

国外计算机科学经典教材

Object-Oriented  
Analysis & Design  
Understanding System  
Development with UML 2.0

面向对象分析与设计  
(UML 2.0 版)

(美) Mike O'Docherty 著  
俞志翔 译



清华大学出版社

# Object-Oriented Analysis & Design

## Understanding System Development with UML 2.0

本书详细介绍了面向对象的分析与设计，全面探讨了面向对象概念、软件开发过程、UML 和多层技术。

本书使用最常见的技术和方法，通过一个贯穿全书的案例分析，对面向对象的软件开发过程和使用面向对象技术的编程过程进行了透彻的讨论。首先阐述了面向对象软件项目的基本概念，然后基于广泛应用的 Rational Unified Process(RUP)方法，介绍了使用 JUnit 进行以测试为驱动的开发过程，最后研究了现实世界中的开发问题。

### 本书特色

- 本书按照典型开发项目的步骤，整合了需求、设计、规范和测试
- 案例分析清晰地说明了如何分析抽象的问题，从而最终得出一个具体的解决方案
- 合作站点上的AQS(自动组卷系统)练习题使读者可以实践本书描述的技术

### 读者对象

无论读者是在校师生还是参加业务培训的开发人员，或者是转向面向对象技术的有经验的程序员，本书都会对您有所帮助。

合作网站：[www.wiley.com/go/odocherty](http://www.wiley.com/go/odocherty)

ISBN 7-302-12546-5



9 787302 125464 >

定价：42.00 元

国外计算机科学经典教材

# 面向对象分析与设计 (UML 2.0 版)

(美) Mike O'Docherty 著

俞志翔 译

清华大学出版社

北京

Mike O'Docherty

Object-Oriented Analysis and Design: Understanding System Development with UML 2.0

EISBN: 0-470-09240-8

Copyright©2005 by John Wiley & Sons, Inc.

All Rights Reserved. Authorized translation from the English language edition published by John Wiley & Sons, Inc.

本书中文简体字版由 John Wiley & Sons, Inc. 授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字：01-2005-6692

版权所有，翻印必究。举报电话：010-62782989 13901104297 13801310933

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

本书防伪标签采用特殊防伪技术，用户可通过在图案表面涂抹清水，图案消失，水干后图案复现；或将面膜揭下，放在白纸上用彩笔涂抹，图案在白纸上再现的方法识别真伪。

#### 图书在版编目(CIP)数据

面向对象分析与设计(UML 2.0 版)/(美)多切蒂(O'Docherty,M.)著；俞志翔 译. —北京：清华大学出版社，2006.4

书名原文：Object-Oriented Analysis and Design: Understanding System Development with UML 2.0

(国外计算机科学经典教材)

ISBN 7-302-12546-5

I. 面… II. ①多… ②俞… III. 面向对象语言，UML - 程序设计 IV. TP312

中国版本图书馆 CIP 数据核字(2006)第 009825 号

出 版 者：清华大学出版社 地 址：北京清华大学学研大厦

<http://www.tup.com.cn> 邮 编：100084

社 总 机：010-62770175 客户服务：010-62776969

组稿编辑：曹 康

文稿编辑：于 平

封面设计：康 博

版式设计：康 博

印 刷 者：北京市人民文学印刷厂

装 订 者：三河市金元印装有限公司

发 行 者：新华书店总店北京发行所

开 本：185×260 印张：23.5 字数：601 千字

版 次：2006 年 4 月第 1 版 2006 年 4 月第 1 次印刷

书 号：ISBN 7-302-12546-5/TP · 8025

印 数：1 ~ 4000

定 价：42.00 元

# 出 版 说 明

近年来，我国高等学校的计算机学科教育进行了较大的改革，急需一批门类齐全、具有国际水平的计算机经典教材，以适应当前的教学需要。引进国外经典教材，可以了解并吸收国际先进的教学思想和教学方法，使我国的计算机学科教育能够与国际接轨，从而培育更多具有国际水准的计算机专业人才，增强我国信息产业的核心竞争力。Pearson、Thomson、McGraw-Hill、Springer、John Wiley 等出版集团都是全球最有影响的图书出版机构，它们在高等教育领域也都有着不凡的表现，为全世界的高等学校计算机教学提供了大量的优秀教材。为了满足我国高等学校计算机学科的教学需要，我社计划从这些知名的国外出版集团引进计算机学科经典教材。

为了保证引进版教材的质量，我们在全国范围内组织并成立了“清华大学计算机外版教材编审委员会”（以下简称“编委会”），旨在对引进教材进行审定、对教材翻译质量进行评审。“编委会”成员皆为全国各类重点院校教学与科研第一线的知名教授，其中许多教授为各校相关院、系的院长或系主任。“编委会”一致认为，引进版教材要能够满足国内各高校计算机教学与国际接轨的需要，要有特色风格，有创新性、先进性、示范性和一定的前瞻性，是真正的经典教材。为了保证外版教材的翻译质量，我们聘请了高校计算机相关专业教学与科研第一线的教师及相关领域的专家担纲译者，其中许多译者为海外留学回国人员。为了尽可能地保留与发扬教材原著的精华，在经过翻译和编辑加工之后，由“编委会”成员对文稿进行审定，以最大程度地弥补和修正在前面一系列加工过程中对教材造成的误差和瑕疵。

由于时间紧迫和能力所限，本套外版教材在出版过程中还可能存在一些不足和遗憾，欢迎广大师生批评指正。同时，也欢迎读者朋友积极向我们推荐各类优秀的国外计算机教材，共同为我国高等学校的计算机教育事业贡献力量。

清华大学出版社

# 国外计算机科学经典教材

## 编审委员会

### 主任委员：

孙家广 清华大学教授

### 副主任委员：

周立柱 清华大学教授

### 委员（按姓氏笔画排序）：

王成山	天津大学教授
王 珊	中国人民大学教授
冯少荣	厦门大学教授
冯全源	西南交通大学教授
刘乐善	华中科技大学教授
刘腾红	中南财经政法大学教授
吉根林	南京师范大学教授
孙吉贵	吉林大学教授
阮秋琦	北京交通大学教授
何 晨	上海交通大学教授
吴百锋	复旦大学教授
李 彤	云南大学教授
沈钧毅	西安交通大学教授
邵志清	华东理工大学教授
陈 纯	浙江大学教授
陈 钟	北京大学教授
陈道蓄	南京大学教授
周伯生	北京航空航天大学教授
孟祥旭	山东大学教授
姚淑珍	北京航空航天大学教授
徐佩霞	中国科学技术大学教授
徐晓飞	哈尔滨工业大学教授
秦小麟	南京航空航天大学教授
钱培德	苏州大学教授
曹元大	北京理工大学教授
龚声蓉	苏州大学教授
谢希仁	中国人民解放军理工大学教授

# 目 录

<b>第 1 章</b>	<b>入门</b>	1
1.1	背景	1
1.2	编程简史	1
1.3	方法学	2
1.4	关于本书	2
1.4.1	内容概述	3
1.4.2	案例分析	3
1.4.3	导航	3
<b>第 I 部分 设置场景</b>		
<b>第 2 章</b>	<b>对象的概念</b>	7
2.1	引言	7
2.2	什么是对象	8
2.3	相同还是相等	10
2.4	描述对象	12
2.5	封装	13
2.6	关联和聚合	13
2.7	图和树	15
2.8	链接和可导航性	16
2.9	消息	17
2.10	启动操作	19
2.11	协作示例	19
2.12	面向对象程序的工作原理	21
2.13	垃圾收集	22
2.14	类	22
2.15	类定义的内容	24
2.16	共享数据和共享操作	26
2.17	类型	27
2.18	术语	27
2.19	重用代码	29
2.20	小结	32
2.21	课外阅读	32
2.22	复习题	32

2.23	练习 1 的答案	33
2.24	复习题答案	33
<b>第 3 章</b>	<b>继承</b>	34
3.1	引言	34
3.2	设计类层次结构	35
3.3	给类层次结构添加实现代码	36
3.4	抽象类	38
3.5	重定义方法	40
3.6	实现栈类	40
3.6.1	使用继承实现栈	41
3.6.2	使用复合实现栈	42
3.6.3	继承和复合	43
3.7	多重继承	44
3.8	使用继承的规则	47
3.9	小结	47
3.10	课外阅读	47
3.11	复习题	47
3.12	复习题答案	49
<b>第 4 章</b>	<b>类型系统</b>	50
4.1	引言	50
4.2	动态和静态类型系统	50
4.3	多态性	51
4.3.1	多态变量	52
4.3.2	多态消息	53
4.4	动态绑定	54
4.5	多态性规则	56
4.6	类型转换	56
4.7	显式类型转换	57
4.8	使用模板进行泛化	59
4.9	小结	60
4.10	课外阅读	60
4.11	复习题	60
4.12	练习 2 的答案	62

4.13 练习 3 的答案	62
4.14 复习题答案	62
<b>第 5 章 软件开发的方法学</b>	<b>64</b>
5.1 引言	64
5.2 软件开发中的经典阶段	65
5.2.1 需求	65
5.2.2 分析	66
5.2.3 设计	66
5.2.4 规范	66
5.2.5 实现	66
5.2.6 测试	66
5.2.7 部署	67
5.2.8 维护	67
5.2.9 关键问题	67
5.3 软件工程和瀑布方法学	68
5.4 新方法学	71
5.4.1 螺旋式方法学	71
5.4.2 迭代式方法学	72
5.4.3 递增式方法学	72
5.4.4 合并方法学	73
5.5 面向对象的方法学	74
5.5.1 UML、RUP 和 XP	74
5.5.2 开发工具的需求	75
5.6 Ripple 概述	76
5.6.1 用例图	78
5.6.2 类图(分析级别)	79
5.6.3 通信图	79
5.6.4 部署图	80
5.6.5 类图(设计级别)	81
5.6.6 顺序图	81
5.7 小结	82
5.8 课外阅读	82
5.9 复习题	82
5.10 复习题答案	83
<b>第 II 部分 理解问题</b>	
<b>第 6 章 收集需求</b>	<b>87</b>
6.1 引言	87
6.2 系统的诞生	88

6.3 用例	89
6.4 业务说明	90
6.4.1 标识业务参与者	90
6.4.2 编写项目术语表	91
6.4.3 标识业务用例	92
6.4.4 在通信图中演示用例	93
6.4.5 在活动图中演示用例	94
6.5 开发人员的说明	95
6.5.1 使参与者特殊化	98
6.5.2 用例的关系	99
6.5.3 系统用例的细节	102
6.5.4 前提条件、后置条件和继承	104
6.5.5 辅助需求	104
6.5.6 用户界面草案	104
6.5.7 系统用例的优先级	105
6.6 小结	107
6.7 课外阅读	107
6.8 复习题	107
6.9 复习题答案	109
<b>第 7 章 分析问题</b>	<b>110</b>
7.1 引言	110
7.2 为什么要进行分析	110
7.3 分析过程概述	111
7.4 静态分析	112
7.4.1 确定类	112
7.4.2 标识类的关系	112
7.4.3 绘制类图和对象图	112
7.4.4 绘制关系	114
7.4.5 属性	117
7.4.6 关联类	120
7.4.7 有形对象和无形对象	120
7.4.8 好的对象	124
7.5 动态分析	124
7.5.1 绘制用例的实现过程	124
7.5.2 边界、控制器和实体	126
7.5.3 通信图中的元素	127
7.5.4 给类添加操作	128
7.5.5 职责	129
7.5.6 状态建模	129
7.6 小结	130

7.7	课外阅读	130
7.8	复习题	131
7.9	练习 4 的答案	133
7.10	复习题答案	133
<b>第III部分 设计解决方案</b>		
<b>第 8 章 设计系统体系结构</b> 137		
8.1	引言	137
8.2	设计优先级	138
8.3	系统设计中的步骤	138
8.4	选择联网的系统拓扑	139
8.4.1	网络体系结构的简史	139
8.4.2	三层体系结构	140
8.4.3	个人计算机	142
8.4.4	网络计算机	142
8.4.5	互联网和万维网	143
8.4.6	内联网	143
8.4.7	外联网和虚拟私人网络	144
8.4.8	客户机—服务器与分布式体系结构	144
8.4.9	用 UML 描述网络拓扑	146
8.5	并发设计	147
8.6	安全设计	148
8.6.1	数字加密和解密	148
8.6.2	一般安全规则	149
8.7	分解软件	150
8.7.1	系统和子系统	150
8.7.2	层	151
8.7.3	Java 层：应用小程序和 RMI	153
8.7.4	层中的消息流	155
8.8	小结	158
8.9	课外阅读	158
8.10	复习题	158
8.11	复习题答案	159
<b>第 9 章 选择技术</b> 160		
9.1	引言	160
9.2	客户层技术	160
9.3	客户层到中间层的协议	162
9.4	中间层技术	163

9.5	中间层到数据层的技术	164
9.6	其他技术	165
9.7	一般前端配置	166
9.7.1	HTML/CGI 和脚本	166
9.7.2	HTML/CGI 和服务小程序	167
9.7.3	RMI	168
9.7.4	CORBA	169
9.7.5	EJB	170
9.8	后端配置	171
9.9	Java 电子商务配置	171
9.10	UML 包	174
9.11	小结	177
9.12	课外阅读	177
9.13	复习题	178
9.14	复习题答案	178
<b>第 10 章 设计子系统</b> 179		
10.1	引言	179
10.2	把分析的类模型映射为设计的类模型	180
10.2.1	映射操作	180
10.2.2	变量类型	180
10.2.3	字段的可见性	180
10.2.4	访问器	181
10.2.5	映射类、属性和复合	181
10.2.6	映射其他类型的关联	182
10.2.7	通用标识符	186
10.3	使用关系数据库实现存储	187
10.3.1	数据库管理系统	187
10.3.2	关系模型	188
10.3.3	映射实体类	190
10.3.4	映射关联	190
10.3.5	映射对象状态	193
10.4	最终确定用户界面	196
10.5	设计业务服务	200
10.5.1	使用代理和副本	201
10.5.2	给业务服务分类	203
10.5.3	会话标识符	204
10.5.4	业务服务的实现	204
10.6	使用模式、框架和库	206
10.7	事务	206
10.7.1	保守并发和开放并发	207

10.7.2	使用事务和对象的一般规则	207	12.6.1	OCL 中的正式规范	250
10.7.3	上层中的事务	207	12.6.2	Eiffel 中的非正式规范	251
10.8	处理多个活动	208	12.7	按合同设计	252
10.8.1	控制多个任务	208	12.7.1	合同和继承	255
10.8.2	控制多个线程	208	12.7.2	减少错误检查代码	256
10.8.3	线程安全	209	12.7.3	履行合同	258
10.9	小结	212	12.7.4	应用程序防火墙	259
10.10	课外阅读	212	12.8	Java 中的非正式规范	259
10.11	复习题	212	12.8.1	使用注释编写合同文档	259
10.12	复习题答案	213	12.8.2	动态检查条件	260
<b>第 11 章</b>	<b>可重用的设计模式</b>	<b>214</b>	12.8.3	使用 RuntimeException 发出违反合同的信号	260
11.1	引言	214	12.8.4	外部系统	261
11.1.1	模式简史	214	12.8.5	启用和禁用动态检查	263
11.1.2	目前的软件模式	215	12.9	小结	264
11.2	模式模板	215	12.10	课外阅读	264
11.3	常见的设计模式	216	12.11	复习题	265
11.3.1	观察器模式	216	12.12	复习题答案	265
11.3.2	单一模式	220	<b>第 13 章</b>	<b>不间断的测试</b>	<b>266</b>
11.3.3	多重模式	223	13.1	引言	266
11.3.4	迭代器模式	224	13.2	测试术语	266
11.3.5	工厂方法和抽象工厂	226	13.2.1	黑盒子测试	267
11.3.6	状态模式	227	13.2.2	白盒子测试	268
11.3.7	门面模式	231	13.3	测试的类型	268
11.3.8	适配器模式	231	13.3.1	单元测试	269
11.3.9	策略模式和模板方法	233	13.3.2	完整性测试	269
11.3.10	次轻量级模式	235	13.3.3	Alpha 测试	269
11.3.11	复合模式	236	13.3.4	beta 测试	270
11.3.12	代理模式	239	13.3.5	用例测试	270
11.4	使用模式	240	13.3.6	组件测试	270
11.5	发现、合并和调整模式	241	13.3.7	构建测试	271
11.6	小结	243	13.3.8	负载测试	272
11.7	课外阅读	243	13.3.9	安装测试	273
<b>第 12 章</b>	<b>指定类的接口</b>	<b>244</b>	13.3.10	接受测试	273
12.1	引言	244	13.3.11	衰退测试	273
12.2	规范的定义	245	13.3.12	说明文档测试	274
12.3	正式规范	245	13.3.13	安全测试	274
12.4	非正式规范	247	13.3.14	衡量标准	274
12.5	动态检查	248	13.4	测试的自动化	275
12.6	面向对象的规范	250	13.5	准备测试	276
			13.6	测试策略	277

13.6.1	开发过程中的测试	277	B.4	系统设计	316
13.6.2	测试阶段中的测试	278	B.4.1	选择技术	316
13.6.3	发布后的测试	278	B.4.2	层图	317
13.7	测试的内容	278	B.4.3	层交互策略	318
13.8	测试驱动的开发	281	B.4.4	包	318
13.9	使用 JUnit 进行测试驱动 的开发示例	282	B.4.5	部署图	319
13.9.1	测试 Car 类	283	B.4.6	安全策略	320
13.9.2	实现 Car 类	284	B.4.7	并发策略	320
13.9.3	重新安排测试	286	B.5	子系统设计	320
13.9.4	为衰退测试创建测试套件	288	B.5.1	业务服务	321
13.9.5	测试 Across 方法	290	B.5.2	ServletsLayer 类图	321
13.9.6	完成 Store 类	290	B.5.3	ServletsLayer 的字段列表	321
13.10	小结	292	B.5.4	ServletsLayer 的消息列表	322
13.11	课外阅读	293	B.5.5	ServerLayer 类图	322
<b>附录 A</b>	<b>Ripple 小结</b>	<b>294</b>	B.5.6	ServerLayer 的字段列表	323
<b>附录 B</b>	<b>iCoot 案例分析</b>	<b>297</b>	B.5.7	ServerLayer 的消息列表	323
B.1	业务需求	297	B.5.8	BusinessLayer 类图	324
B.1.1	顾客的任务陈述	297	B.5.9	BusinessLayer 的字段列表	325
B.1.2	参与者列表	297	B.5.10	协议对象的类图	328
B.1.3	用例列表	298	B.5.11	数据库模式	329
B.1.4	用例的通信图	298	B.5.12	用户界面设计	330
B.1.5	用例的活动图	298	B.5.13	业务服务的实现	330
B.1.6	用例的细节	299	B.6	类的规范	342
B.2	系统需求	302	B.6.1	服务器类的规范	342
B.2.1	用户界面草图	302	B.6.2	业务逻辑类的规范	344
B.2.2	参与者列表	303	B.7	测试计划概述	346
B.2.3	用例列表	303	B.7.1	引言	346
B.2.4	用例图	304	B.7.2	螺旋式递增方式的作用	346
B.2.5	用例调查	304	B.7.3	非代码制品的测试	347
B.2.6	用例细节	305	B.7.4	代码的评估	347
B.2.7	辅助需求	308	B.7.5	测试驱动的开发	347
B.2.8	用例的优先级	308	B.7.6	断言	347
B.3	分析	308	B.7.7	测试阶段	347
B.3.1	类图	308	B.7.8	说明文档的测试	348
B.3.2	属性	309	B.7.9	构建测试	348
B.3.3	操作列表	309	B.7.10	测试建档和记录日志	348
B.3.4	预约的状态机	311	B.7.11	分阶段的测试活动	348
B.3.5	用例的实现	311	B.8	术语表	350
<b>附录 C</b>	<b>UML 表示法小结</b>	<b>356</b>			

# 第1章 入门

本书的目的是让读者基本掌握面向对象的软件开发所使用的过程和技术，并学会使用面向对象的技术编写计算机程序。统一建模语言(Unified Modeling Language, UML)是软件业的标准语言。

不管您是一位在校大学生，或是正在接受商业培训的工作人员，或是一位打算转向面向对象技术的、经验丰富的软件开发人员。无论如何，本书都会对您有所帮助。读者不需要有什么背景知识，本书也不打算讲述有关面向对象的所有知识，而是介绍该过程的基本组成部分，使您能更高效地完成工作。

本书的内容覆盖面比较广，但并不深邃，就到教会您编写代码为止。对如何编写代码的描述实际上就是选择特定的编程语言，为您选择合适的语言做准备。本书是掌握任何纯粹的、面向对象的编程语言的一门前导课程。

第一章主要讲述了本书的背景，概述了本书的内容和学习本书的方式。

## 1.1 背景

近来，新软件通常都是面向对象的。也就是说，使用称为“对象(object)”的抽象物体来编写软件。自然，商业软件开发要比简单地编写几行代码复杂得多：它需要调研商务需求、分析问题、设计解决方案等。在开发的每个阶段都应使用对象，因为对象减少了必须理解的信息量，加强了开发小组成员之间的交流。

## 1.2 编程简史

商业编程有许多发展阶段，“面向对象”是最新的一代：

- **机器码：**编程使用二进制数字。
- **汇编语言：**编程使用字母数字符号作为机器码的速记方式。汇编语言通过汇编程序(assembler)转换为机器码。
- **高级语言：**编程使用有高级结构(如类型、函数、循环和分支)的语言(例如 Fortran 和 COBOL)。高级语言(和以后的各种编程语言)使用编译程序转换为机器码。
- **结构化编程：**编程使用更简洁的高级语言(例如 Pascal、Modula 和 Ada)，它们的特点是程序员犯的错误更少，程序分解为子任务和子系统的方式更规范。
- **面向对象编程：**编程使用数据和函数的独立模块，这些模块对应于问题域中的概念，例如 Customer 或 ScrollBar。这种模块化使程序员的错误更少，并允许在不同的程序中重复使用代码。优秀的面向对象编程语言有 Java 和 Eiffel，因为它们的设计非常简洁，

很纯粹，还可以移植(可以在许多平台上使用)。其他语言有 Smalltalk、C#，以及最初是结构化语言、后来包含了面向对象扩展的任何语言(C++和 Pascal 的各种版本)。

读者可能听说过函数化编程(functional programming)和逻辑编程(logic programming)，目前它们已退出了市场。

目前，这些编程语言或多或少都在使用。选择使用哪种语言取决于使用场合、个人喜好和要解决的问题，例如，视频游戏需要的是速度，所以有时用汇编语言编写。

## 1.3 方法学

20世纪80年代，结构化编程语言开始流行，有经验的程序员开始描述如何控制整个软件开发过程，从任务陈述开始，一直到对已完成产品的维护，都要进行详细的描述。于是诞生了结构化方法学，例如 SSADM [Weaver 等 02]。方法学描述了开发小组编写高质量系统所应遵循的步骤，规定了产品应包含什么内容(文档、图表、代码等)，产品应采用什么形式(如目录、图标、编码样式)。

90年代，面向对象编程开始盛行，开发人员又发明了面向对象方法学，更适合进行面向对象的编程。早期的面向对象方法学有 Booch 方法[Booch 93]、Objectory[Jacobson 等, 92]和 OMT [Rumbaugh 等, 91]。近来，市场上的主导方法学是 IBM 提出的 Rational 统一过程(Rational Unified Process, RUP)，它归 IBM([www.rational.com](http://www.rational.com))所有。大致说来，RUP 是 Objectory、Booch 和 OMT 的综合。另一个日益流行的方法学是极限编程(extreme programming, XP)[Beck 99]，即所谓的“敏捷”方法学，在软件开发业中，敏捷意味着反映软件需求的变化或对问题的理解的变化。

本书使用的方法学是 Ripple(详见附录 A)，这是一种简化的方法学，基于被广泛接受的理论和实践。因此，读者不必学习完整方法学的复杂内容，本书也不会明确地告诉读者在每个阶段应做什么，而是允许读者自由发挥，推陈出新。

## 1.4 关于本书

为了避免混淆，本书不对结构化方法学和面向对象方法学进行详细的比较，而直接介绍面向对象的软件开发，就好像传统方法从来不存在一样。我们将讲述编写面向对象的大型软件时所需要掌握的所有内容(读者只需努力学习，积累经验即可)。

您不会用到以前已很熟悉的结构化技术，但不必担心，面向对象方法确实很有效，自从70年代开始人们就在研究它，在90年代面市以来，每天都有上百万的开发人员在使用它。

与以前一样，软件开发的目的是编码、编码、再编码。无论您的背景是什么，只要有编程经验就行。如果没有编程经验，学习本书是比较直截了当的方法——本书不会让您为了晦涩的术语和难以理解的技巧而头痛。但在开始之前，应熟悉基本的计算机概念，例如硬件、软件和网络。最起码要用高级语言编写过几百行代码。

### 1.4.1 内容概述

本书并没有介绍编写代码的所有细节，但常常需要使用代码段来演示。本书所有的代码段都是用 Java[Joy 等, 00]编写的，因为 Java 非常普及、纯粹、简单，也是免费的。每个代码段的含义都会在文中详细解释。如果您不喜欢使用 Java，这些代码段还能很容易地转换为其他语言，例如 C#，因为本书的代码没有使用 Java 的独特元素。用 Java 完成的所有任务都可以用其他任何一种语言完成。同样，系统设计的讨论也集中于 Java 技术，而不是.NET 技术，因为 Java 和.NET 支持类似的功能，选择 Java 还是.NET 完全是个人喜好。Java 系统设计的所有内容都可以以相同的方式用.NET 实现。

但要注意，为了便于演示，本书描述的类(例如 Iterator 和 List)并不匹配 Java 库中的类。使用本书肯定可以理解 Java 的工作方式，熟悉大多数语法，但本书不能替代单纯介绍 Java 语言的图书[Campione 等, 00]。所以，在编写 Java 代码时，最好手中有一本有关 Java 库的书，因为每个人都不可能记住上千个类的所有细节。

另一个难点是项目管理(规划、调度等问题)。本书没有讨论项目管理，因为我们主要论述的是技术问题，而不是人的因素。

在演示过程中使用的表示法称为统一建模语言(UML)[OMG 03a]，它是已被广泛接受的软件图标准。在 UML 的约定中，一些线条比另外一些粗，一些字符采用黑体或斜体。尽管只有斜体的内容比较重要，但这些约定很难用手工绘制出来(在纸或白板上)，所以在手工绘制时可以忽略其他约定。对于斜体的内容，还可以在适当的时候用其他方式替代。

UML 不能满足所有文档的需要，所以本书的一些文档取自 RUP。

### 1.4.2 案例分析

本书使用的案例分析是一个租用和预约系统，称为 Coot，由一家虚拟公司 Nowhere Cars 开发。因此，书中的许多例子都以某种方式使用汽车。大多数例子使用相同的应用程序域，在阅读本书时就不必在不同的域中来回转换。为了简单起见，大多数讨论都会提及 Coot 中给客户提供 Internet 功能的部门，这个简化的系统称为 iCoot。

由于有许多概念要解释，而且并不是每个概念都与汽车的租赁相关，所以本书的一些例子并不涉及 iCoot(它们更适合汽车销售员或汽车结构)。但是，每个重要的图表都取自完整的系统。

面向对象的新手常常希望有对案例的完整分析，所以附录 B 包含了 iCoot 的完整说明，供读者进一步学习。iCoot 文档虽然只是一个真实有效的软件，但很容易理解。

如果读者希望像学生一样实践本书描述的技术，可以在 [www.wiley.com/go/odocherty](http://www.wiley.com/go/odocherty) 上找到一组 AQS(Automated Quiz System，自动组卷系统)练习题。AQS 是一个提取多项选择题的在线工具。这些练习题按照本书的主要章节来安排，读者在阅读的过程中就可以完成它们。教师如果想把本书用作大学教材或商务培训教材，在注册后就可以获得 AQS 练习题的答案。

### 1.4.3 导航

在阅读完第 1 章后，就可以用很直接的方式按顺序导航和通读所有的章节了。

或者，如果已经熟悉面向对象的概念和术语，就可以跳过第 2、3 和 4 章，直接阅读第 5 章。如果对面向对象领域非常陌生，就应先阅读第 2 章，但可以把第 3 和 4 章放在后面。

如果对 Ripple 的概述和方法学不感兴趣，而是喜欢直接进入细节，就可以跳过第 5 章，直接阅读第 6 章。

其余章节的结构比较严谨，是典型的软件开发过程，所以不应跳读。第 11 章最好当作主要内容来阅读，但由于其中的许多内容比较高级，也可以留到以后学习。

附录 A 可以用作自我测验，也可以在尝试案例分析时用作参考。

附录 B 包含 iCoot 案例分析的所有内容，其顺序按照一般开发过程来组织。使用这些内容和主要的开发章节，就可以了解 iCoot 系统的开发过程。附录 B 还包含 iCoot 项目的术语表，在 iCoot 的开发过程中，这个术语表会不断地更新。使用它可以了解一般术语表的内容，和查找 iCoot 术语定义的方式。

如果需要学习如何绘制 UML 图，就可以参考附录 C。

# **第 I 部分**

## **设置场景**

**第 2 章 对象的概念**

**第 3 章 继承**

**第 4 章 类型系统**

**第 5 章 软件开发的方法学**



# 第 2 章 对象的概念

本章介绍的概念都来自于面向对象编程(object-oriented programming)。一般说来，编程语言都是在帮助使用它们的方法学之前发明的，面向对象的概念非常有意思，因为面向对象的开发允许用真实世界中的术语思考问题，而不是局限于计算机的需要。

## 学习目标：

- 理解软件对象的含义
- 理解对象使用消息进行交流以完成任务的方式
- 理解不再需要某对象时会发生什么(垃圾收集)
- 理解类的含义
- 理解重用代码的方式

## 2.1 引言

面向对象(做事的方式)的基本概念相当容易理解和应用。Alan Kay 是 Smalltalk 的发明者，他早在 1968 年开始就为“所有年龄段的孩子的个人计算机”[Kay 72]工作。他的目标是孩子，所以基本概念非常简单。

为什么所有的事情都与对象有关？开发人员没有更好的理由改变软件开发的根本吗？在早期阶段，使用对象的一些理由看起来相当模糊，尤其是对以前的技术(结构化编程和结构化方法学)没有什么经验时，这些理由就更难理解了。出现(或诞生)面向对象方法，是因为人们发现，在指定的时间和预算内生产出高质量的系统越来越难了，尤其是涉及许多人的大型系统。

一旦阅读完本书，就会觉得下面给出的理由很有道理，而且会对大部分理由持赞同态度。这里只列出了面向对象的一般理由：

- 对象更便于人们理解：这是因为对象派生自我们试图自动化的业务，而不是派生自受计算机上的过程或数据存储需求过早影响的业务。例如，在银行系统中，是根据银行账户、银行出纳员和顾客来编程，而不是直接深入账户记录、取款和存款过程、贷款资格的算法等。
- 专业人员可以更好地交流。随着时间的推移，软件业已经形成了职业阶梯，新人可以积累知识和经验，一步步向上爬。一般，第一步是程序员：修改其他人编写的代码中的错误。第二步是高级程序员：自己编写代码。第三步是设计师：根据需要决定编写什么代码，最后一步是分析师：与客户交流，了解他们的需求，然后总结出最终完成的系统必须能做的工作。

这样的职业阶梯本身并不是什么坏事。但每位专业人士都希望学习一套全新的概念和技

术，用适合其专业的表示法表述他们的观点。这意味着，处于不同位置的人对任务的理解有很大的不同，当文档沿着职业阶梯向下传递，而不是向上传递时，这种差异会更严重，所以在阅读文档时，最好无需理解用于制作文档的技术。这就产生了“穿墙”症状：分析师制作了大量的文档，把它们交给设计师，然后就走了。设计师在经过数周的努力后，使用完全不同的技术，制作了更多的文档，把它们交给程序员，程序员再次从这些文档开始……

在面向对象的方法中，每个人都使用相同的概念和表示法。而且，要处理的概念和表示法都比较少。

- 数据和过程不是人为分离的。在传统方法中，需要存储的数据早早地与操作这些数据的算法分离开来，算法是独立开发的。从需要访问这些数据的过程来看，这会导致数据的格式不合适，或者数据放置的位置不方便。而利用面向对象的开发方式，数据和过程一起放在容易管理的小软件包中，数据从来不与算法分开。最后得到的代码不太复杂，对客户需求的变化也不太敏感。
- 代码更容易重用：在传统方法中，是从要解决的问题开始，利用问题来驱动整个开发，最后得到当前问题的单个解决方案，但明天总是会有另一个问题要解决。无论新问题和已解决的旧问题多么接近，都不能分解单个系统，对它进行调整，因为一个小问题就会影响系统的每个部分。

在面向对象的开发方式中，我们总是要在类似的系统中寻找有用的对象。即使新系统的区别非常小，也可以修改已有的代码，因为对象就像是智力拼图玩具中的各块拼图。如果一块拼图变化了，就会影响临近的几块拼图，但其他的拼图保持不变。

在建立面向对象的系统时，在考虑自己编写代码之前，都应寻找已有的对象(由自己、同事或第三方编写)。就像一位前辈所说：“面向对象的编程并不在于编写代码。”

- 面向对象是成熟的，并已得到了证明：这不是一个新时尚。编程概念是在 20 世纪 60 年代后期产生的，而方法学的产生至少晚了十年。在软件、数据库和网络的领域中应用对象目前已得到了很好的认可。

一旦阅读了整本书，就可以再复习一下这个列表，看看自己是否完全理解和赞同这些理由。

## 2.2 什么是对象

对象(object)是一件事、一个实体、一个名词，可以获得的某种东西，可以想像有自己的标识的任何东西。一些对象是活的，一些对象不是。现实世界中的例子有汽车、人、房子、桌子、狗、植物、支票簿或雨衣。

所有的对象都有属性(attribute)，例如汽车有厂家、型号、颜色和价格。狗有种类、年龄、颜色和喜欢的玩具。对象还有行为(behavior)：汽车可以从一个地方移动到另一个地方，狗会吠。

在面向对象的软件中，真实世界中的对象会转化为代码。在编程术语中，对象是独立的模块，有自己的知识和行为(也可以说它们有自己的数据和进程)。把软件对象看作机器人、动物或人是很常见的：每个对象都有一定的知识，表现为属性；它知道如何为程序的其他部分执行某些操作。例如，Person 对象知道它的头衔、姓名、出生日期和地址，它还可以改名、搬到新地址、告诉我们它有多大等。

在为 Person 对象编写代码时集中一个人的特性，就可以想像出系统的其余部分，这会使编程比其他方式更简单(还有助于从真实世界中的概念开始)。如果 Person 以后需要知道其身高，就可以把这个知识(和相关的行为)直接添加到 Person 代码中。在系统的其余部分中，只有需要使用身高属性的代码才需要修改，其他代码都保持不变。改变的简单性和本地化是面向对象软件的重要特性。

把生物看成某类机器人是很简单的，但把没有生命的物体看成有行为的对象就有点怪异。我们一般不认为电视能改变其价格或给自己一个新的广告。但是，在面向对象的软件中，这就是我们需要做的工作。其关键是，如果电视不做这些工作，系统的其他部分就要做。这就会把电视的特性泄露给代码的其他部分，丧失了我们一开始追求的简单性和本地化(又回到“做事的旧方式”了)。不要被面向对象开发中常见的把软件对象想像成人的理论、把人的特性赋予没有生命的对象或动物吓住。

图 2-1 是一些适合用作软件对象的真实物体。你能想出其他对象吗？能想出不适合用作对象的物体吗？这是一个很难的问题：答案必然是“不能”。在某种情况下，几乎所有的物体都可以用作对象。不适合用作对象的物体是合并了几个概念的物体，例如银行账户对象所具备的某些属性和行为也属于银行职员。记住，真实世界中的某些概念对应于程序中的特定概念。

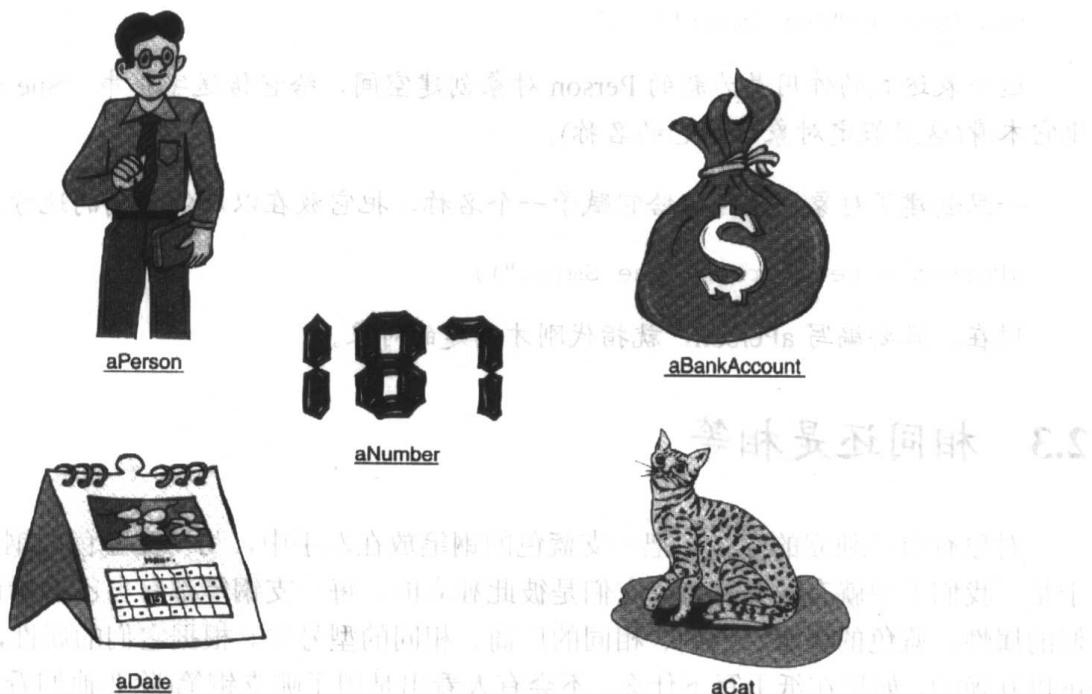


图 2-1 真实世界中的对象

在进一步讨论之前，先要注意，我们并不尽力模拟真实世界，这太困难了。我们只是尽力确保软件受真实世界中的概念影响，使软件更容易开发和改变。系统和计算机的需求也是很重要的考虑因素。一些开发人员不喜欢真实世界和软件过于接近，但是，为医院开发的面向对象系统如果不包含 Patient 对象，就没有什么用。

我们不可能编写出理想的 Person 对象或其他完美的对象。真实世界中的对象可以应用的特性和功能太多了，如果把它们全包括进来，就不可能在系统中编写出其他对象。

在一般的程序中，不需要真实世界中的对象的大多数方面，因为软件系统只是解决一个方面的问题。例如，银行系统对客户的年龄和收入感兴趣，对他们的鞋尺寸或喜欢的颜色不感兴趣。编写对许多系统都有用的对象是很合理的，尤其是编程中已得到很好理解的领域：例如，所有带有用户界面的系统都能使用相同的“可滚动列表”对象。其技巧是一开始就考虑要处理的业务，弄清楚“如果我在这个业务领域工作，‘人’对我意味着什么？是客户、员工、患者，还是其他人？”优秀的软件开发人员首先会为业务建模。

模型(model)是问题域或所提出的解决方案的表示方式，用于交流或思考真实的事务。建模可以增进了解，避免潜在的问题。考虑建筑师为新音乐厅建立的模型：有了它，建筑师就可以说“这就是新音乐厅完工后的样子”，该模型有助于他们提出新点子，例如“我觉得屋顶还要更倾斜一些”。即使还没有开工，也可以通过模型了解许多事情。许多软件开发都涉及到创建和细化模型，而不是删掉代码。

### 实践 1

下面考虑如何在面向对象的编程语言中编写新对象。纯面向对象语言一般提供了一个创建表达式。下面是 Java 中的创建表达式：

```
new Person("Sue Smith")
```

这个表达式的作用是为新的 Person 对象创建空间，给它传送字符串 “Sue Smith”，以初始化它本身(这里假定对象记录它的名称)。

一旦创建了对象，就可以给它赋予一个名称，把它放在以后能找到的地方，例如：

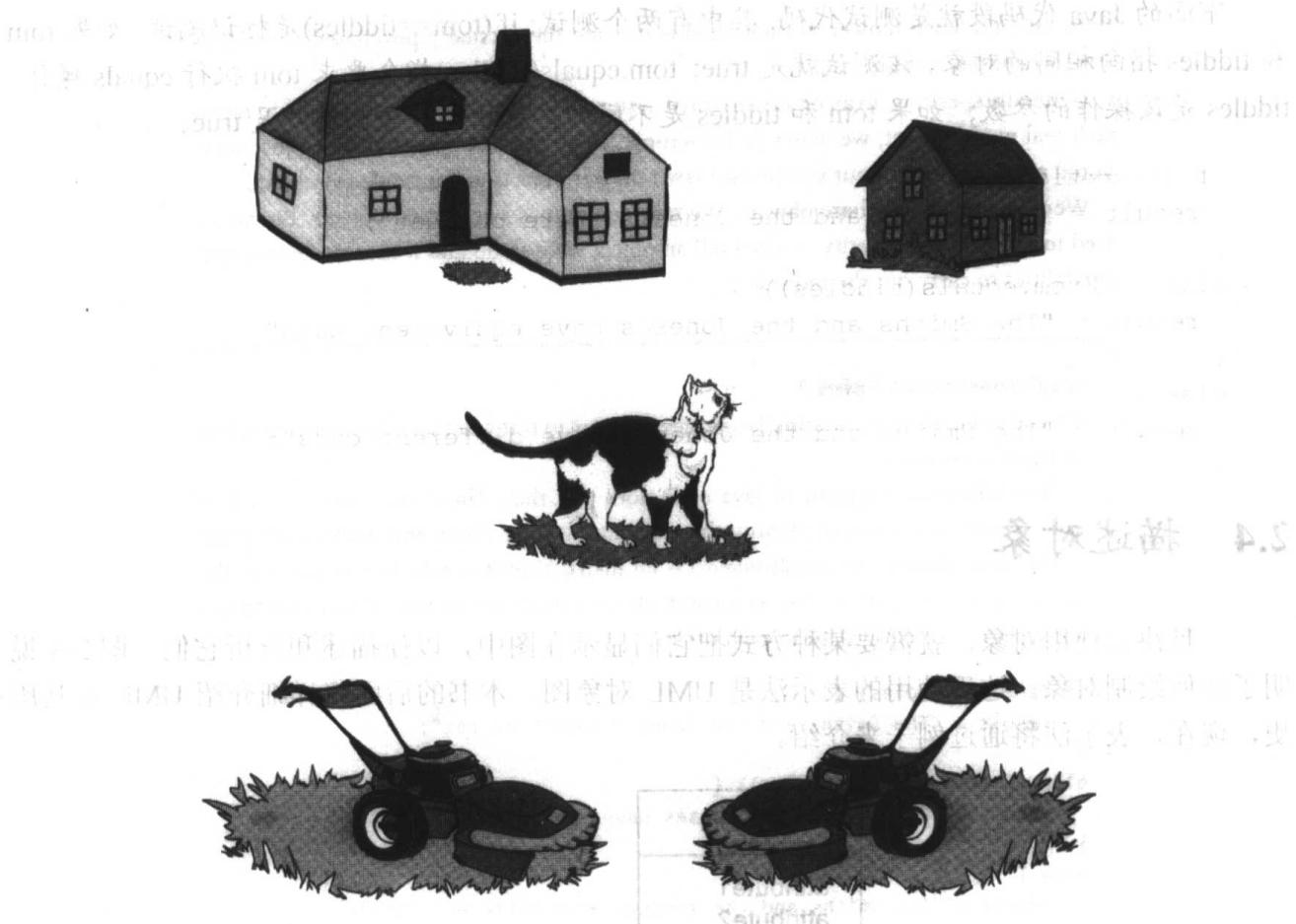
```
aPerson = new Person("Sue Smith");
```

现在，只要编写 aPerson，就指代刚才创建的对象。

## 2.3 相同还是相等

对象有自己独立的存在。把一支蓝色的钢笔放在左手中，另一支蓝色的钢笔放在右手中，于是，我们手中就有两支钢笔，它们是彼此独立的，每一支钢笔都有自己的标识。但它们有类似的属性：蓝色的墨水、半满、相同的厂商、相同的型号等。根据它们的属性，这两支钢笔是可以互换的，如果在纸上写下什么，不会有人看出是用了哪支钢笔(除非他们看到写字的过程)。钢笔是相同的，但它们不是一支笔。在软件和真实世界中，这是一个非常重要的区分。

举另一个例子，考虑图 2-2 中的情形。在 Acacia 大街有两个家庭 Smith 和 Jonese。Smith 住在 4 号，Jonese 住在 7 号。这两个家庭在割草机方面有类似的爱好，实际上，他们都拥有一台 GrassMaster 75 割草机，都是在新型号推出的第一天购买的。割草机非常相似，如果有人在晚上交换了这两台割草机，Smith 和 Jonese 是不会发现的。



除了割草机外，Smith 家还有一只猫 Tom，Tom 是一只圆圆的很友善的猫，已经三岁了，它喜欢的消遣是在花园里抓老鼠。Jones 家也有一只猫 Tiddles，Tiddles 也是一只圆圆的很友好的猫，也已三岁，它喜欢的消遣是追逐毛线球。拜访 Smith 家和 Jones 家的人会注意到 Tom 和 Tiddles 很相似，以为 Tom 和 Tiddles 是同一只猫就没有什么奇怪了，它们喂食的次数非常多，所以长得圆圆的。

在这个例子中，有两台割草机和一只猫。尽管割草机有不同的标识，这些标识记录在其机体的序列号牌上，但它们是相同的，因为它们有相同的属性。猫也有标识，它甚至可以有自己的名字(Me 或 Hfrrr)。猫和割草机的区别是，猫是共享的，而割草机不是。人、物体或动物很少需要与他的标识关联起来：猫不会考虑它与其他猫是否有区别，割草机不需要知道它是割草机才能割草。家人不需要知道 Hfrrr 的喂食次数是其他猫的两倍，只要它时时咕噜咕噜叫即可，家人也不需要知道工棚里的割草机就是他们购买的那一台，只要在家人需要使用它时，它在工棚里即可。

一般说来，在面向对象的系统中，如果使用一个软件对象表示每个真实世界中的物体，就不会犯错。所以，在上面的例子中，应能在系统中找到两个割草机对象和一个猫对象。

有时还可以共享对象，互换相同的对象。但很少需要担心其标识：只要告诉对象应该做什么，它就会使用自己的知识和能力来响应请求。

## 实践 2

面向对象的编程语言允许在需要时测试对象是相同还是相等。

下面的 Java 代码段就是测试代码。其中有两个测试：if (`tom==tiddles`) 是标识测试，如果 `tom` 和 `tiddles` 指向相同的对象，该测试就是 true；`tom.equals(tiddles)` 指令要求 `tom` 执行 equals 操作，`tiddles` 是该操作的参数；如果 `tom` 和 `tiddles` 是不同的，但相等，该测试就是 true。

```
if (tom==tiddles) {  
    result = "The Smiths and the Joneses share one cat";  
}  
else if (tom.equals(tiddles)) {  
    result = "The Smiths and the Joneses have equivalent cats";  
}  
else {  
    result = "The Smiths and the Joneses have different cats";  
}
```

## 2.4 描述对象

一旦决定使用对象，就需要某种方式把它们显示在图中，以便描述和分析它们。图 2-3 说明了如何绘制对象。这里使用的表示法是 UML 对象图，本书的后面将详细介绍 UML 及其历史，现在，表示法将通过例子来介绍。

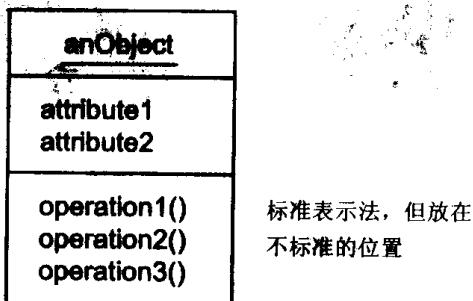


图 2-3 一个对象

方框的三个部分显示了对象的名称(加了下划线)、其属性(知识)和操作(行为)。(在对象图中显示操作并不是合法的 UML，但可以达到我们的目的)。操作名称旁边的圆括号表示需要的参数：即使这里显示的操作没有参数，也最好包含括号，以使操作名称与属性名称明显地区分开(当省略方框中的一个或多个部分时，给操作加上圆括号将非常重要)。

属性隐藏在对象中，访问它们的惟一方式是通过操作来访问。这与真实情况非常类似：我们对电视的“转换频道”操作比较感兴趣，而对进行这种转换的电子电路不感兴趣。

下面举一个咖啡机对象的例子。首先，确定咖啡机需要什么操作：

- 显示饮料种类
- 选择饮料种类
- 收款
- 出货

接着，考虑咖啡机需要知道什么，才能执行这些操作：

- 可用的饮料种类
- 饮料的价格
- 饮料的配方

设计好咖啡机对象后，就可以在对象图中记录这些内容，如图 2-4 所示。

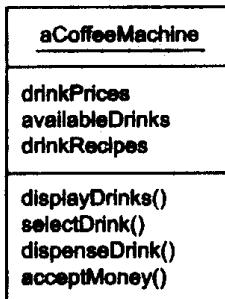


图 2-4 咖啡机对象

## 2.5 封装

封装是指对象在其操作中隐藏属性(对象把属性密封在一个盒子中，其操作放在盒子的边缘)。隐藏的属性称为私有属性。一些编程语言(如 Smalltalk)自动把属性设置为私有属性，一些语言(如 Java)让程序员决定属性的私有性。

封装是编程语言防止程序员相互干扰的一种方式：如果程序员可以绕过操作，就会依赖用于表示对象知识的属性。这会加大将来改变对象内部表示的难度，因为必须找出直接访问属性的所有代码，并修改它们。没有封装，就会丧失简单性和本地化。

以表示圆的一个对象来作为封装的一个例子。圆的操作应能计算出其半径、直径、面积和周长。需要存储什么属性才能支持这些操作？我们可以存储半径，按照需要计算出其他属性，或者存储直径，按照需要计算出其他属性。实际上，只要存储这四个属性中的任一个，就可以按照需要计算出其他三个属性(选择哪个属性取决于个人喜好，或推断圆的正常使用方式)。

假定选择存储直径。要访问直径的程序员都会选择存储直径属性，而不是通过“获取直径”的操作来访问。如果在软件的后续版本中，要存储的是半径，就必须找出系统中直接访问直径的所有代码，并更正它们(在这个过程中会引入错误)。而有了封装，就不会有问题。

理解封装的另一种方式是想像对象是谦恭的。如果要从同事那里借一些钱，在小卖部处购买食物，就不能抢夺同事的钱包，大翻一通，看看里面是否有足够的钱。而应询问他们是否可以借你一些钱，他们就会自己翻钱包。

## 2.6 关联和聚合

对象都不是孤立的。所有的对象都与其他对象有直接或间接的联系，其联系或强或弱。对象彼此联系起来，就会更强大。这种联系允许我们在对象中浏览，找出额外的信息和行为。例如，如果处理表示 Freda Bloggs 的 Customer 对象，要给 Freda 送一封信，就需要知道 Freda 住在 Acer 路 42 号。我们希望把地址信息存储在某个 Address 对象中，这样就可以查找 Customer 对象和 Address 对象之间的联系，确定把信送到什么地方。

在用对象建模时，可以用两种方式连接对象：关联或聚合。有时很难判定两者的区别，但这里有一些规则：

- 关联是一种弱连接：对象可以是小组或家庭的一部分，但它们不完全相互依赖。例如，汽车、司机、一个乘客和另一个乘客。当司机和两个乘客在汽车中时，他们就是关联

的：他们都朝着同一个方向前进，占用相同的空间等。但这种关联是很松散的：司机可以让一个乘客下车，让他走自己的路，这样该乘客就与其他对象没有关联了。图 2-5 显示了如何在对象图中绘制关联：这里省略了属性和操作，以强调结构。

- 聚合表示把对象放在一起，变成一个更大的对象。所生成的大对象通常形成了聚合：例如，微波炉由柜子、门、指示板、按钮、马达、玻璃盘、磁电管等组成。聚合常常形成了“部分-整体(part-whole)”层次结构，它隐含了较大的依赖性，至少是整体对部分的依赖；例如，如果把磁电管从微波炉中取出来，它还是一个磁电管，但微波炉没有磁电管就没有用了，因为它不能烹饪任何食物了。

图 2-6 说明了如何把房子绘制为聚合关系。为了强调这种连接和关联的区别，这里在“整体”端加上了一个白色的菱形框。

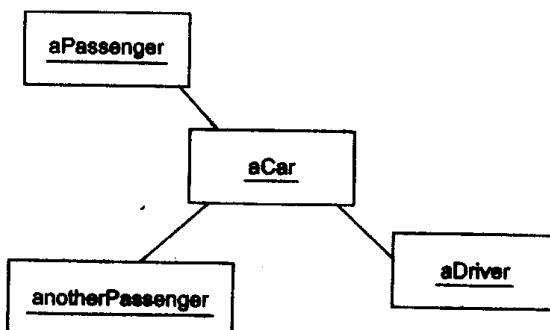


图 2-5 关联

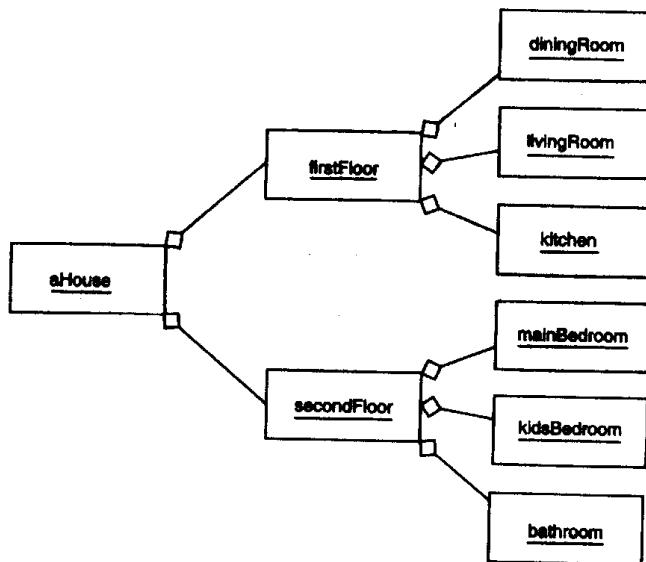


图 2-6 聚合

如前所述，关联和聚合的区别是很微妙的。“如果去除其中一个对象，会发生什么”的测试很有帮助，但它并不总是能解决问题，常常需要仔细的思考和一定的经验。

我们常常需要在关联和聚合之间选择，因为这个选择会影响设计软件的方式。下面是一些例子：

- 朋友：我们希望朋友是关联性的：把朋友聚集在一起，变成一个更大的朋友是没有意义的；朋友会随着时间的流逝离开或回来。

- 电视机中的组件：这是比较容易理解的连接方式，因为它是经典的“部分-整体”层次结构：把按钮和旋钮放在一起，制作出控制面板；把屏幕、电子枪和磁性卷放在一起，做出显像管；把这些小部件组装起来，就会得到较大的组件，再把这些组件放在一个大柜子中，加上后盖。最终用户就看到了一台电视机对象：如果一个组件失败，它们就不再是电视机，而只是一堆没用的垃圾。
- 书架上的书：书架不需要书，就可以成为书架，它只是放置书的一个地方而已。反过来，书放在书架上，就肯定与书架相关联(如果移动书架，书也会移动，如果书架散了，书就会掉下来)。这是典型的关联关系。
- 办公室中的窗户：窗户是办公室的一部分。尽管可以移走已打破的窗户，让办公室少一个窗户，但人们仍希望不久之后就换上一个新窗户，这是可能的聚合关系。

### 练习 1

下面请读者试着回答问题。下面哪些是关联，哪些是聚合？

- (1) 街道上的房子
- (2) 书中的书页
- (3) 交响乐中的音符
- (4) 家庭娱乐系统中的组件(电视机、VCR、大型录音机、扩音器、游戏控制台)

## 2.7 图和树

除了关联和聚合之外，对象还有树(tree)和图(graph)。树是层次结构的另一个名称。如果重新绘制图 2-6 中的对象图，如图 2-7 所示，就会明白为什么聚合常常表示为树(它没有树干，但非常紧密)。为了便于查看，程序员通常把树颠倒过来，如图 2-8 所示。

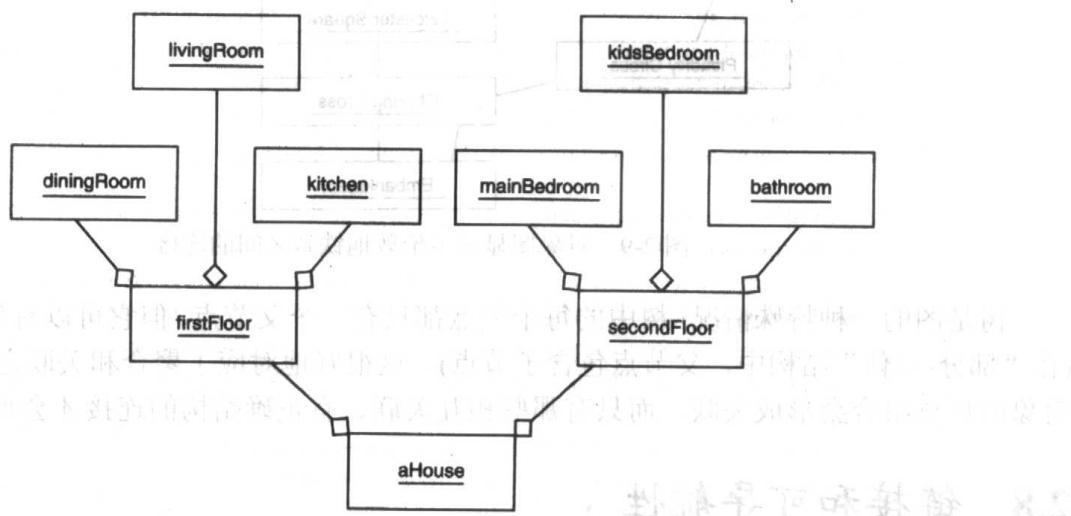


图 2-7 以树表示聚合

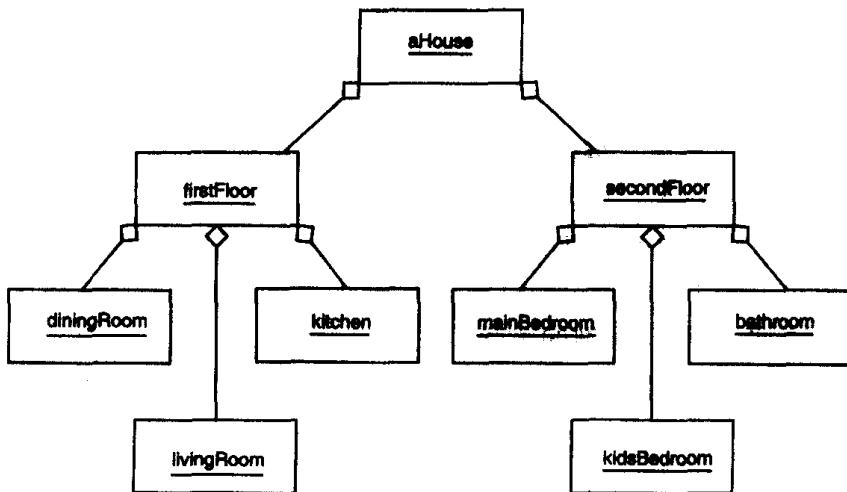


图 2-8 倒置的树

图是对象组之间连接的一个任意集合。关联中的对象常常会形成图，如图 2-5 中的汽车例子。另一个例子包含更有趣的连接，即伦敦的地铁系统。图 2-9 表示伦敦地铁系统的一部分，人们可以从一站(节点， node)到达其他任何一站，通常中间要经过几站。

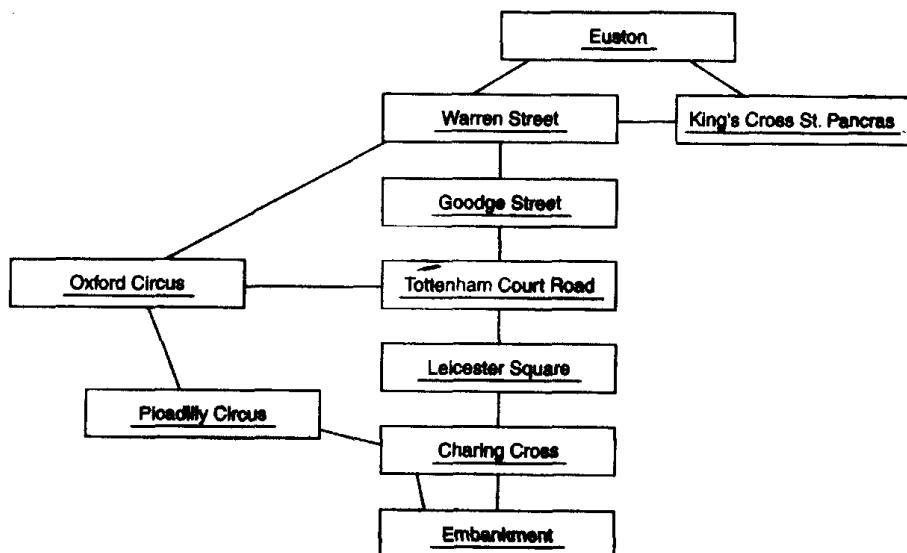


图 2-9 对象图显示了伦敦地铁站之间的连接

树是图的一种特殊情况：树中的每个节点都只有一个父节点，但它可以有任意多个子节点。(在“部分-整体”结构中，父节点包含子节点)。这很好地对应了聚合和关联之间的区别：连接对象的任意组合会形成关联，而只有那些相互关联、有正确结构的连接才会形成树。

## 2.8 链接和可导航性

前面对象图中的连接都称为链接(link)。如果要说明一个对象知道另一个对象在哪里，就可以加上箭头，如图 2-10 所示。该图说明，Customer 链接了 Address 和 String。(String 在编程中很有用，它由一系列字符组成)。

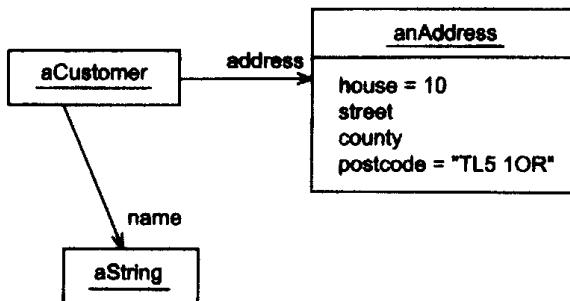


图 2-10 可导航的链接

每个链接都可以看作一个属性：标签或角色，表示属性的名称。因此可以说，`aCustomer` 的属性 `address` 把它链接到 `Address` 对象上，属性 `name` 把它链接到表示其名称的 `String` 对象上。箭头表示可导航性，即知道另一个对象在哪里。因为 `Customer` 端没有箭头，这表示 `String` 不知道它与 `aCustomer` 的关联。可导航的链接在面向对象的程序中常常称为指针(指针是对象在内存中的地址，以便在需要时能找到它)。

图 2-10 中的链接比前面的连接(没有任何箭头)有更多的内容。对象图的一个优点是，它允许显示模型中任意级别的细节，这可以增进对对象的理解，对我们所做的工作更有信心。简单的值显示为属性，重要的对象显示为链接的盒子，中间值根据需要显示为属性或链接的盒子。

图 2-10 还显示了其他信息，这些信息读者肯定可以理解和接受：链接的对象和属性都指定了名称，还显示了一些字面值，例如数字 10 和字符串 TL5 1OR。这里给对象、属性和角色使用的命名约定是很常见的：用一两个描述性的单词，中间没有空格，从第一个单词开始的每个单词的首字母大写。至于字面值，我们都知道如何写数字，如何把字符放在双引号中。

在一些地方，对象会扩展，而在其他地方，对象不会扩展。例如，`aCustomer` 的 `name` 属性显示为一个独立的对象，而 `anAddress` 的 `street` 和 `country` 属性甚至没有值。

所有图的关键都是显示所需要的细节，来达到我们的目的。不要仅因为别人画的图与自己的不同，就认为自己的图是错误的。一般在开发过程中，都必须处理越来越多的信息，但很少在一个地方显示所有的内容(否则，事情就变得混乱、乏味)。

对于值，最后要注意的是，尽管所有的物体都可以建模为对象，但不需要为不重要的值建立对象。例如，数字 10 可以看作一个对象：它的内部数据表示 10，其操作有“加上另一个数字”和“乘以另一个数字”。但是，在许多面向对象的编程语言中，像数字这样的简单值可以用不同的方式处理：只能把它们用作属性值，它们没有标识，不能分解它们。

## 2.9 消息

每个对象都至少与另一个对象联系，孤立的对象对任何人来说都没有用。对象一旦建立了联系，就可以协作，执行更复杂的任务。对象在协作时要相互发送消息，如图 2-11 所示。消息显示在实线箭头的旁边，说明消息的发送方向，回应显示在蝌蚪符号的旁边，表示数据的移动。

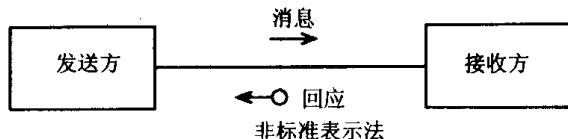


图 2-11 使用消息进行协作

图 2-11 是一个 UML 通信图。通信图看起来很像对象图，但链接没有方向，对象名称没有加下划线。通过通常的方式无法在通信图中显示回应，所以这里使用了蝌蚪符号，这是长期存在的一个约定。理想情况下，还应显示序号，但这里省略了，因为要涉及 UML 编号方案。

消息的内容可以是“现在几点？”、“启动引擎”、“你叫什么名字？”等，如图 2-12 所示。可以看出，接收对象可以提供回应，也可以不提供；“现在几点？”和“你叫什么名字？”应提供回应；而“启动引擎”不需要回应。

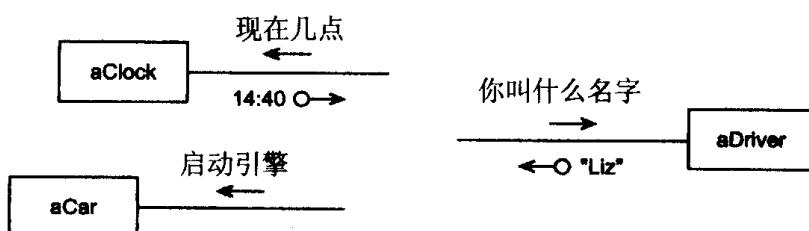


图 2-12 一些示例消息

如前所述，对象是谦恭的，当接收到消息时，肯定会处理该请求。这样，发送对象就不需要处理消息被拒绝的情形。实际上，尽管接收对象的意图是好的，但仍不能执行一些请求。考虑请求失败的原因，参见表 2-1。

表 2-1 请求失败的原因

问 题	例 子	解 决 方 案
发送者不应该发送消息	给企鹅发送 fly 消息	编译器应检查出这类错误的大多数，在测试和维护过程中应检查出其他错误
发送者出错	当微波炉中没有食物时，让微波炉开始烹饪	编译器可以提供帮助，但大多数情况下依赖于好的设计、编程、测试和维护
接收者出错	假定 $2+2=5$	编译器可以提供帮助，但大多数情况下依赖于好的设计、编程、测试和维护
接收者遇到一个可预测但很少见的问题	当电梯中的人过多时，命令电梯“上升”	异常处理使用编程语言的功能把正常操作和非正常操作分开
计算机不能完成它应完成的任务	把桌子上的计算机放倒；宇宙光穿透了中央处理器；把内部位从 0 改为 1；操作系统错误；...	软件开发人员除了向用户界面报告问题或把问题写入日志文件，“优雅地失败”之外，干不了别的
人为错误	对象给磁盘写信息时取出了磁盘	异常处理使用编程语言的功能把正常操作和非正常操作分开

有时，我们不允许失败：如果自动驾驶的飞机因软件错误而失事，我们会相当失望。在这种情况下确保成功有一个专门的术语——**件可靠性**。为了说明这一点，下面是可靠性的一个策略：在飞机上安装三台计算机，让它们确定下一步的任务；如果一台计算机说“向左飞”，但其他两台计算机说“向右飞”，飞机就会向右飞。

了解了消息失败的原因后，就应暂时不考虑问题，而是假定消息总是会成功(这是一个很好的方法，详见第 12 章)。

## 2.10 启动操作

在软件对象收到消息时，就会执行一些代码。每段代码都是一个操作。换言之，消息启动了操作。在 UML 中，可以显示发送者给接收者发送的消息，或者接收者执行的操作，也可以显示两者。

除了回应之外，消息还可以带参数(**parameter**，也称为变元 **argument**)。参数是一个对象或简单的值，接收者用它来满足请求。例如，可以给 Person 对象发送消息“你的身高是多少米？”，一分钟后发送另一个消息“你的身高是多少英寸？”。在这个例子中，“你的身高是多少”就是消息，而“米”和“英寸”就是参数。参数显示在括号中，放在消息的后面，例如 `getHeight(meters)` 或 `getHeight(aUnit)`。如果有好几个参数，可以用逗号把它们分开。

还需要指定哪个对象接收消息，这里说明如何在 Java 中指定接收消息的对象，它使用句点把接收者和消息分隔开：

```
aPerson.getHeight(aUnit)
```

有时，不知道自己设计的消息是应让对象执行操作，还是从对象那里提取一些信息，或者两者均有。消息样式的一个规则是“消息应是一个问题，或者一个命令，但不能两者都是”，这可以避免许多问题。

提出问题的消息要求对象提供一些信息，所以总是有回应。问题不应改变对象的属性(或者与它连接的任何对象的属性)。提出问题的信息如下“你有什么烤肉”，“现在几点”。我们不会希望仅因为我们问了这个问题，柜台上才有更多的烤肉。同样，我们也不会希望时钟上的时间仅因为我们看了时钟才变化。

命令消息告诉对象执行某个操作——这次对象不需要提供回应。命令可以是告诉银行账户“取 100 欧元”，告诉微波炉“停机”。如果发出了合理的命令，对象就会执行它，所以不需要反馈任何信息。命令会改变接收对象或者与它联系的其他对象。

问题和命令的消息都是有用的，但它们都是高级技术，这里不举例子了。

## 2.11 协作示例

为了强化协作的概念，下面看一个较大的例子。首先从人的观点来看，再从软件的角度来分析。这个例子是“从面包店购买一条面包”。从人的观点来看：

顾客走进面包店，问面包师有什么种类的面包。面包师看看柜台，告诉顾客有两种白面包和一种全麦面包。顾客要买全麦面包。现在开始交易：面包师包好面包，交给顾客，并要求顾客付款，顾客给面包师付钱，面包师给顾客找零钱。顾客满意地离开了。

这个协作可以用通信图表示出来，如图 2-13 所示。为了简单起见，省略了顾客进出面包店的过程，在每个消息的旁边还显示了消息的方向，使图形更紧凑。

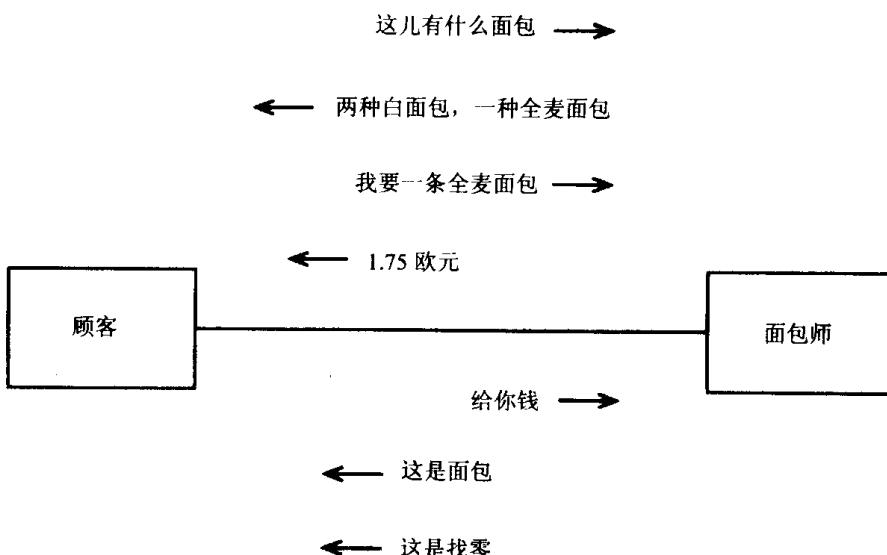


图 2-13 购买一条面包

这类协作很容易用纯粹的面向对象语言编码。但是，大多数面向对象的设计人员不这么做。稍微考虑一下顾客和面包师，主要问题是有一个复杂的双向交互，编程计算机很难添加这种真实世界的复杂性。另外，顾客依赖与面包师的交互，面包师依赖与顾客的交互，改变一个对象就意味着改变另一个对象，这是一个维护难题。

与面包师和顾客对象的设计问题相对的还有面包师与柜台的交互方式：面包师要给柜台发送一个消息，接收回应，但柜台不能给面包师发送消息。柜台是一个被动的对象，它只是立在那里等待别人使用它。柜台仍在做自己的工作，面包师可以取她需要的东西，但这种交互是单向的，复杂性较低，也容易改变。这种交互称为客户-提供者样式：面包师是客户，柜台提供服务。提供者对象的另一个优点是，它们在其他环境下的用处更大，因为它们独立于客户——它们是可以使用的，这是面向对象开发的一个主要目标。

略微思考一下，进行一些实践，通常可以把双向协作转换为客户与提供者的交互。为了达到这个目的，有两个机制：消息回应和消息参数。这两个机制图 2-13 都没有使用。图 2-14 把顾客和面包师之间的交互显示为纯粹的客户-提供者实现。



图 2-14 购买一条面包，客户-提供者样式

客户-提供者不是唯一的方式，但它是最常见的方式，适用于大多数情况。

## 2.12 面向对象程序的工作原理

面向对象的程序在工作时，要创建对象，把它们连接在一起，让它们彼此发送消息，相互协作。但谁启动这个过程？谁创建第一个对象？谁发送第一个消息？为了解决这些问题，面向对象的程序必须有一个入口点(entry point)。例如，Java 在启动程序时，要在用户指定的对象上找到 main 操作，执行 main 操作中的所有指令，当 main 操作结束时，程序就停止。

main 中的每个指令都可以创建对象，把对象连接在一起，或者给对象发送消息。对象发送消息后，接收消息的对象就会执行操作。这个操作也可以创建对象，把对象连接在一起，或者给对象发送消息。这样，该机制就可以完成我们想完成的任何任务。

图 2-15 显示了一个面向对象的程序。main 操作中一般没有很多代码，大多数动作都在其他对象的操作中。如图 2-15 所示，对象给自己发送消息是有效的：我们也可以问自己一个问题“我昨天干了什么”。

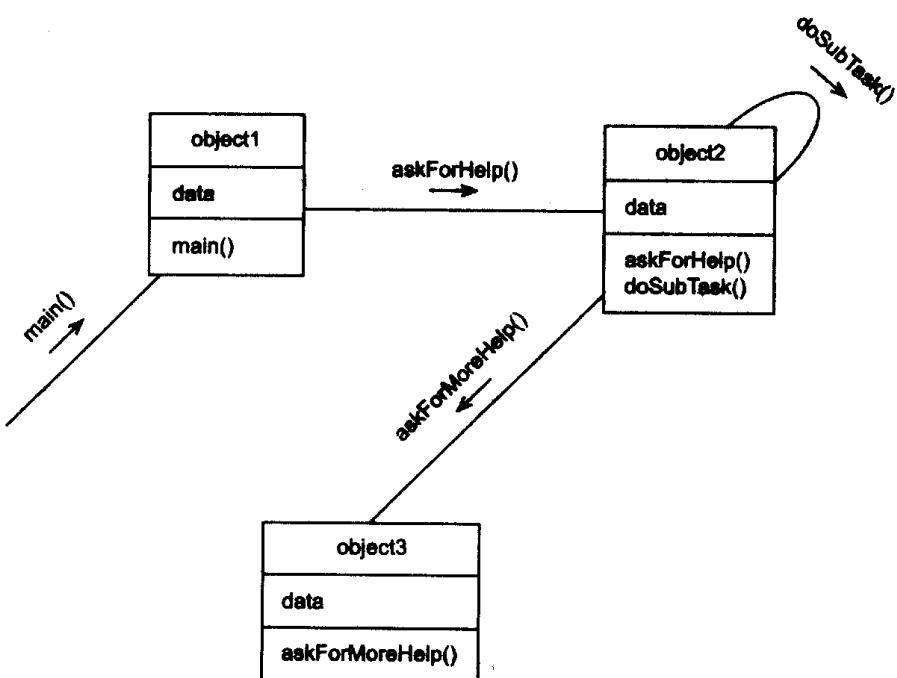


图 2-15 正在运行的面向对象程序

main 操作的理念不仅可以应用于在控制台上执行的程序，也适用于更复杂的程序，例如图形化用户界面(GUI)、Web 服务器和服务小程序(servlet)。下面是其工作方式的一些提示：

- 用户界面的 main 操作创建顶级窗口，告诉它显示自己。
- Web 服务器的 main 操作有一个无限循环，告诉 socket 对象监听某个端口的入站请求。
- servlet 是由 Web 服务器拥有的对象，它接收从 Web 浏览器传来的请求，注意，Web 服务器有 main 操作。

## 2.13 垃圾收集

当创建对象的程序不再使用该对象了，该怎么办？这似乎是一个小问题，但程序中的对象都不是免费的：每个新对象都要占用计算机内存的一个小区域，在程序运行时，可能会创建越来越多的对象，这样，运行其他程序的可用内存就会减少。如果在对象使用完后不重新声明，计算机就可能用尽内存(程序使用的内存通常在该程序结束后，返回给计算机，但有可能同时运行好几个程序，其中一些可能运行几天、几星期或几年)。

最好不要让程序创建越来越多的对象，又不在它们的生命周期结束后，采取措施清理它们。传统上，程序员必须确定何时去除与对象的最后连接，以便显式地删除或释放对象的内存。(结构化的语言没有对象，但有记录、结构和数组，它们也需要释放)。跟踪对象的生命周期是很复杂的，程序员很容易忘记一些已没有用的对象，使问题更严重，这种错误称为内存泄漏。

像 Java 这样的语言规定，程序会自动重新声明对象，程序员不需要做任何事。其理念是每个程序都有一个助手，称为垃圾收集器。它四处巡视，查找未连接的对象，并清理它们。听起来很神奇吧？实际上并非如此。现在，每个程序都有一个运行时系统(run-time system)，这个软件总是在我们编写的代码后面执行，它执行内务操作，例如垃圾收集。

这里不详细讨论垃圾收集器的工作原理，知道垃圾收集器可以删除不能在程序中直接或间接通过名称访问的对象即可。不能访问的对象就不能发送消息，如果对象不能发送消息，就不能回应问题或执行命令，因此必须通过垃圾收集来清除。

纯面向对象语言，例如 Smalltalk、Java 和 Eiffel，有垃圾收集器。复杂的面向对象语言，例如 Object Pascal，有时有垃圾收集器，但这些语言本身就非常复杂了，所以最好避免带垃圾收集器。C++就没有垃圾收集器，程序员必须使用“智能指针”，当对象失去了最后一个引用时，智能指针就会删除该对象。

## 2.14 类

类封装了一组对象的公共属性。考虑类的方式有许多种，图 2-16 显示了其中的一些。把这些图形转换为文字：

- 工厂根据蓝图制造对象
- 集合指定其成员对象具备的特性
- 模板允许生产给定形状的任意个对象
- 字典定义尽可能准确地描述对象



图 2-16 考虑类的不同方式

图 2-17 显示了一些类。在 UML 中，类在类图中被绘制为盒子，这样就很容易看出类和对象的区别，类的名称(在类图上)是不加下划线的，而对象名称(在对象图中)是加下划线的。类和对象很少在同一张图中出现，所以可以根据类进行建模的大多数工作，而对象图仅用于演示和验证。面向对象的程序员常常说“每个对象都是类的一个实例”，因此术语“实例(instance)”就是对象的同义词。

按照约定，类的名称以一个大写字母开头。在类图中，它们显示为黑体，但在手工绘图时，这是比较困难的。在面向对象的环境下，类的名称比较短，而且是单数。

分类就是把事物归类，这是人类很擅长的一个任务，我们从 1 到 1 岁半开始就在分类：玩具、食物、女孩、男孩、小狗，所以很容易在编程中实现分类，这就是其可访问性或接近自然的属性。面向对象的软件开发就意味着自然：接近真实世界，接近我们思考问题的方式。因为在真实世界中，类是继对象之后的一个主要步骤，有足够的理由把类引入程序。从软件的角度来看，另一个理由是类允许在相关的对象之间共享元素的定义，不必重复定义。

再看看图 2-17，这些类有什么相似之处？显然，它们都是交通工具，一些在水上使用，一些在陆地上使用等。绘制一些线条，说明这些类的关系，就会得到图 2-18。

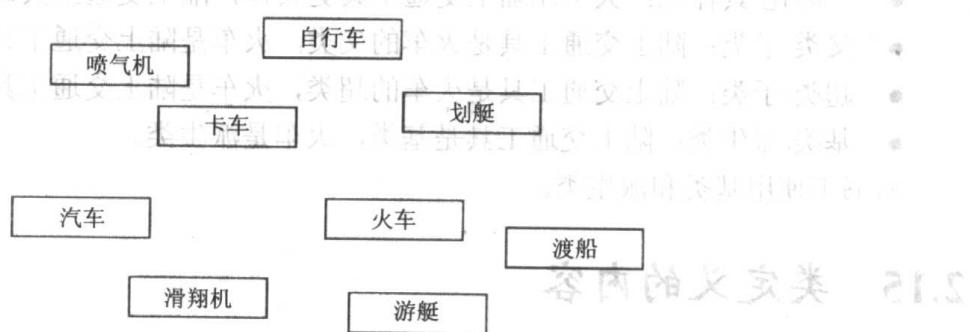


图 2-17 一些类的例子

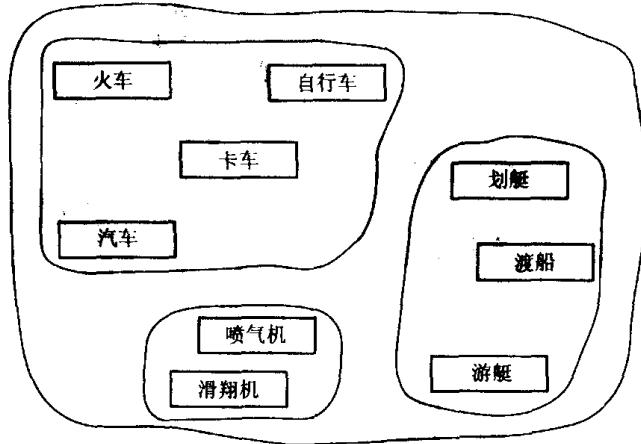


图 2-18 类的组合

这张图显示了类的层次，使用规范的 UML 表示法重新绘制类的关系，会看得更清楚，如图 2-19 所示。其中白色箭头用于从较具体的概念指向左边不太具体的概念。

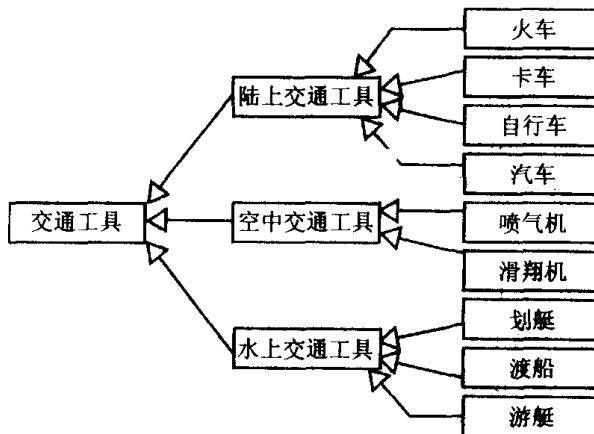


图 2-19 一个层次结构

与前面的聚合层次结构一样，除了公共名称“部分-整体层次结构”之外，这种层次结构也有几个公共名称：

- 继承：火车继承了陆上交通工具的特性。
- 一般化/具体化：火车比陆上交通工具更具体，陆上交通工具比火车更一般化。
- 父类/子类：陆上交通工具是火车的父类，火车是陆上交通工具的子类。
- 超类/子类：陆上交通工具是火车的超类，火车是陆上交通工具的子类。
- 基类/派生类：陆上交通工具是基类，火车是派生类。

本书不使用基类和派生类。

## 2.15 类定义的内容

面向对象的开发人员使用类来描述某种对象拥有的编程元素。没有类，就必须给每个对象添加这些元素。

为了便于演示，图 2-20 显示了一个用 Java 编写的完整类，本书并不详细介绍 Java，但这个例子非常简单，其中包含类的 6 个基本元素(参见表 2-2)。

```

1 // An actor with "name" and "stage name" attributes
2 public class Actor {
3
4     //Fields
5     private String name, stageName;
6
7     //Create a new actor with the given stage name
8     public Actor(String sn) {
9         name = "<None>";
10        stageName = sn;
11    }
12
13    //Get the name
14    public String getName() {
15        return name;
16    }
17
18    //Set the name
19    public void setName(String n) {
20        name = n;
21    }
22
23    //Get the stage name
24    public String getStageName() {
25        return stageName;
26    }
27
28    //Set the stage name
29    public void setStageName(String sn) {
30        stageName = sn;
31    }
32
33    // Reply a summary of this actor's attributes, as a string
34    public String toString() {
35        return "I am known as " + getStageName() +
36                    ", but my real name is " + getName();
37    }
38 }
```

图 2-20 一个简单的 Java 类

表 2-2 类定义的信息

元 素	作 用	图 2-20 中的例子
类名	在代码的其他地方引用类	第 2 行的 Actor
字段	描述这类对象存储的信息	第 5 行的 name 和 stageName
构造函数	控制对象的初始化	第 8 行的 Actor()
消息	以使用对象的方式提供其他对象	第 14 行的 getName(); 第 19 行的 setName(); 第 24 行的 getStageName(); 第 29 行的 setStageName(); 第 34 行的 toString()
操作	告诉对象如何操作	第 15、20、25、30、35、36 行
注释	告诉程序员如何使用或维护类(编译器忽略)	以//开头的行, 如第 1 行和第 4 行

新对象是由 Actor 操作创建的，这是一个特殊的操作，称为构造函数，它只在创建类的实例时使用。在 Java 中，使用下面的表达式创建对象：

```
New Actor("Charlie Chaplin");
```

在这个例子中，表达式会生成 Actor 的一个新实例，它的参与者名字是 Charlie Chaplin，名称是<None>。getName() 和 setName() 等操作称为获取(getter)和设置(setter)操作，因为它们获取和设置信息。

除了上面列出的元素之外，纯面向对象编程语言还允许程序员指定系统的哪些部分可以访问元素，通常至少可以指定元素是公共的(在其他地方可见)还是私有的(仅对对象本身可见)，因此要使用 Java 中的关键字 public 和 private。一些语言允许程序员添加断言(assertion)，即必须总为 true 的逻辑语句，例如这个类的对象总是有正的结余，或者这个消息总是返回非空字符串。断言对于可靠性、调试和维护比较有用。

## 2.16 共享数据和共享操作

在面向对象的程序中，程序需要的所有信息和服务都必须在某个地方可用。如果程序的设计合理，信息和服务就应在明显的地方可用。因此，应将下面列出的信息和服务放在什么地方？

- (1) 储蓄账户的当前利率
- (2) 一月份的天数
- (3) 计算给定年数的复合利息
- (4) 计算今年是否为闰年

这些都应与某类对象相关，其中 1 和 3 与 SavingsAccount 相关，2 和 4 与 GregorianCalendar 相关。但是，这些信息或服务都不与某个特定的 SavingsAccount 和 GregorianCalendar 相关，它们与所有的储蓄账户和日历相关。把这些元素放在特定的对象中是不合适的(如果使用实参，就必须在 SavingsAccount 的每个实例中都放置 interestRate，浪费了空间。同样，如果使用实参，就必须创建一个 GregorianCalendar，才能确定今年是否为闰年，浪费了时间)。

上述信息和服务似乎不适合放在对象中，面向对象的语言通常允许程序员把这些元素放在类中。所以，除了字段、消息和方法之外，还有类字段、类消息和类方法。例如，Java 程序员可以使用关键字 static 表示元素与整个类相关，而不是与类的实例相关。一些语言把类当做对象来处理。

类元素使用起来不是很简单，因为一些语言不把类作为纯粹的对象，例如，类元素之间的继承就无效了。甚至把类看作纯粹的对象的语言也会陷入“元类(metaclass)”的复杂性中。在其他人的设计和代码中，可能会遇到类字段、类消息和类方法，甚至自己还会使用它们，但总是要先试试下面的替代方法：

- 找出或引入另一种对象。例如，不把 interestRate 设置为类字段，而是设置为 Bank 对象上的一个字段(这也有助于扩展软件，以处理更多的银行)。

- 使用单一类，通过仔细地编程，这种类可确保只有一个实例：单一对象(详见第 11 章)。它很适合于“今年是否为闰年”，因为只有一个阳历。

## 2.17 类型

在纯面向对象的环境下，所有的物体都是对象。用编程的话来说，每个种类的值都是一个类。Smalltalk 和 Eiffel 是遵循这个基本原则的两种语言。但是，大多数面向对象的语言还有非对象类型，称为原型(primitive)。之所以存在这种情况通常是为了确保语言简短、性能和历史原因。愤世嫉俗者认为，懒惰的人(部分语言设计者)也可能是一个因素。无论使用什么参数，原型都无处不在。因此，需要习惯它们。

以 Java 为例，我们可以声明一个字段，它是类类型，例如 String，即一系列字符，或者是原型类型，例如 int，即一个简单的数字。在图 2-21 中，anActor 的 age 字段是一个整型原型，由 name 指向的字符串对象包含原型字符。

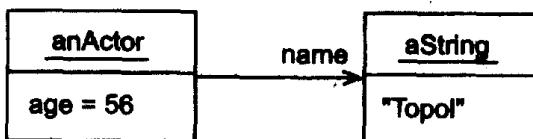


图 2-21 面向对象的类型

对象和原型的主要区别是，原型可以与对象相同的方式用作值，但不能给原型发送消息，给它提供字段，或者以其他方式把它看作对象。(在更深的层次上，大多数情况下，对象可以通过指针来访问，而原型不能)。

如果把某件东西当作对象，把另外一个东西当作原型，就会引起混乱，但这并不是一个大问题。原型适合于简单的值，例如数字和字符，除此之外的所有东西都应是对象。大多数语言都提供了一部分已准备好的原型类型。例如，Java 提供了 byte、short、int、long、float、double、char 和 boolean。即使语言允许定义自己的原型(例如 C++ 和 Eiffel)，也应把这看作高级技术。

数组在 Java 中用[]操作符表示，它位于对象和原型之间：它们是特殊的对象，编译器和运行时系统使用它们可提高效率。但是，如果希望语言纯粹一些，就应避免使用数组，而可以使用类 List 来代替。

UML 包含的一个机制允许用 Integer、Real 和 Boolean 等名称定义独立于语言的原型。但是，本书将使用 Java 原型，因为本书所有的代码段都是用 Java 编写的。UML 标准允许使用依赖语言的类型，只要清楚自己在做什么即可。

## 2.18 术语

前面介绍的对象概念有许多术语，不同的人使用不同的术语来表示相同的概念。更糟糕的是，一些人对术语的使用并不正确。图 2-22 显示了一些术语，在一组中的术语可以互换(本书使用带下划线的术语)。

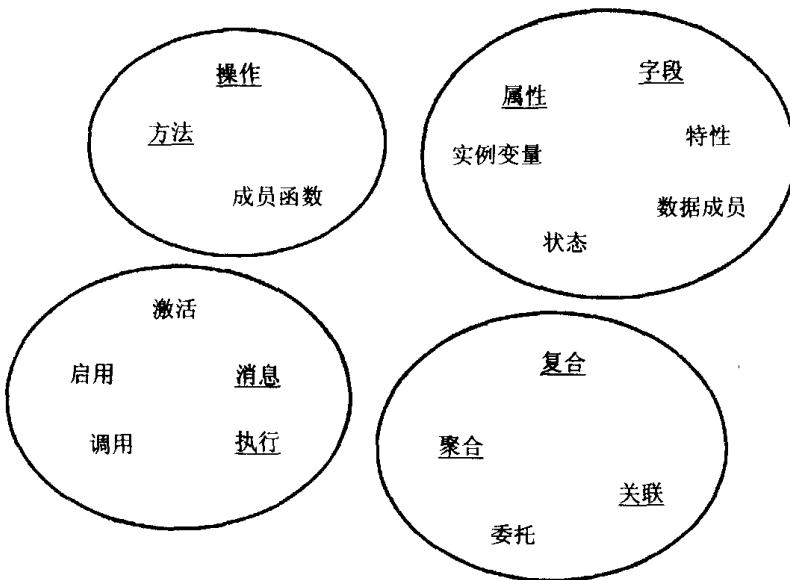


图 2-22 面向对象的术语

还可能遇到集合术语，例如“行为”（操作的集合）、接口（消息的集合）、对象协议（接口的同义词）和数据（字段的集合）。在本书中，只以描述性方式使用表 2-3 中列出的术语。

表 2-3 本书使用的术语

术语	定义
属性	一小段信息，例如颜色、高度或重量，描述对象的一个特性
字段	对象内部的指定值
操作	属于对象的一段代码
方法	操作的同义词
消息	从一个对象发送到另一个对象的请求
调用	执行操作，以响应消息
执行	调用的同义词
关联	两个对象之间的直接或间接连接
聚合	强关联，隐含着某种部分—体层次结构
复合	强聚合，部分在整体的内部，整体可以创建和销毁部分
接口	对象理解的一组消息
协议	通过网络传送消息的认可方式
行为	对象的所有操作的集合

属性可以由对象存储（封装），但不是非如此不可。例如，圆有半径和直径属性，但只需要存储半径，因为直径可以计算出来。为了避免混淆，本书只介绍存储的属性，如果有必要，添加一个或多个操作（如 `getDiameter`）会隐藏派生的属性。

字段与属性不是完全相同。首先，字段表示存储什么内容的决策；其次，字段可以用于存储与另一个对象的连接，如对象图中的可导航链接所示。在开始设计时，属性和关联会生成字段。

在面向对象软件开发的早期，使用术语“属性”和“操作”（因为它们是 UML 术语）。在后期，处理低级设计和源代码时，属于术语“字段”和“方法”（因为它们是编程术语）。

## 2.19 重用代码

前面已提及几次术语“重用”，下面详细探讨它的含义。简言之，重用是指多次使用代码，目的是：

- 开发更快速、简单
- 维护更容易(代码较少，人为错误就较少)
- 更强壮的代码(每次重用代码时，都会重复测试它，错误就会越来越少)

重用之所以很长时间后才出现，是有其历史和技术原因的。从历史的角度来看，软件业最初是考虑如何给计算机编程(一开始是机器码)，之后考虑如何更高效地给计算机编程，接着是考虑如何以可靠系统的方式开发大型系统。

随着越来越多的系统的构建——在过去的几十年中，每年都会编写出上百万行代码，在不同环境下重用代码的潜在趋势也越来越明显。可以肯定，并不需要编写这么多不同的系统。大多数软件都是各个公司的软件部门关起门来，使用不同的编程语言编写的，所以，很容易看出重用为什么很少见。即使打包的软件开始在各个公司中使用，源代码甚至二进制仍是高度机密。对公司商业利益和知识产权的保护甚至会妨碍在公司内部重用代码。

从技术的角度来看，重用是比较困难的，因为开发小组的态度不积极，开发方法也集中于“只需解决当前的问题”。编程语言和方法也没有提供提升重用的功能，而是把数据和操作散布在整个系统的各个地方，难以把相关的代码段收集到一个较大的、能够重用的块中，从而加大了重用的难度。

面向对象是通过对整个域的一般概念建模来驱动的，从而为重用提供了机会。例如，开发薪水系统一般要开发一个 Employee 类。由于驱动开发的是“员工概念对这个公司有什么含义”，而不是“薪水系统需要什么员工信息”，所以，得到的 Employee 类应能用于同一公司开发的其他系统。另外，对象的模块化会减少把属性和操作散布在系统中的趋势，更容易提取和细化 Employee 代码。

这种情形常常会有所改进，我们只需编写系统中针对特定问题的部分，系统的其他部分可以用以前编写好的代码来实现。常见的容易理解的应用程序领域都是这样，例如用户界面、数据库访问、分布式编程、输入/输出、网络访问、电子商务、访问旧(已有)系统、安全(身份验证、授权、完整性检查、来源检查)、文本处理、数学、游戏、服务查找、声音复合与回放、2D 图形、3D 图形、电子邮件、图像处理、多媒体编码和解码、消息传输、事务处理、电话、声音复合和识别、与数字 TV 广播的集成。

重用可以分为如下类别：

- 重用系统中的函数：重用最简单的形式(在传统的系统开发中使用)是编写在不同地方调用的实用函数。例如，系统的各个部分需要搜索一组客户名，于是编写一个可以在各种环境下调用的通用搜索函数。编写可重用的函数不同于编写把复杂过程分解为简单步骤的函数。
- 重用对象中的方法：封装在对象中的方法可以在其他方法中调用。例如，GUICOMPONENT 类中的非公共方法 drawFilledRectangle 可以由需要用当前背景色填充一个屏幕区域的任意 GUICOMPONENT 方法使用。应尽可能重用对象中的方法。对象中的非公共方法常常用以传统的方式分解复杂的过程。

- 重用系统中的类：我们定义的许多类都可以在系统的不同部分使用。例如，如果定义了一个 Customer 类，在市场营销系统中使用，该类的对象就可能出现在系统代码的许多不同地方。这类重用是面向对象方法的基础。
- 在系统之间重用函数：通用函数可以在自己和同事开发的其他系统中重用(在传统的系统开发和面向对象的开发中)。例如，一个函数可以从员工的薪水号中提取该员工进入公司的年份。为了让同事重用这个函数，必须让他们知道有这个函数，最好把它放在重用库(reuse repository)中；重用库是有用函数的数据库，开发人员在编写新代码时，可以查看该库。
- 在系统之间重用类：可以发布和重用整个类(包含所有的属性和操作)，而不能只重用一个函数。例如 Employee 对象封装了在整个公司中使用的员工属性和一组有用的操作。面向对象的爱好者会公布包含类而不是函数的重用库。

为了让开发人员把他们的类开发成可重用的，放在重用库中，最好提供某种形式的奖励(比较常见的是奖励一个带诙谐徽标的奖杯)。还可以给第三方提供对类库的访问权限(可能要收费)。有一点儿实践经验后，在自己的系统中重用类并不困难。

- 在所有的系统之间重用类：软件组件类似于硬件组件。软件组件用于在任何环境下重用，进行了强有力的封装(客户看不到内部的工作)，它带有标准的界面，可以从第三方处获得，通常要交费。每种面向对象的编程语言都有自己的软件组件形式，例如，Java 有 JavaBeans。在传统领域中，没有与之对应的软件组件，因为相关函数的组合对第三方没有什么用处。

软件组件的例子有，可以归类为办公室高效率套件的电子表格，可以归类为家庭上税软件包的收入税对象。软件组件只是遵循其界面样式的合理规则的对象(就像使用命名约定标识获取器和设置器一样)。

- 函数库：高质量的相关函数可以组合为一个库(library)，使它们可以一块使用。例如 stdio 函数库，它最初来源于 UNIX 系统，为 C 程序员提供了输入输出功能。函数库可以在传统的系统开发和面向对象的开发中使用。有时，设计优良的库由 ISO 或 ANSI 进行标准化。函数库还可以在公司内部使用、免费使用，或对外销售，以获取利润。
- 类库：是对函数库的改进，类库提供的是整个类，而不是单一的函数。编写类库需要更多的经验，例如 Java 2 Enterprise Edition(J2EE)库[Bodoff 等, 02]，它为上述所有容易理解的重用领域提供了代码。与函数库一样，类库也可以在公司内部使用、免费使用，或对外销售，以获取利润。
- 设计模式：设计模式描述了如何从容高效地创建面向对象系统的各个部分。由于引入了设计模式，模式也应用于其他领域，如系统架构。每种模式都有一个简短的描述、一个详细的描述、在哪里使用它的建议和代码样例(详见第 11 章)。例如，Adapter 模式的描述是“适配器把一个对象的接口转换为客户希望的另一个接口”。设计模式需要许多经验，但比类库需要的经验少一些。
- 框架：顾名思义，框架是将自己的代码附着其上的已有结构。在面向对象的环境下，框架包含许多预先编写好的类，以及描述开发人员必须遵循的构建规则的文档。例如 Enterprise Java Beans(EJB)框架[Bodoff 等, 02]，它包含 J2EE 库和一个上百页的文档，

