接下来会阐述这个问题的重要性，以及它与对象的关系。

首先，我们解决年龄字段的问题。在大多数业务系统中，用户信息都会存储在服务器端的数据库中。基于安全原因，禁止客户端直接访问数据库。

客户端安全

由于任何人都可以使用 Web 浏览器，所以让客户端（浏览器）可以直接访问数据库是非常愚蠢的行为。因此，当客户端需要检索或更新数据库时，必须请求服务器进行操作。这是基本的安全策略。

该例子是本书中强调的接口 / 实现范式的完美示例。在本例中，客户端会请求服务器端的服务。所以软件系统会提供一个接口，客户端可以向服务器端发送信息并请求特定的服务。

图 13-2 展示了与年龄字段相关的 HTML 文档。假设一个名为 Mary 的用户想更新他的年龄到数据库中。用户打开网页后，在表单中输入了相应的信息（包括年龄文本框中的年龄）然后点击了注册按钮。在这个极其简单的场景中，表单中的信息被发送到了服务器，服务器会处理该信息并更新数据库。

日期输入框中的信息是如何被验证的？如果没有进行验证，服务器端的软件会直接接收 Mary 输入的年龄并进行更新。如果 Mary 输入的年龄不正确，错误的年龄会进入到数据库中。

如果服务器端进行了验证，服务器端的软件可以确保年龄值在合法的范围内。当然数据库自身也可能进行一些检查确保年龄在正确的限制范围内。

不过服务器端验证有一个重大限制，那就是信息必须要发送给服务器。这可能有点违反直觉，估计你会问一个简单的问题：明明验证可以放在客户端，为什么还要在服务器端再做一次？

冗余验证

注意任何 Web 应用都需要在客户端和服务器端做验证，因为在客户端有方式可以绕过客户端验证直接向服务器发送数据，或者用户可以直接禁用客户端脚本而发送非法值。

解决该问题有几个关键点需要考虑：

- 向服务器端发送信息需要更多的时间成本。
- 向服务器端发送信息会增加网络传输。
- 向服务器端发送信息会占用服务器资源。
- 向服务器端发送信息会存在潜在的错误。

基于这些原因以及其他潜在的问题，最终目标是在客户端进行尽可能多的验证。这正是脚本语言发挥作用的地方。

13.3 JavaScript 验证示例

JavaScript 和大多数脚本语言都是基于对象的。和 C++ 一样，你在编写 JavaScript 应用程序时可以不用遵守面向对象的制约。然而，JavaScript 也提供了面向对象的能力。这正是为什么脚本语言（比如 JavaScript 和 ASP .NET）在面向对象的市场上如此重要的原因。你可以在 JavaScript 应用程序中使用对象来改进网页功能。某种意义上，可以认为脚本语言是传统的编程范式和面向对象范式之间的桥梁。请记住，即使使用的技术并不是纯粹的面向对象技术，在 Web 应用程序中照样可以使用对象。

了解脚本语言之前先要了解 HTML 的限制。HTML 是一门标记语言，提供了一些功能，但本身不具备编程能力。例如，在 HTML 中无法编写一个 IF 语句或者循环。因此在早期的 HTML 中几乎没什么办法进行客户端验证。而脚本语言的出现改变了这一切。

网页开发人员可以借助于 JavaScript 和其他脚本语言在网页中加入编程逻辑。这种编程逻辑能力也包含了客户端验证。请看使用 HTML 和 JavaScript 实现的一个非常简单的验证程序。该例子中页面代码如下所示：

```
<html>
<head>
<title>Validation Program</title>
<script type = "text/javascript">
function validateNumber(tForm) {
        if (tForm.result.value != 5 ) {
                this.alert ("not 5!");
        } else {
                this.alert ("Correct. Good Job!");
    }
}
</script>
</head>
<body>
<hr>
<p>
<h1>Validate</h1>
<form name="form">
<input type="text" name="result" value="0" SIZE="2">
<input type="button" value="Validate" name="calcButton"
   onClick="validateNumber(this.form)">
</form>
<hr>
</body>
</html>
```

首先需要注意的是，JavaScript 是嵌入在 HTML 代码中。这点与使用编程语言时有所不同。Java 和 C# 之类的语言以独立的应用程序实体存在，而客户端 JavaScript 通常存活在浏览器作用域中。不过 JavaScript 文件也可以以外部代码文件的方式独立存在（比如 jQuery 库）。

Java 和 JavaScript

尽管 Java 和 JavaScript 都基于 C 语法，但它们没什么直接关系。

图 13-3 展示了该网页在浏览器中的呈现。

在该应用程序中，用户可以在文本框中输入一个数字并单击验证按钮。该应用程序会检查该值是否为 5。如果输入的值不是 5，会弹出一个警告框并指出验证错误，如图 13-4 所示。

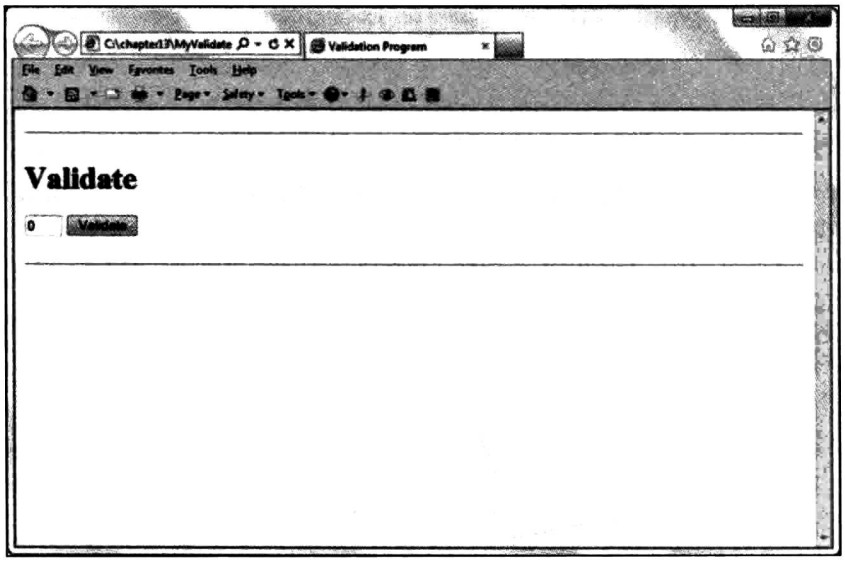


图 13-3 JavaScript 验证应用程序客户端

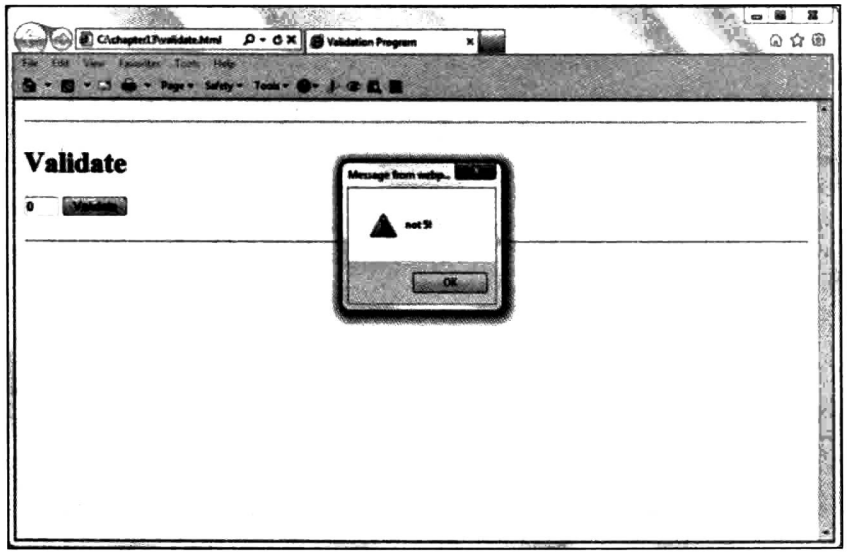


图 13-4 JavaScript 验证警告框

如果用户输入 5，那么警告框会指出该值是合法的。

执行该验证的机制基于该 JavaScript 脚本的两个部分：

- 函数定义。
- HTML 标签。

和大多数编程语言一样，可以使用 JavaScript 定义函数。在这个例子中，我们在应用程序中定义了一个名为 `validateNumber()` 的函数：

```
<script type = "text/javascript">
function validateNumber(tForm) {
        if (tForm.result.value != 5 ) {
                this.alert ("not 5!");
        } else {
                this.alert ("Correct. Good Job!");
    }
}
</script>
```

JavaScript 语法

为了契合本书主题，所以不会引入 JavaScript 的很多语法，如果想要了解 JavaScript 语法请参见其他书籍。

当单击验证按钮时会调用该函数。这些行为会被 HTML 表单定义捕获：

```
<input type="button" value="Validate" name="calcButton"
   onClick="validateNumber(this.form)">
```

当验证按钮被单击时，一个代表该表单的对象通过参数列表传递给了 `validateNumber()` 函数。

13.4　网页中的对象

在 HTML 文件中有多种方式利用对象进行网页开发。可以通过脚本语言使用对象，比如之前小节中的 JavaScript 验证示例。外部对象也可以包括在 HTML 文件中。

网页中可以引入各种外部对象。一些对象可用于播放多媒体（比如音乐和电影），另一些可以执行第三方软件创建的对象（比如 PowerPoint 和 Flash）。

接下来的小节来学习如何将对象嵌入网页中。

13.4.1　JavaScript 对象

在之前小节的 JavaScript 示例中使用到了对象。我们可以查看 `validateNumber()` 函数中的代码来验证。尽管有很多名称（比如组件、部件、控制器等）都可以描述用户接口，但它们都与对象的功能相关。

你可以使用一些对象来创建网页。以下几个元素可以认为是对象：

- 文本框
- 按钮
- 表单

每个对象都有属性和方法。你可以修改按钮的属性，比如修改按钮的颜色，也可以修改按钮上的文字。表单是一个由很多对象组成的对象。比如下面这行代码，使用的标记法是面向对象语言使用的标记法（使用句号来分割对象和对象的属性及方法）。在这行代码中，你可

以看到文本框对象（result）的 value 属性是表单对象（tForm）的一部分：

```
if (tForm.result.value != 5 )
```

另外警告框本身也是个对象。我们可以在代码中用 this 指针来验证：

```
this.alert ("Correct. Good Job!");
```

this 指针

请记住 this 指针是对当前对象的引用，在本例中当前对象是表单。

JavaScript 支持一个特定的对象体系。图 13-5 展示了该体系的一部分。

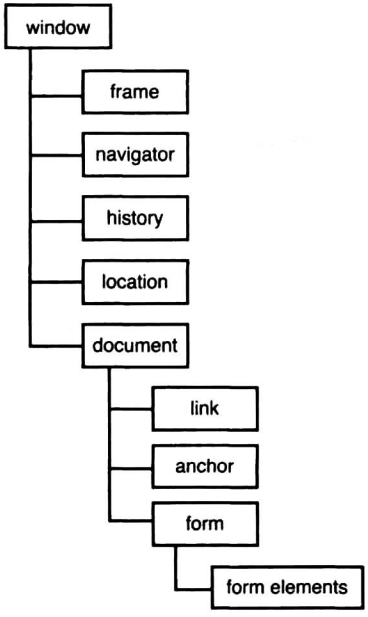


图 13-5 JavaScript 对象树

JavaScript 和其他的脚本语言一样，内置了一些对象。比如内置了 Date 类。该类的实例是包含了诸如 getHours() 和 getMinutes() 方法的对象。你也可以自定义类。以下的代码演示了对 Date 对象的使用：

```
<html>
<head>
<title>Date Object Example</title>
</head>
<body>
<script language="JavaScript" type = "text/javascript">

    days = new Array ( "Sunday", "Monday", "Tuesday",
                       "Wednesday", "Thursday", "Friday",
```

```
                    "Saturday", "Sunday");

    today=new Date

    document.write("Today is " + days[today.getDay()]);

</script>

</body>
</head>
</html>
```