该文档指定了程序员如何编写可重用的企业组件，第三方如何使用 Java 应用程序服务器。大多数框架都是顶尖专家设计的。

那么，如何设计重用？这里先不探讨设计模式和框架，因为这些技术超出了本书的范围，只讨论如何编写可重用的类。即使一个可重用的类也常常有一两个紧密协作的类，所以我们会创建一小组可重用的类，而不是只创建一个可重用的类。下面是一些提示：

- 总是遵循样式规则：样式规则是编写类的建议。如果以怪异的方式或个人的方式编写类，软件潜在的重用用户就很快转而使用其他代码，而不是了解您的个人特质。样式规则可以来源于公司，或者得到了广泛的接受。例如，Sun 控制着 Java 标准，Sun 认为好的样式一般都会被 Java 团体接受。除了很多的面向对象专家之外，Sun 还密切关注着外部专家的看法。
- 完整的文档说明：很少有程序员仅通过阅读源代码，就能理解类的重用方式。类至少要有一个意义明确的名称、一个总结该类的简短注释(一两句话)，一个描述该类如何使用的较长注释(可以好几段)，每个公共消息都要有一个较短的注释，来描述如何使用方法。注释应总是描述对象及其客户之间的协议，指定双方的义务。还要提供与类分开的说明文档，例如设计或教材信息。
- 准备编写比需要更多的代码：在为特定系统实现类时，常常认为“我敢打赌，这里的 foo 方法非常方便”。例如，即使 Circle 类当前只需要 getRadius 消息，也最好添加 getDiameter 消息，使类在其他系统中更有用。
- 使用模式和框架：模式和框架可以减少工作量，它们还能为其他开发人员理解，也就是说，其他开发人员在重用代码前需要了解的内容比较少。
- 设计客户-提供者对象：如果在对象之间需要双向甚至是循环协作，就会出现所谓的编码混乱(code spaghetti)。如果把对象设计为客户-提供者层次结构，事情就会简单得多。例如，可重用的 Employee 没有对其环境进行任何假定，而是提供了通常有效的公共消息。把这个理念进一步延伸下去，Employee 是受 EmploymentHistory 控制的，而 EmploymentHistory 并不了解 Employee。假定对象是完成预先指定的任务，不执行要求它们执行的任务。为了补偿它们被控制的命运，对象也有可以控制的下层对象。
- 使每个对象只有一个目标：这称为高聚合(high cohesion)。编写出的对象应避免用于多个目的，例如维护员工的个人信息及其雇佣历史。
- 把接口与业务行为分开：可重用的对象应可以在任何环境下使用。例如，对象需要在许多不同的接口(工作站、移动电话或 Web 服务器)中直接或间接使用。如果对象包含了某个接口的细节，就会出问题。因此，业务对象应只包含业务行为。还可以提供接口对象，来查看业务对象，但这是可选的。这种接口对象本身也是可重用的。
- 为问题和命令设计对象：如果消息是问题——“现在几点”，或命令——“把时间设置为……”，对象就很简单。消息偶尔既是问题，也是命令，但应考虑把这种消息用于高级技术。合并的消息会产生混淆，例如把时间设置为……，并说明设置之前的时间。

## 2.20 小结

本章的主要内容如下：

- 软件对象表示真实世界的物体，由属性来描述，可以执行行为(操作，通常称为方法)。
- 消息允许对象交流和协作，以完成任务。
- 当创建对象的程序不再需要该对象时，垃圾收集会重新声明这些对象使用的空间。
- 类允许组合类似的对象，在相关对象之间共享元素的定义，所以不必重复这些定义。
- 重用代码可以使开发过程更快、更简单，获得更健壮的代码，更容易维护。

## 2.21 课外阅读

David Taylor 介绍了对象概念的起源以及对象的优点。[Taylor 97]适合于非技术读者，主要为不打算编写实际代码的人介绍了对象。

为了提示 Java 使用技巧，Sun 网站 <http://java.sun.com> 包含了大量的免费信息，包括教材。Java 一直在更新和改进，所以 Sun 网站是 Java 开发人员的主要资源。如果需要了解本书中的代码段，可以查看在线语言教材，它也是[Campione 等，00]。

## 2.22 复习题

1. 在 UML 图中，对象和类如何区分？(单选题)
  - (a) 对象的标签显示为斜体。
  - (b) 类的标签放在一个方框中。
  - (c) 对象的标签加了下划线。
2. 在图 2-23 中，图 1 和图 2 说明了什么？(单选题)

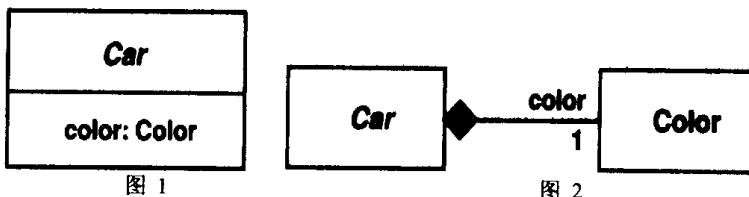


图 2-23 复习题 2 的图

- (a) 图 1：聚合；图 2：复合
- (b) 图 1：属性；图 2：聚合
- (c) 图 1：聚合；图 2：属性
- (d) 图 1：属性；图 2：复合
- (e) 图 1：复合；图 2：属性

3. 对象标识的含义是什么? (单选题)
  - (a) 如果两个对象的属性值相同, 这两个对象就是一样的。
  - (b) 每个对象的类有唯一的系列号。
  - (c) 所有的对象都彼此相同。
  - (d) 每个对象都有唯一的标识, 以彼此区分。
4. 下面哪个术语描述了对象由其他对象组成? (单选题)
  - (a) 一般化
  - (b) 继承
  - (c) 关联
  - (d) 聚合
  - (e) 具体化
5. 什么是封装? (单选题)
  - (a) 使用圆环图描述对象。
  - (b) 确保对象中的数据只能通过操作来访问
  - (c) 密封对象的状态, 使之不能改变
  - (d) 把对象放在集合中

## 2.23 练习 1 的答案

1. 街道与房子是独立的。街道和房子可能同时构建, 但随着时间的推移, 会增加新房子, 拆除旧房子。即使街道上没有房子, 街道也仍是街道。这肯定是关联。
2. 如果书是一本小说, 它在撕下一页后就不是连贯的小说了。所以这是可能的聚合。另一方面, 如果这是报告人的笔记本, 其页面设计为可以取出, 就可以把它看成关联。
3. 交响乐中的音符类似于小说中的页, 即聚合。如果管弦乐队正在演奏交响乐, 不小心漏掉了一个音符, 人们就会注意到。管弦乐队成员应因为未演奏完整的交响乐而受到指责。
4. 家庭娱乐系统中的组件合在一起会变成一个整体(这是使用“组件”这个词的含义)。但是, 如果把游戏控制台搬到儿童的卧室, 家庭娱乐系统仍可以使用, 只是少了一个娱乐项目而已。另外, 还可以添加 DVD 播放器, 这不会把已有的系统突然转变为家庭娱乐系统, 只是改进原系统而已, 所以这是一个关联。

仍涉及到聚合: 电视是一个聚合, 其他组件也是这样。因此, 这个例子在关联中包含着聚合。还可以在聚合中找出关联(例如 DVD 播放器播放 DVD)。把某个组合看作聚合还是关联, 常常取决于看它的角度。

## 2.24 复习题答案

1. 在 UML 图中, 对象与类的区别取决于 (c) 对象的标签加了下划线。
2. 在图 2-23 中, 图 1 和 2 表示 (d) 图 1: 属性; 图 2: 复合。
3. 对象标识的含义是 (d) 每个对象都有一个唯一的标识, 以区分彼此。
4. 对象由其他对象组成是用 (d) 聚合来描述。
5. 封装表示 (b) 确保封装在对象中的数据只能通过操作来访问。

# 第 3 章 继 承

人们普遍认为继承是比较基本的，但它比前面介绍的更复杂，所以读者在第一次阅读本书时，可以跳过本章。在开发软件时不使用继承是完全可以的。但是，如果跳过本章，本书后面的一些论题就不容易理解了。

## 学习目的：

- 理解继承的含义
- 理解抽象类和具体类的区别
- 了解使用继承的场合

## 3.1 引言

继承可以指定类从父类中获取一些特性，再添加它自己的独特特性——这就会描述对象的整个系列。继承可以把类组合到越来越通用的概念中，最后就会推导出我们生活的世界。

从编程的观点来看，需要继承是因为：

- 它支持更丰富、更强大的建模。这有利于开发小组和其他想重用代码的开发人员。
- 它可以在一个类中定义信息和行为，在相关的子类中共享这些定义。这样要编写的代码就比较少。
- 继承是很自然的。它是面向对象的主要动机之一。

子类继承超类的所有字段、消息和方法(以及断言)。例如，如果要给陆上交通工具建模，就可以从图 3-1 的层次结构开始。

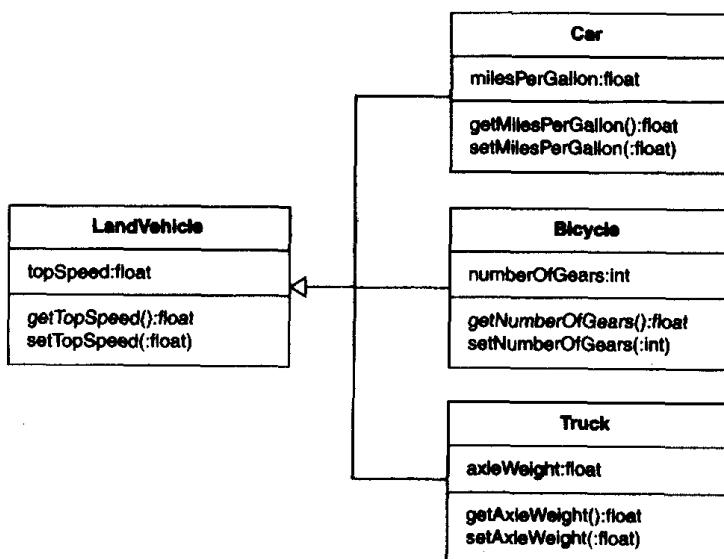


图 3-1 继承是什么

在图 3-1 中，显示了字段的类型、消息参数和消息回应。在 UML 中，类型放在一个冒号的后面，例如:String。为了简便一些，图 3-1 省略了参数名，这是合理的，因为参数很少，它们的含义也很明确。但是，仍要显示参数的类型，而且必须包含冒号，否则标签就是模糊的；:float 表示 float 类型的字段或参数，而 float 表示名称为 float 的字段或参数。如果包含了参数名，它们就应显示为 newTopSpeed:float。

使用这个类层次结构，如果要创建 Car，它就要从 LandVehicle 继承一个字段 topSpeed，另一个字段 milesPerGallon 是 Car 类自己引入的。它还有四个方法(getTopSpeed、setTopSpeed、getMilesPerGallon 和 setMilesPerGallon)，两个是继承来的，两个是 Car 引入的。如果要创建 Bicycle，它就会有 TopSpeed 元素，与 Car 一样，但它有 numberOfGears 元素，而不是 milesPerGallon 元素。

所以，继承不仅可以推导出一般类和特殊类，还可以减少子类的编程量(因为 LandVehicle 中所有的 TopSpeed 元素都会自动出现在子类中，不需要重复)。

## 3.2 设计类层次结构

下面是一个较大的例子。我们要给集合(Collection)建模，集合是可以包含其他对象的对象，供以后使用。在深思熟虑后，我们需要四类集合：

- List：该集合可以把所有的对象按照插入的顺序放置。
- Bag：该集合中的对象没有排序
- LinkedList：该集合中的对象使用序列对象的实现来排序，在该实现中，每个对象都指向序列中的下一个对象。链表很容易更新，但访问比较慢，因为必须沿着链表向下搜索。
- ArrayList：该集合中的对象使用数组来排序，数组是相邻内存位置的序列。数组的访问速度很快，但更新比较慢，因为在每次更新时，必须移动元素，或创建新的数组。

如何把这四个类以及 Collection 放在继承的层次结构中？关键是找出这些概念的主要类似之处。显然，它们本身都是集合，所以 Collection 必须放在最前面。接着，大多数集合中的对象都是有顺序的，只有 Bag 不是，这说明 Bag 应放在 Collection 下面一个与其他类不同的分支上。然后，List 不涉及内部实现，而 LinkedList 和 ArrayList 涉及(LinkedList 和 ArrayList 的唯一区别是它们的时间-空间折衷方案)。所以，List 必然是 LinkedList 和 ArrayList 的超类。这个逻辑过程会得到如图 3-2 所示的层次结构。继承箭头与上一章中的箭头有点儿不同，这种形式也是 UML 的正规部分，比较简洁和紧凑。

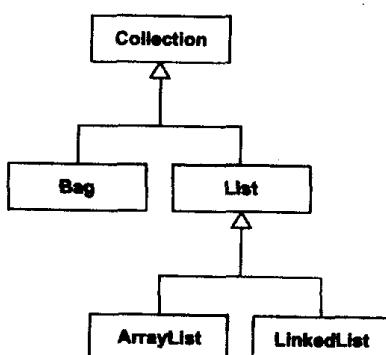


图 3-2 集合的层次结构

上述过程有点虚，因为我们一开始就有了层次结构中的所有类，再从上到下地建立层次结构。事实上，另一种方式更常见：首先确定层次结构的底部有哪些类——`LinkedList`、`ArrayList` 和 `Bag`，接着找出比较一般的概念，以丰富模型，共享元素的定义。因此，把 `LinkedList` 和 `ArrayList` 组合到 `List` 中，把 `List` 和 `Bag` 组合到 `Collection` 中。

在开发层次结构时，会查找可以共享的消息——可以放置消息的层次越高就越好。在查找其他类元素之前，应先查找消息，因为消息表示对象给外界显示的接口，这是它们最重要的特性。

考虑下面三个消息，它们将放在 `Collection` 层次结构中：

- `contains(:Object):boolean` 在集合中搜索对象，如果接收者包含参数，就返回 `true`，否则返回 `false`。
- `elementAt(:int):Object` 在参数指定的位置检索对象。
- `numberOfElements():int` 返回集合中的对象数。

把这些消息放在已有类的什么地方？`contains` 可以用于任何种类的集合，所以需要把它放在 `Collection` 中。而 `elementAt` 要把位置作为参数，所以它必须处理有顺序的对象（它对 `Bag` 没有意义），不能把 `elementAt` 放在 `Collection` 中，而可以把它放在 `LinkedList` 和 `ArrayList` 中，因为这两个类的对象都是有序的。但是，把它放在 `List` 中就可以避免这种重复操作。最后，`numberOfElements` 可以用于任何种类的集合，所以可以把它放在 `Collection` 中。

这些考虑可以得到如图 3-3 所示的消息分布。即使在这个类图中没有显示属性，属性框也保留着，以便看出 `List` 显示消息而不是字段。消息名显示为斜体，以强调还没有考虑方法。

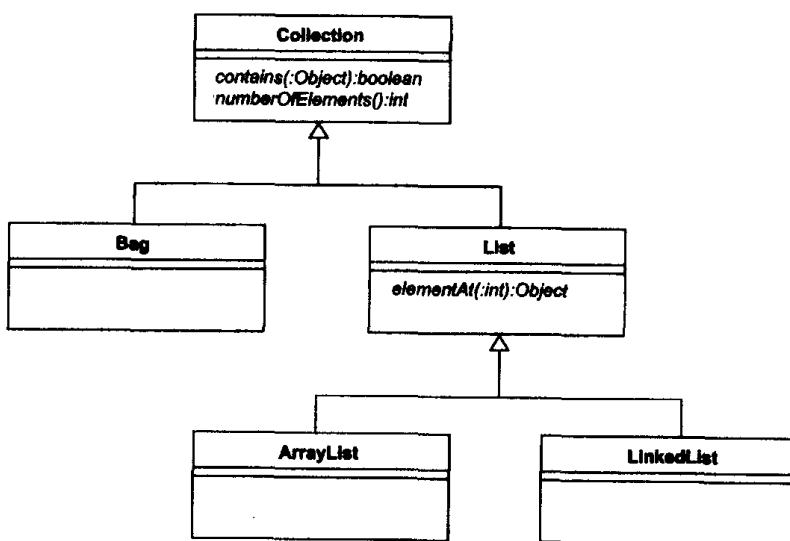


图 3-3 把消息放在层次结构中

这就有了一个对其他开发人员有用的完整的类集。但是，类还不能做任何事情，它们没有实现代码。

### 3.3 给类层次结构添加实现代码

我们已经有了类层次结构，还确定了消息的位置。下面必须添加实现元素(字段、构造函数和方法)。在这个阶段，不必担心层次结构需要的字段，因为这需要进入细节设计。同样，

忽略类的构造函数问题，只是假定已完成的类会配备一组有效、完整的初始化选择。下面看看方法放在哪里，这会引出两个重要的概念：抽象和重定义。

不可能在 Collection 类中编写 contains 方法，因为有序集合和无序集合的搜索算法是不同的。所以，必须先在 Bag 上实现 contains 方法。但 List 及其子类怎么办？稍微思考一下，使用刚才介绍的其他消息，就可以编写出适合于所有 List 的 contains 算法(参见实践 3)。

### 实践 3

```
1 boolean contains(Object o) {  
2     for (int i=0; i<numberOfElements(); ++i) {  
3         if (elementAt(i) == o) {  
4             return true;  
5         }  
6     }  
7     return false;  
8 }
```

注意，Java 中的参数是类型后跟其名称(参见第 1 行)，而 UML 中的参数形式是 name:type。  
for 循环(第 2 行)把 i 设置为从 0 到列表中元素数(不包含)的每个值，在循环中，检索和测试当前对象(第 3 行)。

编写 aMessage 时没有指定接收消息的对象，这表示“给当前对象发送 aMessage”(一些程序员喜欢编写 this. aMessage() )。例如，elementAt(i) 表示“找出当前 List 对象中的第 i 个元素的值”。

现在就能获得继承的好处：我们只编写了一个方法，它可用于 List 的任何直接或间接子类，而 List 可以有许多子类。

对于 LinkedList 和 ArrayList 来说，elementAt 消息的实现代码是不同的，所以必须添加两个不同的 elementAt 方法：一个用于 ArrayList，直接访问元素；另一个用于 LinkedList，沿着列表向下访问。

最后，给 numberOfElements 编写实现代码。这些实现代码取决于是把值存储为字段，还是在要求时计算值。下面看看这两种情况：

- 把元素数存储为字段

在添加一个对象时，字段必须递增，在删除一个对象时，字段必须递减。这个方法可以很快报告对象数，但要占用额外的空间，添加和删除对象时操作略慢。

- 要求时计算元素数

这对于 LinkedList 来说非常慢，因为 LinkedList 要遍历所有的元素。对于 ArrayList 和 Bag 类来说，内部的对象可能以某种方式存储了元素数，所以操作非常快。无论采用什么方式，这种方法都不会浪费存储空间，也不会减慢添加和删除对象的速度。

设计人员需要解决时间-空间折衷问题。例如，本例认为这两种 LinkedList 方法都不适用所有的类，甚至不适用所有的 List 对象。所以，在三个叶子类 Bag、ArrayList 和 LinkedList 中分别放置一个 numberOfElements 方法。

确定了如何实现之后，就得到如图 3-4 所示的层次结构。该图很容易区分它们，消息显示为斜体，方法显示为罗马文本(它也是正确的 UML 表示法)。

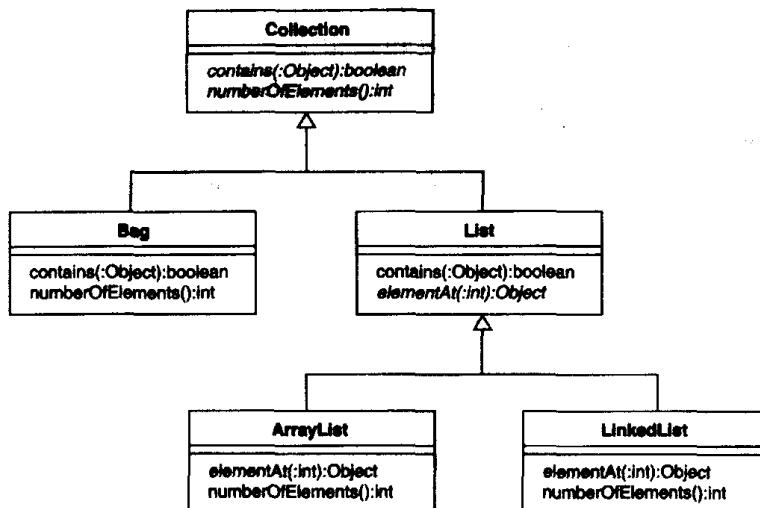


图 3-4 在层次结构中放置方法

在 Collection 类中，现在有了两类消息：带有关联方法的消息和不带关联方法的消息。没有方法的消息是未完成的消息——关联的类把消息作为其接口的一部分，但这里必须向下寻找子类链，才能找到实际方法。

未完成的方法称为抽象方法或抽象操作，因为它不真实，不固定，所以不能使用。抽象方法的对立面是具体方法或具体操作。具体方法有实际的代码行，是固定的，所以可以使用。在 UML 中，抽象方法显示为斜体，具体方法不显示成斜体——如果不能使用斜体，可以把“{abstract}”放在方法的右边。

## 3.4 抽象类

抽象类是至少有一个抽象方法的类——抽象方法可以是该类本身的方法，也可以是从超类继承来的。图 3-5 再次显示了集合层次结构，其中抽象类的名称显示为斜体(这也是正确的 UML)。如果不能使用斜体，可以把“{abstract}”放在类名的上面或左边。

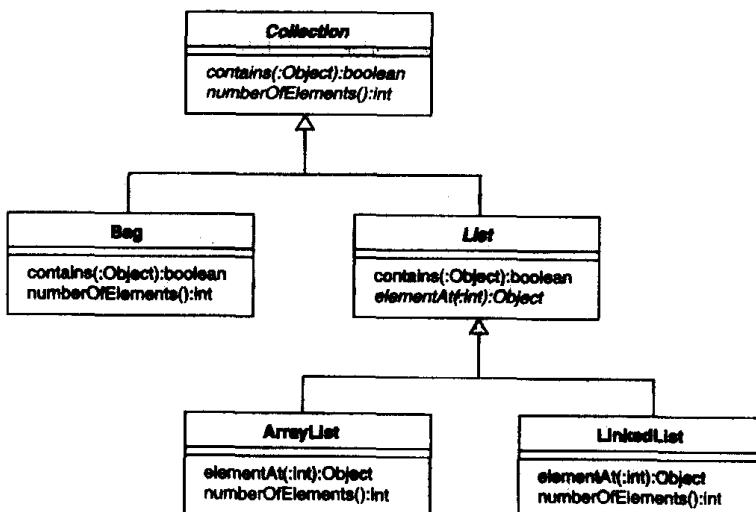


图 3-5 斜体的抽象类名

抽象类有如下优点：

- 它们允许更丰富、更灵活地建模。例如，List 类有三个消息 contains、elementAt 和 numberOfElements，但我们不能为它们提供具体的方法。
- 它们可以共享更多的代码，因为可以编写具体的方法来使用抽象的方法，例如，List 的 contains 方法调用抽象方法。

抽象是一件很自然的事情，所以面向对象提供了这个功能。考虑给水果削皮：我们可以削去任何水果的皮，但无法描述出适用于所有水果的削皮方法。因此，给水果削皮必然是一个抽象方法，水果本身也是一个抽象概念。

通过前面介绍的抽象概念，可能会出现一个大问题：如果创建了一个 Fruit，再给它发送 peel 消息，会发生什么？如果创建一个 Collection，给它发送 contains 消息，会发生什么？我们试着调用抽象方法——不真实存在的事物——对象不知道该做什么。

大多数面向对象的语言都禁止创建抽象类的实例。例如，Java 编译器不会编译包含表达式 new List() 的程序。在这种情况下，编译器会变得类似于妄想狂，因为我们从来不会在新对象上调用抽象方法，但编译器不会检查这一点。

“不能创建抽象类的实例”这一规则对应于现实世界：如果给您一些钱，要您到商店买一些水果，您就会问“买什么水果？”，因为您需要具体的请求。

在设计类层次结构时，应记住，大多数超类都是抽象的。下面说明了继承层次结构是从底向上派生的：

1. 在问题域中查找具体的概念，推导出它们的知识和行为。
2. 在具体的类中找出共同点，以便引入更一般的超类。
3. 把超类组合到更一般的超类中，直到找出最一般的根类为止(例如 Fruit 或 Collection)。

在表示泛化(超类)时，我们希望它们是抽象的，否则它们就可能表示为第 1 步中的具体概念。大多数情况下，大多数超类都是抽象的，因为超类中总是有抽象方法(可能是继承来的)。为了表示这一点，即使还没有在超类中发现抽象方法，Java 和 UML 也允许把类标记为抽象。

如果发现一个具体类继承自另一个具体类，该怎么办？首先可以试着进行如图 3-6 所示的转换。在该图中，有一个具体类 Y，它继承自另一个具体类 X，可以把这个类 X 转换为抽象超类 X1，它有两个具体的子类 X2 和 Y。

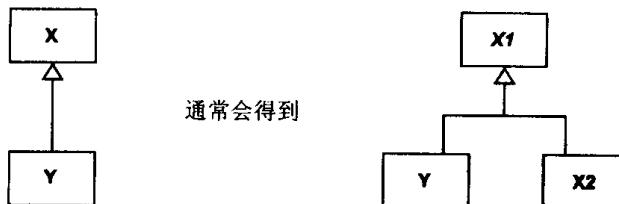


图 3-6 大多数超类都是抽象的

如果研究左边的层次，就会发现，Y 并不是 X 的真实子类——它的工作方式与 X 不同，所以需要改变一些方法的含义，甚至禁用它们。假定 X 是 Fruit，Y 是 Orange，则 Orange 的工作方式与 Fruit 完全相同，但添加了额外的知识和行为，所以不能禁用方法。

对于右边的层次，X 和 Y 都有的每个知识和行为都放在 X1 中，X 中有而 Y 中没有的知识和行为都放在 X2 中。(老 X 分解为两个 X)。现在就有一个更清晰的图：X1 比较抽象，X2 和 Y 的行为方式与 X1 完全相同，X2 和 Y 都有自己的额外知识和行为。

有时，我们喜欢从一个具体的类中继承，以改变已有类中的元素。例如，另一个开发人员已经实现了 `ArrayList` 类，但它还不能完成我们希望它完成的任务，所以引入一个新的子类 `MyArrayList`，进行微小的修改。有时这是合理的，但有一个基本缺陷：创建 `ArrayList` 的实例的已有代码仍需要修改：无法迫使已有的代码创建改进类的实例。

## 3.5 重定义方法

面向对象可以重新定义继承来的元素。最简单的形式是，重定义允许子类修改继承方法的实现代码——消息看起来是相同的，但代码行被替换了。重定义的另一种形式可以使消息在子类中的可见性更高：前面只介绍了公共和私有两种可见级别，这意味着子类可以把私有消息转换为公共消息。重定义的另一种形式可以改变属性的名称或类型。

下面的讨论集中于重新定义方法的内容，因为这是重新定义最重要的原因。重新定义方法有三个原因：

- 继承的方法是抽象的，我们希望给它提供一些代码，把它变成具体的。例如，`contains` 在 `Collection` 中是抽象的，但在 `Bag` 和 `List` 中应是具体的。
- 子类中的方法需要完成一些额外的工作，例如 `toString` 方法必须总结子类引入的新属性。
- 可以为子类提供更好的实现代码(更高效或更准确)，例如，如果给 `LinkedList` 类添加一个索引，就可以重新定义 `contains`，使之比 `List` 使用的线性算法更快。

在进行额外的工作时，应确保超类定义仍完成它以前完成的工作，这提高了代码的共享，简化了维护(例如，如果修改了超类的定义，子类就会自动获得新的行为)。每种面向对象的语言都允许重定义的方法调用超类上的方法(参见实践 4)。

### 实践 4

下面是一个 Java 例子：

```
void initialize() {
    // Invoke the inherited initialize method
    super.initialize();
    // Now do the extra stuff
    ...
}
```

在重新定义元素时，不要改变其含义。在实现子类时，千万不要忘了与超类用户的约定：任何人都可以假定子类与超类的工作方式完全相同，只是子类有其他知识和行为。例如，为 `LinkedList` 重新定义 `contains` 时，如果参数不在列表中，该方法返回 `true` 就是不合理的。

## 3.6 实现栈类

下面详细探讨通过共享重用代码的理念。使用 `Stack`(如图 3-7 所示)，可以把对象放在栈的最上面、查看最上面的对象、确定栈是否为空、从栈的顶部弹出一个对象。

所以，我们需要一个 `Stack` 类，它带有如下四个消息：

- `push():` 把对象添加到栈的顶部
- `peek():Object:` 返回栈顶的对象
- `isEmpty():boolean:` 如果栈中没有对象, 就返回 true。
- `pop():Object:` 从栈顶删除一个对象, 并返回该对象。

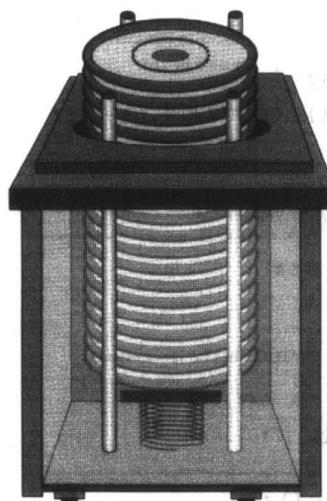


图 3-7 弹簧支撑的盘分配器

现在, 面向对象的软件开发并不意味着编写代码, 所以我们应首先查找类似的已有类。在搜索过程中, 会遇到 `LinkedList` 类, 它是前面开发的一部分。它有可以重用的四个消息:

- `addElement():` 在列表的尾部添加一个对象
- `lastElement():Object:` 返回列表尾部的对象
- `numberOfElement():int:` 返回列表中的对象数
- `removeLastElement():` 删除列表尾部的对象

如果从左向右地查看栈, 而不是从底向上看, 就会发现, 好像我们需要的所有方法都已编写好了, 但消息是错的。必须确定如何把已有的 `LinkedList` 行为融合到新的 `Stack` 类中, 这可以通过继承(如图 3-8 所示)或复合(如图 3-9 所示)来实现。

复合是一个强大的聚合, 组成的对象都在一个复合物中, 组成的对象常常和复合物同时创建和删除。在 UML 中, 为了强调复合比聚合更强大, 使用黑菱形来代替白菱形。

### 3.6.1 使用继承实现栈

我们把新类设置为 `LinkedList` 的一个子类, 如图 3-8 所示。接着, 根据继承的消息定义栈的消息。下面的代码显示了这些定义(在 Java 中, `extends` 表示“继承自”):

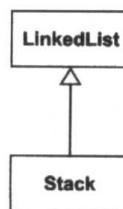


图 3-8 继承的 Stack

```

public class Stack extends LinkedList {

    public void push(Object o) {
        addElement(o);
    }

    public Object peek() {
        return lastElement();
    }

    public boolean isEmpty() {
        return numberofElement() == 0;
    }

    public Object pop() {
        Object o = lastElement();
        removeLastElement();
        return o;
    }
}

```

现在就可以创建栈对象并使用它了：

```

Person aPerson = new Person();
Stack aStack = new Stack();
aStack.push(new Plate("Wedgwood"));
aStack.push(new Plate("Royal Doulton"));
aStack.push(new Plate("Domestic green"));
aPerson.take(aStack.pop());

```

但是，实现 Stack 的这种方式有一个潜在的严重问题。Stack 是 LinkedList 的一个子类，LinkedList 的其他消息也可用于栈对象，这没有什么可惊讶的，因为子类至少要提供与其超类相同的服务。这里的问题是 LinkedList 有一些不适合栈的消息，例如 firstElement，但客户程序员仍能使用它们，例如：

```
aPerson.take(aStack. firstElement());
```

这行代码表示，aStack 的客户删除了栈底部的元素，而这是不允许的。

### 3.6.2 使用复合实现栈

图 3-9 显示的 Stack 带有对 LinkedList 的内部引用。已封装的 LinkedList 的行为由 Stack 使用，但 Stack 客户仍可执行 LinkedList 的其他行为。描述这种情形的另一种方式是：aStack 把其行为委托给 aLinkedList。因为 aLinkedList 是封装的，对它的惟一引用在 aStack 内部。因此，如果使用带有垃圾收集器的纯面向对象语言，在删除 aStack 的同时就会删除 aLinkedList。

使用复合实现的 Stack 如下：

```

public class Stack {

    private LinkedList list;

    public Stack() {
        list = new LinkedList();
    }

    public void push(Object o) {

```

```

        list.addElement(o);
    }
    public Object peek() {
        return list.lastElement();
    }
    public Boolean isEmpty() {
        return list.numberOfElements() == 0;
    }
    public Object pop() {
        Object o = list.lastElement();
        list.removeLastElement();
        return o;
    }
}

```

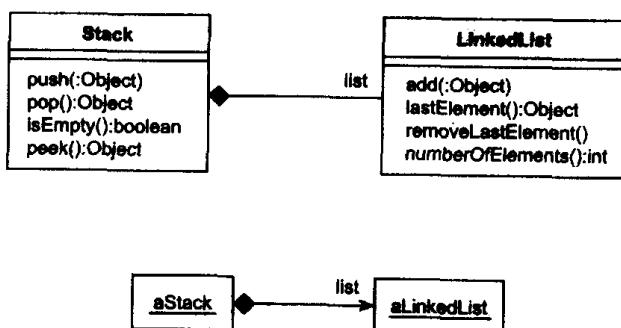


图 3-9 复合的 Stack

这个类提供了栈需要的所有消息，但不能使用下面的代码：

```
aPerson.take(aStack. firstElement());
```

因为不再有 firstElement 消息。

为了实现这个改进的行为，Stack 类必须声明和创建一个字段，其方法必须引用该字段。尽管创建委托的字段和把消息传送给它需要略多的开销，但这是微不足道的，尤其是在当前编译器和运行时系统如此专业化的情况下。

### 3.6.3 继承和复合

继承有一些独特的优点：

- 它是自然的
- 它是优雅的
- 它允许编写一般的代码，例如用于 Fruit 的代码也可以用于 Apple 和 Pear。

但是，继承也有如下问题：

- 很难做得很好
- 在发现设计中的不足时很难改变
- 客户程序员很难理解
- 层次结构会“泄露”给客户代码，也难以改变

复合会得到与继承相同的结果(具体类、具体消息和重用已有的代码)，但有如下优点：

- 较容易开发

- 较容易改变
- 客户容易理解
- 不会泄露给客户代码

在整体上，尤其对于初学者，复合比继承更好，实现适当规模的应用程序时根本不使用继承是完全合理的。继承最好留给专家，尤其是实现可重用代码的大型库(甚至，继承最适合于很好理解的域，例如图形化用户界面、数据库、网络和集合)。

一些语言(如 Eiffel 和 C++)允许私有继承，也称为实现继承。顾名思义，私有继承允许一个类继承另一个类，而继承的元素无需成为新接口的一部分。例如，有这样的功能，Stack 类就可以从 LinkedList 中私有继承：Stack 方法可以直接访问 LinkedList 方法，不必引入委托，也就没有方便或效率等参数；但因为继承是私有的，Stack 的客户不能使用不适合 Stack 的 LinkedList 消息，例如 removeFirstElement，这样纯化论者也会高兴。

私有继承并不在所有的语言中都可用，如果不想把自己固定于一种语言，就应在分析和设计过程中避免这个特性。私有继承通常提供为多重继承的一个附带功能，许多语言都不支持多重继承。

## 3.7 多重继承

在设计继承层次结构时，我们把类一般化为高级抽象类(沿着层次结构向上移动)，还把类特殊化为低级抽象类或具体类(沿着层次结构向下移动)。在向上或向下移动时，即使一般化和特殊化看起来都是有效的，也常常需要在它们之间选择。例如，考虑图 3-10 中的两个继承层次结构。

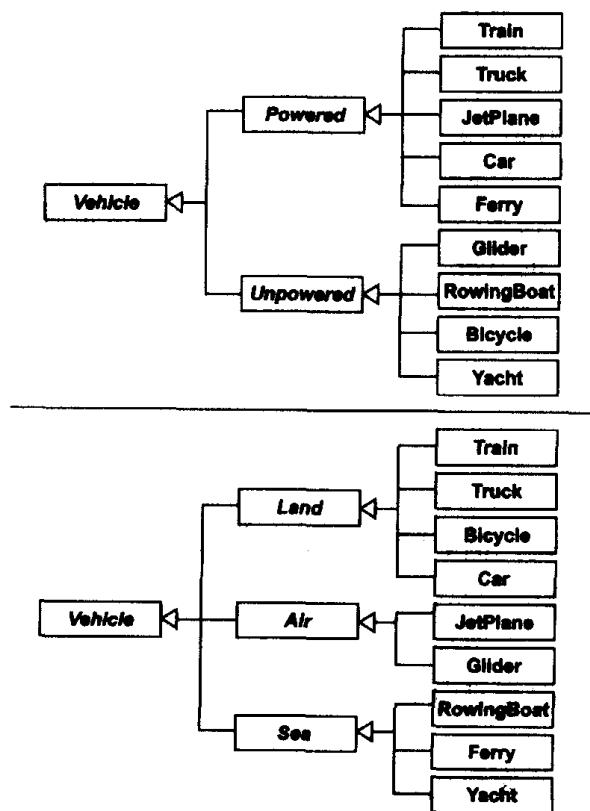


图 3-10 两种继承层次结构

在第一个层次结构中，Vehicles 分类为 Powered 和 Unpowered；在第二个层次结构中，Vehicles 分类为 Land、Air 和 Sea。这样就有了两个层次结构，应选择哪一个？答案取决于问题域是什么(引擎制造商、世界观光旅行家或行李箱制造商)，或者一开始不知道答案，必须试用这两个层次结构，确定它们是否可用。

这种进退两难的局面是单重继承的一个副作用。在单重继承中，类只能有一个父类。单重继承很不错，这就是像 Smalltalk 和 Java 这样的语言只有单重继承的原因。但在多重继承中，每个类都可以有任意多个父类。(Java 有一种多重继承的形式，但只能用于接口，参见第 8 章)。通过多重继承，可以把图 3-10 中的层次结构合并为图 3-11 中的层次结构，综合两者的长处。

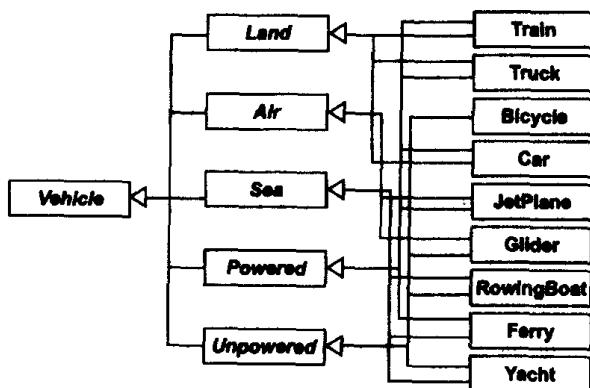


图 3-11 多重继承——综合了图 3-10 中的两种继承层次结构的优点吗？

在图 3-11 中，可以看出多重继承是相当复杂的，不值得采用。目前有许多观点赞同和反对它。多重继承的优点是：

- 它非常强大
- 它允许私有继承
- 它更接近真实情况
- 它允许混合(mix-in)继承

混合继承是一种设计模式，在这种模式下，给主要的类使用单重继承，同时允许各个类继承一个或多个辅助类，每个辅助类都会添加几个简单的元素。例如，Lorry 从 LandVehicle 中继承了大多数重要的元素，再从简单的 Powered 类中继承 engine 属性。

多重继承的缺点是：

- 比较复杂(对于设计人员和客户程序员而言)
- 导致名称冲突
- 导致重复继承
- 使编译器更难编写
- 使运行时系统更难编写

当元素的名称相同，但通过不同的路线继承了不同的实现代码时，就会出现名称冲突。例如，考虑图 3-12 中的层次结构。这里有两个超类，HouseProduct 是为房子购买的物品，EuroProduct 是在欧洲制造的物品。在 HouseProduct 上有一个消息 isPolish，如果相关产品用于上光，该消息就返回 true；相反，在 EuroProduct 上也有一个消息 isPolish，如果相关产品是在波兰制造的，该消息就返回 true。

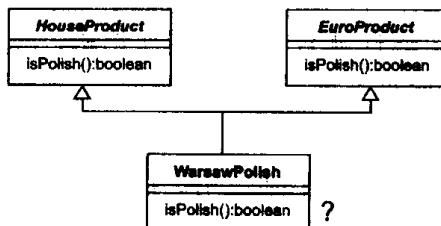


图 3-12 名称冲突

一旦引入了共有的子类 `WarsawPolish`, 它表示在华沙(波兰首都)生产的上光剂, 就会出现问题: 新类有两个含义不同的消息, 但它们不能使用相同的名称, 因为这会使 `aWarsawPolish.isPolish()` 出现混乱。如果重新给两个消息命名, `WarsawPolish` 的客户就会发现, 尽管据说在 `HouseProduct` 和 `EuroProduct` 上有消息 `isPolish`, 但 `WarsawPolish` 没有。

重复继承表示, 从多条路径上继承了同一个元素。例如, 在图 3-13 中, 如果 `getName` 方法在 `Employee` 类中定义了, `Assignee` 就会从 `UKEmployee` 和 `USEmployee` 中继承 `getName`。在这个简单的例子中, 编译器就很容易认为, `Assignee` 应有一个 `getName` 方法。但是, 如果 `UKEmployee` 或 `USEmployee` 选择重新定义 `getName` 方法, 事情就会比较复杂; `Assignee` 会有两个 `getName` 方法(此时, 如何处理名称冲突?)或者只有一个 `getName` 方法(此时, 如何选择 `getName` 方法?)。

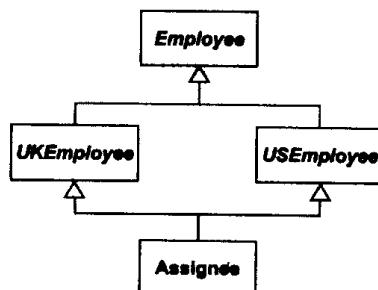


图 3-13 重复继承

编写高效的编译器和运行时系统, 来处理多重继承的问题很难解决。但是, 如果这些就是砍掉多重继承的原因, 向往这些功能的程序员就不可能满意。为了支持多重继承, 聪明的设计人员可以通过好的编程技巧来解决所有的问题。

但是, 多重继承在实现语言中可能不可用, 所以最好不要使用它, 尤其在分析时不要使用。如果在设计过程中使用了多重继承, 就应在主干上使用单重继承, 在适当的时候添加一些简单的混合类。

下面总结了常见语言中的多重继承功能:

- Eiffel 提供了直接的多重继承, 还提供了私有继承和混合继承。Eiffel 有丰富的功能来处理名称冲突和重复继承。
- Smalltalk 只提供了单重继承, 多重继承功能已在试验, 但成功有限。
- C++ 提供了一些多重继承功能, 但该功能并不完整, 设计和实现都不太好, 所以应避免使用它们。

- Java 为类提供了单重继承，为接口(没有方法的抽象类)提供了多重继承。这是一个很好的折衷方案，允许一定程度的多重继承建模和混合继承(仅用于消息)。Java 没有特殊的功能来处理名称冲突和重复继承，但允许重复继承的简单形式。

## 3.8 使用继承的规则

- 不要过度使用：不要认为必须使用继承，甚至总是使用继承。继承有替代方式，例如复合和使用属性(例如，Car 类使用 color 属性，就比使用三个类 Car、RedCar 和 BlueCar 更好)。
- 类应是其超类的一个类型：只要从子类 X 中派生出 Y，就要问问自己：“Y 是 X 的一种类型吗？”例如，Orange(桔子)是一种水果，Truck(卡车)是一种陆上交通工具，所以它们是有效的。相反，Potato(土豆)不是一种水果，Airplane(飞机)不是一种陆上交通工具，所以这些不是好的子类(一些开发人员使用“子类型”这个术语来表示“我遵守规则”，使用“子类”这个术语来表示“我没有遵守规则”)。
- 类应是其超类的扩展：在子类中，应确保只添加新的特性，不要删除、禁用或重新解释特性，来分解超类。

## 3.9 小结

本章介绍了继承，这是共享代码和高级建模的工具：

- 类可以组合为更一般的概念，类可以获得(继承)父类中的一些特性。
- 抽象类至少有一个没有代码的方法(抽象方法)；在具体类中，所有的方法都包含代码行。

## 3.10 课外阅读

面向对象设计和编程理论的一本著名图书是[Meyer 97]，它介绍了有关对象的所有应了解和不应了解的内容。Meyer 使用 Eiffel 语言演绎了他的理念，但 Eiffel 的语法非常简单，完全基于面向对象的概念，所以是对主要内容的一种补充。另外，从选择编程语言的角度来看，甚至是最严厉的理论家也会觉得无法不同意 Meyer 讲过的内容。

## 3.11 复习题

1. 为什么重新定义方法的功能在面向对象的编程中非常重要？(多选题)  
(a) 因为它可以给方法添加额外的工作  
(b) 因为它可以引入抽象方法，再重新定义为具体方法  
(c) 因为它可以在子类中提供更准确或更快捷的定义  
(d) 因为它可以禁用子类中的方法  
(e) 因为它可以改变方法的含义

2. 下面哪些关于多重继承的陈述是正确的? (多选题)
  - (a) 它提供了更多的建模选择
  - (b) 更难编写编译器和运行时系统
  - (c) 简化了继承层次结构
  - (d) 解决了重复继承的问题
3. 下面哪些陈述是正确的? (多选题)
  - (a) 大多数超类都是抽象的
  - (b) 继承优于复合
  - (c) 大多数超类都是具体的
  - (d) 复合优于继承
4. 在 UML 图中, 抽象类如何与具体类区分? (单选题)
  - (a) 具体类显示在虚线框中
  - (b) 抽象类的标签显示为斜体
  - (c) 具体类的标签显示为斜体
  - (d) 抽象类显示在虚线框中
5. 什么是抽象类? (单选题)
  - (a) 对象
  - (b) 没有方法的类
  - (c) 没有具体子类的类
  - (d) 至少有一个未定义消息的类
  - (e) 接口
6. 哪个术语描述了 Stack 类使用 List 的内部实例实现? (单选题)
  - (a) 关联
  - (b) 特殊化
  - (c) 一般化
  - (d) 复合
  - (e) 单一化
7. 在图 3-14 中, 两个图的区别是什么? (单选题)
  - (a) 在图 1 中, color 是公共的, 在图 2 中, color 是私有的
  - (b) 图 2 指出汽车的 color 可以删除和替换
  - (c) 图 1 显示的是抽象类, 图 2 显示的是具体类
  - (d) 它们没有区别



图 3-14 复习题 7

## 3.12 复习题答案

1. 重新定义方法的功能在面向对象的编程中非常重要，是因为：
  - (a) 因为它可以给方法添加额外的工作
  - (b) 因为它可以引入抽象方法，再重新定义为具体方法
  - (c) 因为它可以在子类中提供更准确或更快捷的定义
2. 多重继承：(a) 提供了更多的建模选择；(b) 更难编写编译器和运行时系统
3. 下面的陈述是正确的：(a) 大多数超类都是抽象的；(d) 复合优于继承
4. 在 UML 图中，抽象类与具体类的区分方式是 (b) 抽象类的标签显示为斜体
5. 抽象类是 (d) 至少有一个未定义消息的类
6. 描述了 Stack 类使用 List 的内部实例实现的术语是：(d) 复合
7. 两个图的区别是：(d) 它们没有区别

# 第4章 类型系统

与继承一样，人们普遍认为类型系统是比较基本的，但它相当复杂，所以读者在第一次阅读本书时，可以跳过本章。即使不了解类型系统也可以继续阅读本书。但是，如果跳过本章，本书后面的一些论题就不容易理解了。

## 学习目的：

- 理解类型系统的含义
- 理解多态性
- 理解隐式和显式类型转换
- 了解 Java 模板提供的一般性

## 4.1 引言

类型系统是一个简单的概念：它是一组禁止误用值(原型和对象)的规则。通常，该系统迫使我们在实际使用值之前声明如何使用它，这样，编译器和运行时系统就可以禁止潜在的误用。

类型系统的一个简单例子是声明变量总是包含特定类型的值：

```
int i;  
Employee fred;
```

但为什么我们会误用值呢？最常见的原因如下：

- 不理解值的用法
- 拼错了值的名称

除了防止误用值之外，类型系统还有另外两个优点：确保提供代码的某些说明(fred 是一个员工)；提高运行时系统的性能，因为编译器和运行时系统提供了代码要做什么(编译器)或代码正在做什么(运行时系统)的更多信息。

## 4.2 动态和静态类型系统

类型系统可以是静态的(static，由编译器完成)，也可以是动态的(dynamic，由运行时系统完成)。类型系统的这两个变体可以确保，程序员不会误用值：静态类型系统禁止编译期间的误用，动态类型系统检查是否出现误用，并禁止它。

Smalltalk 是原型的、基于动态类型的语言。在 Smalltalk 中，程序员常常使用命名约定声明值的期望类型。例如，为了确保 addEmployee 方法的参数总是指向 Employee 对象，程序员可以给它命名为 anEmployee。使用 addEmployee 的人根据参数名假定这些参数要传送给 Employee。(类似的命名约定也可以应用于字段和本地变量)。

但 Smalltalk 是基于动态类型的，所以客户程序员仍必须确保不通过 Banana 调用 addEmployee。假定在 addEmployee 方法的内部，要把 getPayrollNumber 消息传送给 anEmployee。因此，如果通过 Banana 来传送，getPayrollNumber 就会传送给 Banana，显然这是没有意义的。Smalltalk 对误用 Banana 的响应是生成运行时错误，并给出一个消息，说明 Banana 不理解 payrollNumber。

Smalltalk 的方法虽然简单，但还是构建出了类型系统，因为运行时系统可以禁止客户程序员误用 Banana——该对象必须回答一个它不理解的问题。但是，我们必须在运行代码后才能发现问题。更糟糕的是，如果测试代码给支持 Banana 的 anEmployee 传送消息，就不会发现错误，除非编写更多的代码，或者在即时设置中运行系统。

如果在 Java 中编写 addEmployee 方法，就编写为参数声明类型：

```
public void addEmployee(Employee anEmployee) {  
    ...  
    pay = anEmployee.getPayrollNumber()  
    ...  
}
```

这里，程序员告诉编译器，客户传送的参数必须是 Employee(该消息不返回任何信息，因为使用了 void 关键字)。因此，编译器会拒绝编译下面的代码段：

```
aPayroll.addEmployee(new Banana());
```

在 Java 中，编译器而不是运行时系统会禁止误用 Banana。

Smalltalk 程序员认为，动态指定类型是比较好的方案，因为：

- 编译很快，也很简单
- 程序员可以工作得更快，程序员的想法更容易转变为程序，因为他们不需要停止思考在当前的环境下必须使用什么类型的对象。
- 没有静态编译器，便于整体测试。
- 面向对象的代码可以重用，所以最终总是能发现错误。

Java 程序员则认为静态指定类型比较好，因为：

- 编译仍相当快，而且程序员并不关心编译器的编写有多困难。
- 可以提高运行时系统的性能
- 可以找出拼写错误
- 强制提供代码的一些说明文档

在面向对象的术语中，静态类型系统保证不把消息传送给对象，除非对象有相应的方法。

## 4.3 多态性

多态性(polymorphism)派生于希腊单词 poly(即许多)和 morph(即形状)。所以多态(polymorphic)表示“有许多形状”。可以把这个术语应用于变量和消息：多态变量(polymorphic variable)表示值在不同的时刻表示不同的类型；多态消息(polymorphic message)表示有多个方法与对象相关。

在纯面向对象的语言中，所有的非原型变量都是多态的，所有的消息都是多态的。下面依

次介绍它们。

### 4.3.1 多态变量

下面的 Java 声明表示，`t` 总是指向 `Truck` 类型的对象：

```
Truck t;
```

因此，下面的赋值是有效的，可以得到如图 4-1 所示的结果：

```
t = new Truck();
```



图 4-1 把 `Truck` 变量关联到 `Truck` 对象上

考虑图 4-2 中的类层次结构，它说明，`Truck` 是一种陆上交通工具(`LandVehicle`)。根据这个层次结构，就可以把 `Truck` 当做陆上交通工具来对待。所以，下面的语句也是有效的，可以得到如图 4-3 所示的结果：

```
LandVehicle lv = new Truck();
```

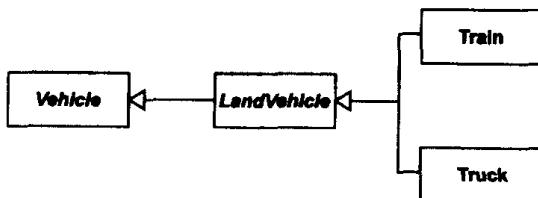


图 4-2 `Truck` 继承

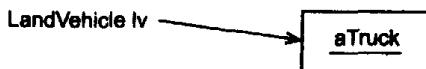


图 4-3 把 `LandVehicle` 变量关联到 `Truck` 对象上

这有点令人惊讶，但考虑一下：刚才告诉编译器，把 `lv` 用作 `LandVehicle`，就能给它发送 `LandVehicle` 消息；而所有的 `LandVehicle` 消息都发送给 `Truck`，所以一切正常。

既然可以使 `lv` 指向 `Truck`，就可以以同样的方式使 `lv` 指向 `Train`(如图 4-4 所示)：

```
lv = new Train();
```

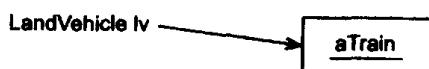


图 4-4 把 `LandVehicle` 变量关联到 `Train` 对象上

前面 lv 指向 Truck，但是现在它指向 Train，所以它必须是一个多态变量。这是很直观的：lv 是 LandVehicle，所以可以指向 LandVehicle 的任意类型。

变量的多态性是由继承控制的。例如，Orange 不是一种 LandVehicle，所以下面的语句是不对的：

```
lv = new Orange();
```

多态允许把变量关联到子类对象上，但不能反向。例如，Vehicle 不是一种 LandVehicle，所以下面的语句是不对的：

```
lv = new Vehicle();
```

如果允许这么做，就可以把任意 LandVehicle 消息发送给 Vehicle，但 Vehicle 肯定不能理解其中一些 LandVehicle 消息。

#### 4.3.2 多态消息

在纯面向对象的语言中，任何消息都可以关联多个方法。这是因为方法在多个类中是独立的，或者方法由子类重新定义了。重新定义的方法一般有类似的算法，但独立定义的方法通常有完全不同的算法。

例如，考虑图 4-5 中的四个类，其中忽略了人类和鸟类的关系。这些类都有一个 flyTo 消息，它表示“飞到给定的地方”；该方法在 Bird 上是抽象的，在其他三个类上是具体的。这样，flyTo 就有三个实现代码，所以是一个多态消息。

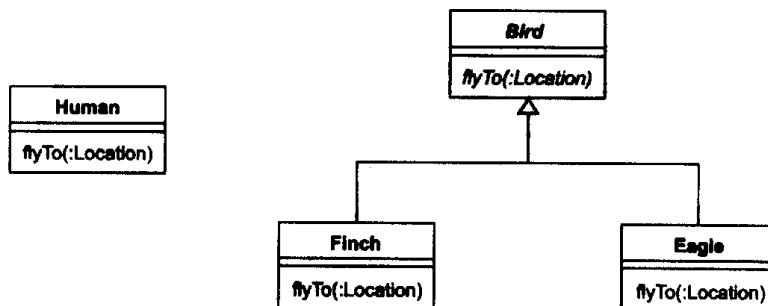


图 4-5 多态动物消息

Human 上的 flyTo 方法可以与 Bird 上的 flyTo 方法做比较，但其含义是完全不同的：对于鸟类，飞行涉及鼓翼而飞和滑行，而对于人类，就需要乘飞机。

多态消息与继承相关，其关联方式与多态变量相同。例如，在下面的代码段中，运行时系统会在 Finch 上执行 flyTo 方法，而不是 Bird 上执行(在 Bird 上的 flyTo 方法是抽象的，所以没有可执行的代码)：

```
Bird b = new Finch();
b.flyTo(someLocation);
```

这里介绍了抽象方法的功能：即使没有在 Bird 上定义 flyTo 消息，也可以声明“所有的鸟都有一个 flyTo 消息”——运行时系统会执行正确的实现代码。如果在上面的代码段中添加一些代码，告诉 Eagle 飞到 someLocation，就要涉及较多的滑行和较少的鼓翼而飞。

```
Bird b = new Eagle();
b.flyTo(someLocation);
```

多态变量只是一个占位符——在这个占位符中要填充变量指向的对象。所以在下面的代码段中，有一个 Cat，但有两个指向它的引用(每个引用都是 Cat 的另一个名称)：

```
Cat tiddles, tom;
tiddles = new Cat("Hfrrr");
tom = tiddles;
```

## 4.4 动态绑定

动态绑定表示，在运行期间把消息关联到方法上。这是面向对象语言处理多态变量和重定义方法的方式。如图 4-6 所示，有两个抽象类 Shape 和 Quadrilateral 和一个具体类 Square。这三个类都有一个 getPerimeter 消息：在 Shape 上，该消息是抽象的；在 Quadrilateral 上，它是四条边长的总和；在 Square 上，它是一条边长乘以 4 的结果。aSquare 是 Square 类的一个实例，在图 4-6 中就显示为 UML 依赖(dependency)，即未封口的虚线箭头。

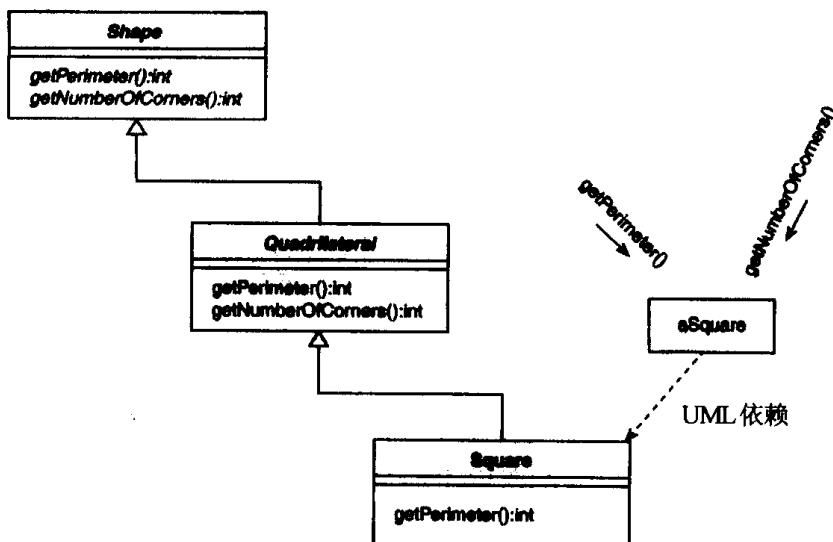


图 4-6 动态绑定和重定义

在下面的代码段中，运行时系统如何获知在发送 getPerimeter 消息时，执行三个方法中的哪一个？

```
Shape sh = new Square();
int i = sh.getPerimeter();
```

从概念上看，接收消息的对象(这里是 Square)知道它自己的类，所以就到类中查找方法；而 Square 定义了 getPerimeter 方法，所以执行这个方法。

图 4-6 还显示了一个 getNumberOfCorners 消息，该消息在 Shape 上是抽象的，在 Quadrilateral 上是具体的。在下面的代码段中，Square 再次到它的类中查找 getNumberOfCorners 方法：

```
int j = sh.getNumberOfCorners();
```

这次没有匹配的方法，所以继续在 Quadrilateral 超类中搜索。Quadrilateral 中有一个匹配的方法，因此就执行这个方法。这个动态绑定算法看起来有点慢，但这只是一个概念算法，可以更有效地实现，尤其是在静态类型系统的帮助下，效率就更高。

有了静态类型，就可以确保动态绑定会在超类链中找到具体的方法（因为每个变量的类型都必须声明，而且不能创建抽象的对象）。有了动态类型系统，就可能遇到抽象方法，或者到达层次结构的最顶层，此时会出现运行时错误消息。

对于多重继承，动态绑定比较复杂，但仍是可行的，也不比单重继承慢多少。

## 练习 2

图 4-7 显示的 Shape 层次结构多了一个类 Triangle。考虑下面的代码段：

```
Shape sh;
Triangle tr = new Triangle();
Square sq = new Square();
```

下面哪个赋值是正确的？（考虑“它是一种……”）

1. sh = tr;
2. sh = sq;
3. sq = tr;
4. tr = sq;
5. tr = sh;
6. sq = sh;

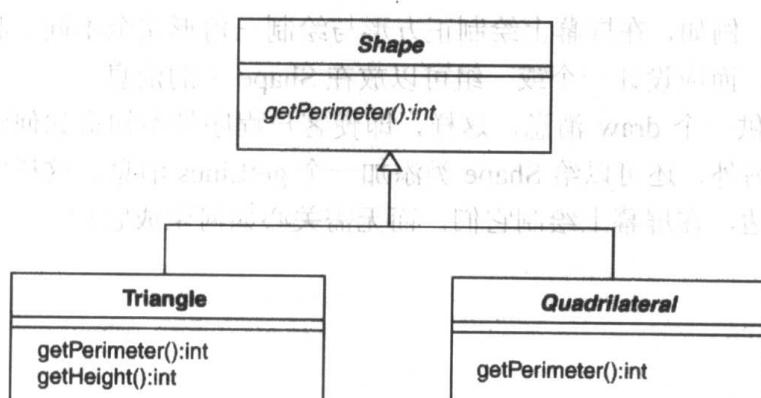


图 4-7 多态性和类型系统

**练习 3** 在继承类中声明方法，若由子类调用父类方法

如果把 sh 关联到 square 上：  
`sh=sq;`

发送下面的消息会有什么结果(假定静态类型系统)?

1. sh.getPerimeter();
2. sq.getPerimeter();
3. tr.getPerimeter();
4. tr.getHeight();
5. sq.getHeight();
6. sh.getHeight();

## 4.5 多态性规则

在结束多态性的主题之前，考虑下面的样式规则：总是使用尽可能高的抽象级别来编程。也就是说，总是把字段、本地变量和方法参数的类型声明为继承层次结构中最高的类，再让多态性完成其他工作。

制定这个规则的原因是，所使用的抽象级别越高，代码的可重用性就越大。例如，根据 Shape 编写的任何代码可以应用于任何类型的 Shape——Square、Quadrilateral、Triangle 和其他以后添加的子类。但为 Square 编写的代码就只能用于 Square 对象，为 Triangle 编写的代码就只能用于 Triangle 对象。

类层次结构的设计人员还有一个责任：他们应确保，只要可能，每个类的类型就隐藏在应用于所有相关类的一般消息中。例如，在屏幕上绘制正方形与绘制三角形完全不同。我们不应把这些区别展示给客户程序员，而应设计一个或一组可以放在 Shape 上的消息。

例如，可以在 Shape 上提供一个 draw 消息，这样，即使客户程序员不知道如何绘制，也可以告诉 Shape 绘制它自己。另外，还可以给 Shape 类添加一个 getLines 消息，这样客户程序员就可以检索组成图形周长的边，在屏幕上绘制它们，而无需关心如何生成它们。

## 4.6 类型转换



在静态类型的语言中，当从一个上下文给另一个上下文传送值时，需要确保新上下文与旧上下文兼容(例如，不能把 Banana 传送给需要 Employee 的上下文中)。甚至在确保新上下文与旧上下文兼容时，仍需要把值转换为另一种类型(例如，把 Employee 对象从 Employee 上下文传送到 Person 上下文中)。

下面是值改变上下文的三种情况：

- 计算表达式：在表达式  $2+2$  中，编译器会把两个整型数相加，生成另一个整型数。这没有问题，因为新上下文与旧上下文都是整型。

另一方面，如果表达式是  $3.75+2$ ，编译器要处理的是一个实数和一个整数，但结果必须是一个实数。在这种情况下，整数与新上下文兼容，因为可以把任意两个类型的数字加在一起，

生成另一种类型的数字，所以该表达式是有效的。但是，计算机以不同的方式表示整数和实数。为了执行上面的代数运算，编译器必须先把整数转换为实数(因为整数可以表示为实数，而实数不能表示为整数)。

一些语言允许在表达式中加入对象，例如 Java 表达式 “The date is ” + aDate。这些表达式必须遵守与上述原型例子类似的规则。在上面的例子中，编译器应把 Date 传送给 `toString` 消息，将它转换为 String。

- 赋值：Person p = new Person() 是不需要转换类型的赋值，因为该表达式的上下文与变量的上下文都是 Person。但是，赋值 Person p = new Employee() 需要从 Employee 转换为 Person。Person 指针的表示方式与 Employee 指针是相同的，但编译器必须确保 Employee 可以用作 Person。如果 Person 是 Employee 的直接或间接超类，这个转换就是安全的，因为客户程序员只能使用 Person 类的元素。

`float f = 2` 是一个需要转换类型的原型赋值例子，其中，必须把整数 2 转换为浮点数。

- 参数传送：这与赋值相同。把实参的值赋予声明的参数，例如 `aUniversity.enrollPerson(new Employee())`。`aLiquid.setBoilingPoint(100)` 是一个原型参数传送的例子，其中参数声明为 float。

把值从一种类型转换为另一种类型称为类型转换(casting，因为是把值转换为新的类型)。上面的例子都是隐式转换，因为客户程序员不必做任何工作：编译器可以确定转换是否有效，并进行转换。一般情况下，如果新上下文的取值范围比旧上下文大，就可以进行隐式转换。在原型值中，“取值范围较大”表示新上下文可以容纳旧上下文的所有值。对于对象值，“取值范围较大”表示新上下文是旧上下文的直接或间接超类。

动态型的语言在不同上下文之间传送值是非常简单的。例如，在以 Banana 为参数，执行 Smalltalk 方法 `addEmployee` 时，不需要进行兼容性测试或类型转换，只要 `addEmployee` 只使用 Employee 和 Banana 都有的元素，该方法就可以运行，不会出错(如果尝试把钱转帐到 Banana 的银行账户上，还是会有问题，但它会显示为运行时错误)。

## 4.7 显式类型转换

隐式类型转换允许在表达式中合并使用不同类型的值，把不同类型的值赋予变量，从而使静态型的编程语言更丰富。我们还可以更进一步：允许程序员使用显式类型转换，从一种上下文转换为兼容、但取值范围较小的上下文。

值的显式类型转换可能会出问题，但指针的显式类型转换没有这么糟糕。这是因为，在转换值的类型时，是在改变值的存储方式——使用取值范围较小的表示方式来存储值，这可能丢失信息。相反，如果转换的是指针，就是改变访问值的方式，但值本身保持不变。

在纯面向对象的语言中，原型总是作为值来访问，对象也总是通过指针来访问(高效、具有多态性)。相反，在像 C++ 这样的语言中，原型和对象都可以作为值来访问，也都可以通过指针来访问：程序员可以选择。其优点是 C++ 程序员必须更小心。下面的讨论都假定使用纯面向对象的语言，而且显式类型转换仅在需要时才使用：允许把隐式转换变成显式转换，但这是没有用的。

Java 允许程序员编写下面的语句：

```
int i = (int) 3.75
```

(int)把实数 3.75 显式转换为整数。程序员注意到不可能从 float 隐式转换为 int，所以告诉编译器把一个正方形的木钉钉到一个圆孔中。实际上，把正方形的木钉钉到圆孔中，会剪掉木钉的尖角。在编程中也是这样：实数的小数部分会被舍弃。

上述类型转换是可行的，因为 int 和 float 是兼容的，而 boolean b = (boolean) 3.75 是没有意义的。一般说来，如果在一个方向上可以进行隐式转换，就可以在另一个方向上进行显式转换。

对象的指针如何进行显式转换？下面的例子可行吗？

```
Employee e = (Employee) new Person();
```

对于纯面向对象的语言，答案是不可行。转换指针不会改变值的表示方式，只改变访问值的方式。所以前面的例子假定编译器允许把 Person 当做 Employee 来对待。如果编译器允许，就应尽可能使用 Employee 支持而 Person 不支持的元素(例如 getPayrollNumber)。

下面两个语句可行吗？

```
Person p = new Employee();
Employee e = (Employee) p;
```

在这个例子中，e 试图指向的对象是 Employee，所以显式转换是安全的——不能通过 Employee 指针误用 Employee。但编译器是比较死板的。为了使这两个语句有效，编译器必须分析它们。如前所述，编译器不会进行这种分析，因为它比较难，通常是不可能的。

但是，程序员偶尔可以这么做。所以，编译器谋取运行时系统的帮助，允许进行显式转换，但这只是因为 Employee 是 Person 的一个子类。之后，运行时系统需要在赋值时检查，p 指向的对象是否为 Employee 的一个实例(或其直接或间接子类的一个实例)，如果不是，就生成一个运行时错误。

对象指针的显式转换涉及到运行时系统，所以有时也称为动态转换。一些人还使用“向上转换(upcast)”和“向下转换(downcast)”来表示对象指针的隐式转换和显式转换，因为隐式转换是沿着层次结构向上转换，显式转换是沿着层次结构向下转换。

向下转换不常用，应只在使用一般消息(返回类型必须与所有对象兼容的消息)检索对象时使用。例如，前面介绍的 Stack 类有一个一般消息，可以弹出一个对象：无论最顶层的对象是什么，它总是返回 Object 类型的指针(它还有一个一般消息，可以压入一个对象：参数类型是 Object，这可能需要向上转换)。

一般消息允许栈处理所有类型的对象——汽车、字符串或其他对象。但它们不允许客户程序员进行如下操作：

```
aStack.push(new Plate("Domestic blue"));
Plate aPlate = aStack.pop();
```

第一行代码可以执行，因为它是一个隐式的向上转换，但第二行代码不会执行。可是有了向下转换的帮助，客户程序员可以进行如下操作：

```
aStack.push(new Plate("Domestic blue"));
Plate aPlate = (Plate)aStack.pop();
```

## 4.8 使用模板进行泛化

一些语言，如 Java，很少需要进行向下转换，因为它有一个功能，称为泛化(genericity)，比向下转换更好。一般类使用一个或多个类参数引用它希望处理的对象类型。在 Java 中，一般类称为模板。例如，Java 程序员可以定义 Stack 类，如下所示：

```
public class Stack<X> {  
  
    private List<X> list;  
  
    public Stack<X>() {  
        list = new LinkedList<X>();  
  
    }  
    public void push(X anX) {  
        list.addElement(anX);  
    }  
    public X peek() {  
        return list.lastElement();  
    }  
    public boolean isEmpty() {  
        return list.numberofElement() == 0;  
    }  
    public Object pop() {  
        X anX = list.lastElement();  
        list.removeLastElement();  
        return anX;  
    }  
}
```

先不考虑新语法，可以看出这个类的重要部分。最重要的方面是类表示为 X，这是客户程序员创建的实际 Stack 的占位符。还要注意字段的类型比实现类更泛化(这正是面向对象代码的一般样式)。

现在，客户程序员可以使用下面的代码处理栈：

```
Stack<Plate> s = new Stack<Plate>();  
s.push(new Plate("Hospital white"));  
Plate p = s.pop();
```

因为程序员声明，Stack 包含 Plate 对象，所以在执行 s.pop()前不需要显式转换。

如果程序员想要一个更快的栈，就可以编写下面的代码：

```
Stack<QuickTrickBrick> s = new Stack<QuickTrickBrick>();
```

Stack 类是一个相当简单的一般类型，因为它的方法不需要使用它包含的对象元素。如果要对栈对象的类做一些假设，就需要更多的编码。例如，假定 Stack 对象只包含食物，就应允许 Stack 方法把压入的对象当做 Food，对种类计数，等。

如果要对类参数做一些假设，就必须使用有限制的泛化(constrained genericity)：给类参数添加约束。下面是用 Java 编写的、修改过的 Stack 类，它只能用于不同种类的 Food：

```

public class Stack<X extends Food> {
    ...
    private int caloriesConsumed;
    ...
    public X pop() {
        X anX = list.lastElement();
        list.removeLastElement();
        // Now add anX's calories to the total,
        // only possible because X is always a Food
        caloriesConsumed = caloriesConsumed + anX.getCalories();
        return anX;
    }
    ...
}

```

## 4.9 小结

本章的主要内容如下：

- 类型系统可迫使我们声明使用值的方式，防止误用它。静态类型系统在编译期间检测误用，而动态类型系统在运行期间检测误用。
- 多态性允许变量存储不同类型的值，允许把一个消息关联到多个方法上。可以使用的值或方法的特定类型在运行期间确定。
- 对象类型之间的转换：在隐式转换中，编译器可以自动在变量的类型之间转换；在显式转换中，程序员必须指定对象要转换为另一种类型。
- 在 Java 的模板中，一般类使用参数引用它希望处理的对象类型。

## 4.10 课外阅读

Java 模板是最近添加到该语言中的概念，其更多信息可参阅 Java 网站 [java.sun.com](http://java.sun.com)。

## 4.11 复习题

1. 如图 4-8 所示，下面发送的消息对应哪些方法(按照给出的顺序)。(单选题)

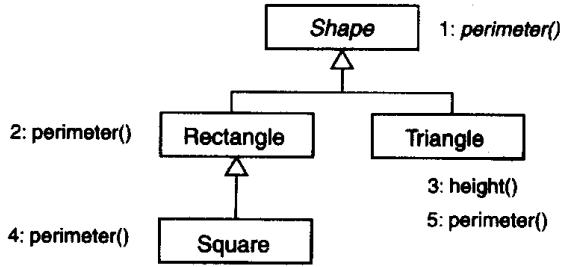
```

tr.height();
sh.perimeter();
sq. height();
sq. perimeter();
sh. height();
tr. perimeter();

```

- (a) 3, 1, 无(错误), 4, 无(错误), 5

- (b) 3, 5, 无(错误), 4, 3, 5
- (c) 3, 1, 无(错误), 4, 3, 5
- (d) 3, 5, 无(错误), 4, 无(错误), 5



```

Shape sh;
Triangle tr = new Triangle();
Square sq = new Square();
sh = tr;
  
```

图 4-8 复习题 1

2. 如图 4-8 所示, 编译器允许执行下面发送的消息吗? (多选题)
  - (a) `sh.perimeter();`
  - (b) `tr.perimeter();`
  - (c) `sh.height();`
  - (d) `sq.height();`
  - (e) `sq.perimeter();`
  - (f) `tr.height();`
3. 如图 4-8 所示, 编译器允许执行下面哪个赋值语句? (多选题)
  - (a) `sq=sh;`
  - (b) `sh=tr;`
  - (c) `tr=sq;`
  - (d) `tr=sh;`
  - (e) `sh=sq;`
  - (f) `sq=tr;`
4. 面向对象的类型系统可以用于: (单选题)
  - (a) 提高运行时的性能
  - (b) 防止误用类
  - (c) 避免拼写错误
  - (d) 确保所有的消息都调用具体的方法
  - (e) 说明
  - (f) 上述都正确
  - (g) 上述都不正确
5. 术语“多态性”的含义是什么? (多选题)
  - (a) 变量可以在不同的时候指向不同类的对象
  - (b) 有相同签名的消息可以在不同的时候调用不同的方法

- (c) 所有面向对象的编程语言都是不同的。
  - (d) 所有面向对象的方法都使用不同的表示法。
6. 什么是一般类? (单选题)
- (a) 没有声明版权的类
  - (b) 表示所有对象的类
  - (c) 把其他类作为参数的类

## 4.12 练习 2 的答案

1. 正确。Shape 变量一定能指向 Triangle, 因为 Triangle 是一种 Shape。
2. 正确。Square 是一种 Shape。
3. 试图用 Square 变量指向 Triangle 一定是错误的, 因为 Triangle 不是一种 Square。
4. 与上一题类似, Square 不是一种 Triangle。
5. Shape 不是一种 Triangle。
6. Shape 不是一种 Square。

## 4.13 练习 3 的答案

1. getPerimeter 消息是在 Shape 上引入的, 所以是正确的; 而 sh 指向 Square, 所以可以在 Square 上调用 getPerimeter 方法。
2. 可以在 Square 上调用 getPerimeter 方法。
3. 可以在 Triangle 上调用 getPerimeter 方法。
4. 编译器可以在 Triangle 上执行 getHeight 方法。
5. 这是错误的, 因为 Square 上没有 getHeight 消息, 它的超类上也没有 getHeight 消息, 编译器会拒绝执行这个操作。
6. 我们试图给 Shape 变量发送 getHeight 消息。如果 sh 指向 Triangle, 消息就可以发送出去; 但 sh 可以指向 Square, 此时消息就是无效的。静态类型系统的编译器会生成一个错误, 并发出警告: 消息可能发送到不知道如何处理它的对象上, 所以编译器会拒绝该语句。

## 4.14 复习题答案

1. 如图 4-8 所示, 所发送的消息对应的方法是(d) 3, 5, 无(错误), 4, 无(错误), 5。
2. 如图 4-8 所示, 编译器允许执行的消息是:
  - (a) sh.perimeter();
  - (b) tr.perimeter();
  - (e) sq.perimeter();
  - (f) tr.height();
3. 如图 4-8 所示, 编译器允许执行的赋值语句是
  - (b) sh=tr;

- (e)  $sh=sq;$
- 4. 面向对象的类型系统可以用于 f
  - (a) 提高运行时的性能
  - (b) 防止误用类
  - (c) 避免拼写错误
  - (d) 确保所有的消息都调用具体的方法
  - (e) 说明
- 5. 术语“多态性”的含义是:
  - (a) 变量可以在不同的时候指向不同类的对象
  - (b) 有相同签名的消息可以在不同的时候调用不同的方法
- 6. 一般类是: (c) 把其他类作为参数的类

# 第 5 章 软件开发的方法学

本章将介绍软件开发进程、编写优秀软件的步骤和在此过程中得到的产品。总之，过程、步骤和其产品称为方法学。

## 学习目标：

- 理解软件生产的经典阶段
- 比较静态(结构化)建模和动态(基于时间的)建模
- 理解 UML 的表示法

## 5.1 引言

软件，尤其是许多人一起开发的大型软件，应使用某种方法来开发。甚至由一个人开发的小型软件也应通过某种方法进行改进。

方法学是做事的系统方法，它是一个可接受的过程，从软件开发的早期阶段(有一个想法或一个新的商业机会)到已安装系统的维护，都可以遵循这个方法学。除了过程之外，方法学还应指定在此过程中要生产什么产品(以及该产品采用什么形式)。方法学也包括用于资源管理、规划、调度和其他管理任务的建议或技术。

优秀的、适用范围广的方法学是成熟软件业的基础——其他方法都不能解决问题。最糟糕的是陷入一片混乱，开发小组的成员东奔西跑，想弄清楚到底如何开发他们要实现的系统。稍有安慰的是，一个组织的业余方法学家设计了一个事实上的开发过程——该组织的每个新人都需要了解这个内部过程，而且这个过程对老员工没有什么用。

尽管大多数方法学都是开发小组用来开发大型软件的，但对于开发中小型系统的人(解决小问题的单个开发人员)来说，理解优秀方法学的基础知识也是非常必要的。这是因为：

- 方法学有助于对编码设置规则。
- 即使是了解方法学的基本步骤，也能增进对问题的理解，提高解决方案的质量。
- 编写代码只是软件开发的许多活动中的一个，完成其他活动有助于在提交源代码之前，找出概念错误和实践性的错误。
- 在每个阶段，方法学都指定了下一步的工作，我们不会为下一步要干什么而烦恼。
- 方法学有助于编写出扩展性更高(容易修改)、可靠性更高(可用于解决其他问题)、更易于调试(因为有较多的说明)的代码。

大型开发项目还得益于：

- 文档说明：所有的方法学都在开发的每个阶段提供了全面的说明，所以完成的系统不会艰涩难懂。
- 等待时间减少了：由于工作流、活动、任务和相互依赖性更容易理解，所以人力(和其他)资源等待工作做的可能性也减少了。

- 工作能及时交付，且不超过预算。
- 用户、销售员、经理和开发人员之间有更好的交流：好的方法学建立在逻辑和常识的基础之上，所有的参与者较容易抓住其根本；因此，开发更有序，误解和浪费资源的情况也较少。
- 可重复性：因为我们有准确定义的活动，所以类似的项目就应在类似的时间期限内交付，成本也类似。如果多次为不同的客户开发类似的系统(例如电子商务购物前端)，就可以使生产过程变成流水线，只关注最新开发的独特方面。最终就可以使开发的某些部分自动化，甚至把这些自动化部分卖给第三方(例如，把购物前端打包的产品)。
- 更准确的成本：在被问及价格时，回答“你要多少钱的产品”的可能性就会降低。优秀的方法学至少能解决如下问题：
  - 规划：确定需要做什么。
  - 调度：确定完成工作的时间。
  - 分配资源：估计和获得人力、软件、硬件和其他需要的资源。
  - 工作流：较大开发工作中的子过程(例如，设计系统的体系结构、给问题域建模，规划开发过程)。
  - 活动：工作流中的各个任务，例如测试组件、绘制类图，或详细列出使用情况，这些任务本身都比较小，不能定义为工作流。
  - 任务：方法学中由人(开发人员、测试人员或销售人员)完成的部分
  - 制品(artifact)：开发成果：软件、设计文档、培训计划和手册。
  - 教育：如果有必要，确定如何培训人员，以完成他们的任务，确定最终用户(职员、客户、销售人员)如何学习使用新系统的方法。

从本书的目的来看，我们不打算介绍工业方法学的细节——这需要一本书的篇幅来论述。本书使用一种特定用途的方法学，称为 Ripple，它派生于 Rational Unified Process，但比它简单。在介绍 Ripple 之前，先了解一下软件开发的过程、活动和制品。

## 5.2 软件开发中的经典阶段

那么，软件开发会涉及什么？无论采用什么方法学，每个开发过程都有许多共有的阶段，从需求分析开始，一直到最后的维护。在传统的方法中，需要从一个阶段到下一个阶段依次进行；而在现代方法中，可以多次进行每个阶段，且顺序是任意的。

下面描述了软件开发中的共有阶段，其中一些阶段可能有不同的名称，但其基本含义是相同的。目前我们对阶段的意图感兴趣，对如何实现该阶段的细节不感兴趣。注意，一些方法学者把需求和分析合并在一起，而一些方法学者把分析和设计合并在一起。

### 5.2.1 需求

需求捕获就是使用新软件找出我们要达到的目标，它包含两个方面。业务建模就是理解软件的操作上下文——如果不理解该上下文，就不可能生产出改进该上下文的产品。在业务建模的过程中，要问的问题是“客户如何从这家商店购买电视？”

系统需求建模或功能规范表示，确定新软件有什么功能，并记下这些功能。我们应很清楚软件能做什么，不能做什么，这样开发才不会转向不相关的领域。我们还应知道系统何时完成，是否成功。在系统需求建模阶段，要问的问题是“电视被买走后，我们要如何更新商品列表系统？”

## 5.2.2 分析

分析表示理解我们要处理的商务。在设计解决方案之前，需要了解相关的实体、它们的属性和相互关系。还需要验证我们的理解是否正确。这涉及到客户和最终用户，因为他们可能是相关问题的专家。在分析阶段，要问的问题是“这个商店要卖什么商品？它们来自何处？价钱如何？”

## 5.2.3 设计

在设计阶段，要确定如何解决问题。换言之，就是根据对要编写什么软件，如何部署它的经验、估计和直觉，做出决定。系统设计把系统分解为逻辑子系统(过程)和物理子系统(计算机和网络)，决定机器如何通信，为工作选择正确的技术，等。在设计阶段，要做的决定是“我们要使用内联网和 Java 消息传输服务，把销售结果报告给主任办公室”。在子系统设计中，应决定如何把每个逻辑子系统分解为有效、可行的代码。在子系统设计阶段，要做的决定是“商品清单中的每行条目都实现为一个散列表，用商品号做关键字”。

## 5.2.4 规范

规范是一个常常被忽略的，或至少被常常忽视的阶段。不同的开发人员以不同的方式使用“规范”这个术语。例如，需求阶段的结果是系统必须做什么的规范；分析的结果是我们要处理什么事务的规范，等。在本书中，这个术语用于表示“描述编程组件的期望行为”。(因为所描述的规范技术是在对象的类上实现的，使用“类规范”这个术语可以避免一些混乱)。类规范是一种清晰、明确的描述，指出了软件组件的用法，以及如果使用正确，它们会如何操作。在规范阶段做出的陈述是“如果商店助手对象登录了，它就可以向商店对象请求当天的特定商品，然后收到一个按字母排序的商品列表。”

本书特别关注规范，因为规范是按合同设计的、至关重要的底层规则。根据合同，只要一个软件请求另一个软件的服务，调用者和被调用者就应履行义务。时刻想起软件合同对开发的所有阶段都有益。

规范可以以下述方式使用：

- 作为设计测试软件的基础，来检验系统。
- 演示软件是正确的(这对保护生命安全的应用程序来说非常理想)。
- 证明软件组件可以由第三方实现。
- 描述代码如何由其他应用程序安全地重用。

## 5.2.5 实现

这个阶段要完成单调乏味的工作，编写代码，把它们组合在一起，形成子系统，子系统再与其他子系统协同工作，形成整个系统。在实现阶段，要实现的任务是“为 `Inventory` 类编写方法体，使它们遵循其规范”。在这个阶段之前，我们已经做出了大多数困难的编码决策(在设计阶段)，但仍有许多地方需要创新：尽管软件组件的公共接口已经设计、指定和说明过了，但程序员还必须确定其内部工作原理。只要最终结果是有效的、正确的，人们就会满意。

## 5.2.6 测试

在完成软件后，就必须根据系统需求对其进行测试，看看它是否符合最初的目标。在测试阶段，要问的问题是“商店助手能使用接口销售吐司面包，并从商品列表中减去相应的数目

吗？”除了这类一致性测试之外，最好看看软件是否能通过其外部接口来分解——这有助于在系统部署后，防止事故或对系统的恶意误用。

程序员还可以在开发过程中进行小的测试，改进他们交付的代码的质量。但一般说来，编写软件的开发人员不应设计、实现或进行大型测试。为了理解其原因，考虑购买一座新房子，并花大量的资金对它进行彻底的装修。我们肯定不会用大锤子重击房子的结构和固定设备，看看它们是否坚固，也不会让路过的陌生人假装夜贼，看看能不能闯入这座房子。这些都是在软件测试过程中需要做的工作。(这有助于测试小组的成员检查软件是否有污点)。

### 5.2.7 部署

在部署阶段，要将硬件和软件交付给最终用户，并提供手册和培训材料。这可能是一个复杂的过程，涉及从旧工作方式到新工作方式的、有计划的渐进转变。在部署阶段，要完成的任务是“在每台服务器上运行程序 setup.exe，并按照提示进行下去。”

### 5.2.8 维护

部署好系统后，它只是刚刚投入使用。它还有很长的路要走，在此过程中，用户每天都要使用它，真正的测试才刚刚开始。在维护阶段，可能发现的问题有“当登录窗口打开时，它仍包含上次输入的密码。”

软件开发人员一般对维护很感兴趣，因为在这个过程中会找出软件中的错误(bug)。我们必须尽快找出错误，并更正它们，发布软件的修订版本，使最终用户满意。除了错误之外，用户还可能发现软件的不足(系统应做但没有做的事情)，提出额外的要求(以改进系统)。从商务的角度来看，我们应积极更正错误，改进软件，以保持竞争优势。

### 5.2.9 关键问题

这些关键问题有助于记住每个软件开发阶段的目的：

- 需求阶段：

什么是我们的上下文？

要达到什么目的？

- 分析阶段：

要处理什么实体？

如何确保有正确的实体？

- 系统设计阶段：

如何解决问题？

在完成的系统中需要什么硬件和软件？

- 子系统设计阶段：

如何实现解决方案？

源代码和支持文件有哪些？

- 规范阶段：

哪些规则控制着系统组件之间的接口？

可以去除模糊，确保正确吗？

- 实现阶段：

如何编写组件，符合规范的要求？如何编写漂亮的代码？

● **测试阶段：**系统是否满足用户需求？系统是否能够正常运行？完成的系统满足要求吗？

● **部署阶段：**系统是否部署在正确的服务器上？部署后系统是否正常运行？可以攻破系统吗？

● **维护阶段：**系统管理员必须做什么？如何培训最终用户？

● **维护阶段：**可以找出和更正错误吗？

● **改进阶段：**可以改进系统吗？

### 5.3 软件工程和瀑布方法学

在 20 世纪 70 年代，软件专家为如何编写软件提出了许多想法。如何用可伸缩的、可移植的方法学代替事实上的专用机制？这些专家提出的想法称为软件工程(software engineering)。其理念是软件生产可以像构建真实的结构那样，例如建桥。根据物理学，工程是系统化的：如果遵循规则，就将交付有效的产品，它有安全的余量，以防止异常情况发生。

这种类比有一些明显的缺陷。大多数编程都是使用命令式(imperative)的编程语言完成的，编程语言要求程序员通过语句、分支、函数，明确告诉计算机要做什么。这类似于桥梁工程师必须告诉每块钢材如何操作，而不是依赖于物理定律。这还不是全部：程序员必须装配各个数据块。如果桥梁建筑师必须找到一块块石头，再告诉每块石头与其他石头如何接合，才能铺成桥梁的路面，这是不可想象的。

除了要求程序员必须明确行为和结构之外，命令式语言也不能很好地处理不准确的数据。一般说来，如果系统中的一块数据略微不准确，系统就不会按期望的那样操作。相反，桥梁建筑师可以使用规格略有不同的铆钉连接支架；工程师可以依赖公差和误差幅度，但程序员不能。

这些缺陷并不能阻止软件工程的推广，方法学在软件生产可以系统化和可预计的假定下日益成熟。这就产生了所谓的瀑布方法学(waterfall methodology，如图 5-1 所示)。

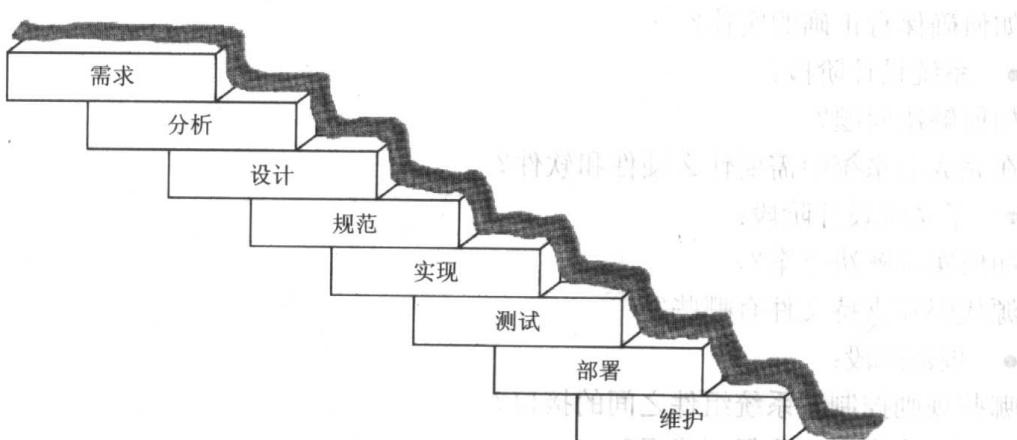


图 5-1 瀑布开发

在经典阶段中，开发非常顺畅(需求、分析、系统设计等)，每个阶段都满意地完成，之后进入下一个阶段。这很容易规划(因为每次的计划都是类似的)和调度(使用问题的复杂性和开发人员的数量来确定开发需要的时间，再把结果除以阶段的总数)。瀑布方法学可以在每个阶段使用具有不同技能的开发人员(这是业务分析员、系统分析员、设计员、程序员、测试人员和系统管理员的经典任务)。每个专家小组在自己的阶段艰难行进，直到确保解决了属于他们的那部分问题为止，然后使用专业术语和表示法，为他们的工作编写文档，把指挥棒传递给下一个专家小组。

瀑布方法学是一个很好的想法，但不切实际。即使我们能确定开发需要的时间，在没有考虑问题的细节之前，是不可能预知在开发过程中会遇到什么困难的(糟糕的设计决策、有害的错误、技术不适当或地震)。所以，任何阶段都可能比预期的时间长。另外，工作也可能会扩展，以充分利用可用的时间，这样，某个问题之前的各个阶段很可能用尽了所有的可用时间。最终结果是整个项目都得延迟交付。实际上，这就是大多数项目的具体情况。

瀑布方法学也可能因其他原因而失败，例如分析失效——分析员不愿签署其文档，因为他们不能确保已经很好地理解了系统实体，能让设计人员完成其工作，并为它们编写文档。而且，这类问题并不限于分析员：设计人员也可能担心他们的设计不适当；规范人员可能担心他们的规范过于模糊，不能用于编程；等，这些会导致更多的延迟。实际上，根本不可能非常完美地完成每个阶段。在整个开发过程中，小组成员会发现他们已完成的工作中有问题。只要出现这种情况，我们就面临着一个两难的选择：要么返回到前面的说明文档中，更正错误，但这意味着逆着瀑布向上走(不推荐这么做)；要么记下问题，在项目的最后修订说明文档(这是很少发生的，所以最后的说明文档不匹配最终系统)。

最终用户该怎么办？他们得到了想要或需要的系统吗？系统的潜在用户是同事或第三方，他们可能会为系统付钱。假定在需求阶段已咨询过这些客户：我们问他们当前如何工作，并集体讨论了可以交付的系统功能，对要交付的系统功能和不交付的系统功能达成了一致。但如果发现有些功能不能交付，因为它们很难实现或时间不够，该怎么办？客户直到测试甚至部署时才会发现，那时再改动就太晚了。如果项目用了两年才完成，该怎么办？最终用户的要求一般是在两年内进行一次大的改动。我们会交付一个不再相关的系统吗？我们需要一种用户参与整个开发过程的方式，这样就不会交付某种令人惊讶的系统了。还需要缩短承诺提供某功能和真正提供该功能之间的时间。但瀑布模型过于死板，不能充分利用用户的反馈，采取正确的措施。

问题还不只此。例如，瀑布方法学总是集中于解决一个问题，所以很难编写出可重用的代码。吃掉一头大象是一个艰巨的任务(如图 5-2 所示)；如果真要吃掉一头大象，您会张开大嘴吞下整头大象，还是先吃掉可管理的一小部分，休息一下，再吃掉下一部分，直到吃掉整头大象？瀑布开发模型使用第一种方法，试图一次生产出完整的解决方案。而使用第二种方法比较可行，分部分地完成整个解决方案。

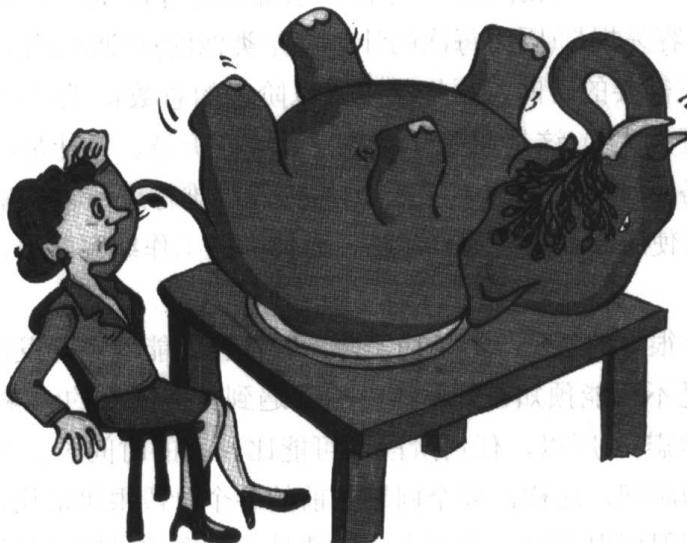


图 5-2 吃掉一头大象

瀑布方法学有这么多问题，其中一些在一开始就很明显，所以很难相信软件开发可以用这种方式完成。但以前是用这种模型开发软件的，现在有一些地方也采用这种模型。一些公司很愿意在瀑布方法学的基础上进行大型项目的开发，并把它集成到软件部门，作为最佳实践，职务提升过程也以此为基础(首先是程序员，晋升为设计员，再提升为系统分析员，等)。但是，大多数面向对象的爱好者和有远见的公司更喜欢较灵活的方式。

指出瀑布方法学的问题后，还应指出它在下述情形下仍是有效的：

- 重复某种区别很小的开发(例如，某家公司的电子商务销售前台与以前版本的区别仅在于商品描述、价格、公司名称和徽标)。
- 作为一个架构，来学习软件开发中使用的技术：尽管瀑布方法学对于实际的开发来说过于简单，但仍包含逻辑顺序的经典阶段，所以适合于学习。
- 是螺旋(spiral)方法学中的一遍过程。
- 作为支持迭代方法学的架构。
- 用于开发人员较少的小项目的快速开发，例如原型(prototyping)、生产原型(production prototyping)、概念证明(proof-of-concept)或快速应用程序开发(Rapid Application Development, RAD)。

人们利用对象的简单和强大，代码在新应用程序中的重用，以及应用程序构建器的出现，开发了编程的四种新样式：

- 软件原型：就像过程原型一样，是用来试验已完成产品中的一些功能。原型不需要是第一流的，也不一定非常强大，因为它只是一个试验品。一旦它完成了使命，就把它放在一边，再开始另一个原型。
- 生产原型：类似于原型，但在完成项目的过程中，保留部分或全部代码。
- 概念证明：是一个项目或软件，用于证明某种技术或某组技术的可靠性。例如，需要向客户证明我们有资格承接某个项目，或需要向管理层证明应在软件生产过程中采用新方法。

- 快速应用程序开发(RAD)是面向对象的爱好者制造的一个词，表示比传统技术更快地建立系统。面向对象的系统在 20 世纪 80 年代成为现实后，对象爱好者就在快速装配小型系统方面给传统开发人员(和经理)留下了深刻的印象。

应用程序构建器是一个工具，它允许程序员以计算机厂家装配硬件的方式装配软件。随着时间的推移，应用程序构建器使用越来越大的组件，系统的构建也比以前快得多。在大多数情况下，面向应用程序构建器的对象需要有一种特殊的接口，才能正常工作，这意味着程序员需要遵循预定义的样式规则，而不是设计自己看着合适就行的对象。例如，Java 应用程序构建器可以处理遵循 JavaBeans 规则的对象[Campione 等，98]。

## 5.4 新方法学

在用新方法学替换瀑布方法学之前，需要接受一个观点：不可能一次开发出一个软件。无论我们怎么努力，第一次开发出的软件都是不完整或不完美的。所以，需要执行几次软件开发的经典阶段，给系统添加功能，完善系统。

### 5.4.1 螺旋式方法学

一种方式是螺旋式方法学(如图 5-3 所示)。与以前一样，首先是需求分析，在这个阶段，它可能比较完整或相当粗略；接着，进行一些探索性分析，增进对我们要处理的事务的理解；然后勾画出一个大致满足需求的系统设计，并设计部分系统；之后，尽管所有的前期阶段都未完成，仍可以编写一些代码。一旦完成了最初的编码，就可以测试已有的部分系统，例如进行一些非正式的测试，或者向用户(最终用户、经理和为系统付费的客户)展示我们已拥有的部分系统。