注意在该例子中，我们创建了一个数组对象来保存代表一周中拥有的所有星期。我们也可以创建名为 today 的对象来保存当前日期的信息。该网页会基于个人计算机中设置的日期来展示当天是星期几。

13.4.2　网页控制器

很多对象类型可以直接内置到 HTML 文档中。网页控制器由一组预先构建的对象组成。可以使用 <object> 标签来使用这些对象。比如，我们可以将 Slider 控制器包含到一个简单的网页中。以下的 HTML 代码展示了如何使用 Slider 控制器：

```
<html>
<head>
<title>Slider</title>
</head>
<body>

<object classid="clsid:F08DF954-8592-11D1-B16A-00C0F0283628" id="Slider1" width="100"
height="50">
  <param name="BorderStyle" value="1" />
  <param name="MousePointer" value="0" />
  <param name="Enabled" value="1" />
  <param name="Min" value="0" />
  <param name="Max" value="10" />
</object>

</body>
</html>
```

当在浏览器中打开该文件时，可以看到图 13-6 中的结果。

注意这是一个真正的对象。它有属性（height 和 width），也有行为（比如 slider）。可以在 <object> 标签中使用参数设置一些属性值。

浏览器兼容性

请注意这些对象不一定兼容所有的浏览器或者所有的操作系统。这些例子中使用的是 Windows 7 系统中的 Internet Explorer 8 浏览器。

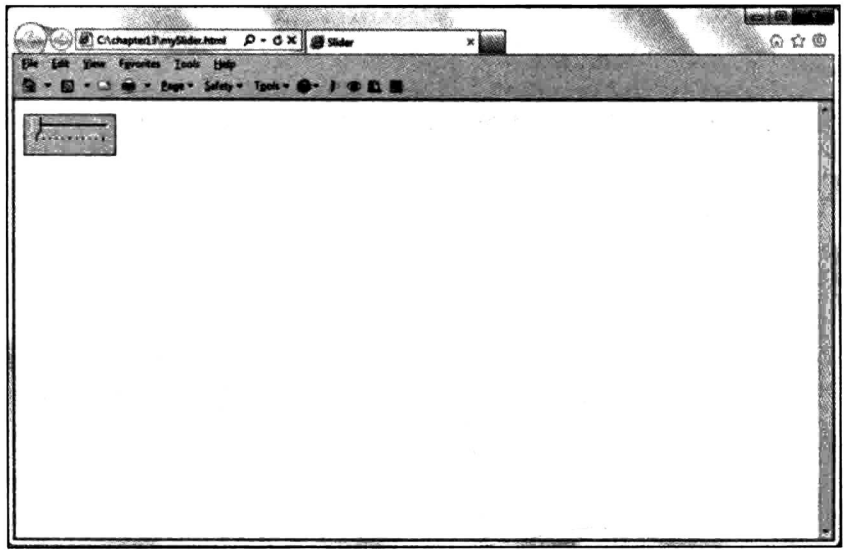


图 13-6 网页控制器

13.4.3 声音播放器

<object> 标签也可以用于在浏览器中内置和启动各种音乐播放器。启动的播放器类型取决于浏览器加载的默认播放器。

例如，以下的 HTML 代码会加载和播放 <object> 标签中指定的音频文件。在本例中，必须将音频文件放置在适当的目录中，可以通过网络访问文件：

```
<html>
<head>
<title>SoundPlayer</title>
</head>

<body>

<object
classid="clsid:22D6F312-B0F6-11D0-94AB-0080C74C7E95">
<param name="FileName" value="fanfare.wav" />
</object>

</body>
</html>
```

13.4.4 电影播放器

电影播放器也可以和声音播放器一样内置在浏览器中。以下的代码播放了 <object> 标签中指定的电影文件（.wmv）。和任何音频文件一样，必须将电影文件放置在恰当的目录中或者网络地址中：

```
<html>
<head>
<title>Slider</title>
</head>

<body>
<object
classid="clsid:22D6F312-B0F6-11D0-94AB-0080C74C7E95">
<param name="FileName" value="AspectRatio4x3.wmv" />
</object>

</body>
</html>
```

13.4.5　Flash 动画

最后一个例子是使用 <object> 标签将 Flash 动画内置到网页文档中。当然还有很多其他对象可以内置到网页中。HTML 代码如下所示：

```
<html>
<head>
<title>Slider</title>
</head>
<body>

<object width="400" height="40"
classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://download.macromedia.com
/pub/shockwave/cabs/flash/swflash.cab#4,0,0,0">
<param name="SRC" value="intro.swf">
<embed src="bookmark.swf" width="400" height="40"></embed>
</object>

</body>
</html>
```

13.5　分布式对象及企业

10 年前或者更早，术语企业计算成为信息技术词汇中的重要词汇。如今，在 IT 领域的主要开发活动都属于企业计算。但是企业计算是什么意思？

对企业计算最基本的定义是它本质上是分布式计算。分布式计算文如其名，指一组分布式的计算机通过网络一起工作。在这个上下文中，网络可以是私有网络或者互联网。

分布式计算的力量在于计算机可以共享网络。在一个真正的分布式环境中，你根本不会知道哪台计算机在真正服务你的请求。事实上你也不需要知道。例如，当在线购物时，你会连接到一个公司的网站。你只需知道你使用一个 URL 进行连接。然而，该公司会把你连接到一台在线的物理机器上。

为什么需要这样呢？假设公司只有一台机器来服务所有的请求。那么如果该机器崩溃了，将会发生什么？假设该公司可以将在线请求分发到 12 台机器中。如果其中一个机器宕机了，那么不会造成毁灭性的影响。

另外可以思考从网站上下载文件这一场景。你很可能遇到过这样的情形，下载网站提供给了你一组网站，让你选择一个物理位置离你最近的网站来下载。这意味着可以通过网络分发负载。计算机网络自身可以进行负载均衡。图 13-7 提供了一个图表，展示了一个分布式系统。

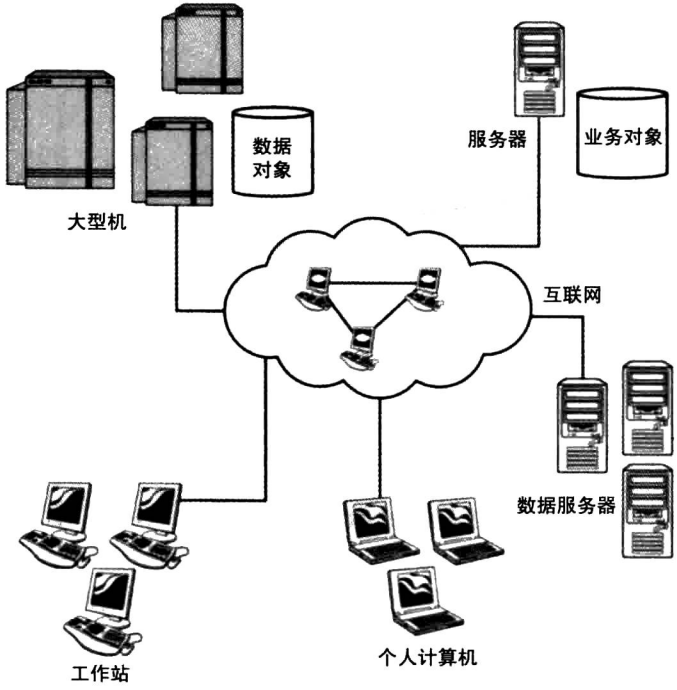


图 13-7　分布式计算

本书主要关注对象和面向对象概念。有一个非常有意思的概念叫作分布式对象。由于这些对象完全自给自足，所以很适合于分布式应用程序。本章重点想说明：*如果应用程序（客户端）请求一些对象（服务器端）的服务，那么该网络服务可以存在于网络上的任何地方，而且无需关心谁在调用它。*我们来进一步探索分布式对象的一些（以前的和当前的）相关技术。

请记住其中很多技术并不优美，但仍然作为遗留系统而存在。例如，在 20 世纪 90 年代后期出版本书第 1 版时，CORBA 的情形与当今大不相同。如今的购物车程序都是使用一组网络服务来实现，可以使用网站内置的格式直接进行维护，而不会使用 CORBA 接口。SOAP 和 XML 以及 SOA 架构如今比比皆是。然而了解一些历史有助于理解技术的进化历程。

13.5.1　公共对象请求代理体系结构

本书反复强调的一个重要原则是对象是完全自给自足的单元。通过网络发送对象并不需要魔法。事实上，本书中很多例子都演示了在网络中传输对象是多么的简单。

企业系统都是基于分布式对象构建的。使用分布式对象有很多优势。最大优势是理论上系统可以调用处于网络任何位置的对象。这是一种非常强大的能力，而且是当今基于互联网的业务的基石。另一个主要优势是可以通过网络中的多台服务器分发系统服务。

通过网络访问和调用对象是一个非常强大的技术。不过有一个明显缺点，那就是可移植性问题。虽然我们可以创建私有的分布式网络，事实上私有网络具有明显的限制。另一个可移植性问题与编程语言有关。比如一个使用 Java 编写的系统可能需要调用 C++ 编写的对象。理想情况下，我们需要在分布式环境中为对象创建一个非私有的语言无关的框架。这就是 20 世纪 90 年代中期互联网兴起之前，CORBA（公共对象请求代理体系结构）对开发社区带来的主要贡献。

CORBA 的主要好处是：CORBA 允许程序使用标准的协议在不同的供应商之间进行通信。这种协同能力包括硬件和软件。因此供应商可以在多个平台和操作系统上使用多种语言编写应用程序，而且可以操作不同的供应商网络。

CORBA 技术和 DCOM 都是各种计算机软件应用程序的中间件。CORBA 代表了其中一种中间件（稍后我们会看到一些其他实现，比如 Java 的 RMI）。不管中间件采用了什么方式，其背后的概念是一致的。这些中间件提供服务使得应用程序可以通过网络进行交互。这些系统通常被称为*多层系统*。例如图 13-8 展示了一个三层系统。在这个例子中，表现层与数据层被处于中间的应用层隔离开来（中间件往往会引入关系型对象映射系统）。这些组件可以运行在一个或多个机器上。这正是术语*分布式*的由来。这些组件（即对象）会通过网络进行分发。网络可以是私有网络，也可以是互联网。

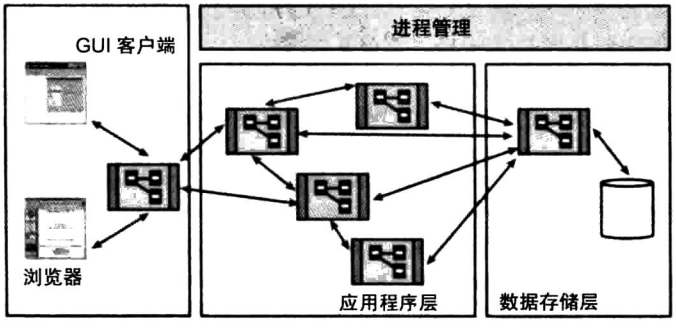


图 13-8　三层系统

这就是对象在分布式计算中所起的作用。OMG 指出：“CORBA 应用程序由对象组成”。所以你可以说对象是分布式计算世界的主体。OMG 还说这些对象“是运行软件的独立单元，它组合了功能和数据，往往（不一定总是）对应现实世界中的某些对象”。

这样的系统中最著名的例子之一是购物车系统。我们可以使用之前讨论过的购物车例子来解释对象的实例化。当你访问电子商务网站来购买商品时，你会被分配一个独立的购物车。因此，每位顾客有一个独一无二的购物车。也就是说，每个顾客有一个对象，这个对象包含了购物车对象的所有属性和行为。

尽管每个顾客对象都有相同的属性和行为，但每个客户拥有不同的属性值，比如名称、地址等。可以通过网络将该购物车对象发送到任何地方。系统中的其他对象代表商品、仓库等。

包装对象

本书之前解释过，对象的一个常用之处是作为包装对象。如今很多应用程序都是遗留系统。修改这些遗留应用程序往往既不切实际又开销巨大。将遗留的应用程序和最新的分布式系统集成起来的优雅的方式之一是创建一个包装对象作为遗留系统的接口。

使用 CORBA 实现一个诸如购物车应用程序的系统，拥有一个优势就是不同语言编写的服务都可以访问 CORBA 中的对象。为了实现这一点，CORBA 定义了一个接口，所有语言都必须遵守该接口。CORBA 的接口概念就是第 8 章中讨论的契约。CORBA 接口被称为接口定义语言（IDL）。客户端和服务器端为了支持 CORBA，都必须遵守 IDL 声明的契约。

本书之前讨论过另一个概念，封送（marshaling）。封送是指将对象分解为一种可以通过网络发送的格式，然后在另一端再重新组装起来。因此如果客户端和服务器端都遵守 IDL，那么对象就可以通过网络进行封送，这样可以做到编程语言无关。

一个叫作对象请求代理（ORB）的应用程序会路由所有通过 CORBA 系统进行传输的对象。显而易见，首字母缩略词 ORB 实际上是缩略词 CORBA 的一部分。ORB 将所有对象传入到 CORBA 应用程序中。ORB 负责从客户端向服务器端路由请求，同时也从对应的目的地获取返回的响应。

CORBA 紧紧契合了分布式计算这一概念。OMG 指出

> CORBA 的本质是使用 OMG IDL 分离实现和接口。

进一步讲，

> 客户端只能通过公布的接口访问对象，只能调用对象通过其 IDL 接口暴露出来的操作（输入和输出），调用中只能包括要求的参数。

我们通过第 8 章中的电子商务案例来了解 IDL 的工作方式。根据图 8-7 创建一个 Shop 类的子类。Inventory 接口的代码如下：

```
interface Inventory  {
     string[]  getInventory ();
     string[]  buyInventory (in string product);
}
```

这个例子在接口中定义了如何展示和购买货物。该接口会被编译为两个实体：

- 一个桩（stub），用来连接客户端和 ORB。
- 一个骨架类（skeleton），表示 ORB 和对象之间的连接。

这些 IDL 桩和骨架类代表了所有交互对象必须遵循的契约。图 13-9 展示了这些 CORBA 组件相互交互的方式。

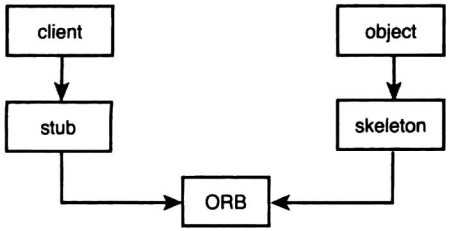


图 13-9　CORBA 组件

有趣的是，当客户端请求对象的服务时，它并不需要知道它正在请求的对象的任何细节，甚至无需知道对象在哪里。客户端只需简单调用它想要的对象（以及服务）即可。对于客户端来说，如果调用是本地调用，那么会调用本地系统中的对象。而该调用会传递给 ORB。如果 ORB 确定需要的对象是一个远程对象，ORB 则会路由该请求。只要一切工作正常，客户端无需关心服务它的实际对象在哪里。图 13-10 展示了 ORB 如何通过网络路由工作。

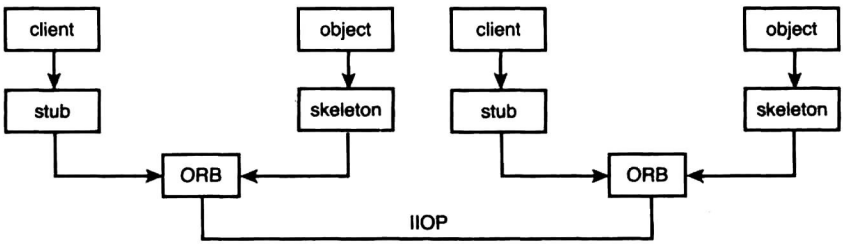


图 13-10　ORB 路由

互联网 Inter-ORB 协议

HTTP 是网页传输的协议，互联网 Inter-ORB 协议（IIOP）是关于分布式对象的协议，可以使用各种各样的编程语言编写这些对象。IIOP 是 CORBA 和 Java RMI 等标准的基础。

本节我们讲述了关于分布式计算的原始的、基础的主题，这些分布式计算接住了 CORBA、DCOM 和 RMI 等开创性技术。接下来的小节我们会继续讨论一些最新的技术，即 Web 服务。如今更普遍使用 SOAP 和 XML 以及 SOA 架构。

13.5.2　Web 服务的定义

过去几年，Web 服务演进非常迅速。事实上，当本书第 1 版发布时，当前流行的技术大多数才刚刚萌芽。我们使用 W3C 提供的对 Web 服务的通用定义，即“客户端和服务器端使用基于 SOAP（simple object access protocol，简单对象访问协议）标准的 XML 消息进行通信”。

SOAP 是一项通信协议，用于通过互联网发送消息。SOAP 理论上是平台以及语言无关

的，它基于 XML。SOAP 使用 HTTP 协议在应用程序之间进行通信，因为客户端应用程序往往是浏览器。SOAP 扩展了 HTTP 的功能，从而提供更具功能性的 Web 服务。

在分布式计算发展的早期，远程程序调用（RPC）已经兴起。使用 SOAP 的主要动机是通过 HTTP 使用 XML 进行远程程序调用。我们可以描述 SOAP 为：SOAP 是基于 XML 的用于分布式应用程序的协议。

RPC

远程程序调用（RPC）是一个通信机制，允许通过共享的网络调用其他计算机上的服务（对象）。如果系统确实是面向对象的，那么 RPC 也可以被称为远程方法调用（RMI）。该方法调用是平滑的，因为开发人员绝不知道（或者不需要知道）服务是本地的还是远程的。因此，往往向开发人员隐藏了实际的方法调用过程的实现细节。RPC 策略通常是私有的。

CORBA 和 DCOM 之类的技术的主要缺点是它们基本上都是私有的，而且拥有自定义的二进制格式。SOAP 是基于文本的，由 XML 编写，比起 CORBA 和 DCOM 使用起来更加简单。这与第 12 章的 12.3 节中指出的优势一样。

实际上，CORBA 和 DCOM 系统为了尽可能地平滑工作，它们必须与相似的系统进行通信。这对当前的技术环境有重大限制，因为你不知道网线的另一边是什么。因此，SOAP 的最大优势就是大多数软件公司都遵循 SOAP 标准。

本书已经强调过多次，对象技术的优势是可以使用包装对象。SOAP 可以作为一种包装对象，它可以将 DCOM、企业 JavaBeans 或者 CORBA 之类的技术产品包装起来，以便能够在互联网上更方便地使用这些产品。即使企业中存在各种不同技术，也可以借助这种包装能力建立一个标准的网络通信方式。

不管如何描述 SOAP，SOAP 和 HTML 一样都是无状态的单向消息系统。因此 SOAP 并不是 DCOM、企业 JavaBeans、CORBA 以及 RMI 等技术的完全替代品，它与这些技术是互补的关系。

为了契合本书主题，接下来主要讲解 SOAP 概念，而不会引入具体的 SOAP 技术、编码等内容。

SOAP

本章中的例子展示了分布式系统中对象的流动图。出版社的网站中有一个完整的示例应用程序，限于篇幅原因本书不会详细讲述。

我们创建一个仓库应用程序作为示例。该应用程序利用浏览器作为客户端，然后借助一组 Web 服务使用互联网上的仓库系统来处理业务。

这个 SOAP 示例使用的模型如下。图 13-11 提供了该系统的可视化图形。

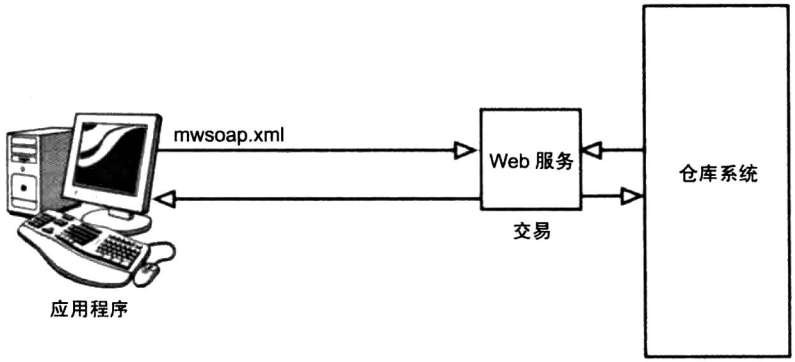


图 13-11　SOAP 示例

mwsoap.xml 文件描述了 Web 服务中处理各种交易的 XML 结构。Invoice.xsd 的描述如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="http://ootp.org/invoice.xsd"
elementFormDefault="qualified" xmlns="http://ootp.org/invoice.xsd"
xmlns:mstns="http://ootp.org/invoice.xsd" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Invoice">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Address" minOccurs="1">
          <xs:complexType>
            <xs:sequence />
            <xs:attribute name="Street" type="xs:string" />
            <xs:attribute name="City" type="xs:string" />
            <xs:attribute name="State" type="xs:string" />
            <xs:attribute name="Zip" type="xs:int" />
            <xs:attribute name="Country" type="xs:string" />
          </xs:complexType>
        </xs:element>
        <xs:element name="Package">
          <xs:complexType>
            <xs:sequence />
            <xs:attribute name="Description" type="xs:string" />
            <xs:attribute name="Weight" type="xs:short" />
            <xs:attribute name="Priority" type="xs:boolean" />
            <xs:attribute name="Insured" type="xs:boolean" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Invoice.xsd 文件描述了 invoice 的构造，以及应用程序应该如何遵循它的定义。该

文件实际上是一种范式，和数据库中使用的范式作用一致。注意在 Invoice.xsd 文件中，invoice 由 Address 和 Package 组 成。 而 Address 和 Package 由 Description、Weight 等属性组成。最后声明这些属性为具体的数据类型，比如字符串、短整数等。图 13-12 展示了图形化的关系图。

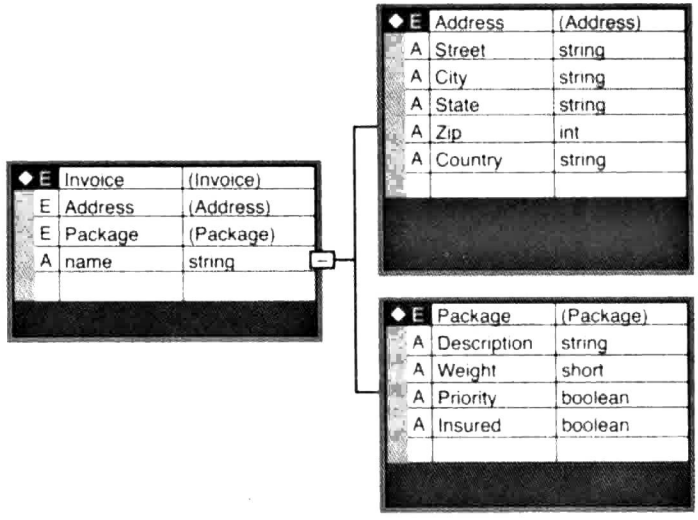


图 13-12 Invoice.xsx（范式的可视化展示）

在该例子中，Invoice.xsd 文件描述了数据的结构，而 mwsoap.xml 文件描述了真正的数据。使用 C# .NET、VB .NET、ASP .NET 或者 Java 之类的语言编写的应用程序使用 Invoice.xsd 文件来构造有效的 XML 文件，然后通过网络发送给其他应用程序。这些应用程序会使用相同的 Invoice.xsd 文件来解析并使用 mwsoap.xml 文件。你可以认为 Invoice.xsd 文件是一组契约，这和第 8 章讲述的契约概念一致。