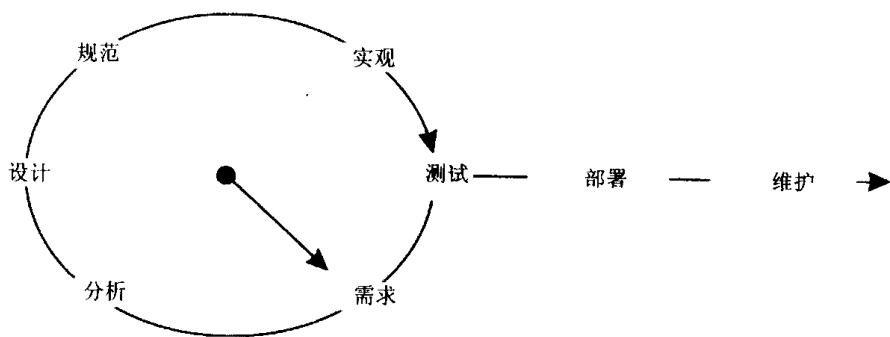


图 5-3 螺旋式开发方法

在完成了一次循环后，就增进了对问题域和解决方案的理解，还使用户参与进来，这样用户就可以更正我们对最终系统中包含的事务或功能的误解。有了新的知识库，就可以再次执行一遍开发过程：现在我们更新了需求；更深地理解(和更正)了分析；加强系统设计；给子系统添加细节；再编写更多的代码，更多地满足需求。

在经过三四遍的开发过程，完成系统后，就可以全面测试和部署系统。与瀑布方法学比较，现在我们更像一个创作雕像的雕刻家：先搭建起六角形网眼轻质铁丝网的基本框架，再一层层地添加粘土，直到得到想要的结果为止。在工作过程中，项目需要多长时间越来越清晰，用户会看到项目正在进行中，我们也越来越自信能开发出一个优秀的系统。当每个人都感到满意时，

系统就完成了。

似乎螺旋式方法学的问题比瀑布方法学少：它使用户参与整个生命周期，每个人都可以看到我们正在工作；它的刚性较少(可以调整改动的次数和每次改动所花的时间)。另外，它比较适合软件开发的创新性，这与构建大桥的工程性相反。

但螺旋式方法学也不是完美的。我们只是把瀑布开发过程进行了三四次，也就是说，尽管问题会越来越小，但它们并没有消失。螺旋式方法仍有一些不灵活的地方，因为在经典阶段要按照有序的方式进行；如果发现了错误，就必须在下一遍开发过程中才能更正它们。因此，螺旋式方法学本身不是非常有用——需要把它与其他方法学结合起来。

### 5.4.2 迭代式方法学

如何构建螺旋式方法学？继续使用前面的雕刻比喻，螺旋式方法学要求完成每一层的粘土添加之后，再进入下一层。例如，假定要雕刻一个人。在头部添加第一层粘土后，如果在处理身体的其他部分之前，要先雕刻出鼻子，该怎么办？如果雕刻尚未完成，却觉得前额太窄了，需要立即改动，该怎么办？

这里需要的方法学应允许重复各个阶段，根据需要前后移动或来回移动。这就是迭代式方法学，如图 5-4 所示(这里，迭代在螺旋式方法学中出现，但它们也可以应用于瀑布方法学)。现在，有一种比较自然的方式，把软件从其早期阶段传递给下述完备、使所有用户满意的整体系统。但如何避免混乱？这里至少要有三个原则：

- 经典阶段提醒我们应在每个阶段做什么，向什么方向移动。
- 在经典阶段的工作中生产出的制品(图、描述、代码等)不应抛弃，但应在进入部署阶段的过程中逐步改进。
- 支持所选方法学的软件生产工具和表示法有助于确保制品的一致性，并在一个地方保存所有的制品。

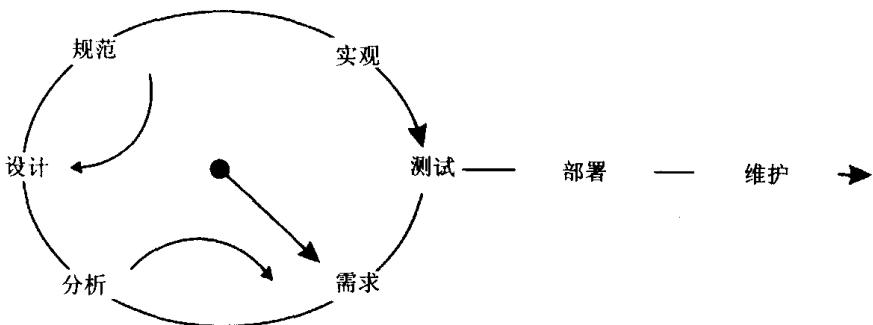


图 5-4 迭代式开发方法

还有一个问题。采用迭代式和螺旋式方法学，仍要吃掉一头大象：仍要开发一个完整的系统，以进行部署，所以需要添加最后一个元素：递增式方法学。

### 5.4.3 递增式方法学

下面最后一次看看前面的雕刻比喻。如果要雕刻出一个家庭——父母、孩子、猫和狗，最好一次完成一个部分，这样用户就可以在交付的部分中看到我们的进度；我们也可以在完成一个部分后，集中精力雕刻下一个部分；还可以要求逐次付款。在软件开发中，这就是递增式方

法学，如图 5-5 所示。

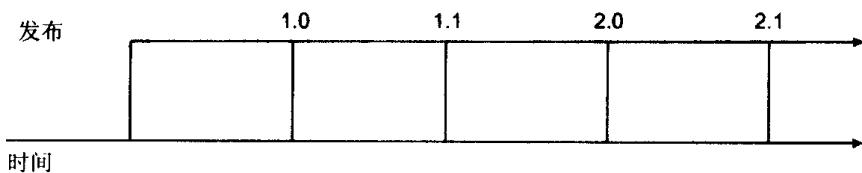


图 5-5 递增式开发

采用递增式方法学，在交付的系统 1.0 版本中，将包含最基本的、最重要的功能。在以后的某个时间，交付 1.1 版本，其中包含附加的功能(替代 1.0 版本)。之后，在对整体进行大的修改后，交付 2.0 版本。这将延续至系统的生命周期。我们不仅从一开始就认识到需要一部分一部分地吃掉整头大象，而且能满足不断变化的需求，适应变化的市场。我们从购买软件的经验中获知，无论是否进行了规划，递增交付都是比较实际的方式。换言之，如果尝试一口吞下整头大象，就会失败。对递增交付进行规划，就可以把失败转变为成功。

但是，无论如何都必须避免为每个新递增的部分重新编写所有的代码——这是一个没有尽头的工作。所以，透彻的分析、合理的设计、可重用的代码和可扩展性是非常重要的。

#### 5.4.4 合并方法学

在大多数情况下，瀑布方法学是不适当的，但它的各个阶段有正确的逻辑顺序。其他方法学(螺旋式、迭代式和递增式)都有比较理想的特性，但都不够完美。所以必须以某种方式合并这四种方法学。但如何合并？

在最高的层次上，从递增式方法学中可知，必须规划一系列递增部分。在每个递增部分，螺旋式方法学建议，开发每个递增部分至少应进行两遍。在每一遍中，瀑布方法学指定了各个阶段和它们的顺序。在每个子瀑布中，迭代式方法学允许重复各个阶段，或合并各个阶段，直到满足需求为止(例如需求和分析的几次循环)。迭代式方法学还允许一发现问题就更正它(例如，在子系统设计过程中发现，系统设计无法实现某个功能，就更正系统设计，再继续下去)。方法学的合并如图 5-6 所示。

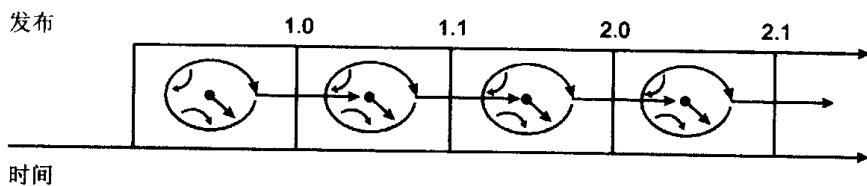


图 5-6 合并螺旋式、迭代式和递增式开发方法

前面的讨论没有指出如何规划和调度某个项目，这取决于项目的规模、开发人员的人数、开发人员的经验、规划和调度这类开发的经理的经验等。就本书而言，详细的规划和调度是管理问题，而不是软件问题，所以本书不作介绍。可以说，在面向对象中，没有完全绝对的事情：经理、顾问和有经验的开发人员决定适合于每个新开发系统的递增式、螺旋式、迭代式方法和制品是相当合理的。规划必须根据逐渐递增的知识和变化的需求来调整。

大多数理论家都认同瀑布方法学的特性，但所使用的术语可能有所不同。本书介绍了螺旋式、迭代式和递增式方法学的定义，但其他人以不同的方式使用这些术语，甚至用作同义词。尤其是，“螺旋式”有时表示为“迭代式”。但是，无论使用什么名称，这里讨论的理念都是有效的。

## 5.5 面向对象的方法学

所有面向对象的专家都相信，合适的方法学是软件开发的基础，尤其是团队合作时，就更是如此。因此，在过去的十年中，人们发明了许多方法学。广义地看，所有面向对象的方法学都是类似的，它们有类似的阶段和类似的制品，但有许多小的区别。

面向对象的方法学不太好理解，例如，开发人员在是否使用某种类型的图时有一些选择。因此，开发小组在进行详细规划或调度之前，必须选择一种方法，认可该方法产生的制品。

一般，每个方法都会包含：

- 每个阶段的哲理
- 每个阶段的工作流和各个活动
- 应生产的制品(图、文本说明和代码)
- 制品之间的依赖关系
- 不同制品的表示法
- 需要给静态结构和动态行为建模

静态建模包括决定系统的逻辑或物理部分有哪些，它们如何连接在一起。动态建模要决定静态部分如何相互协作。粗略来看，静态建模描述了如何构建和初始化系统，而动态建模描述了系统在运行时如何动作。一般情况下，在开发的每个阶段，至少要建立一个静态模型和一个动态模型。

一些方法学，尤其是比较全面的方法学，有其他开发途径，生产出不同类型和不同规模的开发产品。本书使用的方法学是 Ripple，它吸收了所有软件开发(无论是大型系统还是小型系统)涉及的阶段，适合真正的系统开发。

### 5.5.1 UML、RUP 和 XP

在 20 世纪 90 年代中叶，最著名的方法学是 Ivar Jacobson [Jacobson 等, 92]、James Rumbaugh [Rumbaugh 等, 91] 和 Grady Booch [Booch 等, 93] 发明的方法学。它们都有自己的咨询公司，使用自己的方法学和表示法。在 1996 年，Jacobson 和 Rumbaugh 合并为 Rational 公司(Booch 建立)，开发出一系列表示法，称为统一建模语言(UML) [OMG 03a]。著名的“三剑客”把 UML 赠给对象管理组(OMG)来保管和改进。OMG([www.omg.org](http://www.omg.org))是一个非盈利的行业协会，在 1989 年建立，负责推广企业级对象技术的开放标准。它另一个著名的产品是 CORBA[OMG 04]。

一些开发人员把 UML 简单地看做一种表示法，用于集体讨论和高级文档。其他开发人员认为 UML 是一种图示的编程语言，从中生成代码，或者从已有的代码中合成图形。最后，主要的图将精确地匹配已完成的代码——如果 UML 用作图示的编程语言，最终结果就与已经完成的部分相同。

一旦手中有了 UML，“三剑客”就着手设计一个方法学，利用各自工作的最佳方面。在几年内，它们就提出了自己的螺旋式、迭代式和递增式方法，称为 Rational Unified Process(RUP) [Jacobson 等, 99]。可以想像，RUP 不是惟一可用的方法学，甚至不是把 UML 作为表示法的惟一方法学。

另一个常见的方法学是极限编程(extreme programming, XP)[Beck, 99]。XP 称为“快变”方法学，因为它能响应变化。XP 可通过两个基本理念来区别：配对编程和测试驱动的开发。所谓配对编程，是指所有的开发过程都由两个坐在屏幕前的人来进行，而不是一个人。其含义是，配对编程不是提高软件开发的速度，而是改善软件的质量(还有助于开发人员通过共享理念来更快成熟)。按照热衷于测试驱动的开发的人员的说法，连续不断的测试非常重要，它不仅应由开发人员来进行，还应在编写代码之前进行。

### 5.5.2 开发工具的需求

为了提高效率，任何螺旋式、迭代式和递增式方法学需要端对端的开发工具。需要反复改进项目制品，当然就需要使用软件工具。这种工具应允许项目小组的成员制作制品，并存储它们。更准确地说，它应支持：

- 可跟踪性：记录制品与其派生物之间的连接，例如，记录哪个子系统生成了一组已实现的类。大多数可跟踪的信息是由开发人员输入的，而不是由工具推断出来的。
- 修改历史：记录对制品进行的修改，谁进行了这些修改，以及修改的时间。只要可能(对于文档说明来说)，工具就应能提供制品各个版本之间的区别汇总。
- 多用户访问控制：确保对制品的同时访问不会出问题。这里有三个相关的机制：授权(控制谁可以阅读制品，谁可以编辑它们)；多用户阅读—单用户编辑(在某一时刻只允许一个开发人员编辑一个或一组制品，但允许所有授权的用户查看未编辑的版本)；版本控制(允许任意数量的开发人员编辑制品，每个开发人员都生成制品的一个版本——在任意时刻，只有一个版本是官方版本)。
- 减少冗余：确保不在多个地方进行更新。一般情况下，信息可以一次显示在几个文本文件中。如果有一个工具，就可以把文本文件作为单一模式下的替代映射。
- 一致性检查：确保制品与相关的制品保持一致。并不总是需要强制一致性。例如，在理想情况下，工具应能检查为方法编写的代码是否遵循其规范，但现代计算机做不到(这很困难，甚至是不可能的)。甚至在工具可以强制一致性(或可跟踪性)的情况下，开发人员也必须能禁用这个检查。例如，一个开发人员要在整个设计阶段中使用分析类，而另一个开发人员要从一开始就设计一组新类：如果工具需要在设计模型中有每个分析类，它就对第一个开发人员有益，但会妨碍第二个开发人员。
- 联网操作：允许在项目网络上的任意机器上访问所有的制品。目前，联网操作的最佳基础协议是 TCP/IP，它是所有在 Internet 和内联网上工作的协议的基础。
- 在开发过程中测试已制作出来的制品。最明显的例子是为效率和正确性测试实现代码，该规则还可以用于其他制品(例如记录设计评估的结果)。

Rational 公司根据 RUP 和 UML 开发了一个工具 Rose，它是最著名的面向对象开发工具。2003 年，IBM 收购了 Rational 公司，重新编写了该工具，使之成为一组模块化产品(现在称为

Rational 应用开发器, Rational Application Developer)。当然, 还有许多其他可用的开发工具, 这里只介绍 Rational 产品, 因为我们是从历史的角度来看的。

## 5.6 Ripple 概述

本书将介绍软件开发的所有经典阶段的面向对象版本, 讨论它们如何适应面向对象的方法学。面向对象是很容易得到的, 所以开发人员可以涉及所有的阶段, 客户也可以涉及每个阶段, 这有助于开发人员完成其工作, 经理不会插手开发人员的工作, 所以改进了交流。

前面介绍了经典阶段如何适应理想的面向对象方法学——利用螺旋式、迭代式和递增式方法的最佳部分。在后面的章节中, 将介绍发布代码前进行的每个主要阶段: 需求、分析、系统设计、子系统设计、规范和测试。实现阶段不会展开论述, 因为这需要某种编程语言的详细知识。本书会讨论设计模式, 因为它们可以构建出实现理念的框架。

阅读完本书, 您会发现进行了第一遍递增(在一次迭代后, 会开发出版本 1.0)。这似乎像瀑布方法学穿上了一件雅致的外衣那样, 比较可疑, 但它是一本书的特性: 内容的安排是从一个主题到另一个主题, 不会重复。在自己试用这些技术时, 必须准备螺旋式、迭代式、递增式地交付。

这里开始使用的案例 iCoot 肯定不是用瀑布方式开发的。<sup>1</sup>其制品包含在附录 B 中, 取自于两个递增部分, 每个递增部分都由许多迭代组成。

UML 是事实上的标准表示法, 本书一直使用该表示法。书中的图类似于在真实情况中遇到的图。对于 Ripple, 只要可能, 就使用 UML 表示法。

UML 有 13 种类型的图。UML 规范没有要求这些图应在什么方法学中使用, 所以可以在合适的任何阶段使用。

- 用例图对系统的使用方式分类。
- 类图显示类和它们的相互关系(也可以显示对象)。
- 对象图只显示对象和它们的相互关系。
- 活动图显示人或对象的活动, 其方式类似于流程图。
- 状态机图显示生命周期比较有趣或复杂的对象的各种状态。
- 通信图显示在某种情形下对象之间发送的消息。
- 顺序图显示与通信图类似的信息, 但强调的是顺序, 而不是连接。
- 包图显示相关的类如何组合, 对开发人员有用。
- 部署图显示安装已完成系统的机器、过程和部署制品。
- 组件图显示可重用的组件(对象或子系统)及其接口。
- 交互总图使用顺序图显示活动的各个步骤。
- 时间图显示消息和对象状态的准确时间限制。
- 复合结构图显示对象在聚合或复合中的相互关系, 显示接口和协作的对象。

表 5-1 按照各个阶段总结了 Ripple 的制品。可以看出, 一些制品在 UML 中, 一些制品不在 UML 中。这是因为 UML 并不包含所有的信息; 它在很大程度上只允许绘制代码图。对于 UML 没有包括的 Ripple 制品, 可以使用其他表示法。这种表示法可能并不标准, 但其内容应基于广泛接受的理论和实践。

表 5-1 按阶段的 Ripple 制品

阶 段		制 品	UML
起源		任务陈述或非正式的需求 任务 责任 项目计划 工作本 小词典(全面更新) 测试计划	否 否 否 否 否 否 否 否
需求	业务	参与者列表(带有描述) 用例列表(带有描述) 使用情况细节 活动图(可选) 通信图(可选)	否 否 否 是 是
	系统	参与者列表(带有描述) 用例列表(带有描述) 用例细节 用例图 用例调查 用户接口的框架	否 否 否 是 否 否
分析		类图 通信图	是 是
设计	系统	部署图 层图	是 否
	子系统	类图 顺序图 数据库模式	是 是 否
类规范		注释	否
实现		源代码	否
测试		测试报表	否
部署		压缩打包的解决方案 手册 培训材料	否 否 否
维护		错误报告 递增计划	否 否

Ripple 将在各个章节中逐步介绍，但如果读者想先大致了解一下，可参阅附录 A。本书强

调的是与软件相关的制品，其焦点是需求、分析、设计和规范。也讨论小词典、测试计划和任务陈述。其他问题，例如管理、实现、部署和维护等不涉及。

下面是一个活动图和状态机图的例子，但在这里使用该例子并不合适，一般这两个图是可选的。另外，也没有使用所有类型的 UML 图。组件图、交互总图、时间图和复合结构图对本书来说并不必要。这些类型的图可以使用其他图来表示，但时间图除外，这种图比较适用于实时软件和硬件的设计。

除了图之外，UML 还有一种类规范语言，称为对象约束语言(Object Constraint Language, OCL)。本书不介绍 OCL，因为它需要一本书的篇幅来论述，但第 12 章将介绍一个小例子，大致了解一下。大多数的规范讨论都是非正式的，与代码和设计制品中的注释相关。

在下面的几节中，将介绍一些用于 Ripple 的 UML 图的例子。记住，UML 与最全面的标准一样，是相当大的。所以，为了便于实践，这里只介绍表示法的基础知识，不讨论细节。

只要在本书中看到一个图，就应注意 UML 允许压缩与讨论无关的信息。例如，在讨论类时只看到带标签的方框，而这并不意味着类没有属性或操作。

### 5.6.1 用例图

用例是对客户、用户或系统使用另一个系统或业务的方式的静态描述。用例图(use case diagram)显示了系统用例的相互关系和用户了解它们的方式。用例图中的每个椭圆都表示一个用例，每个小人都表示一个用户。用例图(静态制品)在第 6 章介绍。

图 5-7 描述了可通过 Internet 访问的汽车租用商店。在这个图中，很容易获得许多信息。例如，助手可以预约汽车；客户可以查找汽车型号；会员可以登录；用户必须登录，才能预约汽车，等。

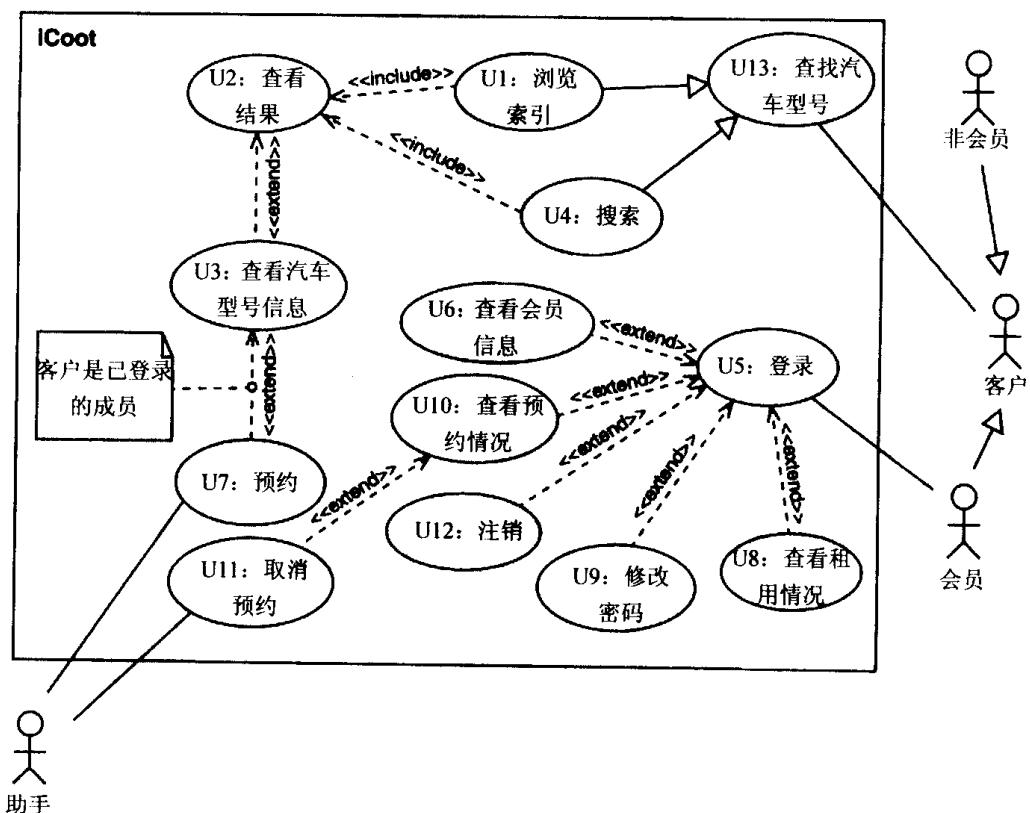


图 5-7 用例图

每个用例都有多个标题，例如“U7：预约”或“U13：查找汽车型号”。其中必须包含使用系统或业务的实际步骤。UML 为用例图指定了表示法，但没有为用例本身的步骤指定表示法。RUP 指定了用例的步骤和几个其他的信息，例如用例的细节。

“U3：查看汽车型号信息”的细节显示在图 5-8 中。很容易看出，查看汽车型号信息包括：客户选择汽车型号，请求它的信息，然后接收该汽车型号的特定信息。用例细节(动态制品)详见第 6 章。

U3：查看汽车型号的信息。(它由 U7 扩展而来，又扩展了 U2。)

前提条件：无。

1. 客户选择一种匹配的汽车型号。

2. 客户请求选中的汽车型号的信息。

3. iCoot 显示选中的汽车型号的信息(构造、引擎规格、价格、描述、广告和海报)。

4. 如果客户是一位已登录的会员，就用 U7 来扩展。

后置条件：iCoot 显示了选中的汽车型号的信息。

非功能需求：

r1. 广告应使用流协议显示，不需要下载。

图 5-8 系统用例的细节

## 5.6.2 类图(分析级别)

类图显示了在业务(在分析阶段)或系统本身(在子系统设计)中存在哪些类。图 5-9 是一个分析级别的类图示例，其中的每个类都表示为一个带标签的方框。类图(静态制品)在第 7 章介绍。

除了类之外，类图还显示了这些类的对象如何连接在一起。例如，图 5-9 显示 CarModel 中有一个 CarModelDetail，表示它的细节。

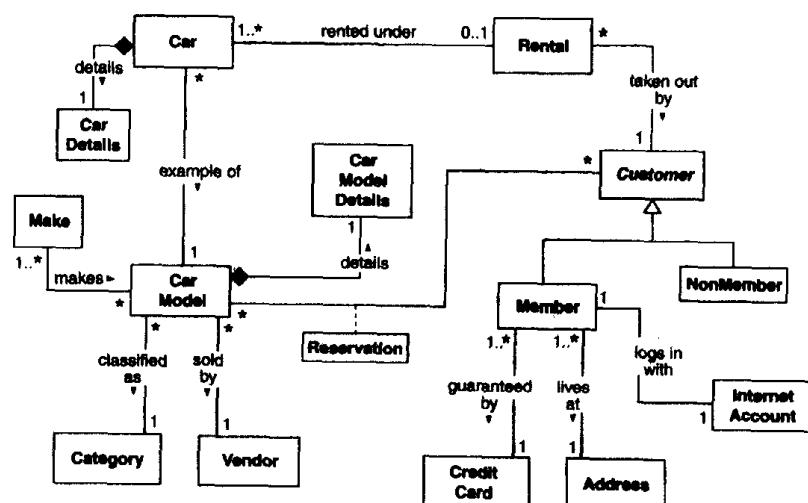


图 5-9 分析级别的类图

## 5.6.3 通信图

通信图显示了对象之间的协作。图 5-10 中的通信图描述了在 Internet 上预约汽车型号的过程：Member 告诉 MemberUI 预定一个 CarModel，MemberUI 告诉 ReservationHome 为给定的 CarModel 和当前的 Member 创建一个 Reservation；MemberUI 再为新的 Reservation 请求一个编号，并把它返回给 Member。通信图(动态制品)在第 7 章介绍。

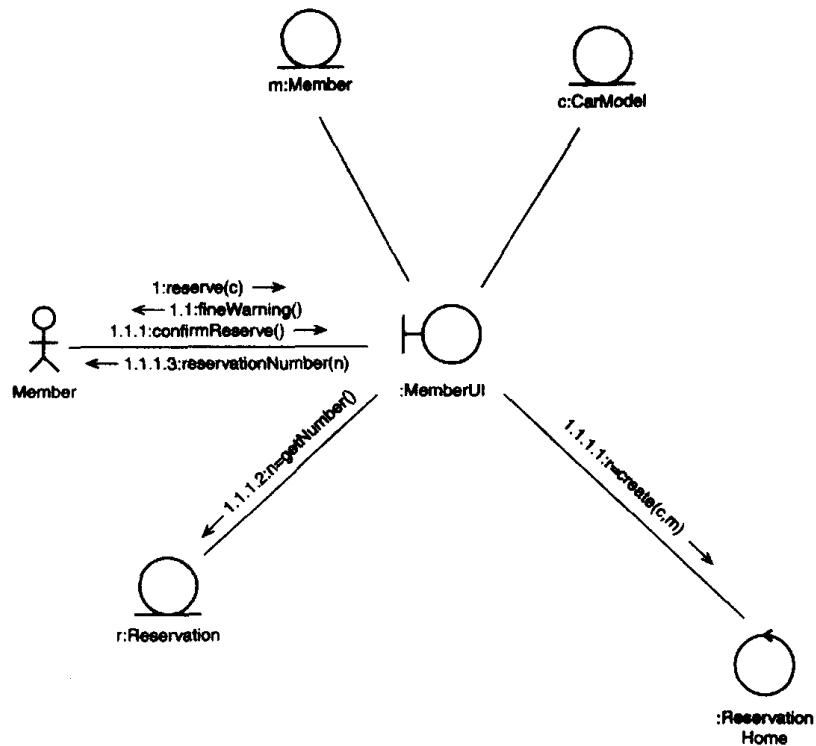


图 5-10 通信图

#### 5.6.4 部署图

部署图(见图 5-11)显示了已完成的系统如何部署到一台或多台机器上。部署图可以包含所有的特性，例如机器、过程、文件和依赖性。

图 5-11 显示了任意多个 HTMLClient 节点(每个节点包含一个 WebBrowser)和 GUIClient 节点与两个服务器(每个服务器包含一个 WebServer 和一个 CootBusinessServer)的通信；每个 WebServer 都与 CootBusinessServer 通信，每个 CootBusinessServer 又都与运行在两个 DBServer 节点之一上的 DBMS 通信。部署图(静态制品)在第 8 章介绍。

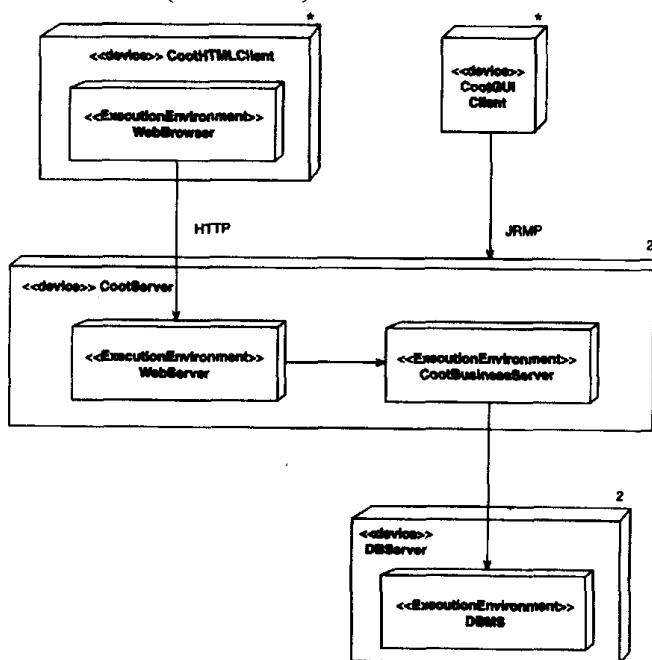


图 5-11 部署图

## 5.6.5 类图(设计级别)

如图 5-12 所示的类图使用的表示法与图 5-9 相同。惟一的区别是设计级别的类图使用更多的可用表示法，因为它更详细，这个类图扩展了分析类图的部分，显示了方法、构造函数和可导航性。设计级别的类图(静态制品)在第 10 章介绍。

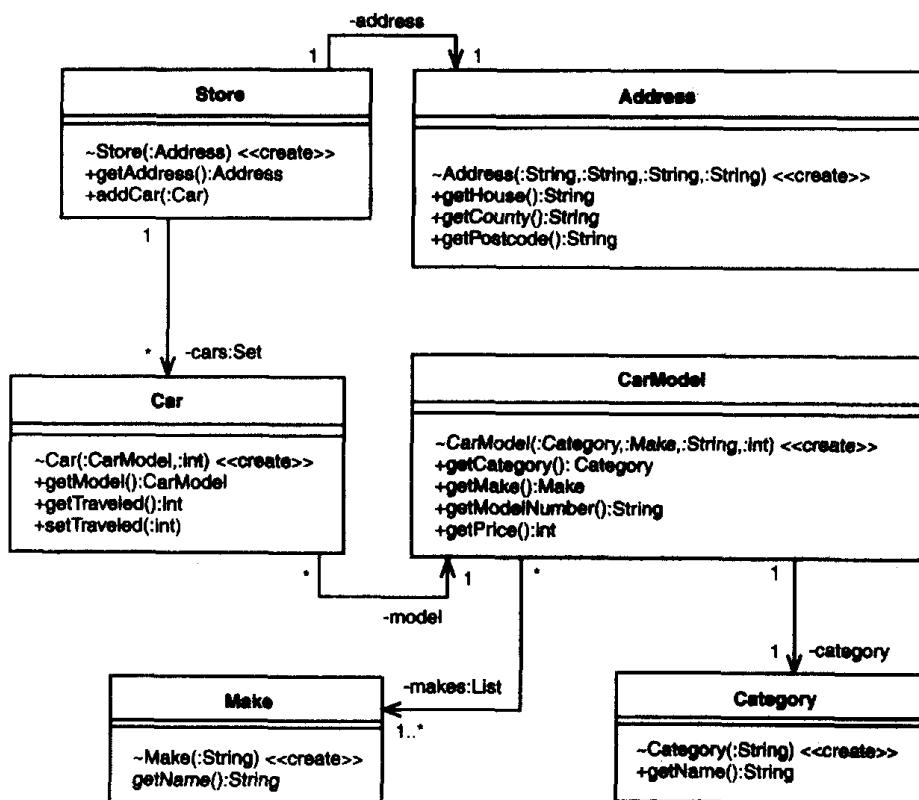


图 5-12 设计级别的类图

## 5.6.6 顺序图

顺序图显示了对象之间的交互。通信图也显示了对象之间的交互，但它强调的是链接，而不是顺序。本书中的顺序图在子系统设计阶段使用，它们也可以应用于分析阶段、系统设计阶段、甚至需求分析阶段的动态建模。

图 5-13 显示了 Member 如何从系统中注销。消息显示为竖条之间的箭头，这些竖条表示对象(每个对象的名称位于竖条的顶部)。

在顺序图中，时间轴的指向是从上向下。所以图 5-13 指定，Member 请求 AuthenticationServlet 注销，AuthenticationServlet 把请求传送给 AuthenticationServer，从浏览器会话中读取 id；AuthenticationServer 找出对应的 Member 对象，告诉它将其会话 id 设置为 0；Member 把这个请求传送给它的 InternetAccount；最后，给 Member 显示主页。顺序图(动态制品)在第 10 章介绍。

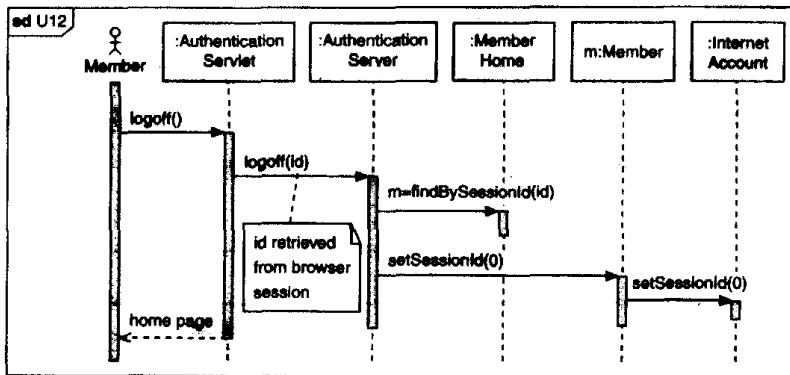


图 5-13 设计阶段的顺序图

## 5.7 小结

本章的主要内容如下：

- 软件开发的经典阶段：需求、分析、系统设计、子系统设计、规范、实现、测试、部署和维护。如何在螺旋式、迭代式和递增式方法学中使用它们。
- 静态建模和动态建模，静态建模描述了如何构建系统；动态建模描述了系统在运行时如何操作。
- 本书后面使用的 UML 表示法和 Ripple 方法学。

## 5.8 课外阅读

Steve McConnell 从规范一直到测试，全面论述了如何管理成功的软件项目[McConnell 98]。这是很容易理解的，因为它采用了高级方法和对话方式。

Rational 统一过程的入门图书[Jacobson 等, 99]简要论述了 RUP 的理论。RUP 的当前情况可查阅 Rational 网站 [www.rational.com](http://www.rational.com)。

XP 的更多信息可参阅它的一个发明者[Beck 99]，或者查阅下面的网站：[www.extremeprogramming.org](http://www.extremeprogramming.org) 和 [www.xprogramming.org](http://www.xprogramming.org)。快变开发的更多信息可参阅[Cockburn 01]和 [www.agileManifesto.org](http://www.agileManifesto.org) 上的 Manifesto of Agile Software Development。

UML 最著名的入门图书是[Fowler 03]。Martin Fowler 没有全面论述 UML，但这本书很适合做本书和 UML 规范[OMG 03a]之间的过渡图书。对于高级 UML 用户，规范仍是一个基本工具，可以回答关于表示法的模糊部分和语言语义等问题，所以可以确保图是正确的。规范很难读懂，通过 UML 规范会发现，UML 是用 UML 定义的，但这是一个好方法：描述非常准确，也可以看出 UML 如何用于证明复杂的大型面向对象的模型，并可以根据模型弄清楚什么是合法的，什么是不合法的。

## 5.9 复习题

1. 下面哪个 UML 制品用于显示系统中过程、资源和对象的发布？(单选题)

- (a) 交互图      (b) 顺序图      (c) 部署图      (d) 通信图  
(e) 状态机图      (f) 类图      (g) 术语表
2. 哪些是软件开发中的传统步骤? (多选题)
- (a) 维护      (b) 设计      (c) 迭代      (d) 递增  
(e) 部署      (f) 分析      (g) 需求分析      (h) 测试  
(i) 重用      (j) 实现      (k) 规范

## 5.10 复习题答案

1. 下面哪个 UML 制品用于显示系统中过程、资源和对象的发布? (c) 部署图
2. 哪些是软件开发中的传统步骤?
- (a) 维护      (b) 设计      (e) 部署      (f) 分析  
(g) 需求分析      (h) 测试      (j) 实现      (k) 规范



# **第 II 部分**

## **理 解 问 题**

**第 6 章 收集需求**

**第 7 章 分析问题**



# 第6章 收集需求

本章将介绍软件开发的需求阶段，详细论述起点和用例，并探讨如何标识和验证业务用例，建立适合系统的业务模型。

## 学习目标：

- 理解需求阶段的目标
- 给业务上下文和系统功能建模
- 在完整的用例模型中记录系统需求

## 6.1 引言

需求阶段的目标有两个：

- 检查业务上下文：首先需要弄清楚开发软件的原因——如果没有好的理由，就不应编写软件。在决定开发软件系统后，就需要理解业务，对业务的理解应与客户理解相同——这也是弄清楚客户是谁的好机会。
- 描述系统需求：这不仅要决定系统的功能，还要找出所有的约束条件：性能、开发成本、资源等。

既然系统需求构成了需求阶段的一部分，那么为什么要给业务建模？图 6-1 是一个定义好的需求阶段的替代方式。这两个开发人员一开始对要开发的系统的认识就比较模糊，而且对客户的关注也不够。这种盲目的方法在程序员新手中比较常见，他们还不知道自己在干什么，要想成为高级专业人士，这种态度可要不得。



图 6-1 自以为是的开发人员

一头扎进编码中不仅仅源于傲慢自大，还可能因为恐惧：“我们不能确定可以开发出客户需要的系统，但我们知道我们能开发出什么系统，只要系统开发完成，就可以先告诉客户，我

们开发的系统就是他们需要的”。尽管系统的开发可能很难离开编码，但首先必须确保我们理解了新系统的业务上下文，然后与客户一起工作，对系统要完成什么任务达成一致。术语“客户”表示对最后交付的系统有兴趣的人。例如，运行系统的内部或外部顾客，潜在的最终用户、经理，甚至是股东。

在考虑编写软件前，必须调查软件运行的业务上下文——如果没有彻底理解业务，就很难开发出能增强该业务的系统(必须把目的定为增强已有的业务，否则就没有必要编写软件了)。这里使用的术语“业务”，其含义非常广泛：应该承认，本书偏重于银行、管理、电子商务等的业务系统，但这里介绍的大多数内容都适用于科学系统、家庭系统或需要软件的其他系统。如果愿意，还可以把“业务”看做“问题域”。

一旦理解了业务，并把这些理解整理为业务需求，就需要考虑软件应为用户完成什么任务。决定软件能干什么和软件不应干什么同等重要，这有助于只编写必要的代码。没有对系统需求的全面理解，就可能把时间浪费在开发客户免费的代码上(“付费”不应按字面理解，您可能给自己编写一个小系统，此时关注的就不是浪费时间了)。

系统需求常常分为两类：功能需求和非功能需求。功能需求是系统必须完成的工作，例如为响应外部刺激而必须提供的服务，如“浏览目录”和“预约汽车型号”。非功能需求是需要指定的其他需求，包括必须支持的客户 Web 浏览器，给广告使用的流视频(而不是可下载的文件)，Internet 冲浪新手很容易使用的用户界面等。

## 6.2 系统的诞生

每个系统都从某个位置开始。顾客可能提供了详细的文档，一般包含专用布局和目录；也可能只提供了任务陈述，这是他们想要的新业务方向的简短描述。

开发人员必须把客户的需求文档或任务陈述转化为完整的、清晰的、可用于开发系统的描述，采用客户能理解的、认可的标准格式。当然，“完整”和“清晰”实际上是做不到的。第一次是不可能非常接近这些目标的。但是，最终应有一个文档描述了系统应完成的所有工作(和系统不应完成的工作)，而且没有误解。

### 案例分析

#### Nowhere Cars 任务陈述

商店将汽车的跟踪自动化了——使用条形码、柜台终端和激光阅读器，这有许多优点：租赁助手的效率提高了 20%，汽车很少失踪，客户群很快变大(根据市场调查，其部分原因至少是专业化和效率的显著提高)。

管理层认为，Internet 会提供进一步提高效率、降低成本的机会。例如，现在不是打印可用汽车的目录，而可以让每个 Internet 冲浪人员在线浏览这些目录。对于有特权的客户，可以提供额外的服务，例如通过鼠标点击进行预约。这个领域的目标是每个商店的运营成本降低 15%。

在两年内，使用电子商务的所有功能，通过 Web 浏览器提供所有的服务，在客户家中完成汽车的交付和收回，以达到虚拟租赁公司的最终目标，将未预约业务的运营成本降到最低。

这个有三个段落的任务陈述包含了许多信息：公司的自动化历史；客户对日期的满意度；在线目录和预约；有特权和无特权的客户；节约成本的历史和目标；公司的最终目标（“虚拟租赁公司”）。当然，管理层的一些想法实现起来还有很长的路要走（客户适应虚拟租赁商店的时间可能会超过两年），但调查至少有两个很好的起点：公司的商店目前提供什么服务？哪些服务适合于在 Internet 上提供？

上述任务陈述是本书后面使用的案例分析的基础。虚拟公司的新系统称为 Coot，客户可使用的 Internet 功能称为 iCoot。

Nowhere Cars 的独特卖点是，它们在延长的期限内给富有的爱好者出租专用汽车。由于不可能出租所有型号的汽车，客户在要租汽车时，必须找到一家租赁店。汽车的租赁方式是先到先获得服务，客户可以在当前可用的汽车中选择。另外，如果客户要租用的某型号汽车目前没有，还可以预约。当有匹配的型号汽车时，助手就会与客户直接签约；客户必须在两天内取车（或交抵押金，先于其他客户取车）。但是，还没有在家中完成汽车交付和收回的服务（部分原因是保险）。会员必须注册，才能电话预约。

## 6.3 用例

Ivar Jacobson 发明了用例，以定义部分业务或系统的使用方式 [Jacobson 等, 92]。尽管初看起来，用例更趋于面向过程，而不是面向对象，但它们是描述系统功能需求的最高效工具。大多数非功能需求可能和密切相关的用例一起记录（其他信息可以单独列出）。

本书中的用例以便于学习的格式包含了所有的基本元素。在本书中，用例用于记录对业务运作方式的理解——业务需求建模——并指定新软件系统应能完成什么工作——系统需求建模。本书中的业务用例采用非正式的描述样式：为便于非专业人员阅读，它们描述了已有的内容。系统用例的说明性比较高：为便于软件开发人员理解，它们指定了需要实现的功能。

用例开始于一个参与者，称为参与者(actor)；之后是业务或系统，最后返回到参与者。每个用例的作用都应是参与者的价值（要不然，干吗要由参与者启动用例？）。当然，价值对不同的人意味着不同的事：它可以是参与者希望获得的一些信息、参与者希望从系统上获得的效果、金钱、购买某种商品，或者激发它们的其他事。由用例驱动，而不是按照传统路径进行，有助于找出对象、属性和操作。

### 案例分析

#### Nowhere Cars 任务陈述

- “会员预约汽车型号”是一个业务用例，它根据当前的情况，描述了会员的预约方式，这可以用任意汽车租赁业务中的术语来表达，也可以详细说明 Nowhere Cars 运作的方式。在业务建模过程中查找业务用例，这是需求分析的第一步。

业务用例可以使用已有的软件系统，也可以根本不涉及计算机。例如，上次给汽车租赁店打电话，预约某种汽车型号，如果当时电话另一端的助手正在进行交易，您能说出他使用的是计算机还是纸笔吗？而且，您在意他使用的是什么吗？

- 预约是一个系统用例，它描述了要开发的系统如何让 Nowhere Cars 通过 Internet 进行预约。系统用例描述了新系统或替代系统要提供的一个服务。在这个例子中，会员肯定使用了某

一种软件——Web 浏览器和后端服务器。我们的部分工作就是明确指定用户应提供的输入和他们可以得到的结果。

为了简单起见，用例，尤其是系统用例，不应重叠。用例以自然语言编写，分解为一系列步骤。如果需要更多的说明，用例可以使用图。

## 6.4 业务说明

本节介绍如何建立业务模型，这是给系统功能建模的前提条件。业务模型可以非常简单，只是一个类图，显示业务实体之间的关系，有时称为域模型。域模型对于小项目来说足够了。但是，对于大多数项目来说，需要建立一个完整的业务模型，表示业务的运作方式，或者至少表示构成系统的部分业务。

用例不是业务建模的惟一方式，但很简单。比较复杂的方法有业务过程建模和工作流分析。用例比较简单，是因为构建用例不需要专业知识，只需要常识和一定量的逻辑。这里建立的用例模型将包含用例本身和其他一些内容。

- 参与者表(带有描述)
- 术语表
- 用例(带有描述和细节)
- 通信图(可选)
- 活动图(可选)

UML 为活动图和通信图定义了表示法和语义。其他制品是 Jacobson 推荐的。

下面按照一般创建顺序，依次介绍业务模型的组件。但要注意，这不是一个硬性的工作流，与面向对象开发的所有方面一样，可以从前、从后迭代，或多次重复，直到得到完整的图为止。

### 6.4.1 标识业务参与者

首先，需要标识业务参与者。参与者是在业务中扮演某个角色的人(如名称所示)、部门或独立的软件系统。

把部门和系统当做参与者的原因是，在逻辑上，它们像人那样进行交互操作：我们对启动交互操作和一系列步骤的人(或部门、系统)感兴趣，而不关心参与者是“实现”为一个人，还是部门或软件。标识参与者有助于标识业务的使用方式，这有助于表示出用例的含义。

与现实生活一样，参与者可以在不同的时刻扮演不同的业务角色。例如，Fred Bloggs 在 Nowhere Cars 商店中是一个助手，直到商店关门为止；如果他在回家之前要租用一辆汽车，就成为一个顾客。在开发的这个阶段，要与其他客户(主要是顾客)一起工作，确定业务如何运作——参与者属于很容易讨论的一类。

#### 案例分析

##### Nowhere Cars 业务参与者表

- 助手：商店的一个员工，帮助顾客租用其汽车和预约汽车型号。
- 顾客：为获得一个标准服务而付费的人。
- 会员：其身份和信用状况已得到验证的顾客，因此，可以访问特定的服务(例如电话预约

或通过 Internet 预约)。

- 非会员：其身份和信用状况没有验证的顾客，因此，他要预约必须交押金，租用汽车必须提供一份驾照副本。
- Auk：处理顾客信息、预约、出租和可用汽车型号目录的已有系统。
- 债务部门：处理未付费用的 Nowhere Cars 部门。
- 法律部门：处理涉及租用汽车的事故的 Nowhere Cars 部门。

## 6.4.2 编写项目术语表

甚至在这个早期阶段，也最好开始维护术语表(glossary)——数据字典的现代替代品。“数据字典”包含“数据”，这是面向对象的理论家觉得不舒服的一个单词，因为它意味着，数据是孤立建模的。把数据从过程中分离出来是以前的老方式；最好把数据和过程存储在一起，因此使用感情色彩较弱的“术语表”。

术语表让查看软件开发制品的人觉得行话不再神秘。它还可以减小同义词的组合，允许在文档的其他地方自由使用同义词中的一个。

### 案例分析

#### Nowhere Cars 术语表

术    语	定    义
Car(业务对象)	由商店保存的、用于出租的 CarModel 的实例
CarModel(业务对象)	目录中的一个模型，可用于预约
Customer(业务参与者，业务对象)	为获得一个标准服务而付费的人
Member(业务对象)	其身份和信用状况已得到验证的顾客，因此，可以访问特定的服务(例如电话预约或通过 Internet 预约)

术语表中的每一项都定义了一个术语，其定义可短可长。前面看到的参与者描述是术语表定义的一个好开头，但术语表定义最后常常比较通用，因为大多数术语都将应用于几个上下文。

从案例分析的术语表中可以看出，可以记录每个术语与开发阶段之间的关系(业务参与者、系统参与者等)。下面是可以使用的关系列表(每一项都可以用于多种关系)：

- 业务参与者：业务需求中出现的参与者
- 业务对象：业务需求中出现的对象
- 系统参与者：系统需求中出现的参与者
- 系统对象：系统需求中出现(在系统内部)的对象
- 分析对象：分析模型中出现的对象
- 部署制品：在系统中部署的某个信息，例如文件
- 设计对象：设计模型中出现的对象
- 设计节点：构成系统体系结构的计算机或过程
- 设计层：子系统的垂直部分
- 设计包：类的逻辑组合，用于组织开发。

在这些项中，与往常一样，对象表示实体或“封装的数据和过程”。每个对象类别——业务、系统、分析或设计对象——都略有不同，一些对象可以归类到多个类别中。例如，当顾客

租用汽车时，我们处理的是一个系统外部的业务对象(显示区域中的物理交通工具)和一个系统内部的系统对象(它从已实现的类中实例化)。

术语表中的项目使用类命名样式(单词组合在一起，首字母大写)。只要在所有的项目文档中使用相同的样式，读者就知道术语表中有它的定义。

### 6.4.3 标识业务用例

有了参与者后，下一个任务就是标识业务用例。每个用例都是业务的一部分。在这个阶段，用例可能涉及许多参与者之间的双向通信，尤其在参与者是人的情况下，就更是如此。以后将看到，系统用例更结构化，因为人们一般会告诉系统要做什么，而不是采取其他方式完成。

如何把业务分解为用例没有设置好的规则？常识、逻辑和经验对此有一定的帮助。和客户一起工作(应总是与客户一起工作)也会有帮助。例如，在与销售层的助手谈话时，应尽力标识出他们每天都要完成的不同任务。因为助手要与顾客打交道，他们还应能标识出顾客使用业务的方式。员工和管理层培训手册、任务陈述、专用需求文档、销售册子和其他文档也可以提供灵感。在尽力找出用例时，应问自己如下问题：“使这个业务运转起来的重要活动是什么？”

#### 案例分析

##### iCoot 业务用例表

- B1: 顾客租用汽车：顾客租用从可用汽车中选择出来的汽车。
- B2: 会员预约汽车型号：当有该型号的汽车时，会员应得到通知。
- B3: 非会员预约汽车型号：当有该型号的汽车时，非会员交纳了押金，就应得到通知。
- B4: 顾客取消预约：顾客通过电话或亲自取消未结束的预约。
- B5: 顾客交还汽车：顾客交还他租用的汽车。
- B6: 顾客获知有某型号的汽车：当有该型号的汽车时，助手会与顾客联系。
- B7: 报告失踪：顾客或助手发现汽车失踪了。
- B8: 顾客重新预约：超过一星期后，顾客可以重新预约。
- B9: 顾客访问目录：顾客在店内或在家中浏览目录。
- B10: 顾客因没有取预约的车而接受罚款：顾客没有取预约好的车。
- B11: 顾客取预约好的车：顾客取预约好的车。
- B12: 顾客成为会员：顾客提供信用卡信息和地址证明，成为会员。
- B13: 通知顾客汽车已超过租用期限：助手与顾客联系，警告顾客他租用的汽车已超过租用期限一星期了。
- B14: 顾客丢了钥匙：为丢钥匙的顾客提供备用钥匙。
- B15: 更新会员卡：当会员卡过期时，助手与顾客联系，更新会员卡。
- B16: 汽车不能还回来：汽车出事或坏了。

现在开始把业务缩小到我们特别感兴趣的范围内。从最初的任务陈述中得知，顾客只希望通过 Internet 进行部分租赁和预约业务，例如，没有给退出租赁的汽车销售建模。

记住，在业务建模过程中，我们对新系统的运作方式不感兴趣。在这个阶段，只是尽力描述业务当前运作的方式。这可能涉及，也可能不涉及已有的软件。

编号模式是任意的：UML 没有指定该模式，列表也没有隐含顺序。一旦有了候选的用例列表，就可以列出每个用例涉及的步骤。UML 没有指定用例的内容(或编号和描述)，所以，可

以自由使用自然语言、一步步的描述、结构化语言(包含 if-then-else 和循环结构的自然语言)或其他方式。

使用步骤而不是自然语言，有助于只涉及用例的主干。如果使用结构化语言，就可能使描述算法化(面向计算机)。为了使用例更简洁、独立于实现代码，本书给用例细节使用未结构化的步骤，如图 6-2 所示。

B3：非会员预约汽车型号。

1. 非会员告诉助手要预约的汽车型号。
2. 助手在 Auk 中查找该汽车型号。
3. 助手请求非会员为预约交纳押金。
4. 助手请求非会员提供驾照和电话号码。
5. 助手检查非会员的驾照。
6. 如果驾照没有问题，助手就会创建新的预约，并记录驾照号码、电话号码，在 Auk 中扫描驾照。
7. 助手给非会员一个预约卡，其中包含惟一的预约号。

图 6-2 Nowhere Cars 的业务用例

#### 6.4.4 在通信图中演示用例

除了写下用例的细节之外，还可以使用通信图提供用例的演示。通信图显示了参与者和对象之间的一系列交互。顺序图关注的是交互本身和交互发生的顺序。为了禁止在开发过程的早期使用过多的技术细节，通信图最好用于业务建模。

因为 UML 可用于每一种可能的情况，所以开发人员可以在每种不同的图中使用许多表示法。本书为了避免介绍所有的细节，只使用基本部分，也忽略了演示这些信息和其他较复杂信息的其他方式。

图 6-3 显示了五个通信元素。每个元素的特性都通过用于表示它的图标来显示：小人表示参与者，线上的圆表示业务对象或实体，与竖线相连的圆表示边界(它管理其他元素之间的交互——通常这是一种软件，但也可以是一个人)。边界图标中有一个小人表示承担某种交互角色的参与者(边界是人，而不是一个软件)。

即使对表示法不是特别了解，也很容易看出图 6-3 中的预约涉及一个非会员、用作业务边界的助手、一个名为 Auk 的软件(用作系统的边界)和两个业务对象。因为在这个阶段，我们要给完成任务的已有方式建模，接口和它访问的系统必须已经实现和部署好，不涉及最终要实现的软件，因为还没有进入那个阶段。

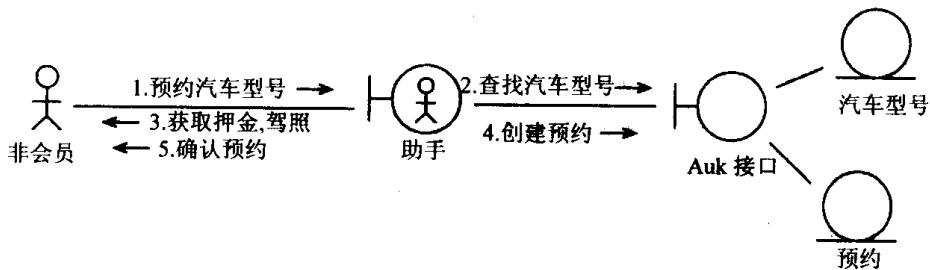


图 6-3 B3:非会员预约汽车型号的通信图

在通信图上，连接两个元素的线条表示这两个元素可以交互。所以在图 6-3 中，非会员向

助手请求服务，助手向 Auk 接口请求服务，Auk 接口向汽车型号和预约请求服务。汽车型号和预约是通过软件接口访问的，这说明它们是已有系统中的软件对象，而不是外部的物理对象(这个区别对于我们来说，可能重要，也可能不重要)。

除了图标和连接之外，图 6-3 把各个交互操作描述为带相关箭头的编号标签。可以把这些看做要从一个元素发送给另一个元素的消息：号码表示消息在序列中的位置。这样，就可以把整个过程解释为：

- 非会员请求助手预约某种汽车型号
- 助手请求 Auk 接口查找该汽车型号(以某种未指定的方式涉及 CarModel 对象)
- 助手请求非会员交纳押金，提供驾照。
- 助手请求 Auk 接口创建预约(以某种方式涉及 Reservation 对象)
- 助手给非会员确认预约。

您可能认为，这个系列的交互应精确匹配用例细节中的步骤系列。但是因为自然语言不是一系列步骤，所以一对一的匹配是不可能的。每个交互更可能表示一步或多步。

尽管不与用例精确匹配，通信图仍是有用的，它增强了用例的细节，有助于生成细节。

在这个早期阶段处理的交互是很简单的，所以可以给每个用例生成一个通信图，但这里没有这么做。为了简短起见，我们描述的任何协作都应是整个用例中的正常路径。在处理系统用例时，将更多地涉及异常路径，但现在应在用例中隐藏它们。例如，如果驾照没有问题，业务用例中的第 6 步就会开始：这隐含着，有时驾照是无效的，但没有指定助手在这种情况下应怎么做。

#### 6.4.5 在活动图中演示用例

UML 包含另一种在业务建模过程中有用的图。在从初始起点到最终目标的过程中，活动图显示了(平行)活动之间的依赖关系。该图类似于流程图或 Petri 网络，它们传统上用于给程序流程或人类的活动建模。图 6-4 显示了用于演示业务用例的活动图。

活动图中的每个圆角方框都表示一个动作；箭头(边界)表示源动作应在目标动作开始之前完成；黑点表示活动的起点；白圆中的黑点表示活动的结束；菱形表示决策——远离菱形的边上的文本(guard)表示沿着该边的原因；粗黑线条称为叉子或连接，用于表示一组并行动作的开始和结束。

对于每个动作，都可以在动作名称的前面，在括号中放置一个名字，显示由谁负责该动作。该名字表示活动中的一个参与者，可以用于标识参与者、部门、系统或对象。参与者还可以通过把动作组合到行、列或单元格中来表示。

从图 6-4 中可以看出，预约汽车型号是很简单的：

1. 非会员告诉助手要预约的汽车型号。
2. 助手请求非会员交纳押金，提供驾照。
3. 在非会员查找押金和驾照的同时，助手也在 Auk 系统上查找该汽车型号。
4. 找到了这些东西后，助手就检查押金和驾照。
5. 如果非会员交了押金，驾照也有效，助手就进行预约，活动结束。
6. 否则，活动结束。

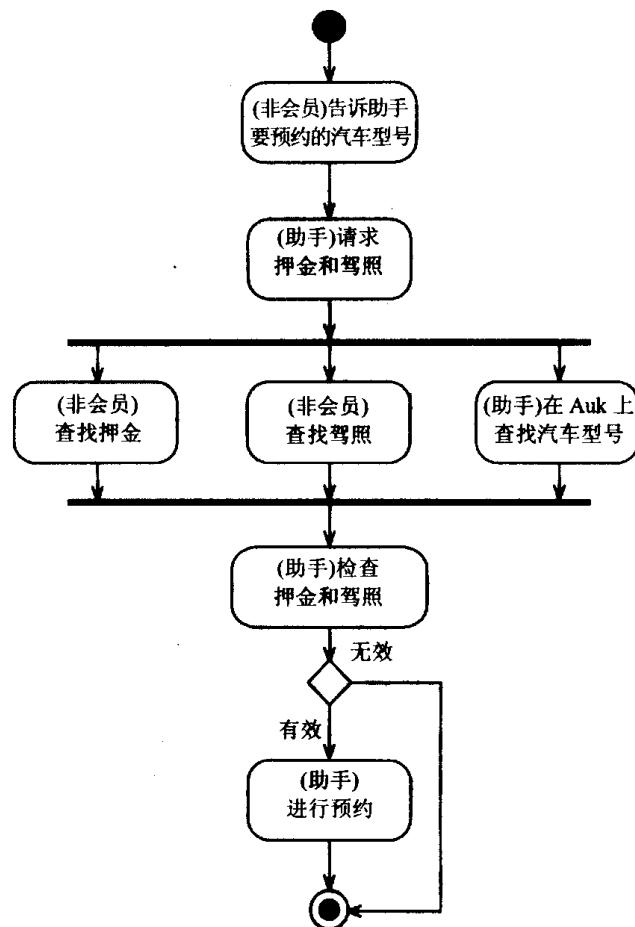


图 6-4 B3: 非会员预约汽车型号的活动图

与通信图一样，这个解释也不会每一步都匹配用例的细节。

与许多 UML 图一样，活动图也可以用于多种目的。例如，活动图可以用于构建整个业务模型，或记录某个软件对象使用的算法。

## 6.5 开发人员的说明

需求分析的第二部分是给要开发的软件建模，以改进业务。无论记录业务是选择使用简单的域模型、完全成熟的用例模型，还是更详细的模型，例如业务过程模型或工作流分析，软件系统的需求都应通过用例模型来分析。这是因为用例很容易生成，客户也容易理解。

系统的用例模型比业务的用例模型更详细、更具说明性。对于 Rippke，系统用例模型包括：

- 参与者表(带有描述)
- 用例列表(带有描述)
- 用例图
- 用例细节(包括所有相关的非功能需求)
- 用例调查
- 辅助需求(不符合任何用例的系统需求)

- 用户界面草图
- 改进的术语表
- 用例的优先级

下面依次介绍如何生成这些制品。

上述列表包含几个以前没有遇到过的制品。甚至在业务建模过程中已经介绍过的制品，在系统建模的过程中也通常包含更多的信息。通信图没有包含进来；尽管在这个阶段也可以使用通信图演示系统用例，但对于 Ripple，应在开发的后期阶段(动态分析)使用它们，那时它们比较重要。

还应花一些时间评估已有的系统。这种系统称为旧系统，因为它被继承为已有业务的一部分。

### 案例分析

#### Nowhere Cars 旧系统

需要决定是可以扩展 Auk，还是应替换它。这个决定不容易做出：一方面，这个已完成的系统已经启动、运行一段时间了，助手很熟悉它；另一方面，很难打开 Auk，给它添加新功能，或逐个过程地编写与它通信的新软件(其效率可能很低)。

假定决定用一个全新的系统替代 Auk，新系统兼容了 Internet 访问，支持顾客希望的“虚拟租赁店”。新系统称为 Coot。为了缓解员工培训的问题，新接口(柜台终端和激光阅读器)的外观和操作方式应类似于 Auk 的接口。

从本书要达到的目标来看，不需要介绍 Coot 的所有功能，而可以集中讨论为顾客提供 Internet 功能的系统部分。这个删减的功能集称为 iCoot。

### 1. 标识系统参与者

首先，需要在客户的帮助下标识和描述系统参与者。这个阶段标识的参与者应只包括直接与系统交互的人(和外部系统)，而不包括更宽泛的业务环境中的参与者。

### 案例分析

#### iCoot 系统参与者表

- 顾客：使用 Web 浏览器访问 iCoot 的人。
- 会员：在一家商店提供了姓名、地址和信用卡信息的顾客；每个会员都有一个 Internet 密码和一个会员号。
- 助手：商店的一个员工，他与会员联系，告诉他们预约的进展情况。

### 2. 标识系统用例

一旦有了参与者，就可以查找用例，这也可以从客户那里获得帮助，每个用例都必须有简短的说明。

### 案例分析

#### iCoot 系统用例表

- U1: 浏览索引：顾客浏览汽车型号的索引。

- U2:查看结果：给顾客显示检索到的汽车型号子集。
- U3:查看汽车型号的细节：给顾客显示检索到的汽车型号细节，例如描述和广告。
- U4:搜索：顾客指定类别、构造和引擎规格，搜索汽车型号。
- U5:登录：会员使用会员号和当前密码登录 iCoot。
- U6:查看会员信息：会员查看 iCoot 存储的会员信息子集，例如姓名、地址和信用卡调节。
- U7:进行预约：会员在查看汽车型号的细节时，预约一种汽车型号。
- U8:查看租用情况：会员查看当前租用的汽车汇总信息。
- U9:修改密码：会员修改用于登录的密码。
- U10:查看预约情况：会员查看还没有结束的预约汇总信息，例如日期、时间和汽车型号。
- U11:取消预约：会员取消还没有结束的预约。
- U12:注销：会员从 iCoot 中注销。

系统用例可以在用例图上描述，显示参与者和他们与特定用例的关系——这有助于了解系统的使用方式。iCoot 的用例图如图 6-5 所示。

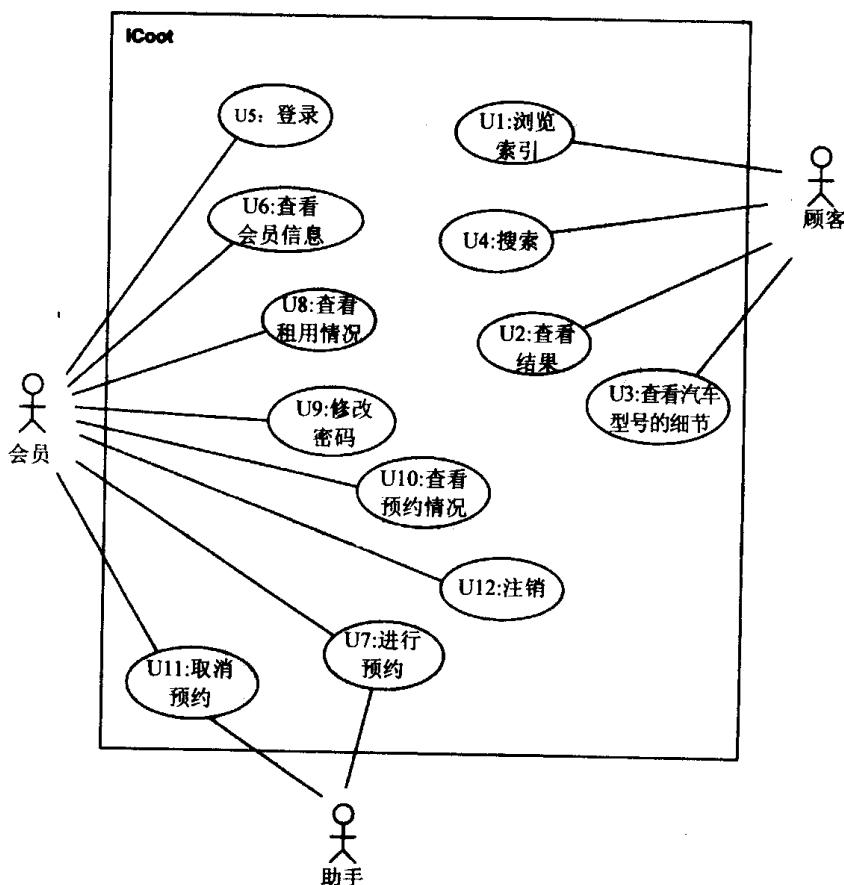


图 6-5 iCoot 的一个简单的用例图

在用例图中，每个用例都在一个椭圆中显示为一个序号和一个标题。包含所有用例的方框表示系统的边界——可以把系统名称放在方框中。在系统边界的外部，显示参与者，在用例和

使用它们的参与者之间添加关联。

用例调查(use case survey, Jacobson 用于表示非 UML 制品的术语)是一个非正式的描述，说明了一组用例如何组合在一起。它是开发人员在与客户一起研究用例图时生成的叙述。用例调查允许客户在没有开发人员的帮助下，也能更好地理解用例。

### 案例分析

#### iCoot 用例调查

任何顾客都可以浏览汽车型号索引(U1)或通过搜索(U4)，在目录中查找汽车型号。在后一种情况下，顾客要指定他们感兴趣的类别、构造和引擎规格。无论采用哪种方式，在每次检索后，都会给顾客显示匹配汽车型号的集合(U2)，以及基本信息，例如汽车型号的名称。然后，顾客就可以选择查看特定汽车型号的其他信息，例如描述和广告(U3)。

已成为会员的顾客可以登录(U5)，访问额外的服务。额外的服务有进行预约(U7)，取消预约(U11)，检查会员信息(U6)，查看已有的预约(U10)，修改登录密码(U9)，查看已有的租用记录(U8)和注销(U12)。

助手涉及到预约的整个过程，例如把汽车开进开出保留区域。

### 6.5.1 使参与者特殊化

参与者可以使另一个参与者特殊化(specialize，继承它的行为)。这会给系统用例模型添加更多的表达方式。例如，可以把顾客设计为抽象的概念，会员应根据该概念进行特殊化；一旦引入了特殊化，引入非会员的概念就是有意义的。

您可以决定是在早期还是在后期在参与者之间引入继承，或者根本不引入继承。继承的每次使用都是有益的，而不是带来混乱。客户必须理解我们生成的制品，至少要理解从业务建模到静态分析的正确方式。这有助于确保正确理解问题域，交付客户真正想要的系统。如果非会员不是顾客，非程序员是否会高兴？这取决于我们的决定。

### 案例分析

#### iCoot 带继承的系统参与者表

决定给 iCoot 参与者引入继承后，完成的系统参与者表如下所示(其中多了一个参与者，继承关系放在括号中)：

- 顾客：使用 Web 浏览器访问 iCoot 的人。
- 会员：在一家商店提供了姓名、地址和信用卡信息的顾客；每个会员都有一个 Internet 密码和一个会员号。(特殊化顾客)
- 非会员：不是会员的顾客。(特殊化顾客)
- 助手：商店的一个员工，他与会员联系，告诉他们预约的进展情况。

可以修改用例图，显示参与者之间的继承关系，其方式与显示类之间的继承关系一样，如图 6-6 所示。Customer 类显示为抽象类，其名称是斜体，因为在字面上看，任何人都不是顾客，而是会员或非会员(如果不能手写出斜体，就可以在参与者名的左边或上方加上关键字 {abstract})。

即使 Customer 是抽象类，也可以把它关联到用例上，表示在该用例中涉及的每类顾客。

另外，一些参与者，例如非会员，就只有间接关联。

这个图现在显式显示了特殊化，也可以在参与者表中添加注释，此时列表的查看方式是不同的。

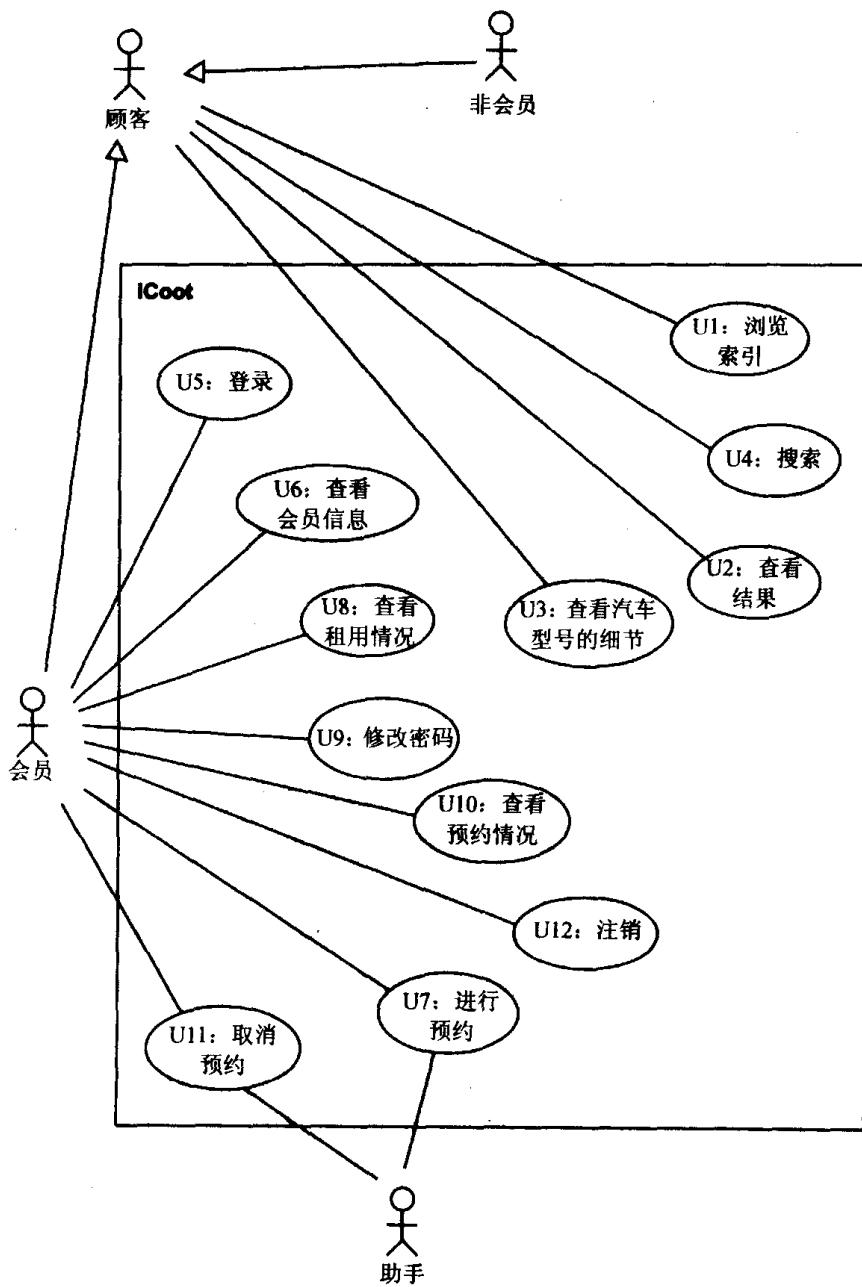


图 6-6 显示参与者之间继承关系的用例图

### 6.5.2 用例的关系

除了参与者之间的特殊化以及参与者和用例之间的关系之外，用例之间还有三种关系：特殊化(specialize)、包含(include)和扩展(extend)。这些关系可以组合相关的用例，分解大的用例，重用行为，指定可选的行为。

- 特殊化：与参与者一样，用例也可以相互继承。为了避免重新定义步骤和添加额外的步骤，可以只特殊化抽象的用例。(纯)抽象用例根本没有步骤，其惟一的目的是组合其他用例。例如，“U1:浏览索引”和“U4:搜索”都是抽象用例“U13:查找汽车模型”的变体。
- 包含：如果第一个用例有一些第二个用例提供的步骤，该用例就包含第二个用例。例如，“U1:浏览索引”在其行为的某个地方包含“U2:查看结果”的全部步骤。包含关系可以用于把相同的步骤提取到许多用例中，或者把大的用例分解为更容易管理的块。
- 扩展：第一个用例给第二个用例增加步骤，就称为扩展第二个用例。例如，在查看结果(U2)时，顾客可以选择查看细节(U3)。扩展关系可以增加可选的额外步骤——通常，这些额外的步骤位于用例的末尾，但也可以位于用例的开头或中间。

包含关系和扩展关系有一个根本区别：在包含关系中，源用例没有目的用例就不能工作，而在扩展关系中，源用例即使没有目的用例也能工作得很好。从另一个方面来看，包含在其他用例中的用例可以独立存在——它们可以通过其他路径直接扩展。而扩展另一个用例的用例通常只能作为扩展的用例而存在。

在第一次建立系统需求模型时，不可能标识出用例的关系。除此以外，还应确定它们是否真的需要，客户是否能认同它们。如果使用了关系和面向对象的其他方面，就有许多方法把用例分解为包含、扩展和继承关系。哪一种关系正确是无法判断的，只能试着开发一个对自己和顾客都有意义的模型。

## 案例分析

### iCoot 用例的关系

用例描述现在如下所示。抽象用例和关系的两端都标识清楚了——基本原则是，看一下表示法，读者就知道需要查看其他用例，才能完整地理解该用例。

- U1:浏览索引：顾客浏览汽车型号的索引(特殊化 U13，包含 U2)。
- U2:查看结果：给顾客显示检索到的汽车型号子集(被 U1 和 U4 包含，被 U3 扩展)。
- U3:查看汽车型号的细节：给顾客显示检索到的汽车型号细节，例如描述和广告(扩展 U2，被 U7 扩展)。
- U4:搜索：顾客指定类别、构造和引擎规格，搜索汽车型号(特殊化 U13，包含 U2)。
- U5:登录：会员使用会员号和当前密码登录 iCoot(由 U6、U8、U9、U10 和 U12 扩展)。
- U6:查看会员信息：会员查看 iCoot 存储的会员信息子集如姓名、地址和信用卡信息(扩展 U5)。
- U7:进行预约：会员在查看汽车型号的细节时，预约一种汽车型号(扩展 U3)。
- U8:查看租用情况：会员查看当前租用的汽车汇总信息(扩展 U5)。
- U9:修改密码：会员修改用于登录的密码(扩展 U5)。
- U10:查看预约情况：会员查看还没有结束的预约汇总信息，例如日期、时间和汽车型号(扩展 U5，被 U11 扩展)。
- U11:取消预约：会员取消还没有结束的预约(扩展 U10)。
- U12:注销：会员从 iCoot 中注销(扩展 U5)。
- U13:查找汽车型号：顾客从目录表中检索汽车型号的子集(抽象，被 U1 和 U4 一般化)。

用例关系可以在用例图中显示出来(如图 6-7 所示)。用例之间的继承关系以正常方式显示，

即用带有白箭头的线条。包含关系显示为箭头开放的虚线，从包含的用例指向被包含的用例，并标记上关键字<<include>>(UML 中放在双尖括号中的单词表示一个常见的概念)。因此，“U4: 搜索”在其行为中包含“U2: 查看结果”的所有步骤。扩展也显示为箭头开放的虚线，从扩展的用例指向被扩展的用例，并标记上关键字<<extend>>。所以，“U3: 查看汽车型号的细节”就是“U2: 查看结果”的一个可选步骤。

可以把图中显示的两个包含关系减少为一个包含关系，即从 U13 到 U2，但这样就不是很清晰。这还意味着抽象用例可以有步骤，而这是我们竭力避免的。最后，它还表示 U1 和 U4 不能控制包含关系出现的位置。

在一些情况下，扩展关系只能在某些条件具备的情况下存在。为了说明这一点，可以添加一个 UML 注释(看起来像一张纸)来说明该条件。注释可以包含任意文本，在图中用虚线连接相关的点，终止于一个小圆圈，使连接更清晰。UML 中的条件表示为约束(constraint，花括号中的文本)，用自然语言、伪代码或 UML 的正式对象约束语言(Object Constraint Language, OCL)来表述。在图 6-7 中，使用了自然语言，反映了我们正处在开发的非正式阶段。可以看出，只有已登录的 Member 对象才允许“U7: 进行预约”。

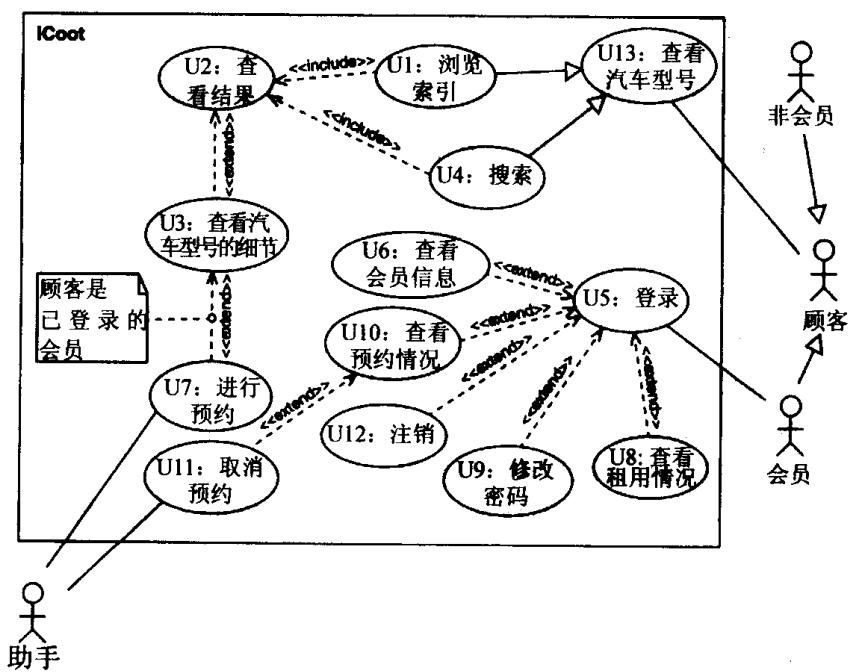


图 6-7 iCoot 的最终用例图

正常情况下，UML 中带有开放箭头的虚线表示依赖性——源用例以某种方式依赖目的用例。其含义是，如果目的用例改变了，源用例就会受影响。对于用例，使用依赖表示法有历史方面的原因，严格说来并不十分正确。例如，扩展用例不需要依赖被扩展的用例，因此，在用例图中，我们说的是用例关系，而不是依赖性。有关表示法的最后一个注意事项是：对于包含关系，下级用例是目标用例，而对于扩展关系，下级用例是源用例，这可能会产生混乱。

## 案例分析

### iCoot 用例调查(完整)

任何顾客都可以浏览汽车型号索引(U1)或通过搜索(U4)，在目录中查找汽车型号。在这种情况下，顾客要指定他们感兴趣的类别、构造和引擎规格。无论采用哪种方式，在每次检索后，都会给顾客显示匹配汽车型号的集合(U2)，以及基本信息，例如汽车型号的名称。然后，顾客就可以选择查看特定汽车型号的其他信息，例如描述和广告(U3)。

顾客有两种情况：会员和非会员。

已成为会员的顾客可以登录(U5)，访问额外的服务。额外的服务有进行预约(U7)，取消预约(U11)，检查会员信息(U6)，查看已有的预约(U10)，修改登录密码(U9)，查看已有的租用记录(U8)和注销(U12)。

助手涉及到预约的整个过程，例如把汽车开进开出保留区域。

浏览索引和搜索汽车型号是查找汽车型号(U13)的两种不同方式。为了查看汽车型号的细节，顾客必须查看搜索型号的结果(通过浏览或搜索路径)。

为了预约汽车型号，会员必须查看其细节(非会员不能预约，即使他们在查看细节也不行)。

要取消预约，会员必须查看已有的预约。

### 6.5.3 系统用例的细节

一旦标识了用例，指定了它们的组合方式，就需要显示细节。UML 没有指定应包含哪些用例细节或如何安排它们，所以这里可以根据品位和经验来选择。对于 Ripple，系统用例的细节包括：

- 用例号和标题
- 用例是否为抽象的
- 与其他用例的关系
- 前提条件(在执行用例之前必须满足的条件)
- 步骤(假定满足了前提条件)
- 后置条件(在完成用例后保证满足的条件)
- 异常路径和在这些情况下应做什么(尽管路径是异常的，但如果它们对指定系统的反应非常重要，也应包含它们)
- 与这个用例相关的非功能需求

图 6-8 显示了本书使用的用例细节的格式；在所有的项中，只有编号、标题、前提条件、步骤和后置条件是必须有的，其他都可以为空。

编号，标题(关系)
前提条件
步骤
后置条件
异常路径
非功能需求

图 6-8 系统用例细节的格式

图 6-9 显示了特定格式的 4 个 iCoot 用例。

### U1: 浏览索引(特殊化 U13, 包含 U2)

前提条件: 无

1. 顾客选择一个索引标题。
2. 顾客选择查看选中索引标题的汽车型号
3. 包含 U2

后置条件: 无

### U3: 查看汽车型号的细节(扩展 U2, 被 U7 扩展)

前提条件: 无

1. 顾客选择一个匹配的汽车型号。
2. 顾客请求选中汽车型号的细节。
3. iCoot 显示选中汽车型号的细节(构造、引擎规格、价格、描述、广告和海报)。
4. 如果顾客是一个已登录的会员, 就用 U7 扩展。

后置条件: iCoot 显示选中汽车型号的细节。

非功能需求:

- rl. 广告应使用流协议显示, 而不应要求下载。

### U5: 登录(由 U6、U8、U9、U10 和 U12 扩展)

前提条件: 会员从本地商店获得一个密码。

1. 会员输入会员号。
2. 会员输入密码。
3. iCoot 强制会员必须登录, 所以会员可以选择盗取(验证无效, 所以盗取)已有的会话。
4. 会员选择登录。
5. 用 U6、U8、U9、U10 和 U12 扩展。

后置条件: 会员登录。

异常路径:

- a1. 如果会员号和密码组合是不正确的, iCoot 会通知会员, 这两个中的一个不正确(为了安全起见, 不会说明是哪一个不正确)。
- a2. 如果会员号和密码组合是正确的, 但会员已经登录, 且没有选择盗取会话, iCoot 会通知会员。

### U13: 查找汽车型号(抽象, 由 U1 和 U4 特殊化)

前提条件: 无

后置条件: 给顾客显示检索到的汽车型号汇总信息。

图 6-9 一些 iCoot 系统用例的细节

这些系统用例都比前面的业务用例更详细。这说明, 现在我们正试图具体说明, 而不是仅仅描述一下: 明确系统将提供的服务, 以免分析员和设计人员进行猜测。

本书使用以自然语言编写的一系列步骤。也可以根据个人喜好添加更符合算法的结构, 例如条件和循环(例如 if-then-else 和 repeat-until)。

在编写用例的细节时, 必须指定系统的功能, 而不是指定交付功能的方式。例如, 如果要包含步骤“2. 顾客单击 Detail...按钮”, 就要限定用户界面的设计人员。除非这是一个绝对要求, 否则就应总是使用中性词语, 例如选择、启动、指出和显示。

理性的设计人员应对需求收集器的特长了如指掌。例如, 设计人员要以不同的顺序完成步

骤，或者只要求最终结果是相同的。“U5:登录”就是一个这样的用例；前三步可以以任意顺序完成，对任何人都没有区别。

#### 6.5.4 前提条件、后置条件和继承

前面已经考虑了用例之间的继承关系，下面关注特殊化对前提条件和后置条件的影响。(建议只继承没有步骤的抽象用例，这种用例仍可以有前提条件和后置条件，如图 6-9 所示)。下面是规则：

1. 当一个用例特殊化另一个用例时，会继承父用例的前提条件，作为起点。子用例添加的新前提条件只能弱化继承的前提条件(使用 or 合并)。
2. 对于后置条件，子用例的起点是父用例的后置条件。子用例添加的新后置条件只能强化继承的后置条件(使用 and 合并)。
3. 子用例添加的前提条件和后置条件对父用例的前提条件和后置条件没有影响。

从已经知道的面向对象理论来看，上述列表中的规则 3 是很明显的(子不能影响父的行为)，但规则 1 和规则 2 可能有点奇怪。只能弱化前提条件和只能强化后置条件的原因是，子用例对父用例的读者是有义务的，这没有什么可惊讶的。例如，“U13:查找汽车型号”没有前提条件，如果“U4:搜索”有一个前提条件“不能在周二进行搜索”，在周二被阻止搜索汽车型号的人就有合理的理由抱怨——“对不起，根据‘查找汽车型号’，我可以在任何时候查找。”

父用例上的后置条件给用户提供了一个保证，子用例不应冲淡这种保证。例如，“U4:搜索”有一个后置条件“给顾客显示匹配的汽车型号”；子用例不能添加任意选择的汽车型号。

规则 1 和规则 2 说明，如果父用例的前提条件是“无”(应用该用例时没有限制)，其子用例的前提条件也必须是“无”；如果父用例的后置条件是“无”(没有对结果做出保证)，子用例可以指定需要的后置条件。

总之，当一个用例特殊化另一个用例时，必须仔细考虑父用例的前提条件和后置条件。

#### 6.5.5 辅助需求

在大多数情况下，可以把非功能需求关联到特定的用例上。例如，非功能需求“广告应使用流协议显示，而不需下载”比较适合“U3:查看汽车型号的细节”，该用例可以让顾客看到广告。

不适合任何用例的非功能需求可以记录在辅助需求文档中，如图 6-10 所示。

辅助需求：

- s1. 客户小程序必须运行在 Java PlugIn 1.2(和更新的版本)上。
- s2. iCoot 必须能处理 100,000 种汽车型号。
- s3. iCoot 必须能同时给一百万个客户服务，且性能没有明显的降低。

图 6-10 iCoot 的辅助需求

#### 6.5.6 用户界面草案

为系统考虑用户界面有助于阐明用例。界面可以在早期阶段与客户一起讨论，并把结果记录为用户界面草案(user interface sketches)。这些草案应看做是基本指南，而不是专业的 GUI 设计，它们有助于标识和分解能根据个人喜好来实现的功能。

例如,图 6-11 中的草案 1 显示的用户界面允许用户选择一个或多个类别、厂商和引擎规格。单击 Retrieve 按钮,就可以进入草案 2,其中显示了匹配的汽车型号列表;单击< Go Back 按钮,就会返回草案 1;单击 Details...按钮可以进入草案 3(没有显示)等。(显然,这些都不是最初手工绘制在白板上的草案,它们是经过讨论通过,然后使用绘图软件包仿制的版本。)

用例和用户界面都表示系统功能的分解,所以最好在两者之间保持清晰的映射,这种映射应一直保持到实现阶段结束为止。例如,在 iCoot 中,有三大类访问:会员访问、非会员访问和助手访问。所以应有三种不同的用户界面。

在每个用户界面中,都应提供对应各个用例的窗口或面板(当然,选择窗口、面板或其他组件是一个设计问题)。例如,图 6-11 中的草案显示了一个笔记本样式的组件,它表示非会员界面。非会员可以搜索类别,浏览索引,所以这些用例都把自己的页面放在笔记本上。这两个用例包含查看结果——另一个用例有自己的面板。在会员界面上应可以重用 Search、Index 和 Results 面板。(为便于移植,助手界面应类似于已有的 Auk 界面)。

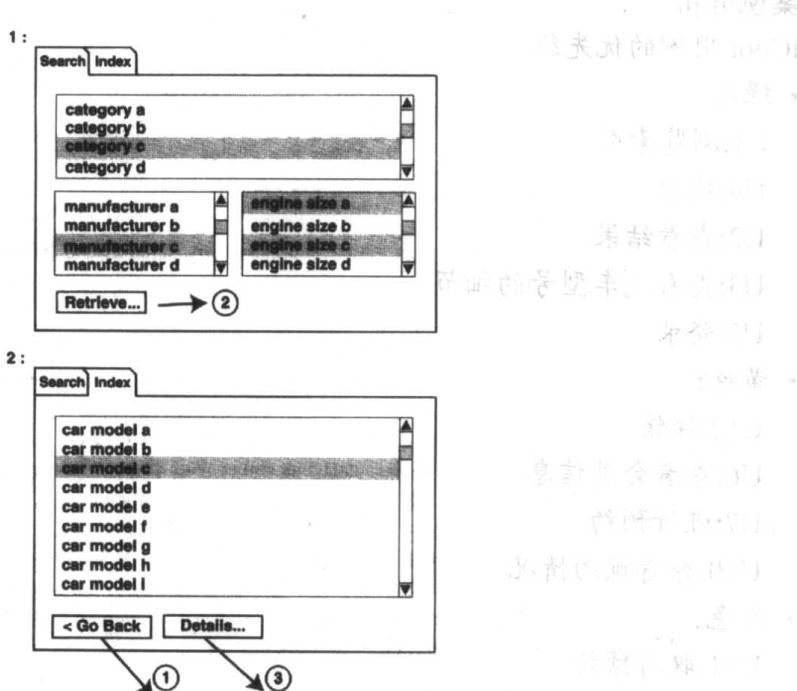


图 6-11 iCoot 的用户界面草案

### 6.5.7 系统用例的优先级

最好按照实现的优先级给系统需求分级,尤其是在递增开发过程中,就更应分级。在用例建模过程中,显然应给用例分级,然后给每个用例打分,表示其紧急程度。优先级和紧急程度有助于规划其他开发过程和进一步的递增开发过程。

有效的打分技术是交通灯(traffic light):

- 绿色(Green)的用例必须在当前的递增开发过程中实现;否则就意味着项目没有达到其最低目标。
- 黄色(Amber)的用例在当前的递增开发过程中是可选的,只有在完成了绿色的用例之后才能尝试完成它(它们是增加的红利,可以用于给客户加深印象)。未在交付日期之前完成的黄色用例必须完全舍弃(只实现了一部分看起来很不专业)。

- 红色(Red)的用例即使时间允许，也不在当前的递增开发过程中实现；它们在当前的递增开发过程之外，一般不允许完成。

实际上，用例的优先级(紧急程度)不仅取决于期望，还取决于在当前的递增开发过程中为各个用例投入的系统体系结构和编码工作量。选择优先级需要一定的技巧、经验和预测能力。把较容易的用例放在前面总是不会错，这样有助于在开始时更多地了解系统，风险也较小。

如果在递增开发过程的最后还有时间(完成绿色用例和所有的黄色用例)，就应：

- 检查项目的状态
- 完成下一个递增开发过程的规划(例如，对未完成的用例再次分级)
- 做一些无关的工作
- 开一个办公室派对
- ...