以下代码是一个包含了具体数据的 mwsoap.xml 文件，其数据嵌入在 SOAP/XML 格式中；

```
<?xml version="1.0" encoding="utf-8"?>
<soap:envelope xmlns:soap="http://www.w3.org/2001/06/soap-envelope">
  <soap:Header>
    <mySOAPHeader:transaction xmlns:mySOAPHeader="soap-transaction"
soap:mustUnderstand="true">
      <headerId>8675309</headerId>
    </mySOAPHeader:transaction>
  </soap:Header>
  <soap:Body>
    <mySOAPBody xmlns="http://ootp.org/Invoice.xsd">
      <invoice name="Jenny Smith">
        <address street="475 Oak Lane"
                 city="Somewheresville"
                 state="Nebraska"
                 zip="23654"
```

```
                country="USA"/>
        <package description="22 inch Plasma Monitor"
                weight="22"
                priority="false"
                insured="true" />
      </invoice>
    </mySOAPBody>
  </soap:Body>
</soap:envelope>
```

13.5.3　Web 服务代码

该模型剩余部分是应用程序自身的代码。C# .NET 中有三个类，分别对应 Invoice、Address 和 Package。

请注意可以用任何语言实现该应用程序。这正是 SOAP/XML 方式的强大之处。每个应用程序都能够解析 XML 文件，这基本上是使用 SOAP/XML 的唯一要求，如图 13-13 所示。如何使用提取出来的数据由应用程序自身决定。

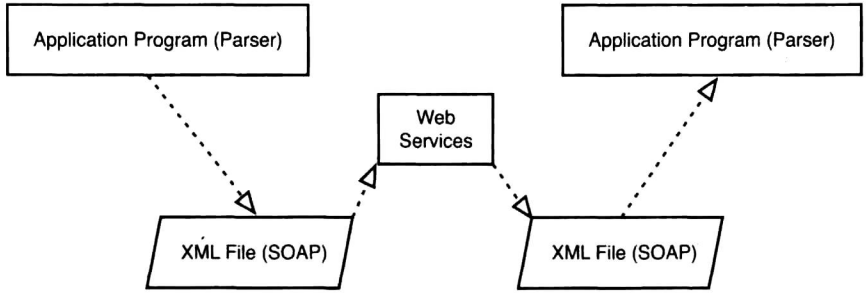


图 13-13　解析 SOAP/XML 文件

这种方式可以与具体语言或具体平台无关。理论上任何语言可以进行解析操作，这也是应用 SOAP/XML 方式的唯一要求。

作为开发人员，直接查看代码可能更有效。接下来小节中展示的 C# .NET 代码展示了图 13-12 中描述的系统的实现细节。对应的 VB .NET 的代码可以在出版社网站中找到。

Invoice.cs

接下来的代码展示了使用图 13-12 中 C# .NET 实现的 Invoice 类：

```
using System;
using System.Data;
using System.Configuration;
using System.Xml;
using System.Xml.Serialization;
namespace WebServices
{
    [XmlRoot("invoice")]
```

```
    public class Invoice
    {
        public Invoice(String name, Address address, ShippingPackage package)
        {
            this.Name = name;
            this.Address = address;
            this.Package = package;
        }

        private String strName;
        [XmlAttribute("name")]
        public String Name
        {
            get { return strName; }
            set { strName = value; }
        }

        private Address objAddress;
        [XmlElement("address")]
        public Address Address
        {
            get { return objAddress; }
            set { objAddress = value; }
        }

        private ShippingPackage objPackage;
        [XmlElement("package")]
        public ShippingPackage Package
        {
            get { return objPackage; }
            set { objPackage = value; }
        }
    }
}
```

13.5.4 表征状态转移

自从诞生了第一个计算机程序后，开发人员一直在复杂和简单的架构之间反复游走。开发人员天生喜欢简单直接的方式，而很多业务经理则更喜欢更具结构性的经过证实的实践。我们可以看到人们经常争论使用 Java 和 .NET 之类的编程语言还是使用 Perl 和 PHP 之类的语言，以及是使用 XML 还是 JSON。当然也包括 Web 服务的争论。很多开发人员认为诸如 CORBA、RPC 和 SOAP 等技术过于复杂。因此一种更加“简单”的方式变得越来越流行。这种方式叫作表征状态转移，也叫 ReST。

例如，实现 SOAP 协议需要一系列不同的工具（还要解决与老版本的 SOAP 标准的兼容问题），而 ReST 只需要支持 HTTP 即可，而且能够与其他技术（包括 SOAP）协同工作。

ReST 是一种无状态的协议，基本上依赖于 HTTP。由于 HTTP 是互联网自身的基石，很

大程度上可以说互联网的架构基于 ReST，这通常称为 RESTful 架构。

ReST 的本质特性是使用 HTTP 请求来创建、更新、读取及删除数据。ReST 可以与很多技术集成，比如 RPC 或 Web 服务，这是因为它提供了所有必要的底层功能。简单来说，ReST 是 Web 服务的轻量级方式。不过它不像 XML 一样是一种标准（有时描述 ReST 为“一种设计风格”）。图 13-14 演示了典型的 HTTP 请求 / 响应数据流的 RESTful 实现。

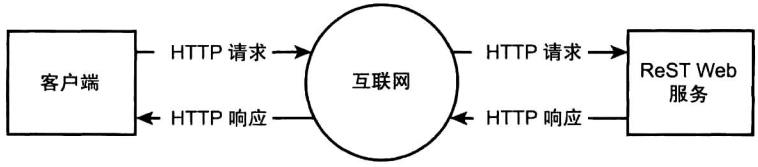


图 13-14　RESTful Web 服务

事实上，尽管其他的策略（比如 SOAP）可能更适合处理一些具体场景，而 RESTful 应用程序只用返回简单的文本或者 XML 文件。SOAP 与 XML 紧密绑定，而 ReST 则不是这样，这完全由开发人员决定。ReST 请求主要是简单的文本参数，基本上不会有复杂的东西。ReST 另一个有趣之处是，它不是天然面向对象的，因为它并不会处理对象，不过应用程序的解析器可以创建对象。

13.6　结语

在本章中，我们讲述了几种在 Web 应用程序间使用对象的技术。嵌入到网页的对象（比如 JavaScript）和分布式系统中的对象是有区别的，请注意它们的关联关系。

过去几年分布式对象发展非常快。如今实现分布式对象有多种技术选择。然而 SOAP 和 XML 的结合使得分布式系统的设计更趋于标准化。

13.7　引用

- Savitch, Walter. 2008. *Absolute Java*, 3rd ed. Boston MA: Addison-Wesley.
- Walther, Stephen. 2008. *ASP.NET 3.5 Unleashed*. Indianapolis, IN: Sams Publishing.
- Skeet, Jon. 2008. *C# in Depth: What You Need to Master C# 2 and 3*. Greenwich, CT: Manning.
- Deitel, et al. 2003. *C# for Experienced Programmers*. Upper Saddle River, NJ: Prentice Hall.
- Deitel, et al. 2003. *Visual Basic .NET for Experienced Programmers*. Upper Saddle River, NJ: Prentice Hall.
- Conallen, Jim. 2000. *Building Web Applications with UML*. Boston, MA: Addison-Wesley.
- Jaworski, Jamie. 1999. *Java 2 Platform Unleashed*. Indianapolis, IN: Sams Publishing.

Chapter 14

第 14 章

对象及客户端 / 服务器端应用程序

第 13 章中讲述了与 Web 服务相关的分布式对象的概念。在第 13 章中，我们描述互联网是对象飞驰的高速公路。本章更关注细节，探索了通过客户端 / 服务器端网络发送对象这一主题。

本章的很多概念与第 13 章紧密相关（因为这两章都是关于通过网络传输数据的）。如果只是为了学习目的，直接钻研客户端 / 服务器端模型内部工作机制可以快速积累学习经验。总之本章会编写一个功能完备的客户端 / 服务器端应用程序。

设置一个小型的 Web 服务的教学模型可能过于复杂，我们可以将客户端 / 服务器端应用程序的这个教学示例放置在单个机器上。在分布式网络中，对象的传输不会遵循特定的路径，而在客户端 / 服务器端中的对象至少从概念层面来说属于一个点到点的传输。

14.1 客户端 / 服务器端方式

正如在前几章中看到的一样，XML 对如今使用的开发技术有着重大影响。例如，分布式对象模型可以构建在一个私有系统中，也可以基于 SOAP/XML 等技术实现一种非私有的方式。

而客户端 / 服务器端模型也是一样。这样的应用程序可以完全构建于私有系统中，或者使用 XML。这两种方式本章都会介绍。我们先使用 Java 实现一个私有的方式，该程序只能执行在 Java 环境中（使用 Java 也能提供非私有的解决方案）。然后我们使用 C# .NET 来演示基于 XML 的方式实现的客户端 / 服务器端模型。

14.2　私有方式

本例中，Java 用于演示如何通过网络实现一个端到端的直连。我在之前的教学生涯中多次使用了此示例。这个例子可以通过客户端向服务器发送一个对象，然后打印出该对象中包含的一些信息。

这种方式的基本流程如图 14-1 所示。

在该设计流程中，客户端创建了一个对象然后发送给了服务器。而服务器创建了该对象的一个引用，从而可以访问该对象。服务器然后更新该对象的属性并且返回给客户端。

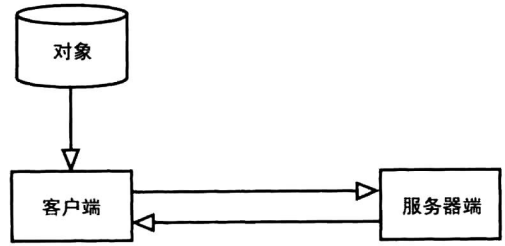


图 14-1　基本的客户端 / 服务器端流程

14.2.1　序列化对象代码

先创建一个简单的 TextMessage 类，其包含属性 name 和 message。该类也包含了一个构造函数，以及取值方法和赋值方法。完整的 TextMessage 类如下所示：

```
import java.io.*;
import java.util.*;

public class TextMessage implements Serializable {

        public String name;
        public String message;

        // TextMessage 类的构造函数
      TextMessage(String n) {
              message = " ";
              name= n;
      }

        // 对象的取值方法
      public String getName() {
              return name;
      }

        // 对象的取值方法
      public String getTextMessage() {
              return message;
      }

       // 对象的赋值方法
      public void setTextMessage(String inTextMessage) {
              message = inTextMessage;
      }
}
```

这是一个非常简单的类。该构造函数通过参数初始化了 name 属性，并设置 message 为空字符串。请注意该类可以被序列化为一个私有的 Java 二进制格式文件。

14.2.2 客户端代码

客户端代码使用 TextMessage 类创建一个对象并将其发送给服务器端，并获取服务器端的处理结果。客户端必须执行以下任务：

- 获取用户信息。
- 创建对象。
- 设置属性。
- 创建套接字连接。
- 创建输出流。
- 输出对象。
- 关闭流 。

以下展示了客户端的代码。该段代码具有详尽的注释：

```
import java.io.*;
import java.net.*;

/*
 * TextMessage 的客户端类
 */
public class Client {

    public static void main(String[] arg) {

      try {

        String message = " ";
        String name = " ";

        System.out.print("Please enter name: ");
        name = getString();

        // 创建 TextMessage 对象
        TextMessage myTextMessage = new TextMessage(name);

        System.out.print("message: ");

        message = getString();

        // 使用赋值方法对 TextMessage 赋值
        myTextMessage.setTextMessage(message);
```

```
        // 创建套接字连接
        Socket socketToServer = new Socket("127.0.0.1", 11111);

        // 创建对象 Object Output Stream
            ObjectOutputStream myOutputStream = new
                ObjectOutputStream(socketToServer.getOutputStream());

        // 将 myTextMessage 对象写入到 OutputStream 中
        myOutputStream.writeObject(myTextMessage);

        // 关闭流
        myOutputStream.close();

      } catch (Exception e) {System.out.println(e);}
    }

    public static String getString() throws Exception {

        // 等待键盘输入（调用 'stdin'）
        BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in), 1);

        String s1 = stdin.readLine();

        return (s1);

    }
}
```

最重要的部分在于客户端代码需要解决网络连接问题。以下代码定义了客户端如何连接到服务器：

```
Socket socketToServer = new Socket("127.0.0.1", 11111);
```

创建套接字时需要两个参数，第一个是 IP 地址，另一个是客户端试图连接的虚拟套接字端口。

IP 地址 127.0.0.1 是个回环地址，意味着客户端试图连接一个本地服务器。总之，客户端和服务器端运行在同一个机器上，先决条件是必须先启动服务器。

使用回环 IP 地址在测试应用程序时相当有用。因为无需请求网络连接，可以在本地测试应用程序的底层逻辑。这样更容易做初级测试。之后的测试可以使用真正的 IP 地址。

在参数列表中除了指定 IP 地址，还需要指定虚拟端口。本例中随机选择了 11111。唯一条件是客户端试图连接的服务器必须监听此端口。

客户端建立起了与服务器的连接之后，会发送对象并收到返回的对象，接下来客户端应用程序结束运行。

这段代码另一个需要注意的地方是该类最后的一个方法中，要求从键盘接收一行字符。

这是捕获用户输入，类似于在手机中输入短信。

14.2.3 服务器端代码

在网线的另一端，服务器会执行以下任务：

- 创建对象引用。
- 监听虚拟端口 11111。
- 等待客户端连接。
- 创建输入 / 输出流。
- 读取 TextMessage 对象。
- 打印信息。

服务器端的代码如下所示：

```
import java.io.*;
import java.net.*;

/*
 * TextMessage 的 Server 类
 */

public class Server {

    public static void main(String[] arg) {

        // 创建来自客户端的对象的引用
        TextMessage myTextMessage = null;

        try {

            // 启动服务并监听端口 11111
            ServerSocket myServerSocket = new ServerSocket(11111);

            System.out.println("Ready\n");

            // 等待直到有客户端试图连接
            Socket incoming = myServerSocket.accept();

            // 创建 ObjectInputStream 对象
            ObjectInputStream myInputStream = new
                ObjectInputStream(incoming.getInputStream());

            // 通过套接字读取客户端传过来的对象
            myTextMessage = (TextMessage)myInputStream.readObject();

            System.out.println(myTextMessage.getName() + " : "
                + myTextMessage.getTextMessage()+ "\n");
```

```
            // 关闭流
            myInputStream.close();

        } catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

服务器端代码和客户端代码都没有使用循环。可以简单引入循环，从而使服务器端持续监听该端口。但该功能与我们讨论的主题关联不大。

服务端完全可以更新该对象并发送回客户端。服务器端可以创建一个输出流并将对象发送回客户端，而客户端也可以创建一个输入流来读取服务器端返回的对象。

14.2.4　运行该私有的客户端 / 服务器端示例

出于简化目的，我们使用命令行终端来运行该客户端 / 服务器端示例，无需创建一个 GUI 通过集成开发环境（IDE）来运行它。下一个小节会创建一个可以通过 IDE 运行的带 GUI 的程序模块。

第一步是启动服务器，然后从另一个命令行终端启动客户端。服务器打印出一条消息说明它已经准备好了，然后会等待输入。客户端启动之后会请求用户名和消息内容，用户必须输入这些信息。

当在客户端输入消息并发送后，服务器端会展示从客户端收到的消息。图 14-2 展示了服务器端会话，图 14-3 展示了客户端会话。服务器端和客户端都包含循环来允许多次输入。该例子出于演示目的会尽量保持简单。

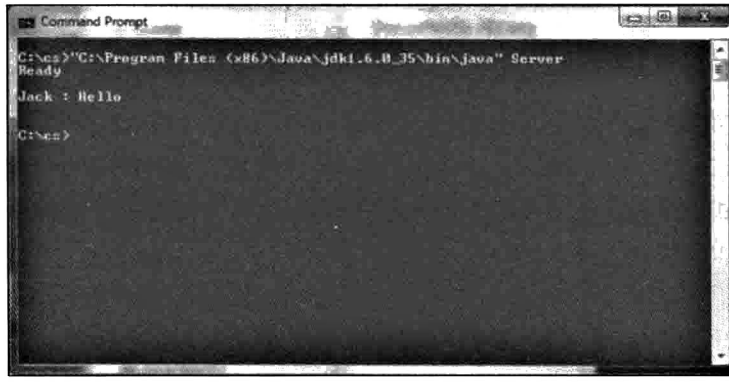


图 14-2　运行服务器

客户端简陋的用户界面要求用户输入用户名以及想发送的信息。在现实世界的短信系统中（比如手机中），服务器会基于用户输入的地址（电话号码）将消息发送给另一个人，而不只是简单打印出消息。

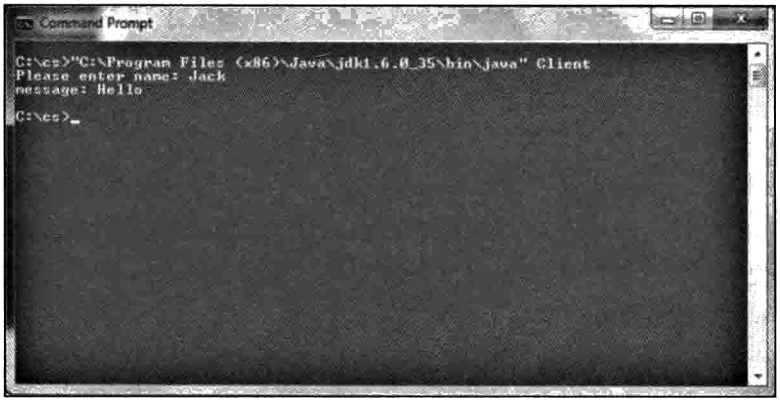


图 14-3 运行客户端

批处理文件

我喜欢创建批处理文件，在里面指定精确的 Java 类路径来帮助测试这种用于教学目的简单示例。这样不仅可以免除在命令行终端输入一大串命令字符，而且支持在一台机器上使用不同版本的 Java 来运行程序。而且批处理文件可以和其他测试技术结合起来。例如，你可以将以下代码放置到名为 server.bat 的批处理文件中，然后在命令行终端输入 server 来启动服务器：

```
"C:\Program Files (x86)\Java\jdk1.6.0_35\bin\java" Server
```

也可以使用这种方式进行编译：

```
"C:\Program Files (x86)\Java\jdk1.6.0_35\bin\javac" Client.java
"C:\Program Files (x86)\Java\jdk1.6.0_35\bin\javac" Server.java
```

14.3 非私有方式

以上例子使用私有的方式实现了一个客户端 / 服务器端模型。就像使用 XML 来处理数据持久化和分布式对象一样，我们可以借助 XML 技术来创建一个非私有的版本。

使用 XML 从理论上来说可以在不同语言以及不同平台的应用程序之间传输对象。请看图 14-4 中更新后的模型。

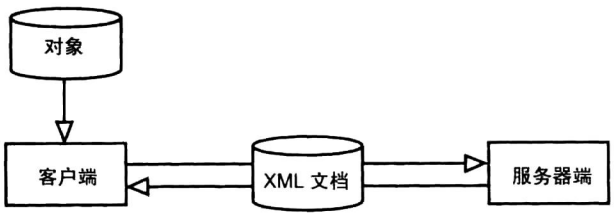


图 14-4 使用 XML 方式进行客户端 / 服务器端通信

虽然私有和非私有的方式很多底层的概念都是相同的，但在私有方式中通过一种私有的二进制格式解耦和重新构造对象，而非私有方式中则使用的是 XML 格式。

我们这次基于 CheckingAccount 类实现一个示例。

14.3.1　对象定义代码

在创建 CheckingAccount 类的 XML 版本时，查看代码可以发现该对象的 XML 定义直接内嵌在类自身中。（请查看第 11 章中对这种方式的讲解。）C# .NET 版本的 CheckingAccount 类代码如下所示：

```
using System;
using System.Collections;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

namespace Server
{
    [XmlRoot("account")]
    public class CheckingAccount
    {
        private String _Name;
        private int _AccountNumber;

        [XmlElement("name")]
        public String Name
        {
            get { return _Name; }
            set { _Name = value; }
        }

        [XmlElement("account_num")]
        public int AccountNumber
        {
            get { return _AccountNumber; }
            set { _AccountNumber = value; }
        }

        public CheckingAccount()
        {
            _Name = "John Doe";
            _AccountNumber = 54321;
            Console.WriteLine("Creating Checking Account!")
        }

    }
}
```

有关该类定义代码的有趣之处在于，该类包含了需要的属性和方法，而属性也包含了对应的 XML 定义中的属性。

总之，在 .NET 例子（C# 和 VB）中，类的创建与 XML 的定义是绑定在一起的。这种方式也可以应用在 Java 中。事实上，这种定义 XML 的方式不会依赖具体的语言或平台。这正是非私有方式的强大之处。

在这个 C# .NET 例子中，请注意我们为该项目创建了一个命名空间。

14.3.2 客户端代码

在这个例子中，客户端执行了以下任务：

- 创建 checkingAccount 对象。
- 创建套接字。
- 序列化对象到 XML 中。
- 创建流。
- 序列化对象到流中。
- 关闭资源。
- 关闭流。

注释主要用来解释程序执行流程。C# .NET 的客户端代码如下所示：

```
using System;
using System.Collections;
using System.IO;
using System.Xml;
using System.Xml.Serialization;
using System.Net.Sockets;
using System.Net;
using System.Text;

namespace Client
{
    class Client
    {
        public static void Connect()
        {
            CheckingAccount myAccount = new CheckingAccount();
            try
            {
                //创建 TCP 套接字
                TcpClient client = new TcpClient("127.0.0.1", 11111);

                //准备将 CheckingAccount 对象序列化到 XML 中
                XmlSerializer myXmlFactory =
                    new XmlSerializer(typeof(CheckingAccount));
```

```
                //创建 TCP 流
                NetworkStream stream = client.GetStream();

                // 序列化对象到 TCP 流
                myXmlFactory.Serialize(stream, myAccount);

                // 关闭所有资源
                stream.Close();
                client.Close();
            }
            catch (Exception ex)
            {
                Console.WriteLine("Exception: {0}", ex);
            }