## 案例分析

### iCoot 用例的优先级

- 绿色：

U1:浏览索引

U4:搜索

U2:查看结果

U3:查看汽车型号的细节

U5:登录

- 黄色：

U12:注销

U6:查看会员信息

U7:进行预约

U10:查看预约情况

- 红色：

U11:取消预约

U8:查看租用情况

U9:修改密码

“U1:浏览索引”是必要的，也很简单(因为它不涉及租用和预约)，所以应放在列表的最前面。“U5:登录”必须在能提供会员服务之前进行，所以必须放在会员服务之前；“U6:查看会员信息”应在“U7:进行预约”之前进行，因为它比较简单(预约有一个复杂的生命周期)；等。

给系统用例分配优先级和紧急程度还说明，应为扩展性和重用性进行开发。下面总结了在业务建模之后，交通灯如何用于开发的其他阶段：

- 绿色：这个组中的用例应完成系统需求、分析、系统设计、子系统设计、规范、实现和测试。
- 黄色：这个组中的用例应完成系统需求，分析和系统设计应完成或接近完成，子系统设计、规范、实现和测试是可选的。
- 红色：这个组中的用例应完成系统需求，分析是可选的，系统设计应支持这些用例，子系统设计、规范、实现和测试不应完成。

当然，在螺旋式、迭代式和递增过程中，“完成”是相对的。

## 6.6 小结

本章的主要内容如下：

- 在编码开始之前，需求阶段应指定功能需求(系统必须能完成的工作，例如“浏览目录”)和非功能需求(系统运转的方式，例如必须支持特定的 Web 浏览器)。
- 使用高级业务用例给业务上下文和系统功能建模，并标识参与者。
- 用完整的用例模型给系统需求建模，完整的用例模型包括用例、用例图、辅助需求、用户界面草案、用例优先级和紧急程度。在这个阶段，通信图和活动图是可选的，但术语表总是必须有的。

## 6.7 课外阅读

作者编写的一本图书[Bustard 等, 00]介绍了业务过程建模理论的有用信息，但没有介绍 UML。

Ivar Jacobson 编写的、介绍对象方法的入门图书[Jacobson 等, 92]，描述了用例的原始理由和它们应包含的信息种类——看看重要的技术最初来自于什么总是不错的。Alistair Cockburn 是用例领域获得广泛尊敬的一位权威人士，除了一本书[Cockburn 00]之外，他还拥有自己的网站 [www.usecases.org](http://www.usecases.org)。

要了解通信图和活动图的更多信息，可以参阅 Martin Fowler 的图书[Fowler 03]。与以前一样，[OMG 03a]完整地论述了表示法。

## 6.8 复习题

1. 如图 6-12 所示，X1、X2 和 X3 是什么？(单选题)

- (a) 角色
- (b) Prima donnas
- (c) 参与者
- (d) 棒

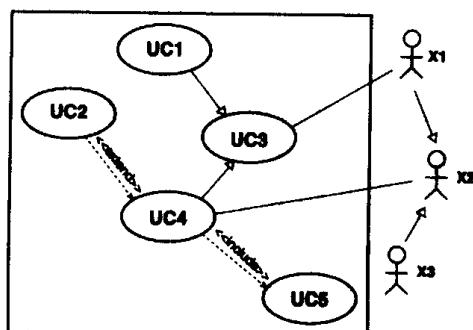


图 6-12 复习题 1、2 和 5

2. 如图 6-12 所示, 下面哪个语句是正确的? (多选题)

- (a) X3 可以使用 UC4 与系统交互。
- (b) X1 可以使用 UC1 和 UC4 与系统交互。
- (c) X3、X1 与 X2 不同。
- (d) UC3 是没有步骤的抽象用例。

### PizzaBase 案例分析

PizzaBase 饭馆想把顾客预订比萨的过程自动化。每张桌子都配备一个触摸式屏幕, 顾客可以用它浏览所供应的比萨, 并点菜。

该饭馆供应两种基本类型的比萨: 自助类只有西红柿酱, 顾客可以选择任意数量的配料, 每种配料的价格都是固定的。预制类有几个小类, 每个小类都有固定的配料。每种比萨都可以预订酥脆型和松软型, 有三种规格: 6 英寸、9 英寸和 12 英寸。

顾客还可以预订饮料, 例如可口类和柠檬类, 每种饮料都有大杯和小杯两种规格。顾客确认了预订的食物后, 就显示总价。之后, 屏幕显示食物的准备和烹饪进度。在顾客吃完后, 可以以方便的方式付费。

3. 在 PizzaBase 案例分析中, 哪些选项是业务用例? (多选题)

- (a) 顾客结账。
- (b) 饭馆准备食物。
- (c) 顾客查看食物的准备进度。
- (d) 顾客选择比萨。
- (e) 顾客在屏幕上选择饮料。

4. 下面的哪个 UML 制品用于显示从系统中取值的步骤? (单选题)

- (a) 用户界面草案
- (b) 术语表
- (c) 状态机图
- (d) 用例
- (e) 类图
- (f) 部署图

5. 如图 6-12 所示, 下面哪个语句是正确的? (多选题)

- (a) UC5 是 UC4 的补充部分。
- (b) UC4 是 UC5 的可选部分。
- (c) UC1 是没有用的。
- (d) UC2 是 UC4 的可选部分。
- (e) UC4 是 UC2 的补充部分。

6. 如图 6-13 所示, A、B 和 C 是什么对象? (单选题)

- (a) A 是实体, B 是控制者, C 是边界
- (b) A 是边界, B 是实体, C 是控制者
- (c) A 是实体, B 是边界, C 是控制者
- (d) A 是控制者, B 是实体, C 是边界
- (e) A 是边界, B 是控制者, C 是实体
- (f) A 是控制者, B 是边界, C 是实体

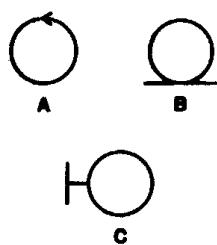


图 6-13 复习题 6

## 6.9 复习题答案

1. 如图 6-12 所示, X1、X2 和 X3 是 (c) 参与者
2. 如图 6-12 所示, 下面的语句都是正确的:
  - (a) X3 可以使用 UC4 与系统交互。
  - (b) X1 可以使用 UC1 和 UC4 与系统交互。
  - (c) X3、X1 与 X2 不同。
  - (d) UC3 是没有步骤的抽象用例。
3. 下面的选项是业务用例:
  - (a) 顾客结账。
  - (b) 饭馆准备食物。
  - (d) 顾客选择比萨。
4. (d) 用例用于显示从系统中取值的步骤。
5. 如图 6-12 所示, 下面的语句是正确的:
  - (a) UC5 是 UC4 的补充部分。
  - (d) UC2 是 UC4 的可选部分。
6. 如图 6-13 所示, (d) A 是控制者, B 是实体, C 是边界

# 第7章 分析问题

本章介绍经典的分析阶段，如何以现代、面向对象的方式完成分析任务。分析是需求和设计之间的桥梁，可以把系统必须提供的服务的清晰陈述转换为对要处理的对象的正确理解。一旦理解了必须处理的对象，就能正确开发出第一流的解决方案。

**学习目标：**

- 理解分析的含义
- 建立静态分析模型
- 理解动态分析如何帮助验证静态模型

## 7.1 引言

分析是找出系统要处理什么的过程，而不是确定如何处理过程。我们需要把一组复杂的需求分解为基本元素和关系，解决方案就建立在这些元素和关系的基础之上。分析是把真实世界建模为对象的第一个机会。

分析模型有静态部分和动态部分。静态分析模型可以使用类图来描述。类图显示了系统要处理的对象和这些对象之间的相互关系。对于动态分析模型，可以使用通信图来证明静态模型是可行的。与以前一样，这里也不涉及 UML 表示法的所有复杂性，只讨论最基本的部分：满足大多数目标的部分。

下面是分析的两个输入：

- 业务需求模型(参见 6.4 节)描述了业务上下文的手工和自动工作流，它使用参与者的面向对象版本、用例、对象、术语表，有时还有通信图和活动图来描述。
- 系统需求模型(参见 6.5 节)包含了系统的外部视图，描述为参与者的面向系统版本、用例和用例图、用户界面草案、增强的术语表和非功能需求。

这些输入都必须转换为由系统处理的对象模型，以及对象的属性和关系。这些对象存在于系统或系统边界上，可以通过一个或多个接口来访问。这个阶段遇到的大多数对象都对应于真实世界中的物理对象或概念(低级的、面向解决方案的对象要在设计阶段才出现)。一旦有了系统对象的模型，就应使它们进入验证过程，验证它们支持解决方案。

## 7.2 为什么要进行分析

为什么要先进行分析？因为分析可防止在彻底理解问题之前设计解决方案。尽管在原则上，可以直接跳到设计阶段，再进入实现阶段，通过试验和更正错误来全面理解问题，但这里介绍的分析技术非常有效。(如果解决方案需要，仍可以使用某种选择好的原型或已证明有效

的概念)。

不可能仅根据业务需求模型就完全理解问题，因为业务需求模型描述了已有的实践：只有添加软件，才可能引入新的实践。另外，手工工作流和自动(或潜在自动)工作流不是分开的：例如，如第 6 章所述，预约汽车型号涉及到人与人和人与计算机的交互操作。即使有了系统用例模型，对问题的理解仍是不全面的，因为用例关注的是外部：用例处理的是参与者和系统边界之间的交互操作——系统本身是一个黑盒子，带有外部才能看到的接口。用例是不严密的：为了使系统更容易开发和理解，用例是用自然语言编写的——因此，它们依赖编写人员对语言的理解和做出的某些假设。当然，业务需求建模和系统需求建模仍必须完成；前者可以用于理解业务上下文，后者能用于与客户达成协议。

完成了静态分析，客户就能确认我们对业务对象的理解是否正确，之后让对象影响我们的设计。在动态分析之后，就可以确信分析对象能够支持需要的系统功能。为了遵循螺旋式开发的原则，动态分析还应有助于建立静态模型。在设计数据库模式(用于需要存储的业务对象)时，静态分析模型也是很有价值的。

### 7.3 分析过程概述

在 Ripple 中，在客户满意之前，分析需要重复经历如下步骤：

1. 使用系统需求模型查找候选的类，以描述与系统相关的对象，并在类图上建立它们。
2. 确定类之间的关系(相关、聚合、复合和继承)。
3. 确定类的属性(对象的已指定的简单特性)。
4. 检查系统用例，确定已有的对象支持它们，在检查过程中微调类、属性和关系，这个用例的实现过程将生成一些操作，来补充属性。
5. 需要时更新术语表和非功能需求——用例本身不需要更新，但可能需要某些更正。

术语“实现”意味着使之变成真的。在用例的实现过程中生成的操作应在设计过程中删除——在设计阶段，应建立我们的自信，而不是设计解决方案。

需要给客户展示类图和属性，让他们找出错误(正确理解业务的客户可能对类的理解比我们更好)。小组成员应汇总客户在类图上找出的信息。类图对非程序员来说是非常容易理解的，所以这些信息可以变成注释，例如“刚才您说非会员不能预约汽车？不是这样的，如果他们付了押金，就可以预约”。确定何时把类图展示给客户取决于开发小组，但一般至少要给客户展示两次：一次是让客户找出错误，一次是向客户说明已更正了错误。

一般情况下，不要给客户展示对象操作或通信图，因为：

- 它们会大大增加复杂性
- 对于非程序员的客户来说，它们是肤浅的，因为前面已经用系统用例演示了动态行为。
- 它们隐含着编码，对于非程序员来说，这是肯定要避讳的。
- 它们在设计之前是要删除的。

一些客户，例如技术经理，喜欢看一些动态分析，以增强自信心：这很好，但应在单独的技术会议上给他们展示。

本章的其余部分详细讨论静态和动态分析。

## 7.4 静态分析

静态建模设计确定系统的逻辑或物理部分，以及如何把它们连接在一起。也就是说，它描述了如何构建和初始化系统。

### 7.4.1 确定类

在本书的前几章中，类没有系统地标识出来。因为现在要检查业务需求建模和系统需求建模的过程，以系统用例的形式获得好的候选类。

候选类常常在用例中用名词(noun)来表示。只要稍微实践一下，就可以快速删除表示下述含义的名词：

- 系统本身，例如“系统”或“iCoot”：前面说过，系统只是开发过程的一个边界。
- 参与者，例如助手或主任办公室：它的一个例外是需要在内部存储参与者的信息(例如对于会员，就需要存储一个密码)。在大多数情况下，参与者是边界的匿名驱动力。
- 边界，例如“顾客小程序”或“主任办公室链接”：在这个阶段，要用有趣的信息和行为标识出与业务相关的对象。边界是允许参与者获得对象的特定软件。
- 小类型(例如字符串或数字)：可以假定这些由实现语言或它的库提供。

在这个过滤过程之后剩下的候选类的简短描述应添加到术语表中。如果不能给类写出简短描述，就可能希望它表示更多的内容，此时应把它分解为多个类。

### 7.4.2 标识类的关系

一旦有了候选类的列表，就可以绘制出它们之间的关系。有四种类型的关系：

- 继承：子类继承了超类的所有属性和行为。
- 关联：一种类型的对象与另一种类型的对象关联。
- 聚合：强关联——一个类的实例由另一个类的实例构成。
- 复合：强聚合——复合的对象不能由其他对象共享，且与构成它的对象一起消亡。

继承与其他三种关系不同：继承描述了类在编译期间的关系，而其他三种关系描述了对象在运行期间的连接关系。根据 UML 标准，所有运行期间的关系都可以使用统一的术语“关联(association)”。但是，大多数人都把“关联”看做“不是聚合或复合的关联关系”。

在关系之间选择是很困难的——需要使用直觉、经验和推测。在分析过程中，这些关系的出现频率如下：

关联 > 聚合 > 继承 > 复合

就设计和实现而言，关联、聚合和复合之间的区别很难界定。

### 7.4.3 绘制类图和对象图

类图显示了存在哪些类，这些类有什么关系(正式的类图还可以显示属性和操作，但这需要更多的空间)。对于聚合、复合和关联，类图显示了允许的运行时关系，而不是显示实际的运行时关系。

图 7-1 显示了 iCoot 的 UML 类图。每个类都表示为带有类名的方框(如果不是手工绘制，

类名应显示为粗体)。如果类是抽象的，其名称就显示为斜体。如果手工标记抽象类，就可以在类名的上面或左边加上关键字{abstract}，而不是使用斜体。

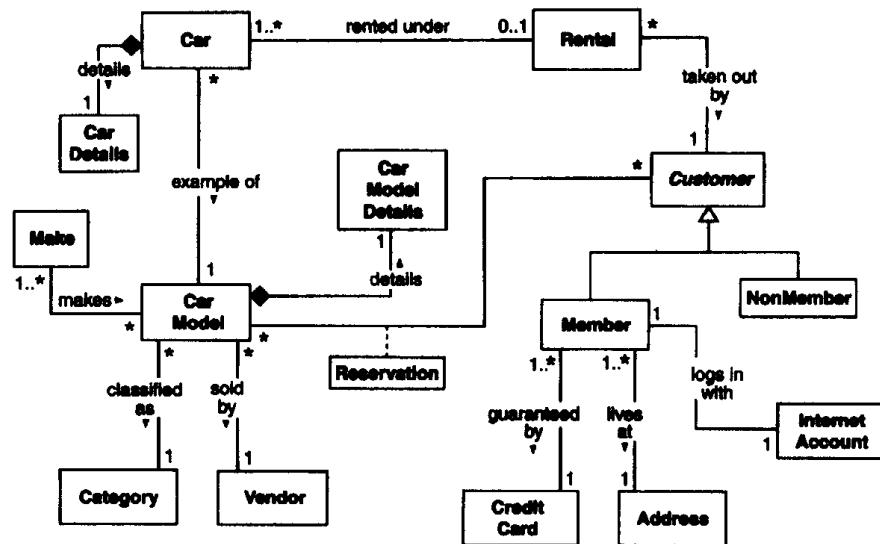


图 7-1 iCoot 的分析类图

类之间的关系显示为带各种注释的线条。即使不了解 UML 的相关知识，仅通过文本也很容易看出类图中的信息。例如，可以从图中看出“Car 可以在 Rental 下出租”，“Rental 可以由 Customer 取出”，等。

尽管类图中的关系通常在类之间绘制，但运行时的关系实际上存在于对象之间：例如，根据图 7-1，Car 的实例应在运行时连接 Rental 的实例。UML 允许绘制运行时的对象和编译时的对象，如图 7-2 所示。尽管 UML 允许在同一张图中混合类和对象，但如果图中没有对象，人们一般使用术语“类图”，如果图中没有类，则称为对象图(图中有类和对象时，可以称之为类图或对象图)。

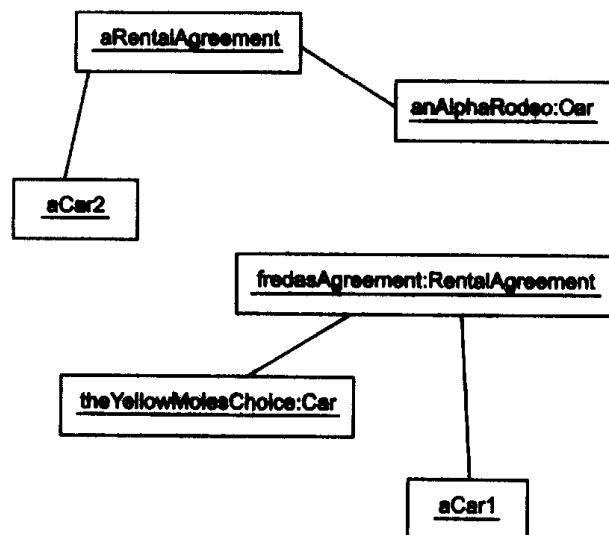


图 7-2 用 UML 描述对象

在对象图中，对象显示为通过链接来连接的方框——该链接是“已实现”的关联。所以很容易看出类和对象之间的区别：对象标签是有下划线的。除了对象的名称之外，标签还包含一个冒号和对象的类，例如 fredasRental:Rental。可以显示对象的名称或对象类，或者同时显示两者。如果只显示对象的类，就必须包含冒号，以区分类名和对象名，例如:Rental。

图 7-2 显示了两个 Rental 对象：在第一个 aRentalAgreement 对象中，租用了 aCar2 和 anAlphaRodeo。在第二个 fredasRentalAgreement 对象中，租用了 theYellowMolesChoice 和 aCar1。可以看出，对象名对应于程序中使用的各个变量名称。

对象图对演示某个运行期间的情况很有用，但它们是可选的。为了简洁，下面不再把类和对象放在同一张图中。

#### 7.4.4 绘制关系

图 7-3 显示了如何在类图中描述继承：实线上的白箭头从子类指向超类。为了强调子类的层次结构，箭头可以以左边的样式合并。因此，SportsCar 和 Saloon 都是 Car 的子类。

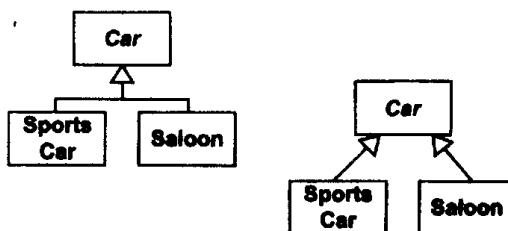


图 7-3 用 UML 描述继承

两个类之间的聚合关系表示为聚合端带有一白菱形的线条。所以图 7-4 显示，Engine 是 Car 的一部分。

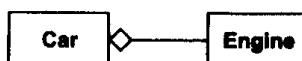


图 7-4 用 UML 描述聚合

复合的表示方式与聚合类似，但复合端是一个黑菱形。图 7-5 显示，Body 总是 Car 的一部分。



图 7-5 用 UML 描述复合

关联显示为未修饰的线条(如图 7-6 所示)。因此，Driver 与 Car 关联，但 Driver 不是 Car 的一部分(这可能是聚合)，Driver 不总是一个 Car 的一部分(这可能是复合)。

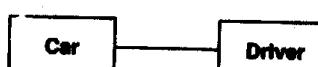


图 7-6 用 UML 描述关联

在开发分析类模型时，为了简单起见，应确保信息只能用一种方式推断出。例如，图 7-1 中的 iCoot 类图可以从 Rental 类的 rented under 关联和 taken out by 关联中计算出某个会员租用了多少辆汽车。因此，显示从 Member 类到 Car 类的 hasOutForRent 关联是多余的，但用例模型隐含着，这个关联是需要的。

### 1. 多重性

除了继承之外的其他关系都在两端表示了允许参与关系的运行时对象的数量(关系中的多重性)：

- n: 表示 n
- m..n: m 到 n 范围内的任意数值(包含 m 和 n)
- p..\*: 从 p 到无穷大的任意数值。
- \*: 0..\*的缩写
- 0..1: 可选。

对于复合，复合端的数字总是 1，因为根据 UML 规则，复合的对象不能在复合体中共享，因此，在这种情况下，数字 1 就是多余的。在其他情况下，如果没有显示数字，就必须假定没有指定它，或者在这个阶段，该数字是未知的。假定某个遗漏的数字隐含着某个默认值(如 1)是错误的。

在图 7-7 中，可以推断出：

- Car 有一个 Engine。
- Engine 是 Car 的一部分。
- Car 有 4 或 5 个 Wheel。
- 每个 Wheel 都是 Car 的一部分。
- Car 总是由一个 Body 组成。
- Body 总是 Car 的一部分，它随着 Car 一起消亡。
- Car 可以有任何多个 Driver。
- Driver 至少可以驾驶一辆 Car。
- Car 在某一时刻至多有 7 位 Passenger。
- 一位 Passenger 在某一时刻只能在一辆 Car 上。

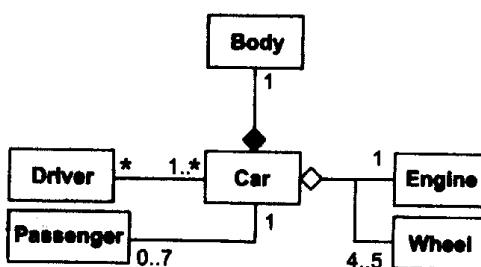


图 7-7 用 UML 描述多重性

即使前面介绍的表示法都很明确，事情也没有那么简单，因为我们仍在使用自然语言和读者对真实世界的假定。例如，Car 和 Passenger 之间的关联可以看做如下陈述，而不是上面列表

中的陈述：

- 对于每辆 Car，至多只能承载 7 位 Passenger。
- 一位 Passenger 只能在某辆特定的 Car 上。

根据建模人员的说法，似乎最初的解释比较明确(我们常常说一辆车可以承载多少位乘客，何时会描述为谁可以乘坐某辆车？)。如果区别不是很清楚，或给它提供比较书面的说明，就可以使用详细的描述——类调查。或者可以给图加上注释，来进一步说明。

聚合和复合之间的区别是很微妙的。在图 7-7 中，为什么 Engine 是聚合，而 Body 是复合？这个区别与对象共享和对象生存期有关。复合的对象永远都不能是多个复合体的一部分，它会随着复合体的消亡而消亡；而聚合的对象可以共享，也可以比其聚合体更耐用。汽车在出厂时肯定有一个全新的引擎，但引擎可能在以后因磨损而更换，所以引擎不一定与汽车一起报废。相反，汽车的车身是汽车的内在组成部分，它是汽车的灵魂，如果车身损毁了，汽车就报废了(但总是可以先卸下引擎)。共享的问题在这个例子中不重要：车身肯定不会是两辆汽车的一部分(这不合理)，引擎就不一定了。

这似乎有点混乱。UML 中存在复合只是因为编程语言，如 C++，中有一些特性。在 C++ 中，一个对象可以是另一个对象占用的内存的一部分：此时，子对象肯定随着较大对象的消亡而消亡。这个语言特性还引出了规则的第二部分：复合的对象不能同时是两个对象的一部分。即使复合的对象是独立的，由于 C++ 中没有垃圾回收器，复合体本身被删除时，就要删除复合的对象，这会强制共享。如果 UML 也需要复合关系来创建复合的对象(这适合于车身例子)，就更好了。但是，UML 禁止先创建复合的对象，再把对象传送给复合的构造函数。

复合其实是一个编程问题，它在设计阶段比在分析阶段更常见。就分析样式来看，当有明显的部分-整体关系时，最好使用聚合(如引擎例子)，复合应用于有明显的共享生存期的情形(例如车身例子)，只是这种情形非常少见。

从设计的角度来看，要在内部隐藏委托对象，而不是从另一个类中继承，以便给对象添加行为时(其例子如第 3 章的 Stack 和 LinkedList)，复合也是很有用的。

从实现的角度来看，一对多和多对多关系常常需要在运行期间使用集合对象(列表、树、集等)。例如，Car 可以使用某种 List 对象来处理其乘客。在类图中使用多重性的一个优点是，可以在以后指定这些杂乱的实现细节。

## 2. 关联标签、角色和注释

除了继承之外的所有关系都可以给定一个关联标签，它表示关联的性质。如果关联名称的理解方式不明显，就可以使用一个黑色的箭头。例如在图 7-8 中，至少有一个 Wheel 推动 Car 前进。

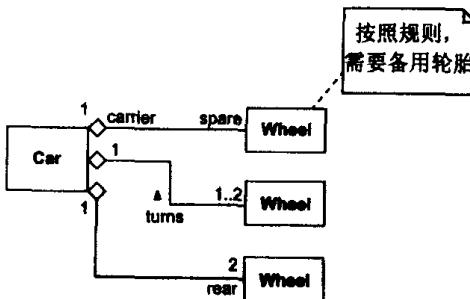


图 7-8 UML 中的关联标签、角色和注释

除了关联名称之外，还显示了角色。角色表示对象在关联中承担的那一部分——角色显示为承担该角色的对象旁边的一个标签。例如，图 7-8 显示了如下角色：

- Car 有一个 Wheel 用作 spare。
- spare Wheel 把一辆 Car 作为其 carrier。
- Car 有两个 rear wheel。

在原则上，关联名称和角色可以在同一个关联上合并，但大多数情况下，它们应是二选一(以避免混乱)。

图 7-8 还显示了 UML 注释，UML 注释是在看起来像一张纸的图标中包含的任意文本，该图标与图中的相关部分通过一条虚线连接。如果虚线的目标不清楚，就可以在其目标端放置一个小白圆圈和黑色的边框——当目标是另一条虚线时，这是很有用的。注释可以显示在任意图中，用于提供使用其他 UML 表示法很难显示或显示起来比较混乱的额外信息。

#### 7.4.5 属性

属性是对象的一个特性，例如对象的大小、位置、名称、价格、字体、利率等。在 UML 中，每个属性都可以指定一个类型，可以是类或原型。如果选择指定类型，该类型就应显示在属性名称右面的冒号之后(也可以在分析阶段不指定属性类型，因为类型很明显，或者因为还不想提交)。

在类名的下面添加一个分隔线，就可以在类图中显示属性。为了节省空间，可以把它们单独保存在属性列表中，并加上描述。如果使用软件开发工具，就可以放大，以显示属性(及其描述)，或者缩小，只显示类名。如果不能在这个阶段为属性提供简短的描述，该属性就应分拆为几个属性，甚或自成一类。

图 7-9 显示了 Engine 的属性：capacity、horsePower、manufacturer、numberOfCylinders 和 fuelInjection。此图给 manufacturer 指定了 String 类型，给 fuelInjection 指定了 boolean 属性。也就是说，我们对 manufacturer 的其他信息，如 address，没有兴趣，对 fuelInjection 的变体没有兴趣(只想知道它是不是有)。

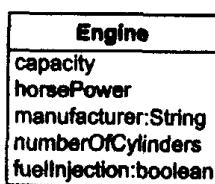


图 7-9 用 UML 描述属性

一开始显示属性类型，就会遇到许多问题：String 是什么？boolean 是什么？如果类型是一个类的名称，就不会有问题。本书不针对某种特定的编程语言或库。所以，最好使用常见的原型(例如 int、boolean 和 float)和一两个明确的类(如 String 表示包含一系列字符的对象)。

UML 允许用独立于语言的表示法定义自己的原型，例如 integer、Real 和 Boolean，但应避免使用这个功能，因为在开始设计时，就必须考虑与特定语言相关的内容(另一个原因是在 Java 中，像 integer 这样的类型是类，不是原型)。

还应避免使用数组类型，尽管大多数面向对象的语言都支持数组，但它常常是对象和原型

的交叉。其原因是，如果使用集合类，如 List 和 Set，类可能更好理解。在设计阶段，使用数组可能比较多，但仍需要小心，不要因改进性能而牺牲好的样式。

为了简单起见，应避免在制品中包含派生的属性。例如，圆的属性包括半径、直径、周长和面积。但是，只要存储其中的一个属性，就可以在运行期间计算出其余属性。所以只需在类图中存储四个属性中的一个。在这种情况下，半径似乎是明显的选择，因为它比其他属性的访问次数多(所以也不计算它)，其他属性可以使用乘法计算出来(比除法快)。

就 UML 而言，属性和关联(所有三个变体)都只是类的属性。换言之，每个属性都可以显示为属性，或者显示为把属性名作为角色的关联(但与原型值或数组的关联看起来会很古怪)。这意味着，可以在类型名的后面给属性增加多重性，例如\*表示对于多值属性，[0..1]表示可选属性。这是 UML 避免在某种情况下是显示属性还是关联的棘手问题的方式。在本书中，不给属性显示多重性，除非是可选属性。

图 7-10 显示了在检查 iCoot 系统用例时找出的分析对象的全部属性。为了完整，所显示的一些属性来自于完整的 Coot 系统(例如 totalAmount)。为了避免在系统中处理图像和视频，广告和海报是指定其存储位置的属性(例如使用 URL)。

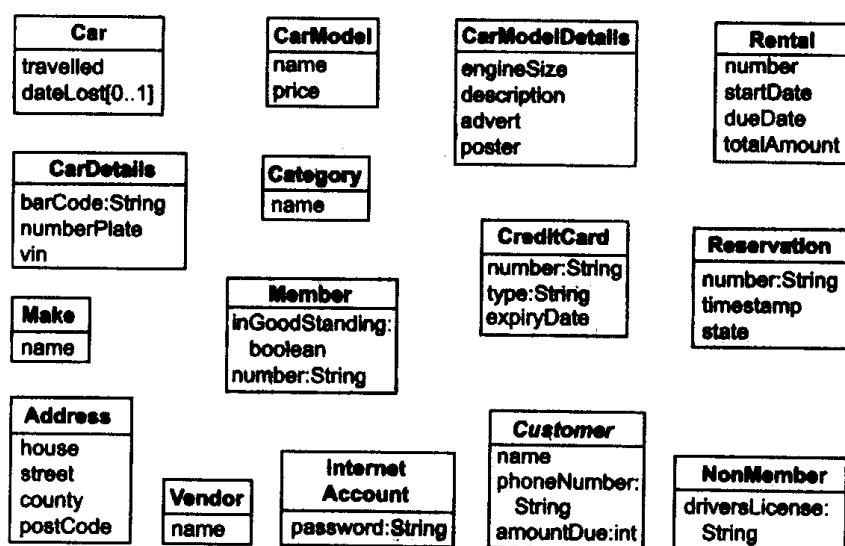
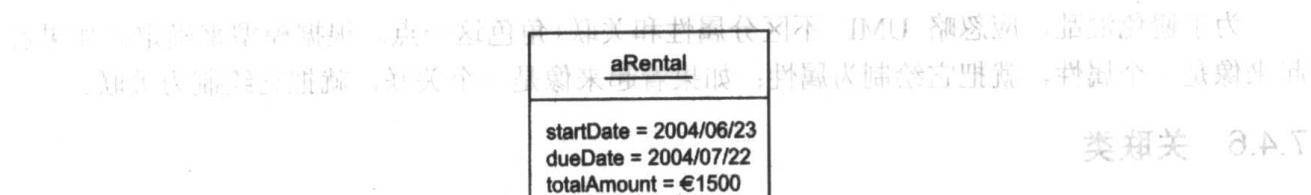


图 7-10 iCoot 的属性

dateLost 属性是可选的(由[0..1]表示)：如果 Car 丢了，就记录其丢失日期，否则什么也不记录。在编程术语中，可以使用 null 指针表示某个 Car 没有丢。如果可选属性有原型类型，例如 int，就必须保存一个值，表示“此处没有值”，例如，模型允许设置 0 或 -1。

属性的多重性偶尔也是有用的，但不要过多使用它们。例如，本书只有一个属性(dateLost)需要多重性(从长远来看，使用“汽车状态”会更好)。

如前所述，UML 允许绘制运行时对象和编译时类。图 7-11 显示了如何在对象图上指定运行时属性的值。



选择属性还是关系  
我们常常需要在为信息建模的几种方法中选择。例如，从顾客的角度来看，如何为 Car 的颜色建模？图 7-12 给出了四种方法：



图 7-12 在属性和关系之间选择

(1) 在 Car 和 color 类之间引入聚合。

(2) 给 Car 添加一个属性 color，其类型是 color。

(3) 给类的每种颜色引入一个 Car 的子类。

(4) 在 Car 和 color 之间引入复合。

这些选项都可行，只是其中一些显得有点违背常理。该选择哪个选项？

中心议题是：哪个建模选项最适合当前的情况？换言之，哪个选项最自然？就选项 1 而言，说“color 是 Car 的一部分”显得有点笨拙。选项 2 似乎比较好：就汽车买主而言，颜色只是汽车的一个属性。选项 3 肯定有点过头：给每种颜色的汽车都设置一种新类型，而汽车的颜色可能有数十种。选项 4 似乎比选项 1 好一些：汽车出厂时都会喷涂一种颜色，即使以后改变其颜色，原来的颜色也可能保留在新颜色的下面。总之，选项 2 在购买汽车时是最合适的。

但如果从汽车厂家的角度给汽车建模，选项会不同吗？在这种情况下，颜料厂家可能比较重要——如果颜料用光了，我们需要知道到哪里可购买到颜料。所以需要把 color 建立为一个单独的类，且有自己的关系和属性；因此，选项 4 是最好的选择。

可能选择选项 3 吗？可能。例如，心理学家要了解汽车的颜色对司机行为的影响——也许红色的汽车会激发危险的驾驶行为，而绿色的汽车可使司机谨慎驾驶。在这种情况下，红色和绿色的汽车完全不同，应给它们建立为单独的类。

这个例子的寓意是，分析员必须选择最适合当前情况的表达方式，这是没有固定答案的。不要过多地考虑哲学体系，而应利用常识、经验、直觉，反复斟酌，找出成功的实现方式。

为了避免混乱，应忽略 UML 不区分属性和关联+角色这一点。根据模型来确定：如果看起来像是一个属性，就把它绘制为属性；如果看起来像是一个关联，就把它绘制为关联。

#### 7.4.6 关联类

关联偶尔也有与它相关的信息或行为。关联类可以和关联一起引入，如图 7-13 所示。这个图表示，一个 CarModel 可以与任意多个 Customer 对象关联，一个 Customer 也可以与任意多个 CarModel 对象关联。对于每个链接，都有一个对应的 Reservation 对象，它包含号码、时间戳和状态。在本例中没有给关联指定名称，因为它隐含在关联类的名称中。

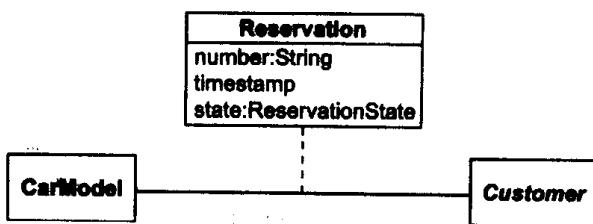


图 7-13 iCost 的关联类

关联类表示的属性和操作仅因为关联类存在而存在：属性和操作与关联两端的对象都无关。在上面的例子中，当顾客进行预约时，就在运行期间在 Customer 和相应的 CarModel 之间建立一个新链接。在需求捕捉和分析阶段，必须记录下预约号、时间戳和状态。但是，这些属性对 Customer 和 CarModel 都没有什么意义，它们位于这两个对象之间。所以，使用关联类比较合适。

在设计时，必须用更具体的类替换关联类，因为大多数编程语言都不直接支持关联类。但是，它们在分析过程中非常有用。

#### 7.4.7 有形对象和无形对象

我们常常为无形(intangible)对象建模，例如目录中描述的产品。也常常为有形(tangible)对象建模，例如送到门口的实际产品。目录中的对象描述了可以从提供商处预订的产品的属性，但该产品不一定已生产出来了。送到门口的对象肯定已生产出来，它是目录中描述的产品类型的一个实例。一般，每种无形对象有许多有形对象。

把有形产品和无形产品建立为一个对象是一个常见的错误。例如，如果为汽车经销商编写一个销售系统，就会发现在分析过程中，我们处理的是描述可销售汽车的“目录表”、卖给顾客的“汽车”和购买汽车的“顾客”。很容易得出结论：应创建如图 7-14 所示的三个具体类。但实际上，这里有两个“汽车”概念：目录表中的汽车是无形的，它描述了该类型的所有汽车的特性，但这种汽车可能还不存在；而顾客拥有的汽车是有形的，它肯定存在，因为它可以驾驶，与另一个顾客拥有的同类型汽车是不同的。

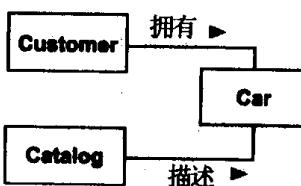


图 7-14 购买汽车

## 1. 错误的建模

为了强调有形性问题，假定除了销售汽车之外，经销商还给顾客提供服务。与销售相关的信息包括：

- modelNumber：表示制造这类汽车的过程。
- availableColors：这类汽车在出厂前可以喷涂的颜色。
- numberOfCylinders：这类汽车的引擎拥有的气缸数。

与服务相关的信息包括：

- owner：汽车的注册主人。
- vehicleIdentificationNumber：制造汽车时在车身上固定的一个小板子上刻上的惟一数字，表示汽车的注册目的，帮助警察找出被盗汽车的主人。
- mileageAtLastService：汽车在上次接受服务时已行驶的英里数——通过它可以计算出汽车在上次接受服务以后又行驶了多少英里。

使用如图 7-14 所示的 Car 的概念，只能把这些属性都放在一个类上，如图 7-15 所示。从已知的对象建模知识来看，应避免使一个类有两组完全不同的任务——这种类的内聚力很脆弱，它们的任务也不会构成一个块。

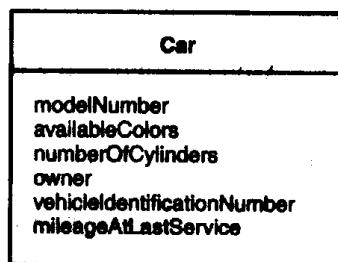


图 7-15 显示其属性的 Car 类

假定要销售 Alpha Rodeo 156 2.0 型汽车，就必须创建一个 Car 类，并设置其相应的属性，这会得到如图 7-16 所示的结果(可能的属性值显示为花括号中的一个列表——这不是严格的 UML，但很方便)。

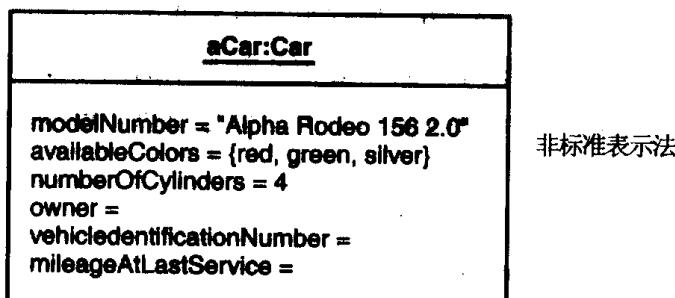


图 7-16 用于销售的汽车

现在假定顾客开来了 Alpha Rodeo 156 2.0 型汽车，接受第一次服务。此时有两个选择：可以创建一个新的 Car，来表示这个顾客拥有的汽车，如图 7-17A 所示；也可以使用已有的 Car，得到如图 7-17B 的结果。

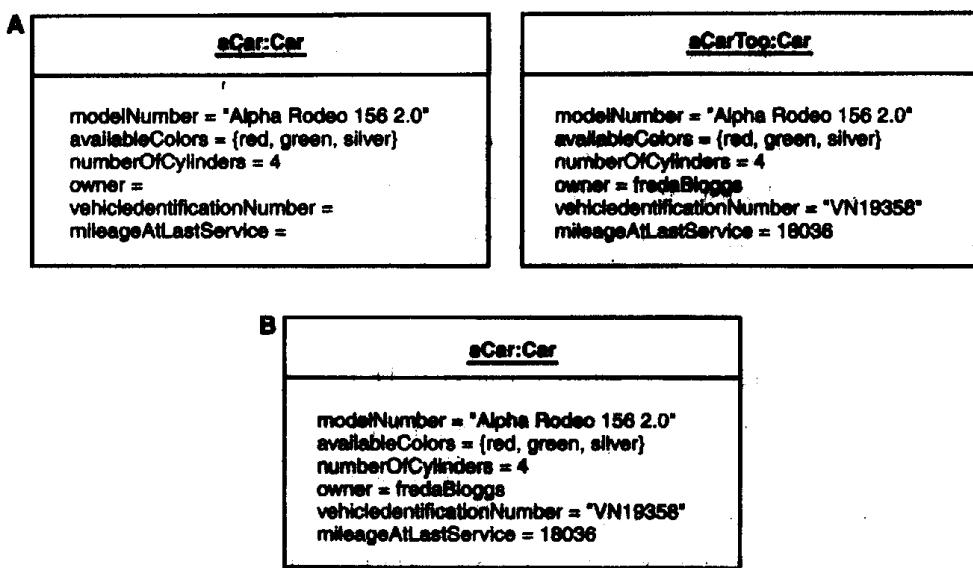


图 7-17 给汽车提供服务

如果选择选项 A，第一个 Car 对象上的一半属性都没有用，第二个对象上也有冗余信息。如果选择选项 B，每次就只能给一辆“Alpha Rodeo 156 2.0”型汽车提供服务(否则先得到服务的第一个对象的信息就会丢失)。

如图 7-14 所示的原始模型很自然、很合理，但实际上它是没有意义的。前面已指出，我们有一个无形概念，负责第一组属性，还有一个有形概念，负责第二组属性。

## 2. 正确的建模

舍弃图 7-14 中的模型，用一个新的无形概念 CarModel 和一个有形概念 Car 来代替，得到如图 7-18 所示的类图。现在可以把属性 modelNumber、availableColors 和 numberOfCylinders 放在 CarModel 上，把属性 owner、vehicleIdentificationNumber 和 mileageAtLastService 放在 Car 上。把这个新模型应用于前面的例子，就会创建如图 7-19 所示的运行时对象。这里有一个 CarModel，它表示“Alpha Rodeo 156 2.0”型汽车，以及两个 Car 对象，它们表示这类汽车要接受服务的独立实例。有了这个新模型，就不必担心某类型的汽车销售了多少辆、有多少辆汽车返回接受服务、有多少辆汽车同时接受服务，因为型号可以按照逻辑，简明地处理所有的可能性。

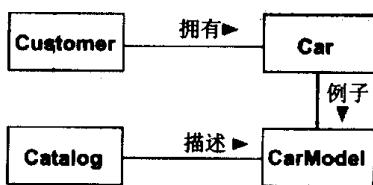


图 7-18 有形汽车和无形汽车模型

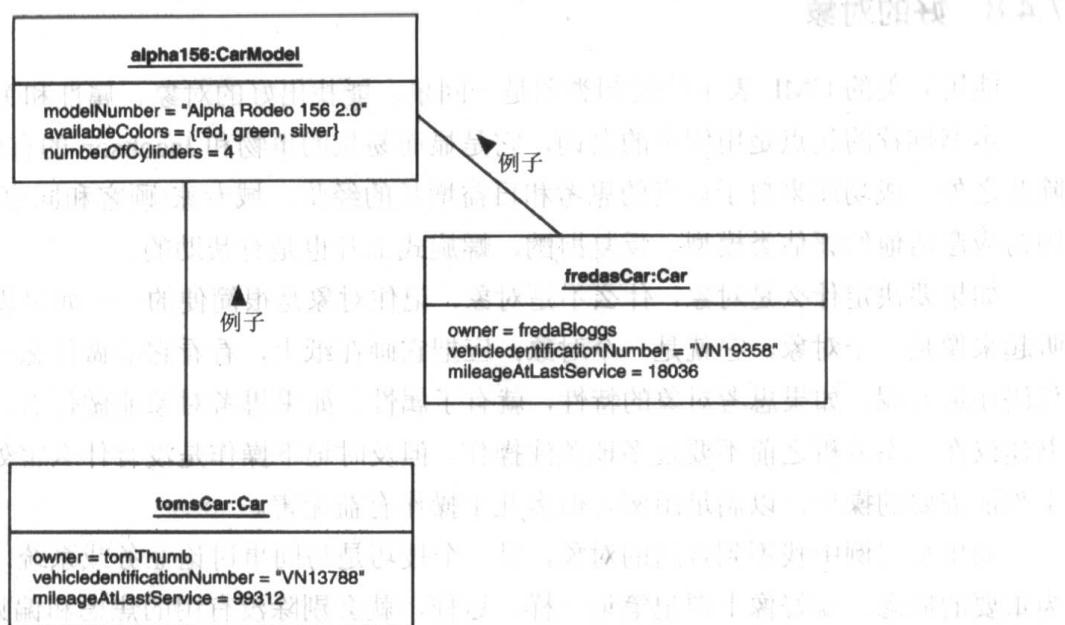


图 7-19 用于销售的汽车模型和已销售出去的汽车

一般情况下，一个无形对象可以产生许多有形对象。另外，无形对象的属性是固定的，而有形对象的属性是随时间变化的。在前面的例子中，有一个 CarModel，它表示已销售出去的、任意数量的“Alpha Rodeo 156 2.0”型号的 Car 对象。CarModel 的属性不会随时间而变化(厂家偶尔可能改变规格，例如添加新的颜色，但这不是很频繁)。该模型显示了两个 Car 对象，它们分别表示一辆其主人至少有一次返回接受服务的“Alpha Redeo 156 2.0”型汽车。当汽车卖出时，其主人就改变了，每次接受服务时，mileageAtLastService 都会改变。vehicleIdentificationNumber 不会改变，但这是身份属性的特殊情况，在该对象的生命周期，这种属性把这个对象与其他类似对象区分开来。

#### 练习 4

在结束有形性的主题之前，考虑一个视频出租系统。图 7-20 中的哪两个类图是正确的？

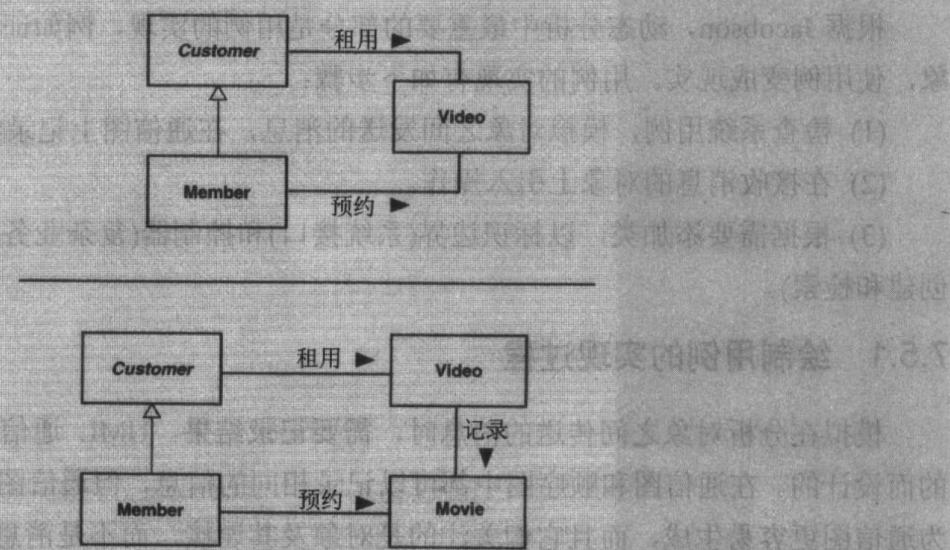


图 7-20 视频出租系统的有形和无形对象

#### 7.4.8 好的对象

能用完美的 UML 表示法绘制类图是一回事，能找出好的对象、属性和关系是另一回事。

本书推荐的起点是用例中的名词，它是显而易见的事物和 Jacobson 的有价值工作的组合。除此之外，成功还来自于认真的思考和日益增长的经验。域专家(顾客和同事)也能提供帮助，因为应邀请他们评估类模型。反复斟酌，螺旋式上升也是有帮助的。

如果要决定什么是对象，什么不是对象，记住对象是很简便的——如果思考或讨论的东西听起来像是一个对象，它就是一个对象：应把它画在纸上，看看它能做什么——此时，离编写代码还远着呢。如果思考对象的特性，就有了属性。如果思考对象能做什么，就有了操作。本书建议在动态分析之前不要过多地关注操作，但及时记下操作是没有什么害处的。动态分析用于验证需要的操作，以满足用例，但多几个操作有益无害。

如果从用例中找不到合适的对象，另一个技巧是与同事讨论业务或系统。让他们记下您认为重要的概念，就好像上课记笔记一样。这样，就会剔除没有用的焦虑和偏见。

前面介绍了如何从属性和关系中选择，考虑有形和无形对象，标识关联类，这些都有助于避免常见的错误。

## 7.5 动态分析

进行动态分析有如下原因：

- 确认类图是完整、正确的，以便尽早更正错误：这包括添加、删除或修改类、关系、属性和操作。
- 相信当前的模型可以在软件中实现：在继续之前，我们自己不是惟一有自信的，客户也同样重要。
- 验证最终系统上用户界面的功能：在进行详细的设计之前，最好按照用例中的线条，把对系统的访问放在各个界面上。

根据 Jacobson，动态分析中最重要的部分是用例的实现，例如证明用例可以实现为协作对象，使用例变成现实。用例的实现有如下步骤：

- (1) 检查系统用例，模拟对象之间发送的消息，在通信图上记录结果。
- (2) 在接收消息的对象上引入操作。
- (3) 根据需要添加类，以标识边界(系统接口)和控制器(复杂业务过程的占位符或者对象的创建和检索)。

### 7.5.1 绘制用例的实现过程

模拟在分析对象之间传送的消息时，需要记录结果。UML 通信图和顺序图就是为这个目的而设计的。在通信图和顺序图中都可以记录相同的信息，但通信图更适合于用例的实现，因为通信图更容易生成，而且它们关注的是对象及其连接，而不是消息传送的顺序。

6.4.4 节介绍了一个简单的、业务级的通信图。在业务需求例子中，参与者、对象和系统采用相当自由的形式混合在一起，而分析例子(图 7-21)比较清晰。如图 7-21 所示，在 iCoot 小组的成员看来，该图信息的非正式描述为：

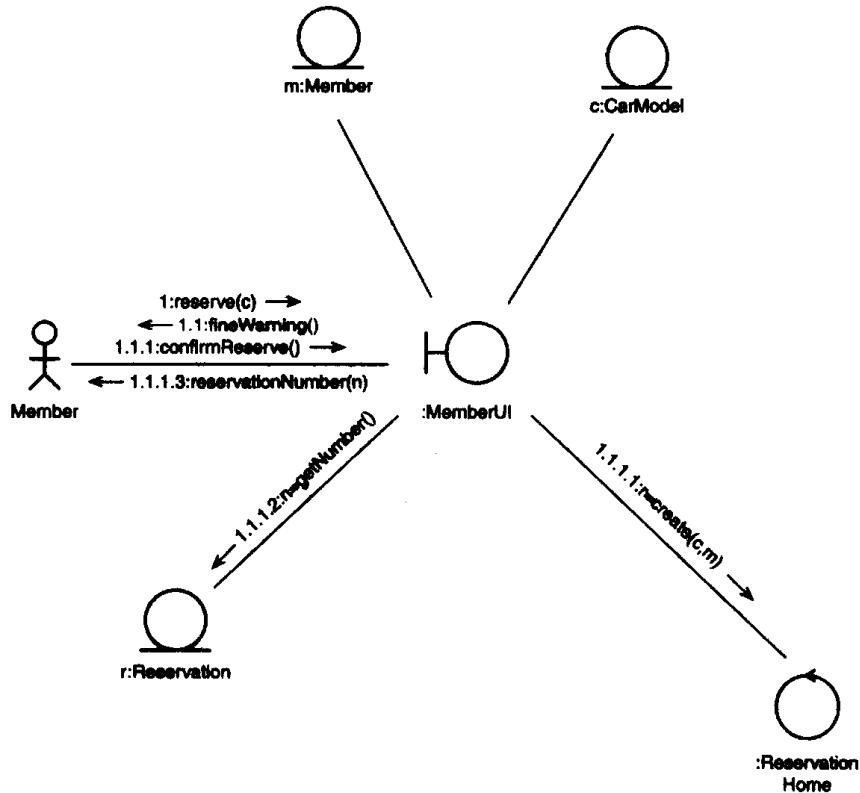


图 7-21 U7:进行预约的通信图

Member 参与者向 MemberUI 预约某个 CarModel，Member 被警告说，如果有该型号的汽车，但 Member 没有来取车，就要被罚款。一旦 Member 确认要进行预约，MemberUI 就让 ReservationHome 创建一个新的 Reservation，并传递 CarModel 和 Member(因为 Member 已登录，所以有用户界面)；最后，MemberUI 从新的 Reservation 中获得一个号码，并把它传送给 Member。

一般，分析级的通信图可以显示：

- 与边界交互的参与者(例如，Member 与 MemberUI 交互)。
- 与系统内部的对象交互的边界(例如，MemberUI 与 ReservationHome、Member、CarModel 和 Reservation 交互)。
- 系统内部的对象与外部系统的边界交互(例如，内部的 ReportGenerator 对象与 HeadOffice 边界交互)。

不需要显示位于系统外部的业务对象，也不需要显示不直接与系统交互的参与者。

这里可以不使用双向交互，而使用更面向计算机的客户—提供商模式：参与者启动与边界对象的交互；边界对象启动与系统对象的交互；系统对象启动与其他系统对象、其他系统边界的交互。

## 7.5.2 边界、控制器和实体

清晰的通信图把对象显示为带标签的方框。为了表达额外的信息，UML 允许开发人员使用图标来代替方框，表示对象的特性。图 7-22 显示了图 7-21 中图标的 UML 含义。

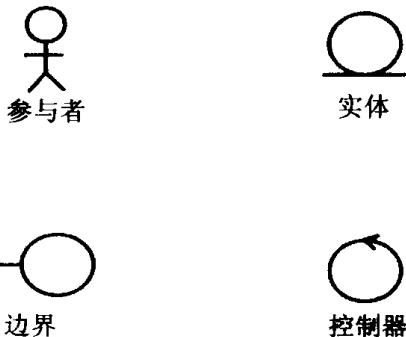


图 7-22 通信图的 Jacobson 图标[Jacobson 等, 92]

- 参与者：存在于系统外部的人(通常)或系统(偶尔)。
- 边界：位于系统边缘上的对象，在系统和参与者之间。对于系统参与者，边界提供了通信路径。对于作为参与者的人，边界表示用户界面，以执行命令和查询，显示反馈和结果。每个边界对象通常都对应于一个用例或一组相关的用例。更准确地说，这种边界通常映射为一个用户界面草案(此时，它可以是整个界面，也可以是一个子窗口)。边界对象在设计过程结束后仍旧存在。
- 实体：系统内部的一个对象，表示一个业务概念，例如顾客、汽车或汽车型号，包含有用的信息。一般实体是由边界和控制器对象操纵的，而不是有自己的行为。实体类出现在分析类图中。大多数实体在设计过程结束后仍旧存在。
- 控制器：封装了复杂或凌乱过程的系统内部对象。控制器是一个服务对象，提供下述服务：控制系统过程的全部或部分、创建新实体、检索已有的实体。没有控制器，实体就会充满混乱的细节。控制器只是为了便于分析，所以许多控制器在设计过程结束后就不存在了。一个重要的例外是“家(home)”的概念。家是用于创建新实体、检索已有实体的控制器。家还可以包含实体消息，例如 carModelHome.findEngineSizes()。家是这么清晰的一个概念，所以常常在设计过程结束后仍旧存在。

RUP 方法是为设计阶段保留所有的控制器，以及在动态分析过程中找出所有的操作。在 RUP 中，分析模型和设计模型没有区别——我们只是从分析模型开始，一次次地丰富它，直到把它转换为可以实现的设计模型为止。对于本书，这种方法过于乐观。

从分析中得到的有价值的结果有：

- 好的实体对象和已验证的属性
- 反映用例的高级边界对象
- 模型正确的自信
- 家(忽略所有的实体消息)。

为了便于实现，设计人员不应有机会修改这些基本结果。其缺陷是分析人员不应考虑编程细节，例如如何实现对象的属性、关系或操作。

建议设计人员从新的类图开始，类图中应有在分析过程中找出的实体对象。然后把选中的边界和家添加到这个图中。

### 7.5.3 通信图中的元素

图 7-23 再次显示了分析通信图，其中的注释解释了各个表示法。表示法的细节如下所示：

- 参与者的显示方式与用例图相同。
- 对象显示为带标签的图标或带标签的方框。
- 两个对象之间的线条表示链接，与对象图相同。
- 消息显示为顺序号(表示消息在通信中的位置)、消息名称(采用通常的格式)和参数列表(括号中)。
- 开放端的箭头显示消息发送的方向(这个箭头不必位于消息的端部，一些开发人员把它放在下面)。
- 标签，用于表示对象和参数，可以显示为 name、name>Type、:Type 或字面量(例如，carModel, m:CarModel, :CarModel, 10 或“abc”)。
- 把回应值赋予一个名字可以显示为 n=getNumber()。
- 条件消息可以显示为消息旁边的防护(guard，括号中的条件)，例如 4:[Only on Saturday]-readPaper()。
- 迭代可以显示为顺序号旁边的\*。

尽管实际上，参与者不会给边界对象“发送消息”，但消息这个隐喻是表示交互的一种方便、简洁的方式。与对象图不同，通信图上的对象标签不加下划线(其部分原因是类不能显示出来，以避免混乱)。

下面看看顺序号。最初消息是发送给对象的，以执行一个方法。在这个方法中，可以发送更多的消息。这些消息都要给定顺序号，显示交互的级别。例如，消息 1 会执行方法 1，它第一个消息是 1.1，第二个消息是 1.2，等。方法 1.2 中发送的第一个消息是 1.2.1。如图 7-23 所示，通信图显示一个顶级消息(reserve)的实现时，顺序中的第一个数字不会高于 1。这种编号模式会导致许多层嵌套(双向交互时尤其如此)，但至少比较准确。

如果处理一个图上的并发通信，就可以给每个通信(或线程)指定一个名称，作为顺序号的一部分。例如，可以把 a 和 b 作为两个线程的名称；2.2a 和 2.2b 将同时发生，线程 a 上的 2.3a 以后发生。

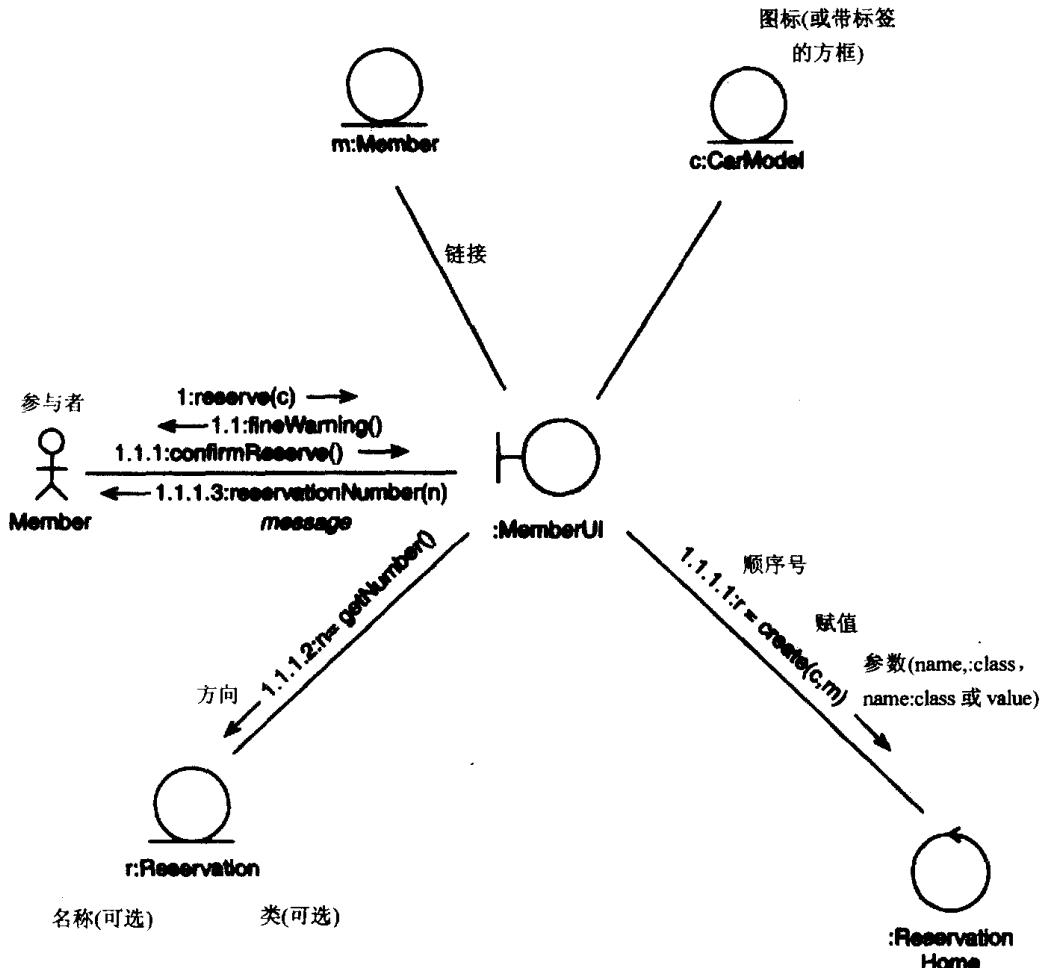


图 7-23 UML 通信图中的元素

#### 7.5.4 给类添加操作

通信图上的每个消息都对应类上的一个操作，所以应记录操作，得到用例实现的完整集合。在类图上，操作可以显示在属性部分下面的单独分隔区中，如图 7-24 所示。另外，操作也可以记录为单独的操作列表，以节省空间。应根据是否使用开发工具、纸或白板来确定选择什么方式。无论采用什么方式，都应确保包含操作的描述。

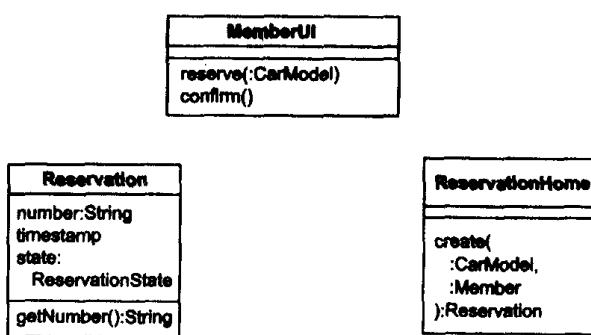


图 7-24 iCoot 的一些操作

UML 操作的一般格式如下：

```
opName (paramName1:paramType1, paramName2:paramType2) :ReturnType
```

每个参数名、参数类型和返回类型都是可选的(如果参数名省略了，参数类型前面的冒号必须保留，以避免混淆)。对于空参数列表，可以省略括号，但本书保留了括号，这样操作就可以与属性区分开来。

### 7.5.5 职责

只要给对象添加细节，就应考虑[Wirfs-Brock 和 McKean 02]描述的职责(responsibility)。这有助于找出和正确指定操作及属性(在静态分析过程中已找出许多属性，但在用例的实现过程中，会发现需要更多的属性)。只要发现系统需要的信息或行为，就应思考“哪个对象负责这个信息或行为？”信息最终以属性(和对象)的方式出现，行为最终以操作(和协作)的方式出现。

再进一步考虑职责，确保每个对象都不会负责多个任务(或角色)。如果一个对象负责提取顾客的预订信息，同时要处理该预订，就需要两个对象(一个边界和一个控制器)。有一组职责的对象称为具有强内聚力，这是我们要达到的目标。

另外，应把对象看做客户(提出问题，给出命令)或提供者(提供答案和服务)。双向协作比较复杂，难以维护。提供者相对于其客户而言是松耦合。双向协作会在两个方向上产生强耦合。本书描述的每个用例的实现都把参与者当做与边界交互的启动者，而边界又会与实体通信，这有助于生成客户和提供者对象。当然，深入系统就会发现，提供者也可能是一个客户，客户可能是一个提供者。

### 7.5.6 状态建模

有时，实体的生存期非常复杂，需要显示在状态机图上。例如，图 7-25 显示了 iCoot 系统中 Reservation 的复杂生存期模型。在这个图中，圆角方框表示状态，其标签给出了它的名称。箭头表示到另一个状态的转化——箭头上的标签表示进行转化的触发器。引出一个箭头的黑色圆表示初始状态——从中生成对象的状态。指向带一圆环的黑色圆的箭头表示最终状态——对象终止其生存期的状态。

图 7-25 从开始到结束状态显示，在 Reservation 创建后，就处于等待状态，一旦到达 Concluded 状态，在系统中就不再承担任何任务。该图的其他部分已描述了许多内容。下面是一个完整的描述——状态机调查。这个描述说明需要一张图来表达。

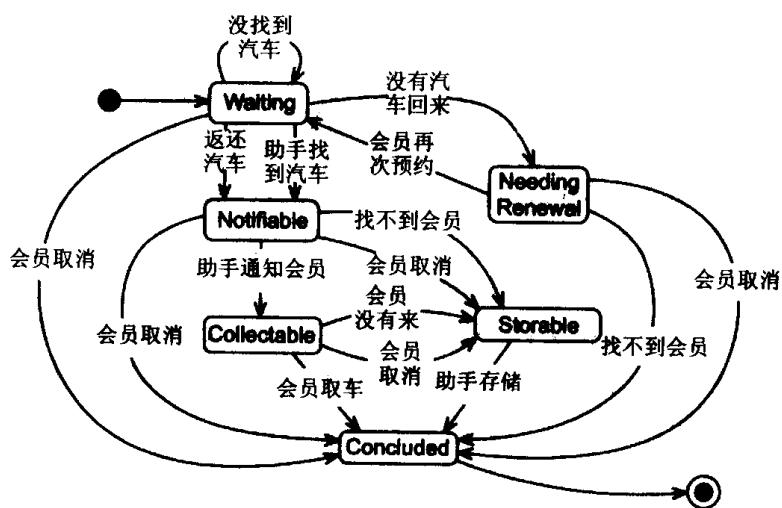


图 7-25 用于预订的状态机图

当会员通过 Internet 预约汽车型号时，Reservation 就等待助手来处理(所以顾客可以在没有助手的帮助下进行预约)。如果以后的某个时刻助手发现在停车场的显示区域中有一辆合适的、未预约的汽车，或者顾客返还了一辆汽车，Reservation 就变成 Notifiable。此时，汽车移动到保留区域。

如果在一星期内都没有给某个 Reservation 提供汽车，该 Reservation 就变成 NeedingRenewal：必须电话或亲自通知会员，让他们取消预约，或在下周重新预约。如果会员取消了预约，或在五天内联系不上，Reservation 就变成 Concluded。

一旦 Reservation 变成 Notifiable，助手就必须在三天内电话或亲自通知会员。如果联系上顾客，Reservation 就变成 Collectable，否则就变成 Displayable(移动到保留区域的汽车必须返还到显示区域)。

一旦 Reservation 变成 Collectable，会员就必须在三天内取车；如果会员取了车，Reservation 就变成 Concluded，否则就变成 Displayable。

一旦 Displayable Reservation 的汽车返还到显示区域，Reservation 就变成 Concluded。

会员可以在任何时候通过 Internet 电话或亲自取消预约。

系统会告诉助手当前预约的状态(还未结束)，这样助手就可以采取适当的措施。

除了前面看到的行为状态机之外，UML 还有协议状态机。后者用于显示消息合法传送给对象的顺序，但它们是类似的。

## 7.6 小结

本章的主要内容如下：

- 如何执行软件开发的分析过程。
- 如何建立静态分析模型，在类图中显示系统面向业务的对象、对象的属性和关系。
- 动态分析如何使用通信图提高和验证静态模型，如何使用状态机图为复杂的生存期建模。

## 7.7 课外阅读

找出合适的对象、属性和关系来源于认真的思考、天赋和经验。尽管这是一个相当哲学化的活动，但可以从许多地方获得帮助。有三本畅销的书值得一看，分别是[Fowler 96]、[Larman 01]和[Martin Odell 98]。Fowler 给出了现实世界的许多分析模型例子，其他两本书讨论了广泛的议题，解决了找出概念性对象的问题。

在找出职责这方面，[Wirfs-Brock 和 McKean 02]是基础。

把系统分解为边界、控制器和实体的理念最初源自于[Jacobson 等 92]。

本书仅在概念上介绍了状态机图，Martin Fowler 在[Fowler 03]中提供了更多的信息，完整的图可参见 UML 规范[OMG 03a]。

## 7.8 复习题

1. 如图 7-26 所示, Car 和 Engine 之间的关系最可能的实现方式是什么? (单选题)
  - (a) 一个字段, 其类型是 Car, 在 Engine 中。
  - (b) 一个类 CarEngine, 它有一个 Car 类型的字段和一个 Engine 类型的字段。
  - (c) 一个字段, 其类型是 Engine, 在 Car 中。
  - (d) 一个字段, 其类型是 Engine, 在 Car 中; 一个字段, 其类型是 Car, 在 Engine 中。

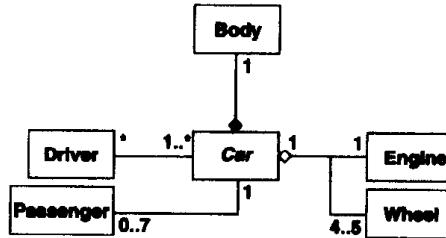


图 7-26 复习题 1 和 2

2. 如图 7-26 所示, 下面哪些陈述是正确的? (多选题)
  - (a) 汽车总是有相同的车身。
  - (b) 一些汽车有备用轮胎。
  - (c) 汽车有一个引擎, 引擎在汽车之间不共享。
  - (d) 所有的汽车都有四或五个轮胎。
  - (e) 汽车必须有至少一个司机。
  - (f) 乘客不可能是司机。

### PizzaBase 案例分析

PizzaBase 饭馆想把顾客预订比萨的过程自动化。每张桌子都配备一个触摸式屏幕, 顾客可以用它浏览所供应的比萨, 并点菜。

该饭馆供应两种基本类型的比萨: 自助类只有西红柿酱, 顾客可以选择任意数量的配料, 每种配料的价格都是固定的。预制类有几个小类, 每个小类都有固定的配料。每种比萨都可以预订酥脆型和松软型, 有三种规格: 6 英寸、9 英寸和 12 英寸。

顾客还可以预订饮料, 例如可口类和柠檬类, 每种饮料都有大杯和小杯两种规格。顾客确认了预订的食物后, 就显示总价。之后, 屏幕显示食物的准备和烹饪进度。在顾客吃完后, 可以以方便的方式付费。

3. 在 PizzaBase 案例分析中, 在分析阶段的属性列表是哪一个? (单选题)
  - (a) 可口、基本类型、价格、规格、柠檬、付费方式
  - (b) 口味、品种、付费方式、总价、显示、肉类、西红柿
  - (c) 进度、品种、口味、价格、触摸式屏幕、规格、饮料
  - (d) 基本类型、价格、品种、规格、进度、口味
4. 如图 7-27 所示, 哪个图是 PizzaBase 饭馆中比萨的最佳模型? (单选题)

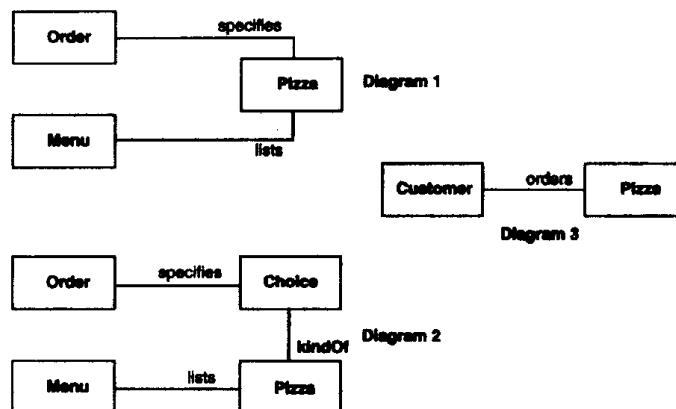


图 7-27 复习题 4

- (a) 图 1  
 (b) 图 2  
 (c) 图 3
5. 在 PizzaBase 案例分析中，分析类最有可能是哪个列表？(单选题)
- (a) Payment, Order, Drink, Topping, Pizza, Order, Restaurant, Base, Sauce
  - (b) Customer, Table, Pizza, Topping, Drink, Restaurant, Order
  - (c) PizzaBase, Cola, Restaurant, Lemonade, Customer, Do-it-Yourself, Prefab, Table, Order
  - (d) Restaurant, Pizza, Topping, Display, Order, Payment, Touch
  - (e) Screen, Order, Pffer, Topping, Size, Meal, Pizza, Restaurant
  - (f) Pizza, Customer, Cook, Table, Crust, Topping, Drink, Restaurant
6. 在 UML 中，哪个图用于显示在对象之间传送的消息？(多选题)
- (a) 活动图
  - (b) 对象图
  - (c) 通信图
  - (d) 状态机图
  - (e) 顺序图
  - (f) 部署图
7. 如图 7-28 所示，哪个图标用于表示包含有用信息的业务对象？(单选题)
- (a) A
  - (b) B
  - (c) C

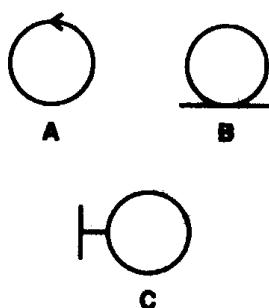


图 7-28 复习题 7

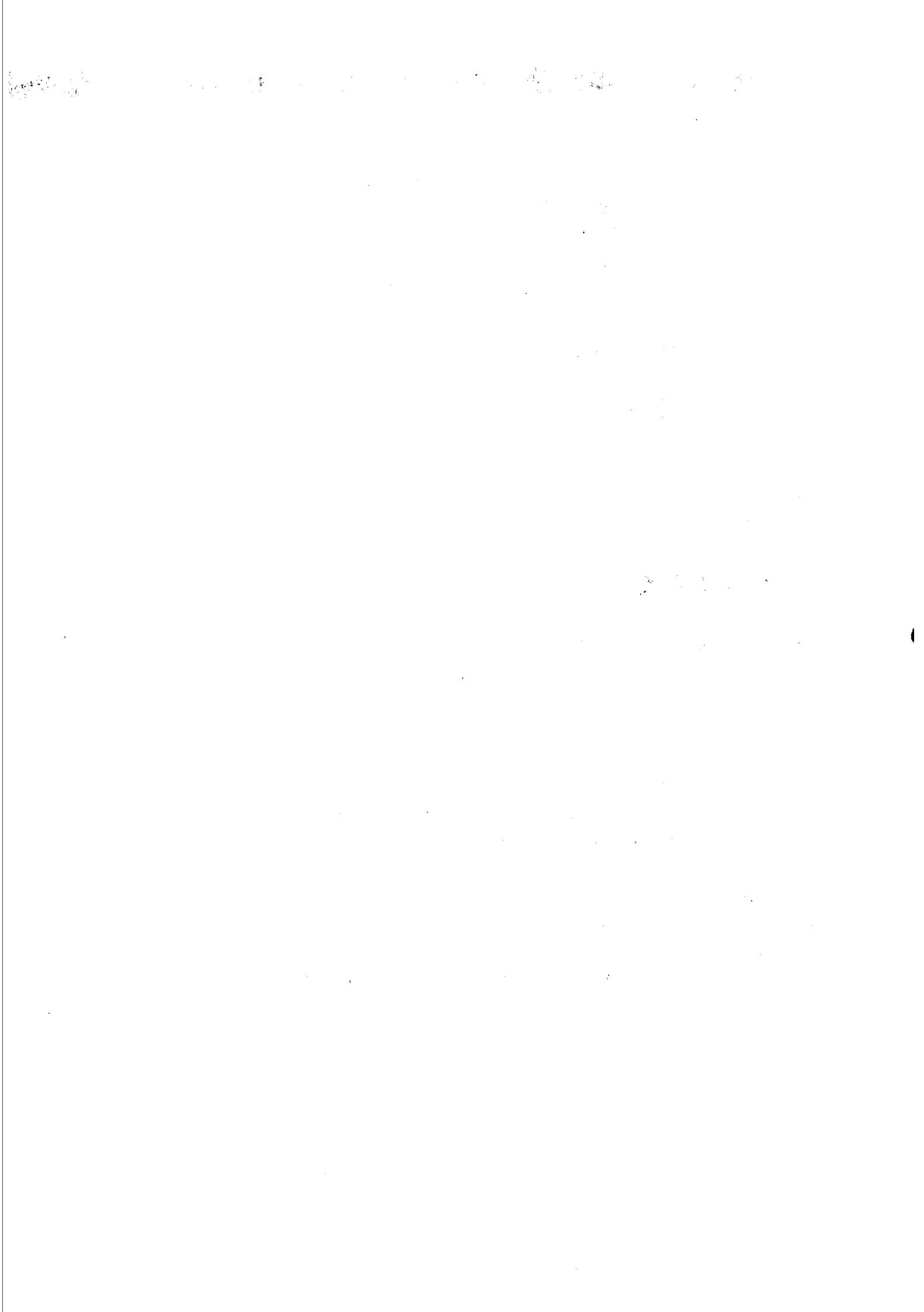
8. 什么是关联类？(单选题)
  - (a) 它描述了可以存在于类之间的各种关系。
  - (b) 它在另外两个类之间的关联中添加属性和/或行为。
  - (c) 它关联对象和该对象所属的类。
9. 如图 7-28 所示，哪个图标用于表示系统之间或人与系统之间的通信路径？(单选题)
  - (a) A
  - (b) B
  - (c) C
10. 如图 7-28 所示，哪个图标用于表示协作系统过程、创建对象或检索对象的对象？(单选题)
  - (a) A
  - (b) B
  - (c) C

## 7.9 练习 4 的答案

图 7-20 中的下半部分是正确的。如果有疑问，可以考虑属性 barCode 和 title 的位置。title 属于 Movie，它的每个记录都有相同的 title。但是，Movie 的每个物理 Video 副本都有自己的 barCode。这说明，有时无形类的名称可以与其有形对象名完全不同。

## 7.10 复习题答案

1. Car 和 Engine 之间的关系最可能的实现方式是 c。
2. 如图 7-26 所示，下面的陈述是正确的：
  - (a) 汽车总是有相同的车身。
  - (c) 汽车有一个引擎，引擎在汽车之间不共享。
  - (d) 所有的汽车都有四或五个轮胎。
3. 在分析阶段的属性列表是 (d) 基本类型、价格、品种、规格、进度、口味。
4. 如图 7-27 所示，b. 图 2 是 PizzaBase 饭馆中比萨的最佳模型。
5. 分析类最有可能是 (b) Customer, Table, Pizza, Topping, Drink, Restaurant, Order。
6. 在 UML 中，(c) 通信图 和 (e) 顺序图用于显示在对象之间传送的消息。
7. 图标 B 用于表示包含有用信息的业务对象。
8. 关联类是 (b) 它在另外两个类之间的关联中添加属性和/或行为。
9. 图标 C 用于表示系统之间或人与系统之间的通信路径。
10. 图标 A 用于表示协作系统过程、创建对象或检索对象的对象。



## **第Ⅲ部分**

### **设计解决方案**

**第 8 章 设计系统体系结构**

**第 9 章 选择技术**

**第 10 章 设计子系统**

**第 11 章 可重用的设计模式**

**第 12 章 指定类的接口**

**第 13 章 不间断的测试**



# 第8章 设计系统体系结构

本章将介绍如何从分析阶段进入设计阶段，集中论述设计的系统体系结构元素。

## 学习目标：

- 理解系统设计的步骤和系统如何分解为物理和逻辑组件
- 在 UML 部署图上演示体系结构决策
- 理解联网系统中产生的并发和安全问题
- 理解如何分解系统，在 UML 部署图中添加分解决策

## 8.1 引言

分析和设计是非常不同的阶段，但其边界有时很模糊。这种模糊有时是故意的，例如在 RUP 中就是这样，有时是无意的，这是因为软件开发得不好。分析和设计之间有清晰的分界比较好，可以确保在考虑解决方案之前，很好地理解问题。有了这种分界，分析就是调查问题，而设计就是找出解决方案。简言之，分析=内容，设计=方式。

把分析模型转换为设计模型并没有硬性规则。其他方式忽略了软件开发中涉及的人的因素和创造性——如果可以系统化，软件开发就像工程一样，第 5 章已经介绍过这个内容。设计过程由开发完整系统的需求、小组的经验、重用机会和个人喜好驱动。一旦设计人员研究了需求，分析了制品，就可以开始设计了；我们不关心分析对象和设计对象之间是否有严格的对应关系，只要设计可以得到有效的解决方案即可。

在设计阶段，要确定技术选择(例如，编程语言、协议和数据库管理系统)，必须确定这些选择对设计的影响情况。技术选择会影响可用的库、模式和框架，甚至会影响所使用的 UML 表示法。

设计越通用，与特定技术的关系就越小——这会减少开发人员精通多项技术的需求，防止使用已废弃或得不到支持的技术。通用化的缺点是不能从特定的技术中获得最大利益(在这里，重用和性能是非常重要的因素)。

历史证明，技术的出现和消亡比支持这些技术的理论更快。例如，编程语言包括 COBOL、Fortran、Pascal、Ada、Modula、PL/I、C、C++、Smalltalk、Eiffel、C# 和 Java，但这些技术的使用仅由两个理论来控制：结构化编程和面向对象编程。因此，一般化比特殊化更安全。

在一般化设计的重用方面，必须进行一些调查，才能发现每种实现方式的新重用机会。在性能方面，使设计一般化不会使性能的损失超过 10%(给系统加上额外的容量，使用更多的机器或更快的机器，就能把这 10% 的性能损失弥补回来)。

在整体上，除了讨论目前可用的常见技术之外，本书将仅依赖面向对象的理论和一般的 UML 表示法，尽可能进行一般化的讨论。一个例外是必须在图中添加原型类型的地方，将使用 Java 原型：这将在查看 Java 代码段时避免混乱，因为一些 Java 类有类似于原型的名称(这不是 UML 的严格通用用法，但 UML 标准允许不太严格的用法)。本书的大部分都避免使用 Java 数组，而使用集合类，因为集合类比较好。

## 8.2 设计优先级

面向对象的软件开发是递增的，不可能一次就设计出完整的系统。所以在每个设计阶段的开始，都需要规划应设计系统的哪些部分。在此，用例优先级是有帮助的，用例紧急程度也是有帮助的，后者在需求阶段使用交通灯来模拟：绿灯必须完全设计好；黄灯不必设计，但必须支持；红灯禁止设计，但仍应支持(“设计”表示确定解决方案，“支持”表示可能找出合理的解决方案，这需要进行一些预测)。

实际上，我们要找出系统的体系结构，为所有的用例支持实际的、有效的解决方案。在这个体系结构中，对最重要的用例进行详细设计，对不太重要的用例进行部分设计。在递增过程中，要适当地调整优先级、紧急程度和设计。

## 8.3 系统设计中的步骤

设计可以看做有两个不同的活动：系统设计和子系统设计。系统设计注重于从较高的层次来考察任务，之后进入子系统的设计(参见第 10 章)。当然，在面向对象的传统中，可以模糊边界，反复斟酌，螺旋式上升，但有两个活动的理念仍是最基本的。

系统设计包括如下活动：

- 选择系统拓扑：硬件和过程如何在网络上分布。
- 选择技术：选择编程语言、数据库、协议等(参见第 9 章)；一些决策可能要在设计阶段的后期才能做出。
- 设计并发策略：并发意味着事情同时发生——多个过程、用户、机器；软件必须能协调这些事情，以避免混乱。
- 设计安全策略：安全有许多方面，每个方面都正确处理和控制。例如，考虑顾客的个人数据——必须确保这些数据不被罪犯盗走，确保数据不会无意展示给其他顾客。
- 选择子系统部分：开发一个解决所有问题的系统常常是不切实际的，我们需要开发若干个软件，然后确保这些软件可有效地通信。
- 把子系统分解为层(layer)或其他子系统：每个子系统一般都需要进一步分解为可管理的块，然后才能进行详细设计。
- 决定机器、子系统和层如何通信：通信决策通常是其他步骤的一个副产品。

## 8.4 选择联网的系统拓扑

系统拓扑是指系统如何分解为几个物理和逻辑组件。本节将概述网络体系结构的历史，接着讨论当前的体系结构问题：瘦客户和胖客户、网络、客户—服务器应用程序和分布式应用程序。我们还将论述如何使用 UML 部署图演示体系结构决策。

### 8.4.1 网络体系结构的简史

大多数现代的联网系统都有三层体系结构。为了说明“三层”的含义，理解它的优点，需要先简述历史上的其他体系结构。

在 20 世纪 40 年代，计算机是大型的单块集成电路设备，一次只能运行一个程序。这些单块集成电路机器后来演变为大型机，它可以同时运行多个程序，一般每个用户运行一个程序或者每批运行一个程序(批包含类似的数据集，它们按顺序在程序中运行，以处理电子账单)。大型机可以同时处理多个程序，因为它们采用了时间片(time-slicer)，在 CPU 上逐个位地运行每个程序，每个程序依次获得时间片，进行一些处理。

首先，用户和批管理员要通过电传打字机访问大型机，电传打字机是一种电子打字机，操作员用它一次给大型机发送一行文本(文本表示程序命令或程序数据)。接着，运行在大型机上的程序发送回一行或多行文本，作为回应。随着技术的进步，电传打字机被哑终端(dumb terminal，也称为绿色屏幕 green screen)代替，哑终端使用阴极射线管(Cathode Ray Tube，CRT)代替纸张，用于文本的输入和输出。有了哑终端，操作员就可以准备一整屏的命令或数据，然后一次把它们派送给运行在大型机上的程序(如图 8-1 所示)。

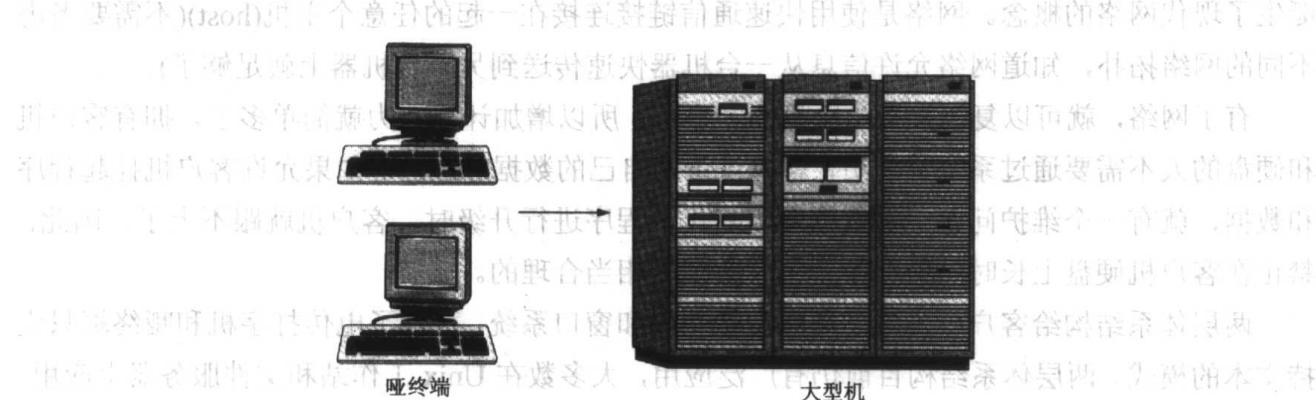


图 8-1 大型计算机：一层体系结构

大型机模型目前仍用于大型业务工作，是一种一层体系结构。这意味着，对于任何给定的程序，都只在一台机器上进行一个层次的计算活动(电传打字机和哑终端都不进行任何处理)。换言之，一层体系结构没有网络：尽管在每个终端和大型机之间都有一条线路(可能是电话线)，但不能构成现代意义上的网络。

大型机的主要优点是建立很简单；主要缺点是只能通过购买新大型机或升级已有的大型机，来提高计算能力。

下一代体系结构在 20 世纪 70 年代非常风行，称为两层体系结构(但当时并不这么称呼)。其理念是在每个客户机上都带有一个可选的硬盘，都有处理能力，所以大型中央机就不必完成所有的处理。因此，添加或替代客户机就是改变中央机的一个便宜方式。小型机和工作站引入为客户机，分别用于访问中型计算机和文件服务器(如图 8-2 所示)。这里显示的小型机—中型计算机组合(例如 VT100 和 PDP/11)与工作站—文件服务器组合(例如 Solaris 工作站和 Solaris 文件服务器)是一样的，它们只是来自于不同的团体。

图 8-2 两层体系结构



图 8-2 两层体系结构

在两层体系结构中，程序和数据必须从大型中央机传送给客户机，这需要快速连接，从而诞生了现代网络的概念。网络是使用快速通信链接连接在一起的任意个主机(host)(不需要考虑不同的网络拓扑，知道网络允许信息从一台机器快速传送到另一台机器上就足够了)。

有了网络，就可以复制大型中央机和客户机，所以增加计算能力就简单多了。拥有客户机和硬盘的人不需要通过系统管理员，就可以管理自己的数据和程序。如果允许客户机挂起程序和数据，就有一个维护问题：在数据发生变化，程序进行升级时，客户机就跟不上了。因此，禁止在客户机硬盘上长时间地存储程序和数据是相当合理的。

两层体系结构给客户机带来了高级图像功能和窗口系统，替代了电传打字机和哑终端只支持文本的模式。两层体系结构目前仍有广泛应用，大多数在 Unix 工作站和文件服务器上应用。

## 8.4.2 三层体系结构

三层体系结构(如图 8-3 所示)在 20 世纪 90 年代开始流行，是在联网系统上分隔用户界面、程序逻辑和数据的方式，这是它很快流行起来的原因。在三层系统上，任何程序都至少涉及三台机器：数据层存储数据，提供对数据的安全并发访问，一般要使用数据库管理系统(DBMS)；中间层也称为业务逻辑层或服务器层，它使用大型处理器和大量的内存运行多线程的程序代码；客户层给用户显示用户界面，以便用户输入数据，查看结果。

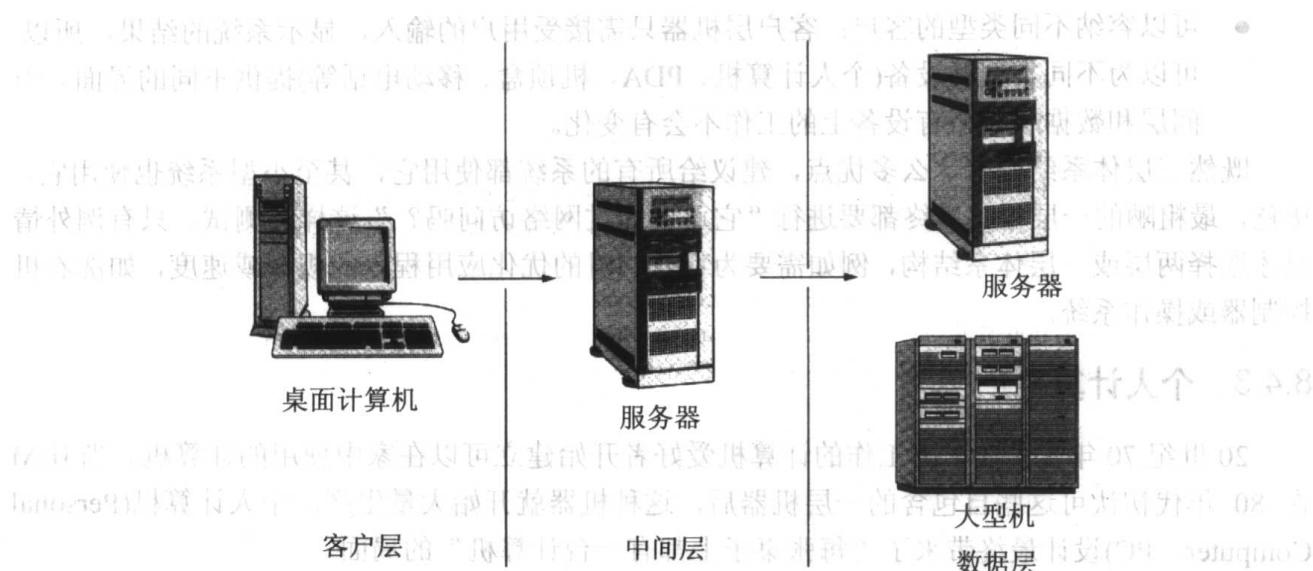


图 8-3 三层体系结构

三层体系结构有如下优点：

- 分解重要的部分：任何大型系统都必须考虑安全性、大量数据的有效管理、高吞吐量的编程逻辑和简单的用户界面。为这些部分单独编程，开发人员的工作就较容易完成，并可以优化每个部分的设计。
- 使用正确的机器完成工作：并没有什么适合一切工作的计算机。运行用户界面是一个简单的工作，不需要大型机，甚或文件服务器；执行编程逻辑要使用大量的CPU和内存，但不需要巨大的磁盘空间，所以可以使用强大的服务器；管理大量数据(例如一百万电子顾客)需要目前所有的计算能力和磁盘空间，所以这需要大型服务器或大型机。
- 改进性能：可以在数据层或中间层复制适当的机器，将计算负载分散开(负载平衡，load balancing)，每一层都专业化，以便于优化。
- 改进安全性：三层系统在部署时，常常使客户机在互联网上运行。因此，必须有一个严格的安全策略来保护内部机器、程序和数据。在三层体系结构中，可以确保中间层的安全——阻止外部的无意或恶意攻击。数据层在防止破坏的中间层的后面，所以不必保护其通信或硬件；这意味着数据层更容易编程，运行也较快(但是，人们越来越觉得，内部层需要防止员工的破坏)。
- 保护投资：当已有的大型机多年来一直存储大量数据，并进行批处理时，就不可能抛弃所有的东西，再从头开始，以获得网络和三层体系结构的好处(一般情况下，最好不要重新编写程序，这样最后得到的另一个1.0版本还会缺乏通常的功能，存在性能问题和缺陷)。只要大型机厂家可以使大型机通过网络来访问(大多数厂家已经这么做了，因为有了客户机—服务器的革新)，中间层就可以看做大型机的客户，以及真正客户的服务器。这样就可以把原来的系统转换为现代系统。
- 灵活：除了能随意增加和减少机器这个明显的灵活性之外，如果用三层体系结构设计系统，部署也会非常灵活。例如，只要逻辑分区是正确的，就可以在一层上开发系统，然后部署到三层、二层(中间层和数据层合并起来)或一层(所有三层都合并在一起)上。

- 可以容纳不同类型的客户：客户层机器只需接受用户的输入，显示系统的结果，所以可以为不同类型的设备(个人计算机、PDA、机顶盒、移动电话等)提供不同的界面。中间层和数据层在所有设备上的工作不会有变化。

既然三层体系结构有这么多优点，建议给所有的系统都使用它，甚至小型系统也使用它。毕竟，最粗陋的一层程序最终都要进行“它可以通过网络访问吗？”这样的测试。只有例外情况才选择两层或一层体系结构，例如需要为特殊的目的优化应用程序的规模或速度，如洗衣机控制器或操作系统。

#### 8.4.3 个人计算机

20世纪70年代，在车库工作的计算机爱好者开始建立可以在家中使用的计算机。当IBM在80年代初认可这些自包含的一层机器后，这种机器就开始大量生产。个人计算机(Personal Computer, PC)设计最终带来了“每张桌子上都有一台计算机”的局面。

在80年代末期，显然，通过公司网络把PC、文件服务器和大型机连接起来，PC就更有用。结果得到了两层体系结构，其中，PC执行复杂的任务，例如电子表格的编辑和文档的生成，而公司网络上的机器提供电子邮件，最少也能访问公司数据。

自从90年代以来，PC越来越普及。在公司中，它们最初得益于IBM的认可，接着得益于其简便和低成本(与工作站相比)。PC还提高了在家工作的可能性。在家用市场，没有什么竞争：工作站的高成本和在家访问外部网络的低速度，使PC无可替代。

目前，PC设计使早期由在车库工作的计算机爱好者开发的其他产品无立足之地。惟一的例外是Apple产品(在图形设计人员中尤其流行)。

#### 8.4.4 网络计算机

90年代中期，大型机、Unix和PC争夺市场的大战如火如荼，PC的维护成本开始成为一个令人头痛的问题。如前所述，如果给某人提供一台带有硬盘的机器，硬盘上的数据和程序就不同于中央数据和程序，这会导致错误(例如，顾客数据过期)和高昂的额外成本(每台客户机都必须送回中央管理站，安装电子邮件程序、字处理程序等的新版本)。

因此，出现了网络计算机的概念。网络计算机根据需要从大型中央服务器或大型机中获得数据和程序，这需要一个快速的网络(每秒数十或数百兆)。为了防止个人损坏模型，网络计算机没有硬盘(它们可以有硬盘，但只由操作系统用于缓存数据和程序)。网络计算机只是没有磁盘的工作站，一般运行某个版本的Unix，因为Unix很适合在快速网络上根据需要处理程序负载。

网络计算机是一个非常好的方法，多年来，没有磁盘的工作站取得成功是不足为奇的。但网络计算机没能突破PC的束缚，它们甚至没有在其流行的领域内突破有磁盘的工作站的束缚(无论成本如何，无论老板浪费了多少时间来阻挠，甚至专业人士也保留一些自治权)。

Java爱好者使用网络计算机在公司中传播Java。网络计算机在本质上是Unix机器，而Java程序可以在Unix上运行，所以Java和Unix突然遍及各地。但是，网络计算机制争获得流行。网络计算机从来不能用于家庭，因为互联网连接(甚至带宽)太慢。为了超越PC，家用用户需要有Web浏览器才能访问三层体系结构。

术语“瘦客户(thin client)”和“胖客户(fat client)”与网络计算机几乎同时出现。其意图是强调瘦的优点，意思是客户机(由网络计算机代表)比肥大、带有磁盘、连接到网络上的PC更好。瘦客户是不错：无论是坐在PC前，还是坐在工作站前，所有重要的工作都应以网络为中

心。在公司中，这可以使用访问三层系统的 Java 小程序来达到。在家中，最好的选择是访问三层系统的 Web 浏览器。

#### 8.4.5 互联网和万维网

在 80 年代中期，许多研究人员和政府雇员都在使用遍及整个世界的计算机网络，该网络称为互联网(Internet)。互联网起源于 US 研究和防护机构，该机构在十年前开发出了 ARPAnet。互联网的特性是可以自由访问中央服务器，允许机器和人员根据互联网地址定位其他机器。互联网地址有小数点格式(如 100.99.88.32)和符号 ASCII 格式(如 www.nowherecars.com)。在内部，互联网依赖于低级协议 TCP/IP：理解 TCP/IP 的任何机器都可以访问互联网上的所有设备。

在 90 年代早期，互联网在研究团体中稳固地建立起来(用作电子邮件和文件传输的工具)，此时，在瑞士研究所 CERN 工作的 Tim Berners-Lee 提出了文档的观点，该文档包含互联网上其他文档的超链接。Berners-Lee 发明了一种文档布局语言(超文本标记语言，HTML)和一个协议(超文本传输协议，HTTP)，使他的观点成为现实。HTTP 允许任何机器通过超链接从其他机器上加载文档。文档的位置可以使用统一资源定位器(Uniform Resource Indicator，URI)指定，URI 的格式是 <http://www.nowherecars.com/index.html>。

Berners-Lee 的发明被命名为(信息的)万维网(World Wide Web)。在 90 年代中期，如果需要使文档能为其他人读取，就可以把文档部署到 Web 服务器上，要读取文档的人可以在其客户机上运行 Web 浏览器。互联网服务提供者(Internet Service Provider)出现后，万维网就遍布全世界，可以在家中访问互联网了。目前，互联网和万维网的区别已经模糊了。有许多术语都可以互换使用，例如互联网、网络、万维网、网或信息高速公路。

#### 8.4.6 内联网

互联网有两个主要问题：很慢、不安全。这些性能问题的出现是因为，信息在到达其目的地之前，必须旅行数千公里，这常常会导致连接和服务很慢。另外，信息必须与其他在运输线上的信息竞争。互联网的不安全是因为，每个人都能访问互联网，因此，任何人都可以截听和读取正在传输的信息——即使加密了信息，一些人也有可能解密它。

因此人们引入了内联网(intranet)的概念，它表示运行在封闭环境下的“微型互联网”。内联网表示“内部的网络”，与互联网正好相反，互联网表示“各个环境之间的网络”。内联网一般由一个公司或政府控制。通过防止从外部访问内联网，可以提高性能(使用强大的机器，且与外部通信没有竞争)，而且不需要过分担心安全。对于全球性的内联网，信息仍要旅行很长的距离，所以在整体上，仍必须先考虑局域网的性能。

就安全而言，需要确保未授权的人不能进入任意一个站点，连接他们自己的计算机。除此之外，还必须处理所有常见的问题，例如行业间谍、不满或粗心的员工、监视机器和网络的电磁辐射波的间谍——据谣传，一个政府机构要求其员工在 Faraday 笼子中使用计算机，通过笼子外部的感应提供主电源：这是一个妄想狂的问题。

对于大多数内联网，都可以达到本地的性能，有安全优势，还允许员工访问互联网(收发电子邮件、收集信息和监视竞争对手的网站)。为此，需要使用互联网防火墙。防火墙是一种软件，允许内联网上的机器访问任意 TCP/IP 地址，且使内联网地址在外部不可见。互联网防火墙还可以执行其他任务。例如，可以确保只有 Web 浏览器才能从员工的机器上访问互联网，

防止流氓程序潜入，与外界交流机密信息；可以配置互联网防火墙，允许外界的电子商务请求通过隧道进入电子商务服务器。在家中，可以使用个人防火墙防止 PC 受到黑客和病毒的攻击。

#### 8.4.7 外联网和虚拟私人网络

网关与防火墙 3.4.7

如何利用内联网的性能优势和安全优势进行业务对业务的通信？这很容易——使用外联网(extranet)，这是一个或多个内联网之间的安全连接。术语“外联网”是内联网上的一个双关语，是“外部内联网”的简称。建立外联网最简单的方式是，在每个内联网的边界上运行互联网防火墙上的一套软件，它执行两个任务：

- 允许信息在防火墙之间传送。
- 使用强大的加密技术保护在互联网上传送的信息。

外联网也称为虚拟私人网络(Virtual Private Network, VPN)。最简单的 VPN 甚至比外联网出现得还早，即工人在家中使用专门建立的软件给其公司 LAN 拨号。

图 8-4 演示了内联网、外联网和互联网的相互关系。

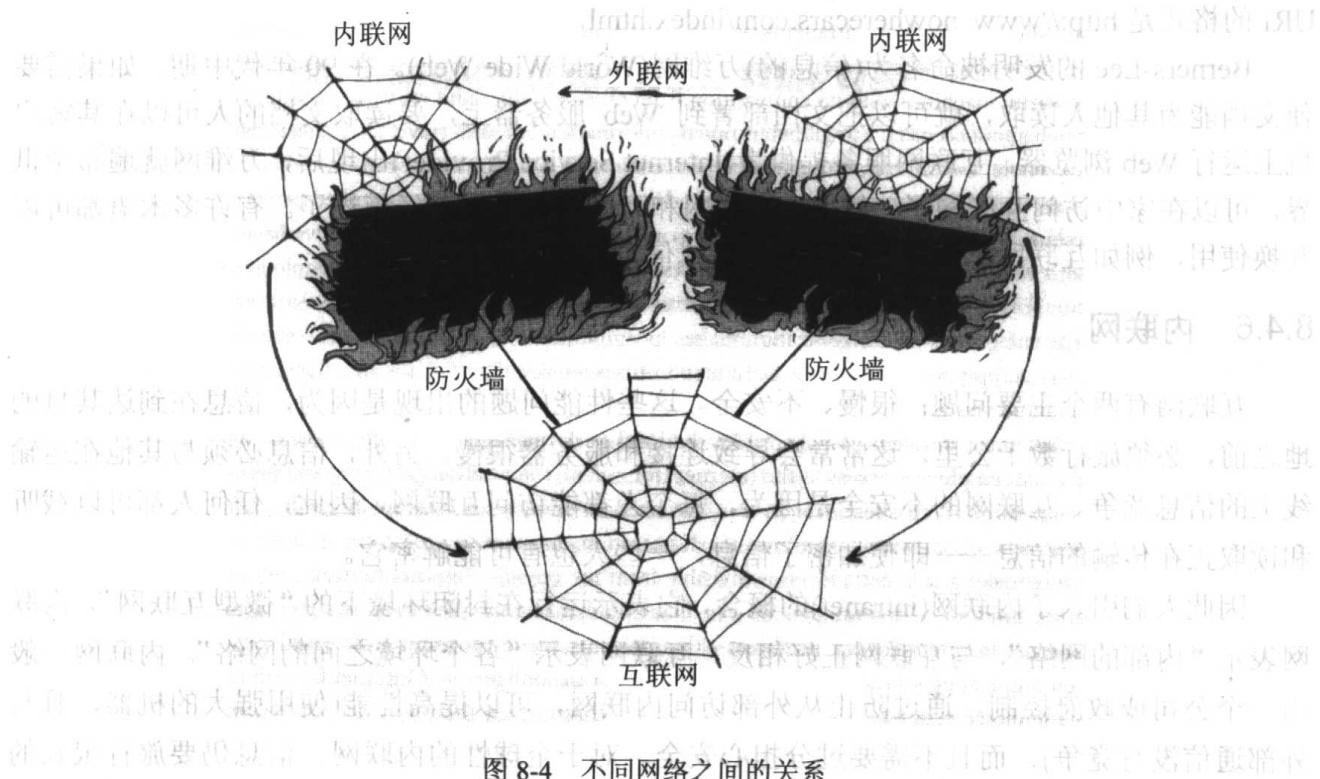
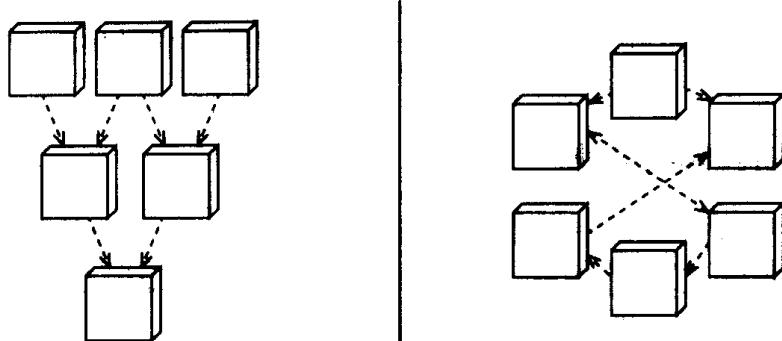


图 8-4 不同网络之间的关系

#### 8.4.8 客户机—服务器与分布式体系结构

只要连接多台机器或多个软件系统，就必须在客户机—服务器与分布式体系结构中选择，如图 8-5 所示。尽管客户机—服务器起源于大型机领域，但目前它只意味着，大量小型的简单客户机给几个大型多线程的服务器发送请求，服务器会处理这些请求。另一方面，分布式(或对等 peer-to-peer)体系结构的特点是，一组独立的对等机器根据需要在任意方向上通信。

图 8-5 展示了客户机—服务器与分布式体系结构的区别。客户机—服务器模型显示，一台或多台客户机向一台或多台服务器发送请求，服务器处理这些请求并返回结果。分布式模型则显示，多台对等机器直接相互通信，没有中央服务器。这种点对点连接使得数据可以在整个网络中流动，从而提高了系统的弹性和效率。



客户机 - 服务器结构

分布式体系结构

图 8-5 客户机 - 服务器体系结构和分布式体系结构

客户机—服务器体系结构最常见的例子是电子商务模型：顾客的 Web 浏览器给公司的 Web 服务器发出请求，公司的 Web 服务器给后端系统发出命令。大多数二层和三层系统都是客户机—服务器。

分布式体系结构的一个例子是，一个繁重的计算任务散布到许多互联网机器上。如果需要处理非常多的数据或过程，而且可以分解这些数据或过程，就可以把它们分布到独立的机器上。

一个例子是 SETI@home，它是一个非盈利组织，主要寻找外星人的无线电信号。搜索外太空文明(SETI)的活动开始于一个 NASA 项目，该项目使用无线电望远镜扫描天空，记录下遇到的任何信号。然后处理这些数据，看看它们是否包含外星人发出的连贯的无线电信号。最终，SETI 项目被取消(因为缺乏资金或信任)，再次提出该项目时，它已重新命名为 SETI@home。开发 SETI@home 的爱好者收集相同的无线电数据，把它分布到连接上互联网的机器上：每个参与的机器都使用运行为屏幕保护程序的软件分析该数据，并把有趣的结果发送回中央服务器，做进一步的分析。更多的信息可参阅 SETI@home 网站 [setiathome.ssl.berkeley.edu](http://setiathome.ssl.berkeley.edu)。

虽然 SETI@home 有一个中心数据库，但它仍是分布式体系结构，因为大多数处理工作都由独立的节点完成。把大量的机器汇集在一起，完成一个复杂任务的这个理念已用于素数研究和癌症研究。其基本理念正在栅格计算小组的领导下研究。

术语“客户机—服务器”和“分布式”(或“对等”)也用于描述软件体系结构，独立于软件部署到物理机器和网络上的方式。运行在程序中的对象就是一个好例子：正常情况下，把对象编写为可以在不同环境下重用不同客户机对象的服务器，但对于特定目的的应用程序，也可以编写以分布方式协作的对象组。

网络通信链接是双向的，也就是说，该链接最初对客户机是开放的，但服务器也可以给客户机发送信息(客户机是否接收信息取决于客户软件的设计人员)。因此，严格地说，客户机—服务器和分布式体系结构的区别是人为的，由设计人员用于以某种方式构建其解决方案。

一般说来，客户机—服务器体系结构很容易开发，它们还提供了理论上最高的性能(例如，在服务器处理一个请求时，客户机通常是空闲的)。分布式体系结构通常较难开发，但提供了比较好的性能。体系结构的选择常常比较明显。例如，购买过程需要按步骤实现(顾客请求商品的详细信息，提供者提供该商品的详细信息，顾客选择要购买的商品，提供者提供购买表单，顾客提交购买表单等)，因此是一个很自然的客户机—服务器交互。相反，多用户飞行模拟器

需要每个飞行员的计算机显示驾驶员座舱、场景和其他航行器的复杂实时图像，惟一需要的通信是每台机器对飞机当前位置的定期广播。因此，多用户飞行模拟器是一个自然的分布式体系结构。

#### 8.4.9 用 UML 描述网络拓扑

系统体系结构可以用 UML 在部署图上描述(如图 8-6 所示)。这个简单的部署图只显示了节点、通信路径和多重性。在这个图中，每个节点都表示一个主机(用 UML 关键字<<device>>表示)。通信路径表示两个节点以某种方式通信。节点可以指定多重性，表示运行期间存在多少个节点：因此，在这个图中，显示了复制的节点(CootServer 和 DBServer)和繁殖的节点(CootHTMLClient 和 CootGUIClient)。

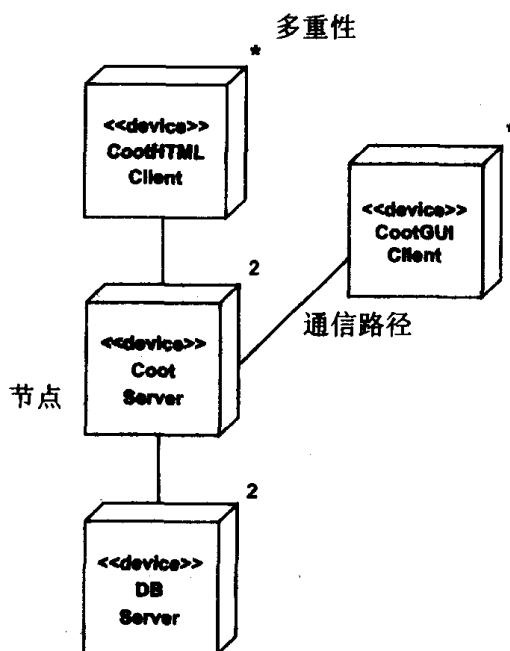


图 8-6 Coot 的基本部署图

部署图类似于类图和对象图，因为它们都可以显示可能的体系结构(节点类型)和实际的体系结构(节点实例)。在显示节点实例时，就像对象图中的对象一样，节点标签也采用 name:Type 格式，也应加下划线。

如果大多数部署图是有意义的，就都需要附带的描述。按照 Jacobson 格式，这称为部署调查。

#### 案例研究

##### iCoot 部署调查

这个描述可以用于图 8-6：

iCoot 数据层包含两个数据库服务器(称为 DBServer)。有这两个节点可提高通过量和可靠性。

中间层与数据层通信包含两个服务器(CootServer)，也用于提高通过量和可靠性。

每个 CootServer 都可以由任意多个 CootHTMLClient 节点同时访问。

最后，还提供了从 CootGUIClient 节点的访问。

## 8.5 并发设计

大多数系统，尤其是联网的系统，会在同一时刻发生许多事件，也就是说，它们是并发系统。这隐含着：系统应作为一个整体，各个过程运行为系统的一部分。如果可以让所有的用户和过程形成有序的队列，系统的开发就要容易得多，但实际上必须通过编程使混乱变得有序。

并发引入了如下问题：

- 如何确保在他人访问之前，完全更新信息。例如，在添加新汽车型号的所有信息之前，必须阻止他人访问这些信息。
- 如何确保信息在读取的同时不被更新；例如，在查看一个汽车型号的信息时，不要删除该汽车型号。

在低层次上，数据库事务和线程监视器用于保护各个过程内部的数据。在较高的层次上，需要使用系统规则和业务规则控制并发活动。

并发的最简单方式是限制系统或引入额外的业务规则，尤其是当用户的访问次数没有明显减少时，就更是如此。例如，在 iCoot 中，在顾客访问汽车目录时，不应试图更新它，而可以在一个独立的数据库中更新目录，每天切换一次数据库。这样互联网子系统就可以假定该目录是只读的，从而使代码更容易编写(用户访问次数略微减少——很少需要告诉顾客，他们要预约的汽车型号目前暂时没有——但我们选择告诉顾客)。这是对系统的人为限制。

业务规则也可以使开发人员的工作更容易完成。实际上，有时引入业务规则是因为，没有更好的方法解决问题。例如，考虑音乐会门票的购买系统。顾客 Fred 走进巴黎的预订办公室，几乎在同时顾客 Beryl 走进了位于纽约的另一个预订办公室。两位顾客都决定购买伦敦同一场音乐会的门票。可惜，只剩下一张票了。如何决定谁买到这张票？两个顾客都在问：“有票吗？”两个售票员都查看系统，都回答“有”。现在就有一个竞争：第一个说“好，我要一张”的顾客会买到这张票，这依赖于售票员的效率和从巴黎和纽约到实际服务器位置的网络延迟。

在售票的例子中，至少要确保只有一张票时，售票员不是漫不经心地卖出两张票。这是一个过程级的并发问题，因为它可以由服务器过程控制(请求串行化：第一个到的人买到票，第二个到的人得到一个错误消息)。但从业务的角度来看，这还不够好，因为顾客几分钟前被告知有票，但最后没有买到票，他就会对系统不满。

为了避免顾客对系统不满，可以引入一个额外的业务规则：当售票员查询是否有票时，如果有，就临时保留它——该预订一直持续到售票员取消查询或预订过期(例如，如果售票员没有取消，临时预订就持续十分钟)为止。有了这个新的业务规则，就可以确保，只有第一个对票务服务器进行虚拟访问的顾客，才会被告知有票(票务服务器可以把查询和预订合并到一个业务服务中)。

假定 Fred 成功买到票。给 Fred 提供服务的售票员有十分钟的时间劝说 Fred 给预订的票付款，如果 Fred 改变了主意，售票员可以在这段时间内取消预订(如果 Fred 的付款方式失败，售票员也要取消预订，这是设置临时预订的另一个原因)。Beryl 现在认为，在她到预订办公室之前音乐会的票就卖完了，所以不会谴责预订机构给她提供了不正确的信息(如果最后一张票没有被 Fred 买走，Beryl 后来发现音乐会的票没有卖完，只需告诉她“有人退了票” )。

并发问题及其解决方案的详细论述超出了本书的范围。无论如何，都必须坐下来，认真考虑并发问题。下面是一些注意事项：

- 设计优秀的并发系统的外观和操作方式与单用户版本没有区别。
- 业务服务对并发用户和单用户是相同的。
- 为了确保业务对象的并发操作的安全，只需添加消息和支持对象；因此，业务消息(和相关的属性)可以单独设计。

如果并发情况会给系统带来困难，在确保该情况不再有问题之后，再实现并发操作。要使系统强壮起来，因为现实是残酷的。

## 8.6 安全设计

安全的详细论述尽管很吸引人，但需要一本书的篇幅来论述。所以这里只是概述一下。

安全系统可以阻止无意或恶意的误用。安全是一个相当宽泛的术语，可以分解为五个方面：

- 私密性：必须隐藏信息，只有授权的人才能读取它。
- 验证：需要知道信息从何而来，以便决定信任或不信任它。
- 不能反驳的信息：这是验证附带的一个功能，确保信息的初始提供者不能否认它们是信息的来源，如果事情出错，这将有所帮助。
- 完整性：必须确保信息在从其来源传送给我们的过程中，不受到无意或恶意的破坏。
- 安全性：必须控制对资源(例如机器、过程、数据库和文件)的访问。安全性也称为授权(authorization)。

在这里，信息不仅表示数据，例如业务文档和用户密码，还表示可执行的代码。代码是一个问题，因为它可以通过网络动态加载。

上述前四个需求可以使用数字加密来满足(详见下一节)。安全需求相当复杂。在正常情况下，运行一段代码时，操作系统会对代码要做的工作施加某种控制，一般这意味着控制对文件、目录和其他程序的访问。操作系统有错误和安全漏洞。另外，操作系统提供的控制是比较死板的；例如，用户或开发人员无法开发自己的抽象控制，如“只有 Freda 和 Ben 可以加入扑克的虚拟游戏”或者“只有家庭成员才能使用移动电话启动微波炉”。

如果系统通过网络来操作，安全方面就更重要。这是因为，在联网环境下，黑客可以劫持运行在机器上的程序，让他们为所欲为。更糟糕的是，由于 Java 和 ActiveX 的流行，代码段可以在网络上传输，在不同的机器上执行。Java 是惟一的主流技术，给代码段在网络中传输增加了一层需要的安全性，而且无论使用什么操作系统，都可以开发自己的控制。也就是说，如果没有使用 Java，就应防止代码在网络上传输：根据需要在每台客户机或服务器上安装代码，依赖操作系统和自己编写的程序来确保安全。

### 8.6.1 数字加密和解密

下面介绍数字加密和解密——密码学(cryptography)——如何用于提供私密性、验证、不能反驳的信息和完整性。首先，基本概念：加密信息就是搅乱信息，即使有人盗取了信息，也看不懂。为此，加密方法必须是可逆的，信息的接收者必须知道该方法。加密的逆过程就是解密。

大多数小孩子都玩过“凯撒密码(Caesarian Cipher)”的游戏(有的地方可能不这么叫)。该游戏的玩法是，获取一个消息，然后按照字母表把每个字母向下移动一定的位置，如果到了消息的末尾，就把移动后的字母组合在一起。例如，如果选择移动四个位置，消息“Mister Watson, can you hear me?”就加密为 QMWXIVAEXWSRGERCYSLIEVQI。(在加密的版本中没有考虑

标点符号、字母大小写和空格，以避免给间谍留下线索)。只要接收者知道算法是 Caesarian，偏移量是 4，就可以解密出原来的消息。

“凯撒密码”是加密的一种普通形式，它涉及到接收者必须知道的一个额外信息(4)和算法(Caesarian)。额外的信息称为密钥，因为它解开了密码，开门不仅要旋转钥匙，还需要有钥匙。

加密机制的成功有两个因素：

- 破解密码的难度——试验，错了再试验。
- 把密钥安全地交给接收者。

这两个因素都是有问题的。例如，“凯撒密码”很容易破解，实干家只需猜测使用了凯撒加密方法，然后至多试验 26 种可能的偏移量，就可以看到原始的消息。至于密钥的安全发布，可以试着把密钥的信息告诉每个接收者，这种方法不但不方便，还容易失败——如果有健忘症的接收者把密钥写在一张纸上，就会出现安全漏洞。

可以使用不易破解的加密算法，例如一次性加密(one-time pads)和量子加密(quantum cryptography)，但这些算法要么不方便(密码手册)，要么太昂贵(量子加密，至少目前比较贵)。所以，对于日常使用的算法，尤其是网络编程中使用的算法，一般应选择需要的安全级别，再选择提供该级别的高效算法。这是比较合适的，因为如果加密的信用卡信息需要上百万年才能破解，自重的罪犯不会盗取它；他们只会入室盗窃或者偷走别人的包。

数字加密和数字解密是上述理念的延伸，即加密在计算机之间传送的数字信息，而软件可以完成这方面的全部工作。数字加密的级别一般表示为位强度(bit strength)：128 位加密是目前最低级的加密，1024 位加密是比较理想的。

数字密钥基于素数，因此破解它们很困难，因为需要找出素数因子，而这是一个难度很大的过程。密钥本身使用数字证书来发布，在可信任的证书权威机构的帮助下，证书可以验证密钥，这样就不会被黑客欺骗了。

下面说明了前面四个安全方面如何使用密码技术实现：

- 私密性：在数字加密和解密中，密钥的安全发布是使用公共和私有密钥对、证书和证书权威机构来实现。
- 验证：这需要使用证书和证书权威机构证明密钥的来源；粗略说，如果可以成功地解密信息，而且知道密钥的来源，就表示信息必然来自与密钥相同的来源。
- 不能反驳的信息：验证需要证明密钥的来源，所以，一旦验证了一段信息，这个验证就是不能反驳的。
- 完整性：首先，把加密的信息和未加密的信息都传送给客户机。然后，客户机解密其中已加密的信息，并把结果与未加密的信息进行比较——显然，两者应匹配(它们不匹配的几率很小)。因此，可以相信接收到正确的信息。

上述完整性检查有两个问题：第一，发送了未加密的信息；第二，信息发送了两次(加密的版本和未加密的版本一样大)。通过安全线路来发送信息，就可以解决第一个问题。根据消息摘要进行优化，就可以解决第二个问题——摘要是使用不可逆算法从信息中生成的一个小位串。我们发送的是加密的摘要，而不是加密的信息。最终结果是进行了相同程度的完整性检查，但信息只发送一次(为便于记录，加密的摘要称为数字签名)。

## 8.6.2 一般安全规则

与并发一样，在设计安全系统时，必须坐下来，仔细考虑安全问题，以确保系统不被误用。

下面是保护(联网)系统时要注意的事项:

- 防止未经授权就访问服务器,无论是无意还是恶意。
- 界定内部网络的敏感信息: 敏感信息包括与其他公司交易的业务信息; 业务策略; 个人电子信息; 信用引用机构的信息; 与国家安全相关的信息等。
- 防止盗取导出的信息: 确保在内联网外部传送的信息只能由指定的接收者读取。
- 保护员工和顾客的密码,这不仅是整个安全策略的基础,它们还是高度个性化的。
- 防止服务器代码访问不需要的资源。

● 防止客户机代码访问不需要的资源: 防止未经授权就访问客户的资源,防止客户受到无意的伤害(因为我们希望提供高质量的服务,而不希望客户控告我们)。如果您足够勇敢,就可能考虑雇佣讲究合乎道德的黑客、顾问,他们会尽力闯入您的系统。一旦讲究合乎道德的黑客证明您的系统可靠,就不必太担心系统的安全了。

## 8.7 分解软件

对于任何大业务,把所有的业务实体和业务过程都放在一个软件系统中是不切实际的—结果必然是过于复杂,难以使用。我们可以把软件分解为系统,如果需要,再分解为自治的子系统,最后分解为层(也可以称为子系统)。

### 8.7.1 系统和子系统

考虑 Customer 这个简单概念在大公司的各个部门中如何看待,例如销售、市场、会计、采购、发货等部门。如果试图用一个软件系统支持所有这些部门, Customer 就会有上百个属性和上百个操作: 这肯定是一个灾难。

业务应有许多不同的系统,每个系统都由不同的开发小组来实现,这样重用对象不当的机会就会降到最低。然后,信息需要在系统之间传送,就应通过设计优良的接口以定义明确的受控方式来传送。为了进一步降低复杂性,每个系统还应分解为各个子系统。

图 8-7 把一个公司的系统显示为树林中独立的树。在每棵树的下面是系统需要访问的数据库,在树的顶部是用户界面。通信沿着狭窄的受限路径(索桥),通过目的明确的界面,从业务逻辑到业务逻辑地进行。每个系统都有自己独立的数据,但仍可以使用一个 DBMS 进行部署,因为 DBMS 很适合管理多个数据库。

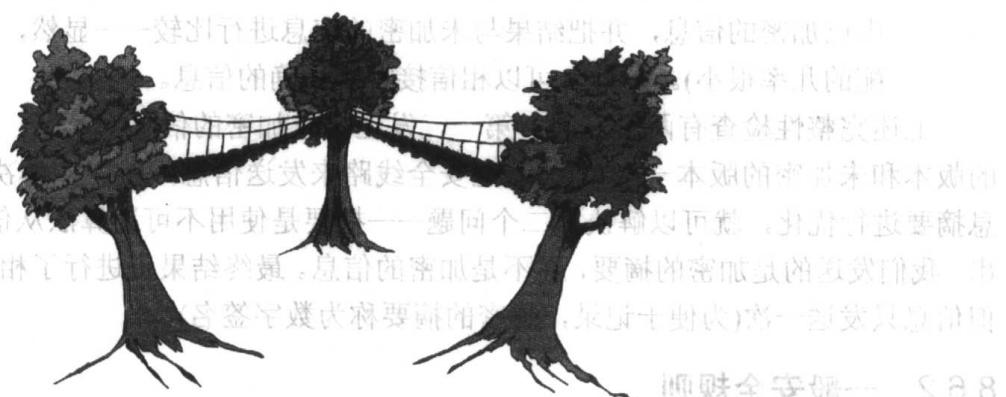


图 8-7 多个系统的合作

在案例分析中，有一个系统 Coot，它由两个子系统组成：iCoot 为会员和非会员提供访问，另一个子系统为助手提供访问，且类似于已有的 Auk 界面。后一个子系统不详细介绍。

## 8.7.2 层

在软件系统中，通常采用代码的多个层(如图 8-8 所示)。每一层都是一组合作对象，它们依赖于底层提供的功能。层不必包含对象，例如，Unix 系统库通过 C 函数层提供对底层操作系统功能的访问。

层将实现过程分解为多个可管理的块，有助于降低复杂性。层还提高了重用的可能性，因为每一层的编写都独立于其上的层。正如 Bertrand Meyer 在[Meyer 97]中所说：

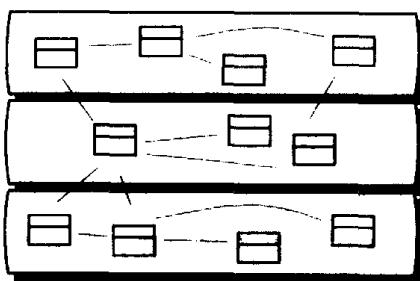


图 8-8 层中的对象

严格的软件系统，即使是按照今天的标准的小系统，也会涉及非常多的领域，所以无法在一层上处理所有的组件和属性，从而保证其正确性。需要一种多层次方法，每一层都依赖其下的层。

无论层的总数有多大，最顶层都常常表示用户界面，最低层表示操作系统或网络连接。为了简单起见，省略了最低层，只显示到一个众所周知的层上，该层在其他地方介绍。在下面几节中，将介绍有一、二或三层的系统中使用的常见层。

层可以打开(显示下层上的一些对象，供上面的层使用，例如管理下层中的对象，但没有完全隐藏它们)，也可以关闭(完全封装下面的层，例如上面的层完全看不到底层的对象)。确定某一层是应打开还是关闭不容易，这需要使用技巧、判断力、经验和预测。一般说，关闭的层需要更多的编码，运行也比打开的层慢(因为信息需要更多的复制和转换)。另一方面，打开的层一般不太安全(因为下面的层是未受保护的)，较难维护(因为每一层的上面都有多个依赖它的层)。

在某种程度上，可以交换各个层，而不会伤害其他代码。例如，总是可以去掉最顶层，用另一个层代替。而且，可以用一个有相同接口的层替换关闭的中间层，而不会影响上面的层。最常见的是，把系统移植到另一个平台上时，最底层会重复实现，把用户界面移植到新设备上时，顶层会重复实现。

下面论及的一些技术将在第 9 章详细论述。

### 1. 单层系统的层

图 8-9 显示了单层系统采用的一个简单的多层次模式。在最下面是数据库层，其任务是在

DMBS 和业务层之间来回传送数据。假定大多数应用程序都有数据存储的需求，如果系统因某种原因结束，数据就不会消失。如果一个更简单的系统在文件中存储数据，数据库层就是一个文件系统，而不是 DBMS。在数据库层的上面是业务层，它由实体对象和支持实现的对象组成。最后，在业务层的上面是用户界面层，它包含的对象负责把可用选项显示给用户，把用户命令和数据传送给业务层，并显示从业务层传来的数据。

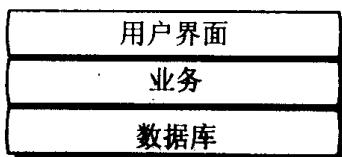


图 8-9 单层系统中的层

## 2. 两层和三层系统中的层

两层和三层系统使用网络从运行在客户机上的用户界面到达在服务器上运行的业务层。这可以使用另外两个层来实现，如图 8-10 所示。网络层包含的对象使网络对用户界面来说是透明的。就用户界面而言，它只能直接访问服务器对象。服务器层包含的对象把业务层的使用简化为一组可管理的业务服务。除了简化客户机之外，这还可以本地化服务器层上的安全措施，容纳不同种类的客户机，而不会影响业务层。

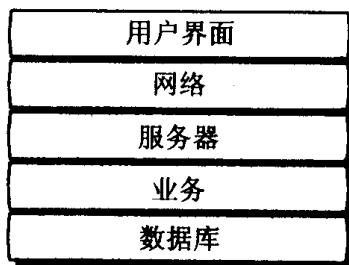


图 8-10 两层和三层系统中的层

对于两层系统，数据库层、服务器层和业务层都位于相同的机器上。对于三层系统，数据库层横跨网络，但 DBMS 隐藏了信息。

如果使用 HTML 窗体从客户机到达中间(或服务器)层，用户界面—网络情况就不太清晰。在这种情况下，用户界面部分在客户机上(如 HTML 页面和窗体)，部分在服务器上(如服务小程序和 JSP)。

## 3. 转换层

如图 8-11 所示，不同的层有不同的关注点。例如，在设计用户界面时，关注的是菜单、对话框、笔记本、窗口、可用性、直观性等。而对于网络，关注的是协议、带宽和不同类型的主机。移动到服务器上，就应关注安全、多线程和通过量。在业务层上，这部分最直接从业务分析中获得，所以最感兴趣的是抽象、属性、操作、多态性、重用和面向对象建模的其他基础方面。最后是数据库层，传统上希望处理的是键、表、SQL、锁定、函数依赖性和数据库理论

的其他方面。简言之，如果直接把这些不同的层连接在一起，结果会过于复杂和耦合(强耦合，一个对象的实现与另一个对象的实现紧密相关，将使代码难以维护)。

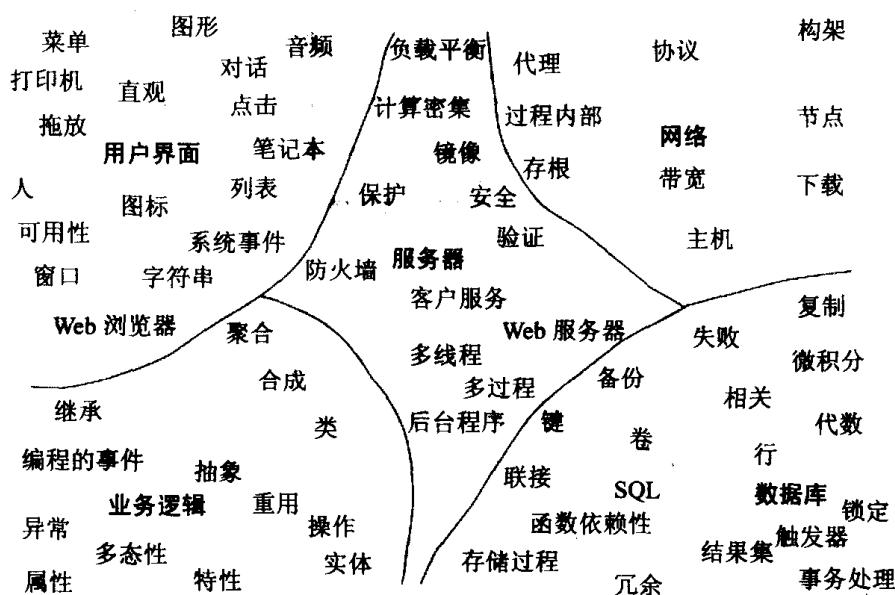


图 8-11 大型系统中的不同关注点

可以引入额外的层，作为转换器，来降低复杂性和耦合性。转换层对于把业务层(单层系统)或网络层(多层系统)转换为最终用户需要的最低功能尤其有帮助——这种层常常称为控制器。控制器管理着用户界面与系统其他部分的通信(这非常符合 Jacobson 的控制器概念)。另一个常见的转换层是所谓的持久层，它位于业务层和数据库层之间，去除了业务层对所使用的存储机制的依赖，以后改变存储机制就比较容易了(例如从文件改为 DBMS)。图 8-12 显示了增加了控制层和显示层的多层系统。

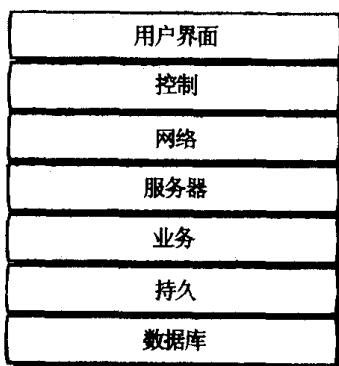


图 8-12 多层系统中的转换层

### 8.7.3 Java 层：应用小程序和 RMI

为了演示有适当 GUI 的 Java 客户，图 8-13 显示了一个三层系统的完整层集，该系统把 RMI 小程序作为其前端。RMI 是一个 Java 网络协议。

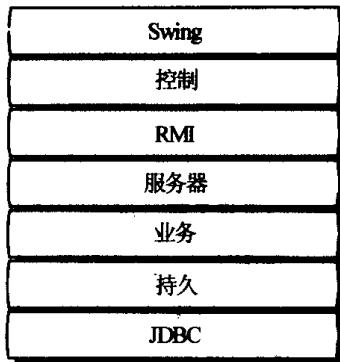


图 8-13 RMI 应用小程序层

在这个图中，用户界面层是使用 Swing 库(Java GUI 组件的可移植库)实现的。在用户界面层的下面是控制层，它包含访问业务服务的所有代码；这些代码必须隐藏在用户界面对象中，以后添加的每个新界面(例如移动电话)都要重新实现它。

95%的网络层都是 RMI 框架提供的——我们只需遵循控制层和服务器层中的几个简单规则，就可以在任何客户机上访问服务器对象。

服务器层、业务层和持久层与上一小节描述的类似。最后是 Java 数据库连接(JDBC)库提供的数据库层。JDBC 允许使用动态或预编译的 SQL 访问任意关系数据库。图 8-13 包含了持久层，所以可以用面向对象的数据库或文件系统代替 JDBC，而不会伤害业务对象。

## 案例分析

### iCoot 层

为了演示 HTML/CGI 和服务程序的配置，图 8-14 显示了用于 iCoot 的层。

对于 iCoot，没有持久层，因为通过 JDBC 访问的关系数据库用于在系统的整个生命周期中为系统服务。JDBC 层本身由标准 Java 库中的类提供。对于用户界面，有两种模式：CGI 和服务程序、RMI+小程序。对于第一个版本，将在 ServletsLayer 上使用 HTML/CGI 和服务程序。对于以后的版本，将使用 SwingLayer 为桌面提供 RMI+小程序机制。小设备，如 PDA 和移动电话，则使用 MicroLayer。MicroLayer 是使用 Java 2 Micro Edition(J2ME)实现的。

对于第一个版本，控制层将在 JSP 的帮助下由服务小程序提供。对于 RMI 版本，为适当的 GUI 设计了 ControlLayer。这意味着控制层在 HTTP/CGI 网络的下面，在 RMI 网络的上面。在这两种情况下，控制器会转换(或协作)Serverlayer。因此，ServerLayer 下面的所有内容都可以不加丝毫修改地重用(ControlLayer 也由 GUI 的两种模式重用)。

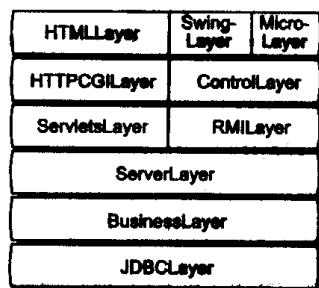


图 8-14 iCoot 的层

#### 8.7.4 层中的消息流

在分层的系统中，每一层都是该层下面一层的客户。因此，消息应从上面的层流向下面的层，如图 8-15 所示。每个消息都是一个问题(检索某种信息，例如 `getAddress`)或一个命令(做某个工作的指令，例如 `setAddress`)。

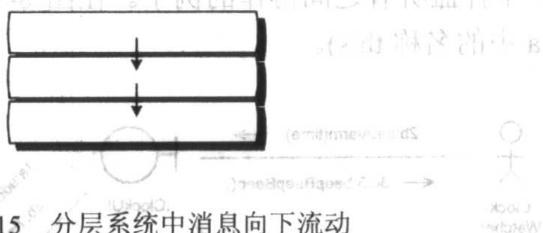


图 8-15 分层系统中消息向下流动

许多发送到一层中的命令都对该层管理的信息有影响——否则，发出该命令就没有任何意义。但是，如果上面的层需要知道改变了什么信息，该怎么办？例如，上面的层是用户界面，它需要用新信息更新显示，此时有两个选择：

- 给上面的层添加信息，说明哪个命令改变了什么信息。
- 只要信息有变化，就让下面的层给上面的层发送消息。

第一个选项的问题是，它会用逻辑上属于下层的信息影响上面的层——使上层的编码更复杂，上面的层与下面的层的耦合更紧密。第二个选项的问题是，下面的层必须对上面的层有所了解，才知道哪个对象发送了消息，所以，下面的层会受到上层信息的影响，使下面的层更复杂，也使它与上面的层的耦合更紧密(理想情况下，下面的层根本不应与上面的层耦合起来)。

#### 1. 事件

当发生了某个有趣的事件时，一层可以采用某种方式通知其上面的层，而不在两个方向上增加复杂性或耦合性吗？答案是“是的，事件”。图 8-16 显示了用于事件的真实类比。事件源检测到发生了某个有趣的事件时，就向监听的人(事件监听者)大声说出其细节(广播)。事件可以是属性事件，它表示事件源的一个属性值发生了变化，事件也可以是与属性值无关的纯事件。例如，当时间发生变化时，`Clock` 实体可以每秒广播一个属性事件，当发出警报时，它可以广播一个纯事件。

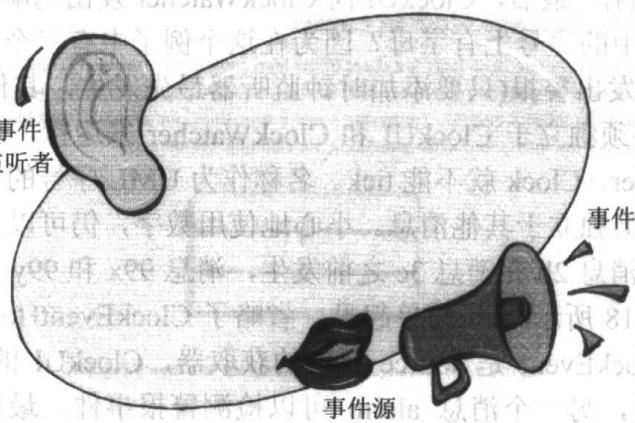


图 8-16 事件类比

正如“广播事件”所示，我们希望事件源检测事件的发生，并告知正在监听的人，这样，事件源就不会受监听者的影响，事件监听者也不会受到事件何时发生的影响。在分层模式中，可以在每一层使用事件源向上面一层的监听者广播事件，达到把信息保存在正确的位置和最小化耦合的目的。

但这如何通过消息来实现？第一步是理解其工作原理，为此，图 8-17 显示了一个 Clock、事件源、ClockUI、事件监听者之间协作的例子。在图 8-17 中，名称 self 是当前对象的 UML 表示法(它对应 Java 中的名称 this)。

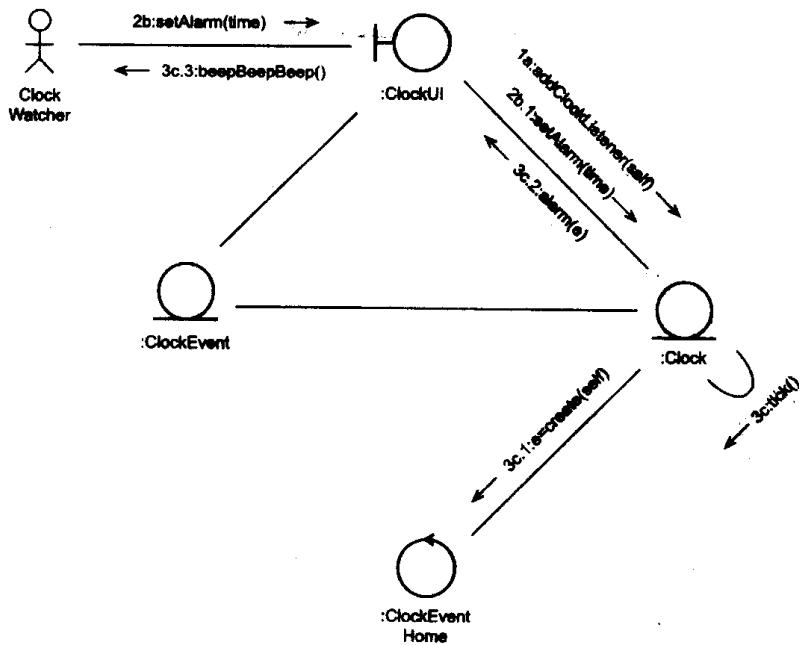


图 8-17 显示事件工作原理的通信图

作为初始化过程的一部分，ClockUI 给 Clock 发送 addClockListener 消息，注册它自己。接着，ClockWatcher 在 ClockUI 上设置警报，把警报设置传送给 Clock。Clock 定期给它本身传送 tick 消息，最终会发现该发出警报了。此时，Clock 创建一个 ClockEvent(在 ClockEventHome 的帮助下)，并给出事件的信息(这里惟一的信息是事件源)。接着，Clock 给 ClockUI 发送 alarm 消息，并把 ClockEvent 作为参数(事件对象记录事件源，如果 ClockUI 监听着多个 Clock，就可以确定是哪个 Clock 在响)。最后，ClockUI 向 ClockWatcher 发出鸣响。

为什么这个通信图中的序号上有字母？因为在这个例子中有三个相互独立的顺序：添加时钟监听器；设置时间；发出警报(只要添加时钟监听器最先发生，其他两个事件就可以以任意顺序发生多次)。时钟必须独立于 ClockUI 和 ClockWatcher 来运行，否则，在设置警报后，控制就会返回 ClockWatcher，Clock 就不能 tick。名称作为 UML 序号的一部分说明，消息依赖于包含相同名称的消息，但独立于其他消息。小心地使用数字，仍可以显示独立消息的顺序，例如，在上面的例子中，消息 2b 在消息 3c 之前发生，消息 99x 和 99y 同时发生。

闹钟的类图如图 8-18 所示(为了简单起见，省略了 ClockEventHome 和 ClockWatcher)。从这个图中可以看出，ClockEvent 是 source 属性的获取器，ClockUI 的一个消息 setAlarm 允许 ClockWatcher 设置警报，另一个消息 alarm 可以检测警报事件。最后，Clock 类的一个消息 setAlarm 用于设置警报，另一个消息 addClockListener 用于注册监听者(这个图中有一些新的 UML 表示法，稍后介绍)。

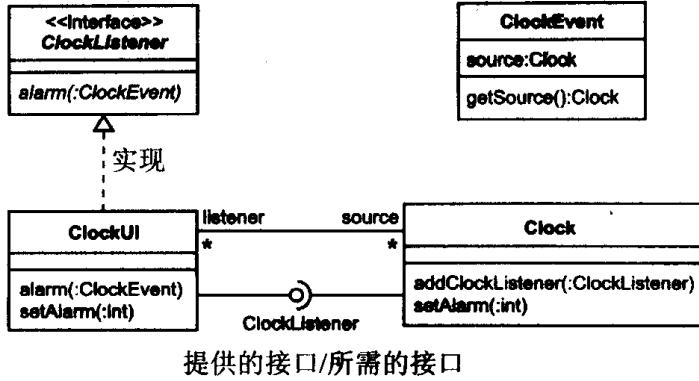


图 8-18 描述为类图的事件

面向对象的语言一般不必有真正的广播机制，所以图 8-18 中的 Clock 必须维护监听者的内部列表，当发生事件时，还必须给每个监听者发送消息。监听者必须确保它们为事件进行了注册。Clock 现在和 ClockUI 耦合在一起，这是我们应避免的。为了解决这个问题，可以引入一个抽象类 ClockListener，它只列出检测 Clock 事件所需的消息。只要 ClockUI 继承了 ClockListener，就可以用 Clock 注册 ClockUI，ClockUI 也可以接收 alarm 消息。因此，即使 Clock 和 ClockListener 耦合在一起，也不与 ClockUI 耦合在一起。(ClockListener 位于与 Clock 相同的层上，而 ClockUI 在该层的上面)。

用<<interface>>关键字表示的接口是一个纯抽象类，这种类没有具体的方法，也没有属性。接口可以用降低了的耦合性指定通信，所以在图 8-18 中有一些特殊的 UML 表示法。虚线白箭头(标记为“实现”)表示继承，特殊情况下，超类是一个接口。ClockListener 表示法标记为“提供的接口/需要的接口”，它允许表示一个类通过特定的接口使用另一个类。在本例中，看起来像棒棒糖的部分表示 ClockUI 实现(一种)ClockListener，而杯子形状的部分表示 Clock 只依赖于 ClockUI 是一种 ClockListener 这一点。图 8-18 中有多余的地方，在自己的图中使用“虚线的继承”或“棒棒糖和杯子”模式时，要注意这一点。

## 2. 使用事件的消息流

图 8-19 显示了在使用层时消息如何流动。一般的消息都是在层中向下流动，而事件消息是向上流动的(事件消息显示为虚线箭头，表示下面的对象对接收者一无所知)。

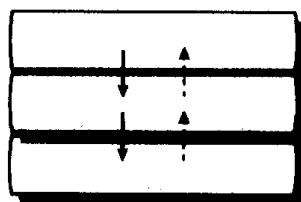


图 8-19 使用事件的控制流

在客户代码中最常使用事件，因为这是使用户界面用下面一层中的信息更新的简便方式。事件很少在服务器端使用，因为服务器代码是多线程的，会显著增加事件驱动的编程的复杂性(例如，出现死锁的可能性会增加)。事件不应通过网络来广播：在给许多客户发送事件消息时，如果一些客户已毁坏或很难联系上，就不希望服务器阻拦。

因此，建议避免在服务器端进行事件驱动的编程。如果要通过网络来广播，应使用机器对机器的消息传输方式，例如 Java Messaging Service(JMS)提供的方法。

## 8.8 小结

本章的主要内容如下：

- 系统设计的步骤，系统如何分解为物理和逻辑组件，特别关注了网络拓扑。
- 如何在 UML 部署图上演示体系结构决策。
- 联网系统中的并发问题：如何确保信息完全更新后，他人才能访问它，如何确保在读取信息的过程中不更新信息。
- 如何通过保证私密性、验证、不能反驳的信息、完整性和授权，来确保系统不被无意或恶意误用。
- 公司的软件如何分解为多个系统、子系统和层。

## 8.9 课外阅读

HTML 规范的完整信息可参阅[W3C 99]，更容易理解的描述可参阅[Raggett 等 97]，Raggett 是 UML 标准的主要领军人物之一。

密码学的更多信息可参阅[Singh 00]，这本书由 Simon Singh 编著，Simon Singh 是一位科学新闻记者，而不是理论家。

## 8.10 复习题

1. 为什么层在子系统设计中非常重要？(多选题)
  - (a) 更容易改变实现方式
  - (b) 减少了实现代码中类的数量
  - (c) 提高了重用性
  - (d) 降低了复杂性
2. 如果两个顾客在世界的不同地方，要购买音乐会的最后一张票，如何分配这张票？(单选题)
  - (a) 引入一个额外的业务规则，把可用票的查询和临时预订合并起来。
  - (b) 使顾客参与软件“竞争”，以买到票。
  - (c) 不允许卖出最后一张票，因为这对其中的一位顾客是不公平的。
3. 如图 8-20 所示，Z 是什么？(单选题)
  - (a) 类
  - (b) 事件
  - (c) 接口
  - (d) 边界
  - (e) 属性

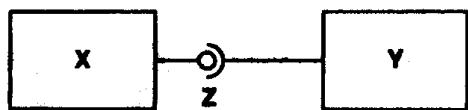


图 8-20 复习题 3

## 8.11 复习题答案

1. 层在子系统设计中非常重要的原因是：
  - (a) 更容易改变实现方式
  - (c) 提高了重用性
  - (d) 降低了复杂性
2. 如果两个顾客在世界的不同地方，要购买音乐会的最后一张票，好的方法是 (a) 引入一个额外的业务规则，把可用票的查询和临时预订合并起来。
3. 如图 8-20 所示，Z 是 (c) 接口

# 第9章 选 择 技 术

本章介绍客户端和服务器端的主要可用技术，让读者了解如何进行正确的选择。

**学习目标：**

- 理解客户端的技术
- 理解服务器端的技术
- 理解连接客户和服务器的协议
- 理解网络技术

## 9.1 引言

即使有了完整的需求和分析文档，甚至有了最初的体系结构图，仍无法选择要使用的实现技术。如前所述，这个决策拖的时间越长，系统对未来变化的敏感程度就越小。另一方面，在选择技术之前等待的时间越长，利用这些技术的优点的机会就越小。

在开发的这个阶段(详细设计之前)，选择技术是一个很好的折衷方案。一旦做出了决策，就可以做出正确的选择，同时确定对特定技术的独特特性的依赖程度。本书中的所有图都采用独立于语言的 UML。例如，这些图使用了标准的 UML 操作签名，而不是使用某种更特殊的方式，例如 Java 语法。相反，在图中显示 Java 原型，而不是 UML 原型，因为它们更紧凑，更清晰地说明我们使用的是原型值，而不是对象值。

## 9.2 客户层技术

下面看看在多层系统上运行在客户机上的软件(本书的重点)。主要有两个选择：可以加载特定目的的应用程序(程序、可执行代码，也可以是其他名称)或驻留客户软件的 Web 浏览器。可以运行的客户应用程序有：

- 人对人的通信：电子邮件、即时消息传输、USENET 新闻、聊天
- 文件传输或文件交换
- 远程登录
- 专用应用程序(不适合更一般的类别，例如多用户飞行模拟器)

驻留在 Web 浏览器上的客户可以使用如下技术：

- HTML 窗体
- JavaScript
- 专用插件(不适合更一般的类别，例如使用 Flash 的交互式动画)
- ActiveX 控件
- Java 小程序

上述技术都使用某种协议(例如电子邮件使用 IMAP, 即时消息传输使用 AIM, HTML 窗体使用 HTTP/GUI)与至少一个机器(邮件服务器、消息传输服务器或 Web 服务器)通信。一般说来, 应用程序和浏览器插件可以用任意语言编写(Java、C++、Eiffel、Fortran 或 COBOL)。应用程序总是这样; 对于浏览器插件, 需要所选语言的编译器, 把机器码转换为 DLL(Windows)或共享库(Unix)。

客户应用程序的缺点是, 需要在客户机上安装软件, 之后才能使用。但是, 对于一些情形(例如桌面发布), 这是一个很好的选择。Web 浏览器本身是一个应用程序, 它尤其适合于编写为客户软件, 因为多层开发人员可以改进它, 以浏览器开发人员做梦也想不到的方式运行。例如, Web 浏览器可以用于从全球广播公司处读取新闻标题, 作为 HTML 页面, 还可以使用 Java 小程序检查银行账户的信息, 这些功能都不需要事先在客户机上安装(当然浏览器本身除外)。

对于用 Java 编写的大型客户层, 通过网络加载小程序是不切实际的——在这种情况下, 需要在本地安装小程序(或 Java 应用程序)。8.4.4 节认为, 无论通过网络加载的应用程序有多大, 使用网络计算机都比较好。实际上这是相当困难的。一般情况下, 网络计算机使用互联网技术(通过内联网)来获得软件。就其本质而言, 互联网模式的网络有一个瓶颈: 频繁加载大程序是不太可能的。这个问题可以使用好的本地缓存来缓解, 但网络计算机的最佳类型仍是无磁盘的 Unix 工作站, 它的瓶颈问题要小得多。目前, 这意味着: 不要通过互联网加载大程序; 如果要通过内联网加载程序, 应确保该程序不是特别大(<2MB)。但是不要忘了, 瘦客户 GUI 可以压缩为 100KB 的 Java 小程序, 它甚至可以在互联网上加载。

每个浏览器技术都有其优缺点。例如:

- HTML 有非常丰富的可视化效果, 获得了广泛的支持, 但 HTML 窗体是比较原始的, 不能自动在客户机上验证。另外, 人们必须在数十个页面上查看信息, 而且在绘制页面前, 每个页面都要显示为空白, 这不是交互的最佳方式。
- JavaScript 允许一些客户端的编程(例如 HTML 窗体上的数据验证)。但 JavaScript 是解释性的(所以比编译性的技术慢), 不纯粹(从面向对象的观点来看), 不同的浏览器提供不同级别的支持(导致编码困难)。
- 在理论上, 插件可以提供任意类型的客户交互。但是, 插件常常需要先下载和安装, 才能使用; 每个插件都需要不同的编程技巧, 提供者必须为每个新平台(操作系统/CPU 组合)移植插件, 所以, 不是每个客户都支持它们。
- ActiveX 控件是位于 Web 浏览器上的 32 位 Windows 二进制代码, 这使 ActiveX 控件具有和插件与 Java 小程序一样的优点, 但它们只能运行在 Windows 上。
- Java 是一种简单、纯粹、面向对象的语言, 提供了最佳的解决方案。Java 还提供了安全措施, 当用户没有显式、详细的授权时, 它阻止用户访问本地机器上的资源。这部分因为有太多的合作诈骗, 在编写本书时, 大多数 Web 浏览器只支持 Java 的旧版本。要获得 Java 的完全版本, 需要 Sun 提供的 Java 插件(Java 化装为一个浏览器插件), 或者需要购买预装了 Java 的 PC。

在原则上, Java 是在三层系统中实现客户端的最佳选择。因为在 Web 浏览器中缺乏杰出的支持, 大多数开发人员都选择使用 HTML 窗体(公平地讲, 浏览器支持的问题也使其他客户选项苦恼, 例如插件和 ActiveX——HTML 窗体是唯一可行的, 因为它们已存在了那么长的时间)。许多 Web 浏览器也可以驻留旧的客户应用程序, 包括文件传输器、电子邮件和 USENET 新闻。

大多数重要的技术都移植到较新的设备上，例如个人数字助手(PDA)和移动电话。TCP/IP 是普遍适用的，甚至在小型客户设备上也适用，它还可以在无线连接上工作。因此，把三层系统移植到新设备上，一般只需重新设计用户界面，使之更小(更原始)。

### 9.3 客户层到中间层的协议

客户软件无论是作为应用程序运行，还是在 Web 浏览器上运行，都必须使用某种协议与服务器通信。大多数协议都是分层的：在底层有一个底层协议，例如 TCP/IP，在顶层，要建立更多的协议，专门用于特定的任务。例如，在 TCP/IP 的顶部，可以放置加密和解密信息的安全套接字层(Secure Sockets Layer, SSL)，以保证私密性和完整性。在 SSL 的顶部，可以运行安全 HTTP(Secure HTTP, HTTPS)，这是一个安全协议，允许客户通过 URI 请求文档，得到该文档的内容。

有多个层是相当合理的。例如，Java 有一个机制，叫做远程方法调用(Remote Method Invocation, RMI)，它允许对象给运行在不同机器上的另一个对象发送消息——消息使用 Java 远程方法协议(Java Remote Method Protocol, JRMP)发送。为了通过互联网防火墙，RMI 准备在需要时背负 HTTP。所以，当一个对象给另一个对象发送 RMI 消息时，会得到如图 9-1 所示的运行时情形——图 9-1 说明一个消息使用 JRMP 编码，接着使用 HTTP，然后是 SSL，最后是 TCP/IP；在服务器端，给消息解包，发送给接收者。回应通过相反的路径发送回来。

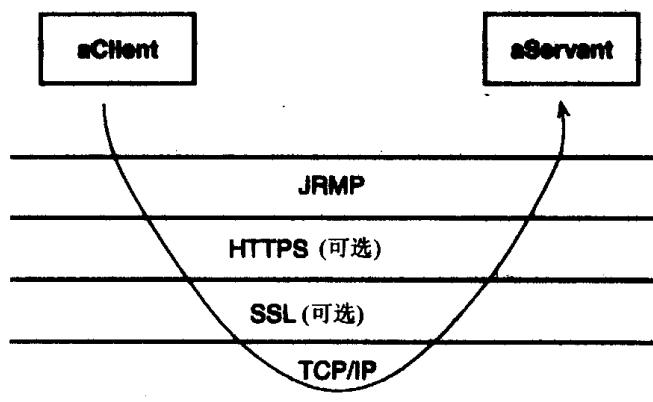


图 9-1 RMI 使用网络的方式

常用的协议分为两类：专用协议和通用协议。专用协议包括：

- IMAP(电子邮件)
- AIM(AOL 即时消息传输)
- NNTP(USENET 新闻)
- HTTP/CGI(HTML 窗体)
- FTP(文件传输)
- Telnet(远程登录)

通用协议(适用于许多任务)包括：

- TCP/IP(低级传输，也称为套接字)
- JRMP(用于 Java 对 Java 的通信)
- IIOP(用于 CORBA 通信，类似于 RMI，但有多种实现语言)

开发人员常常使用运行时系统和库的高级抽象。例如，RMI 和 CORBA 程序员只给对象发送消息，RMI 和 CORBA 代码完成所有的打包和解包任务。HTML 窗体的程序员只设计窗体的布局，公共网关接口(Common Gateway Interface, CGI)机制负责把窗体数据传送给服务器上的可执行代码。

## 9.4 中间层技术

介绍了进入服务器第一线的方式后，就需要决定从那里接管什么类型的软件。服务器应用程序一般是多线程的代码，是为高通过量(能同时处理上千甚至上百万个客户)设计的。服务器应用程序监听某些客户连接(connect)的端口(port，连接点)。

就软件的风格而言，服务器类似于客户端。在客户端，可以运行驻留在 Web 浏览器上的独立的应用程序或代码。在(中间层)服务器上，可以运行独立的应用程序，或者运行 Web 服务器，把代码放在 Web 服务器上。独立的应用程序包括：

- 邮件、消息传输、新闻和聊天服务器
- FTP 后台程序
- Telnet 后台程序
- RMI 注册表(RMI 对象的查找机制)
- CORBA 命名服务(CORBA 对象的查找机制)
- Java 命名和目录接口(JNDI)服务器(通用的命名映射服务，可以用于替代 RMI 注册表、CORBA 命名服务、用户注册表等)
- 专用服务器(例如，驻留 CORBA 或 RMI 对象的过程、EJB 客户程序、.NET 客户程序)  
可以驻留在 Web 服务器上的服务器代码包括：
  - Java Server Pages(JSP)，用于随时建立 Web 页面
  - Active Server Pages(ASP)，类似于 JSP，但编码一般是 Visual Basic，而不是 Java
  - CGI 脚本(这些可以是解释性的文件，用 PERL 等语言编写，或者是可执行的程序)
  - 服务小程序(Java 服务器对象，可以由 Java 小程序、JSP 或 HTML 窗体访问)

RMI 注册表和 CORBA 命名服务允许 RMI 和 CORBA 客户根据名称查找服务器对象(或者使用 JNDI)。

CGI 脚本是用某种命令语言(如 PERL)编写的文本文件，或者是自然的可执行代码，以自然的方式从编程语言编译而来。CGI 脚本一般由 HTML 窗体调用；Web 服务器从窗体中提取数据，传送给 CGI 脚本，作为环境变量；脚本把结果放在标准输出中，返回给客户浏览器，根据浏览器的类型显示结果(HTML 页面、图像、音频文件等)。

服务小程序(Servlet)是 Web 服务器按照要求实例化的 Java 对象。服务小程序通常传送 HTML 窗体中的数据，以进行处理；与 CGI 脚本一样，服务小程序返回的结果指定用户在浏览器上看到的内容。服务小程序是独立于平台的，很快会替代 CGI 脚本。

JSP 是文本文件，包含了散布着 Java 代码的 HTML。在第一次调用时，JSP 会转换为服务小程序，原始的 HTML 用打印的语句替换，其中包含 Java 代码；再编译和调用服务小程序。JSP 一般用于使 Web 页面个性化——例如为当前登录的顾客插入某个语句。它们可以直接调用或者通过服务小程序来调用。

ASP 类似于 JSP，但 ASP 使用 Microsoft 技术，所以移植性不是很好。

## 9.5 中间层到数据层的技术

到目前为止，我们在客户端运行代码，通过某个协议进入中间层，在中间层调用一些代码。下一步该做什么？在三层系统中，通常的答案是访问数据层。下面是具体的方式：

- 在中间层上包含数据库—客户代码，以便访问运行在数据层上的 DBMS。一般使用 Java，可以在 Java 数据库连接(JDBC)机制[Campione 等 98]的帮助下访问 DBMS。
- 使用前面讨论的任一种客户—中间层技术与数据层通信。毕竟就数据层而言，中间层只是另一个客户。
- 做一些专门的工作，例如访问运行在数据层机器上的服务器，或直接在中间层上运行代码(二层配置)。
- 使用非 TCP/IP 协议访问数据层(通常只有在访问旧系统时才这么做)。
- 在中间层服务器上包含 Enterprise Java Beans(EJB)客户代码，然后通过 EJB 访问数据层(Java 对象一般用于在内联网上提供数据，处理服务)。
- 在中间层服务器上包含.NET 客户代码。.Net 框架是 Microsoft 应对 EJB 框架(和 J2EE 的其他部分)竞争的产品。本书使用的是 Java 例子，所以不介绍.Net。

图 9-2 显示了根据前面讨论的技术如何根据它们在三层系统中的正常位置组合在一起。

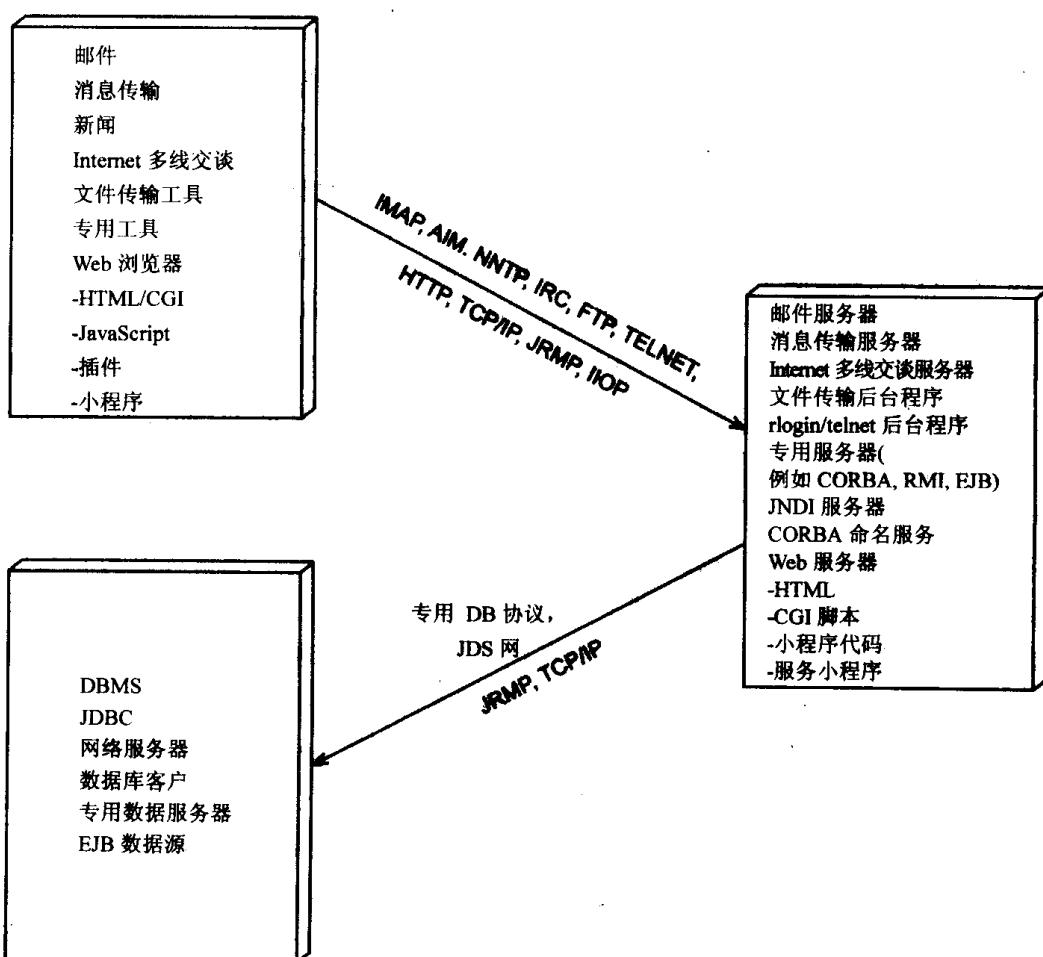


图 9-2 三层技术小结

## 9.6 其他技术

还有许多其他技术值得一提。前面没有讨论这些技术，是因为它们相对较新，可能不会存在很长时间，或者它们只提供了前面已讨论的技术的替代品。

- **身份验证：**用户通过网络访问系统时，最好验证一下该用户的身份。为此，可以开发自己的机制，也可以使用已有的技术，例如，服务小程序就可以构建一个 Web 浏览器，在继续之前先显示登录屏幕。但是，我们需要的是一种迫使用户登录并在整个系统(从 Web 浏览器到中间层服务，再到后端)中保持其身份的机制。这个理想目标称为单签署(如果要跨越其他域，就称为全球签署(global sign-on))。Java 的单签署(single sign-on)是 J2EE 的一部分。对于全球签署，新兴的技术包括 Microsoft .Net Passport ([www.microsoft.com](http://www.microsoft.com))和 Liberty Alliance 开发的技术([www.projectliberty.org](http://www.projectliberty.org))。
- **XML：**在联网系统中，常常需要把业务数据从一台机器传送给另一台机器。可扩展的标记语言(eXtensible Markup Language, XML)[Yergeau 等 99]是一种文本语言，它把数据描述为结构化的名称-值对。XML 文档是简单的字符串，所以可以通过为系统选择的任意协议来传输，例如 HTTP 或 RMI。在 XML 文档中，可以开发任意结构和名称-值对。对于 XML 来说，当一个文档到达另一台机器时，目标机器必须能识别出数据的含义，而不仅仅识别出其结构。例如，如果用值 NETAMOUNT 定义了 XML 文档，除非接收者知道该文档是一张发票，NETAMOUNT 是税前额，否则该文档就没有什么用。所以，为常见的业务信息定义 XML 文档类型时要遵循标准。即使没有标准的文档定义，XML 仍是有用的，因为它允许使用相同的库来分析、生成所有的文档(例如库可用于 Java)。XML 可以用于存储制品，传输这些信息，例如，可以以 XML 格式存储电子表格文档。
- **事件和消息：**事件是把信息广播给感兴趣的团体的一种常见机制。事件的问题是它们在网络上不能很好地发挥作用，因为很难使事件是线程安全的，其效率也很低(从服务器的观点来看)。通过网络广播的一种较好方法是使用机器对机器的消息传输(machine-to-machine messaging)。消息是可以给感兴趣的客户广播的信息块：一旦服务器启动了广播，就不再参与信息的传输，没有给特定的客户传输信息并没有什么不利影响(消息也可以用保证传输到的模式来发送)。消息传输的更多信息可参阅 Java Messaging Service(JMS)[Bodoff 等 02]。
- **SOAP：**简单对象访问协议(SOAP)[W3C 03]类似于 RMI 和 CORBA。其区别是，SOAP 是基于 XML 的协议，因此将来有可能成为事实上的标准。
- **Web 服务：**Web 服务的理念是顾客(在家中或在公司里)在胖服务器上存储和处理信息，并可以通过互联网或外联网访问，Web 服务是要付费的。Web 服务的一个简单例子是基于浏览器的电子邮件，较复杂的例子是数字照片的处理、管理和打印。Web 服务使用 TCP/IP 和与 XML 相关的技术，如 SOAP，部署为基于互联网的三层系统。更多的信息可参阅 Web 服务互操作组织([www.ws-i.org](http://www.ws-i.org))和 Sun 领导开发的 Java 相关服务([java.sun.com](http://java.sun.com))。

## 9.7 一般前端配置

探讨了可以在多层系统中使用的许多技术之后，下面看看在客户层和中间层之间通信的一些典型配置。在阅读下面几个标题时，注意从客户层到中间层是一个用户界面练习：把所有复杂的业务逻辑放在中间层，就可以根据个人喜好或可用的时间选择任意一种或所有前端配置。第 10 章将介绍如何设计中间层和数据层，以便随时进出前端。

### 9.7.1 HTML/CGI 和脚本

最初，HTML 是内嵌超链接的多媒体文档的一种简单页面标记语言[W3C 99]。Web 浏览器会话从用户输入目标 URI(如 <http://www.blueskyuniversity.com/salespitch.html>)，按下回车键(或选择书签或喜欢的事物)开始。接着，浏览器分析该 URI，发现 HTTP 用于连接 Web 服务器，并提取 Web 服务器的 IP 地址([www.blueskyuniversity.com](http://www.blueskyuniversity.com))。之后，浏览器联系该 Web 服务器，给它传送一个 HTTP 请求，以提取文件 salespitch.html。Web 服务器检索相应的文件，本例是一个 Web 页面，并把它返回给浏览器，显示出来。

HTTP 的基本机制很适合于浏览分布式多媒体文档，但没有给用户提供输入数据的方式，例如他们要购买哪本书。公共网关接口(CGI)解决了这个问题。利用 CGI，在浏览器中显示的 HTML 包含一个或多个窗体(文本字段、下拉列表和按钮)，用户可以在这些窗体中填充信息。用户只需在窗体中输入数据，单击按钮(一般是 Submit 或 Proceed to Checkout 按钮)。按下了按钮后，Web 浏览器就从窗体中提取出名称一值对，把它们以 CGI 脚本(命令文件，解释性文件或可执行文件)的名义传送给 Web 服务器，该脚本应处理数据。有关脚本和名称一值对的所有信息都可以由 Web 浏览器打包为一个 URI：

```
http://www.blueskyuniversity.com/cgi-bin/buy.pl?b=Gemma&q=2
```

(名称一值对也可以在 HTTP 请求体的中间，但最终结果是一样的)。

Web 服务器接收到请求后，就检测该请求是否需要运行过程，因为 URI 中有/cgi-bin 部分，这部分指定了服务器上保存该脚本的位置。Web 服务器找到要运行的脚本(在这里是名为 buy.pl 的 PERL 脚本)，启动它，然后给它传送名称一值对，作为环境变量(可以由过程检索的操作系统值)。在上面的购书例子中，有两个名称一值对 b=Gemma 和 q=2，分别表示书和数量。

脚本现在可以根据传送给它的值，进行它需要的任何处理(这里假定使用已输入的信息，传送两本书，并记入顾客的信用卡)。一旦脚本完成，它写入标准输出的所有内容就传送给客户浏览器，作为交互的结果。例如，结果可以是一个新的 HTML 页面、一张图片或一个音频剪辑。在上面的购书例子中，发送回的应是一个预订确认页面。

HTML/CGI 配置如图 9-3 所示。其中显示了 Web 服务器提供对 HTML 文件、脚本和其他媒体文件的访问，而客户机上的 Web 浏览器显示 HTML 页面，通过 CGI 窗体提供捕获到的数据。

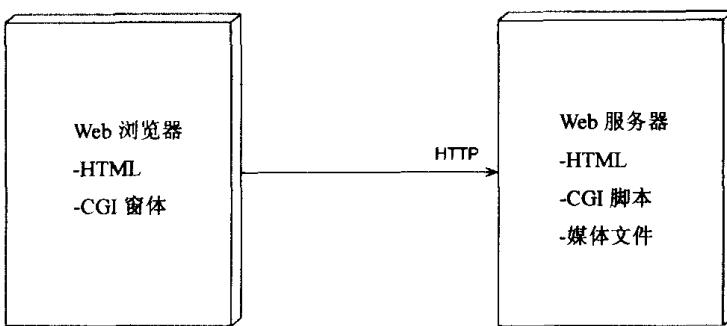


图 9-3 使用 HTML/CGI 和脚本进行配置

因此，CGI 允许从客户机上收集窗体数据，用于在中间层上处理。CGI 的主要优点是，无论客户机的 Web 浏览器有多老，都支持 CGI。所以，只要安装了 Web 浏览器，任何机器都可以成为三层系统上的客户机。

CGI 有许多缺点：

- 交互性差：CGI 窗体的输入组件有限，只有文本输入字段、列表、下拉列表和按钮。现代的专业用户要使用笔记本、工具栏、数字增减按钮、菜单等。使用客户端的脚本语言，如 JavaScript，可以缓解其中的一些问题，但开发人员要付出相当多的时间和努力。即使如此，最终结果也不理想。更糟糕的是，每次单击 Submit 按钮，都会把用户带到另一个页面，把整个浏览器窗口显示为背景色，以便重新刷新。相反，一般 CGI 的刷新要快得多。
- 速度慢：用户必须等待窗体的内容派送给服务器，开始处理，直到运行结束，结果才能返回并显示出来。于是，用户就有了交互操作慢的印象。
- 客户端上没有数据验证：可以在客户端使用 JavaScript，在派送之前验证窗体，以改进安全性。但是，仍必须在服务器上进行检查，以防止受到黑客的攻击，所以最好能进行客户端验证，提高传送给服务器的正确窗体的数量。
- 服务器过载：在服务器上，有一个严重的问题，因为我们试图为每个窗体启动一个过程。这个问题非常严重，它使我们不能同时处理上千个窗体。使用工具 FAST-CGI 可以在内存中保存过程，避免为每个请求启动一个过程，但仍需要为每个请求制作程序数据区域的单独副本。
- 脚本不能移植：在历史上，编程语言、脚本语言和命令语言都不是完全不能移植的。因此，如果决定修改中间层服务器窗体，例如从 Windows 改为 Unix，就需要做一定的移植工作。
- 不安全：在默认情况下，名称一值对传送给 Web 服务器时是未加密的，所以个人信息可以在传输过程中读取。为了避免这种情况，可以在 SSL 上运行 HTTP(在提交窗体之前，在 URI 的开头查找 https://)。但这是必须自己完成的额外配置(也可能要付费)。

### 9.7.2 HTML/CGI 和服务小程序

HTML/CGI 和服务小程序配置是 Sun 及其伙伴的一个聪明的设计，它几乎与 HTML/CGI 和脚本配置相同(如图 9-4 所示)。客户机有相同的 Web 浏览器、HTML 页面和 CGI 窗体，区别在 URI 上：不是指定要运行的脚本，而是指定要实例化的服务小程序(Java 对象)：

<http://www.blueskyuniversity.com/servlet/BuyServlet?b=Gemma&q=2>

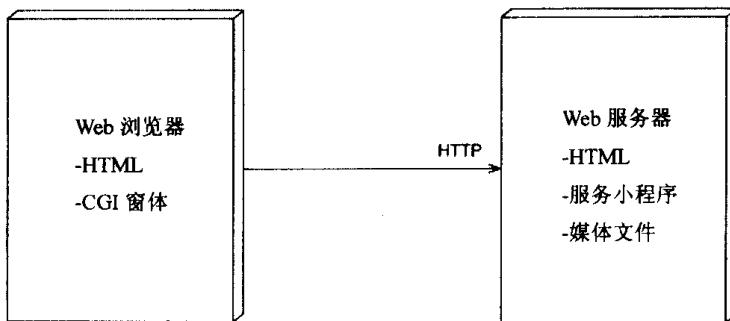


图 9-4 使用 HTML/CGI 和服务小程序进行配置

Web 服务器发现我们要运行一个服务小程序，因为 URI 中有 /servlet 部分。接着是服务小程序的类名(BuyServlet)。Web 服务器第一次接收到特定服务小程序的请求时，就实例化该服务小程序，并让它处理请求。之后，服务小程序仍旧存在，后续的所有请求都由这个服务小程序处理。(每个请求都在执行过程的不同线程中由服务小程序处理。所以，这里就不是把脚本运行为单独的过程，而是让 Web 服务器把多个客户请求传送给一个 Java 对象)。

为了把请求传送给服务小程序，Web 服务器使用几个标准消息之一，最常用的是 doPost (:HttpServletRequest,:HttpServletResponse)；名称一值对在 HttpServletRequest 的内部。当 doPost 完成其后端处理后，响应就放在 HttpServletResponse 中。Web 服务器提取 HttpServletResponse 的内容，以正常方式把它送回 Web 浏览器。

服务小程序比脚本优越的地方有：

- 性能：服务小程序无需多个过程或多个数据区域，就可以同时处理几个请求，所以对请求的服务效率就高得多。另外，服务小程序一般比用 PERL 等语言编写的解释性程序运行得更快。
- 可伸缩性：多个服务小程序可以由一个(Web 服务器)过程处理，所以可以同时管理上百万个客户请求，且不会使服务器超载。
- 可移植性：如果改变操作系统，惟一的移植工作就是在新平台上安装 Java 的适当版本。服务小程序保持不变。
- 易于使用：由于服务小程序是标准的 Java 对象，因此开发人员可以利用可重用的对象库，来完成常见的计算任务，所以服务小程序的编码量比脚本少。

### 9.7.3 RMI

远程方法调用(RMI)[Campione 等 98]是一个奇特的术语，表示“通过网络给其他 Java 对象发送消息的 Java 对象”。RMI 是标准 Java 库的一部分，所以只要安装了 Java 2 标准版(J2SE)或企业版(J2EE)，就可以使用 RMI。RMI 设计人员使 RMI 使用起来非常简单，使用远程对象和使用本地对象几乎一样简单。他们之间只有以下几点区别：

- 服务器对象必须部署在要通过 RMI 联系的机器上；它们可以提供远程客户机需要的所有服务，或是其他对象的生产车间。

- 驻留服务器对象的机器必须运行一个命名服务，这个服务器过程运行在一个著名的端口上，允许远程机器根据名称查找服务器对象。RMI 有自己的命名服务，称为注册表(Registry)，但最好使用可插入的 Java 命名和目录接口(Java Naming and Directory Interface, JNDI)，以利用命名服务提供的所有功能。
- 通过网络传送的所有消息都可能抛出一个异常 RemoteException，它确保客户程序员不会忘记他们是在本地 Java 过程的外部工作。

除了这些微小的限制之外，开发人员还可以使用 RMI 建立客户机—服务器或分布式体系结构。

RMI 配置上的每个主机都可以实现为小程序或应用程序。实现小程序的优点是，只要在网络的某个地方部署了 Web 服务器，客户就不需要事先安装系统软件——所有的客户软件都可以在需要时通过 Web 浏览器和 Web 服务器加载。而对于应用程序，必须完成下面所列的一项工作：

- 把客户软件部署到每个客户机上。只要更新了客户软件，就需要在所有的客户机上重新部署。这个选项不适合客户机上的系统软件比较大的情况(大软件就意味着依赖所使用的网络的速度)。
- 部署 Web 服务器，再使用小引导软件加载客户代码。引导软件仍需要安装在每个客户机上，但至少可以保证，客户机在每次启动时，都会得到软件的最新版本。
- 使用 Java WebStart 在需要时加载客户软件。Java WebStart 是 Sun 提供的一个工具，支持 Java 应用程序的动态加载(一般是为小程序保留的机制)。要使用 Java WebStart，需要部署 Web 服务器，再确保每个客户机上都安装了 Web 浏览器、Java WebStart 和 Java PlugIn(J2SE 的一部分)。如果要把客户机运行为应用程序而不是小程序，这是最简单的选项。

图 9-5 显示了 RMI 小程序客户机的前端配置。在中间层上，必须运行某种服务器过程，该过程包含与系统相关的代码和作为命名服务的一部分运行的通用代码。

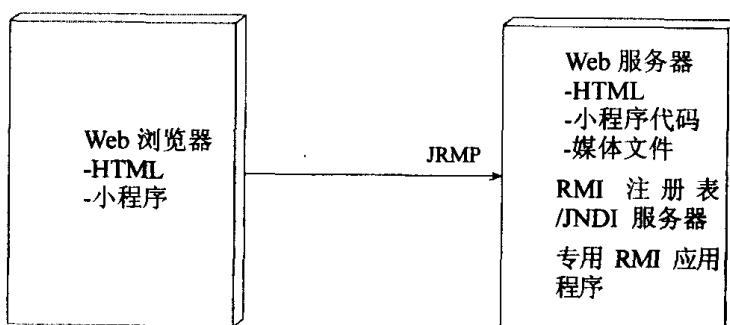


图 9-5 用于小程序的 RMI

#### 9.7.4 CORBA

公用对象请求代理程序体系结构(Common Object Request Broker Architecture, CORBA) [OMG 04]是由对象管理组(OMG)开发的，OMG 是一个致力于开发开放式对象标准(也拥有 UML)的行业协会。CORBA 类似于 RMI，但它比 RMI 早，所以它在行业中立得比较牢。CORBA 和 RMI 的主要区别是：

- CORBA 是多语言的：主软件可以用 C++、Eiffel、C#、Java、甚至非面向对象语言如 C 或 COBOL 编写(服务器软件对客户机来说仍是面向对象的)。
- CORBA 有自己的命名服务(但它也可以通过 JNDI 来使用)。
- Java 主机从 CORBA 上接收未检查的异常，从开发人员的代码中接收已检查的异常——这可能会产生混乱(已检查的异常必须由程序员处理，而未检查的异常不需要)。
- CORBA 比较费钱(用 Java 实现的免费版本适合于开发和测试)。
- RMI 可以访问整个 Java 2 平台(至少是标准版本)，这有益于获得媒体文件的访问权限。CORBA 的一般性使之比 RMI 更复杂，而 RMI 更强大(它支持对象图的动态交换和编译的类)。另外，可以给每个新的应用程序优化 RMI 性能。除此之外，Java 程序员应把 CORBA 和 RMI 看做可以互换的产品。非 Java 程序员应把 CORBA 看做通过网络传送消息的最佳选择(它优于专用的、不能移植的产品)。Sun 开发了一个搭桥协议 RMI-over-IIOP，它允许 RMI 主机与 CORBA 主机混合在一起，但有几个限制。

图 9-6 显示了小程序访问 CORBA 的前端配置。对于非 Java 主机，CORBA 客户软件必须部署在每个客户机上(还要给每个新版本重新部署)。用 Java 编写的 CORBA 主机有与 RMI 主机相同的部署选项。

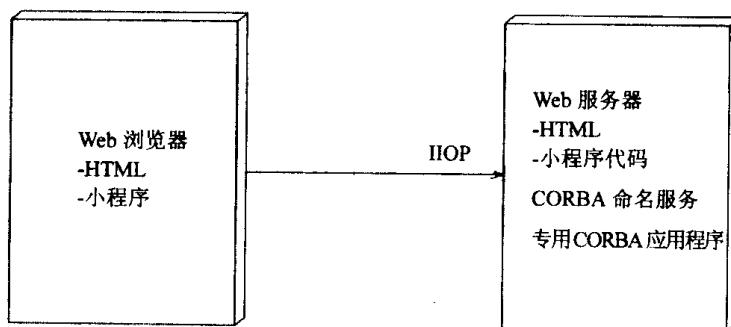


图 9-6 用于小程序的 CORBA

### 9.7.5 EJB

Enterprise Java Beans(EJB)是 J2EE 的一部分[Bodoff 等 02]，是分布式 Java 系统的框架，完全支持事务管理、安全性和持续性。在 Java 术语中，bean 是带有已知样式的接口的对象。EJB 实现方式可以用于许多业界领先企业，例如 IBM、Sun 和 BEA Systems。Sun 还有一个参考实现方式，这样，即使不购买该实现方式，也可以开发和测试 EJB 软件。在许多方面，EJB 框架是 Microsoft 的.NET 策略的竞争对手(尤其是包含 J2EE 的所有其他功能)。当然，.NET 是专用的，移植性不太好，而 EJB 是开放的，可以移植。

EJB 框架的详细论述超出了本书的范围，知道 EJB 有三个主要变体就足够了：

- 实体 bean：对应于 Jacobson/UML 中的实体对象，它是包含业务信息和业务行为的业务对象。实体 bean 可以由 EJB 实现代码使用用户选择的关系数据库自动存储，开发人员也可以选择使用专用代码存储实体 bean(把旧系统和面向对象的数据库集成起来)。访问实体 bean 需要进行事务处理：开发人员可以选择默认设置，或者在一组选项中选择(表示不同速度/准确性的折衷)。为了便于移植，所有的事务处理都由框架指定，而

每个 EJB 运行时系统的实现代码必须确保，实现代码支持这些事务处理。从效率的角度来看，消息应在本地发送给实体 bean，而不是通过网络发送。

- 会话 bean：它以 EJB 客户机的身份管理业务工作。客户机通过网络把消息发送给会话 bean，会话 bean 使用实体 bean 和其他会话 bean 来满足请求。会话 bean 为它访问的所有实体 bean 和会话 bean 提供默认的事务处理。另外，客户可以控制事务处理在何时开始和结束(在有限的范围内，控制它们发送给其他 bean 的方式)。
- 消息驱动的 bean，它与业界强大的机器对机器的消息传输实现方式无缝地集成在一起。在完全的事务处理控制之下，消息可以点对点地发送，或者一次发送到几台机器上。

图 9-7 演示了 EJB 应用程序前端的配置。为了让 EJB 工作，EJB 必须降级为 Enterprise Java Server(EJS)，这常常使用更一般的术语“应用程序服务器”。这不同于 RMI 和 CORBA 的情况，在 RMI 和 CORBA 中，必须提供自己的服务器来包含业务对象。与 RMI 和基于 Java 的 CORBA 一样，也有各种替代的配置。

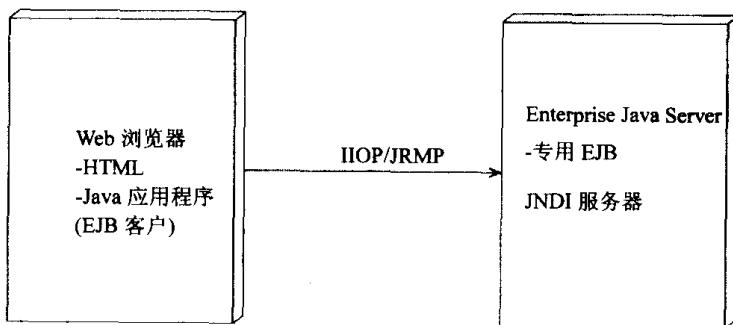


图 9-7 EJB 应用程序前端

## 9.8 后端配置

前面介绍了三层系统中的客户如何使用 CGI、RMI、CORBA 或 EJB 实现。对于互联网应用程序，最常见的选择是 HTML/CGI。对于内联网，可以选择喜欢的机制，但为了便于维护，无论先实现了什么类型的访问，都应力争使三层系统中的中间层可应用于所有类型的客户(如果有受控制的用户群——准备安装 Java 的正确版本的用户，就可以选择喜欢的任何前端技术，甚至可用于互联网)。前面介绍的任意配置都可以用作二层系统的前端。

那么，当我们到达中间层时，会发生什么？最简单的解决方案是在中间层上安装 DBMS 客户程序，从我们选择部署的服务器软件(脚本、服务小程序、RMI 服务器、CORBA 服务器或 Enterprise Java Server)上访问它。访问数据库客户程序可以在中间层代码所在的过程中进行，也可以在其他过程中进行。另外，可以把服务器软件用作 .Net 客户程序或 EJB 客户程序。从中间层上访问的 Enterprise Java Server 或 .Net 服务器提供了业务服务，所以在逻辑上它应属于中间层。但是，所有这些技术都允许在多台机器上，以许多不同的方式部署，因此，服务器驻留在什么地方并不重要。

## 9.9 Java 电子商务配置

因为电子商务系统需要吸引互联网用户的注意，所以我们把 HTML/CGI 前端作为默认机

制。如果部署了其他类型的前端，就有丢失顾客的风险。大多数顾客在面对“必须安装某个软件才能使用这个站点”时，都会选择到其他站点去。没有驻留在 Web 浏览器上的前端也会出问题：希望电子商务的顾客为了商家的利益安装软件是不合理的。

在 JSP(或用于 Microsoft 的 ASP)的帮助下，可以开发出适合于每个用户的交互操作(但不能解决浏览器中固有的重复刷新和速度慢的问题)。一旦到达中间层，就可以选择喜欢的技术访问数据层和中间层的其他部分。

那么，如何把 CGI、JSP、服务小程序、EJB 和 DBMS 合并为一个一致、可伸缩的整体？基本理念是使用 CGI 访问提供了业务服务的服务小程序，服务小程序依赖可重用的 EJB 完成大部分工作；EJB 依赖 DBMS 存储企业数据；最后，服务小程序把结果数据传送给 JSP，以建立个性化的 Web 页面，并发送回客户机。完整的过程如图 9-8 所示(非标准的、UML 样式的关键字用于部署每个对象如何在 EJB 框架中发挥作用)。

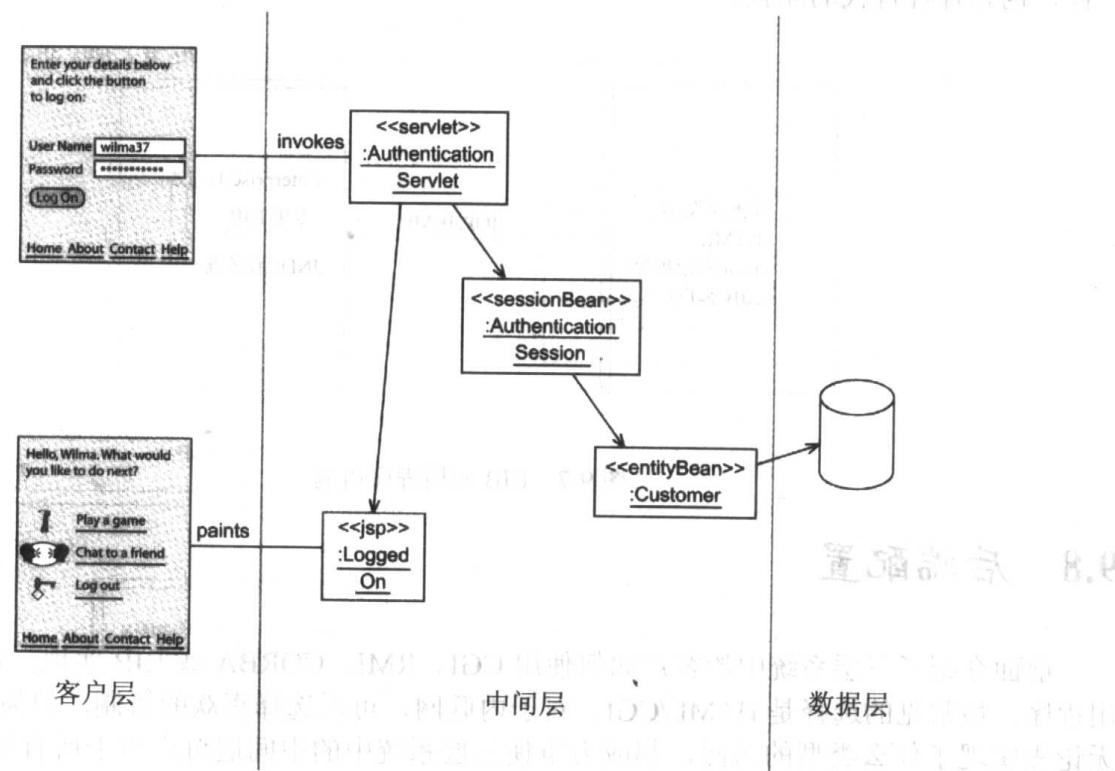


图 9-8 Java 电子商务技术

在图 9-8 中，Wilma 导航到站点的登录页面，所以她可以访问只有注册顾客才能访问的服务。Wilma 填入了她的用户名 Wilma37 和密码，单击 Log on 按钮。Wilma 的窗体数据传送给 AuthenticationServlet，进行处理。AuthenticationServlet 调用一个 EJB 会话 bean，即 AuthenticationSession，来检查用户名和密码。AuthenticationSession 让 EJB 运行时系统查找用户名为 Wilma37 的 CustomerEntity(如果没有这个顾客的记录，就把一个错误发送给服务小程序)。接着，AuthenticationSession 比较 CustomerEntity 的密码和服务小程序传送过来的密码。如果密码匹配，AuthenticationSession 就给 AuthenticationServlet 返回一个会话令牌(唯一的标识值)；如果密码不匹配，AuthenticationSession 就把一个错误发送给服务小程序。

在成功登录后，AuthenticationServlet 会把会话令牌和浏览器会话(通常作为一个 cookie，这是一小块传送到浏览器的信息)以及其他相关信息(例如 CustomerEntity 的键)关联起来。之后，

服务小程序处理 JSP 的请求，把它传送给包含结果数据(例如 Wilma37 的真名)的 Java 对象。JSP 运行其内嵌的 Java 代码，把个性化数据嵌入静态的 HTML 中。把完成的页面传回浏览器并显示出来后，业务处理(用例)就完成了。

如果登录不成功，AuthenticationSession 产生的错误就会被 AuthenticationServlet 检测到。服务小程序把这个请求传送给错误页面，该页面是一个静态的 HYML 页面，包含一般化的消息，例如 User name or password incorrect，该页面也可以是一个动态页面(由 JSP 建立)，其中包含一些特定的信息，例如 The password you entered was incorrect。把错误页面返回给浏览器，标志着业务处理(用例)的结束。

## 案例分析

### iCoot 配置

由于 HTML/CGI 和服务小程序配置非常简单、移植性强、负载小，所以可用于 iCoot 的第一次递增。以后的递增也提供一个适当的 Java 客户，以便顾客可以通过 RMI 使用 GUI 快速而优雅地访问系统。在服务器端，使用一个专用的业务服务器，它使用 JDBC 访问关系数据库。该业务服务器使用 EJB 会话 bean 提供对 Java 事务的访问，但不使用 EJB 实体 bean，因为开发人员还没有对它们的用法进行过培训。

在前面的拓扑部署图(8.4.9 节)中可以增加这些体系结构决策。图 9-9 把过程显示为子节点，每个子节点都用 UML 关键字<<ExecutionEnvironment>>标记，通信路径现在显示为可导航——路径名称可以用于表示通信协议。

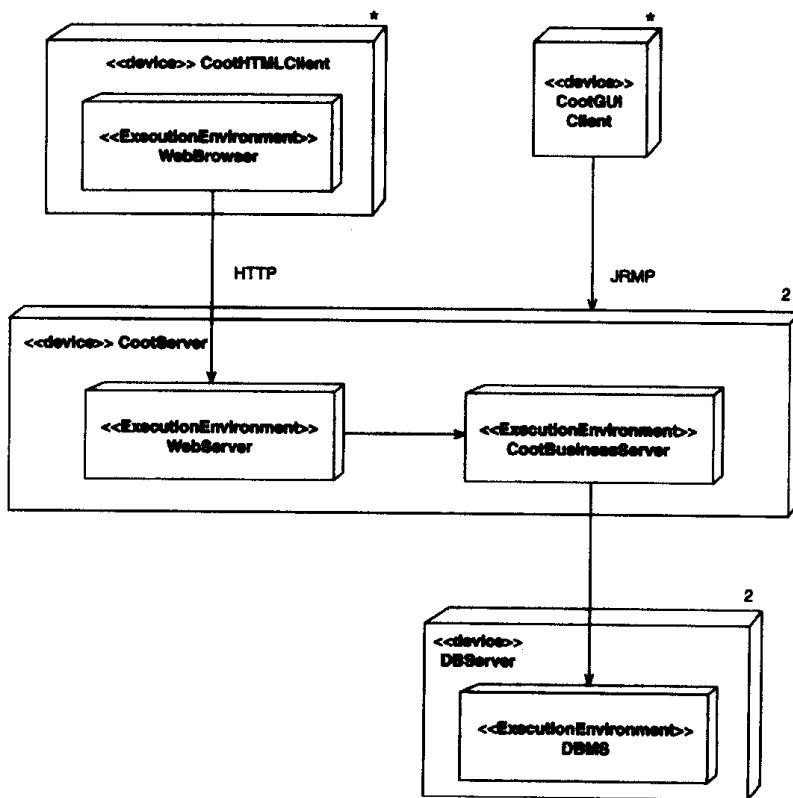


图 9-9 iCoot 更详细的部署图

这个扩展图的部署调查如下：

iCoot 数据层有两个数据库服务器(称为 DBServer)。有这样两个节点可以提高通过量和可靠性。每个 DBServer 都拥有一个 DBMS 过程，以管理对数据的访问。

中间层与数据层通信，它包含两个服务器(CootServer)，也是为了提高通过量和可靠性。每个 CootServer 都拥有一个 CootBusinessServer(用于处理业务请求)和一个 WebServer(用于处理静态 HTML 内容，把业务请求传送给 CootBusinessServer)。CootBusinessServer 的数据访问由 DBMS 提供。因为它们都专用于所选的产品，所以不指定 WebServer 与 CootBusinessServer 之间的通信协议和 CootBusinessServer 与 DBMS 之间的通信协议。

每个 CootServer 都可以由任意多个 CootHTMLClient 节点同时访问。每个 CootHTMLClient 都拥有一个 WebBrowser，以使用 HTTP 访问某个 WebServer 节点。

最后，还可以提供从 CootGUIClient 节点的访问。每个 CootGUIClient 都使用 JRMP 访问一个 CootServer 节点。因为未来递增版本的一个主题是机制允许这种请求进入 CootBusinessServer，所以不给出细节。也不给出 CootGUIClient 过程的任何细节。

## 9.10 UML 包

UML 概念“包(package)”可以组合相关的类。图 9-10 所示的包图(package diagram)把每个包显示为一个左上角有一个标签的方框。包名显示为黑体，位于方框的中间，如果要显示包的内容，包名就显示在标签中。包的内容可以是类或其他包。图 9-10 也显示了一个包到另一个包之间的依赖关系(虚线开放箭头)，它表示，源包使用目标包中的一些内容。

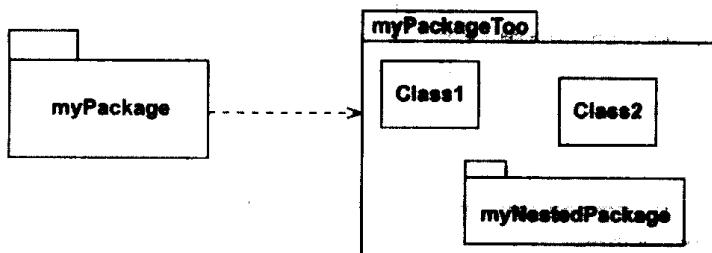


图 9-10 UML 包图

包可以用于表示：

- 层
- 子系统
- 可重用的库
- 框架
- 应一起部署的类
- ...