            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
    }
}
```

14.3.3　服务器端代码

在该例中，我们使用了循环（事实上使用了多个循环）来实现这个服务器版本。这里我们也用代码注释展示了程序运行流程。服务器提供以下功能：

- 创建 checkingAccount 对象引用。
- 连接到套接字并且监听。
- 设置输入流。
- 创建流。
- 从流中读取字节。
- 序列化对象到流中。
- 关闭所有资源。

服务器端的 C# .NET 代码如下所示：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net.Sockets;
using System.Net;
using System.Xml;
using System.Xml.Serialization;
using System.IO;
using System.Runtime.Serialization;

namespace Server
{
```

```
class Server
{
    public Server()
    {
        TcpListener server = null;
        TcpClient client = null;
        try
        {
            //创建并启动套接字
            server = new TcpListener(IPAddress.Parse("127.0.0.1"),
                    11111);
            server.Start();

            //设置输入缓冲区
            Byte[] bytes = new Byte[256];

            //无限循环
            while (true)
            {
                //以阻塞模式获取到来的请求
                client = server.AcceptTcpClient();
                Console.WriteLine("Connected!");

                //打开流
                NetworkStream stream = client.GetStream();

                //从流中读取数据
                int i;
                while ((i = stream.Read(bytes, 0, bytes.Length)) != 0)
                {
                    //准备一个支持序列化器的流
                    MemoryStream ms = new MemoryStream(bytes);
                    //准备序列化器
                    XmlSerializer myXmlFactory =
                        new XmlSerializer(typeof(CheckingAccount));
                    //从流中创建 CheckingAccount 对象
                    myRestoredAccount =
                        (CheckingAccount)myXmlFactory.Deserialize(ms);
                    //现在演示确实创建了对象
                    Console.WriteLine("Name: {0}, Account Number: {1}.",
                        myRestoredAccount.Name,
                        myRestoredAccount.AccountNumber);
                    //抛出异常从而退出循环
                    throw new Exception("ignore");
                }
            }
        }
        catch (Exception ex)
        {
```

```
                if (!ex.Message.Equals("ignore"))
                        { Console.WriteLine("Exception: {0}", ex); }
            }
            finally
            {
                //关闭资源
                client.Close();
                server.Stop();
            }
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
    }
}
```

14.3.4　运行非私有客户端 / 服务器端示例

你可以使用 Visual Studio 创建一个工程，并且在一个简单的应用程序中使用该 C# .NET 代码，从而运行该示例。如下所示：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace Server
{
    class Program
    {
        static void Main(string[] args)
        {
            Server server = new Server();
        }
    }
}
```

14.4　结语

本章中我们涵盖了客户端 / 服务器端通信的概念。我们使用了两种截然不同的方式。第一种是使用 Java 创建一个私有的二进制系统通过网络传输对象。第二种方式是非私有的，使用了 .NET（C# 和 VB）语言。Java 也可以应用于这种非私有的基于 XML 的场景。

本章以及第 11 章和第 13 章都揭示了，不管网络是点对点的网络还是分布式网络，用开放的标准（比如 XML）可以通过各种网络传输对象。

14.5 引用

- Savitch, Walter. 2008. *Absolute Java,* 3rd ed. Boston, MA: Addison-Wesley.
- Walther, Stephen. 2008. *ASP.NET 3.5 Unleashed.* Indianapolis, IN: Sams Publishing.
- Skeet, Jon. 2008. *C# in Depth: What You Need to Master C# 2 and 3.* Greenwich, CT: Manning.
- Deitel, et al. 2003. *C# for Experienced Programmers.* Upper Saddle River, NJ: Prentice Hall.
- Deitel, et al. 2003. *Visual Basic .NET for Experienced Programmers.* Upper Saddle River, NJ: Prentice Hall.
- Jaworski, Jamie. 1999. *Java 2 Platform Unleashed.* Indianapolis, IN: Sams Publishing.
- Flanagan, David, et al. 1999. *Java Enterprise in a Nutshell.* Sebastopol, CA: O'Reilly.
- Farley, Jim. 1998. *Java Distributed Computing.* Sebastopol, CA: O'Reilly.
- Oracle: http://www.oracle.com/technetwork/java/index.html

14.6 本章中使用的示例代码

本章展示了 Java 和 C#.NET 版本的代码。其他语言的代码版本在出版社网站上有电子版。

第 15 章 Chapter 15

设计模式

软件开发的有趣之处在于，当设计软件系统时，实际上是在对现实世界系统进行建模。例如在信息技术领域，可以说 IT 等同于业务，或者至少说 IT 实现了业务。为了编写业务软件系统，开发人员必须彻底理解业务模型。结果开发人员往往非常熟悉公司业务方面的知识。

其实对现实世界进行建模这一概念贯穿本书。例如，当我们讨论使用继承来抽象出哺乳动物的行为和属性时，该模型基于现实世界中的模型，而不是出于自身目的勉强创建的模型。

因此，如果创建了一个哺乳动物（mammal）类，可以使用其衍生出无数个其他类，比如猫和狗等。因为所有的哺乳动物都共享某些行为和属性。狗、猫、松鼠以及其他哺乳动物都可以使用这个类，因为它们都符合一种模式。我们可以借助这个模式审查一个对象来确定它究竟是哺乳动物还是爬行动物，而爬行动物有符合某些其他行为和属性的模式。

纵观历史，模式被应用到了生活中的很多方面，比如软件工程方面。这些模式将软件开发领域中的软件重用升华到了一个新的高度。本章我们会讲述设计模式，这是软件开发中一个相对较新的领域（关于设计模式的精华之书发布于 1995 年）。

设计模式大概是过去几年中面向对象运动带来的最具影响力的开发活动之一。模式本身可以完美地融入可重用的软件开发过程中。因为面向对象开发目的就是为了重用，而模式和面向对象的开发关系密切。

设计模式的基本概念就是最佳实践的指导原则。最佳实践是指当创建了优秀的有效的解决方案时，这些解决方案会被记录下来，其他人可以从之前的成功中获得经验，也可以从失败中吸取教训。

面向对象的开发过程中最重要的一本书是《设计模式：可复用面向对象软件的基础》，由 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 编写。本书是软件领域中重要的里程碑，而且在计算机科学词库中占据重要地位。本书的作者被称为四人帮（Gang of Four）。在编写面向对象的主题时，你会经常看到对四人帮（简写为 GoF）的引用。

本章意图解释什么是设计模式。（解释所有设计模式已经远远超出了本书的范围，一本书根本讲不完。）四人帮将设计模式归为三类，分别是创建型模式、结构型模式和行为型模式，每个类别中我们会选取一种模式进行讲解，并且提供一个对应的例子。

15.1 为什么使用设计模式

设计模式的概念兴起之时并不是为了软件重用。其实设计模式的重大影响在于建造建筑和城市。Christopher Alexander 在《模式语言：城镇、建筑、构造》中强调说，“每个模式描述了一个在现实环境中反复出现的问题，然后描述了针对该问题的核心解决方案，你可以重复使用这个方案上百万次，而无需自行解决该问题”。

模式的 4 个元素

GoF 描述了一个模式包含 4 个必要的元素：

- 模式名称，使用一到两个词语来描述一个设计问题、对应的解决方案以及后果。可以将模式名称放入设计词汇中。模式名称可以建立起更高层级的抽象。建立模式词汇表有助于和同事讨论模式，甚至自己在文档中使用这些模式词汇。这样更容易思考设计并且权衡不同设计方案之间的利弊。为模式找到一个合适的名称并放到词汇表并不容易。
- 适用于该模式的待解决的问题，需要解释该问题的详细内容。可以描述具体的设计问题，比如如何用对象表示算法。也可以描述一个稳固设计中的类和对象结构。有时该问题会包含一系列条件，在应用该模式之前必须全部符合这些前置条件。
- 解决方案，描述了设计方案，比如类与对象之间的关系，各自的职责和协作等。该方案并不是一种特殊的具体的设计或实现，因为模式更像是模板，可以应用到多种情况中。模式提供了一种设计问题的抽象描述，以及如何使用通用的元素（这里指类和对象）来解决问题。
- 效果，效果是指应用该模式的结果以及利弊。尽管效果自身不说话，但当描述设计决策时，可用效果来评估设计决策以及帮助理解应用该模式的利弊。对于软件而言效果往往关心空间和时间的权衡，也可以关于语言和实现问题。因为重用通常是面向对象设计的一个因素，模式的效果包括了对系统灵活性、扩展性以及可移植性的影响。显式列出效果会有助于你理解并评价它们。

15.2 Smalltalk 的模型 / 视图 / 控制器

纵观历史，不得不提 Smalltalk 中引入的模型 / 视图 / 控制器（MVC）。其他面向对象的语言也使用了这种模式。MVC 经常用来演示设计模式的起源。在 Smalltalk 中，模型 / 视图 / 控制器范式用于创建用户接口。Smalltalk 大概是第一个广受欢迎的面向对象的语言。

> **Smalltalk**
>
> Smalltalk 是 Xerox PARC 提出的几个伟大的想法的最终结果。这些想法包括鼠标以及使用视窗环境等。Smalltalk 是一门精彩的语言，它提供了所有面向对象的语言需要遵循的根本特性。对于 C++ 的抱怨之一是它不是真正的面向对象语言，而 Smalltalk 则是真正的面向对象的语言。尽管在应用面向对象的早期，C++ 有更多的追随者，而 Smalltalk 始终有一个非常专注的核心支持团队。Java 是一个最受 C++ 开发人员喜爱的面向对象的语言。

设计模式以以下方式定义了 MVC 组件：

模型是应用程序对象，视图是屏幕展示，控制器则定义了用户接口如何响应用户输入。

之前的范式的问题在于模型、视图和控制器都被集中到了单个实体中。例如，单个对象会包括这三个部分。使用 MVC 范式，这三个部件被分离开来并且拥有截然不同的接口。所以如果你想修改应用程序的用户接口，你只需修改视图即可。图 15-1 演示了 MVC 设计。

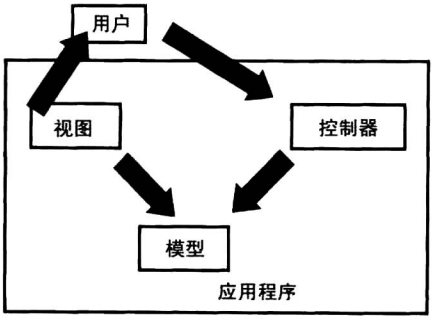


图 15-1 模型 / 视图 / 控制器范式

学习面向对象的开发过程其实就是关于接口与实现的设计问题。我们想尽可能地隔离接口和实现，也想尽可能隔离不同的接口。例如，如果多个接口之间没有任何关联（或者解决方案与问题无关），那么就不需要组合这些接口。MVC 是实践接口分离的先锋之一。MVC 显式定义了特定的组件之间的接口，这些不同的组件一起解决了一个非常通用的且基本的编程问题。这个编程问题就是创建用户接口，并且建立业务逻辑和业务背后的数据之间的联系。

如果遵循 MVC 概念分离用户接口、业务逻辑和数据，那么系统会更加灵活和稳定。例如，假设用户接口在客户端机器上，业务逻辑在应用程序服务器中，而数据则存放在数据服务器中。使用这种方式开发应用程序，只修改 GUI 界面而不会对业务逻辑或数据造成影响。同样，如果修改了业务逻辑（比如改变了一个特殊字段的计算方式），那么无需改变 GUI。最后，如果想使用另一种数据库存放数据，可以修改服务器上数据存放的方式而无需影响 GUI 或者业务逻辑。当然，这些假设都基于这三者之间的接口没有改变。

MVC 示例

我们可以用列表框（listbox）这个例子来进一步演示。假设 GUI 包含一组电话号码。列表框是视图层，电话号码列表是模型，而控制器是绑定列表框与电话列表的逻辑。

MVC 缺点

尽管 MVC 是一个优秀的设计，但某些情况下会过于复杂，比如会增加前端设计的复杂度。这是采用面向对象设计的常见问题，即优秀的设计与丑陋的设计只在一线之间。问题在于对于一个完整的设计，你究竟会引入多少复杂度？

15.3 设计模式类型

设计模式一书中将 23 种模式归为 3 类。里面大多数例子是用 C++ 编写的，少部分使用 Smalltalk 编写。本书的发布时间正是 C++ 和 Smalltalk 兴起之时。发布时间是在 1995 年，正处于互联网革命的开端，以及 Java 编程语言开始流行之时。当设计模式的优势显露之后，迅速有大量相关书籍面世。这些书使用的语言大部分都是 Java。

设计模式与实际使用的语言是不相关的。设计模式本质上是一本设计书籍，里面的模式可以应用在任何语言中。本书的作者们将模式分成三个类别：

- 创建型模式。帮你创建对象，你无需直接实例化对象。你可以根据给定的条件创建对象，这给程序带来更大的灵活性。
- 结构型模式。将一组对象组合成更复杂的结构，比如复杂的用户接口或者账单数据。
- 行为型模式。定义系统中对象之间的通信方式，控制复杂程序中的流向。

接下来的小节我们会从每个类别中挑出一个模式配以示例进行讲解。本章节最后会列出一系详细讲解设计模式的书籍。

15.3.1 创建型模式

创建型模式包含以下模式：

- 抽象工厂模式。
- 构造器模式。
- 工厂方法模式。
- 原型模式。
- 单例模式。

之前已经说过，本章节主要讲述什么是设计模式，而不是详细描述 GoF 书中的每个模式。因此，每个类别中的模式我们只会讲解一个。在创建型模式中，我们选择单例模式进行讲述。

单例设计模式

图 15-2 中展示的单例模式是一种创建型模式，当用类创建对象时始终只有一个对象实

例。例如，假设一个网站中有一个计数器对象来记录网站的点击次数，你当然不想每次页面被点击时都实例化一个计数器对象。你需要在第一次点击时实例化一个计数器对象，之后一直使用这个对象来进行计数。

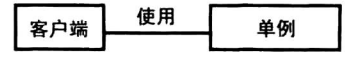


图 15-2 单例模式

尽管可能有其他的方式来限制对象的创建，但通常最好由类自身来解决该问题。不过很多时候使用外部工厂的方式也非常实用。诸如工厂模式、抽象工厂模式以及桥接模式都是重要的设计模式。

关注业务

请记住面向对象的重要规则之一是对象的职责由自身管理。这意味着，类自身应该处理其生命周期，而不应该交由诸如静态类等语言结构来处理。

图 15-3 是《设计模式》书中单例模式的 UML 模型。请注意属性 uniqueinstance 和方法 Instance()。uniqueinstance 是一个静态的单例对象。其他属性和方法用于实现该类的业务逻辑。

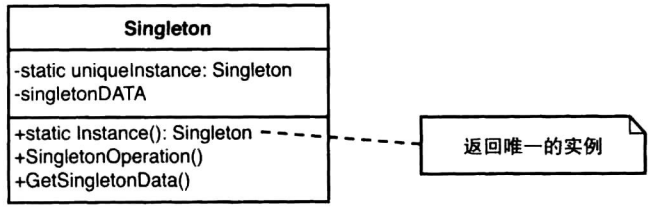


图 15-3 单例模式的 UML 图

其他类必须通过 Instance() 方法来访问单例中的实例。正如其他的面向对象设计方式一样，应该使用构造函数控制对象的创建。我们可以将 Instance() 方法作为接口，然后用户可以调用 Instance() 方法来间接调用构造函数。

接下来的 Java 代码演示了常见的单例模式：