从编程的角度来看，包方便地映射已有的语言结构，例如 Java 包和 C++ 命名空间，但包只是一个编译期间的概念，有助于组织代码，便于开发、部署和维护。

## 案例分析

### iCoot 包

图 9-11 显示了 iCoot 的设计。在这个设计中，系统中的每一层都映射为不同的包。没有显示标准 Java 库(例如 java.sql)中包之间的依赖关系，该依赖关系由 business 包用于访问数据库。如果包含所有的依赖关系，该图就会非常大、非常杂乱。

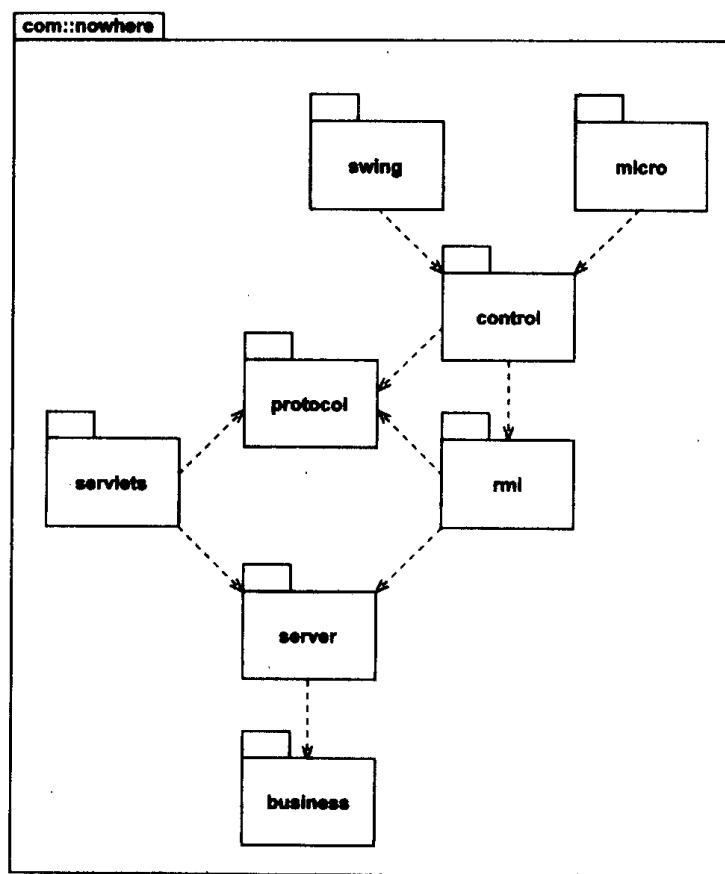


图 9-11 iCoot 的包

iCoot 的主包是 `com::nowhere`，它是“包 nowhere 嵌套在包 com 中”的简称。UML 从 C++ 中借用这个表示法表示命名空间操作符(在编写 Java 源代码时，“`::`”替代了“`.`”)。尽管在 iCoot 中使用了事件，但 `control` 包和上述的包没有依赖关系。这是因为事件监听器和事件类型位于 `control` 包中。`protocol` 包包含轻型复制对象的定义，这些复制对象由服务器层导出，用于简化对 BusinessLayer 的网络访问(详见第 10 章)。

许多开发人员都使用包图来显示层。但是，这种方法有几个问题。第一，层是在决定如何把源代码组织到包中(组织形式也可能不同)之前选择的。第二，包图不允许显示一些重要的信息，例如 iCoot 中存在 `HTTPCGILayer` 是很重要的，但它不映射为可能实现的包，或映射为从库中借用的包(它只表示 HTTP 协议要与 CGI 一起使用)。因此，对于 iCoot 来说，使用特殊的层图比较合适，并有一个附带文档，来描述层的交互策略(参见图 8-14)。

包图也不适合于显示水平分解的部分(子系统)；此时应使用部署图。图 9-12 显示了系统设计完成后的 iCoot 部署图，根据 UML 制品和包详细列出了 CootServer 过程的内容。

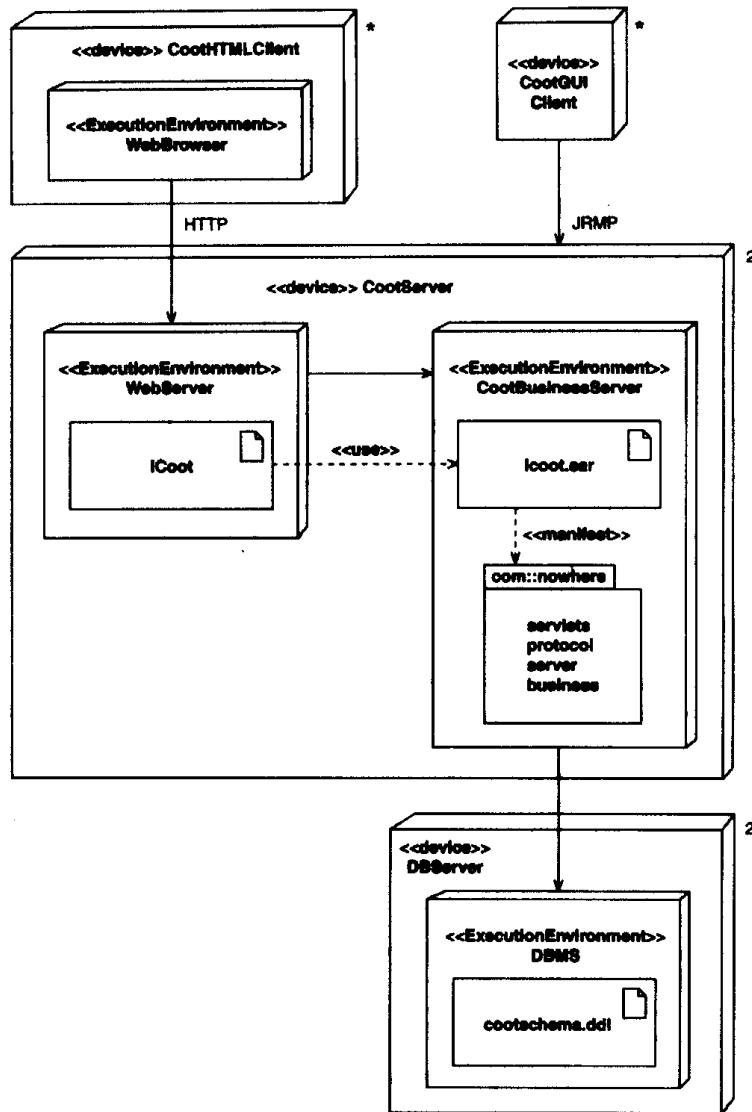


图 9-12 系统设计后的 iCoot 部署图

在 UML 中，制品是可以部署的内容，通常是一个文件。(文件可以是几乎所有的东西，例如程序、DLL、文件夹、XML 数据或 README 文档)。制品由一张纸图标或<<artifact>>关键字来表示。图 9-12 中的制品是 iCoot，静态 HTML 页面的文件夹，icot.ear，服务小程序的压缩文档，JSP 和 EJB(Java 命名约定.ear 是 enterprise archive 的缩写)和 cootschema.ddl(创建数据库的专用脚本)。

添加从制品到元素组的依赖关系，标记为<<manifest>>(制品是元素的表达方式)，就可以显示模型中的哪些元素会引出某个制品。这里，icot.ear 显示为四个 com::nowhere 包的表达方式。也可以在部署图中显示其他依赖关系。例如，使用关键字<<use>>，就可以说明 iCoot 使用 icot.ear。

## 案例分析

### 更新的 iCoot 部署调查

iCoot 数据层有两个数据库服务器(称为 DBServer)。有这样两个节点可以提高通过量和可靠性。每个 DBServer 都拥有一个 DBMS 过程，以管理对数据的访问。

cootschema.ddl 制品包含了创建数据库表的命令，其格式为该数据库所特有。这会使用与数据库相关的工具(这里没有给出细节)，部署到每个 DBMS 过程中。注意 cootschema.ddl 包含整个 Coot 系统的模式，因为 iCoot 和 Coot 使用相同的数据。

中间层与数据层通信，它包含两个服务器(CootServer)，也是为了提高通过量和可靠性。每个 CootServer 都拥有一个 CootBusinessServer(用于处理业务请求)和一个 WebServer(用于处理静态 HTML 内容，把业务请求传送给 CootBusinessServer)。CootBusinessServer 的数据访问由 DBMS 提供。它们都专用于所选的产品，所以不指定 WebServer 与 CootBusinessServer 之间的通信协议和 CootBusinessServer 与 DBMS 之间的通信协议。

在每个 CootServer 中，iCoot 文件夹包含静态的 HTML 页面，部署到 WebServer 上，而 icoot.ear 文档部署到 CootBusinessServer 上。icoot.ear 文档包含 com::nowhere 包中的服务小程序、JSP、业务对象和(最终)RMI 修饰工具。

每个 CootServer 都可以由任意多个 CootHTMLClient 节点同时访问。每个 CootHTMLClient 都拥有一个 WebBrowser，以使用 HTTP 访问某个 WebServer 节点。不需要把制品部署到 CootHTMLClient 节点上。

最后，还可以提供从 CootGUIClient 节点的访问。每个 CootGUIClient 都使用 JRMP 访问一个 CootServer 节点。因为未来递增版本的一个主题是机制允许这种请求进入 CootBusinessServer，所以不给出细节。也不给出 CootGUIClient 过程的任何细节。没有指定部署到 CootGUIClient 节点上的制品。

## 9.11 小结

本章的主要内容如下：

- 可用于客户机和服务器的主要技术
- 可用于连接客户机和服务器的中间协议
- 联网系统的前端常见的技术选择
- UML 包如何用于在部署图中显示相关类的群集

## 9.12 课外阅读

对于 HTML/CGI 客户层来说，要访问中间层上的服务小程序，开放源 Apache 软件基础中的 Struts 框架[Robinson 和 Finkelstein 04]是比较好的。Struts 从实现交互式 HTML 前端的过程中吸取了大多数好的内容。要下载 Struts，查看说明文档，可访问 [www.apache.org](http://www.apache.org)。

如果对 CORBA 编程感兴趣，可阅读[Bolton 01]，这本书囊括了 Java 和 C++。与往常一样，任何技术最后的参考书都是 CORBA 规范[OMG 04]。

要大致了解重要的多层 Java 技术，可参阅[Campione 等 98]，它介绍了 RMI、JDBC、服务小程序和 Java IDL(CORBA 的 Java 接口)。这本书的内容也可以在线获得，网址是 [java.sun.com](http://java.sun.com)。

为了学习 J2EE 的更多内容，包括 JSP、服务小程序、EJB、事务处理、安全性、XML、XSL、SOAP、JMS 和 Java ServerFaces(用于交互式 HTML 界面)，可参阅[Bodoff 等 02]和 [java.sun.com](http://java.sun.com) 的在线版本。Sun 的网站还介绍了 Java 在 Web 服务方面的最新进展。

## 9.13 复习题

1. Java 作为实现技术有哪些优点? (多选题)

- (a) 纯粹的面向对象
- (b) 由 Microsoft 开发
- (c) 网络已准备就绪
- (d) 可移植性
- (e) 可伸缩性
- (f) 安全性

2. 服务小程序可以直接替代什么? (单选题)

- (a) XML
- (b) 小程序
- (c) ActiveX 控件
- (d) CGI 脚本
- (e) CORBA

## 9.14 复习题答案

1. Java 作为实现技术有如下优点:

- (a) 纯粹的面向对象
- (c) 网络已准备就绪
- (d) 可移植性
- (e) 可伸缩性
- (f) 安全性

2. 服务小程序可以直接替代 (d) CGI 脚本

# 第 10 章 设计子系统

本章将介绍子系统的设计(subsystem design，也称为详细设计，detailed design)。在选择了网络拓扑、实现技术、子系统和层后，就要确定每个子系统和层应包含什么。

## 学习目标：

- 理解如何设计业务层
- 理解如何设计用户界面
- 理解如何把运行时对象映射为可存储的数据
- 理解多线程

## 10.1 引言

由于子系统设计的内容很多，而且对每个项目来说，子系统设计都是不同的，所以不能给专业化、面向对象的多层次系统设计总结出详细的步骤。本章将概述各个主要任务，实际上，如果把这些任务作为设计的核心，也会依序完成其他任务。

首先考虑如何设计业务层。这通常包括确定该层有哪些对象、它们如何连接、它们的界面如何。必须把分析阶段开发的、面向业务的类模型转换为适合于所选的编程语言、面向实现的类。

对于本书，子系统设计的讨论或多或少独立于编程语言和其他技术。但是，为了便于演示，会偏向于 Java 和关系数据库(尽管偏向于 Java，但该设计可移植到任何包含纯面向对象功能的语言上；同样，偏向关系数据库也不会使设计只限于某个 DBMS)。

子系统设计需要把概念性的分析模型转换为可实现的类，之后在系统设计模型中制订策略。为了与已讨论过的系统设计规则相一致，子系统设计可以按照如下步骤进行：

- (1) 把分析类模型作为指导，设计业务层的类和字段。业务层包含问题域中的实体和它们需要的各种支持类。
- (2) 确定持久数据如何存储，设计存储布局。持久数据是直到系统关闭才消失的数据。
- (3) 引用分析阶段生成的草案，最终确定用户界面的外观和操作方式。
- (4) 参考用户界面设计，遍历系统用例，注意中间层必须支持的业务服务。业务服务是客户机内发送给服务器的问题和命令，例如“购买一本书”或“预约汽车型号”。
- (5) 把业务服务传送给服务器对象，其消息可通过网络获得。服务器对象使用业务层，以各种客户机认可的方式实现业务服务。
- (6) 最终确定必要的措施，以确保并发控制和线程安全。并发控制就是使用业务规则控制对系统的访问：用户名和密码，在买票前预约等。线程安全即确保过程中的数据不被破坏，平行活动不互相干扰。

## 10.2 把分析的类模型映射为设计的类模型

在从分析转为设计时，一些类会舍弃掉(例如控制器)，引入其他类(例如实现多重性的集合类)。对于本书，设计人员可以自由决定业务对象、边界和前面开发的内容如何转换为可实现的代码(这不同于 RUP 方法，在 RUP 方法中，分析的类模型要转换为可实现的设计模型)。

对于前面提及的每个设计类，都需要选择其字段的名称和类型。通常，字段派生自分析阶段确定的属性或关联。不可能仅通过源代码就判断出某个字段最初是一个属性、复合、聚合还是关联。这是需求捕捉和分析阶段生成的高级制品非常重要的一个原因。

除了属性和关联之外，还需要考虑继承。继承关系不需要映射为新东西，只需决定是否要保留它们即可。继承非常复杂，必须小心处理：在生成的系统中，类没有或很少有继承关系是相当合理的(我们所使用的继承可能是在设计阶段引入的，而不是在分析阶段引入的)。

### 10.2.1 映射操作

到目前为止，操作仅引入为一种记录用例实现的方式——换言之，在模拟系统用例时，操作是验证分析类支持实现代码的一种副产品。为了便于设计，应忽略这些分析操作。那么，设计操作从何而来？既然进入设计阶段，就可以使用编程术语“消息”和“方法”，来代替 UML 术语“操作”。那么，消息从何而来？对于大多数对象，无论它们位于哪一层，都因为下面一个或多个原因来添加消息：

- 允许客户机对象读取或修改字段值。
- 允许客户机对象访问派生的数据(例如消息除了可以读取圆的半径之外，还可以读取直径)。
- 经验和直觉告诉我们某个消息是有用的。
- 要使用的某个框架或模式需要某些消息。

另外，在为中间层设计业务服务时，也会为服务器对象设计消息。在使用业务对象判断业务服务是否令人满意时，也可能产生更多的消息。简言之，在把分析的类、属性和关系元素映射为对应的设计类、属性和关系时，就会产生许多消息。

### 10.2.2 变量类型

在引入一个字段时，需要确定其类型是原型还是类。在大多数情况下，可以把字段的类型限制为：

- 在每个面向对象的编程语言中都有的原型类型和简单类(例如 int、float、boolean、String、List、Set 和数组或[])。
- 自己设计的类
- 选择使用的模式和框架中的类

在一些语言中，数组和集合结合得不是很好——数组是固定大小的，可能根本就不是对象。因此，选择应依据优雅原则(集合)或考虑性能的提高(数组)。

### 10.2.3 字段的可见性

除了提供字段的名称和类型之外，还必须声明它们的可见性。字段的可见性指定哪些代码

可以读取或修改其值。下面的可见性足以满足大多数需求：

- **Private**(在 UML 中表示为-): 仅在定义类中可见。
- **Package**(在 UML 中表示为~): 仅在定义类和该包的所有类中可见。
- **Protected**(在 UML 中表示为#): 仅在定义类、该包的所有类以及定义类的所有派生类(在包中或包外)中可见。
- **Public**(在 UML 中表示为+): 在所有的地方都可见。

自然，如果语言允许，开发人员可以把字段设置为 **Private**，除了有封装的好处之外，编译器还会有更多的优化机会。有时，开发人员把字段设置为 **Protected**，这样子类的开发人员就有更多的机会修改超类的行为(但这会增加子类和超类之间的耦合性)。

可见性是 **package** 的字段并不好，因为它们可以被包中的任意代码访问，而拥有该字段的对象并不知情。偶尔为了编程原因(性能和简短性)，开发人员也会给字段设置 **package** 访问权限，但只有所选的语言提供了把值设置为只读的方式(例如，在 Java 中是使用关键字 **final**)，才会这么做。这种方式甚至适用于 **public** 字段，因为这些字段在包的外部也可见。

可见性也可以应用于消息。对于消息：

- 如果消息是包的接口的一部分，消息就是 **public**。
- 如果消息是类本身和包中其他类使用的实现代码，消息就是 **package**。
- 如果消息是类本身、其子类和包中其他类使用的实现代码，消息就是 **protected**。
- 如果消息是类本身使用的实现代码(降低了耦合性，允许编译器进行更多的优化，这与字段相同)，消息就是 **private**。

并不是所有的语言都支持 UML 中使用的这四种可见性，但 Java 支持。

#### 10.2.4 访问器

最好为字段提供访问器(accessor)消息。访问器消息有两个变体：读取器(getter)返回字段的值，设置器(setter)给字段设置新值(参见练习 5)。访问器允许集中访问字段，这更便于维护。访问器也便于编译器进行优化(尤其是关联的变量为 **private** 的情形)。在类中，可以略微放松规则：通常可以直接读取字段，但使用设置器来设置字段。

#### 练习 5

这个 Java 类显示，一个字段有一对访问器：

```
...
private int count;

public int getCount() {
    return count;
}
public void setCount(int c) {
    count = c;
}
...
```

#### 10.2.5 映射类、属性和复合

在 UML 中，分析类图和设计类图使用相同的表示法。但是，在设计图中有更多可用的表示法。

类字段的表示法与实例字段一样，但类字段不加下划线。同样，类消息使用的表示法也与实例消息相同，但类消息要加上下划线。这种下划线策略与类名(在类图上)和对象标签(在对象图上)的下划线策略不同——类名不加下划线，而对象标签加下划线。是的，这很容易混淆。其理由是给不常绘制的版本加下划线。

图 10-1 显示了把 iCoot 分析类图的一部分转换为设计类图。在映射到设计图时，解决了三个主要问题：要实现的类、属性的类型和如何映射复合。在这个例子中，保留了两个分析类，而且不需要引入新的支持类(除了 String 类之外)。所有的属性都转换为某种适当类型的私有字段。可选的属性以相同的方式映射，惟一的区别是，在运行期间，对应的字段应使用 null 值。复合关系要多加考虑。为了便于分析，复合是双向的：从容易找出的任一个对象开始到另一端的对象。但在设计时，必须考虑到，字段只能指向一个方向(在运行期间，可以从固有对象指向远端的对象，但不能反向)。所以，需要确定是在关系的每一端都有一个字段，还是只在一端有一个字段(还要判断在哪一端有这个字段)。

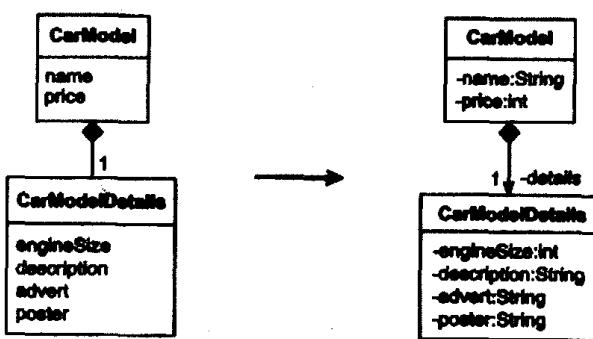


图 10-1 从分析中映射属性和复合

复合的初始关系是从复合者到被复合者。这反映了复合者是关系的拥有者，而且复合类似于属性：正常情况下，固有对象使用属性或被复合对象的服务，但反过来不行。注意这不是绝对的：如果觉得更适合自己的系统，就可以把字段放在被复合的对象中，或者放在复合和被复合的对象中(UML 还有数据类型的表示法，专门用于内嵌的属性[OMG 03a])。

一旦决定了复合的方向，就可以给类图添加箭头来表示它，再添加角色和可见性来表示复合如何映射为字段(参见图 10-1)。在运行期间，消息只能沿着箭头的方向传递。还可以删除复合，在 CarModel 中显示-details:CarModelDetails 字段。大多数 UML 创建工具都允许开发人员随意扩展和压缩这类信息。

在图 10-1 中，advert 和 poster 字段都是 String 类型，这是因为不需要在 iCoot 系统中处理和存储原始媒介。而可以在 Web 服务器的控制下，把这些复杂的类型部署为文件，例如\*.ram 和\*.png 文件，再把相关的 URI 存储为属性值。例如，CarModel 可以把 “/adverts/mcgs.ram” 作为其 advert 字段的值。

## 10.2.6 映射其他类型的关联

处理完复合后，下面看看其他类型的关联的映射。有三种关联：(一般)关联、聚合(强)和复合(更强)。为了便于映射，不需要区分聚合和关联，因为面向对象的编程语言对它们没有区分。所以后面对这个论题的讨论使用术语“关联”。

从分析阶段继承下来的大多数关联，以及专门为设计阶段添加的新关联，最终都映射为对象上的字段。一些关联映射为类字段。无论映射为什么字段，由于字段只能在一个方向上导航，

所以需要决定是要双向导航，还是要单向导航，这样，关联的实现可取决于每端的多重性：一对一、一对多，还是多对多。

### 1. 一对一关联

先看看图 10-2 上部的分析类图。它显示了实现关联的三种方式：可以把字段 account 放在 Member 中，指向 InternetAccount；也可以把字段 member 放在 InternetAccount 中，指向 Member，还可以合并两者，实现双向关联。

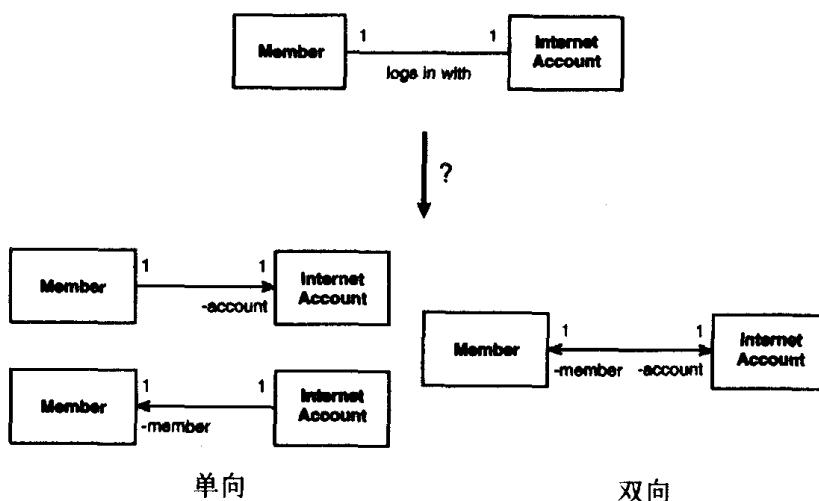


图 10-2 映射一对一关联

如果使用双向关联，就需要添加代码，确保关联的步调一致。例如，如果不告诉原始的 **Member** 其 account 已改变(可能是 null，表示“没有值” )，就改变 **InternetAccount** 的 member 字段，使之指向另一个 **Member**，这是没有意义的。使用 **Member** 和 **InternetAccount** 的适当方法肯定可以实现这种同步，但这是很困难的，因此，在大多数情况下，选择单向关联。在单向关联中，自然的选择是最合适的：让 **Member** 指向 **InternetAccount**，而不是让 **InternetAccount** 指向 **Member**。

如果选择单向关联，只要在业务层下有一个关系数据库，就总是可以在运行期间派生辅助信息。例如，如果选择把一个字段放在 **Member** 中，任何 **InternetAccount** 都可以根据需要询问数据库其 **Member** 在哪里。

可选关联(一端的多重性为 0..1，或两端的多重性都为 0..1)类似于一对一关联，但字段可以取 null 值(表示远端没有对象)。

### 2. 一对多关联

图 10-3 显示了一个一对多关联的例子。在一对多关联中，仍必须确定是在一端还是两端放置字段，这也适合于一对一的情形。与以前一样，有一个很自然的解决方法，也可以像设计人员那样使用技巧和判断力，采用另外一种方式。

与一对一的情形不同，如果这次在一端放置字段，就必须准备存储多个关联对象。例如，如果给 **CreditCard** 添加 members 字段，该字段就要存储一个或多个 **Member** 对象。所以，**CreditCard** 需要使用某个集合类来存储其 **Member** 对象(根据个人喜好，还可以是数组)。在图 10-3 中，使用了 **List**。这有一个进退两难的问题：自然选择出的集合是 **Set**，但在给 **Set** 添加对象时，大多数实现代码都会检查要添加的对象是否已在 **Set** 中，所以，对象在 **Set** 中不会重复，这是一个非常

昂贵的操作。因此，如果可以肯定不打算把同一个对象添加两次，就应使用 Bag。如果不能肯定，也应使用 Bag，但要手工进行重复测试。如果实现语言没有提供类似于 Bag 的集合，例如 Java 就没有提供，就应使用 List，尤其是项目要按顺序排列时，就一定要使用 List，这样搜索会较快。

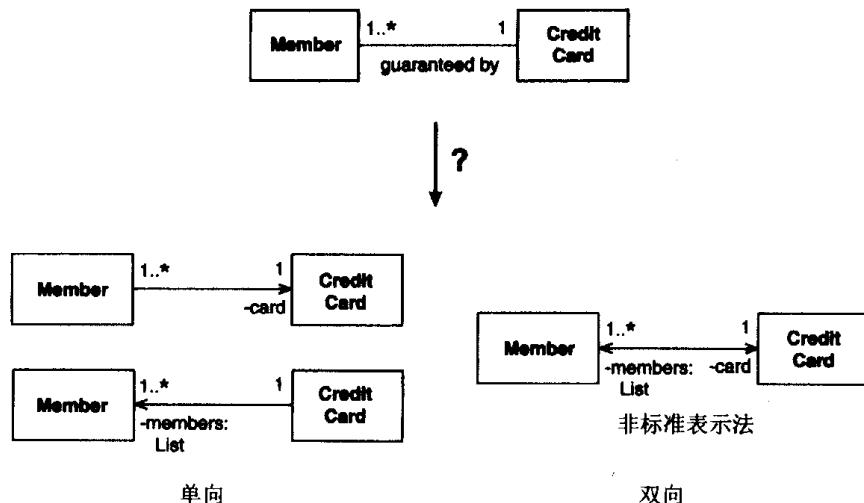


图 10-3 映射一对多关联

图 10-3 中关于带集合值的字段的表示法，不是合法的 UML。肯定可以把字段显示为可导航的关联，并用一个角色来表示可见性和字段名，但>List 部分是不标准的。在 UML 中显示这个关系的另一种方法比较复杂，信息也不完整，这里显示的版本很紧凑，并允许绘制出集合内部对象类型之间的关系。

### 3. 多对多关联

这是最复杂的一种情况。考虑图 10-4 中的例子。该例子有一个 Make，可以制造出任意数量的 CarModel 对象，每个 CarModel 都可以由一个或多个 Make 对象制造出来。这三种映射标记为 A、B 和 C——这里再次选择了 List 对象，这次它允许按照重要性对 Make 对象排序。

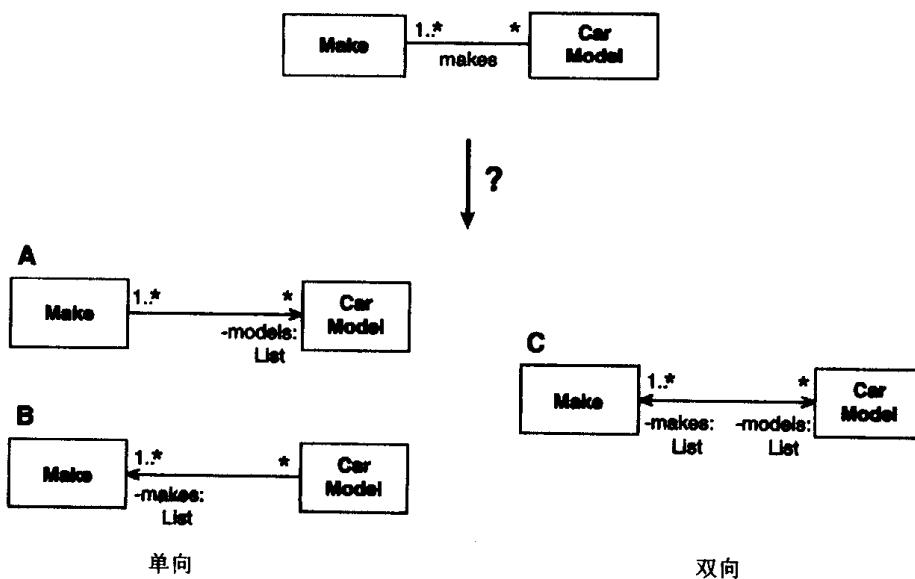


图 10-4 映射多对多关联

与许多对多关联一样，这个例子的关联也没有所有者。如果根据问题域来决定从一个 Make 开始，导航到它的 CarModel 对象上，则选项 A 不错。相反，如果从一个 CarModel 开始，则选项 B 比较好。如果从 Make 或 CarModel 开始的可能性相同，则选项 C 比较好。

使用选项 C，同步问题比一对一和一对多的情形更严重：必须搜索整个集合，找出对象，而不能直接访问各个对象。在这种情况下，引入一个关联类(association class)来处理复杂性比较好，如下所述。

#### 4. 关联类

我们在分析阶段就遇到了关联类。如果数据与某个关联相关，就需要关联类。图 10-5 上部的分析类图就是关联类的一个例子。

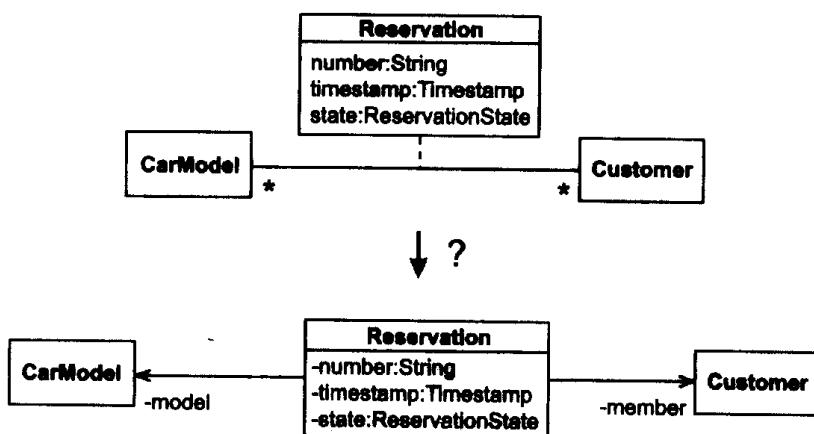


图 10-5 映射关联类

在图 10-5 中可以看出，Customer 对象可以预约任意数量的 CarModel 对象，CarModel 对象可以由任意数量的 Customer 对象预约，但对于每个 Reservation 链接，都需要记录预约号、时间戳和状态，这些数据不属于 Customer，也不属于 CarModel，它属于关联本身。因此，应把 Reservation 引入为一个关联类，并带有适当的属性。

在设计关联类时，最容易的是创建一个设计类，给所有的属性设置字段，再另外设置两个字段，指向被关联的对象，如图 10-5 下部所示。接着，为每个链接创建该关联类的实例。

如果要从 CarModel 导航到它的 Reservation 对象，就必须给 CarModel 添加一个字段，并带有所有的同步问题(类似的过程也适用于 Customer)。另外，还可以在 **ReservationHome** 类上提供一个 `findAllReservationsFor(:CarModel)` 消息。(home 允许创建和查找特定类的对象，通常是在底层的数据库中查找，每个 home 对象都是单一的(singleton)，我们要编写代码，确保只存在 home 的唯一实例)。

关联类是把分析阶段的任意关联映射到设计阶段的最一般方式。因此，如果需要，可以引入关联类来表示分析类图中的每个关联。这会缓解前面提到的同步问题。一些代码生成工具就是以这种方式操作的，但如果手工编写代码，就会觉得这太麻烦了。

#### 案例分析

iCoot 业务层类图

在图 10-6 中, iCoot BusinessLayer 的所有分析类都映射为设计类, 可以直接在任意流行的面向对象语言中实现。这个类图封装在一个 UML 框架中, 包名显示在左上角。这些类的字段详见 8.5 节。

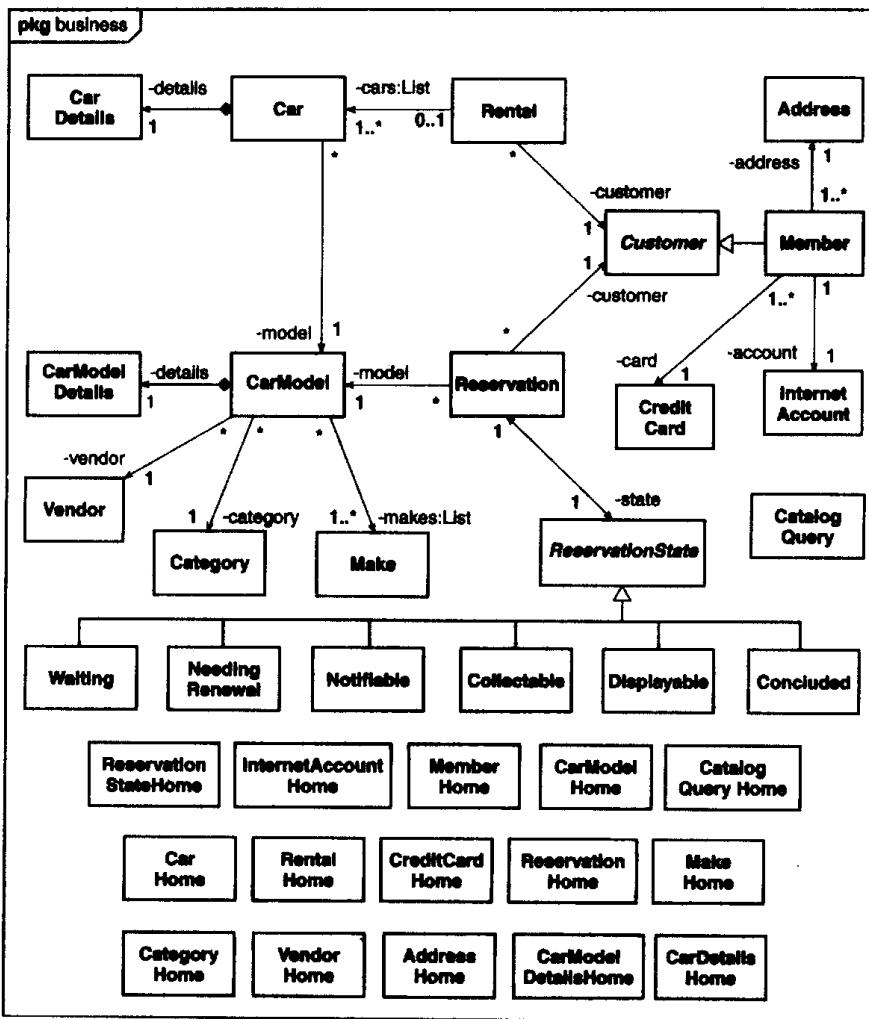


图 10-6 iCoot 的 BusinessLayer 类

### 10.2.7 通用标识符

大多数业务对象在其生存期的某个时刻, 都需要按键(key)来检索。键是属性值或值的组合, 对每个实例来说, 键都是惟一的。例如, 银行账户由其账户号码和有序码构成, 进行惟一地表示。

在软件系统中, 处理不同类型的键是很麻烦的。因此, 最好引入一个惟一的号码, 从同一个类的对象中区分每个业务对象。这有助于同步对象的副本, 追踪它们在网络上的移动轨迹, 高效统一地处理键(例如通过 home 对象)。对象和链接存储在数据库中时, 也可以使用这种通用标识符——当系统关闭时, 通用标识符可替代内存地址。

通用标识符的合适类型是整型。整数的特定类型取决于每个类能容纳的不同对象数: 16 位整数(Java 中的 short)允许有 65000 个不同的实例, 32 位的整数(Java 中的 int)允许有 40 亿个不同的实例。如果 40 亿还不够, 则可以使用 64 位的整数(Java 中的 long), 其实例数不限(大约 18 百万的三次方)。

如果使用通用标识符，就应在对象的构造期间设置，之后它就是固定不变的。图 10-7 演示了 CarModel 类的通用标识符的实现。UML<<create>>关键字用于表示 CarModel 操作创建 CarModel 的实例。在 UML 中，这是一个很好的方式，因为没有关键字，读者就必须假定与类同名的操作是“构造函数”，但实际上构造函数是否存在，它们的名称是否遵循相同的规则，取决于语言本身。(有构造函数是很常见的，但仍取决于语言本身)。因此，本书把术语“构造函数”用作“用于创建类实例的操作”的缩写方式，但在图中仍使用<<create>>关键字。

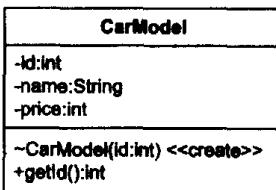


图 10-7 实现通用标识符

一般，构造函数不是公共的，这就允许开发人员控制给哪些通用标识符赋值，例如使用 home(如果构造函数是公共的，任意客户机就都可以使用别处的标识符来创建 CarModel 了)。在图 10-7 中，构造函数有包访问权，所以对象可以由驻留在同一个包中的 CarModelHome 类来创建。

## 10.3 使用关系数据库实现存储

大多数系统都有数据存储的需求，所以下面介绍运行时对象如何映射为可存储的数据，集中讨论如何使用关系数据库完成这个任务，因为关系数据库是目前业界最流行的数据存储技术。一旦设计好业务层和数据库模式，就可以考虑需要编写什么代码，把一个转换为另一个。

### 10.3.1 数据库管理系统

应用程序的数据常常另外存储，所以在应用程序关闭时，数据不会消失。这种数据称为永久性(persist)数据。数据库管理系统(Database Management system, DBMS)管理多个数据库中任意数量的数据，数据库就是把数据隔开的区域。术语 DBMS 和数据库常常可以互换。这里 DBMS 用于：

- (1) 使用数据定义语言(Data Definition Language, DDL)创建一个模式，来描述要存储的数据。
- (2) 使用数据操纵语言(Data Manipulation Language, DML)添加、删除和更新数据库中的数据。
- (3) 使用数据查询语言(Data Query Language, DQL)从数据库中检索数据。

理想情况下，DDL、DML 和 DQL 都应是声明性的语言。在声明性的语言中，指定了内容，而不是步骤。例如，我们不知道也不知道 DBMS 如何在磁盘上物理组织数据，而只是说明存储什么类型的数据，数据值是什么，要检索什么数据。

在过去的 40 年中，出现了 DBMS 的许多变体——索引文件(indexed file)、层次型(hierarchical)、网络型(network)、关系型(relational)和面向对象式(object-oriented)等。在用于编写代码的编程语言和访问数据库中数据的方式之间总是有语义上的分歧，所以，需要在软件系统和数据库之间进行某种运行时映射。一些工具(例如 EJB 实现工具)可以生成映射代码。但是，

即使在项目中使用这类工具，仍需要理解底层的规则，才能有效地使用它们(以及理解运行时错误消息)。

关系数据库常用于在多层 Internet 系统中存储数据。这里选择关系模型是因为：

- 关系数据库是最常见的。尽管存在面向对象式的数据库，但它们的使用并不广泛，尤其在商界并不多见。(大多数关系数据库都有面向对象式的扩展，但为了简单起见，可以忽略它们)。
- 所有的关系数据库都根据严谨的数学，支持相同的核心模型，因此在 DBMS 的各种变体中，关系数据库是很独特的。
- 所有的商业关系数据库都支持相同的 DDL/DML/DQL 语言：结构化查询语言(Structured Query Language, SQL)。
- Java 有一个全面的、可移植的库，用于连接关系数据库，称为 Java 数据库连接(Java Database Connectivity, JDBC)。

公平地看，选择关系数据库的上述原因有点乐观。一些关系 DBMS 使用 SQL 的非标准版本——一定要先检查 SQL 版本。例如，ANSI SQL-92 标准(JDBC 就基于此标准)也不是一个非常严谨的规范：

- 在某些数据库中，冒号在 SQL 语句中是可选的，而在其他数据库中，冒号是必须有的。
- 所选的 DBMS 可能把几个数据类型压缩为一个，例如，DATE、TIME 和 TIMESTAMP 都实现为 TIMESTAMP。
- DBMS 不支持一些数据类型。
- 用于某个数据类型的位数不标准：INTEGER 在一个 DBMS 中是 16 位，在另一个 DBMS 中是 32 位。

这些问题给可移植的库，如 JDBC，带来了一些困难。如果不需要非常高的精确性和性能，关系模式将适合于任意流行的商业系统，但如果要把一个完整的系统从一个 DBMS 移植到另一个 DBMS 上，就需要进行一些移植工作。

### 10.3.2 关系模型

关系模型是一种数学模型，它很整洁、可靠，易于优化(对 DBMS 来说)。但是，关系模型像一个有索引卡的文件柜，而不像一个复杂、连接紧密的对象库。所以，语义上的分歧很大。

面向对象式的模型很容易映射为关系模式，但很难预测几个映射中哪一个最适合于某种软件系统(考虑存储效率和访问速度)。所以，需要使用某种映射工具或持久层来试验，找出最适合系统的映射。

下面介绍一种映射：直接的、纯粹的映射。这样就可以确保它能在关系数据库中存储面向对象的数据。对于这种映射，假定控制了数据的存储方式。实际上，我们不可能这么幸运：模式在数据库管理员(Database Administrator, DBA)的控制之下，或者数据库已经存在，是一种不能修改的老系统——在这些情况下，就必须对映射进行“逆向工程”。

本节不介绍如何编写代码，在软件系统及其数据库之间传送数据，因为这可能是整个系统中最难编写的代码。如果使用像 EJB 这样的框架，工具就会生成基本映射所需的所有代码。为了手工完成工作，需要大量的精力，对数据库编程接口进行仔细的研究。但这里只需知道，底层技术包括使用 SQL 语句(通过像 JDBC 这样的库)把数据库中的数据读到运行时对象中，再编写新数据，返回数据库即可。

数据库层由业务层封装，我们不必为客户端程序员和数据库层调整业务层和数据库层——映射代码可以完成其他工作。

## 1. 表

关系模型基于数据表(也称为关系)，表包含列和行。图 10-8 显示了 ADDRESS 表，它有四列字符串数据(HOUSE、STREET、COUNTRY 和 POSTCODE)和一个整型数据(ID)。可以看出，ADDRESS 表存储了三个不同的地址。每一列存储一个地址属性的值，所以 ID 为 2 的地址表示“8 Yewbrook Rd, Cheshire, SK4 3QT”。

ADDRESS				
ID	HOUSE	STREET	COUNTRY	POSTCODE
2	8	Yewbrook Road	Cheshire	SK4 3QT
9	Dunroamin	Dairy Avenue	Greater Manchester	M19 4IK
6	74	Old Ladbroke Grove	Lancashire	M20 7HJ

图 10-8 ADDRESS 表

每一列都可以存储某种类型的值。SQL 标准定义了几十种类型，以供选择。本书要使用如下类型：

- VARCHAR(X): 一个字符串，至多有 X 个字符。
- INTEGER: 整数，常常(但不总是)有 32 位。
- DATE: 阳历的一天。
- TIMESTAMP: 日期和时间的组合。
- BOOLEAN: 真或假。

例如，对于上面的 ADDRESS 表，可以把 INTEGER、VARCHAR(20)、VARCHAR(40)、VARCHAR(40) 和 VARCHAR(10) 指定为 ID、HOUSE、STREET、COUNTRY 和 POSTCODE 的类型。

在关系模型中，每一行都必须是唯一的。对于 ADDRESS 表，这是通过引入唯一的 ID 属性来实现的。由于每一行都是唯一的，所以每个表都是一组行。图 10-8 中的行可以以任意顺序显示。

## 2. 键

键是一个值或值的组合，它唯一地标识了一行。一些表包含明显的键，例如，保险政策总是有一个政策号。其他表需要创建键，例如，顾客表需要包含一个顾客号来唯一地标识购买商品的人。组合几个值的键(例如银行账号和有序码)称为组合键。

一些表有多个候选键，例如，汽车有驾照号和车辆识别号码(Vehicle Identification Number, VIN，显示在车身的一个小牌子上)。在这种情况下，必须选择一个候选键，用作主键——DBMS 假定主键是最常用于查找某一行的键，它可以进行优化。

使用组合键查找一行比使用简单键慢一些。例如，在 ADDRESS 表中，可以使用(HOUSE 和 POSTCODE)作为键，因为该组合对于每一座房子来说都是唯一的。但是，字符串组合使用起来并不方便，也较慢。所以在 ADDRESS 表中，引入了一个人为的键(ID)，来唯一地标识每座房子。这也允许记录两个顾客住在同一座房子中的情况(可以存储 ID 两次，但地址只存储一次)。

在本书的模式图中，主键显示为黑体。如果手工绘制模式图，可以在列名的旁边加上“+”。

### 3. 把对象模型映射为关系模型

在把对象模型映射为表时，可以先从分析类图或设计类图开始。分析图更接近于关系模型，因为它没有显示关联的方向，所以似乎是一个好的选择。但是，设计类图把类型赋予字段，在设计表时需要知道字段的类型。为了解决这个两难的问题，应使用分析模型来设计表，再根据设计图确定类型。设计模型和数据库表之间的任何分歧都可以通过映射所编写的代码来去除。

#### 10.3.3 映射实体类

为了把面向对象的模型中的实体(业务对象)映射到关系模式中，需要引入一个与实体类同名的表。(ADDRESS 表就是这样一个例子)。实体表中的每一行都表示业务域中的一个独特对象。

对于每个简单的字段(原型或字符串)，都可以在表中添加一个与字段同名的列和一个对应的 SQL 数据类型。指向(非字符串)对象的实体字段必须另外处理，稍后在映射关联时介绍。

为了便于进行面向对象的编程，引入一个整数属性(如 ID)作为主键是很重要的。这是因为它简化了映射代码(如果手工编写代码，这就很重要)，更容易从一个对象导航到另一个对象，效率也比较高(如果对象构成了复杂的图，这尤其重要)。它还便于与通用标识符联系起来。从现在开始，名为 ID 的 INTEGER 列会添加到每个实体表中，但它不显示在随附的类图中。

#### 10.3.4 映射关联

在介绍如何把分析类模型映射为设计类模型时，论述了必须把双向分析关联转换为单向指针。关系数据库直接存储双向关联，所以没有这个问题。如果数据库模式允许从实体 A 导航到实体 B，DBMS 和查询语言就允许从实体 B 导航到实体 A。因此，数据库模式更类似于分析类模型，而不是设计类模型(但应在生成数据库模式之前设计业务层，因为设计过程比分析模型难，有助于选择属性类型)。

##### 1. 一对关联

对于一对关联，可以给一个实体表添加外键。外键就是一个表中指向另一个表中主键的一项。换而言之，外键就是一个表的一行对另一个表的一行的引用。为了强调哪些列表示外键，列名显示为斜体。(如果手工绘制，可以在列名前加上“>” )。

在图 10-9 中，CARMODEL 表包含一个外键 CARMODELDETAILSID，它允许在 CARMODELDETAILS 表中查找 CarModel 的信息。(这里显示的类图使用了分析表示法，没有可导航性，但每个表都添加了通用标识符，只是它们都不是分析层次的属性)。也可以给 CARMODELDETAILS 表添加外键 CARMODELID——这反映了组合关系的逻辑本质。由于关系数据库的双向属性，不需要在两个表中都添加外键。

作为外键的一种替代方法，还可以把两个表合并为一个表。为了便于维护，只有一个表不能独立表示一个实体时，才应合并表。例如，如果 CarModelDetails 只是 CarModel 的属性组合，为了方便程序员编程，应给 CARMODEL 表添加 RELEASEDATE、CATALOGDATE、CLIP 和 POSTER 列，删除 CARMODELDETAILS 表。这样，开发人员就可以把 CarModelDetails 看做一个实体，因为它有一个通用标识符。

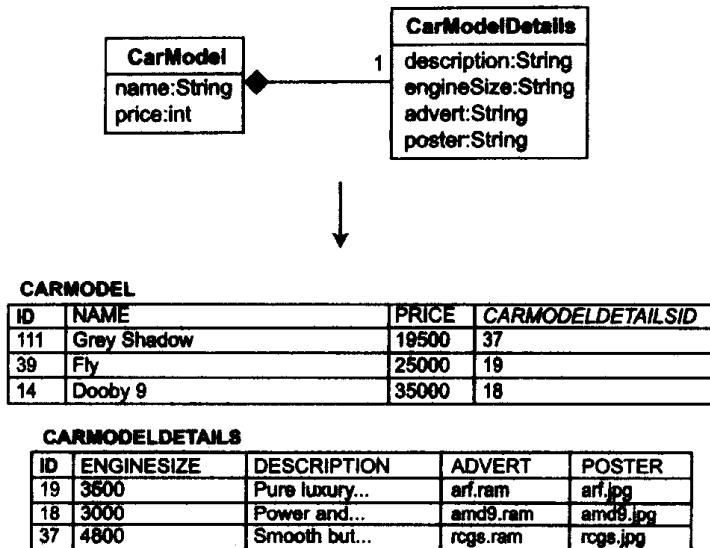


图 10-9 把一对关联映射为外键

对于可选关联的特殊情况(多重性是 1 和 0..1)，可以给可选的一端添加外键。关系数据库偶尔也支持可空的(nullable)列。可空列允许单元格包含 NULL 值，即“这里没有值”。因此，也可以在 1 端把可选的关联存储为可空的外键。但是，为了简单和从容一些，最好避免可空值。

## 2. 一对多关联

对于一对多关联，可以在“多”表中添加外键，如图 10-10 所示。在这个例子中，每个 **CreditCard** 都保证有多个 **Member**。所以，**MEMBER** 表中的每一行都有对其卡的引用(**CARDID**)。(这假定每个会员在系统中都只记录了一个卡)。在完成的系统中，这个 **MEMBER** 表也有一个外键，表示会员的地址。

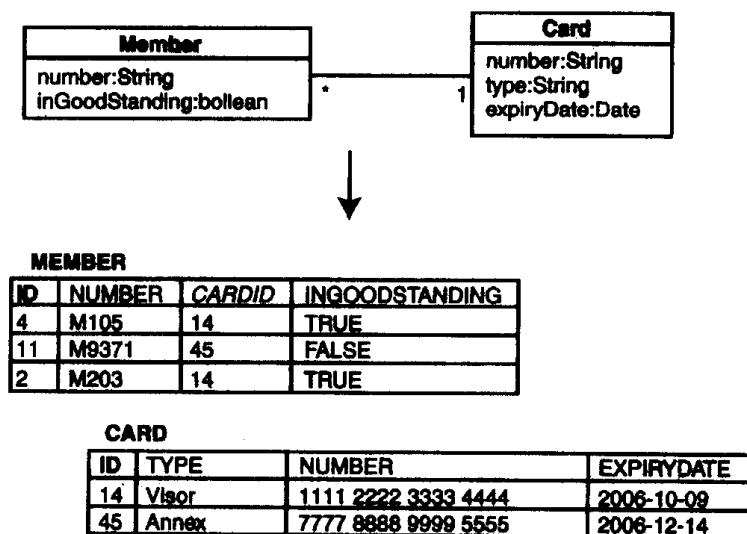


图 10-10 把一对多关联映射为外键

### 3. 多对多关联

对于多对多关联，一个外键不足以标识关联两端的多个实体。考虑图 10-11 上部的分析类图。其中，一个 Make 可以生成许多 CarModel 对象，一个 CarModel 对象可以由许多 Make 生成(Make 对象在生成 CarModel 的过程中协作)。在纯关系模型中，表中的每个值都必须是原子化的，即不是值的集合。所以不能把 CarModel 的所有 Make 对象都存储在 CARMODEL 表的 MAKEID 列中。

由于不能有多值属性，因此需要使用链表。顾名思义，链表中的每一行都表示一个表中的实体与另一个表中的实体之间的链接。图 10-11 中的下半部分显示了一个链表 MAKECARMODEL，它链接了 CarModel 对象和 Make 对象。例如，Make 8 和 Make 9 都是 CarModel 39 的生产厂家，Make 8 还生产了 CarModel 111。从技术上看，链表有一个组合主键，它包含两个外键。

还可以使用链表存储一对一和一对多关联。但是，链表最适合于多对多关联的特殊情况和关联类。

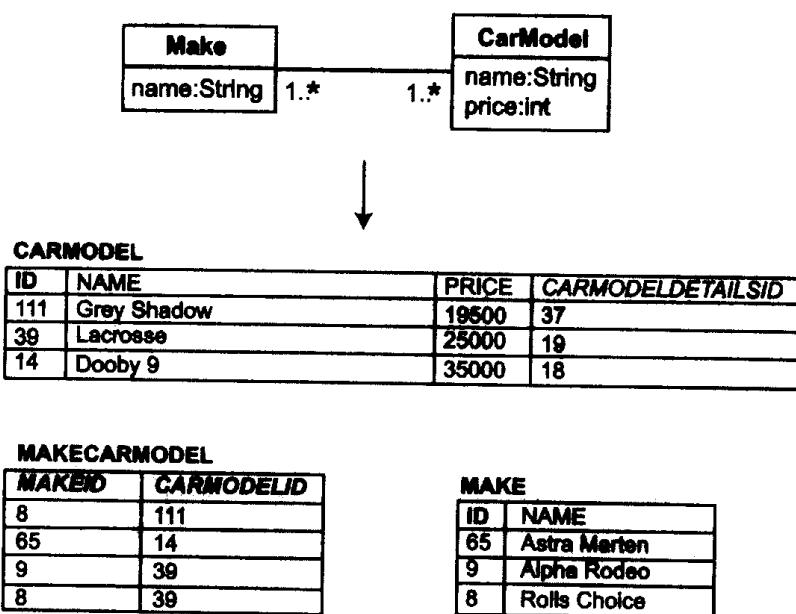


图 10-11 把多对多关联映射为链表

### 4. 关联类

关联类由于有自己的数据，无论关联两端的多重性如何，都必须映射为链表。与普通的链表不同，表示关联类的链表有属性列——它们甚至有 ID 列(如果关联类本身表示一个实体)。

例如，图 10-12 显示的表表示关联类 Reservation。RESERVATION 表有一个用于预约的主键，两个外键(分别引用关联两端的对象)，以及两个属性列。

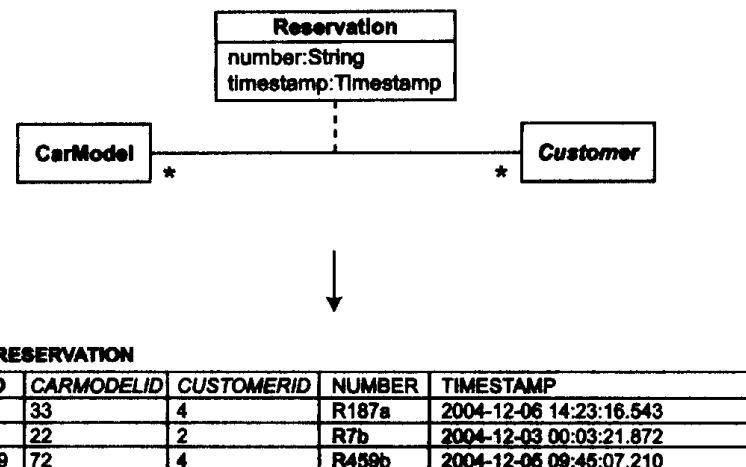


图 10-12 把关联类映射为链表

### 10.3.5 映射对象状态

对于关联了状态机的业务对象，例如状态机图中的对象，需要记录每个对象所处的状态。在业务层，业务对象可以有一个简单的字段表示其状态，可能是 String 或 int，也可以有一个复杂的字段指向一个状态对象(用 State 模式描述)。

如果选择使用 String 或 int，就需要确保字段只能有某些值。对于 iCoot，Reservation 有 6 个状态，因此必须选择 6 个字符串(waiting、notifiable、collectable、needingRenewal、storable 和 concluded)或 6 个整数(0~5)。从数据库的角度来看，也可以采用一种简单的方法：给 RESERVATION 表添加一列 STATE，把它的类型设置为 VARCHAR 或 INTEGER。(INTEGER 版本比较快，但较难调试)。这种方法如图 10-13 所示。

RESERVATION					
ID	CARMODELID	CUSTOMERID	NUMBER	TIMESTAMP	STATE
7	33	4	R187a	2004-12-06 14:23:16.543	0
1	22	2	R7b	2004-12-03 00:03:21.672	2
99	72	4	R459b	2004-12-05 09:45:07.210	0

图 10-13 把对象状态映射为一列

或者采用较复杂的方法：给每个状态引入一个新表，使用外键表示预约所处的状态。例如，图 10-14 显示了 6 个状态表中的两个，分别表示预约 7 处和 99 处于等待状态，预约 1 处于可取车状态。这个复杂的方法比较好。它是否比简单方法更高效取决于系统中的使用模式，这是很难预测的。

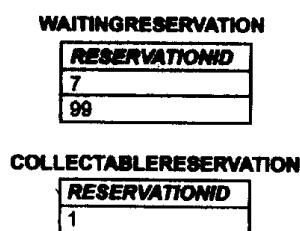


图 10-14 把对象状态映射为状态表

在存储状态时，有时必须处理状态属性——在业务对象处于特定状态下时与该业务对象相关的数据。例如，当某个预约处于等待状态时，需要知道该预约上次的更新时间(因为在等待一个星期之后，该预约就需要更新了)。同样，对于可以取车的预约，需要知道汽车何时处于可取车状态(因为如果顾客没有在三天内取走车，汽车就变成可存储状态)。

在使用状态表的方法中，可以把状态属性存储为额外的列，在图 10-15 中演示了这种方法，其中有两个示例状态。在使用状态列的方法中，必须在业务对象表中为每个状态属性添加一列，业务状态表中可以有许多列。每个列都必须是可空的，这样当对象不处于相关的状态下时，就可以省略数据。在图 10-16 中演示了这种方法，其中有两个示例状态。最好避免使用可空列，所以在这种情况下，使用状态表比较好。

WAITINGRESERVATION	
RESERVATIONID	LASTRENEWEDDATE
7	2004-10-10
99	2004-10-18

COLLECTABLERESERVATION	
RESERVATIONID	DATEPUTASIDE
1	2004-12-04

图 10-15 给状态表添加状态属性

RESERVATION				
ID	CARMODELID	CUSTOMERID	NUMBER	TIMESTAMP
7	33	4	R187a	2004-12-06 14:23:16.543
1	22	2	R7b	2004-12-03 00:03:21.872
99	72	4	R459b	2004-12-05 09:45:07.210

STATE	LASTRENEWEDDATE	DATEPUTASIDE
0	2004-10-10	NULL
2	NULL	2004-12-04
0	2004-10-18	NULL

图 10-16 把状态属性映射为可空的列

## 1. 映射继承

为了把继承层次映射到表上，可以为每个类引入一个表，其中的列对应于类添加的属性。这样，就可以找出对象的所有属性，表必须共享相同的主键。例如，图 10-17 显示了映射为三个表的 Customer 类层次，这三个表是 CUSTOMER 表、NONMEMBER 表和 MEMBER 表。CUSTOMER 表有三列，分别对应于 Customer 类定义的属性；NONMEMBER 表有一列，对应于 NonMember 类定义的属性；MEMBER 表有两列，对应于 Member 类定义的属性。

与通常一样，通用标识符(已添加到 Customer 类中)用作主键，这次它显示为三个表的主键。例如，在检索 Customer 33 的数据时，从 CUSTOMER 表中选择 NAME、AMOUNTDUE 和 PHONE；接着，测试在子类表中是否有键为 33 的一行(如果有，则 Customer 33 实际上是子类的一个实例)；在这个例子中，Customer 33 有 NONMEMBER 数据，所以选择其 License。最终结果是图 10-18 中的 NonMember。

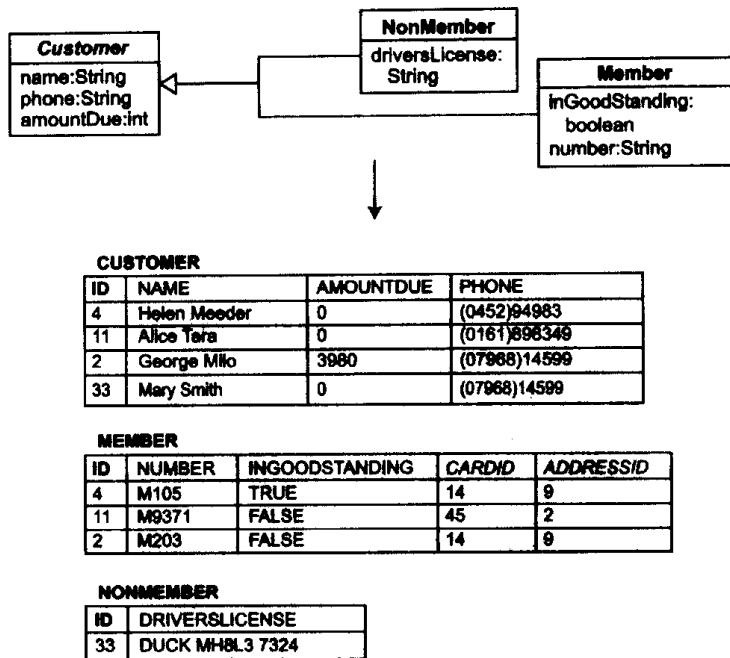


图 10-17 把继承映射为多个表

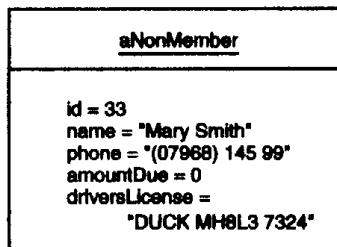


图 10-18 重新构造的非会员

与对象状态的映射一样，也可以把所有的属性放在一个表中(这说明，作为类实例的对象类似于处于特定状态下的对象)。例如，在图 10-19 中，有一个 CUSTOMER 表，它的 CLASS 列指定每一行的类。另外，该表有一个可空列，用于表示层次结构中的每个属性。

与以前一样，选择映射为多个表还是一个表取决于个人喜好、经验、预测和试验。

**CUSTOMER**

ID	NAME	AMOUNTDUE	PHONE	CLASS
4	Helen Meeder	0	(0452)94963	...
11	Alice Tara	0	(0161)898349	...
2	George Milo	3980	(07968)14599	...
33	Mary Smith	0	(07968)14599	...

**MEMBER**

NUMBER	INGOODSTANDING	CARDID	ADDRESSID	CLASS
M105	TRUE	14	9	...
M9371	FALSE	45	2	...
M203	FALSE	14	9	...
NULL	NULL	NULL	NULL	...

**DRIVERSLICENSE**

DRIVERSLICENSE	CLASS
NULL	0
NULL	0
NULL	0
DUCK MH8L3 7324	1

图 10-19 把继承映射为一个表

```

ADDRESS (ID:INTEGER, HOUSE:VARCHAR(99), STREET: VARCHAR(99), COUNTRY: VARCHAR(99), POSTCODE:VARCHAR(99))
CAR (ID:INTEGER, TRAVELED: INTEGER, DATELOST:DATE, CARDETAILSID: INTEGER)
CARD (ID:INTEGER, TYPE: VARCHAR(99), NUMBER: VARCHAR(99))
CARDETAILS (ID:INTEGER, BARCODE: VARCHAR(99), NUMBERPLATE: VARCHAR(99), VIN: VARCHAR(99))
CARMPDEL (ID:INTEGER, NAME: VARCHAR(99), PRICE: INTEGER, CARMODELDETAILSID:INTEGER, CATEGORYID:INTEGER, VENDORID: INTEGER)
CARMPDELDETAILS (ID:INTEGER, ENGINESIZE: VARCHAR(99), DESCRIPTION: VARCHAR(256), ADVERT: VARCHAR(99), POSTER: VARCHAR(99))
CATEGORY (ID:INTEGER, NAME: VARCHAR(99))
COLLECTABLERESERVATION (RESERVATIONID: INTEGER, DATENOTIFIED:DATE)
CONCLUDEDRESERVATION (RESERVATIONID: INTEGER, REASON: VARCHAR(99))
CUSTOMER (ID:INTEGER, NAME: VARCHAR(99), PHONE: VARCHAR(99), AMOUNTDUE: INTEGER)
DISPLAYABLERESERVATION (RESERVATIONID: INTEGER, REASON: VARCHAR(99))
INTERNETACCOUNT (ID:INTEGER, PASSWORD: VARCHAR(99), SESSIONID:INTEGER)
MAKE (ID:INTEGER, NAME: VARCHAR(99))
MAKECARMODEL (CARMODELID: INTEGER, MAKEID: INTEGER)
MEMBER (ID:INTEGER, NUMBER: VARCHAR(99), INDOODSTANDING:BOOLEAN, CARDID: INTEGER, ADDRESSID: :INTEGER)
NEEDINGRENEWALRESERVATION (RESERVATIONID: INTEGER, DATERENEWALNEEDED:DATE)
NONMEMBER (ID:INTEGER, DRIVERSLICENSE: VARCHAR(99))
NOTIFIABLERESERVATION (RESERVATIONID: INTEGER, DATEPUTASIDE:DATE)
RENTAL (ID:INTEGER, NUMBER: VARCHAR(99), STARTDATE:DATE, DUEDATE:DATE, TOTALAMOUNT: INTEGER)
RENTALCAR (RENTALID: INTEGER, CARID: INTEGER)
RESERVATION (ID:INTEGER, NUMBER: VARCHAR(99), TIMESTAMP: TIMESTAMP, CUSTOMERID: INTEGER, CARMODELID: INTEGER)
VENDOR (ID:INTEGER, NAME: VARCHAR(99))
WAITINGRESERVATION (RESERVATIONID: INTEGER, LASTRENEWEDDATE:DATE)

```

图 10-20 iCoot 的数据库模式

### 案例分析

#### iCoot 数据库模式

iCoot 已完成的数据库模式如图 10-20 所示。每个表都显示为名称后跟列类型(列类型位于括号中)。与以前一样，主键显示为黑体，外键显示为斜体(主 - 外键是粗斜体)。为了完整起见，这个模式包括了完整 Coot 系统的几个部分，例如 NONMEMBER 和 DATELOST。

## 10.4 最终确定用户界面

下面介绍用户界面的设计。前面在满足系统需求时，有一些粗略的草图。本节介绍如何为瘦客户机生成简单、优秀的界面。

在开发的早期阶段，也就是需求和分析阶段，考虑用户界面的功能是很有用的。这是因为在用例驱动的方法中，参与者与系统交互的方式是极为重要的，而且大多数参与者都是人。因此，应获得如下内容：

- 用户界面草图。这些有助于在用户的帮助下，在需求捕捉阶段生成系统用例。
- 通信图中的边界对象。在动态分析阶段，使用通信图显示用例的实现过程；在这些图中，每个参与者都通过边界对象与系统交互。

但仍需要设计用户界面：必须提取粗糙的边界对象，粗略的用户界面草图和准确的用例，把它们转换为可以直接实现的用户界面描述(对于我们访问的系统，应把系统对系统的界面设计为层的一部分(就是我们访问的系统)，或把其界面设计为业务服务的一部分)。

大多数软件系统都可以在不详细考虑用户界面的情况下(完成了系统用例，就可以只考虑系统内部的对象)成功地设计出来。这有两个主要原因：

- 系统的当前行为取决于其内部的构造，不取决于人们与它的交互方式。比如说，汽车由一个引擎、四个车轮和一个车身组成，根据我们使用的界面和使用它们的方式，可以让汽车把人从 A 送到 B，或者让它拖拽一辆大篷车，也可以让它撞向一堵墙，无论我们如何与汽车交互，它的行为仍是一辆汽车。
- 我们喜欢编写可重用的代码。如果关注一组特定界面的需求，生成的系统就只能用于这些界面——这是传统开发的一个问题：“今天解决今天的问题，不管明天的事”。用例驱动似乎与重用的原则相悖；但是，就系统的内部而言，用例只是确保开发人员不理会对非相关区域的一种方式而已。

本节不深入探讨人机交互的理论，而是介绍用户界面设计的一些基本原则(尤其适合于访问多层系统的瘦客户机)。

### 1. 以用例为指导

从系统的角度来看，用例只是使开发人员不越界。但对于用户，用例就是全部。所以，用例应用于给用户界面建立结构。

一般应尽可能使用例简单：它们应不包含太多的功能，不应难以管理。所以，尽管每个用户界面最好能表示许多用例，但界面本身仍可以是很简单的。

应组合相关的用例，在用户界面的结构中反映出来。例如，给 iCoot 顾客显示一个基于 Web 的用户界面，其中包括十多个用例。在这个简单的界面中，活动应组合为“预约”、“出租”、“浏览目录”等。

还应避免把单个用例或一组相关的用例分隔为多个界面。例如，对某种汽车型号感兴趣的顾客不应进入另一个不同的界面来预约它。

### 2. 简单

简单就其本身来说是一个好的原则。另外，许多用户是新客户，尤其是那些通过 Internet 访问系统的用户，所以简单是特别重要的。最好让专业用户觉得用户界面非常简单、整洁：不要让用户界面有太多的功能，以免降低其效率。

在电子商务环境下，我们不希望在购买某件商品之前，还需要进行学习。当顾客来到某个站点时，如果给他们显示非常长的使用说明，他们就会马上离开。要让顾客立即使用该站点，必须把详细的步骤缩减为“单击这里来购买”或“在下面输入信用卡号码，单击下一步”。

界面越简单，就越容易移植到更简单的平台上(例如移动电话、机顶盒、个人数字助理、家用设备)。要生成可移植的用户界面，就需要在设计中使用控制层。

### 3. 使用选项卡

应使用选项卡组合相关的用例。选项卡是一组一次只显示一页的页面，其他页面就通过页

边上的标签来显示。选项卡的优点是，其尺寸和位置是相同的，即使交互操作涉及到多个用例，用户也可以只关注屏幕上的一个区域。这提高了用户体验和用户效率。大多数 GUI 库都支持选项卡，例如 Java 的 Swing(图 10-21 显示了 iCoot 作为一个 Swing 小程序运行)。对于 HTML/CGI 界面，只能模拟一个选项卡，最终结果是类似的，但在移动到新的页面上时，屏幕会闪烁，而且延迟时间较长(图 10-22 显示了 iCoot 作为模拟的 HTML/CGI 选项卡运行)。

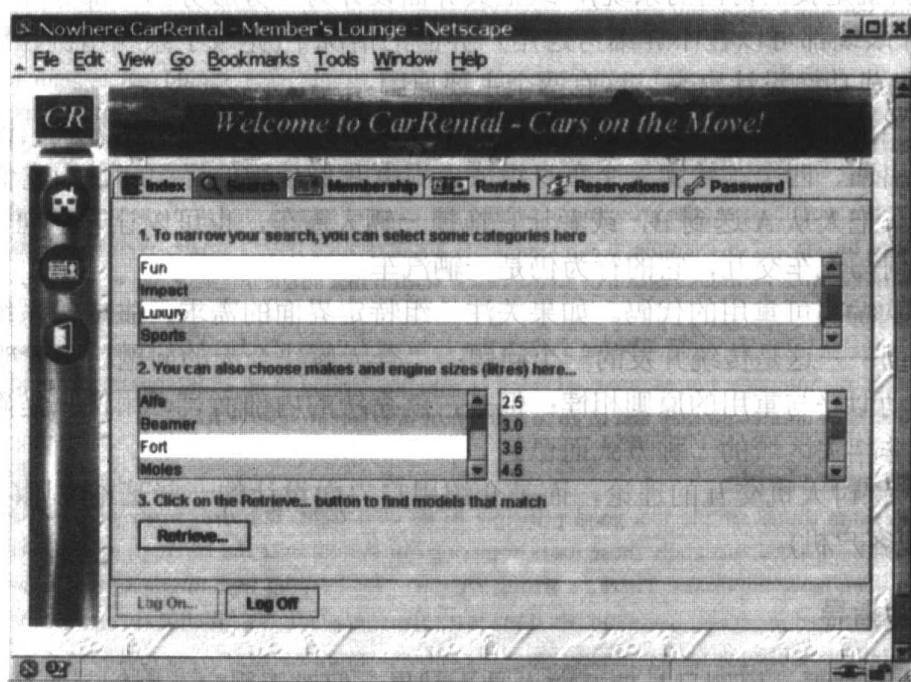


图 10-21 iCoot 使用 Swing 选项卡

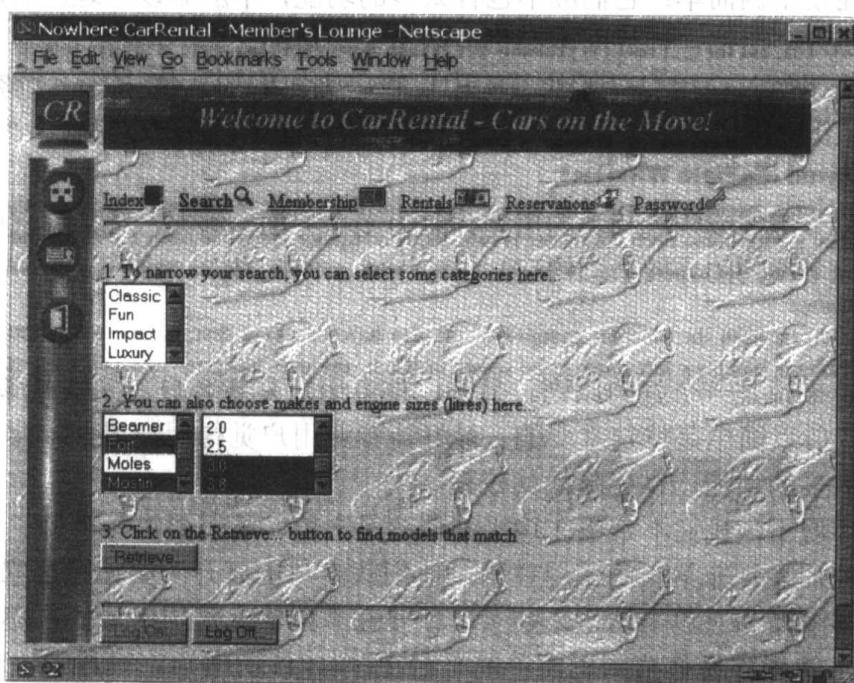


图 10-22 iCoot 使用模拟的选项卡

用例常常映射为一个页面。有了向导，即使复杂的用例也可以放在一个面板上。

#### 4. 使用向导

向导是一系列页面，用于引导用户按照步骤完成复杂的活动。当用户选择一个活动时，就显示向导中包含步骤 1 的页面；用户完成了步骤 1，单击 Next 按钮，进入步骤 2，依此类推。在完成活动之前，用户可以使用 Back 按钮重复前面的步骤。向导的每个页面都包含字段、列表和其他窗口小部件，用户使用它们输入当前步骤需要的数据。当然，每个页面都有简单的说明，例如“在下面的框中输入旧密码，然后单击 Next”。

向导的逐步引导和简单的说明，允许用户完成复杂的活动，且不需要记忆如何完成该活动。另外，向导与选项卡一样，也使用户只关注屏幕的一个区域。

与选项卡一样，如果使用纯 HTML/CGI 界面，就只能模拟向导，但最终结果比不使用它们好一些(如果不使用向导，HTML 页面就会很长，包含活动的所有步骤，用户必须滚动页面，才能执行各个步骤，非常不便)。

#### 5. 避免使用多个窗口

多个重叠的窗口是为了方便计算机专家而不是初级用户而开发的。即使用户对打开 Web 浏览器非常熟练，也不能指望他们能处理多个浏览器窗口或弹出对话框。

除了内在的复杂性之外，还必须注意，在一些平台上，多个窗口是没有意义的。在 iCoot 中，用户导航到某个汽车型号上，单击 Make a Reservation 按钮来预约它。预约汽车型号是要付预约费的：系统需要请求用户确认。多窗口系统的开发人员会很自然地弹出一个对话框。但是，iCoot 可能运行在电视(带有机顶盒)上。由于屏幕分辨率和空间有限，机顶盒不支持对话框，所以无法传送出消息(如图 10-23 所示)。在前面的页面上叠加对话框的内容，就可以避免使用多个窗口(如图 10-24 所示)。如果用户单击 No，系统就返回前面的页面(等价于取消对话框)；如果用户单击 Yes，就进行预约，系统返回前面的页面。

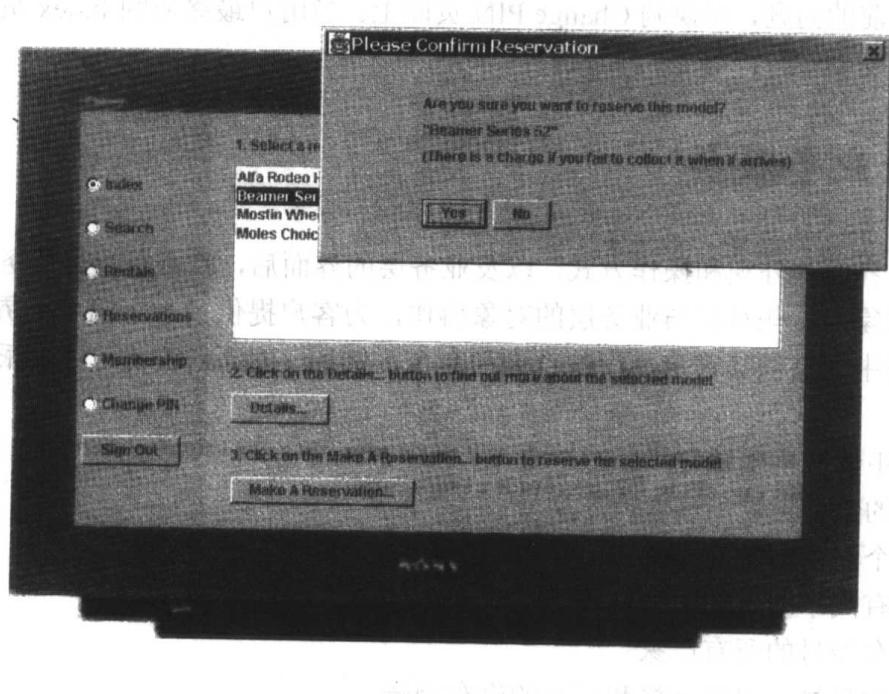


图 10-23 多个窗口的问题

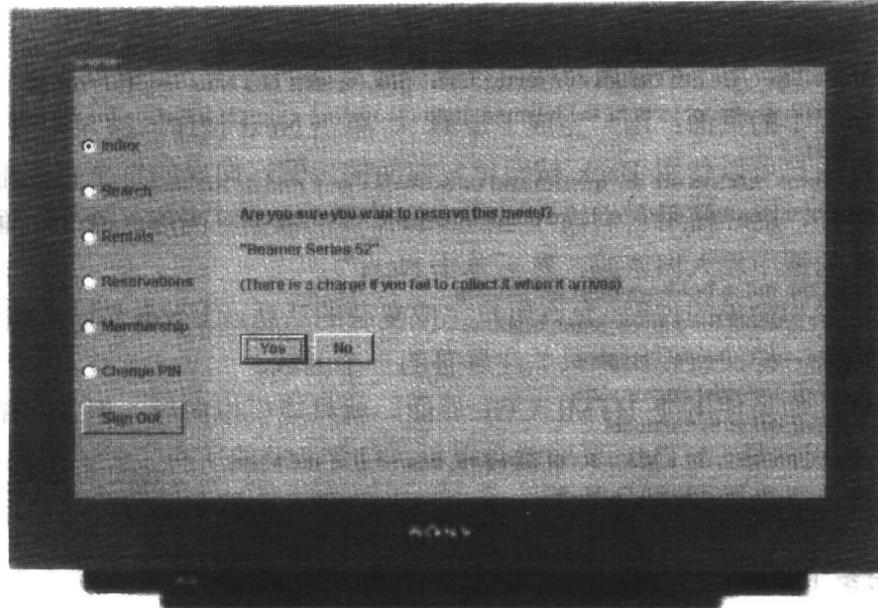


图 10-24 避免使用多个窗口

图 10-24 中的界面是运行在选项卡上的向导(wizard running inside a notebook): 屏幕左侧的单选按钮对应于选项卡的标签，允许用户在主要用例之间切换：在选项卡页面上，向导会引导用户完成“U7:进行预约”用例(这是“U1:浏览索引”的扩展)。选项卡和向导可以一起使用，把复杂的交互过程限制在屏幕的一个区域上，这样用户就不会觉得厌烦或犯糊涂。只要遵循“一次只做一件事”的原则，就可以完成得很好。甚至在按照这个规则设计用户界面时，仍可以允许用户在未完成的活动之间切换。例如，在图 10-24 中，用户可以忽略当前的问题，切换到 Change PIN 页面上。当用户最终返回 Index 页面时，再回答这个问题。

## 10.5 设计业务服务

确定了用户界面的外观和操作方式，以及业务层的界面后，就可以设计服务器层了。它包括许多服务器对象，这些对象与业务层的对象协作，为客户提供了一个简单的界面。

业务服务是中间层供其客户使用的查询和命令。例如，对于 iCoot，可以派生出如下的业务服务：

- 从目录中读取索引标题
- 从给定的索引标题中读取汽车型号
- 读取每个汽车型号的目录
- 读取所有汽车型号的引擎规格
- 读取汽车型号的所有厂家
- 读取给定目录、引擎规格和厂家的汽车型号
- 读取给定汽车型号的详细信息
- 预约汽车型号

- 读取给定会员的信息
- 改变会员的密码
- 读取给定会员租赁的汽车
- 读取给定会员的预约记录
- 取消预约

业务服务表示客户机上的用户界面和中间层上的业务逻辑之间的信息流小结。在遍历用例时，参考用户界面设计和系统体系结构，就可以生成业务服务。(把服务器的接口分解为它提供的服务，这种方法可以用于各种通信情形。例如，可以扩展本节讨论的内容，确定子系统中的层如何通信)。

为了使业务服务在 HTML/CGI 界面上有意义，必须假定用户界面位于客户层和中间层之间：HTML 窗体、服务小程序和 JSP 结果页面构成了一个用户界面，只是有些部分运行在中间层上。甚至可以在单独的中间层机器上执行服务小程序，该机器与进行业务处理的机器不是同一台机器，从而减少了业务对象上的负载——在这种情况下，运行服务小程序的机器是中间层的一个客户机，而不是中间层的一部分。所以，在下面的几小节中，术语“客户机”表示：

- 一段代码，例如应用小程序，运行在与业务层不同的单独机器上。
- 服务小程序，运行在单独的进程中，或运行在与业务层不同的单独机器上。

利用业务服务实现客户机-服务器通信，可以：

- 简化客户机代码。因为大多数客户机接口都只需要系统的一部分功能，每个接口可以只有系统功能的删节版本。
- 使业务层比较复杂。可以使业务层尽可能复杂、强大或可重用，即使业务层有变化，也不必担心对客户机的影响。
- 建立即插即用的服务器层。最好能提供不同类型的接口，来处理不同的用户喜好或客户机功能——例如，应用小程序/RMI 与 HTML/CGI-服务小程序，而不必每次都重新实现中间层服务。

### 10.5.1 使用代理和副本

在客户机请求业务服务时，结果可能包含一个或多个业务对象。例如，如果客户机请求是“读取汽车型号的所有 Make 对象”，结果就是 Make 对象的一个列表。除了检索业务对象之外，客户机还可以给服务器发送业务对象。例如，如果客户机请求是“取消预约”，就不需要指定预约。

至少在逻辑上，业务对象在客户机和服务器之间来回流动。如果处理的是一层系统，就没有问题：只需在运行时系统上传送对象指针，根据需要给对象发送消息即可。但是在处理网络时，事情就不是这么简单了。

在 RMI 等技术的帮助下，可以使消息通过网络在对象之间传送：每个对象都位于一个运行时系统内部，但消息可以从外部到达该系统内部。或者，可以在网络上传送对象的副本：每个运行时系统都有一个它需要的对象副本，在本地给它们发送消息。在子系统的层之间通信时，有类似的问题：是在层之间传送对象引用，还是传送对象副本？(应使用一个开放的层，以越过其边界来传送引用，而使用封闭的层可以传送副本)。

为了更加精确，有两个基本的策略：代理(proxy)是一种客户机对象，它知道如何把接收到的消息传送给位于其他位置的业务对象。副本(copy)是包含业务对象数据的客户机对象。

使用代理的主要优点是：

- 所有的客户机都会看到相同的对象，所以它们总是在处理相同的信息。
- 所有的运行时系统都合并为一个统一体：分布式对象的处理方式与本地对象相同。

代理的主要缺点是：

- 业务层对象必须确保其安全。例如，在从客户机上接收消息时，必须确保客户机有发送消息的权限。这会使业务层比较复杂(另一种安全模式较容易实现：业务层对象只能由服务器层访问，服务器层必须保护其业务服务)。
- 增加了网络通信量。在面向对象的系统中，一般要先找到对象，再给它发送消息；使用代理技术后，对象的内部检索很快，但给对象发送消息较慢(因为每个消息必须在网络上传送)。
- 增加了中间层的负担。运行在中间层上的业务层一直在执行所有的方法。

使用副本的优点是：

- 减少了网络通信量：要检索更多的信息(所有对象的数据)，但方法执行得很快(因为它们在本地运行)。
- 处理并不都在中间层上进行，因为一些业务方法在客户机上执行。
- 对业务方法的直接访问不必保护其安全，因为只有服务器层有直接访问权限，服务器层本身就是安全的。
- 对象的实现不必是并发安全的：一旦以客户机的名义创建了一个副本，该客户机就对副本有独占访问权。

副本的主要缺点是：

- 要复制的数据太多。例如，如果复制一个汽车型号，就可能需要复制所有的 CarModelDetail 对象、所有的 Make 对象、所有的 Reservation 对象、进行了预约的所有 Member 对象等。
- 副本的步调可能不一致，因为每个客户机都有对象的一个独立副本。

对于某些应用程序，客户机副本已过时。例如，如果用户从一个 Web 浏览器上发出搜索引擎请求，所显示的结果页面会随着 Web 页面的增加、更新和删除而变化。严格说来，用户必须注意，结果只有在执行查询的那一刻是准确的。

通过仔细的编码，可以缓解代理和副本的一些问题。例如，在使用代理时，可以在本地高速缓存一些对象的数据，减少网络通信量；采用某些复杂的机制，可以确保对象的步调保持一致。使用副本时，可以根据需要复制引用的对象，减少复制过多的问题。例如，在读取汽车型号时，客户机可以只显示顶级属性——引用的对象仍为 null，除非客户机导航到这些对象(使用封装，这个过程可以对客户机代码透明化)。

如果幸运，还可以访问支持可配置代理、可配置副本(或包含两者)的框架或库。但是，即使有这种技术，开发人员仍必须选择是使用代理还是副本。

另外，只要加上一点手工，就可以采用一种更简单的方法：轻型副本(*lightweight copy*)。在这种方法中，当客户机请求业务对象时，它只显示需要的基本信息，如果客户机已经有了该信息，信息就不会传送给客户机。换言之，当客户机需要给服务器标识业务对象时，客户机将只给服务器传送对象的通用标识符。

为了使轻型副本工作，需要业务服务(以便指定客户机需要的信息)和通用标识符(以便客户机有效地把对象传送回来)。传送给客户机的信息包括业务对象的简单属性副本(数字、字符串等)和引用属性的通用标识符，以便客户机进一步的导航。

## 10.5.2 给业务服务分类

最好将业务服务组合为相关行为组(换言之，就是对象上的消息)。这些服务器对象有几个自己的属性，它们不应以客户机的名义记录状态(每个客户机都必须记忆它自己的状态)。服务器对象驻留在多层系统的服务器层上。

为 iCoot 设计的服务器对象如图 10-25 所示。业务服务分类为预约、身份验证、会员信息、目录信息和租赁(分别是 ReservationServer、AuthenticationServer、MembershipServer、CatalogServer 和 RentalServer)。

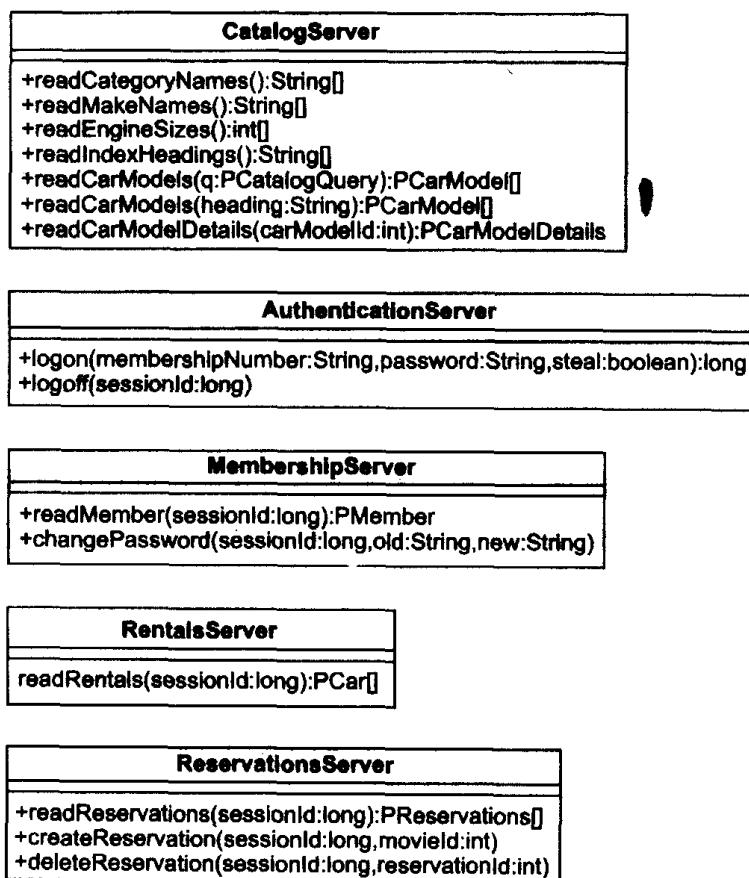


图 10-25 iCoot 的服务器对象

图 10-25 中的业务服务消息可确保客户机和服务器都不能给另一端传送不必要的信息。例如，考虑 CatalogServer。为了搜索汽车型号，客户机必须先检索所有的目录名称、引擎规格和厂家名称。这些从服务器返回为 String 对象和 int 对象。

接着，客户机允许用户建立查询(例如，“由 Alpha Rodeo 或 Beamer 生产的所有跑车”)。客户机把查询传送给服务器，作为 PCatalogQuery。在 PCatalogQuery 中，有三个数组属性，分别标识用户感兴趣的目录名、引擎规格和厂家名。服务器返回一个匹配 PCarModel 对象的数组(P 表示协议——它有助于服务器对象的开发人员避免名称与重型业务对象名混淆。使用数组是因为它们比集合更紧凑)。

为了减少返回给客户机的信息量，PCarModel 对象不包含任何细节，只包含价格、型号和通用标识符。当用户请求某个汽车型号的细节时，客户机不是传送整个 PCarModel(服务器必须有它的属性)，而是只传送通用标识符，作为 readCarModelDetails 消息的一个参数。

### 10.5.3 会话标识符

为了防止黑客的攻击，常常需要限制客户机对权限服务的访问。例如，对于 iCoot，就有受限服务，如“预约汽车型号”，它需要客户登录；还有无权限的服务，如“从目录中读取索引标题”，它可用于每个人，甚至是黑客。

一些客户机-服务器协议，如 HTTP，有一个标准的挑战(challenge)机制，用于给用户显示登录屏幕：输入的用户名和密码要在服务器上进行验证，如果验证成功，就给客户机提供一个惟一的会话标识符。但是，如果要提供即插即用的业务服务，例如处理任意前端的服务，就必须自己实现一个机制。图 10-25 显示了实现它的一种方式。所有权限服务都把一个 long 名称作为其第一个参数：这一定是服务器层创建的会话标识符，否则权限服务就会失败。为了获得会话标识符，客户机必须使用 AuthenticationServer 上的 logon 消息(steal 参数由客户机用于指定是否应终止这个会员的已有会话——这是单登录机制的一部分)。

在调用 logon 方法时，AuthenticationServer 会使用业务层检查会员号和密码。如果客户的凭证是正确的，AuthenticationServer 就生成一个随机号，并把它关联到业务层的对应 Member 对象上。随后，只要调用了权限服务，相关的服务器对象就可以使用会话标识符查找 Member 对象，并进行处理。自然，如果最初的凭证或会话标识符无效，客户机就会接收到错误消息。会话标识符必须很难伪造——随机生成的 64 位号码就足够了。黑客一般不会猜测 64 位随机号码，因为他们成功的机会太渺茫了。

提供可移植的、有限制/无限制的机制的另一种方法是，把会话标识符设置为一个对象。通过封装，在选择采用难以伪造的信息时会有更多的灵活性。

### 10.5.4 业务服务的实现

设计好业务服务(服务器对象上的消息)后，就需要根据业务层确定它们的实现方式。为此，需要遍历用例，绘制顺序图，显示出需要发送的消息。这个过程称为业务服务的实现(business service realization)。它类似于分析阶段中用例的实现：遍历用例，绘制通信图，描述出业务对象支持的实现方式。要描述业务服务的实现，应使用顺序图，而不是通信图，因为顺序图更紧凑(这里专指实现方式，所以有更多的信息显示出来)。

图 10-26 显示了 AuthenticationServer 的 logoff 方法的顺序图。顶部是交互过程中涉及到的对象。与通信图和对象图中的对象不同，顺序图中的对象没有带下划线的标签。时间轴从上指向右，所以首先，顶部的 Member 参与者通过 AuthenticationServlet 启动注销，AuthenticationServlet 再把 logoff 消息发送给 AuthenticationServer，AuthenticationServer 把 findByName 消息发送给 MemberHome，依此类推。

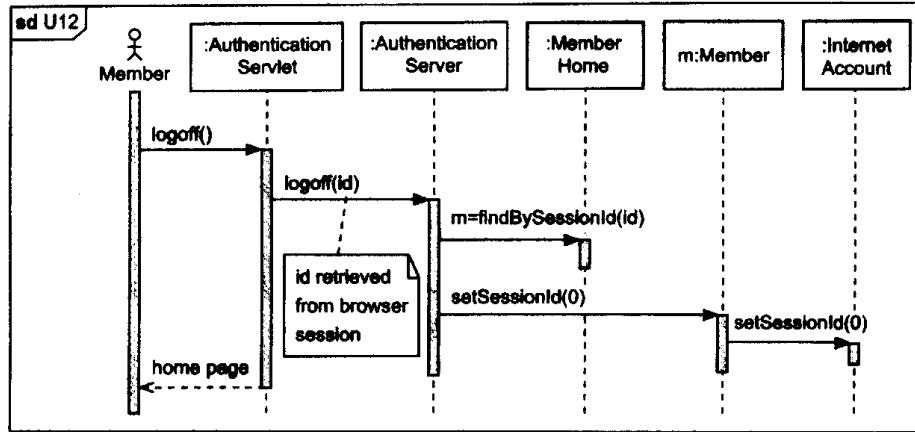


图 10-26 描述业务服务的实现

顺序图上垂直的虚线称为生命线(lifeline，在手工绘制的图中，这些线条不是虚线)。垂直的灰色矩形称为活动条(activation bar)，它们表示对象何时执行方法(在手工绘制的图中，这些活动条可以是白色的，或者省略)。每个顺序图都可以放在一个方框(frame)中，这个方框在左上角有一个操作符，就像前面设计类图那样，区别是操作符不同(所有的 UML 图都可以放在一个方框中，操作符表示图的内容)。在顺序图中，操作符是 sd 后跟顺序图名。在类图中，使用 pkg 操作符，表示包含类的包。在本书中，只有 iCoot 的顺序图和设计级类图才加方框。方框还可以用于包含循环和对其他顺序图的引用(参见附录 3)。

### 案例分析

#### iCoot 的顺序图

iCoot 小组的一个成员描述了如图 10-26 所示的交互过程，如下所示：

当 Member 参与者选择注销时，其浏览器就会告诉 AuthenticationServlet 要注销。接着，AuthenticationServlet 把注销消息以及它从浏览器会话中检索到的 Member ID 发送给 AuthenticationServer。

AuthenticationServer 使用 ID 从 MemberHome 中检索对应的 Member，再给该 Member 发送 setSessionId 消息，并把 0 作为参数，该消息传送到 Member 的 InternetAccount 上。InternetAccount 存储会话标识符 0，表示 Member 已注销。

最后，给 Member 参与者显示主页，以便他们访问顾客服务或再次登录。

对于业务服务的实现，需要显示在服务器对象和业务层对象之间流动的消息，但不需要显示业务层对象的内部工作过程。根据系统的规模，在实现图上显示每个业务服务有时是可行的，有时不可行。一般应显示最重要的业务服务——用例有助于确定哪些业务服务是最重要的。在每个用例中，都有几个正常的业务服务和几个异常的业务服务。从这里的情况来看，显示最常见的业务服务的正常路径就足够了。

在绘制顺序图时，会发现业务层对象的接口上有越来越多的消息。业务层对象上产生的消息，以及利用直觉、经验、库、模式、框架和推测产生的其他消息会发送到规范阶段。在规范阶段，应尽力完成对象接口，描述需要的行为(参见第 12 章)。

## 10.6 使用模式、框架和库

模式(参见第 11 章)是小型编程问题的可移植解决方案——在面向对象的环境中，模式是一小组协作对象。模式可以使开发人员工作得更快，生成更好的代码，且没有重复劳动。本书的其他地方已提及一些基本模式：单一模式(类只有一个实例)、构造模式(用于创建对象)、家庭模式(用于创建对象，查找已有的对象)和状态模式(表示对象的生命周期)。每个模式都有名称、描述和一些使用示例。每个开发人员都应熟悉常见的模式。

框架(framework)在一些方面类似于模式：它是把系统的各个部分组合在一起的方式。但是，框架有两个重要的区别：第一，它们比较大；第二，一些代码已经编写好了(其形式为部分实现的类或代码生成工具)。与模式一样，应找出已为特定的问题域设计出的框架，以避免编写不必要的代码。

一种非常全面的流行框架是 Enterprise JavaBeans。EJB 框架允许开发人员构建中间层业务逻辑，且不必编写任何代码来处理持久性、事务、安全、并发性、发布和线程安全。尽管这是一个大框架，但值得好好了解一下。

库是可以随时使用的预先编写好的类集。应了解可用于问题域的库，这样，如果库中有合适的类，就不必编写新类了。Java 2 Platform 库就是一个这样的库，它有三个变体：企业版、标准版和微型版，分别适用于大型(如电子商务)、中型(如桌面排版系统)、和小型(如 PDA 电子邮件客户机)系统。

## 10.7 事务

每个并发系统要强壮，就必须围绕事务来建立。事务也称为工作单元，用于分隔一组数据库访问(数据库访问表示“读取一些数据”或“写入一些数据”)。事务是相当复杂的，其详细论述超出了本书的范围。

事务用于确保：

- 数据库中的信息不会被系统问题破坏。我们要确保数据库从一个统一状态切换到另一个统一状态下：数据永远不会部分更新，它要么全部更新，要么完全不更新。
- 客户机不会获得过时的信息。要避免下述情况：客户机 A 读取一个顾客的地址，客户机 B 修改该顾客的地址，客户机 A 根据旧地址执行某个动作。

数据库客户机启动一个事务，访问一些数据，然后提交事务。如果这个提交成功了，从事务开始以来进行的所有更新就发送到数据库中(如果事务仍是激活的，更新就处于未决状态)，客户机就可以肯定它操作的是最新的信息。如果提交因系统问题或另一个事务的崩溃而失败，客户机就可以回滚事务，取消所有未决的更新(因此，以最新信息为基础更新的数据会保持不变)。因为 DBMS 保证事务中的访问要么完全成功，要么完全失败，所以数据库会从一个统一状态切换到另一个统一状态下。显然，DBMS 必须确保所有的访问都在一个事务内部进行。

事务可以非常短——只访问一行，也可以非常长——对相关行的几个访问。DBMS 一般允许客户机选择短事务或长事务：在默认情况下，对数据库的每个访问都在新的(短)事务中进行；另外，客户机也可以选择手工启动、提交、回滚(长)事务。在理论上，事务可以嵌套：这就允许把小型工作单元封装在大型工作单元中。但在实际上，大多数关系 DBMS 都不支持嵌套的事务。

### 10.7.1 保守并发和开放并发

并发控制(使用事务控制多个客户机对数据的同时访问)可以是保守的，也可以是开放的。在保守并发中，DBMS 确保当一个事务处于激活状态时，没有其他事务能进行有冲突的访问。在开放并发中，事务可以随意访问，但在提交事务时，DBMS 会检查是否有其他事务在执行有冲突的访问。

在默认情况下，关系数据库使用保守并发。开放并发在面向对象的数据库中比较常见，但有时它是位于关系数据库顶部的面向对象框架提供的一个选项(为关系数据库实现开放并发的一种方式是，在更新一行之前重新读取它：如果该行中的数据已改变，就说明另一个事务改变了它，但是，这只是一个部分解决方案)。

为了实现保守并发，关系 DBMS 要锁定在事务中访问的数据，直到该事务完成为止。例如，如果事务修改了账户 123 的余额，DBMS 就锁定包含新余额的行；在第一个事务结束(和解锁)之前，其他事务都不能读取该行。锁定也可以应用于读取，例如，如果事务读取账户 456 的余额，在第一个事务结束之前，其他事务就不能修改包含旧余额的行(否则，第一个事务就会对旧值进行一半工作，对新值进行另一半工作，这是没有意义的)。

由于性能的关系，关系 DBMS 允许客户机放松锁定模式的某些限制，但语义的准确性有一定的损失。

### 10.7.2 使用事务和对象的一般规则

在从面向对象的模型映射为关系数据库时，会有一些利益的冲突。一方面，我们有一个复杂的对象图，想在一个长事务中处理整个块；另一方面，关系数据库在默认情况下有一个保守锁定机制，对象数据散布在许多表中：所以应使用短事务，避免锁定数据库的大部分数据(这就是开放并发在面向对象的数据库和面向对象的框架中比较常见的原因)。

要避免这种冲突，需要技巧和经验。如果使用专业的框架，如 EJB，它就会解决大部分问题。如果手工进行映射，就应考虑下面的建议：

- 组织对象和访问路径，以减少重叠。例如，在 iCoot 中，可以确保每个会员都只登录一次。因此，重叠发生的次数就会很少：助手访问会员的数据，同时该会员在登录时，才可能出现重叠。
- 使用主键，使数据库访问都有各自的主题。大多数关系 DBMS 的麻烦在于，如果不能确定要访问哪一行，就锁定整个表。例如，如果 ID 是 CUSTOMER 表的主键，像“获取顾客 789 的姓名”这样的查询就只锁定一行，而涉及非主列的查询，如“获取姓为 Bloggs 的顾客”就可能锁定整个表。所以，确保为每个实体选择一个主键，并告诉数据库，然后尽可能根据该主键进行数据库访问。
- 使事务尽可能短。我们希望控制事务的开始和结束，这样一次可以进行好几个对象访问，但不要过度。

### 10.7.3 上层中的事务

事务有一个连锁反应：它们通常位于数据库层中，但会出现在持久层中。一旦它们出现在持久层中，通常就会出现在业务层中；一旦它们出现在业务层中，服务器层的开发人员就必须理解它们，了解如何正确使用它们。

在退出客户机之前，事务一般不能隐藏，为此，服务器层必须把事务封装到简化的请求(业务服务)中。

## 10.8 处理多个活动

正常情况下，在使用计算机时，希望同时完成几件事——写信、读电子邮件、运行很长的计算过程、浏览 Web。我们还希望服务器能同时完成上千件事(同时处理多个客户机的请求)。为此，大多数操作系统都允许执行多任务：每个程序在受保护的区域内利用其代码(程序指令)和数据(程序变量)运行为一个独立的进程。

一些编程语言允许在一个进程内执行多个任务。这些任务通常称为执行线程，或简称为线程。正常情况下，每个线程表示程序内部的一个活动，它和其他活动一同进行。

本节将探讨有关多线程的问题，以及如何确保代码的线程安全(这对于业务层来说是最重要的)。

### 10.8.1 控制多个任务

操作系统管理的每个进程都可以是空闲的(也许在等待用户的输入)，也可以是活动的(执行某些计算)。进程数常常超过 CPU 数，所以操作系统必须在活动的进程之间分配 CPU 时间：操作系统允许每个进程执行一小段时间，再转而执行下一个进程。这种时间分段(time-slicing)由一个软件控制，称为调度器(scheduler)。我们不需要知道调度器用于给进程分配 CPU 时间的算法，只需假定每个进程都会获得它需要的时间段即可。大多数操作系统还有一个额外的功能，即给每个进程分配优先级，这样一些进程获得的可用时间段就比另一些进程多。例如，给鼠标点击的检测分配高优先级，给用户应用程序分配低优先级。

任何像样的操作系统都会禁止一个进程访问其他进程的代码和数据，但程序员应确保妥善地管理对外部资源(例如文件和数据库)的访问。例如，在字处理程序打开一个文件时，它就可以锁定该文件，防止其他进程编辑该文件。对高度共享的资源，如数据库，的并发访问一般通过事务管理和业务规则来控制。

从用户的角度来看，多任务允许在桌面计算机上同时打开多个应用程序。可以随意在这些应用程序之间切换，一次完成一件事，或者同时开始几个任务，每个任务看起来都是独立完成的。多任务的优点还有，开始一个较长的计算后，不必等到它完成，就可以干其他事情。例如，在等待 Internet 搜索完成的同时，可以查看时间或收集消息。

从服务器的角度来看，多任务除了能同时给许多客户机提供服务之外，还可以提供更高的通信量(服务器更高效地处理客户机)。例如，假定脚本 search.pl 用于通过 HTML/CGI 执行 Internet 搜索。一些由客户机启动的搜索执行得很快，只需要几毫秒，而一些搜索需要几秒钟。如果一个客户机启动了一个时间较长的搜索，接着另一个客户机启动了一个较简单的搜索，则第二个搜索会立即执行，无需等待第一个搜索完成。

### 10.8.2 控制多个线程

线程不同于进程，是因为它们在其进程中都共享相同的数据区域。因此，除了保护外部资源之外，程序员还必须保护内部数据(每个进程内部的代码区域一般由运行时系统隐藏在线程

中，所以不需要对保护它采取什么措施)。在其他方面，线程都只是微型进程；它们由调度器控制，可以给它们分配不同的优先级。

从客户机的角度来看，采用多线程有如下优点：

- 用户可以同时运行许多应用程序，在一个应用程序中做许多事情。例如，在一个电子邮件进程中，可以编辑消息，当收到新邮件时获得通知，查看实时钟表等。
- 即使应用程序正在忙碌，用户也可以与用户界面交互。例如，假定在数据库查询工具中，用户输入了一个查询，按下 **Retrieve** 按钮；接着，在执行查询的同时，用户注意到查询中有一个拼写错误。如果查询工具只有一个线程，在返回并显示无效的结果之前，用户就不能编辑查询。另一方面，如果把用户界面和计算机查询放在不同的线程中，用户就可以在第一个线程完成之前执行另一个搜索，应用程序会立即删除不正确的线程，不正确的结果也不会显示出来。
- 即使应用程序正在忙碌，用户界面也可以更新。考虑一个运行为单个线程的查询工具。如果用户启动一个搜索，然后在结果显示出来之前，重新设置应用程序窗口的大小，会发生什么？操作系统(桌面)会抓取窗口的一角，在屏幕上移动它，这样窗口的边界就会按照希望的那样移动。但是，应用程序必须绘制窗口的内容：因为应用程序在忙碌，窗口的内容要在查询结果返回来之后再重新绘制。用户会看到一个相当业余的用户界面，它会在不恰当的时间重新绘制。如果查询使用独立的线程，用户就可以在等待的同时重新设置窗口的大小，重新绘制窗口的操作也会立刻执行。

从服务器的角度来看，采用多线程的优点是：

- 它允许同时给许多客户机提供服务，且系统没有多个进程的开销。进程的启动、执行和卸载比线程更昂贵。例如，要求一台机器同时运行 1000 个进程，该机器就会崩溃，而要求该机器运行 4 个进程，每个进程有 250 个线程，就不会出问题。对于某些联网的应用程序来说，这是至关重要的，例如，服务小程序在一个进程上运行，该进程有多个线程，但 CGI 脚本在默认情况下运行在多个进程中，所以，如果要利用该服务小程序的优势，就必须采用多线程。
- 它会减少服务器上的空闲时间。例如，如果中间层机器使用一个服务器线程访问数据库服务器，当查询在数据层机器上执行时，该机器就处于空闲状态。而使用多个线程，中间层机器就可以在执行查询的同时干其他事情。
- 它会减少超时现象。在一些协议中，如果服务器在一定的时间(如 2 分钟)内没有响应，客户机请求会自动失败。如果所有的客户机请求都必须排队，等待单个服务器线程的服务，超时现象就比较多(每个请求都要等待队列中排在它前面的请求获得服务之后，才能获得服务)。而在多线程中，短请求会有较快的响应，只有网络问题、服务器过载和过于复杂的请求会有超时现象。

理想情况下，编程语言及其运行时系统处理多线程的大量信息——调度、优先级、时间分段等。这样，程序员只需编写由线程运行的代码，并执行它们即可。

### 10.8.3 线程安全

多线程会出问题，因为线程在完成之前会被干扰(允许其他线程运行)。例如，考虑下面的情况：

两个线程 A 和 B 都访问对象 O。

线程 A 开始使用获取器方法读取 O 的一个字段 F。

在 A 读取了值的一半时，调度器干扰了它，让 B 运行一段时间。

B 开始通过其设置器修改 F。

调度器允许 B 完成其修改，之后唤醒 A。

A 被唤醒后，读取 F 的剩余内容。

线程 A 读取了一半旧值和一半newValue，这显然是没有意义的。这种数据损坏也会出现在外部资源上(假定 A 读取一个文本文件，B 则修改它)。

在访问数据库中的数据时，DBMS 会提供一个复杂的事务机制，以确保数据不被损坏。但是，在多线程代码的内部，我们必须自己保护数据。在面向对象的程序中，保护数据的关键是确保每个数据都只能通过一个管理该数据的对象来访问。于是，只要确保一次只有一个线程访问对象(互斥，*mutual exclusion*)，数据就是安全的。编程语言还允许强制互斥操作(稍后介绍如何在 Java 中实现互斥操作)。

在多线程中能安全使用的代码称为线程安全或 MT 安全(与非线程安全或 MT-hot 相反)。一般说来，所有的对象都应是线程安全的，所有的应用程序都应采用多线程。

## 1. 不变性

不变的对象就是数据不会改变的对象。其中术语“数据”表示：

- 对象的字段值
- 存储在外部资源中、由对象管理的值(例如文件中的文本)
- 对象中由该对象指向的值
- .....