```
public class ClassicSingleton {
  private static ClassicSingleton instance = null;

  protected ClassicSingleton() {
   // 只是为了防止实例化
  }
  public static ClassicSingleton getInstance() {
   if(instance == null) {
     instance = new ClassicSingleton();
   }
   return instance;
  }
}
```

我们可以为网页计数器例子中提到的计数器类创建一个单例模式版本：

```
public class Counter
{
  private int counter;
  private static Counter instance = null;

  protected Counter()
  {
  }

  public static Counter getInstance() {
      if(instance == null) {
         instance = new Counter ();
        System.out.println("New instance created\n");
      }
      return instance;
  }

  public void incrementCounter()
  {
    counter++;
  }

  public int getCounter()
  {
    return(counter);
  }

}
```

该代码主要关注点是对创建对象的限制，即只能创建一个计数器对象。这段代码如下：

```
public static Counter getInstance() {
    if(instance == null) {
       instance = new Counter ();
      System.out.println("New instance created\n");
    }
    return instance;
}
```

注意如果 instance 是 null，说明对象还没有被实例化，这种情况下会新创建一个 Counter 对象。如果 instance 不为 null，那么说明实例化了一个 Counter 对象，则不会创建新的对象。这种情况下会直接返回对该对象的引用。

多个引用

对于单例对象会出现多个引用。如果在应用程序中你有多个引用且每个引用都指向这个单例对象，你则需要管理这些引用。

这段代码当然有趣，不过值得理解应用程序如何实例化和管理单例对象。请看以下

代码：

```
public class Singleton
{
 public static void main(String[] args)
 {
  Counter counter1 = Counter.getInstance();
  System.out.println("Counter : " + counter1.getCounter() );

  Counter counter2 = Counter.getInstance();
  System.out.println("Counter : " + counter2.getCounter() );

 }
}
```

对单个 Counter 对象的两个引用

请注意该例子中两个单独的引用指向了计数器。因此当计数器改变时，两个引用都会反映出更新。

这段代码使用了 Counter 单例对象。请看对象是如何被创建的：

```
Counter counter1 = Counter.getInstance();
```

这里并没有使用构造函数。getInstance() 方法控制对象的实例化。图 15-4 展示了这段代码执行的过程。注意 New instance created 这一消息只输出了一次。当创建 counter2 时，得到的是原始对象的副本，与 counter1 完全一样。

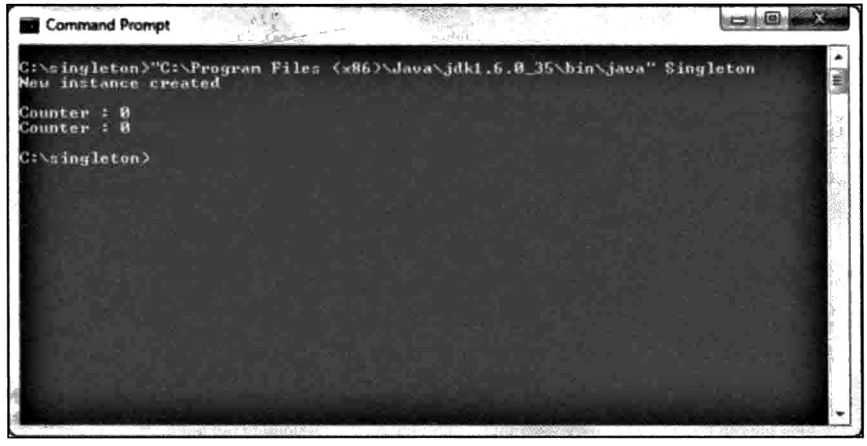


图 15-4 使用 Counter 单例

我们可以证明 counter1 和 counter2 是完全一样的。请看以下更新后的应用程序代码：

```
public class Singleton
{
  public static void main(String[] args)
  {
    Counter counter1 = Counter.getInstance();
    counter1.incrementCounter();
    counter1.incrementCounter();
    System.out.println("Counter : " + counter1.getCounter() );

    Counter counter2 = Counter.getInstance();

    counter2.incrementCounter();
    System.out.println("Counter : " + counter2.getCounter() );

  }
}
```

图 15-5 展示了这个单例应用程序的输出。注意该例中，我们增加了 counter1 两次，所以计数器应该是 2。当创建 counter2 引用时，它引用了 counter1 这个对象，所以增加计数器后，变成了 3（2+1）。

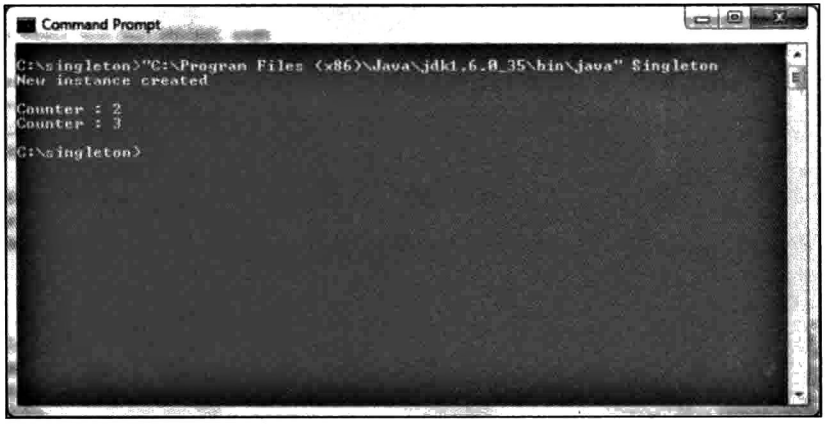


图 15-5 使用更新后的 Counter 单例

回顾批处理文件

正如之前章节中演示过的一样，在教学示例中为了使得测试既简单又实用，我喜欢使用批处理文件来精确指定 Java 类路径。你可以将以下代码放到一个名为 Singleton.bat 批处理文件中，然后在命令行终端输入 Singleton 来启动这个单例应用程序。

```
"C:\Program Files (x86)\Java\jdk1.6.0_35\bin\java" Singleton
```

15.3.2 结构型模式

结构型模式用于使用一组对象来创建更复杂的结构。下面的 7 种模式属于结构型模式：

- 适配器模式。

- 桥接模式。
- 组合模式。
- 装饰器模式。
- 外观模式。
- 轻量模式。
- 代理模式。

我们将适配器模式作为结构型模式的示例。适配器模式也是重要的设计模式之一。该模式是分离实现与接口的经典示例。

适配器模式

适配器模式可以为现有类创建不同的接口。适配器模式提供了一个包装类。换句话说，你创建了一个新类，使用新的更理想化的接口包含（包装）了现有类的功能。包装类的一个简单例子是 Java 类 Integer。Integer 类包装了一个整数值。你可能想为什么要这样做，请记住在面向对象的系统中，任何事物都是对象。在 Java 中，原生类型（比如 int、float 等）不是对象。当需要对这些原生对象执行操作（比如转换）时，你需要把它们作为对象来对待。你创建一个包装对象并“包装”这个原生类型。因此可以用以下方式使用原生类型：

```
int myInt = 10;
```

将其包装到一个 Integer 对象中：

```
Integer myIntWrapper = new Integer (myInt);
```

现在你可以将其转化为字符串：

```
String myString = myIntWrapper.toString();
```

这个包装类可以把原生的整型当作一个对象，因此可以利用对象的所有特性。

我们用一个邮件工具接口例子来阐述适配器模式。假设你购买了一些代码，这些代码实现了邮件客户端的所有功能。该工具几乎能满足你对邮件客户端的所有要求，但你想要做一丁点修改，即你想修改获取邮件的 API。

下面的类提供了一个邮件客户端的简单示例：

```
package MailTool;
public class MailTool {
   public MailTool () {
   }
   public int retrieveMail() {

    System.out.println ("You've Got Mail");

         return 0;
   }
}
```

当调用 retrieveMail() 方法时，你的邮件会展示一个初始的欢迎信息“You've Got Mail”。现在假设你想将公司客户端的接口从 retrieveMail() 修改为 getMail()。你可以创建一个接口来强制实现：

```
package MailTool;
interface MailInterface {
   int getMail();
}
```

接下来可以自定义邮件工具来包装原生工具，并实现自定义的接口：

```
package MailTool;
class MyMailTool implements MailInterface {
   private MailTool yourMailTool;
   public MyMailTool () {
    yourMailTool= new MailTool();
        setYourMailTool(yourMailTool);
   }
   public int getMail() {
      return getYourMailTool().retrieveMail();
   }
   public MailTool getYourMailTool() {
      return yourMailTool ;
   }
   public void setYourMailTool(MailTool newYourMailTool) {
      yourMailTool = newYourMailTool;
   }
}
```

在这个类中，创建了一个原生邮件工具的新实例。该类实现了 MailInterface 接口，这会强制实现 getMail() 方法。在这个方法中直接调用原生邮件工具的 retrieveMail() 方法。

可以实例化新的邮件工具，使用新类来调用 getMail() 方法：

```
package MailTool;
public class Adapter
{
  public static void main(String[] args)
  {
    MyMailTool myMailTool = new MyMailTool();

    myMailTool.getMail();