换言之，对于不可变对象来说，不可能改变对象拥有的字段和该对象可直接或间接访问的内部或外部数据。

不可变对象的优点是，它们都是线程安全的——因为不能改变任何数据，也就不可能破坏任何数据。不可变对象也比较高效(它们可以透明地共享，存储在内存的只读区域中)。

一些语言提供了确保对象不可变的功能，例如 C++ 的 `const` 关键字和 Java 的 `final` 关键字。但是这些功能是局部的。较好的方法是通过编程样式强制对象的不可变性(不提供设置器，锁定文件等)。

只管不可变对象很不错，但大多数对象都需要改变。例如，Customer 不允许改变其 `address` 属性，就没有什么用。所以，必须了解如何使可变对象是线程安全的。

## 2. 固定值

确定如何确保对象的线程安全是一个挑战。更糟糕的是，通常必须找出对象的整个系列，确定它们是如何在一起使用的。其原因之一是死锁。死锁是指，线程 A 等待线程 B 完成某个任务，而线程 B 在等待线程 A 完成另一个任务，结果，两个线程会无限制地等下去。为了避免死锁，必须考虑对象是如何协作的，线程是如何在它们之间分配的。

获得线程安全的一个简单技巧是查找对象中的固定值。固定值是一个不可变的字段。例如，Math 对象中有一个 Pi 值，它从来不需要改变。不可变的固定值是自动线程安全的，所以像前

面的“读取被中断”就不会出现。

确定了对象的哪些字段是固定的后，就可以把对象分成两个部分：一是固定值，它们不需要特定的代码来保护，一是可变的值，它们需要特定的代码来保护。固定值只需在创建了对象后是不可变的，换言之，可以在对象的构造函数中操纵固定值：只要在构造函数执行完后不改变其值即可。其原因是只有一个线程能进入构造函数：该线程请求运行时系统创建对象。在构建对象时，其他线程都不能进入该对象，因为此时对象还不存在(假定构造函数在执行过程中，没有使对象可用于其他线程)。

### 3. Java 中的同步

把每个共享的资源封装到一个对象中，就可以解决大多数多线程问题。接着，对象负责确保一次只允许一个线程进入。编程语言应支持这个互斥限制。

例如，在 Java 中，方法可以标记为同步：运行时系统保证在对象的同步方法中，一次只能激活一个线程。这是在监视器的控制下，给每个对象关联一个锁来实现的。监视器允许到达对象的一个同步方法的第一个线程进入，其他线程被锁在同步方法的外部，直到第一个线程离开为止。Java 的互斥限制没有应用于非同步方法，线程可以随时出入非同步方法。

图 10-27 显示了 Java 对象的使用情况，其中有四个线程试图进入该对象。这个对象有需要保护的 MT-hot 值和不需要保护的固定值。名为 T1、T2 和 T3 的三个线程当前是激活的，T2 在方法 M2 的外部，因为 T1 已经通过另一个同步方法 M1 进入对象。为了使这个模式起作用，程序员必须确保只有 M1 和 M2 中的代码能访问 MT-hot 值。另一方面，固定值可以在任何方法中访问。

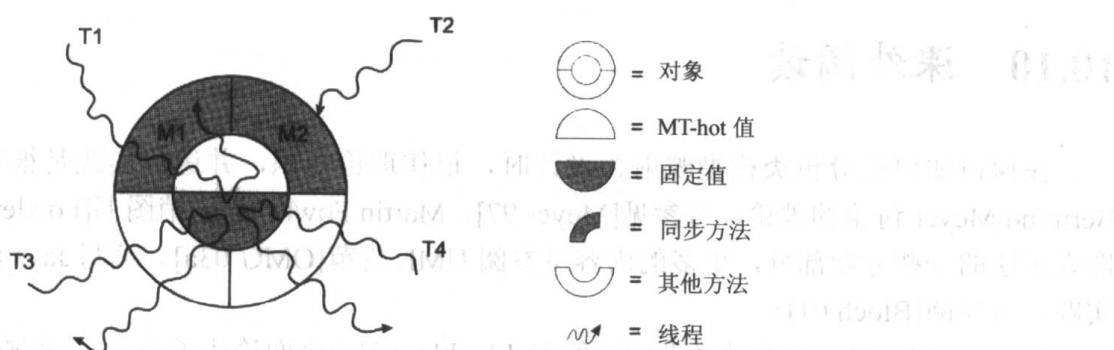


图 10-27 Java 中的同步

因此，为了确保 Java 对象是线程安全的，需要同步对所有 MT-hot 值的访问。实际上，这需要经验和慎密的思考，才能避免死锁和不必要的同步(这是很重要的，因为互斥限制可以减少对象的通信量)。

### 案例分析

#### iCoot 中的线程安全

如何确保 iCoot 的线程安全？可以分别考虑每个层(使用层的另一个优点)：

- 为了符合服务小程序的编程样式，服务小程序(分布式接口的一部分)是无状态的，所以是多线程安全的。会话数据(例如当前用户的 PMember)存储在每个客户机的一个 HttpSession

中，使用 Java 的同步机制来保护(使用同步块)。(作为标准 HTML/CGI+服务小程序机制的一部分，Web 服务器存储会话对象，Web 浏览器存储会话标识符)。

- 即插即用的服务器对象也是无状态的，因此是线程安全的：每个业务服务都返回一个响应(作为协议对象)，该响应可以从业务层中检测到，只由请求它的客户机使用。

- 通过仔细的编程，确保业务层是多线程安全的，使多个线程可以通过它在服务器层上运行，而不会破坏从数据库层读取的缓存数据。

偶尔数据库层因为有事务机制，所以在默认情况下是并发安全的。程序员只需确保事务在每个业务服务的开头创建，在业务服务结束时提交。

## 10.9 小结

本章介绍了子系统的设计——即确定要实现什么对象，这些对象有什么接口。

- 陈述了业务层的设计，确定如何从分析类模型中派生出它来。
- 介绍了对象模型如何映射到关系数据库模式上。为了简单起见，没有详细论述如何编写代码，在运行期间执行实际的映射。
- 简要介绍了设计用户界面的技巧之后，讨论了如何把中间层提供的功能组合到业务服务类上，隐藏业务层的复杂性(便于不同类型的用户界面使用)。
- 阐述了查找模式、库和框架，以避免编写新代码的重要性。
- 讨论了数据库事务和多线程(进程之间的并发)的问题，涉及到的概念，并举了 Java 中的互斥限制的例子。

## 10.10 课外阅读

在探讨如何把分析类模型映射为设计时，记住理论知识，并进行实践是很重要的。关于 Bertrand Meyer 讨论的理论，可参见[Meyer 97]。Martin Fowler 的畅销图书[Fowler 03]解释了类图表示法的一些专业部分，更多的内容可参阅 UML 规范[OMG 03a]。编写 Java 源代码的最佳实践，可参阅[Bloch 01]。

Scott Ambler 是一位方法学专家，他在 [Ambler 03]中全面论述了对象-关系映射。

在[Constantine 和 Lockwood 99]中，Larry Constantine 根据系统的使用方式提供了设计用户界面的建议(当然是基于用例)。

J2EE 涵盖了多层设计和实现的所有方面，从 GUI 和 HTML 前端到中间层上的服务小程序和 EJB，工具自动生成的对象一关系映射等。用于 J2EE 的模式在[Alur 等 03]中进行了描述。

对 Java 中线程安全和一些可重用模式的讨论，可参阅[Lea 99]。

## 10.11 复习题

1. 图 10-28 显示了什么图？(单选题)

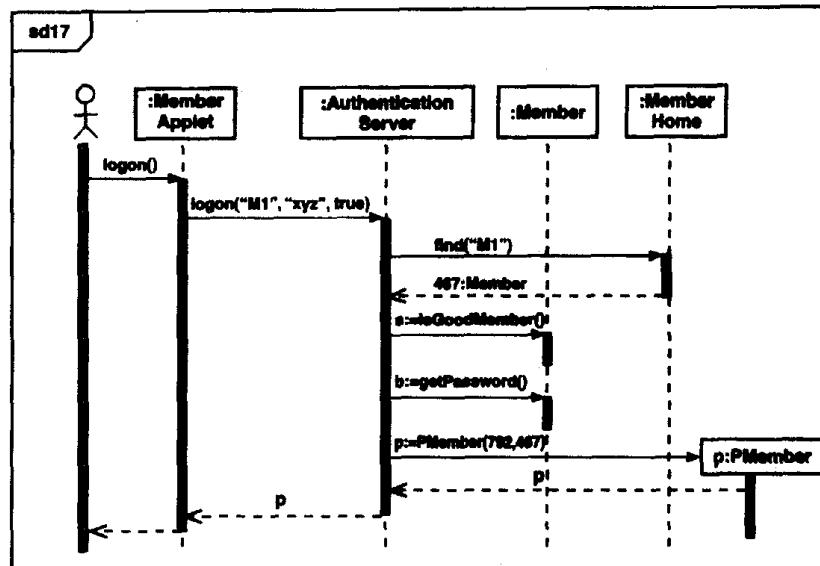


图 10-28 复习题 1

- (a) 状态机图      (b) 活动图      (c) 类图      (d) 用例图  
 (e) 顺序图      (f) 通信图      (g) 部署图
2. 当前，数据库管理系统最常见的类型是什么？(单选题)  
 (a) 网络      (b) 关系      (c) 面向对象      (d) 层次      (e) 索引文件
3. 在 UML 图中，类消息如何与实例消息区分？(单选题)  
 (a) 类消息显示为黑体      (b) 类消息显示为斜体  
 (c) 类消息有下划线      (d) 类消息带有关键字<<static>>
4. “死锁”的含义是什么？(单选题)  
 (a) 两个进程或线程拒绝相互通信。  
 (b) 对象的监视器允许其锁定早一点终止。  
 (c) 一个对象等待一个资源，该资源却正被另一个对象使用，而第二个对象正在等待第一个对象使用的资源。
5. 什么是线程？(单选题)  
 (a) 运行在一个节点上的独立进程，有它自己的内存和 IO。  
 (b) 进程中一个与其他活动共享内存的活动。  
 (c) 设计人员的思考过程。

## 10.12 复习题答案

1. 图 10-28 显示了 (e) 顺序图
2. 当前，数据库管理系统最常见的类型是 (b) 关系。
3. 在 UML 图中，类消息如何与实例消息区分？因为(c) 类消息有下划线。
4. “死锁”的含义是什么？(c) 一个对象等待一个资源，该资源却正被另一个对象使用，而第二个对象正在等待第一个对象使用的资源。
5. 什么是线程？(b) 进程中一个与其他活动共享内存的活动。

# 第 11 章 可重用的设计模式

在编写软件时，避免重复是很重要的。如果有人找出了问题的解决方案，就可以利用这个方案，而不是寻找另一个解决方案。一定要记住，面向对象的编程并不是编写代码，重用的代码越多，我们拥有的技巧就越多。

## 学习目标：

- 理解模式的含义，以及它对面向对象软件开发人员的含义
- 介绍一些常见的模式
- 介绍应用于软件的模式和使用它们的方式

## 11.1 引言

设计模式是开发人员避免重复的一种方式，模式可以把其他开发人员的知识和经验应用于某个特定的问题。模式还可以把我们的知识和经验与其他人交流。每个模式都描述了在现实世界中完成某个任务的行之有效的方式。

当把设计模式引入软件界时，它们引起了诸多方面的兴趣。结果，现在它们已应用于许多其他领域：

- 人机交互
- 并发
- 重用
- 教学对象技术
- 分布式计算
- 项目管理
- 网站
- 风险管理
- 反模式(事件出错)
- 问题解决
- 组织

### 11.1.1 模式简史

最初，模式与软件并无瓜葛。在 20 世纪 60 年代，一位建筑师 Christopher Alexander 开始在建筑和城市规划中编写模式。Alexander 研究了人类语言后，认为由可重用块组成的句子(即模式)可以用于记录知识，构建复杂的通信内容，解决问题。因为有了这些模式，Alexander 认为自然语言就是一种模式语言(pattern language)。

Alexander 相信，如果可以为建筑和城市规划开发出模式语言，建筑和城市规划就可以获得改进。其中一些改进得益于简洁地描述专业知识，而更多的改进仍得益于改进了的交流方式，以及允许公共成员参与具体的过程：为了确保成功，建筑物的最终居民应对其设计有一定的影响[Alexander 等 77]。

在 80 年代后期，面向对象编程在研究人员和从业人员中越来越普及，包括 Kent Beck、Ward Cunningham、Erich Gamma、Bruce Andenron 和 Richard Helm 在内的专家开始思考模式如何应用于软件。像有经验的设计人员(和 Smalltalk 程序员)一样，这些专家开始标识和记录协作对象的模式，随着时间的推移，这些模式显示出了强大的生命力。在 90 年代早期，这些专家的工作最终在面向对象编程、系统、语言和应用程序(OOPSLA)协会中集中讨论。

最终，这些工作构成了一本语义图书[Gamma 等 95]，它包含对模式的一般论述，以及可推广使用的 23 个模式，这本书的重要性表现在，十年后，其中的 23 个模式仍是最基本的。该书的作者常常称为“四人组”。

### 11.1.2 目前的软件模式

目前，因为有了最初的那本书，“设计模式”最常用于表示软件设计模式。如果上下文非常明显，开发人员就常常简称为“模式”。真正的专家仍使用术语“模式语言”来描述已有的模式和它们用于记录知识的方式，与他人交流，构建解决方案。

设计模式的最初发明者利用 Alexander 的想法，在设计过程中涉及到最终用户。但是，模式并不是以现在的方式来使用。这可能是因为软件设计，甚至是小型系统设计，都不易被非程序员所理解，或者是因为我们对软件还了解得不够，或生成的模式对外行来说不够简单(涉及最终用户还是比较理想的，但我们一般使用其他制品，例如用例、分析类图和 GUI 草图)。

前面说过，如果软件是一门科学，那么它就不是非常精确的科学。软件设计模式之间的差别比自然语言或体系结构中存在的模式之间的区别大得多。这是因为软件设计很难压缩为一组相互锁定的模式，或者只是因为我们还不知道该如何做。

模式肯定可以由软件开发人员用于记录知识，这样其他开发人员就不必重复这个工作了。标识和描述模式需要技巧和经验，甚至专家没有在几个不同的应用程序中证明新模式的有效性之前，也不应描述该模式。

除了记录知识之外，模式还可以用作新设计的预先做好的部分。模式独立于编程语言和应用程序域，所以它们使用起来不能简单地采用框架或库的使用方式。对于每个特定的情况，模式都需要细调。尽管如此，它们仍能节省大量的时间。

模式还可以用于编写解决方案的文档。例如，我们不是描述“发送到这个对象的消息通过网络传送到确实实现的、有类似接口的对象”，而是记录“这个对象是一个网络代理”。对于复杂的模式，模式的名称可能很长。

## 11.2 模式模板

模式应用得很广泛，所以需要以广泛接受的格式显示。理想情况下，其格式应易于读取和理解；但为了完整和正确，每个模式描述都要相当正规。[Gamma 等 95]就是这样：尽管每个模式都是基本的，进行了很好的描述，但结果该书变成了一本参考书，而不是教材。在本书中，

模式描述是不正规的，显示了其目标、结构和机制，还有代码示例和演示。

[Gamma 等 95]描述了可以由模式作者填充的模板。为了说明如何描述完整的模式，下面再次列出模板标题，并解释几个词语：

- 模式名称：模式的简短名称(通常是一两个词)，例如 Memento。显然，这个名称应表示模式的目标。它还应在应用程序域中是惟一的。
- 分类：每个模式都归类为创建(creational，考虑对象是如何创建的)、结构(structural，考虑把对象放在一个更大的结构中)或行为(behavioral，考虑对象之间的协作，以达到特定的目标)。
- 意图：总结模式用途的简短描述(一两个句子)，例如，“不违反封装原则，捕获和具体化对象的内部状态，以使对象可以在以后恢复到这个状态”。
- 也称为：模式的别名。
- 动机：描述使用模式解决的设计问题。
- 可应用性：这个模式可以应用的区域和如何识别这些区域。
- 结构：一个或多个类图和顺序图，它们演示了模式的工作原理。[Gamma 等 95]使用 OMT (UMT 的前驱)；显然，这里将使用 UML。
- 参与者：涉及对象及其工作(其责任)的简短描述。
- 协作：描述参与者之间的协作。
- 结果：描述的优缺点(以及对缺点的改进建议)。
- 实现：实现模式的建议，包括有用的技巧和应避免的问题。
- 样本代码：模式的完全实现。在[Gamma 等 95]中，实现代码用 C++ 或 Smalltalk 编写，本书使用 Java。
- 已知用法：模式在现实世界中的使用范围。
- 相关模式：类似于这个模式、但有区别的其他模式，以及使用这个模式时有价值的其他模式。

## 11.3 常见的设计模式

一般说，在实现除小程序之外的其他软件之前，必须熟悉核心模式。本节给出最常见模式的非正规描述，完整、正规的描述可在其他地方获得，但这里的内客足以打好基础。最好自己研究其他模式，至少是其他有名的模式(原型、桥、构建器、存储器、命令、修饰器、责任链、解释器、介体和访问器)。

### 11.3.1 观察器模式

定义对象之间的一对多依赖关系，当一个对象改变状态时，它的所有依赖对象都会自动获得通知。[Gamma 等 95]

在某个方面，一个对象的状态常常取决于另一个对象的状态，一个常见的例子是显示实体状态的 GUI 组件。例如，图 11-1 显示了一个工具，它预览顾客考虑购买的汽车。当顾客改变汽车的各个属性时，汽车的图片和显示的价格也会随之变化。GUI 对象本身是类 CarView 的一个实例，由三个面板组成，每个面板都显示汽车的一些信息。在其内部，GUI 由 Car 对象支持，

该对象为每个可能的选项都提供了一个字段(带有响应的读取器和设置器)。价格派生自其他属性，并可以使用 `getPrice` 获得。

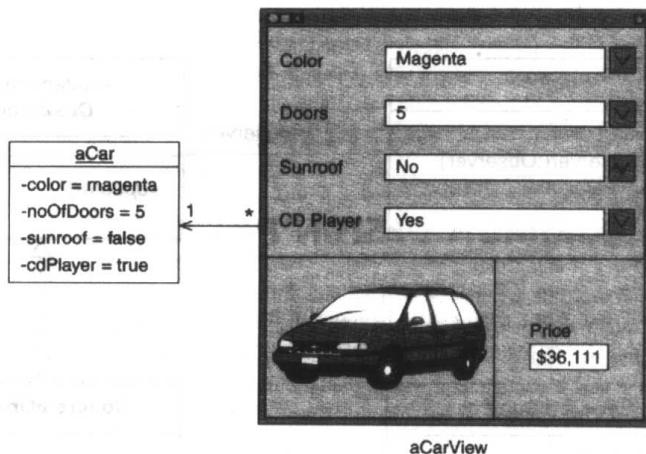


图 11-1 给汽车购买者提供的 GUI

应把逻辑放在什么地方，来检测汽车显示何时需要更新？如果放在 GUI 中，GUI 程序员就必须知道 Car 的操作方式——遮阳篷顶会影响价格，而颜色不会。这种方法会使 GUI 程序员的工作很难完成，还会使知识散布在系统中，很难管理。如果把知识放在实体中，如何确保 GUI 在适当的时候更新？可以让汽车实体对 GUI 有一定的了解，但我们又希望实体的开发独立于 GUI，使之可以在许多汽车应用程序中重用——把它组合到特定的 GUI 上会限制它的使用。

这样，就有了一个进退两难的情况：改变属性的知识应放在它所属的实体中，但是又不能把它放在实体中，因为这样实体就会组合到 GUI 上。用观察器(Observer)模式就可以解决这个困境。观察器的理念是给所有的 GUI 提供一个简单的接口，允许实体发出状态改变的信号，这意味着实体和 GUI 之间只有较轻的耦合(实体及其业务之间也有这种情形，任何对象都或多或少地需要观察它正在执行的工作)。

下面详细介绍观察器，先是抽象的定义，之后是汽车例子的实现。在图 11-2 中，改变了属性的对象称为主体(subject)，观察改变的对象称为观察器(observer)。Observer 类有一个简单的接口，它包含一个 `update` 消息，当属性发生改变时，这个消息由主体发送，为观察器提供了一个刷新的机会。主体有一组 `Observer` 对象，它们记录变化中的一个方面——使用 `addObserver` 消息进行记录(如果需要，观察器可以使用 `removeObserver` 消息在以后取消记录)。主体还有一个受保护的 `notify` 消息，它在观察器上迭代，给每个观察器发送 `update` 消息(为了便于预测，可以确保 `update` 消息以记录观察器的顺序发送)。

`Subject` 类尽管有完整的实现代码，但设计为一个超类——不修改一些属性，可重用的实现代码就不会使用得很频繁，所以把该类标记为抽象。要实现可观察的对象，就可以使其类继承自 `Subject`，如 `ConcreteSubject` 所示。`ConcreteSubject` 的实现代码必须确保，只要属性值有变化，就调用 `notify`，例如在 `setAttribute` 内部调用，使 `update` 消息发送到每个观察器上(为了避免进行不必要的通知，实现代码必须确保，只有新属性值不同于旧值时，才会调用 `notify`，在大多数情况下，我们处理的都是属性，不同就意味着不等于，而不是不相同)。

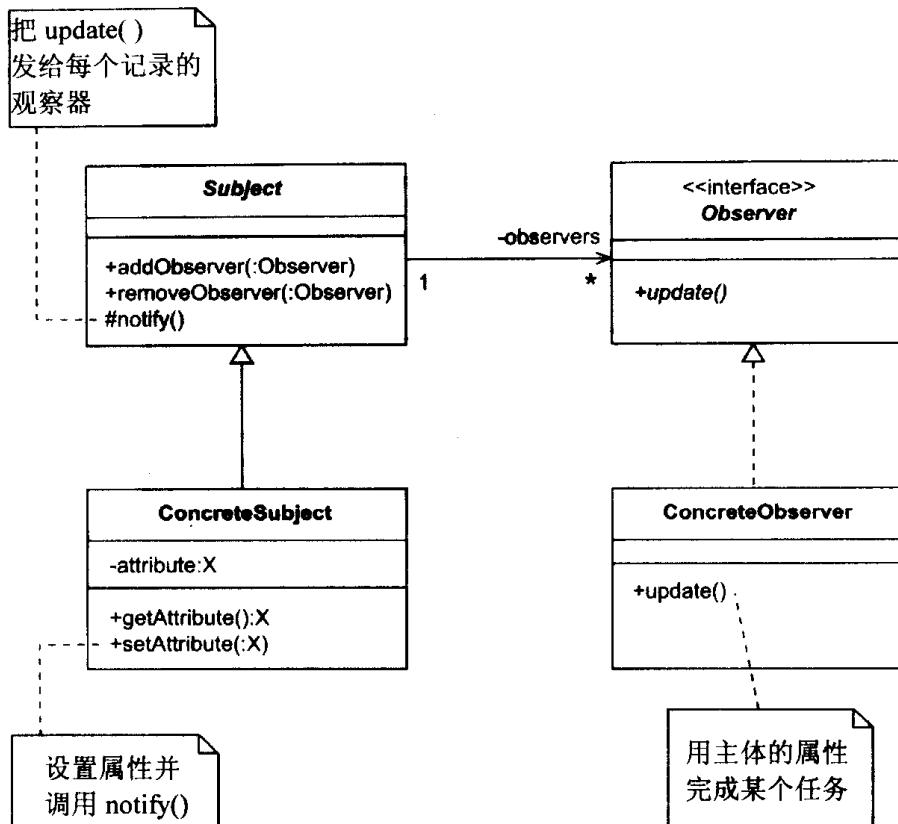


图 11-2 Observer 类图

Observer 类没有具体的方法(因为不知道 Observer 需要做什么才能刷新)。因此, Observer 可以是一个接口(纯抽象类)。在编写 Observer 类时, 它需要继承自 Observer, 实现 update 方法, 以某种方式刷新其内容, 如图 11-2 中的类 ConcreteObserver 所示。一般, ConcreteObserver 会使用 ConcreteSubject 读取器, 在执行 update 方法的过程中检索新属性值。

Observer 模式的消息顺序如图 11-3 所示。最初, anObject 用 aConcreteSubject 注册了两个观察器, 一段时间后, anObject 使用 setAttribute 修改了 aConcreteSubject 的状态。在内部记录了新值后, aConcreteSubject 把它自己发送给 notify 消息(在正式的 UML 样式中, 使用堆叠在一起的活动条, 表示 notify 方法的持续时间)。在 notify 方法内部, update 消息发送给 aConcreteObserver1, 然后发送给 aConcreteObserver2。在 update 方法中, 每个观察器使用 getAttribute 读取主体的当前状态。

ConcreteSubject 知道它在处理 Observer 对象, 而不是 ConcreteObserver 对象(每个 addObserver 和 removeObserver 的参数都是 Observer)。因此, ConcreteSubject 耦合到 Observer 类上, 而没有耦合到 ConcreteObserver 上; 这两个抽象类使主体独立于观察器。显然, 观察器与主体是紧密耦合的, 但因为观察器通常在上面的层中(在子系统设计阶段), 所以这没有问题。抽象类比较一般, 可以添加到类库中。

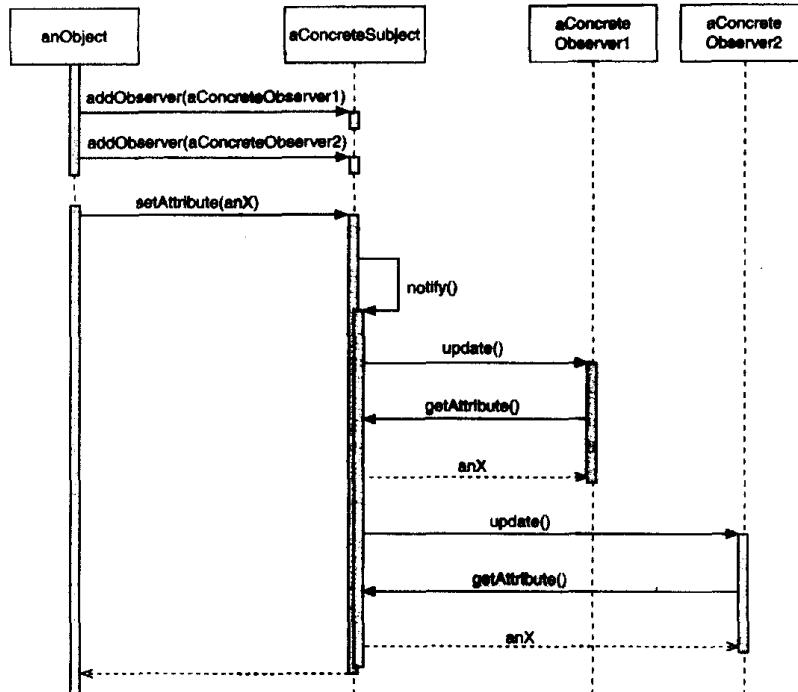


图 11-3 Observer 顺序图

把观察器应用于图 11-1 中的汽车显示工具上，就会得到如图 11-4 所示的类图。一个观察器 `aCarView` 用主体记录它自己，也用作更新源。消息流如图 11-5 所示。最初， `addObserver` 消息用于记录。接着在一段时间后，用户选择添加一个遮阳篷顶，这会把 `setSunroof(true)` 消息发送给 `aCar`。假定遮阳篷顶属性是 `false`，在 `notify` 中把 `update` 消息发送 `aCarView`。在 `aCarView` 的 `update` 方法内部，使用获取器读取当前的属性值。

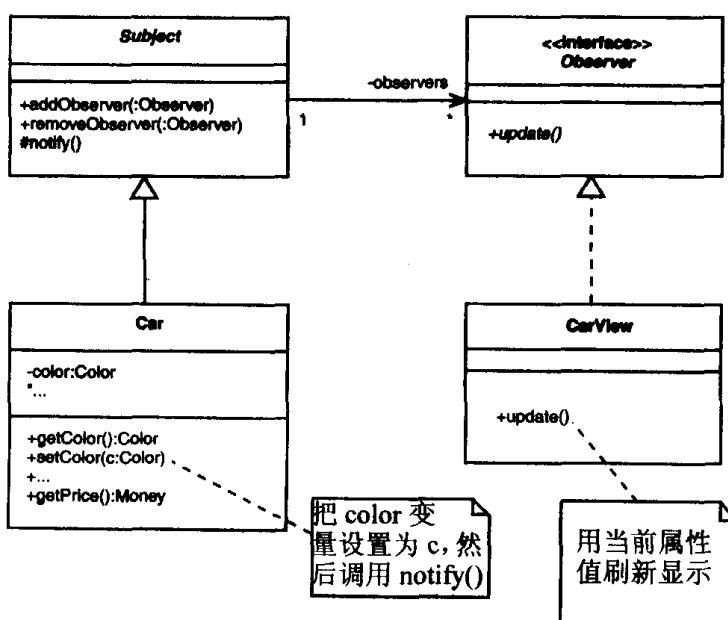


图 11-4 汽车观察器的类图

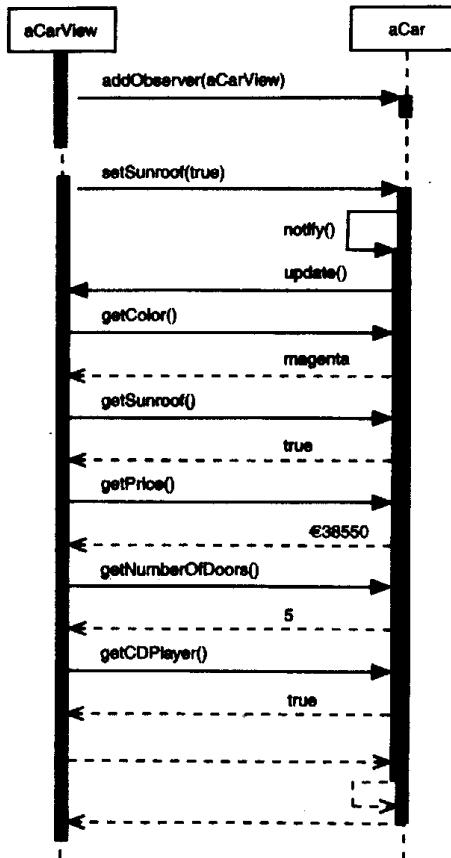


图 11-5 汽车观察器的顺序图

观察器模式常常用于让 GUI 在实体变化时更新它自己，该模式还可以用于其他地方，尤其是在多层系统中。例如，业务层中的实体由控制层上的控制器观察(控制器用作业务层和用户界面层之间的转换器)。控制器又由 GUI 组件观察(第 8 章介绍了在层之间发出信号的另一种方法，即使用事件，其效果与观察器相同)。

在多层系统中，有了 GUI(与基于 HTML 的界面相反)，观察器模式在客户端就非常有用，但是在服务器端，常常需要处理多个线程，额外的复杂性将使观察器模式难以利用。在理论上，也可以使用观察器模式从服务器向客户机发送更新信号，但是，在这种情况下，多个线程的问题会因多台机器和网络问题(例如超时和崩溃的客户机)而加剧。简言之，最好限制观察器模式在客户端的使用，只把它用作从一层向其上层传送通知的方式。在其他情况下，为了简单和安全起见，应避免使用观察器模式(服务器上的例外情况可以通过其他方式来处理，例如操作系统时钟的回调和机器对机器的消息传送，Java 消息传输服务就是一个例证)。

### 11.3.2 单一模式

确保类只有一个实例，并提供访问它的一个全局点。[Gamma 等 95]

有时，让对象是类的惟一实例是很有效的。这常常是因为该对象表示系统或应用程序域中的惟一组件，例如，在系统中存储数据库中的数据，数据库本身就可以表示为类 DB 的一个实例。另一个常见的例子是单一实例需要在系统的不同部分之间共享，以节省内存空间，减少创建时间。例如，使用一个 Calendar 回答“二月有多少天”等问题，而不是创建许多 Calendar 对象。

在设计模式中，对象是其类的惟一实例，该对象就称为单一对象(singleton)。为了让单一对象发挥作用，它应满足下述三个条件：

- 它必须很容易找到。
- 任何人都不能创建另一个对象。
- 直到需要时才创建它。

为了使单一对象容易找到，可以把它存储在类字段中，通过类消息来访问。这是很合理的，因为在面向对象的编程中，类长时间地用作共享数据和服务的入口点，为了确保只能创建一个单一对象，可以把用于创建单一对象的代码设置为私有，禁止普通代码访问它。

单一对象在需要时才创建不是那么重要，但仍是需要的，以避免创建和初始化从来不使用的对象。客户机程序员不会发出指令“现在创建对象”，所以我们可以选择在系统启动时创建对象，或者根据需要创建对象——一般情况下选择根据需要创建对象，使用一个简单的技术“懒惰初始化(lazy initialization)”，在这种技术中，条件逻辑用于在访问期间检查单一对象是否还未创建。

图 11-6 把单一模式实现为一个类图。其中，有一个私有类变量 instance，它指向 Singleton 对象，这个 Singleton 对象是根据需要由 getInstance 类方法创建的(UML 通过下划线区分类元素和对象元素)。其中还有一个构造函数，用<<create>>关键字表示。由于构造函数是私有的，不能在类的外部使用。最后，Singleton 类包含实例方法 instanceMethod1、instanceMethod2 等的定义(当然，和其他对象一样，Singleton 也可以有字段)。在图 11-7 中，有一个不太抽象的单一对象例子。新类的名称不同，但命名约定 getInstance 仍旧保持不变。

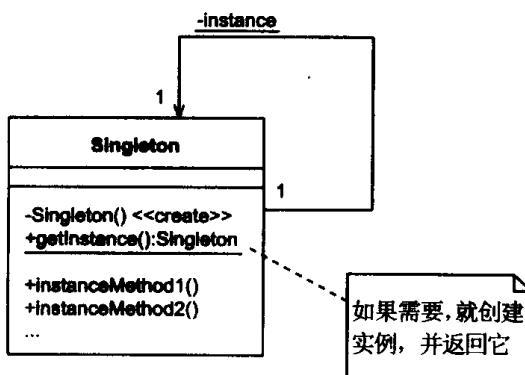


图 11-6 Singleton 类图

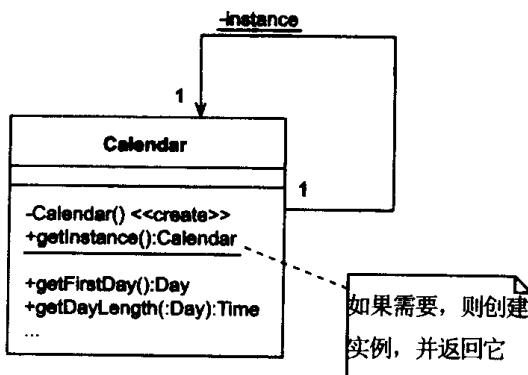


图 11-7 Calendar 类图

此时，图 11-6 所示的类图可能难以理解，因为它们在同一个方框中混合了实例和类概念。这是类图的一个缺点，但应该习惯它。为了阐明单一模式，可以看看要发生的事件(如图 11-8 所示)，这个图较容易理解。在这个图中，类本身是一个单独的对象，包含构造函数(Calendar)、类字段(instance)和类方法(getInstance)。而且，单一对象包含了实例字段和实例方法(一些编程语言如 Smalltalk 和 Java，的确把每个类建模为一个不同的对象)。在 UML 中，标记为 Calendar 的类是元类(metaclass)，而不是使用正式的 UML 关键字，命名规则也有点变化，使图更清楚。图 11-8 生成了如图 11-9 所示的对象图。其中，单一对象的创建和访问都由类管理，单一对象本身与其他对象一样，也有名称、类型和一些字段。

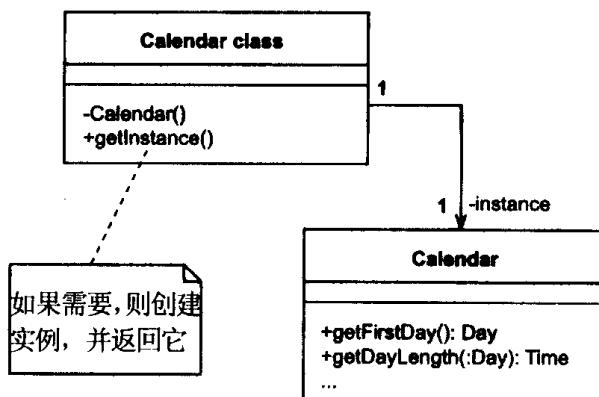


图 11-8 Calendar 是一个类对象



图 11-9 Calendar 对象图

在编程术语中，单一模式的实现是很容易的。程序员为了使用单一对象，必须先标识类，再调用GetInstance)。具体方法随语言的不同而不同。下面是 Java 的方式：

```

public class Calendar {

    private static Calendar instance; // static means "class field"

    private Calendar() { //Must declare a private constructor
        // Initialize any fields here
    }

    public static getInstance() {
        if (instance == null) {
            instance = new Calendar();
        }
        return instance;
    }

    //Declare any instance fields here...
}
  
```

```
//And now for the instance methods:  
public Day getFirstDay() { ... }  
public Time getDayLength(Day aDay) { ... }  
...  
}
```

下面的 Java 代码段显示，日历用于找出一年的第一天，并给用户显示结果：

```
Date d = Calendar.getInstance().getFirstDay();  
System.out.println("The first day of the year is"+d);
```

在第 2 章中，类字段和类消息都标识为管理集中式数据和服务的方式。从日历例子可以看出，单一模式还提供了对集中式数据和服务的访问。哪种比较好？一般说来，单一模式比较好，这有两个原因：

- 类元素不是非常面向对象的，因为大多数编程语言都不支持它们的继承和重定义。因此很难描述出不同种类的 Calendar 的层次，例如 Calendar、GregorianCalendar、JulianCalendar、IslamicCalendar 等。
- 设计人员和程序员必须在运行期间处理对象，为什么不应处理类(所以单一模式使用一个类字段和一个类方法，仅此而已)？

如果要使用类元素，就可以使用单一对象。

### 11.3.3 多重模式

毫无疑问，单一模式是很有用的，我们常常会遇到它。还有一种与它相关的模式，称为多重模式(multiton)，在[Gamma 等 95]中没有这种模式(但 [www.patterndigest.com](http://www.patterndigest.com) 上有它的一种形式)。术语“多重”是“多值单一对象”的双关语，但应该承认，这是一个矛盾修饰法。也可以认为多重模式是有一组限定值的任意类型(这类似于一些语言，如 C 语言或 UML 中的枚举，只是这种枚举中的每个值都是一个原型，而不是一个对象，所以它们不是很有用)。

例如，为汽车展示厅设计一个销售系统，其中的汽车有下述五种颜色：日落的红色、午夜的深蓝、早晨的橙色、中午的黄色和下午的灰色。在面向对象术语中，每种颜色都表示为一个独立的对象，这些对象都有自己的数据和行为(例如，某种颜色的汽车比最初的价格高出的金额)。这五个对象很容易访问。而且，为了避免错误，应禁止程序员创建自己的颜色。这个设计乍看起来很像单一模式，只是其中的“一”替代为“五”。

图 11-10 显示了类 CarColor 包含 CarColor 对象的懒惰初始化 Map，每个对象都可以用其名称来检索(Map 是一个集合，可以根据名称插入和检索对象)。getInstance 方法现在带一个 String 参数，可以指定五个实例中我们感兴趣的那一个。CarColor 还有 retrieveAllInstances 消息，可用于获取所有实例的列表——以便在 GUI 中显示可用的颜色。图 11-11 把 CarColor 及其内容分开显示，这样很容易看清将发生的情况：类有一个对 Map 的引用，该引用又有对五个 CarColor 对象的引用。由于构造函数是私有的，客户机程序员不能创建自己的 CarColor 对象。

如果某种语言没有为声明带一组限定值的类型提供特定的语法，多重模式就是很有用的。Java 在其语法中有多重模式的一种形式，所以不必编写自己的多重模式。

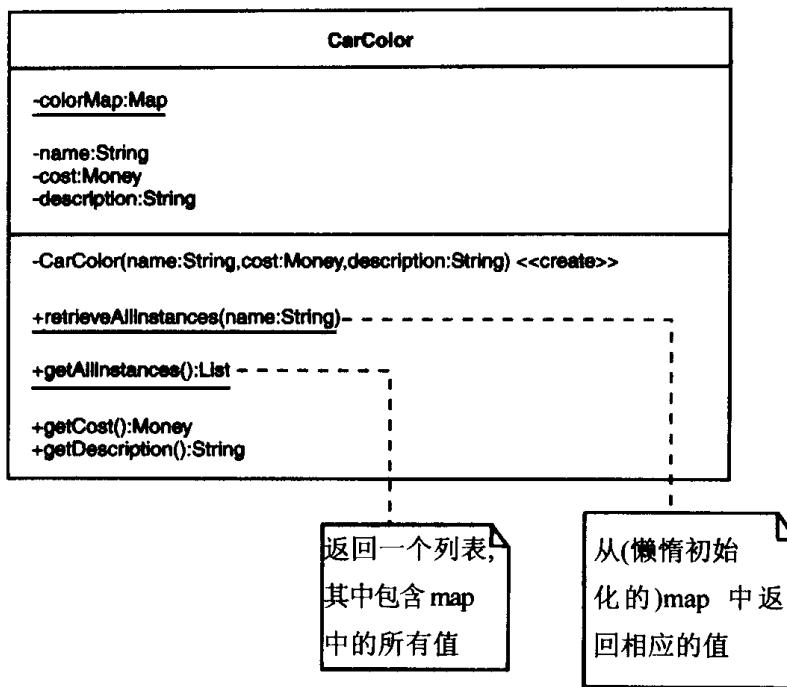


图 11-10 使用多重模式的汽车颜色

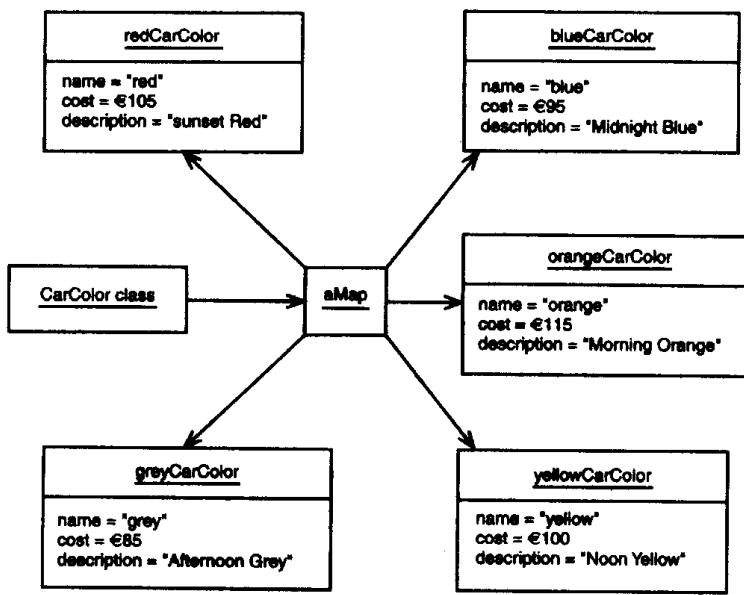


图 11-11 多重对象图

#### 11.3.4 迭代器模式

为顺序访问集合对象的元素提供一种方式，且不暴露其底层表示法。[Gamma 等 95]

我们常常需要为集合中的每个对象做一些事情。例如，如果仓库中有一个项集合，就要把所有项的值加在一起，计算出仓库的总存量值——换言之，需要对仓库中的每一项“把其值加到总数中”。在编程语言中，一般不直接支持这类任务，所以必须使用对象和消息。

迭代器是一种简单的模式，它允许使用标准的循环，从任意类型的集合中一次检索一项，

一旦在循环中有对对象的引用，就可以执行需要的操作。对于未排序的集合，我们不关心对象的检索顺序，只要每个对象都检索一次即可；对于有序的集合，对象应按顺序检索。迭代器需要集合类实现代码的协作，该实现代码至少提供一个创建和返回迭代器的方法。

图 11-12 显示了一个迭代器模式，用 Iterator 接口表示(因为在这一层次不希望提供可重用的实现代码，所以类可以是接口)。另外，Iterator 是集合的一个基本层次结构：顶层类 Collection 有一个抽象方法 createIterator，它创建和返回适当类型的迭代器，在 Collection 的下面有一个无序的变体 Bag 和一个有序的变体 List。

必须提供 Iterator 的多个具体实现代码，因为不可能编写出一个可用于各种集合的迭代器(但一些迭代器可用于多个集合)。假定每个具体的集合都有自己的迭代器，于是有了如图 11-12 所示的类 BagIterator 和 ListIterator。必须依赖集合类的实现代码的协作，这些实现代码必须找出或实现一个用于其集合的迭代器，并从 createIterator 中返回它。给定图 11-12 中的类模型，和适当的方法实现代码，客户机程序员就可以编写出统一的迭代代码，如下面的 Java 代码所示：

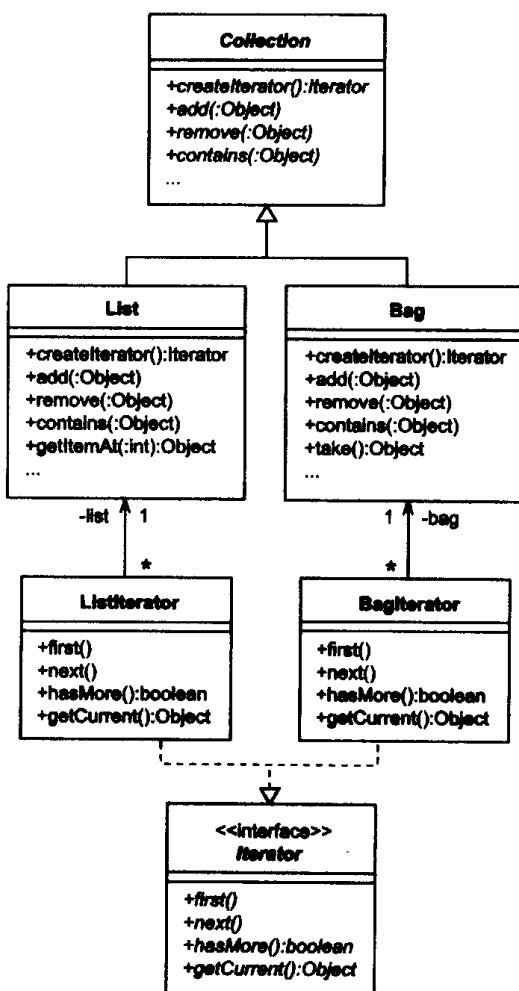


图 11-12 迭代器类图

```
Collection items = new Bag(); // "new List()" would work just as well
```

```
items.add(new widget());
items.add(new FDobrie());
```

```

Iterator i = items.createIterator(); // Iterator now at first element
while (i.hasMore()) {
    Object item = i.getCurrent();
    ...// Do something with item
    i.next();
}

```

显然，ListIterator 的实现代码也很简单。下面是其 Java 版本：

```

public class ListIterator implements Iterator {
    private List list;
    private int index; //Automatically set to 0

    protected ListIterator(List l) { //Constructor
        list = l;
    }
    public void first() {
        index = 0;
    }
    public void next() {
        index = index + 1;
    }
    public Boolean hasMore() {
        return index < list.getSize();
    }
    public Object getCurrent() {
        return list.getItemAt(index);
    }
}

```

为了完成 ListIterator，需要给 List 的实现代码添加类的下述方法：

```

public Iterator createIterator() {
    return new ListIterator(this); // "this" means "the current object"
}

```

为集合添加某种双向迭代器，以使其元素有序是很常见的，这样客户机就可以逆序迭代了。使用 Iterator 的一个子接口 TwoWayIterator 就可以逆序迭代，这会增加 last 和 previous 消息。接着，给定具体的实现代码，例如 ListTwoWayIterator，每个有序的集合就可以提供称为 createTwoWayIterator 的工厂方法(factory method)来补充 createIterator。

### 11.3.5 工厂方法和抽象工厂

为创建对象而定义接口，但让子类确定实例化哪个类。工厂方法允许类把实例化推迟到子类[Gamma 等 95]。

这里应提及工厂方法，因为它很重要。但不需要新的描述，因为前面在迭代器的实现代码中已经使用了这个模式(createIterator 和 createTwoWayIterator)。

简言之，工厂方法创建并返回某种类型的对象。工厂方法对客户机程序员来说很方便，因为：

- 客户机不需要知道要创建的对象的具体类型，而是使用较高级的抽象(在面向对象的编程中，这总是不错)。在前面的例子中，客户机程序员需要知道 Iterator 和 TwoWayIterator 接口，但不需要知道具体类型 ListIterator、BagIterator 和 ListTwoWayIterator。
- 类型可以时时改变，或从一个平台改到另一个平台上，而不会影响已有的客户机代码，因为客户机不知道要创建的对象的具体类型(另一个松耦合的例子)。
- 客户机不需要关心具体类的创建细节(例如构造函数的参数)。

工厂方法“允许类把实例化推迟到子类”，也就是允许 Collection 的实现代码利用还未定义的机制。换言之，Collection 可以提供抽象的 createIterator 方法，它在 List 和 Bag 中有不同的定义。

在相关的模式“抽象工厂(Abstract Factory)”中，类包含多个工厂方法——用于为整个对象系列提供一个创建点(“抽象”表示可以建立一个工厂的层次结构)。

### 11.3.6 状态模式

允许对象在其内部状态变化时改变其行为。对象看起来会改变其类[Gamma 等 95]。

如第 7 章所述，对象有时有复杂的生命周期，此时需要使用状态机来给其行为建模。在实现这类对象时，最好避免在对象自己的方法中编写对应状态机的所有细节，否则复杂性会弥漫到所有的代码中，使代码很难管理。状态(State)模式是把状态机的实现与原对象分隔开来的一种方便方式，可以方便地查看和修改与状态相关的行为。

考虑图 11-13 的状态机图，其中显示了典型自动售卖机的活动：自动售卖机可以销售许多罐饮料中的一罐，这些饮料的价格都相同。当启动该机器并准备好后，它就处于 Ready 状态，等待顾客送入一罐饮料的钱数。顾客走到机器前，放入一些钱，当钱数达到饮料的价格时(payment made 事件)，售卖按钮就会点亮——机器现在处于 Paid 状态。接着顾客按下一个按钮(choice made 事件)，机器就处于 Chosen 状态，此时它会检索饮料。最后，饮料掉到下面的槽中(drink dispensed 事件)，机器准备下一次销售操作。那么，如何使用状态模式，把这个状态机转换为对象？

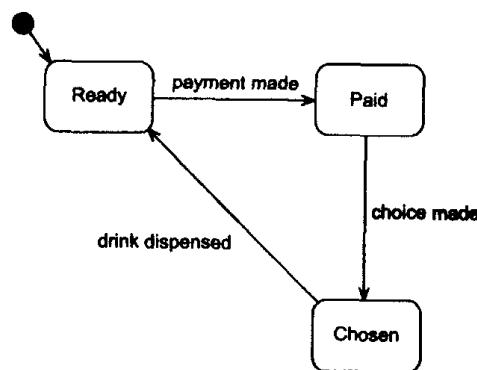


图 11-13 自动售卖机的状态机图

状态模式的一般形式如图 11-14 所示。其中，对象 Context 的生命周期比较有趣。在 Context 的状态机中有两个事件转换为 Context 上的事件 someEvent 和 anotherEvent。为了避免在 Context 内部编写这些事件的代码，添加了一个对象 ContextState：发送给 Context 的消息会传送给 ContextState。但 ContextState 是一个超类；其活动都添加到子类中，每个活动对应于状态机图

中的一个状态(这里是 StateA 和 StateB)。最好在 Context 处于状态 A 时, 其 state 变量是 StateA 的一个实例; 当它处于状态 B 时, 其 state 变量是 StateB 的一个实例。所有发送给 Context 的状态消息都委托给 state 对象, 所以程序员可以把与状态相关的活动放在 ContextState 的子类中, 更便于查找。

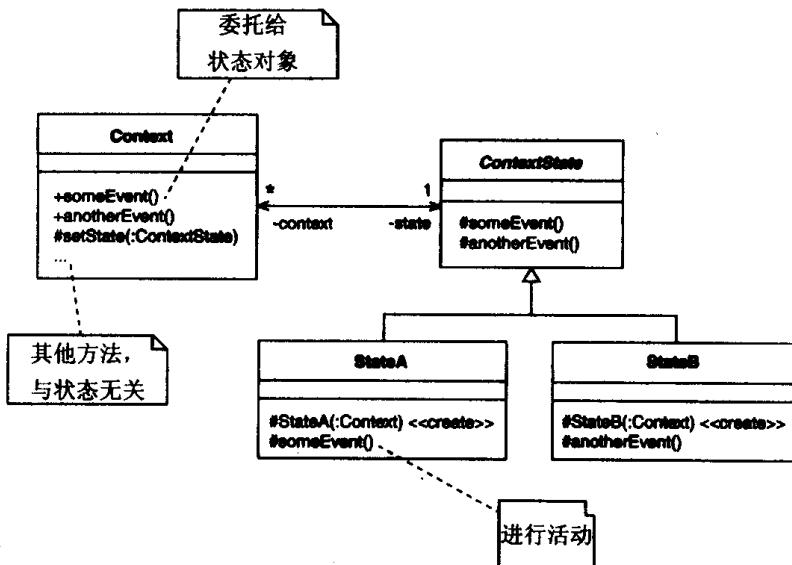


图 11-14 State 类图

在创建 Context 时, 它根据起始状态把 state 变量设置为 StateA 或 StateB 的一个实例。在这个初始化过程中, 还设置了 state 对象的 context 变量。例如, 对于用 Java 编写的 Context 类, 有:

```

public Context() { //Constructor
    new StateA(this);
}
  
```

对于 ContextState, 有:

```

public ContextState(Context c) { //constructor
    context = c;
    c.setState(this);
}
  
```

现在, 对于导致切换到另一个状态的事件, 事件方法的实现代码可以使用 Context 的 setState 消息, 使 context 处于新状态, 如下所示:

```

public void anEvent() {
    ... // State activity, followed by:
    new StateB(context);
}
  
```

上述模式意味着两个链接, 一个从 Context 链接到 ContextState, 另一个从 ContextState 链接到 Context, 但在状态对象内部仍保留着状态转换行为。状态类上的所有方法和构造函数都是受保护的, 因为它们不应由客户机访问。ContextState 类是抽象的, 因为只应创建其子类的实例。

下面看看图 11-15，其中显示了用状态模式实现的自动售卖机示例(为了简单起见，省略了构造函数和 setState)。要使用自动售卖机的人只需创建一个 VendingMachine 对象，在适当的时候给它发送 pay、choose 和 dispense 消息即可。VendingMachine 在内部先创建 Ready 的一个实例，当 pay 消息到达时，Ready 实例就会被 Paid 的一个实例替代；当 choose 消息到达时，Paid 实例就会被 Chosen 的一个实例替代；当 dispense 消息到达时，Chosen 实例就转换回 Ready。整个过程如图 11-16 所示，包括对象的创建。在具体类的每个状态方法中，都可以实现任意行为。这可以采用状态本身的操纵形式，或采用 VendingMachine 的操纵方式(因为有对 context 的引用，所以很容易给它发送消息)。

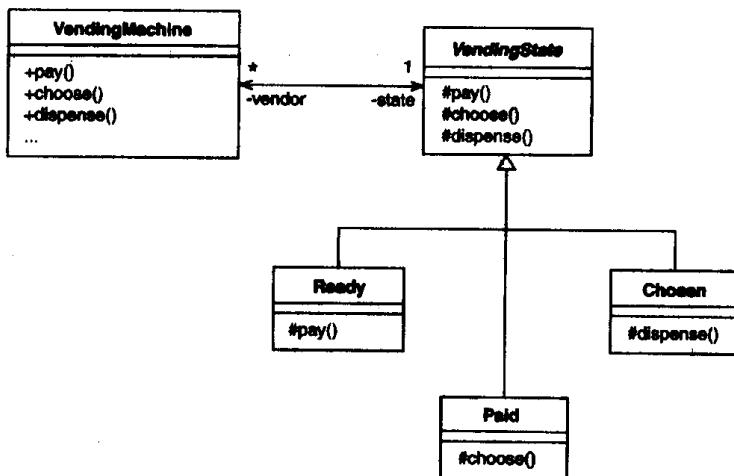


图 11-15 自动售卖机类图

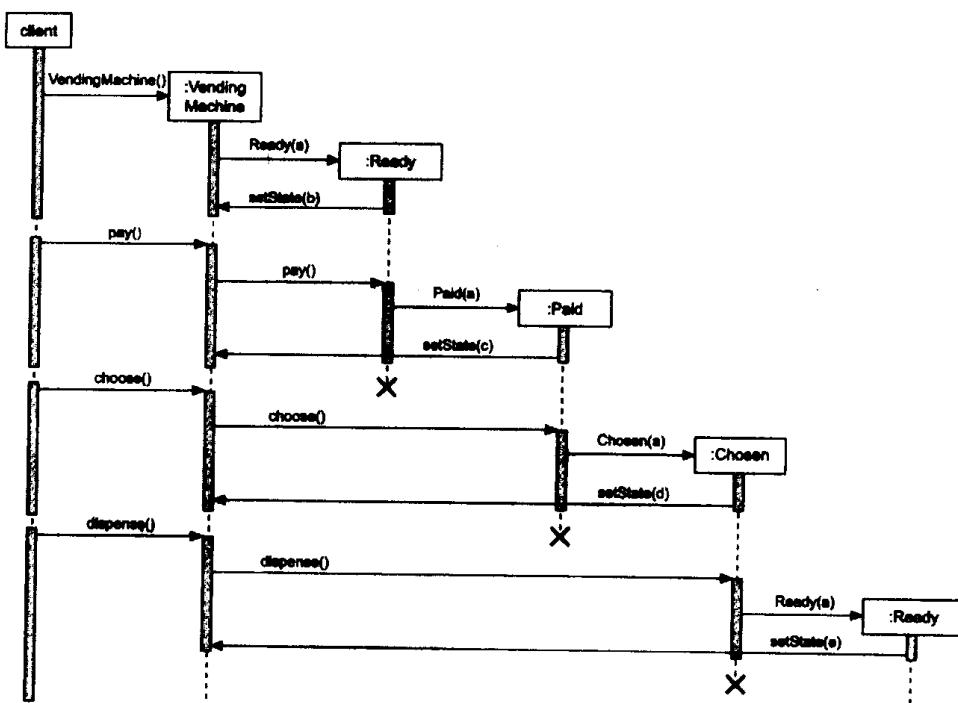


图 11-16 自动售卖机顺序图

您可能认为，类上的方法，如 `VendingState`，是抽象的。但是，如果把它们设置为抽象，具体子类的实现代码就必须重新定义与它们不相关的方法(因为每个事件都只能在特定的状态下发生)。所以在自动售卖机的例子中，即使 `choose` 和 `dispense` 事件从来也不会发生，但是 `Ready` 的编写者也必须提供 `choose` 和 `dispense` 的实现代码(因为饮料选择按钮在投入足够的钱数后才会激活，机器在顾客选择出饮料后才会提供饮料)。尽管可以给不相关的方法提供某种实现代码，并在调试时报告错误，但最终结果并不方便，显得有点笨拙。较好的方法是为 `VendingState` 上的每个方法提供可以报告错误的实现代码，这样，子类的编写人员就只需重载在该状态下可能发生的方法。

图 11-16 包含了一些新的 UML 表示法。生命线(lifeline)显示了对象在顺序图中可存活多久。可以在创建对象的地方显示它(例如 `Ready`)。如果对象由另一个对象创建，就可以用一个从创建者的活动条传送到新对象的消息来表示(可以标记上构造函数的细节)。也可以在对象生命线的末端加上一个大的黑色 X(可以带有传送进来的消息，如 `close`)，来表示对象生命周期的结束。在 UML 中，标记 `stop` 表示对象可以删除，或者在此点之后就不能使用该对象了。

状态机常常要解决的另一个问题是状态数据，例如与某个状态相关的属性。已付的钱数和已选择的饮料就是自动售卖机的状态数据。状态数据也很符合状态模式：只需给 `Context` 和 `ContextState` 类添加获取器和设置器，就像处理状态方法那样。接着就可以给实际使用数据的子类添加字段、具体的获取器和设置器了。例如，图 11-17 显示了给自动售卖机添加顾客选择的饮料后的效果(为了简单起见，省略了构造函数和状态设置方法)。这里，顾客做出的选择作为一个参数传送给 `choose`，再通过其构造函数传送给 `Chosen`。`Paid` 类中 `choose` 的 Java 实现代码如下所示：

```
public void choose(Drink d) {
    new Chosen(vendor,d);
}
```

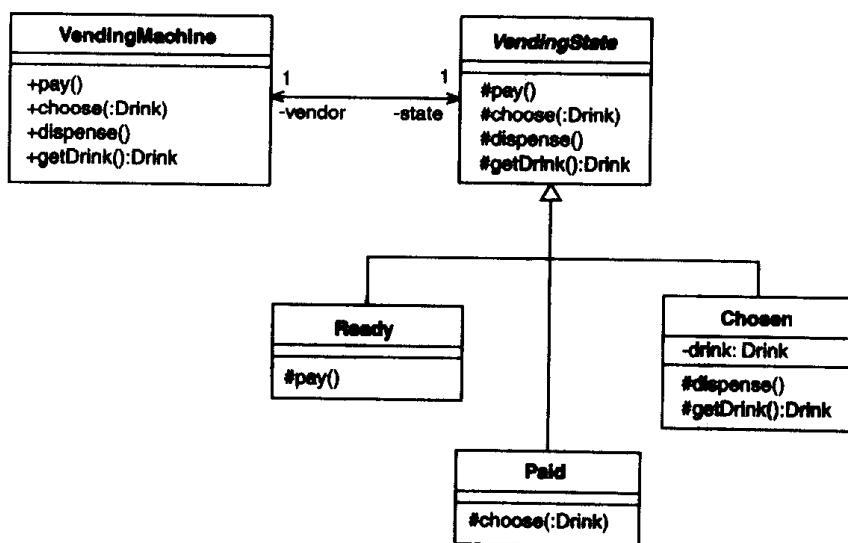


图 11-17 带有状态数据的售卖机

### 11.3.7 门面模式

为子系统中的一组接口提供统一的接口。门面模式(Facade)定义了使子系统易于使用的高级接口[Gamma 等 95]。

在使用对象实现子系统或层时，涉及的对象数可能相当大。例如，对于像 iCoot 这样的系统，业务层有数十个类，包括新类和从其他地方重用的类。这是合理的，因为我们要把规模较大的编程任务分解为可管理的、在必要时能相互协作的对象。另一个方法是使用较少的、较大的对象，这种大对象较难正确实现，许多对象都有多个用途(例如内聚力较差)。但是，生成大量有许多连接的复杂对象，层或子系统的客户就有一个问题：如何在不了解内部接口和协作的情况下，使用这么多对象来执行一个简单的任务？

在这里，门面模式就有用武之地了：每个门面都把子系统或层某一部分的复杂性转换为一个对象，该对象带有可用服务的一个子集。例如，在 iCoot 系统中，引入了六个服务器对象，把多线程、面向事务的业务对象转换为更简单的“请求一响应”协议，所有的联网客户机都可以使用该协议。在“请求一响应”协议中，当来自客户机(Web 浏览器或应用小程序)的简单请求到达时，相关的服务器对象就使用需要的业务对象和消息组合，生成一个简单的回应。这样，用户界面程序员的工作就非常简单了。门面模式还可以用于不同种类的接口，且使额外的编码最少。

图 11-18 显示了两个门面模式，为复杂子系统的客户提供有限的服务，在较复杂的方法中，客户直接使用子系统对象。

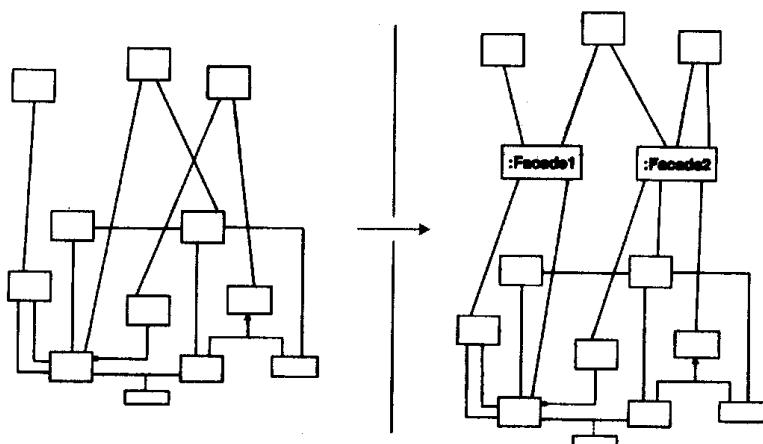


图 11-18 带有和不带门面模式的子系统

### 11.3.8 适配器模式

把类的一个接口转换为另一个客户机需要的接口。适配器允许接口不兼容的类一起工作。[Gamma 等 95]

适配器(adapter)模式在一个对象中封装了另一个对象，这样被封装的对象就可以在不同的环境下使用。我们常常需要这么做，以便连接最初不能一起使用的对象。适配器模式的基本形式如图 11-19 所示。其中，客户机和被适配者分别设计，但为了便于重用，它们应能一起工作。为此，应插入一个适配器，其接口正是客户机所需要的——在适配器的内部，把输入的消息传

送给被适配者上的实际消息，并根据需要转换消息名称、参数和返回类型。在每个适配器方法的内部，还可以提供被适配者缺乏的行为，或者修改已有的行为。



图 11-19 Adapter 对象图

在图 11-20 中，显示了适配器模式的一个简单应用。其中设计要求类 Queue 包含一队对象，允许在队尾添加对象，从队首删除对象，确定队列中有多少对象(getCount)。List 类已经实现了，它允许在尾部添加对象，删除第一个对象(使用 removeFirst)，确定有多少个对象(使用 getSize)——这是 Queue 需要的消息，只是名称不同而已。我们不想强制客户机使用 List 类，因为它的名称和消息是错误的，客户机可以给非队列式的行为使用 List 类(例如使用 removeElementAt 从队列的中间删除一个对象)。

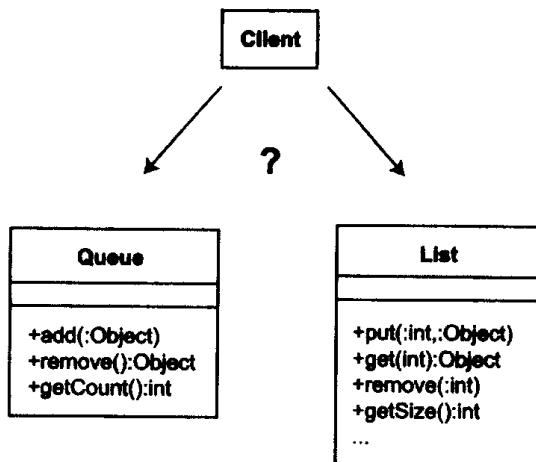


图 11-20 Queue 和 List

解决方案是把队列对象实现为一个适配器，以避免编写大量的代码，但仍给客户机提供了它们需要的内容(如图 11-21 所示)。完成后，Queue 会把它的三个消息转换为对应的 List 消息，但忽略其他不想要的 List 行为，如下面的 Java 实现代码所示。

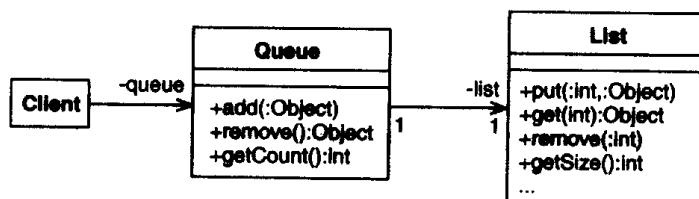


图 11-21 Queue 类图

```

public void add(Object o) {
    list.add(o);
}
  
```

```
public Object remove() {  
    return list.removeFirst();  
}  
public int getCount() {  
    return list.getSize();  
}
```

### 11.3.9 策略模式和模板方法

定义一系列算法，封装每个算法，使它们可以互换，策略允许算法随使用它的客户的不同而不同。[Gamma 等 95]

有时，一组任务仅在一些细节上有差异。例如，派生一辆汽车类似于派生一辆有篷货车或卡车，但每辆车都有自己的特性，例如刹车距离、全景可见性和传动器的高度。

下面举一个更简单的例子：使一组对象有序。把每个新元素插入到列表的正确位置，就可以确保列表的元素有序，这与处理索引卡类似。更具体地说，在插入新元素时，要向后扫描已有的列表，找到第一个比新元素大的元素，然后把新元素插到这个元素的前面(如果一直到列表的末尾都没有找到比新元素大的元素，就把新元素插入到列表的末尾)。每次的任务都是相同的，但细节不同：如果列表包含数字，“大于”就表示“与 0 的距离更远”；如果列表包含名称，“大于”就表示“与 A 的距离更远”。

在软件中，有一个问题：因为每次的细节都不同，所以就不能轻松地实现可用于各种元素的 SortedList 类。下面是两种解决方案：

- 使 add(:Object) 方法使用 if 语句确定要怎么做(如果 o 是一个数字，就这么做，否则就那么做)。这个解决方案的问题是，不同的逻辑会掩埋在方法内部，很难找到。更糟糕的是，客户不能使用这个类处理按信用级别排序的顾客列表、按价格排序的产品或其他情况。
- 提供一个抽象方法 biggerThan(:Object, :Object)，如果第一个参数大于第二个参数，它就返回 true(这个方法由 add 内部的一个通用算法调用)。接着，需要提供 SortedNumberList 子类，它以一种方式实现 biggerThan，再提供 SortedNameList 子类，它以另一种方式实现 biggerThan(这实际上是另一个模式，称为模板方法)。排序算法的内核很容易实现，但我们打乱了集合层次结构：如果 SortedList 需要 SortedLinkedList 子类和 SortedArrayList 子类，该怎么办？最终会得到 SortedNumberLinkedList、SortedNameLinkedList、SortedNumberArrayList、SortedNameArrayList——非常混乱(使用多重继承的解决方案也不易于使用)。

策略(strategy)模式就用于解决这个问题。该模式基本上提取细节(策略)，把它与主任任务(环境)完全分隔开。在上面的例子中，就是把 biggerThan 方法与 SortedList 类分隔开，如图 11-22 所示。其中，SortedList 与以前一样有一个 add 方法，但现在 biggerThan 在一个接口 Comparator 中。Comparator 表示一个对象，它知道如何比较类型相似的两个值：它有每种类型的对象的子类(NumberComparator 和 NameComparator)。每个 SortedList 都必须有一个可用于添加对象的 Comparator。所以，给 SortedList 提供 comparator 属性，由构造函数设置它。现在，如果要建立数字的有序列表，就可以执行下面的代码(参见图 11-23 中的消息序列)：

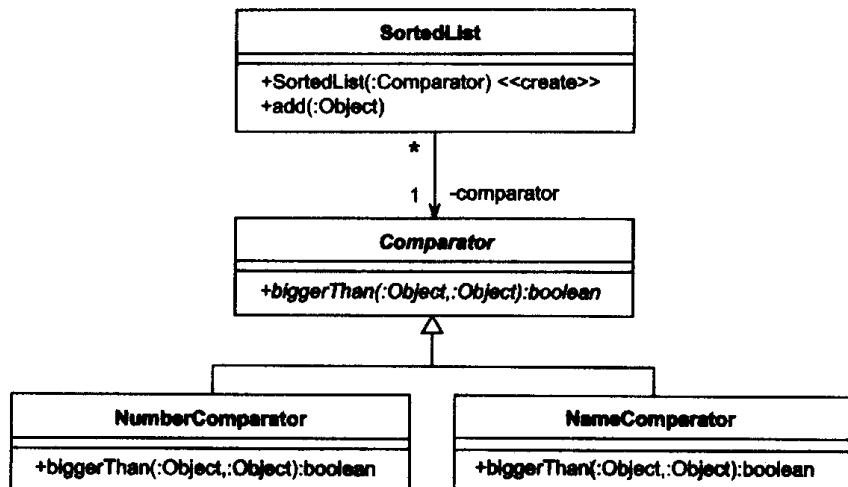


图 11-22 使用策略模式的 SortedList 的类图

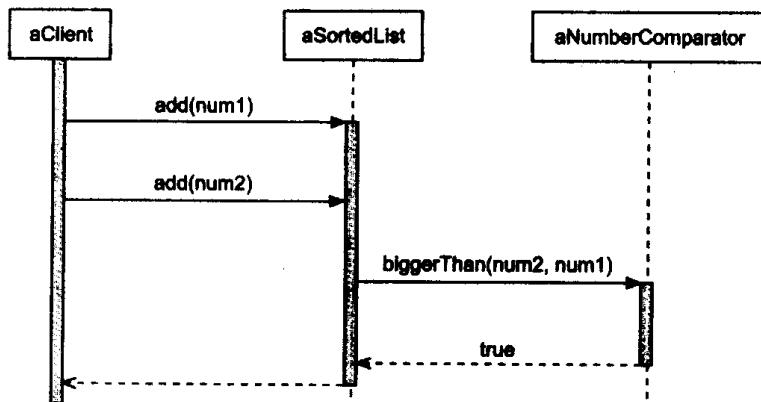


图 11-23 SortedList 顺序图

```

Comparator c = new NumberComparator();
SortedList l = new SortedList(c);
l.add(aNumber);
l.add(anotherNumber);
  
```

如果要建立名称的有序列表，就可以执行下面的代码：

```

Comparator c = new NameComparator();
SortedList l = new SortedList(c);
l.add(aName);
l.add(anotherName);
  
```

策略模式的优点是，它允许客户机程序员实现自己的策略，而不会干扰 SortedList 类。例如，如果要建立 WeddingGift 对象的有序列表，应实现自己的 WeddingGiftComparator，其 biggerThan 方法用于比较价格。就好像这是一个不完整的拼图，任何人都可以加入最后一块拼图块。

当然，SortedList 例子中描述的插入排序(insertion sort)非常慢。跳到列表的中间开始排序会快得多：如果还没有到要插入的位置，就跳到后面的一半；如果已经越过了要插入的位置，

就跳到前面的一半；重复这个过程，直到找到正确的位置为止。这个算法称为二叉树搜索(binary chop search)，前面的讨论没有提及它，因为它会妨碍我们的讨论。

### 11.3.10 次轻量级模式

使用共享来有效地支持大量细化的对象。[Gamma 等 95]

次轻量级(flyweight)是一个简单的模式，允许客户机程序员考虑在对象由多个客户机使用时，用构造方法来创建自己的对象。正常情况下，这会避免创建许多相同的对象，节省内存，提高性能。

例如，如果编写一个应用程序，来处理人的姓氏，就会发现许多人的姓氏都相同。此时不需要创建姓氏 Sam 的许多实例，而可以只创建一个对象，给它提供许多引用。图 11-24 给出了两个方法。我们需要已经创建的姓氏的缓存或池。所以，不是在每次需要时都创建一个姓氏，而是在池中查找该姓氏是否已存在。如果存在，就使用它，否则就创建一个新的姓氏，把它添加到池中，然后使用它(如图 11-25 所示)。

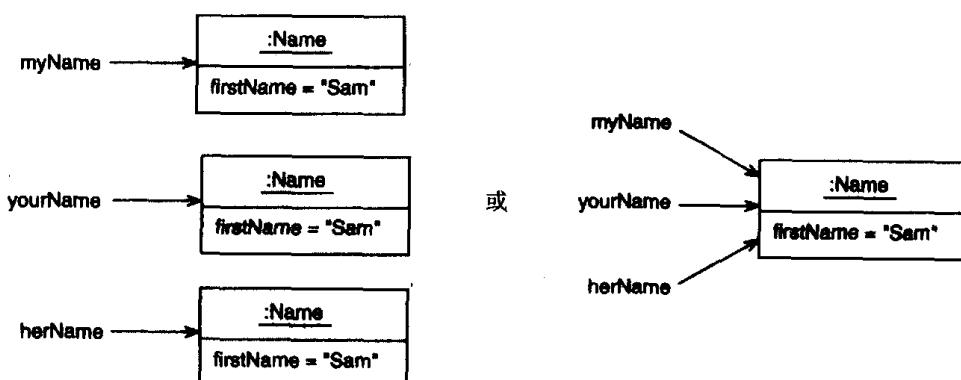


图 11-24 共享或不共享

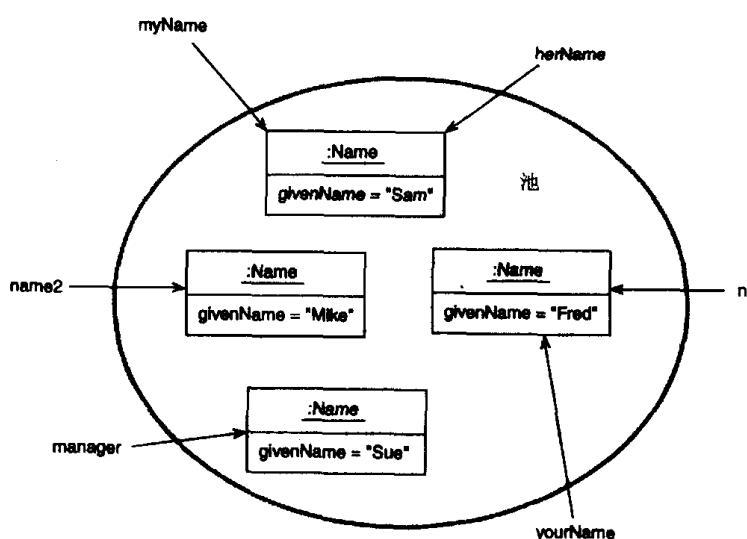


图 11-25 共享池中的对象

理想情况下，要对客户机程序员隐藏搜索池和创建新对象的复杂性：此时工厂方法非常合适。图 11-26 显示了一个一般的次轻量级模式和专用于姓氏例子的次轻量级模式。

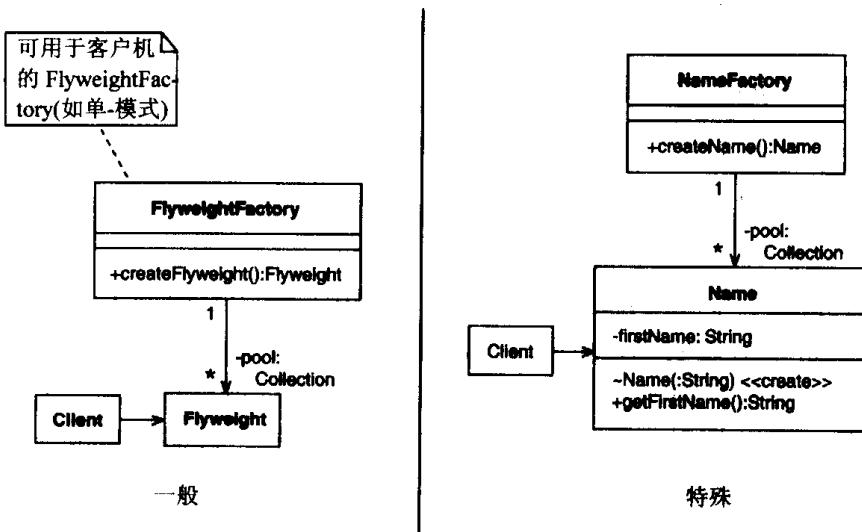


图 11-26 Flyweight 类图

次轻量级模式不一定是 100% 共享的：如果让客户机在外部存储一些状态，再把它作为一个参数传送给每个次轻量级消息，就有了一个混合体。举一个简单的例子，假定要在姓氏中添加名字。如果把名字添加到次轻量级对象中，就会丧失大多数共享机会(Sam J. 比 Sam 少见得多)。但是，如果让客户机自己管理名字，只要名字作为每个消息的参数传送给次轻量级对象，次轻量级对象就仍有它需要的所有信息。例如，在下面的 Java 代码段中，如果姓氏或名字包含字母 j，就执行某个动作：

```

public class Name {
    ...
    public boolean containsLetter(char letter, char secondInitial) {
        String namePlusSecondInitial = givenName + secondInitial;
        return namePlusSecondInitial.containsIgnoringCase(letter);
    }
}

Name n = aNameFactory.createName("sam");
char secondInitial = 'J';
if (n.containsLetter('j', secondInitial)) {...
}

```

次轻量级对象内部的状态称为内在(*intrinsic*)，由客户机在外部管理的状态称为外在(*extrinsic*)。当次轻量级对象的状态是不可变(只读)时，次轻量级模式特别合适，也最容易实现，否则就会有更外在的状态，共享性也较差。例如，把姓氏 Sam 中的第二个字符改为 i，是否会影响姓氏的其他用法？

### 11.3.11 复合模式

把对象复合到树结构中，表示部分-整体层次结构。复合模式允许客户机统一地处理各个

## 对象和对象的复合。[Gamma 等 95]

部分-整体层次结构是复合或聚合的另一个名字。复合(composite)模式并没有描述严格 UML 意义上的复合，而是描述了可用于实现聚合或复合的一种设计。因此，复合模式允许建立对象的层次结构，且可以有任意多个层次。另外，复合模式允许以相同的方式处理层次结构上的所有层次——即删除一个部分就像删除一个对象那样简单。

图 11-27 显示了复合模式的一个常见形式。其中有一个抽象类 Component，表示层次结构的一部分。在它的下面是类 Basic，表示最小的组件(不能包含其他组件的组件)。在 Basic 的旁边是 Composite 类，它表示可以包含其他组件的组件(所以链接回 Component)。由于 Composite 可以包含任意类型的 Component(Composite 或 Basic)，所以可以建立任意多个层次，而不是两个层次。

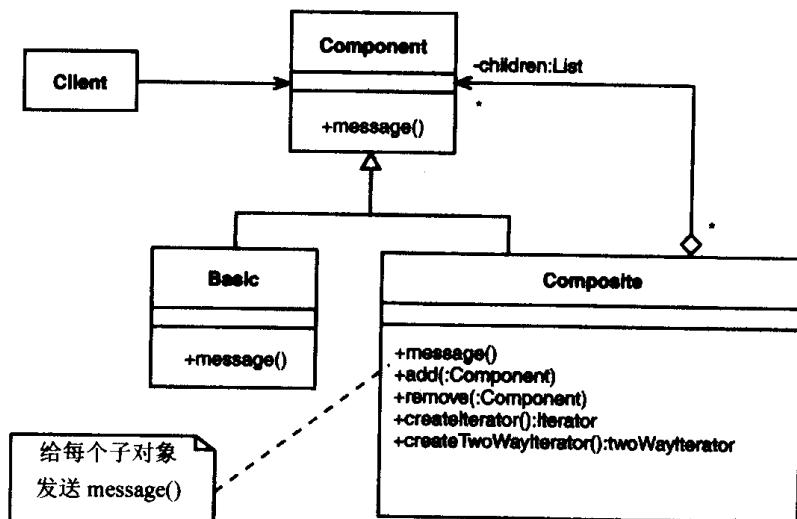


图 11-27 Composite 类图

多重性为\*的聚合初看起来有点古怪，但这有三个原因：第一，Basic 组件不是必须在 Composite 中(但常常在 Composite 中)；第二，该模式并没有禁止把一个 Component 一次放在多个 Composite 中，但这种情况非常少见(如果出现这种情况，关系就可能是 UML 复合)；第三，模式没有指定 Component 应在其 Composite 死亡时也死亡(但常常是这样)——同时共享的死亡是 UML 复合的另一个要求。

Composite 有一个链接返回其上面的层次，这常常称为递归(与递归复合相同)。递归在计算机术语中表示，一次次地重来。最终结果有点像俄罗斯玩偶：大玩偶里有一个小玩偶，小玩偶里有一个更小的玩偶，依次类推，最后是一个最小的玩偶。但与俄罗斯玩偶不同的是，每个 Composite 都可以包含任意多个组件——这就允许建立更大的层次结构，每个父对象都有任意多个子对象(一个房子有三层，每一层有五个房间，每个房间有四面墙)。

图 11-28 显示了一般 Composite 模型描述的两个层次结构。下面是使用 Java 建立层次结构 A 的过程：

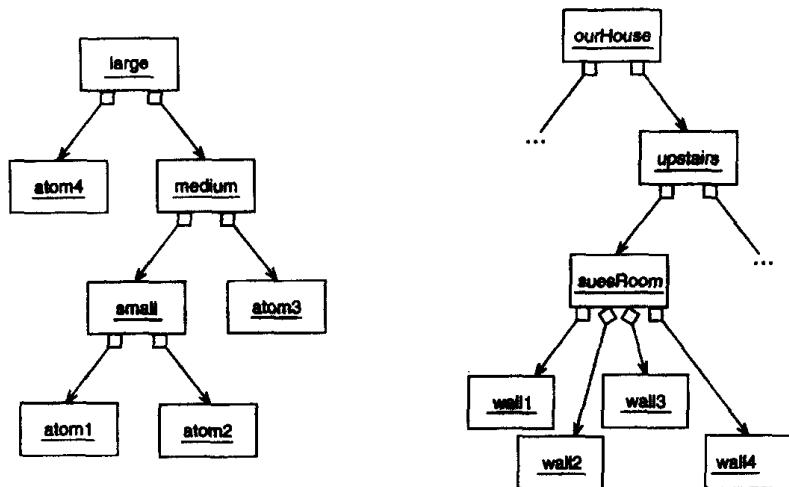


图 11-28 Composite 对象图

```

Component small = new Component(), medium = new Component(),
large = new Component();

Basic atom1 = new Atom(), atom2 = new Atom(),
atom3 = new Atom(), atom4 = new Atom();

small.add(atom1);
small.add(atom2);

medium.add(small);
medium.add(atom3);

large.add(atom4);
large.add(medium);

```

Composite 例子中的子对象是按从左到右的顺序排列的，一般最好保持添加组件的顺序，因为这可能对客户机程序员非常重要。提供 `createIterator` 方法，返回迭代器模式的一个实例。为了完整起见，提供 `createTwoWayIterator` 方法，允许客户机向后迭代。

除了处理层次结构之外，复合模式还允许客户机统一地处理各个对象和对象的复合。在设计中，这表示 `Component` 层次也出现在 `Basic` 和 `Composite` 层次上，如图 11-27 中的 `message` 所示。Composite 的递归性质的一个好处是，可以利用 Composite 上的 `message` 把消息传送给它的子对象。为了说明其工作方式，考虑图 11-28B 中的房子。如果喜欢粉红色，就可以把房子中的每面墙都漆成粉红色。为此，应把消息 `paint("Pink")` 传送给 `ourHouse`，`ourHouse` 会把 `paint("Pink")` 传送给每一层，包括 `upstairs`。每一楼层都会把 `paint("Pink")` 传送给每个房间，最后各个房间会把 `paint("Pink")` 传送给每面墙。由于墙是基本对象，它们会做实际的工作。最终，由客户机程序员发送的一个消息会把该房子的每一面墙都漆成粉红色(读者可以试着在顺序图中绘制这些消息)。

此时读者可能会问，集合和复合有什么区别？尤其是迭代器在讨论中突然出现。集合和复合都包含对象，但集合类的大多数客户都有四个基本类(可能是已排序的版本)：

- Bag：无序的对象，有重复(在 Java 中不可用)
- Set：无序的对象，没有重复

- List: 特定顺序的对象

- Map: 查找表, 需要一个键来添加或删除对象

这些类的共同特性是客户机程序员不必建立结构; 只需添加和删除对象即可(对于 List, 程序员必须指定每个对象的位置, 但不必创建对象占位符或依次添加或删除对象)。相反, Composite 类要求客户机程序员必须逐段地建立结构, 明确地把小段合并为大段, 生成某种类似树的结构。实际上, 如果客户机想要或必须显式处理结构(因为它无法事先确定), 或者客户机希望顶级消息方便地传送到所有的子对象, 就应使用复合模式(您能想出一种方式, 使用策略模式让集合类把顶级消息传送到所有的子对象吗)。

复合模式有时用作集合类的内部实现方式的一部分, 来建立索引等。另一个可能遇到复合模式的领域是用户界面, 因为 GUI 通常处理为组件的层次结构。

### 11.3.12 代理模式

为另一个对象提供代理或占位符, 以控制对它的访问。[Gamma 等 95]

代理(proxy)模式涉及一个对象(代理)把它自己插入到客户机和另一个对象(真正的主体)之间, 如图 11-29 的下半部分所示。这么做有无数原因, 例如安全控制、懒惰初始化或远程访问。在上面的例子中, 客户机给代理发送消息, 代理把消息传送给真正的主体(这可能涉及检查安全许可、第一次创建真正的主体, 进行网络通信)。尽管不是肯定需要, 但最好引入一个接口, 例如 Subject, 列出 RealSubject 和 Proxy 上的消息。其一, 这可以确保代理和真正主体的接口步调一致; 其二, 从建模的角度来看这是合理的, 因为代理和真正主体在做相同的事。

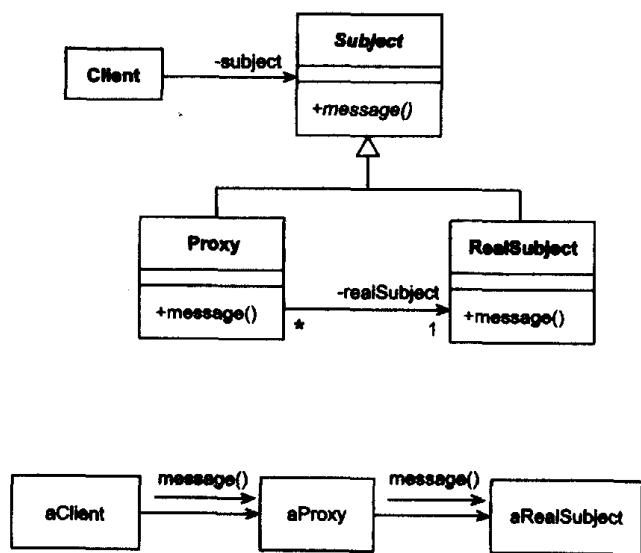


图 11-29 Proxy 类和对象图

代理模式最常见的用法是在网络编程领域, 涉及用一段代码调用另一段驻留在另一台机器上的代码, 两台机器之间有某种网络连接。在对象技术领域, 这个机制已成为框架的一个部分, 如 CORBA、J2EE 和 .Net。

在面向对象的编程中, 运行一段代码的惟一方式是给对象发送一个消息。但如何给位于另一台机器上的对象发送消息? 答案是提供一个本地代理, 它的接口与远程对象的接口相同, 但

其方法执行网络通信。这样，就对客户机隐藏了网络通信的复杂性。

使用代理发送远程消息如图 11-30 所示(它合并了部署图的元素和通信图的元素)。其中，有一个远程系统按照薪水册上的号码查找员工的电子邮件地址。服务器有一个过程包含一个 ContactImpl，称为 s，它等待传入的客户请求。客户机有一个过程包含一个 ContactClient(称为 c)和一个 ContactsProxy(称为 p)，它与 s 有相同的接口。当 c 需要电子邮件地址时，就给 p 发送 lookup 消息，p 完成必要的操作(创建通信信道、识别消息的类型、传送参数、等待响应)，把消息传送给 s。在服务器端，一些补充代码(未显示)接收传入的请求，把它作为正常的消息传送给 s。s 接收到消息后，就执行其 lookup 方法，就好像消息在本地生成一样。s 完成后，就把回应返回给 p。p 接收到回应后，就完成其 lookup 方法，把结果返回给 c。

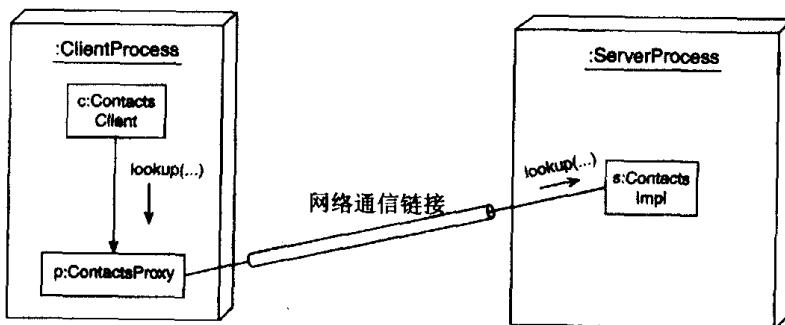


图 11-30 使用代理实现的远程对象

这个代理例子涉及到的类和接口如图 11-31 所示。在这个图中，与大多数设计级类图不同，ContactsClient 和 ContactImpl 之间的关联是不能导航的。这是因为对于远程对象，代理没有指向真正主体的物理指针：它只有对通信信道的引用，或对主体的地址的引用(不希望使通信信道一直处于打开状态)。

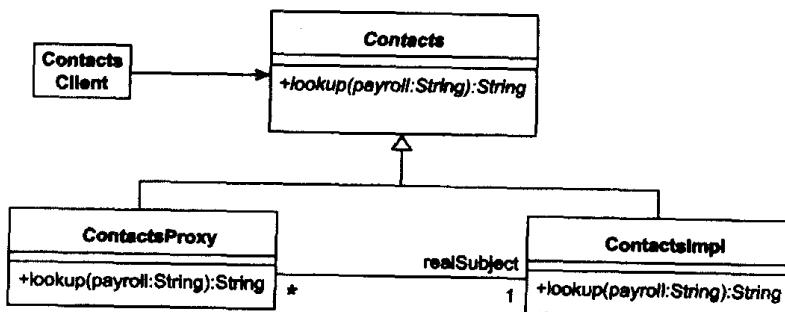


图 11-31 远程对象的类图

## 11.4 使用模式

模式使用得越多，就越会发现它们是某些任务的第二性质，尤其是理解得很透彻的任务，例如创建对象(工厂)、远程对象(代理)和复杂的生命周期(状态)。对于其他任务，必须找出它们有效的区域。也可以使用几个模式完成一个任务(在把工厂方法用作迭代器的一部分时，就会看到这方面的一个例子)。

## 11.5 发现、合并和调整模式

对于如何发现和合并模式并没有确定的规则，但最好先实践本章介绍的模式。接着，找出在已有的代码、库和框架、或同事编写的代码中使用模式的方式。再回过头来看看[Gamma 等 95]和本章提及的其他资源，以扩展自己的知识。这些资源有许多可吸收利用的部分，最终应达到更高的生产率、得到质量更高的代码。

我们会很快发现，每次使用模式，都需要对模式进行调整。下面是调整模式的一些原因：

- 语言的翻译：刚开始使用的模式可能在自己选择的编程语言中没有实现。所有的编程语言，甚至是面向对象的语言，都是有差异的。一些差异比较大(例如纯语言或混合语言的差异)，一些差异比较小(例如消息的访问保护)，必须调整标准模式，以适应各种语言。
- 观念的差异：对于每个设计问题，都有多个解决方案。因为模式是由一个专家或一组专家生成的，这并不意味着模式是完美的。一旦完全理解了模式，就可以修改它，尤其读者也变成专家之后，就更可能会修改它。
- 样式的差异：设计和编码样式可以来自许多源：经验、公司编码规则、库或框架的要求等。如果模式不符合项目使用的样式，就应调整它(和周围的人用相同的方式完成工作，往往比采用更好的方式完成工作更重要——更便于重用，易于维护)。
- 不同的问题：每个模式都尽可能通用——本章介绍的一些类模型从来不会不加任何修改就使用。例如，在观察器模式中，有 `Observer`、`Subject`、`ConcreteObserver` 和 `ConcreteSubject` 类，但 `ConcreteObserver` 和 `ConcreteSubject` 类不会有实际的应用。没有任何一种模式能完全适合某个具体的问题。
- 复合、继承和多重继承：语言和开发人员在使用继承和复合的方式上是不同的。我们可能把一种表示方式转换为另一种表示方式，并试图使模式的整体效果保持不变。
- 结果：每个好的模式描述都包含了结果、作者指定使用模式有特定优点、缺点和折衷的区域(毕竟任何一个模式都不是完美的)。因此，要确定可以接受什么样的折衷结果。
- 清楚：有时，带有所有功能的、完全成熟的模式过于复杂，简化一下是有好处的(尤其是在学习阶段)。

所有这些要素在本章的模式描述中都有某种程度的阐述。至少语言翻译是必要的，因为这里使用的表示法是 UML 表示法，编程语言是 Java，而不是[Gamma 等 95]所使用的 OMT 和 C++/Smalltalk。

毫无疑问，观察器模式是一个好模式，但在它最简单的形式中有一些缺点，图 11-32 显示了对观察器模式的调整，以克服这些缺点：

- 只要主体的状态有变化，就调用 `update`。观察器可以读取没有改变的属性，并刷新它自己。

在调整后的模式中，客户机在注册阶段使用新参数 `a` 来指定 `Observer` 感兴趣的属性(当然，`Observer` 可以为多个属性注册)。`notify` 方法也带一个参数：它只通知已注册的 `Observer` 对象，`a` 发生了变化。观察器模式可以观察一些属性，忽略其他属性。