  }
}
```

当调用 getMail() 方法时，其实是使用新的接口来调用原生工具的 retrieve-Mail() 方法。这是一个非常简单的例子，然而通过创建这个包装类，可以增强接口并且为原生类添加自定义的功能。

适配器的概念非常简单，但借助该模式可以创建新的强大的接口。

15.3.3 行为型模式

行为型模式包含以下类别：

- 责任链模式。
- 命令模式。
- 解释器模式。
- 迭代器模式 。
- 中介者模式。
- 备忘录模式。
- 观察者模式。
- 状态模式。
- 策略模式。
- 模板方法模式。
- 访问者模式。

我们将迭代器模式作为行为型模式类别中的示例。这是一个非常常见的模式，多种编程语言都实现了该模式。

迭代器设计模式

迭代器提供了一个标准的机制来遍历集合（比如列表）。该机制提供的功能要保证集合中的每个条目都必须至少被访问一次。迭代器模式隐藏了信息，保证了集合的内部结构是秘密的。迭代器模式也规定可以创建多个迭代器，迭代器之间不会相互妨碍。Java 语言自身实现了迭代器。以下的代码创建了一个列表，并且插入一组字符串：

```
package Iterator;

import java.util.*;
public class Iterator {
    public static void main(String args[]) {

        // 实例化 ArrayList.
        ArrayList<String> names = new ArrayList();

        // 向 ArrayList 中添加值
        names.add(new String("Joe"));
        names.add(new String("Mary"));
        names.add(new String("Bob"));
        names.add(new String("Sue"));

                //迭代名称
        System.out.println("Names:");
        iterate(names );
    }
```

```
    private static void iterate(ArrayList<String> arl) {
        for(String listItem : arl) {
                System.out.println(listItem.toString());
            }
    }
}
```

然后创建了一个枚举类型，从而可以通过该枚举进行迭代。方法 iterate() 用来执行迭代功能。在这个方法中，我们使用 Java 枚举方法 hasMoreElements()，该方法会遍历这个列表并且列出所有名字。

15.4 反模式

设计模式是从有益的经验中总结出来的，反模式则来自于失败的经验。大多数软件项目最终不成功的原因都会被记载下来，最终总结为反模式。事实上，Johnny Johnson 文章“制造混乱”一文中指出，整整三分之一的项目最终都会被彻底取消。显而易见，大多数失败原因都是因为错误的设计决策。

设计模式都是为了尽力解决一个特定类型的问题，而反模式则恰恰相反。换句话说，反模式反映了解决一个问题时失败的经验。简单来说，设计模式是一种好的设计实践，反模式则是一种需要避免的实践。

Andrew Koenig 在 1995 年 11 月《C++ 报告》中描述了两种类型的反模式：

- 一种是描述了解决问题的糟糕的解决方案，导致情形向坏的方向发展。
- 另一种是描述了如何使用好的解决方案来摆脱糟糕的情形。

很多人相信反，模式比设计模式更有用。这是因为反模式可用来解决他们已经遇到的问题。这会导致人们停止于对问题根源的分析。通过数据分析可以得出最初的设计（可能是一种设计模式）为什么不成功。失败的方案会暴露出反模式。因此，反模式有点像事后诸葛亮。

例如，在文章“重用模式和反模式”中，Scott Ambler 识别了一个叫作健壮的组件的模式，并且进行了如下定义：

> 健壮的组件是指该组件拥有详尽的文档，它不是为了满足具体项目的需求，而具有通用性，并且经过完整测试，而且有几个示例来展示如何使用该组件。如果组件满足以上几点，那么它更容易被重用。更容易理解和使用健壮的组件。

不过当声明了一个可重用的方案后，很可能有大部分场景下没人会重用它。因此，为了阐述反模式，他写道：

> 除了原始的开发人员，其他人员也必须审查该组件。如果其他人员也对该组件感兴趣，那么必须保证重新设计该组件为健壮的组件。

因此，反模式会导致修订现有设计，而且持续重构这些设计直到找到一个可以工作的方案。

15.5 结语

本章我们探索了设计模式这一概念。模式是日常生活的一部分，也是你思考面向对象设计的方式。而生活与信息技术息息相关，往往可以在真实生活情况中找到根本的解决方案。

本章只是概述了设计模式，你可以通过本章最后列出的引用书籍来深入探索设计模式。

15.6 引用

- Alexander, Christopher, et al. 1977. *A Pattern Language: Towns, Buildings, Construction.* Cambridge, UK: Oxford University Press.
- Gamma, Erich, et al. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston, MA: Addison-Wesley.
- Larman, Craig. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development,* 3rd ed. Hoboken, NJ: Wiley.
- Grand, Mark. 2002. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML,* 2nd ed., volume 1. Hoboken, NJ: Wiley.
- Ambler, Scott. "Reuse Patterns and Antipatterns." *2000 Software Development Magazine.*
- Jaworski, Jamie. 1999. *Java 2 Platform Unleashed.* Indianapolis, IN: Sams Publishing.
- Johnson, Johnny. "Creating Chaos." *American Programmer,* July 1995.

15.7 本章中使用的示例代码

以下是 C#.NET 版本的代码。其他语言在出版社网站上有电子版。本章已经展示了这些例子对应的 Java 代码。

C# .NET

Counter.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Counter
{
    class Counter
    {
        private int counter;
        private static Counter instance = null;

        protected Counter()
```

```
        {

        }

        public static Counter getInstance()
        {
            if (instance == null)
            {
                instance = new Counter();
                Console.WriteLine("New Instance of Counter...");
            }
            return instance;
        }

        public void incrementCounter()
        {
            counter++;
        }

        public int getCounter()
        {
            return counter;
        }
    }
}
```

Singleton.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Counter
{
    class Singleton
    {
        public Singleton()
        {
            Counter counter1 = Counter.getInstance();
            counter1.incrementCounter();
            counter1.incrementCounter();
            Console.WriteLine("Counter = " + counter1.getCounter());

            Counter counter2 = Counter.getInstance();
            counter2.incrementCounter();
            Console.WriteLine("Counter = " + counter2.getCounter());
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
    }
}
```

MailTool.cs

```
using System;

namespace MailAdapter
{

    class MailTool
    {
        public MailTool()
        {

        }

        public int retrieveMail()
        {
            Console.WriteLine("You've got mail!");
            return 0;
        }
    }
}
```

MailInterface.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace MailAdapter
{
    interface MailInterface
    {
        int getMail();
    }
}
```

MyMail.cs

```
namespace MailAdapter
{
    class MyMailTool : MailInterface
    {
        private MailTool yourMailTool;
        public MyMailTool()
        {
            yourMailTool = new MailTool();
            setYourMailTool(yourMailTool);
        }

        public int getMail()
        {
```

```
            return getYourMailTool().retrieveMail();
        }

        public MailTool getYourMailTool()
        {
            return yourMailTool;
        }

        public void setYourMailTool(MailTool newYourMailTool)
        {
            yourMailTool = newYourMailTool;
        }
    }
}
```

Adapter.cs

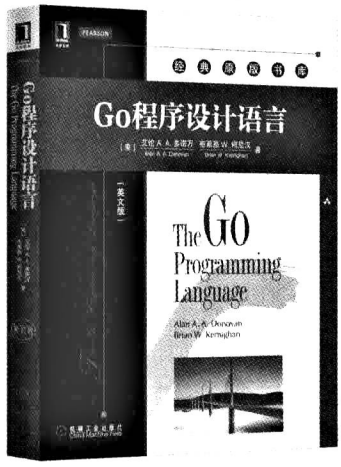
```
using System;
using System.Collections.Generic;
using System.Text;

namespace MailAdapter
{
    class Adapter
    {
        public Adapter()
        {
            MyMailTool myMailTool = new MyMailTool();
            myMailTool.getMail();
            Console.WriteLine();
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
    }
}
```

Iterator.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;


namespace Iterator
{
    class Iterator
    {
        public Iterator()
        {
```

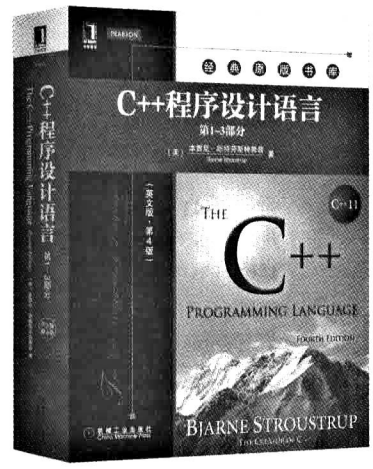
```
            // 实例化 ArrayList
            ArrayList myList = new ArrayList();

            // 向列表中添加值
            myList.Add("Joe");
            myList.Add("Mary");
            myList.Add("Bob");
            myList.Add("Sue");

            //迭代列表中的元素
            Console.WriteLine("Names:");
            iterate(myList);
        }

        static void iterate(ArrayList arl)
        {
            foreach (String listItem in arl)
            {
                Console.WriteLine(listItem);
            }
        }
    }

}
```

推荐阅读



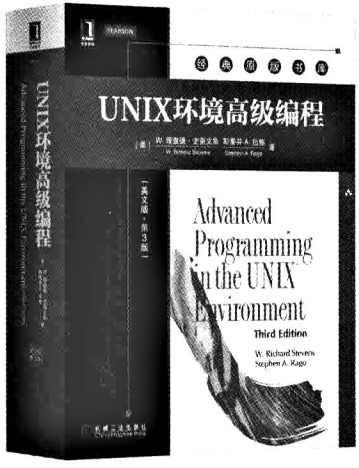
Go程序设计语言（英文版）

作者：Alan A. A. Donovan Brian W. Kernighan
书号：978-7-111-52628-5
定价：79.00元



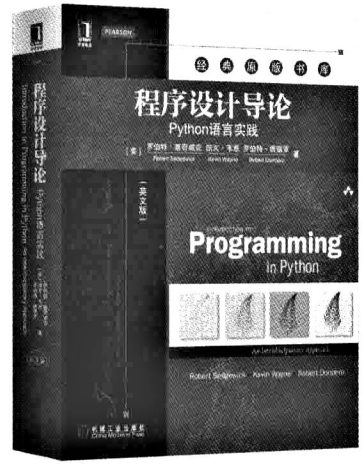
C++程序设计语言（英文版·第4版）

作者：Bjarne Stroustrup
第1～3部分：ISBN：978-7-111-52386-4，169.00元
第4部分：ISBN：978-7-111-52487-8，89.00元



UNIX环境高级编程（英文版·第3版）

作者：W. Richard Stevens Stephen A. Rago
ISBN：978-7-111-52387-1
定价：159.00元



程序设计导论：Python语言实践（英文版）

作者：Robert Sedgewick Kevin Wayne Robert Dondero
ISBN：978-7-111-52401-4
定价：139.00元

The Object-Oriented Thought Process
Fourth Edition

面向对象编程是现代编程语言的基础，包括C++、Java、C#、Visual Basic .NET、Ruby和Objective-C。对象的概念也是许多Web技术的基础。在开始使用面向对象开发环境前，学习面向对象的基本概念至关重要。虽然编程技术在不断演化，但无论在哪个平台，面向对象的概念依旧长存。

本书第1~10章涵盖了基本的面向对象概念，第11~15章将这些概念应用到了一些常用的面向对象技术中。

第四版相对于之前的版本拓展了很多主题：

- 移动设备开发，包括手机应用、移动应用以及混合开发等
- iOS环境下的Objective-C代码示例
- 使用XML及JSON实现可读性强的数据交换
- 使用CSS、XSLT等技术实现数据渲染与转换
- Web服务，包括简单对象访问协议(SOAP)、RESTful Web服务等
- 客户端/服务器端技术以及封送对象
- 持久化数据和序列化对象

Pearson
www.pearson.com

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/程序设计
ISBN 978-7-111-55308-3
9 787111 553083 >
定价：69.00元