- 观察器模式必须给主体返回已发生的变化，这涉及较多的通信。

在调整后的模式中，`update` 方法把属性的新值作为一个参数。因此，`Observer` 不必返回

Subject，来使用获取器。

- 观察器只能观察一个主体。

在调整后的模式中，Subject 作为一个参数传送给 update。这就允许 Observer 用任意多个主体注册，且仍能识别它们。

在使用这个 Observer 的修改形式时，主体必须通知观察器派生的属性何时发生变化，而不仅仅是存储的属性何时发生变化。

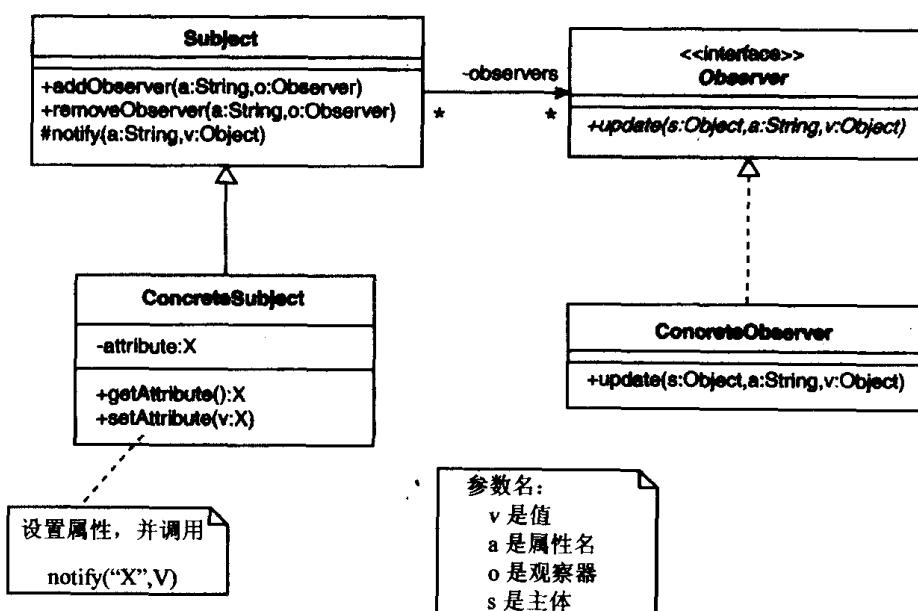


图 11-32 改进后观察器模式的类图

## 案例分析

### iCoot 中的模式

下面看看上述模式对 iCoot 系统的影响：

- 观察器模式由 GUI 接口用于客户端(桌面和移动设备版本)。这些接口使用两层模型，其中包括 GUILayer 和 ControlLayer，所有向上的通信都在其 Java 版本中使用观察器模式：事件委托。在事件委托中，所有的通知都采用给监听器发布事件广播的形式。一些事件传送属性值中的变化，一些纯事件，如“登录成功”，不传送。
- 单一模式用于服务器端，其中每个实体有一个 home，以管理实体的创建和查找。一些 home 也有一般实用方法，例如 carModelHome.findEngineSizes()。
- 不使用多重模式。Java 的语法中最近添加了多重模式，但使用 int 类型的公共类常量仍比较普遍，例如 CarColor.MIDNIGHT\_BLUE(类常量是只读的(如 final)类字段)。
- 迭代器模式在客户机和服务器上用于访问 List 对象。
- 工厂方法在客户机和服务器上使用广泛。例如，业务层需要创建包含数据库中数据的实体时，就使用 home 中的工厂方法：CustomerHome.getInstance().create("Fred Bloggs", "02723359853",0)。
- 不使用抽象工厂模式，但是实际上，home 可以继承自一个常见的超类。
- 状态模式用于表示 Reservation 的状态机。

- 门面模式用于封装层。最简洁的例子是 ServerLayer 对象，它对客户隐藏了 BusinessLayer 的复杂性。
- 适配器模式在基于 Java 的客户机上用于通过控制器对象(它有一般的接口)连接 GUI 组件，它可以给事件监听器发送消息。
- 在下面的递增版本中，策略模式在用户界面中允许检索到的汽车根据用户选择的属性来排序。
- 模板方法用于抽象类，在子类之间共享逻辑(以提高代码的质量，改进可维护性，减少需要的工作量)。
- 次轻量级模式用于服务器，其中，业务层维护一个实体缓存，其中填充了数据库中的数据。次轻量级模式用于避免服务器进程中的数据重复，防止应用程序的不同部分使用某一数据的不同版本。
- 代理模式用于基于 Java 的客户机。ControlLayer 使用代理访问 ServerLayer 上的门面模式(来回传送轻量级的副本)。
- 复合模式用于 GUI 客户机，它是标准 Swing 功能的一部分，以构建组件树。

## 11.6 小结

本章的主要内容如下：

- 软件开发人员如何应用其他开发人员对某个问题的知识和经验，使用设计模式避免重复工作。模式描述了在现实世界中完成某个任务的行之有效的方式。
- 模式如何用于记录知识、编写解决方案的文档，帮助重用。
- [Gamma 等 95]提出的模式模板。
- 14 个最重要的模式，提供了这些模式的基础知识。
- 使用模式的实用建议：如何发现和合并模式，如何调整模式，使之适应特定的问题。

## 11.7 课外阅读

Gang of Four 图书[Gamma 等 95]尽管是用 OMT、C++ 和 Smalltalk 介绍的，但仍是非常有用的参考资料，它包含许多其他的论述和建议。

这本书出版之后，人们又出版和发表了许多图书和研究论文，并探讨了更多的领域。有关模式历史和状态的进一步论述，可访问网站 [www.hillside.net](http://www.hillside.net) 和 [www.patterndigest.com](http://www.patterndigest.com)。

对如何在 Java 中实现模式的完整介绍，可参阅[Grand 02]和[Grand 99]。

# 第 12 章 指定类的接口

本章介绍代码规范过程，这是传统上在实现之前的最后一个开发阶段。规范阶段允许记下软件如何以比前面更清晰、更精确的方式执行操作。

## 学习目标：

- 理解规范的含义和为什么要生成规范
- 理解正式和非正式规范之间的区别
- 以面向对象的方式编写规范
- 理解按合同设计的原理
- 在 Java 中使用规范和按合同设计

## 12.1 引言

在某种意义上，尽管前面在所有的阶段都指定了软件的行为(从系统用例一直到数据库模式)，但本章的内容主要与系统组件的接口相关，尤其是类。

为什么要编写规范？下面是一些原因：

- 删除早期阶段遗留下来的模糊性内容：模糊性内容表示制品可以以多种方式理解。用例一般用自然语言编写，这肯定是模糊不清的；分析阶段并没有告诉设计人员代码应如何实现，所以分析制品并不精确匹配最终的代码；而在设计阶段，虽然明确指定了编写什么类，它们有哪些消息和字段，但没有具体说明消息如何工作，或合法的消息参数、合法的字段值或其他这类细节由什么构成。
- 加深对实现方案的理解：对设计理解得越深刻，就越容易实现它。编写规范时需要仔细思考我们的设计，获得更透彻的理解。
- 提高对系统能工作的自信心：在尝试准确记录软件如何操作时，常常会发现遗漏了什么，或者假定了某个不可能的事，或者犯其他的人为错误。因此，规范可以在我们付诸实践之前找出错误。
- 有助于调试软件：软件出错时，可以检查实现代码是否匹配规范。因此，我们就有识别和更正错误的起点。
- 有助于测试软件：规范描述了软件的执行方式，测试可以验证软件是否按规范描述的那样操作。因此，测试可以用于检查软件是否符合规范。基于系统用例的测试可以验证生成的系统是否满足系统需求。基于规范的测试对于库、模式、框架和可重用对象来说是必不可少的，因为这些都没有涉及实际的系统。
- 有助于修改软件：如果决定以某种方式修改系统——例如添加功能，就可以检查这些修改是否违反了规范。如果违反了规范，这些修改就与当前的系统不兼容(或者规范中有错误)。我们不应违反规范，而应更仔细地检查所做的修改。

- 允许把已实现的任务交给其他开发人员：尽管其他开发人员仍会与我们联系，以澄清细节，但他们将较少需要这么做。
- 提供更好的文档说明：更好的文档说明将使系统更易于维护，重用性更好，误用的机会更少。

## 12.2 规范的定义

一般说来，规范是对软件所需的行为进行完整、清晰的描述。该软件可以是一个完整的系统、子系统、层、类或函数。规范应是完整、清晰的，但并不总是这样，或者不可能是这样。甚至不完美的规范都有其用处(有总比没有强)。

规范描述了一个或多个边界。例如，函数库的边界包括函数的签名(名称、返回类型和参数类型)和全局数据值；对象的边界包括消息和属性，类的边界(与对象的边界略有区别)包括类消息和类属性。尽管属性在逻辑上位于边界上，但它们应能由对象和类消息访问。

如果每个边界都在一端有一个客户，在另一端有一个提供者，规范就会描述如下内容：

- 客户可以通过边界传送什么信息，它们何时可以传送信息。例如，客户只能在事务开始后，给顾客对象传送新地址信息。
- 客户可以从边界上检索什么信息，它们何时可以检索信息。例如，客户只能在登录后，从图书服务器上检索要销售的图书列表。
- 事件何时广播给已注册的客户，客户会接收到什么信息。例如，在分布式聊天系统中，每个客户都会获知每个进入聊天室的新客户名。
- 边界管理的信息的合法状态。例如，集合对象可能会声明，它从来不会包含负的对象数。

大多数情况下，规范关注的都是软件的公共边界。但是也可以指定内部边界的性质。例如，考虑一个函数库，它为集合结构提供了排序功能。这个库有一个可重用的 swap 函数，用于交换集合中的两个值。swap 函数不可能是排序库的公共接口的一部分，因为它是一个具体的实现细节；但是，仍可以指定它的行为——例如可以指定，在调用该函数后，左边的值会占据右边的值以前占据的位置，而右边的值会占据左边的值以前占据的位置。

规范有两个变体：正式规范是科学的、严格的，而非正式规范是实际的、局部的，但不是没有用处。

## 12.3 正式规范

大多数的软件开发都是一个不严密的活动。历史上，软件像开发更多的是基于希望，而不是肯定它应完成设定好的工作。这是因为没有支持它的科学，这不像城市工程或硬件工程，它们得到了机械和量子理论的支持。大多数好的软件都是人类的独创性、经验、推测、重用和测试的结果。

这很适合于不那么重要的应用程序，例如字处理程序和消费操作系统。我们可以容忍字处理程序偶尔在编辑的过程中崩溃(这就是有自动恢复功能的原因)。消费操作系统有时也会崩溃，我们除了重新启动之外，别无选择(这就是系统管理员运行自动启动的备份的原因)。我们可能会浪费时间、感到绝望或生气，但没有人是一帆风顺的。毕竟，使用计算机所带来的好处一般

超过了不完美的计算机带来的缺陷：从整体上看，有了计算机，生产率比以前提高了。

但是，一些情形需要完美或接近完美的可靠性。对安全性要求很高的(safety-critical)系统就是一个这样的例子，这些系统包括核电站中心的软件、航空交通管理系统、自动驾驶的航行器和医疗设备。这类系统(或者它们的操作系统)是不能失败的。实际上，我们不可能保证有 100% 的可靠性，因为这太困难了，或者太昂贵了，或者需要了解每个可能的异常条件。如果失败之间的间隔很长(例如核电站的控制系统平均在一万亿年内发生故障的次数少于一次)，或者可靠性比人高(例如，自动驾驶的飞机犯的错误是人类飞行员的十分之一)，就是可以接受的。

提高可靠性的一种方法是复制硬件或软件。例如，可以安装三台计算机，来控制自动驾驶的飞机。每台计算机都使用不同的硬件、不同的操作系统和不同团队开发的软件。把这三台计算机放在一个投票系统中，在该系统中，除非至少两台计算机同时发出命令，否则飞机就忽略该命令。从理论上看，使用这种技术，失败的可能性非常小。(这仍不可能是完美的，因为两台计算机可能同时以相同的方式失败，使飞机执行不正确的命令)。对这类可靠性保证的进一步讨论超出了本书的范围。

提高可靠性的另一种方式是，从数学上证明软件是正确的(即软件完成了预设的任务)。这需要三步：

1. 用数学语言生成一个正式规范，描述软件如何动作。
2. 证明该规范是可行的，例如，它没有逻辑矛盾或不可能的情形。
3. 证明软件遵循该规范。

正式规范语言有 Vienna 开发方法(Vienna Development Method, VDM, 来自于 IBM Vienna)、Z(来自于牛津大学)和对象约束语言(Object Constraint Language, OCL, 是 UML 的一部分)[OMG 03b]。

例如，如果为 squareRoot() 函数的行为编写正式规范，就要指定下面的内容：

- 输入值必须是正的(假定非复数)。
- 结果的平方等于输入值。(当指定可逆函数时，这是一个常见的技巧：把逆函数应用于结果，一定会得到输入值)。
- 结果是正的。(因为  $2 \times 2=4$ ,  $(-2) \times (-2)=4$ ，所以 4 有两个平方根，这里选择返回正的平方根)。

下面用 VDM 指定 squareRoot() 函数的边界条件：

```
squareRoot(x: R)y: R
pre x ≥ 0
post (y2=x) ∧ (y ≥ 0)
```

其中 R 表示实数，pre 表示前提条件(在调用函数前必须为真的条件)，post 表示后置条件(在调用函数后必须为真的条件)。规范中的每个逻辑表达式都表示为一个断言(assertion，所以  $y \geq 0$  断定，y 大于或等于 0)。即使不了解 VDM，也很容易看出上述规范表达的是：

squareRoot() 函数把一个实数 x 作为参数，返回一个实数 y。要进行正确的操作，该函数要求 x 大于或等于 0。如果满足前提条件，函数就可以保证 y 的平方等于 x，且 y 大于或等于 0。

因此，可以指定 squareRoot() 函数的确切行为，而无需了解其实现方式。

正式规范是一个需要大量训练和技巧的学科，也是一个很长的过程。有时，可以使用计算

机验证规范(自动证明定理, automated theorem proving)。在非常有限的条件下, 甚至可以验证实现代码是否匹配规范。但是, 计算机不能编写出规范来。

即使可以依赖正式规范的数学内容和软件的实现方式, 也不能依赖操作系统、硬件或编译器。所以, 对安全性要求很高的软件有时用汇编语言实现, 然后部署到删节的操作系统上, 以便有机会证明最终结果是正确的(甚至这样也不能阻止宇宙射线穿透芯片, 把一个位从 1 变为 0, 所以也需要外部的自动防故障装置)。也不能证明正式规范满足系统需求, 因为后者通常一开始是用自然语言表述的, 并会随着时间的推移做微妙的改变。

正式规范的生成和使用非常困难, 也很费时, 所以它们一般只用于对安全性要求很高的系统和需要非常可靠的技术的系统, 例如投票系统。从整体上看, 正式规范对软件业来说是不切实际的, 但我们仍可以使用其底层的原理, 并获益良多。通俗地说, 本章后面描述的技术称为非正式规范。有了正式规范理论的支持, 至少可以弄清楚非正式规范应包含什么内容。

## 12.4 非正式规范

所有的程序员都在某种程度上使用非正式规范——给函数添加注释, 便于其他程序员理解就是一个简单的例子。函数注释描述了下述列表中的一些或所有信息:

- 客户何时可以调用函数
- 应传送什么参数
- 函数完成什么任务
- 返回什么结果(类型和值)
- 函数对全局数据有什么影响
- 如果出问题, 函数会采取什么措施

这些信息也可应用于子例程、过程和方法, 因为这些都是函数的变体。

通常, 函数注释不会提及内部的实现方式——与规范的其他形式一样, 注释只描述边界上的条件。注释偶尔也会提及实现方式, 但仅在必要时提及, 且使用抽象的术语来说明。例如, 客户程序员知道某个 `sort` 函数的时间/空间折衷方案是很有用的, 例如“这个函数的执行速度是  $n \log n$ , 需要  $2n$  的内部空间来处理”。

下面的 C 代码(C 语言非常类似于 Java 语言)包含一个注释, 它用作 `squareRoot` 函数的非正式规范:

```
/*
 Returns the positive square root of x.
 Preconditions: x >= 0.
 */
float squareRoot(float x);
```

读了这个注释就知道, 只要提供一个正的 `x`, 就会得到 `x` 的正平方根。

大多数程序员都很熟悉“前提条件”这个术语, 或者很容易理解它。如果要获得最大的可靠性, 就可以使用“要求”来替代“前提条件”, 用“确保”替代“后置条件”, 或者用自然语言编写前提条件和后置条件。上述非正式规范是不完整的, 这与前面给出的正式 VDM 版本不同: 例如, 我们没有说过结果的平方等于参数。在下一节中, 要讨论为什么这个后置条件在编

程中不像表面上那么直接。

函数签名(名称、返回类型和参数类型)包含许多信息，所以可以构成非正式规范的一部分。例如，`float squareRoot(float x)`隐含着，该函数带一个 `float` 值，返回一个 `float` 值；结果是参数的平方根。但是，甚至非正式规范都不应依赖隐含的信息，因此也不应依赖显式的注释。注释和签名都可以包含在许多 UML 制品中，所以源代码不是展示非正式规范的惟一地方。

本书前面提及的另一种非正式规范是用例。每个用例都描述了下述信息的部分或全部：

- 何时可以使用用例(前提条件)
- 用例做什么工作(步骤和后置条件)
- 用例对系统有什么影响(步骤和后置条件)
- 在异常情况下会发生什么

于是，用例描述了外部参与者为正确使用系统所需要知道的所有内容，因此它们类似于函数注释。这没有什么可惊讶的，因为按照定义，用例就是用来描述边界的——在这种情况下就是系统边界。本章感兴趣的是子系统设计和实现阶段生成的规范，所以下面不讨论用例。

一些编程语言，如 Eiffel，可以用特定的语法在源代码中记录非正式规范，与注释分开。

## 12.5 动态检查

正式规范的作者使用与软件实现方式不同的数学表示法。正式规范器可以证明它们生成的规范没有逻辑错误。在有限的环境下，它们甚至可以证明某个软件的实现方式遵循该规范。但是一般情况下，正式规范器依赖程序员使用知识和直觉来生成遵循规范的代码。

非正式规范包含一种实际的技术，称为动态检查。动态检查是嵌在实现方式中的代码，可以验证软件正在执行(例如它没有违反规范)。为了说明动态检查的工作原理，考虑平方根函数。下面是该函数的 C 实现代码(省略了计算过程)：

```
/*
    Returns the positive square root of x.
    Preconditions: x>=0.
*/
float squareRoot(float x) {
    if (x < 0) {
        fail("squareRoot", "Parameter x can't be negative");
    }
    .../* Code to calculate y */
    return y;
}
```

在这个函数中，有一段代码检查前提条件  $x \geq 0$  是否满足——如果不满足，函数就退出。为了方便，提供了一个 `fail` 函数，输出函数名和错误消息，然后终止程序(使用 C 库函数 `exit`)。(一些语言提供了更优雅的方式，使用异常处理来发出失败的信号，但功能变化得太多了)。

客户程序员现在有了保护：在大多数情况下，客户用一个正数调用 `squareRoot`，函数会成功完成，但是，偶尔客户会用一个负数调用 `squareRoot`，此时程序不会继续执行，直到更正了错误为止。

您可能认为 `squareRoot` 函数在执行一般的错误检查，这肯定是程序员常常做的工作，无论

他们是否理解规范过程。有趣的是这段错误检查代码只能放在这里，才能验证客户程序员没有违反规范——如果他们违反了规范，代码中就会有一个错误。

动态检查只对非正式规范有意义，对正式规范没有意义。这有许多原因，其中一些如下：

- 正式规范的目标是保证实现方式不违反规范。而在动态检查中，实现方式可以包含错误，因此在运行期间可以找出违反规范的地方。
- 一些正式需求是不可能进行检查的。不能检查并不意味着应把它从正式规范中删除。
- 一些正式需求不能使用指令性(imperative)代码表达出来(指令性编程语言目前最常见的类型是程序员必须明确地告诉计算机做什么的语言，而在声明性(declarative)编程语言中，程序员只需说明结果应是什么即可)。这种需求仍应包含在正式规范中。
- 一些正式需求不能精确地表达为指令性代码。例如，在 `squareRoot` 中，可以指定函数返回的结果是平方根。一种方法是指定结果的平方等于参数，例如“对于结果 `y`,  $(y*y)==x$ ”。但指令性算术是不精确的：`squareRoot(4.0)` 返回 `2.0`, 但 `squareRoot(3.79512)` 返回 `1.94811`, 这个数字就不是绝对正确。

为了处理这种不精确性，如何说明结果接近某个值？有一个计算分支学科，称为“数字方法(numerical method)”专门研究这类问题。要使用数字方法，需要计算出最大的可能错误值，并在规范中使用它。

在 `squareRoot` 的例子中，如果把最大的可能错误值封装在一个函数 `squareRootError` 中，再访问 `pos` 函数，`pos` 返回其参数的正数部分，就可以指定“对于结果 `y`,  $pos((y*y)-x) \leq squareRootError()$ ”。这个规范可以添加到函数注释中，在最后编码为一个检查：

```
/*
 Returns the positive square root of x.
 Preconditions: x>=0.
 Postconditions: For result y,
                 (y>=0) and pos((y*y)-x) <= squareRootError().
 */
float squareRoot(float x) {
    if (x < 0) {
        fail("squareRoot", "Parameter x can't be negative");
    }
    .../* Code to calculate y */
    if (y < 0) {
        fail("squareRoot", "Calculation gave negative result");
    }
    if (pos((y*y)-x) > squareRootError()) {
        fail("squareRoot", "Calculation was inaccurate");
    }
    return y;
}
```

现在不只确保客户程序员在他们的代码中没有错误，还保证在我们的代码中没有错误。

如果不能为最大的可能错误计算出一个值(或者觉得这太麻烦)，就应把规范以自然语言添加到注释中：

```
...
The square of the result is equal to x,
```

subject to the accuracy of the current platform.  
\*/

## 12.6 面向对象的规范

正式规范语言，如 VDM，最初是用来和结构化软件开发一起使用的。一些语言已经扩展为和面向对象的语言一起使用。其他语言，如 OCL，则专门用于面向对象的软件开发。结构化表示法和面向对象表示法之间的主要区别是，面向对象表示法使用任意数量的类作用域，而不是一个全局作用域（“作用域”表示命名空间、分区、子区域或可以围起来的其他东西）。

面向对象的规范比过程式规范简单，因为所有的规则都与其相关的概念（类和对象）有关。面向对象的规范语言必须可以断言：

- 消息何时可以合法地传送给一个对象（消息前提条件，用公共属性表示）
- 每个消息的有效参数（消息前提条件）
- 消息对接收对象的影响（消息后置条件，用公共属性表示）
- 每个消息的有效回应（消息后置条件）
- 对象总是满足的条件（类不变式(class invariant)，对类的实例总为真的条件，用公共属性表示）

除了对象、对象属性、对象消息之外，还有类、类属性和类消息；所以，上述五类也可以应用于类，但通常很难从表示法上看出其区别。

例如，考虑 Container 类和 add 消息。假定要断言：

- 前提条件：传送给 add 消息的对象引用必须不为空。
- 前提条件：传送给 add 消息的对象必须不在 Container 中。
- 后置条件：在 add 消息之后，Container 中的对象比以前多一个。
- 后置条件：在 add 消息之后，Container 包含作为一个参数传入的对象。
- 不变式：Container 包含的对象数总是正的。

在下面两节中，将探讨如何在 OCL(正式规范语言)和 Eiffel(一种指令性编程语言，包含非正式规范的语法)中表达 Container 断言。

### 12.6.1 OCL 中的正式规范

OCL 是 UML 的正式规范语言，很适合把 UML 用作表示法的面向对象方法。对于 Container 例子，可以使用下面的 OCL(其中 contains 方法只有在接收器包含 o 时才返回 true)：

```
context Container::  
    add(o:Object)  
        pre: (o<>null) and not contains(o)  
        post: contains(o) and (size = size@pre +1)  
context Container::  
    inv: size>=0
```

OCL 允许使用 context 关键字把断言连接到一个类上，这表示对于复杂的语言，如 C++，也可以在规范中引用全局函数和全局数据。后置条件常常需要在执行方法之前指定属性值——这就允许讨论方法对接收器属性的影响。在本例中，size@pre 表示在执行 add 之前 size 的值。

上述 OCL 片段可以理解为：

在给 Container 发送 add 消息时，参数 o 必须是非空的，o 不能已在 Container 中。一旦方法执行完，Container 就包含 o，Container 的大小比执行方法之前大 1。Container 的大小总是大于或等于 0。

### 12.6.2 Eiffel 中的非正式规范

Eiffel 是 Bertrand Meyer 在 20 世纪 80 年代中期开发的一种语言[Meyer 90]，它是用非正式的、可动态检查的规范设计的。Eiffel 程序员可以把断言放在源代码中，但与注释和实现代码分开：断言由 Eiffel 编译器识别为语法的一部分。Eiffel 使用 require 关键字代替 pre，用 ensure 替代 post，用 old 替代@pre。对于 Container 例子，可以使用下面的 Eiffel 片段：

```
class CONTAINER

feature {ANY} .-Public stuff follows

    size: INTEGER

    add(o:OBJECT) is
        require
            (o != void) and not contains(o)
        do
            ...
        ensure
            contains(o) and (size = old size + 1)
        end

    invariant
        size >= 0

end
```

Eiffel 混合了规范和实现代码，这对程序员非常方便。另外，编译器懂得断言语法，所以断言可以验证(以确保它们不包含语法错误或语义错误)。

与前面 squareRoot 函数的 C 版本一样，程序员可以编写代码，动态地检查断言。在 Eiffel 中，编译器很容易识别断言，因为断言与方法体是分开的，编译器还可以自动生成检查代码，因为它们懂得断言语法。

就前提条件和后置条件而言，自动检查的作用就好像插入了 if 语句，来检查方法开头的 require 部分和最后的 ensure 部分。但不变式如何检查？不变式子句中的断言对类的对象来说必须总为 true，但是，为了检查断言，需要执行一些代码，在指令式面向对象语言中，可以执行代码的惟一地方就是在方法中。所以，不变式需要在运行方法时检查。

如果不改变在每个方法的开头处检查，就不能确定对象本身是否违反了不变式；如果它们在方法的最后检查，就不能确定调用方法时是否已经违反了不变式。因此，不变式必须在方法的开头处(在前提条件之前)和结尾处(在后置条件之后)检查。在前提条件之前和后置条件之后检查的原因是，前提条件和后置条件都可以改变属性(即使不应改变)。实际上，不变式给每个

方法增加了额外的前提条件和后置条件。

您可能认为，如果总是可以确定方法没有违反不变式，就能肯定在输入一个方法时，不变式是安全的。但是，系统中的其他对象可能修改属性(给它们发送消息)，这会潜在地违反不变式。因为其他对象没有检查不变式的业务，所以必须假定可能在启动每个方法时违反不变式。

## 12.7 按合同设计

只要软件在运行，就可能出错：程序中可能有错；用户可能提供了不正确的数据；程序依赖的外部软件可能失败；运行时系统、操作系统或硬件可能失败。

如果每个软件都是运行在某个平台(操作系统加硬件)上的独立进程，就可以把软件分解为进程中的活动和进程外的活动。在很大程度上，可以使用优秀的设计、正确的编程规则和合理的测试原则控制进程内发生的情况，但不能控制进程外部的情况(即使访问另一个可靠实现的进程，平台提供的进程之间的通信也可能失败；另外，进程还可能访问硬件，如网络或文件系统，而这经常会失败)。在编写出一流、可靠、高效的软件之前，必须深刻地理解各种不同的失败，弄清楚由谁负责处理这些失败。

在第一次学习编程时，大多数人都假定不会出错。而运行一个程序时，出现错误，程序就会失败，于是我们认识到，最好防止出现使程序失败的错误——错误恢复(error recovery)，或者确保程序在停止时给操作者提供解释性消息——从容的失败(elegant failure)。程序中可能有许多错误，所以应在许多地方检查错误。更糟糕的是，最佳编程实践要求，应独立实现软件模块(函数、子系统、类)，这将很难确定由谁负责检测错误，谁负责处理它们。从而可能导致大量的错误检查代码，其中大多数都是多余的。

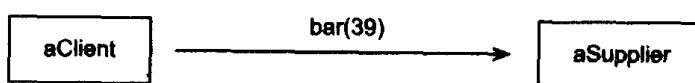


图 12-1 给对象发送消息

下面看看面向对象程序中最糟糕的情况：按敬畏程度来设计(design by fear)。例如，一个对象叫做 aClient，在它的 foo 方法中，要给 aSupplier 传送一个消息 bar，如图 12-1 所示。(两个对象运行在同一个进程中)。假定这两个对象的代码都是独立编写的(即使两个类都由同一个人实现，两个类的实现代码也有明显的区别，编写代码的人在经过一定的时间后就会忘记类中的约定)。

假定编写 Client 实现代码的是 Beryl，编写 Supplier 实现代码的是 Fred。在理想的情况下，Beryl 简单地编写了 `result = aSupplier.bar(anObject)`，并假定 result 是有效的。在同样理想的情况下，Fred 假定 bar 正确调用，执行计算，返回结果。考虑到世界是不完美的，两个程序员都格外小心。

在编写 foo 方法时，Beryl 认为，在发送消息之前，最好检查一下 aSupplier 是否处于接收 bar 消息的对应状态下。所以她添加了检查 aSupplier 状态的代码。完成后，她认为，由于不能确保要传送的参数是否有效，所以最好也检查一下。编写完两段检查代码后，她添加了发送消息的代码。对于返回的结果，Beryl 认为，由于不能确保得到的结果是否有效，因此添加了测试结果的代码。但是，Beryl 是非常仔细的。她认为在回应了消息后，aSupplier 可能没有处于

有效状态下——毕竟，消息可能有程序员不能预料的影响。所以，Beryl 添加了第四段错误检查代码，以确保在继续之前，`aSupplier` 是有效的。

Beryl 的 `foo` 方法如下所示(用 Java 编写):

```
void foo() {  
    ...  
    if (!... check aSupplier ...) {  
        fail("Invalid state of a supplier");  
    }  
    if (!... check anObject...) {  
        fail("Invalid parameter for a supplier message");  
    }  
  
    int result = aSupplier.bar(anObject);  
  
    if (!... check result ...) {  
        fail("Invalid result from a supplier message");  
    }  
    if (!... check aSupplier...) {  
        fail("Invalid state of a supplier");  
    }  
    ...  
}
```

为了避免嵌套的 `if` 语句，检查代码的逻辑用 Java 的 `not` 操作符 “!” 来表示。`fail` 函数是 `Client` 上的一个方法，可以输出类名、失败的方法名和传送的参数，然后用 `System.exit(-1)` 停止进程。

在 Fred 编写 `bar` 方法时，他也尽可能编写强健、防止出故障的代码。在执行计算之前，他检查当前的对象是否处于接收 `bar` 的有效状态下。接着，检查参数是否有效——毕竟，相信客户是很笨的。一旦确定接收者已准备好，且传送了有效的数据，Fred 就添加计算结果的代码。在 `return` 语句之前，Fred 认为最好检查一下结果是否有效——因为他编写的计算代码或调用的其他代码可能会出错。到此 Fred 仍不满意，他又添加了一段检查代码，以确保计算不会破坏当前对象。

Fred 的 `bar` 方法如下所示:

```
public int bar(Object anObject){  
    if (!... check this object...) {  
        fail("Invalid state of this object");  
    }  
    if (!... check anObject...) {  
        fail("Invalid parameter from a client");  
    }  
  
    int result = ...  
  
    if (!... check result ...) {  
        fail("Invalid result for a client");  
    }  
    if (!... check this object...) {
```

```

    fail("Invalid state of this object");
}
return result;
}

```

最后得到非常多的代码——错误检查代码可能超过了正常的代码：aSupplier 的状态在计算前和计算后分别检查了两次；anObject 检查了两次；result 检查了两次。读者可能不像 Fred 和 Beryl 那么仔细，但编写出来的代码仍可能类似于上述代码。当 foo 和 bar 编写为独立的函数、过程或子例程时，就会出现这种情况。

如果采用下述常见技巧之一，错误检查代码就可能会更烦琐：

- 结果代码：结果代码是从例程返回的一个数字，表示成功或失败，例如 0 表示成功，负数表示失败。结果代码一般可应用于任何类型的例程，无论例程是否返回结果，这会导致许多“如果结果代码正确”的检查。
- 全局错误变量：这里，程序员不是返回一个成功值，而是设置一个全局值(例如 C 函数库中的 errno)。函数在正常情况下返回它们的值，如果有问题，就设置错误变量。但是，这会导致许多“如果错误变量不是 0”的检查。(全局错误变量的面向对象版本是类 Error 上的一个类属性)。
- 异常：异常是提供代码产生的一个错误，会使程序跳转到与正常路径代码分隔开的处理程序代码。使用异常，可以从正常代码中删除错误检查代码。但是，如果没有指定谁负责处理异常的设计规则，异常会使事情更糟糕。

在 20 世纪 80 年代，Bertrand Meyer 描述了如何以注重实效的方式合并形式方法的最佳特性和面向对象编程，以便于生成强健、一流、高效的代码。Meyer 把这个技术称为“按合同设计(Design by Contract)”。Meyer 是在 Eiffel 中描述这个技术的，而使用断言和动态检查实现“按合同设计”，就可以在其他语言中使用该技术——前面已经介绍了在 C 中使用该技术的一些例子(不要忘记“设计”这个词，合同的概念与软件开发的所有阶段都相关)。

在按合同设计时，要假定客户对象和提供者对象之间建立了一个合同，对双方进行了约束。其理念是，只要客户履行了其义务，提供者就也要履行其义务(如图 12-2 所示)。

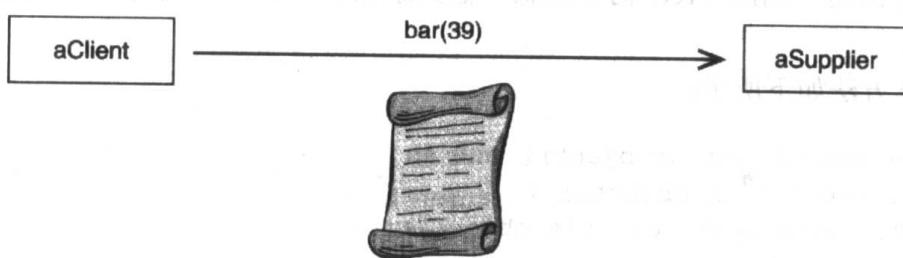


图 12-2 客户和提供者之间的合同

使用正式的规范术语，可以准确地表述合同约定的义务：

- 客户必须遵循提供者的不变式。
- 使用提供者方法要满足其前提条件。
- 提供者方法的实现代码必须保证其后置条件。
- 提供者方法的实现代码必须保证提供者的不变式。

在方法完成其任务时，可以违反合同，只要在任务完成时满足合同的要求即可。例如，如果 Supplier 不变式是  $a+b=4$ ，且在调用 bar 时两个属性值都是 2，bar 就可以把 a 设置为 10，把 b 设置为 -6；但是，在大多数编程语言中都不能同时设置两个值，所以在 bar 中  $a+b$  有一段时间是 12 或 -4。

类也可以有内部的合同，对于非公共属性，其形式是不变式，对于非公共方法，其形式是前提条件和后置条件。内部合同对实现代码和维护代码的人都很有利，但其重要性不如外部合同。

### 12.7.1 合同和继承

面向对象编程支持继承，所以应考虑继承对合同的作用。由于多态性的存在，从客户的角度来看，必须保证每个子类合同与其超类合同一样好，或比其超类合同更好。没有这个规则，程序员就可能通过超类变量来使用子类变量。例如，如果类 Customer 的合同指定，所有的顾客都至少有两个银行账户，再引入只有一个银行账户的 SimpleCustomer 子类，则通过 Customer 变量访问 SimpleCustomer 对象的人就可能试图访问第二个不存在的银行账户。

沿着类层次结构向下，类不变式将使用 and 来合并：只能向不变式中添加内容。例如，如果类 X 有不变式 i1，子类 Y 添加了不变式 i2，则 Y 的总不变式就是 i1 and i2。其含义是，为属性提供了额外的保证。

只有在子类中重新定义相关的方法，前提条件和后置条件才会成为一个问题。对于前提条件，必须确保重新定义的方法至少要接受它在超类中接受的所有消息；只能弱化(weaken)前提条件。因此，给重定义方法添加的任何前提条件都使用 or 与继承方法上的前提条件合并。例如，如果 foo 有一个前提条件 pre1，子类添加了前提条件 pre2，则结果就是，重定义方法的前提条件就是 pre1 or pre2。

对于后置条件，必须确保每个子类都至少与其超类有相同的后置条件；只能强化(strengthen)后置条件。因此，后置条件使用 and 来合并。例如，如果 foo 的后置条件是 post1，子类添加了后置条件 post2，重定义方法的后置条件就是 post1 and post2。

- 如果省略了不变式，就表示“这个类不需要保证什么”；如果超类上没有不变式，而给子类添加了一个不变式，结果就是添加给子类的不变式。
- 如果省略了前提条件，就表示“这个消息可应用于所有的情况”；如果重定义方法原来没有前提条件，现在给它添加了一个前提条件，新的前提条件将不起作用(不能弱化能接受任何消息的方法)。
- 如果省略了后置条件，就表示“这个消息不能保证什么”；如果重定义方法原来没有后置条件，现在给它添加了一个后置条件，结果就是添加重定义方法的后置条件(添加的任何内容都会强化后置条件)。

利用按合同设计的理念和正式规范术语，程序员 Beryl 就可以使代码更清晰：

```
void foo() {  
    ...  
    if (!... check aSupplier's invariants ...) {  
        fail("Invariants broken for a supplier");  
    }  
    if (!... check bar's preconditions...) {  
        fail("Preconditions broken for a supplier message");  
    }  
}
```

```

int result = aSupplier.bar(anObject);

if (!... check bar's postconditions...) {
    fail("Postconditions broken by a supplier method");
}
if (!... check aSupplier's invariants ...) {
    fail("Invariants broken by a supplier method");
}
...
}

```

同样，Fred 也可以修改其代码，如下所示：

```

public int bar(Object anObject) {
    if (!... check invariants ...) {
        fail("Invariants broken");
    }
    if (!... check preconditions...) {
        fail("Preconditions broken");
    }
    ...
    int result = ...

    if (!... check postconditions...) {
        fail("Postconditions broken");
    }
    if (!... check invariants ...) {
        fail("Invariants broken");
    }
    return result;
}

```

## 12.7.2 减少错误检查代码

按合同设计的主要优点是把客户的责任和提供者的责任清晰地分开。在发送 bar 消息之前，客户在控制之下。于是，客户有责任保证在发送消息之前不违反合同。在执行方法时，提供者在控制之下。于是，提供者有责任保证在方法返回之前不违反合同。

下面把这个责任理念应用于更乐观的 Fred 和 Beryl。这次 Fred 和 Beryl 互相信任：Fred 编写 bar 方法时假定 Beryl 不会违反合同；而 Beryl 相信 Fred 也不会违反合同，所以直接使用 bar 的结果。Beryl 的代码现在如下所示：

```

void foo() {
    ...
    if (!... check aSupplier's invariants ...) {
        fail("Invariants broken for a supplier");
    }
    if (!... check bar's preconditions...) {
        fail("Preconditions broken for a supplier message");
    }
    ...
    int result = aSupplier.bar(anObject);
}

```

```
...  
}
```

Fred 的代码则如下所示：

```
public int bar(Object anObject) {  
    int result = ...  
    if (!... check postconditions...) {  
        fail("Postconditions broken");  
    }  
    if (!... check invariants ...) {  
        fail("Invariants broken");  
    }  
    return result;  
}
```

按合同设计说明了如何删除所有的冗余代码，把检查代码减少一半。这样开发就更快，性能更好，错误更少，维护更容易。

还可以使用按合同设计进一步减少检查代码。在许多情况下，我们编写的代码都不可能违反合同。例如，考虑下面 Supplier 的实现代码：

```
// A supplier  
// Invariant: x < 5  
public class Supplier {  
  
    private int fieldX;  
  
    /*  
     * Constructor  
     */  
    public X() {  
        fieldX = 1;  
    }  
    /* Getter for x  
     * Preconditions: true (call this method any time)  
     * Postconditions: true (no side effects)  
     */  
    public int getX() {  
        return fieldX;  
    }  
    /* a method  
     * Preconditions: o != null  
     * Postconditions: for result r, r > x  
     */  
    public int bar(Object o) {  
        fieldX = o.toString().length() % 5;  
        return fieldX + 1;  
    }  
}
```

在这个例子中，x 表示 UML 样式的属性——x 的值可以通过消息 `getX` 从外部访问。为了强调公共属性和私有实现代码之间的区别，字段有另一个名称 `fieldX`。

从编写的方式来看，Supplier 代码总是遵循不变式和后置条件：

- 在创建 Supplier 后，x 小于 5，因为 fieldX 由构造函数设置为 1。
- 在执行 getX 后，x 小于 5，因为 getX 没有副作用。
- 在执行 bar 后，x 小于 5，因为 fieldX 设置为被 5 除后的余数。
- 在执行 bar 后，结果大于 5，因为 bar 返回 fieldX + 1。

由于 Supplier 不可能违反合同，因此不需要检查代码(原则上，平台失败会违反合同，但为这种情况添加检查代码是不可行的)。

现在考虑 Client 上 foo 的下述实现代码：

```
void foo() {  
    Supplier aSupplier = new Supplier();  
    int result = aSupplier.bar(new Plate("Wedgwood"));  
    ...  
}
```

在发送 bar 消息时，foo 不可能违反 aSupplier 的不变式，因为它没有触碰新对象的属性。另外，foo 也不可能违反 bar 的前提条件，因为它传送的是非空对象。所以，foo 不需要任何检查代码。最终，这个程序没有任何检查代码。

现在有了定义明确的义务、相互信任，所编写的代码减少了一半的错误检查代码，甚至在最后删除了所有的错误检查代码。但软件存在于不完美的世界中，该如何处理：客户和提供者代码都可能包含错误；如果从当前的进程跳转到另一个进程或另一个硬件上，就可能出现不能控制的情况。进程中的错误可以通过履行合同(enforcing the contract)来处理，进程外部的错误可以通过构建应用程序防火墙(application firewall)来处理。下面就依次介绍这些技术。

### 12.7.3 履行合同

按合同设计依赖于如下假定：只要客户对象履行了其义务，提供者就也要履行其义务。但程序员可能犯错误；有时义务也没有履行，所以需要动态检查，来检测错误。但由谁负责检查合同义务的履行情况？如果检测到错误，又该怎么办？

在客户和提供者的例子中，bar 方法应负责在返回前，检查前提条件和不变式，因为只有 bar 方法或其提供者才有可能在方法开头和结尾之间违反合同。所以，对合同中提供者部分的动态检查(后置条件和不变式)应在 bar 方法的最后进行。

在 bar 开始之前检查不变式和前提条件的问题要复杂一些。客户代码负责确保不违反合同中的客户部分。但是，如果 bar 的每个客户都必须有检查合同的代码，这些代码就会在整个系统中不断地重复。

面向对象理论指出，责任应赋予相关的对象。这说明，检查合同中的客户部分应是提供者提供的一个服务(因为前提条件和不变式与提供者相关)。所以，为了避免重复客户的合同检查代码，应把这些代码放在 bar 的开头。这有两种方式：bar 阻止提供者在客户中犯错，或者 bar 为客户提供一个服务，来检查它是否有错误(在按合同设计中，第二个解释比较好)。

把非正式方法、按合同设计、信任和动态检查合并起来，bar 方法的 Java 实现代码如下所示：

```
public int bar(Object anObject) {  
    if (!... check invariants ...) {  
        fail("Invariants broken");  
    }
```

```

}
if (!... check preconditions...) {
    fail("Preconditions broken");
}

int result = ...

if (!... check postconditions...) {
    fail("Postconditions broken");
}
if (!... check invariants ...) {
    fail("Invariants broken");
}

return result;
}

```

有了上述实现代码，客户中就不需要合同检查代码了。

#### 12.7.4 应用程序防火墙

只要位于一个进程中，就可以依赖合同来提供一流、强健的代码和性能。但跳出当前的进程后，就没有合同来保护我们了。所以，应考虑超出进程范围之外的东西，这至少包括：

- 其他进程
- 操作系统
- 用户界面
- 文件系统
- 网络
- 数据库
- 设备

可以在边界上构建一个应用程序防火墙，保护进程内部的代码，例如，用户界面防火墙可以在把用户数据传送给业务对象之前，检查这些数据的有效性，数据库防火墙可以捕获数据库错误(这与捕获内联网的 Internet 防火墙没有关系)。

有时，可以防止无效数据进出应用程序防火墙。例如，如果用户必须输入用户名和密码才能登录，就可以在用户登录之前，在 GUI 上禁用 Login 按钮。

### 12.8 Java 中的非正式规范

如果使用 Eiffel 等语言实现设计，最佳实践(使用非正式规范和动态检查)就内置于该语言内部。在使用其他语言时，应使用类似的规则，但必须手工编写动态检查代码。本节介绍如何给 Java 应用规范理论，因为 Java 是 Eiffel 的一种常见、注重实效的替代语言。

#### 12.8.1 使用注释编写合同文档

至少应在源代码中使用注释编写类的合同文档。这是因为代码最终是由查看源代码的程序员使用、重用或维护的。源代码中与合同相关的任何注释都应出现在设计和规范文档中。

每个类都应在其顶部有一个很长的注释——类注释，它描述了类的作用和其公共属性的不变式。注意，查看源代码的其他程序员可能不熟悉正规的术语，所以应使用“对于这个类的所有对象，应用下面的条件”等词语。类注释还可以包含类如何使用的例子。

类上的每个公共消息都应在其顶部有一个注释——消息注释，它描述了对应方法的作用，列出了所有的前提条件和后置条件。为了便于不熟悉术语的读者阅读，应使用“要求(requires)”和“保证(guarantees)”等词语，而不是“前提条件”和“后置条件”。

记住，在实现方法时，调用者应负责确保在调用方法之前，满足不变式和前提条件。换言之，编码方式应是乐观的。当然，它的一个副作用是，应确保实现代码不违反后置条件或不变式。

## 12.8.2 动态检查条件

在原则上，最好在每个公共方法的开头添加不变式检查和前提条件检查，在公共方法的结尾添加后置条件检查和不变式检查。但是，如果没有特定的语言机制，例如 Eiffel 提供的这种语言机制，还可以选择其他方式。

最好在每个要多次重用的公共方法开头添加检查(例如代码用于一个库)。第三方可能会欣赏检查代码提供的额外帮助，因为他们很熟悉类。

对于公共方法结尾的检查，就是一个个人喜好的问题了。如果方法不可能违反合同中的提供者部分，这些检查就是多余的。如果觉得代码可能违反合同，也许就不相信自己了：随着实现面向对象代码的经验的增长，自信心也会增加，这样，这些检查就会变成多余的了。

在方法结尾包含检查的惟一原因是，相信代码所涉及的第三方代码可能违反不变式，破坏属性：大多数代码都是在客户—服务器样式中调用，所以这种情况并不常见。

## 12.8.3 使用 RuntimeException 发出违反合同的信号

只要选择履行合同，就需要一种发出违反合同信号的干净利落的、优雅的方式。所以，应避免使用 fail 消息。但在对应 fail 方法的内部需要什么代码，才能优雅地失败？一种选择是退出程序：在 Java 中，可以使用 System.exit(-1) 来退出程序。但是，这个技术会删除如何到达失败点的大多数信息——它只说明最后执行的那个方法，没有说明哪个方法调用了它。

除了已有的明确性之外，Java 方法还可以抛出异常，向调用方法发出信号，说明当前方法不能继续执行下去。Java 有一种特殊类型的异常，叫做 RuntimeException，它可以用在这种情形。RuntimeException 用来说明方法因为实现代码或调用方法中的一个错误而不能继续执行下去。调用方法不会捕获 RuntimeException，而是让运行时系统给它自己的调用方法重新抛出这个异常，在上一级的调用方法中，再为更上一级的调用方法重新抛出这个异常，最终，异常会到达进程的最外层，即 main 方法，它会暂停这个抛出过程中的所有方法。

在 main 抛出 RuntimeException 时，程序会被运行时系统停止，用户(或测试人员、管理员)可以通过失败方法查看跟踪情况，其跟踪路线是：

```
RuntimeException in bar() "Preconditions violated"
main() called foo() on Client
foo() called bar on Supplier
```

这就是调试所需要的信息。

给违反合同发出信号的 Java 样式如下面的例子所示：

```

public class Supplier {
    ...
    // Precondition: o != null
    public int bar(Object o) {
        if (o==null) {
            throw new RuntimeException("Preconditions violated");
        }
        ...
    }
    ...
}

```

(如果要为用户提供更多的细节, 可以抛出 `RuntimeException` 的子类, 例如 `IllegalArgumentException`。)

在较大的程序中运行 Java 代码时, 例如在 Web 服务器中运行服务小程序, 或者在 GUI 上运行业务组件, 就不应让 `RuntimeException` 对象(或其他类型的异常)到达 `main` 方法, 否则整个程序都会停止。程序边界上的一般代码可以捕获未处理的异常, 并向管理员或用户报告异常。例如, Web 服务器可以在系统日志文件中添加错误报告, 再给用户显示错误消息, 例如“您的请求不能完成, 因为...”加上一些建议。

## 12.8.4 外部系统

在自己的进程中运行代码时, 可以安全地禁用动态检查。相信自己的代码; 相信同事的代码; 相信模式、库和框架中的代码。但是, 仍需要构建应用程序防火墙, 防止代码被外界破坏。应用程序防火墙代码从来不会关闭, 甚至在测试后也不会关闭。外部系统可以分类为客户和提供者, 下面看看如何处理它们。

### 1. 外部客户

来自外部客户的请求旨在业务对象, 所以必须履行业务对象的合同。于是, 应用程序防火墙应以业务对象的名义检查合同的客户部分。但是, 合同的内容最好封装在业务对象内部(因为这是合同内容应在的位置)。把不变式和前提条件检查封装在业务对象的消息中, 就可以解决这个两难问题。

例如, 考虑下面的类:

```

//Invariant: i1
public class Foo {

    public boolean invariantOK() {
        ...//Return true if i1 is okey
    }

    public boolean okForBar(s, f) {
        ...//Return true if s, f satisfy p1
    }

    //Precondition: p1
    public void bar(string s, float f) {
        if(!invariantOK()) {
            throw new RuntimeException("Invariants violated");
        }
    }
}

```

```

        if(!okForBar(s, f)) {
            throw new RuntimeException("Preconditions violated");
        }
        ...
    }
}

```

应用程序防火墙中的代码可以提取出 bar 的参数值(例如从用户界面或服务小程序中), 使用 Foo 上的服务消息测试调用的 bar 是否有效, 如下所示:

```

//Application firewall code
if (aFoo. invariantOK() && (aFoo. okForBar(s, f)) {
    aFoo.bar(s, f);
}
else {
    ...//Signal error to client
}

```

还可以提供每个 Foo 消息的检查版本和非检查版本, 例如 bar 和 barWithChecks。这会把应用程序防火墙代码移到 Foo 中, 作为客户的一个服务。这个方法的缺点是, 即使消息的检查版本从来都不会使用, 也要提供它们——只有对象从应用程序防火墙中调用时, 才会使用它们。

另一个方法是, 把消息的检查版本放在一个单独的类 FooWithChecks 中。这个方法比应用程序防火墙中的代码好一些, 但要提供可能从来不使用的整个类。

## 2. 外部提供者

从进程中跳出, 以访问外部提供者时, 必须检查外部提供者的状态和返回的信息。这里也可以使用应用程序防火墙的理念。但是, 如果应用程序防火墙检测到一个错误, 该怎么办? 最初, 应用程序防火墙是由某个客户代码调用的, 当应用程序防火墙检测到一个错误时, 就必须告诉客户代码有一个问题。为了避免出现结果代码或错误变量, 最好抛出一个异常。但是, 与“违反合同”的情形不同, 不能让客户代码忽略这类异常——即使在没有错误的代码中, 外部问题也是可以预见的。所以, 必须迫使客户处理这种情况。

为此, Java 提供了第二类异常, 由 Exception 类表示, 编译器会迫使客户代码处理该异常。在 Java 中, Exception 对象称为已检查的异常, 因为编译器要求提供检查代码, 而 RuntimeException 对象称为未检查的异常, 因为不需要(或者不应该)提供检查代码。

为了向进程外部的提供者通知出现了错误, 可以执行下面的代码:

```

Public class ExternalResourceUser {
    public void useResource()
        throw Exception // Checked exceptions must be listed
    {
        ... // Use the external resource
        ... // Detect a problem
        throw new Exception ("External resource failure");
    }
}

```

有了 RuntimeException, 就可以使用 Exception 的子类, 例如更具体的 IOException。

## 12.8.5 启用和禁用动态检查

动态检查对系统性能的含义是：检查每个方法中的前提条件、后置条件和不变式，会使代码的运行速度比不检查慢十倍。我们希望生成的软件在部署时不违反任何断言——如果没有违反断言，就不需要进行检查。另一方面，在开发过程中，希望常常违反断言。为了满足开发和部署的不同需求，需要一种开关动态检查的方式。

Eiffel 实现方式在是使用运行时开关还是使用编译时开关上是不同的。但选择关闭什么，打开什么是很简单的。例如，在开发过程中，关闭库代码的后置条件(因为库已经调试好了)。使用非 Eiffel 语言时，必须多做一些工作。

有两种方法可以启用和禁用动态检查。第一种方法是使用运行时开关(使用命令行参数或环境变量)：如果开关是打开的，运行时系统就会进行检查；如果开关是关闭的，运行时系统就不进行检查。第二种方法是使用编译时开关，建立系统的两个版本，一个带有检查代码，一个没有检查代码。在这两种情况下，都有更多的选择：例如，可以禁用库代码中的所有后置条件检查或所有的检查。

原则上，Java 实现代码可以使用运行时开关和编译时开关，但实际上，情况会复杂一些。Java 程序员只能通过仔细的编程，使用编译时开关，运行时开关(断言机制)是可以使用的，但不应用于检查合同的所有部分。

### 1. 实现编译时开关

使用解释清晰的技巧就可以激活编译时开关。首先，引入一个类，把开关的值存储为一个类常量(有固定值的类字段)。下面是这个类(其中 final 表示“常量”):

```
public class ContractSwitch {  
    public static final DO_CHECKS = true;  
    //it's okay to have a public field if it's final  
}
```

现在，可以根据开关的值，把动态检查代码封装在 if 语句中，如下所示：

```
public class Supplier {  
    ...  
    // Preconditions: o != null  
    public int bar(Object o) {  
        if (ContractSwitch. DO_CHECKS) {  
            if (o==null) {  
                throw new RuntimeException("Preconditions violated");  
            }  
        }  
        ...  
    }  
}
```

当编译器进入 If (ContractSwitch. DO\_CHECKS)语句时，会发现类常量是 true，所以 if 语句中的内容总是会执行：编译前提条件的检查，并包含在当前的构建体中。但是，外层的 if 语句不需要代码，因为它在执行时是多余的。其结果是当前构建体包含合同检查代码，但不包含检查开关的代码。

为了构建禁用检查的系统版本，要找出 ContractSwitch 类的定义，把 DO\_CHECKS 设置为

false，并重新编译系统。现在，编译器遇到 If (ContractSwitch. DO\_CHECKS)语句时，就会认为，由于常量是 false，所以外层的 if 语句总是失败，无法保证执行其内部的代码。结果是构建出来的系统不包含检查代码和开关检查代码。

这个编译时开关技术是 Java 语言规范的一部分，所以它可用于所有的 Java 编译器。稍微思考一下，就能看出如何使该技术更有选择性(添加更多的类常量)。Java 编译器按照包来编译类，所以必须把 ContractSwitch 类添加到每个包中。

## 2. 使用断言机制

Java 的断言机制允许程序员插入像 assert o != null 这样的语句。于是，可以使用运行时开关控制断言的开和关，例如断言的失败是否应使程序自动停止。尽管使用的是运行时开关，但是 Java 实现方式也可能在开关关闭时，删除所有与断言相关的进程和性能开销。

传统上，编程语言中的断言机制用于履行合同，在任意点插入错误检查代码。根据 Java 断言机制的文档说明，它不应用于履行合同的客户部分。其原因是，一些检查非常重要，不能禁用(由于安全原因)。更好的方法是说明，一些驻留在应用程序防火墙中的检查是强制的，断言机制不应用于这些检查。这个方法与本章前面的内容兼容。

尽管断言机制为检查合同提供了开关式机制的所有特性，也不能把它用于检查合同。阅读说明文档的其他程序员不希望以这种方式使用它。另外，如果检查在运行期间失败时断言是打开的，那么抛出的异常类型就是 Error，而不是 RuntimeException。Error 是不可恢复的问题，程序员不应试图处理它，例如运行时系统中的错误或内存耗尽。在断言机制抛出 Error 时，并不符合应用程序防火墙用 RuntimeException 对象发出信号的规则。

将来可能在 Java 中添加自动履行合同的功能，但现在最好使用编译时开关技术。

## 12.9 小结

本章的主要内容如下：

- 规范，完整、清晰地描述了软件需要的行为。正式规范是科学的、严格的，使用特殊的语言；非正式规范是注重实效的、部分的，可以用自然语言表达，或者使用 Eiffel 等语言的语法来表达。
- 规范如何以面向对象的方式来编写。面向对象的规范允许判定类的前提条件、后置条件和不变式。
- 按合同设计，它合并了形式方法和面向对象编程，易于生成强健、一流、高效的代码。客户对象和提供者对象之间建立的合同给双方规定了义务。
- 如何在 Java 中使用规范和按合同设计，尤其是使用注释和 Exception 对象。还介绍了 Java 的断言机制，但它不是一种潜在的解决方案。

## 12.10 课外阅读

OCL 的帮助文件可参阅[Clark and Warmer 02]和 OCL 规范本身[OMG 03b]。

按合同设计的描述，可参阅其作者的[Meyer 97]。Bertrand Meyer 还介绍了面向对象编程的

各个方面。

在 Java 中做什么和不做什么的讨论，包括异常的正确用法，可参阅[Bloch 01]。

## 12.11 复习题

1. 什么是“类的不变式”？(单选题)
  - (a) 类的源代码是版本化的，因此不能修改
  - (b) 类的对象有常量字段。
  - (c) 类的实例总是 true 的条件。
2. 术语“按敬畏程度设计”的含义是什么？(单选题)
  - (a) 设计会引起惊慌。
  - (b) 不知道何时可以信任代码。
  - (c) 因为时间紧迫，过快地设计系统
3. 什么是“按合同设计”？(单选题)
  - (a) 设计代码就好像在发送消息的对象和接收消息的对象之间有合同一样。
  - (b) 通过增加错误检查的代码量，强制履行每对对象之间的合同
  - (c) 用防火墙保护使用合同的软件
  - (d) 在合同的约束下设计软件系统

## 12.12 复习题答案

1. “类的不变式”是 (c) 类的实例总是 true 的条件。
2. 术语“按敬畏程度设计”表示 (b) 不知道何时可以信任代码。
3. “按合同设计”表示 (a) 设计代码就好像在发送消息的对象和接收消息的对象之间有合同一样。

# 第 13 章 不间断的测试

本章将介绍测试的各方面内容。和软件开发的其他领域一样，测试也常把工程训练用作指南。

## 学习目标：

- 理解测试中使用的许多术语
- 了解测试策略
- 学习一个测试驱动的开发例子

## 13.1 引言

软件的开发是一个复杂的事务。无论多么努力，只完成需求、分析、设计、规范和实现阶段，都不能去除所有的错误。但是，利用正确的实践，就可以确保不发生大多数严重的错误。另外，还需要一个单独的测试阶段，其目的是在发布前去除所有遗留的错误。

现在，在开发结束前添加一个测试阶段往往效率很低：必须在开发过程中测试代码和其他制品。应确保许多不同的人参与到测试中来：开发人员、同行(不直接参与当前项目的人)；顾客、项目经理、测试人员(主要负责测试阶段的人)。基本上，测试活动有三个阶段：开发过程中的测试(由开发人员进行)、测试阶段中的测试(由专业的测试人员进行)和发布之后的测试(在软件生存期内，所有的用户和开发人员将收集反馈，更正发现的错误)。

本书主要由开发小组的成员使用，所以下面将介绍测试驱动的开发方式，这是不间断测试的一种形式，在这种方式下，开发人员将在开发过程中测试其代码。这种方法的优点是：

- 提高软件的质量(测试人员越多就越好)
- 降低测试阶段的成本
- 给程序员展示他们正取得进展(而不仅仅生成代码行)
- 在测试阶段，减少与程序员相关的错误量
- 有助于程序员因样式或性能原因而重新组织代码，无需破坏已编写好的代码

## 13.2 测试术语

- 测试(test)：检查软件的某个方面是否正确，例如“测试是否能登录”、“测试采购子系统”或“测试所用的内存在最大负载下是否超过了 500 MB”。
- 错误(error)：程序员犯的过错(例如误解)。错误常常会产生一个或多个故障(这与使用“错误”表示在系统运行过程中发生的不良条件不同——参见“失败”的定义)。

- 故障(fault): 是不正确的代码。与汽车引擎中的一个故障使汽车不能平稳驾驶的情况类似，软件故障会阻止系统正确运转。例如，假定数组从1开始索引，编写了一个方法，如果数组实际上是从0开始索引的，该方法就有一个故障。使用术语“故障(fault)”或“过失(defect)”比使用“缺陷(bug)”更好，“缺陷”是不太正式的术语。当开发人员遗漏了某个东西(例如顾客需求或处理某个情况的代码)时，就会发生“省略故障(fault of commission)”。当把需求提交给代码、设计或规范时出问题(例如认为三角形的面积是底×高，或者试图从已关闭的文件中读取)，就会发生“提交故障”。
- 失败(failure): 是一个系统误操作，通常是由一个或多个故障造成的。例如，“有人因缓存超限故障而闯入系统”。
- 更正(fix): 修复故障。
- 验证(verification): 根据需求文档(系统用例)检查软件是否正确。
- 有效性验证(validation): 检查软件是否是顾客需要的，例如它是否以顾客和最终用户能接受的方式执行顾客需要的函数。
- 规范: 描述了代码的接口，安排什么可以进入，什么必须出来。因此，规范测试是黑盒子测试和用例测试的总称。

### 13.2.1 黑盒子测试

在黑盒子测试中，进行测试的内容(系统、子系统、类或方法)都当作不可测知的对象(如图13-1所示)。访问盒子的惟一方式是通过其发布的接口——消息、参数等。因此，测试人员要评定黑盒子所做的是否正确，必须捕获、分析消息的回应，检查所有的外部副作用(例如在数据库中创建项)；检查响应请求或执行命令所花的时间。接口的每次使用都会产生内部副作用(因为盒子会从一个状态变为另一个状态)，但测试人员只能看到这些内部副作用对接口后续使用情形的影响：即使可以获得盒子内部的细节，也会因黑盒子测试的目的而忽略它们。

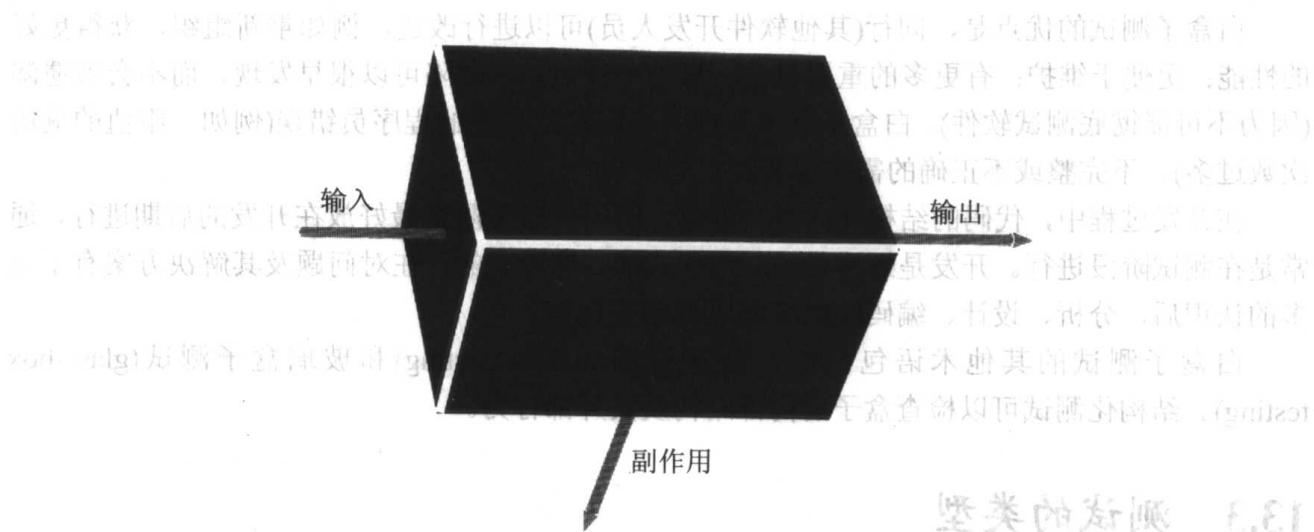


图 13-1 黑盒子测试(将待测对象看作一个黑盒子)

黑盒子测试产生自“我们不关心代码如何获得其结果，只要代码获得这些结果即可”的原则。这非常符合“在生成软件之前写下的系统需求是软件开发最重要的方面”这一观点：在需

求阶段之外，分析员、设计人员和实现人员可以自由发挥，以他们认为合适的方式满足需求。黑盒子测试使测试人员与软件的内部复杂性绝缘。

黑盒子测试的缺点是，由于忽略了内部结构和代码行，可能会错失潜在的改进机会——毕竟，软件可能是正确、充分的，但质量未必高。另一个缺点是，测试阶段未捕获的故障要在很久以后才能发觉。

黑盒子测试的其他术语包括行为测试和功能测试。行为测试从外部的观点来看，强调测试集中于盒子的行为方式，而不是集中于行为的完成方式(这需要了解盒子的内部情况)。

### 13.2.2 白盒子测试

在白盒子测试中，可以查看盒子的内部，检查结构和细节，直至检查代码行(如图13-2所示)。

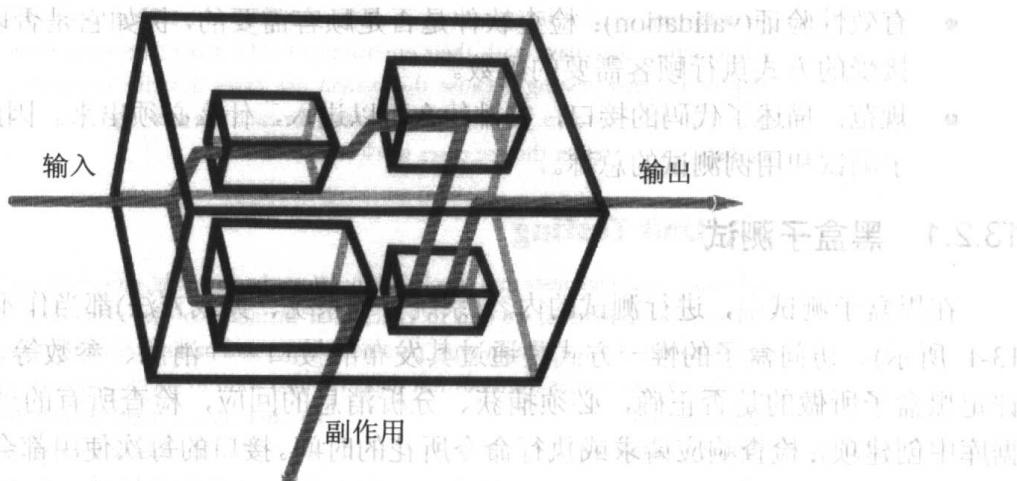


图 13-2 白盒子测试

白盒子测试的优点是，同行(其他软件开发人员)可以进行改进，例如重新组织，获得更好的性能；更便于维护；有更多的重用机会。另一个优点是，缺陷可以很早发现，而不会被遗漏(因为不可能彻底测试软件)。白盒子测试发现的缺陷包括典型的程序员错误(例如一组值的遍历次数过多)、不完整或不正确的请求说明。

在开发过程中，代码的结构和内容会变化，所以白盒子测试最好放在开发的后期进行，通常是在测试阶段进行。开发是螺旋式的、迭代式的、递增式的，在对问题及其解决方案有了更多的认识后，分析、设计、编码和系统需求都需要调整。

白盒子测试的其他术语包括结构化测试(structural testing)和玻璃盒子测试(glass-box testing)。结构化测试可以检查盒子的内部结构及其外部行为。

## 13.3 测试的类型

大多数测试专家都把测试过程分为不同的层次：

- 单元测试：是最低的层次。单元是单个代码块(传统上是一个函数或过程，在面向对象的软件开发中，则是一个类)。

- 完整性测试：检查独立的代码块如何协同工作。
- 系统测试：检查所有协同工作的子系统的操作，以及它们与环境的交互情况。系统测试最重要的形式是用例测试，因为系统用例完整地描述了系统必须做的工作。为每个用例设计测试，就可以提供各种测试，检查用例是否能同时工作。

### 13.3.1 单元测试

给每个类设计测试，检查它是否正常、高效、从容地工作。在设计单元测试时，目的是为了检查类的实例是否有正确的消息签名(名称、返回类型、参数类型和参数名)，它们是否能保证从测试数据中获得正确的回应。还要检查每个对象是否以有效的状态开始工作，在使用时是否从一个有效的状态变为另一个有效的状态(对象的状态在其属性值中存储)。对于黑盒子测试，只能使用消息(例如 `getCustomerId`)来验证状态；如果进行白盒子测试，就可以检查字段，在方法内部查找故障、不合适的样式或检查质量情况。如果类使用了类字段和类方法，就必须检查类本身及其对象。

在单元测试中，有一个进退两难的局面：如何在测试单元时不测试其合作者？在面向对象的情况下，可以确定，已封装的对象只是单元的一部分，因此黑盒子测试不需要对此采取特定的措施(对于白盒子测试，可以简单地忽略合作者)。但是，如果要进行纯粹的单元测试，可以把合作者替代为模拟的对象，它提供测试类所需的最少行为。也可能需要使用模拟对象(有时称为 *stubs*)，作为代码的替身，在其他情况下不能使用该替身(因为它没有实现，或者需要一些当前不想涉及的配置，如数据库)。

### 13.3.2 完整性测试

在完成的系统中，每个对象都与其他对象协作，所以完整性测试是单元测试后的一个必要步骤(尤其是因为不同的类要把由不同的开发人员编写)。例如，可以测试业务层上的对象协同工作的情况。

完整性测试一般在一个或多个层次上进行：

- 构成一个逻辑单元的合作类
- 层
- 包
- 库
- 框架
- 系统或子系统

子系统测试是完整性测试的一种形式，测试的类集构成了一个完整的子系统(由系统设计人员定义)。

### 13.3.3 Alpha 测试

在 alpha 测试中，系统是正在进行的一个工作。其目标是确保开发出看起来正确、对顾客有用的东西来。例如，在系统设计的框架内部，可以实现某个核心功能，接着让同行和仔细挑选的顾客试用。如果反馈比较好，就说明我们走的路是正确的(并可以收集有效的改进信息)；如果反馈不好，就必须回到绘图板上，否则，最好的情况是所发布的系统没有用，最遭的情况是顾客拒绝为该系统付费。

alpha 测试常常在原型上进行，原型就是很快生产出来、且很便宜的系统版本。如果 alpha 测试失败，我们就不会浪费太多的时间或资金。一旦完成 alpha 测试，就应在正式开发的更完善的解决方案中抛弃原型代码。尽管很难抛弃原型代码，但必须记住，原型意味着开发速度很快，只用于试验——其设计或编码很可能质量不高。抛弃原型代码时，应保留从问题及其解决方案中吸取的所有营养(例如草拟用例和 UML 制品)。

### 13.3.4 beta 测试

在开发和内部测试全部完成后，系统的整个递增版本完成时，进行 beta 测试。它要用户在真实的环境下试用软件，这么做的原因是，要在最后发布之前找出最明显的故障并更正它们。无论开发过程如何完美，都有开发人员或特定目的的测试小组没有发现的故障。正常情况下，故障没有发现是因为不可能进行彻底的测试(或至少是不可行的)，也就是说，测试人员必须使用各种技巧和判断力预测出软件的使用方式。beta 测试用于填补预期使用和实际使用的缺口。

beta 测试应由最终用户进行，而不是由测试小组的成员进行。根据软件的本质，beta 测试人员可以是同事、真正的顾客或上述两者。beta 测试和生成反馈都是很令人沮丧的任务，所以一般需要提供某种激励。对于同事，可以许诺“如果你对我的软件进行 beta 测试，我就对你的软件进行 beta 测试”，或者简单地说明“这对公司有好处”。对于真正的顾客，可以打折销售发布版本。对于只有一个最终客户的系统，必须使他们相信，beta 测试是开发过程的一个正常组成部分，并不意味着软件质量低下——beta 版本可以由客户的几个雇员来测试，然后在客户公司中进行更广泛的发布。

### 13.3.5 用例测试

在用例测试中，要依次对某个用例设计测试，确认系统满足用例的要求。为此，可以为每个用例考虑许多可能的场景，每个场景都是一个合法的事件序列——因此有了术语“场景测试”(用例看起来像是一个事件序列，但它常常描述了其他的路径和副本——所以，每个场景都是路径和副本的特定选择)。用例测试是黑盒子系统测试的一种形式。

除了通常的事件序列，还可以测试用例中指定的异常序列，即事情没有按照计划进行时系统应怎么办。例如，租赁汽车通常是成功的，但有时可能要指定顾客欠账时该怎么办。在多用户系统中，并行执行用例测试也很重要，以模拟部署后会发生什么。

用例测试在由用例驱动的开发(如本书前面所述)中尤其重要。我们的开发是根据用例的优先级，以递增的方式进行的，所以用例测试应是递增地运行。

### 13.3.6 组件测试

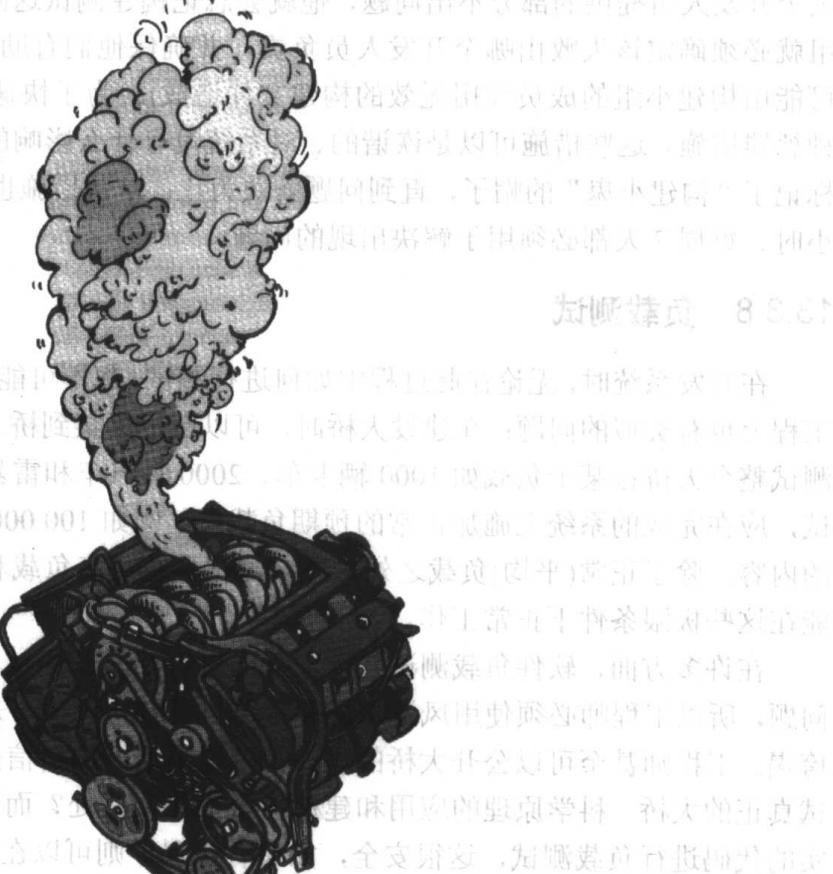
按照一般的术语，组件是软件的一个逻辑独立块，有一个定义好的接口。与硬件组件类似，软件组件独立于其周围的环境，并隐藏其内部的内容。如果要使用组件，就只能使用其接口，不能“打开组件”——这样，我们就不必考虑组件内部的复杂性，但仍能连接组件，构建更大的组件。

下面这些东西都可以看做一个组件：对象(也许包含其他对象)、协作对象组、子系统、整个系统。把系统看做一个组件，就可以把不同位置的系统链接在一起。例如，iCoot 系统运行在一个分支内部，它每周通过业务对业务链接给总公司提交报告——在这种情况下，系统是完全独立的，各个系统都是独立的组件。

所以，组件测试就是在构成组件的各个单元上运行测试。通常，可以选择是进行黑盒子测试、白盒子测试，还是进行这两个测试。

### 13.3.7 构建测试

构建测试用于确保软件可以成功构建。构建软件就是把系统的所有部分集中在一起，编译、链接和部署它们及其需要的资源(数据库和不是源代码的文件)。构建测试有时称为烟雾测试。在开发新的引擎时，一组机械工程师要频繁地启动和加速引擎；如果引擎冒烟，就说明该引擎有问题(如图 13-3 所示)，工程师就会停止引擎，诊断问题并修复它，然后进行下一次烟雾测试，检查修复情况。



失败的烟雾测试

对于除试用系统之外的其他系统，可以使用构建工具来确保系统的构建是自动的、一致的和完整的。例如，Ant 来自开放源代码的 Apache 项目([ant.apache.org](http://ant.apache.org))，它把开发人员编写的 XML 文件作为输入，这些文件描述了如何构建完整系统的所有部分。XML 文件还包含各个部分之间的依赖信息，这样 Ant 工具就可以按正确的顺序构建它们。例如，可以指定某组源文件必须在链接到可执行文件之前编译。

一旦系统构建出来，且没有故障(例如语法错误造成的、由编译器报告的故障)，构建测试的第二阶段就启动系统，进入评定场景——如果这些场景可以完成，且没有故障(例如资源没有找到或程序抛出了异常)，测试就是成功的。在成功的构建测试之后，前面实现的代码和资源

就能协同工作，在部署完成的系统时也就不会有大问题。如果场景失败，就说明在继续部署之前，必须处理故障，因为不可能生成未构建、部署或执行其基本功能的系统。

最好频繁地进行构建测试，因此有了术语“每夜构建(nightly build)”。这似乎有点过分，但对于有许多部分的大型系统来说有好处，因为各个部分常常是由不同的人开发的，这些人在不同的地方，不可能每天联络。定期的构建检查也很重要，因为它可以避免把问题拖延下去，变成更大的问题(因为每个人的工作会随时间而变化)。甚至对于只有一个开发人员的小系统，定期构建测试也有上述优点。开发人员一天几次地构建他们的代码是很自然的——成功的构建和几个场景说明，他们正在取得进展。

对于中大型项目，通常有一个专业的构建小组，负责编写构建文件、从开发人员处收集各个部分，进行构建测试。这样，一般的开发人员就不必进行许多管理活动了——实际上，只要某个开发人员提供的部分不出问题，他就会忘记构建测试这回事。如果构建测试失败，构建小组就必须确定该失败由哪个开发人员负责，并确保他们有助于修复相关的故障(当然，失败也可能由构建小组的成员使用无效的构建文件造成)。为了快速解决问题，开发小组可以采取某种惩罚措施。这些措施可以是诙谐的、对系统没有什么影响的措施，例如让开发人员戴上一顶标记了“构建小鬼”的帽子，直到问题解决为止；这些措施也可能相当严厉，例如要求每天 24 小时、每周 7 天都必须用于解决出现的问题。

### 13.3.8 负载测试

在开发系统时，无论在此过程中如何进行测试，都不可能测试出系统在正常负载下的行为。工程上也有类似的问题：在建设大桥时，可以在把梁架到桥上之前测试各个梁的强度，但不能测试整个大桥在某个负载如 1000 辆卡车、2000 辆汽车和雷暴雨下的情况。对于软件的负载测试，应在完成的系统上施加正常的预期负载——例如 100 000 个用户试图同时查看其银行账户的内容。除了正常(平均)负载之外，还要测试系统在高负载和最大负载下的情况，检查它是否能在这些极限条件下正常工作。

在许多方面，软件负载测试易于工程师进行的负载测试。如果大桥垮了，就会出现严重的问题，所以工程师必须使用风道测试和计算机建模等技术，来确保大桥在极限条件下不会出现垮塌。工程师甚至可以公开大桥的设计，以说明他们的自信是有理论根据的——如果不实际测试真正的大桥，科学原理的应用和建模能带来什么好处？而对于软件，可以在部署条件下对真实的代码进行负载测试，这很安全，如果有错误，则可以在运行之前修改代码。

软件负载测试的另一个优点是，如果系统在运行时出现故障，就可以重新启动系统(这就是软件不如工程产品可靠的原因)。在对安全性要求很高的系统中，例如自动驾驶的飞机，不能重启系统，所以应使用不同的规则：如果可以证明自动驾驶的飞机比人操纵的飞机更可靠，就有利润可赚。

#### 1. 渗透测试

软件中的一些故障只有在软件连续运转了很长时间后才能显现出来。例如，系统不删除运行过程中生成的临时文件，最终就会用尽磁盘空间，但只有在一定的负载下运行很长时间，才会用尽磁盘空间。渗透测试(soak testing)表示在高负载下连续运行软件，确保它不会耗尽自己的资源，或者产生其他累积性问题。

## 2. 应力测试

工程师常常测试结构的断点，这有两个原因：第一，不管理论上有什么限制，都要找出实际的断点是什么；第二，确定结构失败时会发生什么情况，找出结构会在哪一部位失败。软件的应力测试(stress testing)就是使系统过载，找出系统不再能运转的点。为此，要找出其局限之处，看看系统在超过该断点后会如何运作。在进行应力测试时，要剥夺系统需要的资源，看看会发生什么。例如，剥夺系统运转所需要的一些计算能力、内存或磁盘空间。

一般说来，对于软件，我们的目标是“优雅地失败(graceful failure)”: 系统尽可能把相关的损失降到最低，且处于运转状态。例如，如果建立一个新闻网站，它一般同时处理一百万个请求，则一个特别的新闻项会使1亿个用户同时访问系统。在这种情况下，整个网站不是陷入停顿，而是正常处理一千万个请求，并通知另外9千万个用户，该网站太忙，用户必须以后再试。在实时监视的帮助下，我们知道出了这个问题，并可以确定是否增加网站的容量。

应力测试的另一个术语是“负测试(negative testing)”，“负”的意思是“悲观”，因为软件超出了其操作范围，进入了未设计的领域。

### 13.3.9 安装测试

安装测试就是查看软件在其操作环境下的运转情况。测试的主要工作是在特别建立的测试环境(试验台)下进行的，所以没有在实时环境下检查软件，该软件就没有进行完全的测试。可移植的、在任意多个环境(多个操作系统或设备)下运行的软件尤其如此。安装测试的其他术语包括平台测试和环境测试。

### 13.3.10 接受测试

最后，软件的成功依赖于它是否满足用户的需要。即使为了公司的利益，管理层提交了软件，如果不能得到最终用户的接受，用户就不会正确使用它，也就达不到最初的目的。如果软件要卖给第三方，而且它对使用它的人来说是有用的，该软件就能卖出去。自然，我们希望软件是优秀的，而不仅仅是将就着能使。

接受测试允许最终用户在软件发布之前试用软件，看看是否能接受它。在接受测试的过程中，可以要求用户执行特定的任务，或者要求用户试着完成他们通常用其他方式执行的任务。在进行测试时，可以监视用户的使用效果，收集反馈。监视工作包括给用户录像，推测他们看得见和听得见的反应，记录他们的键盘操作和鼠标移动(一般的规则是“越少就是越多”——完成工作所需要的交互越少就越好)。监视和收集反馈是以用户为主，而接受测试的其他部分在更高的层次上进行，例如，与经理讨论生产率，或让心理分析学者解释视频录像。

由于在测试阶段的后期执行完全接受测试，因此修改软件以去除负面结果的操作就比较昂贵。所以，我们可以正常地把一些非关键的修改推迟到以后的递增版本中。螺旋式和迭代式的开发帮助我们避免了很多严重问题。

### 13.3.11 褪退测试

只要通过添加、删除或编辑源代码，修改了软件，就需要确保软件的新版本至少与原版本一样好。修改代码会引入错误。引入新行为时，因新行为代码包含错误而不得不修复的情况并不少见。

衰退测试就是确保修改的内容不会使代码衰退的过程，即代码不会比以前更糟糕。因此，通常要在每次修改后，对软件进行一整套测试——软件的新版本应像旧版本一样通过这些测试(理想情况下，除了正确通过这些测试之外，新版本还应高效、快速、安全、一流等)。

### 13.3.12 说明文档测试

说明文档是手册和培训材料的统称。说明文档对系统的成功来说非常重要：如果没有人能使用软件或维护它，就无法开发它。文档说明应专门针对系统管理员和最终用户，即使这意味着必须编写两套说明文档，也是如此。

说明文档测试应利用其他作者、系统管理员、最终用户和指导教师的输入，来检查说明文档是否正确、有效。

### 13.3.13 安全测试

安全测试就是确保系统是否安全的过程，但是，安全意味着什么？如第8章所述，安全有如下方面：

- 私密性
- 身份验证
- 不能反驳
- 完整性
- 安全性

系统必须受到保护，不被黑客、怀有恶意的第三方或犯罪团伙蓄意破坏。因此，必须保证无效或未授权的信息或软件不能进出系统。有道德的黑客是这方面的专家：他们得到许可后，就会尝试在试验台上攻击系统，然后在软件发布前修复他们发现的漏洞。

大多数安全测试都执行为系统测试的一部分，通常在测试阶段进行。这是因为每个系统在设计时都应考虑安全性，而安全测试是为了确保黑客(和小鬼)不能超出系统的边界。

### 13.3.14 衡量标准

衡量标准是用于评定软件的质量或效率的度量方法。一般的衡量标准包括：

- 代码覆盖面：有一个广泛接受的原则：代码只应存在于使用它的系统中，否则开发它就是浪费时间，会增加不必要的维护成本和系统规模。对于重用的代码，有一个矛盾：在实现一个类时，希望添加额外的、将来可能有用的公共方法，这样，代码就可以在其他系统中重用；但是，所添加的额外方法当前系统并不使用。为了协调这一矛盾，库和框架的代码覆盖率应较低，而专用于当前系统的代码覆盖率可以较高。
- KLOC：这是一个旧的衡量标准，是千行代码的缩写。KLOC 用于衡量开发人员的生产率：程序员编写的代码行越多，他们的生产率就越高。但是，快速生成大量代码的程序员可能生成了不需要的代码，而有条理的、力求准确的程序员可能生成的代码较少，但质量很高。最佳的面向对象开发人员应尽可能避免编写自己的代码，而是尽量重用库、框架和其他应用程序中的代码，所以 KLOC 已经是一个多余的衡量标准。KLOC 有时仍用于表示应用程序或系统的相对规模，例如 XYZ 系统有 15000 KLOC。

- 事务处理时间：这里使用的“事务处理”是处理入站的请求——例如事务处理“买一本书”。数据库事务“给 CUSTOMER 表添加一个记录”。可以度量每秒或平均时间内处理的事务数，作为推测系统性能的一种方式。
- 继承层次结构的深度：有人认为，有许多层的继承层次结构太深了，还有人认为，这没有硬性的规则，只取决于要完成的任务。公平地说，大多数继承层次结构，即使是复杂的结构，都只有 6 层。因此，如果继承层次结构有 50 层，就需要进一步探讨。
- 继承层次结构的宽度：这个衡量标准与层次结构中的类数相关。与深度一样，这是一个灰色区域。但是，如果一个层次结构只有三层，但每一层有 50 个类，则该结构需要重新组织。
- 方法的规模：多于 50 行的方法应拆分为几个更小的方法；每个小方法更容易编写和维护。
- 耦合度：如果组件之间有许多链接，来回传送许多消息，这些组件就称为紧密耦合。这基本上是一个维护问题，因为修改一个组件的接口，会对其他组件有很大的影响，反之亦然。应使用大量松散耦合的对象，或者在客户-服务器模式下编写系统。
- 内聚度：内聚(cohesive)对象有一组一致的责任。例如，表示菜单中比萨饼的对象和某人预定的比萨饼是弱内聚系统的一些部分只把比萨饼用作菜单中的一项，而其他部分把它与一个特定的顾客结合使用。弱内聚的另一个例子是可以用作集合(无序、没有重复)或用作列表(有序、有重复)。强内聚支持简化、易于维护和重用。

大多数衡量标准都是主观的，所以必须仔细解释它们。最重要的衡量标准是“软件达到了需求中陈述的目标吗？”

## 13.4 测试的自动化

现在我们知道，在全面测试软件的过程中，有许多工作要做。而且，测试是必需的，以确保代码的质量、正确性和安全性。但是，如果测试太费时间，开发人员就不愿意进行测试，尤其是那些非测试小组的成员，就更不愿意了。

因此，测试过程的自动化程度越高越好。理想情况是要进行的测试与系统软件分开，或者点击按钮即可开始测试。测试自动化软件包括：

- 负载工具：模拟系统预期可以承受的负载。这些工具可以模拟多个客户同时访问，在客户-服务器系统上给服务器添加负载；模拟人的动作，给用户界面添加负载；模拟外部的访问，给过程添加负载；模拟其他同行的访问，给分布式系统中的同事添加负载。
- 测试框架：包括可重用的代码和技术，该框架负责进行测试，允许测试开发人员只关注测试的编写。
- 性能监视工具：可以监视所有的事务，包括单个进程使用的虚拟内存，机器之间的网络通信。
- 衡量标准收集工具：从源代码或正在运行的程序中收集衡量标准。例如，可以使用这种工具分析源代码，查找常见错误，检查代码的质量(其方式与字处理程序中的样式检查器一样)。
- 规范检查工具：使用软件接口的非正式规范(由测试开发人员用某种具备特定功能、可执行的语言编写)，尝试测试底层软件是否遵循该规范。例如，如果指定某个消息总是

返回小于 10 的值，该工具就可以为许多随机生成的参数验证该消息是否返回小于 10 的值。

- 断言检查工具：检查开发人员插入的断言代码。断言检查要求在运行期间执行表达式，确保它们不会失败(结果等于 false)。因此，每次运行代码时，断言都可以用于测试代码。这些工具的区别是，一旦配置好，测试人员就可以启动它们，然后坐下来等待结果。

## 13.5 准备测试

从项目经理的角度来看，测试策略的主要组件是：

- 测试计划：需要说明“如何正确达到系统的目地，满足顾客的需求”。草拟的测试计划应在开发开始之前编写，确定开发的所有阶段和所有要生成的制品。测试计划要覆盖从主要问题(例如测试哲学体系和测试技术)到各个测试的具体细节，指定测试名称、描述、过程和预期的结果。
- 试验台：类似于实时环境，为测试而构建的环境。例如，如果实现一个客户—服务器系统，用于 Unix 服务器群(包含业务逻辑)、运行在大型机上的数据库服务器和任意平台上的多个客户，就可以构建一个试验台，其中包括：
  - 10 台客户机，使用各种不同的操作系统
  - 3 个 Unix 服务器
  - 一台大型机
  - 一个基于 TCP/IP 的 LAN试验台允许安全地测试软件，之后在比较敏感的实时环境中开发它。
- 测试工具：有助于进行测试或开发测试。该工具支持测试过程，使测试不会被打乱。(也可以说，该工具有助于把测试与要测试的代码关联起来)。例如，该工具包括一个为运行测试提供代码的软件框架和一个启动它们的工具：测试人员只需进行测试即可。
- 测试案例：这是若干个逻辑单元，由许多测试组成，用于检查软件的某个方面。例如，有一个测试案例基于一个逻辑软件(类、包、子系统)或一个业务逻辑(买书、查看银行陈述、进行查询)。
- 测试套件：相关的测试案例集。可以把测试套件合并到更高级的测试套件中。例如，把“买共享软件”和“卖共享软件”测试案例收集到“贸易”测试套件中；把“贸易”、“银行”和“资产”测试套件合并到“资金管理”测试套件中。
- 测试过程：如何进行每个测试的指令。它一般包含如下步骤：
  1. 建立测试。
  2. 进行测试(比较预期结果和实际结果)。
  3. 报告测试结果。

每个步骤都包含更小的步骤，例如“从菜单栏中选择 File->New”和“输入文件名”。

- 测试数据：为运行测试而合成的数据。在部署系统之前，并没有真正的数据，所以测试数据是对系统最终要处理的信息种类的猜测。测试数据包括文件中的数据、数据库中的数据、一般用户传送给系统的数据。  
一些测试数据是必需的，因为测试软件不能没有它们。例如，除非数据库中有数据，

否则就不能测试用于从数据库中提取数据的组件。正常情况下，测试数据必须是开发人员根据直觉和经验手工编写的，有时还可能从以前的项目中得到测试数据，理想情况是从用户正在使用的系统的旧版本中获得测试数据。

## 13.6 测试策略

软件应实现后再测试，但这是不够的，原因如下：

- 没有考虑其他开发制品，例如用例和 UML 图。
- 开发人员会把错误的查找推迟到测试阶段，那时修复错误会比较昂贵。
- 这很不自然：只要代码能使用，每个程序员就想试试它们(这会让程序员觉得安全，而且取得了进展)。

所以，我们需要一个连续的测试策略：随时测试已生成的所有内容。我们处理的是不同种类的制品，所以策略也必须涉及不同种类的测试(手工测试、用样例代码测试、自动测试等)。还需要一个明确的测试阶段，因为开发人员无论多勇敢、热情多高，都不会试图破坏他们的成果(他们的目标总是确保“结果看起来不错” )。

对于非代码的制品，测试要涉及由下述人员进行的手工检查：

- 开发小组，因为他们是专家，而且热衷于成功。
- 顾客，因为顾客是这方面的专家，如果我们发布的是正确、有效的系统，就必须涉及他们。
- 同行，他们是不直接涉及当前项目的开发人员(所以没有什么制品)。

对于代码制品，测试涉及：

- 开发小组，他们可以证明所取得的进展(生成质量更好的产品)。
- 同行，他们参与了代码的评估，有助于找出错误，重新组织代码。
- 测试小组，负责验证代码的正确性、检查代码的效率。

### 13.6.1 开发过程中的测试

在开发过程中，应手工测试所有的非手工制品。如果所使用的工具调整为特定的方法，就可能具有跟踪功能，有助于检查依赖的制品是否一致。但是，仍有一个手工元素：只有这么一个工具可以进行跟踪，禁用或取消自动跟踪功能可能有许多原因(例如个人喜好、经验、项目规则或在室内采用某种方法)。

对于代码，最好鼓励程序员在进行编码的同时进行测试。测试框架，如 JUnit，有助于这么做：在编写一段系统代码之前，程序员要编写一些样例代码来测试新代码段。因此，当编写完新代码段时，程序员就可以立即验证它们是否成功。JUnit 程序员可以进行单元测试、衰退测试和完整性测试。

程序员还应在代码中添加动态断言检查。这有助于连续测试在规范阶段定义的类不变式、方法的前提条件和后置条件。程序员还可以在方法中添加断言，检查其他常见错误，例如不可能到达的分支。

同行评估(peer review)应定期组织，例如在每个开发螺旋的最后组织。把要评估的所有制品都发布给开发小组的成员和其他同行，再开会进行详细探讨，找出错误和不足之处。其目标

是在继续之前证明或验证制品。同行评估过程的正式程度取决于：公司政策、顾客政策(例如，防护项目比较严格)、项目政策、已完成的开发阶段(找对了开发路子吗？或者这是一个比较成熟的项目吗？)。

在顾客评估中，要求顾客验证制品。顾客不应评估复杂的制品，例如顺序图或代码。同行评估和顾客评估不应合并，否则同行会习惯于顾客对结构的批评，而顾客必须参与具体的技术讨论。

在第一次递增后，所有的递增都应进行定期(每天或每周)构建测试。

### 13.6.2 测试阶段中的测试

在测试阶段，测试小组应在软件发布之前测试和验证软件。这一般包括：

- 重新运行开发人员的测试。
- 根据设计和规范阶段，进行子系统测试(包括层)。
- 主要根据需求阶段开发的系统用例，进行系统测试。
- 由最终用户和系统管理员进行接受测试。
- 如果可能，在各种平台上进行安装测试。
- 说明文档测试，即测试手册和培训材料。
- beta 测试
- 收集衡量标准，包括性能标准和编码样式标准。

测试小组的成员应进行黑盒子测试和白盒子测试。

### 13.6.3 发布后的测试

在发布后，测试仍继续。最终用户或管理员每次启动或与系统的某部分交互时，都要再次测试系统。此时软件中应有一个报告系统，以便将系统实时使用过程中发生的失败报告给项目小组(通过支持热线、电子邮件或网站)。报告系统还可以用于提供反馈，例如改进建议或新特征的请求。

在各个递增版本之间，项目小组应修复所发现的错误。这些修复是否在下一个递增版本之前发布是一个项目管理问题。一般情况下，为了减少工作量，会把许多修复工作合并到一个修复包(fix pack)中，顾客可用它来更新其安装。改进和新特征应推迟到下一个递增版本。

在递增版本之间应用于系统的修复内容应进行衰退测试。

## 13.7 测试的内容

现实世界是不可预测的，所以不可能测试软件可能发生的所有情况。另外，可能的输入和结果数会随着系统的规模呈指数增长，所以也没有时间测试所有的输入和输出。因此，测试案例必须根据经验和推测来确定：必须使用对一般会出错的事情的经验，推测哪些测试会覆盖最多的可能性，例如，按 10 条路径来检查系统的测试，就好于按一条路径来检查系统的测试。设计测试的另一个方法是认为，找出错误的测试是不好的测试：测试最好不围绕可能的问题来进行，因为这会对安全性产生错觉。

测试的种类随着要处理的应用程序的类型(例如桌面应用程序或服务器)和应用程序域(电

子商务系统、内嵌系统、对安全要求很高的系统等)而不同。因此，不可能编写出适用于每个系统的测试。下面列出了开始测试的一些方式(其中一些方式与黑盒子测试相关，一些与白盒子测试相关，另外一些与两者都相关):

- 测试代码是否遵循其规范：规范有两种主要形式：用例和基于类的规范。它们说明了正确输入的结果是什么，所以可以使用它们作为设计测试案例的指南。还应测试规范中包含的所有异常路径(例如，规范提到，如果输入了不正确的密码会发什么)。一般不可能测试违反前提条件的所有情形，因为测试空间是无限的。而应使用应用程序防火墙来确保总是遵循前提条件。
- 测试边界条件和区域中间的条件：代码常常要处理区域。例如，如果产品 id 是 4 到 10 个字符，就应测试代码是否至少有 4 个字符、10 个字符和 7 个字符(正好介于 4 和 10 之间)的产品 id。
- 测试 1 溢出错误：在 1 溢出错误中，开发人员要么太过，要么不足。例如，在使用 for 循环迭代 10 次时，在 Java 中可能会编写 `for(i=1;i<10;i++)`，而实际上应编写 `for(i=1;i<=10;i++)`。这个循环的第一个版本只迭代 9 次。
- 测试特殊情况：有时，开发人员只关注 99% 的时间内发生的事情，而忘记了剩余的 1%。特殊情况常常需要特定的测试编码，但异常情况与通常情况一样重要。
- 测试不寻常的情况：例如，在设计基于 HTML 的客户时，开发人员假定该客户一次只发出一个请求，但如果客户点击 Submit 按钮后，因为没有接收到预期的结果，客户又点击了结果页面上的 Back 按钮，此时该怎么办？服务器上的 Web 应用程序从同一个浏览器会话中接收两个相同的请求，但实际上只发送了一个请求。
- 测试内存错误：在使用像 C++ 这样的语言时，内存错误是很常见的，因为该语言允许直接访问内存，而且该语言也较难编程。还有经典的内存泄漏(memory leak)错误，即内存由应用程序使用，但从来没有重新声明过。甚至像 Java 这样的语言都有自动垃圾回收器，可以避免高级内存泄漏，例如在集合中添加对象，当不再需要它时却忘记删除它了。
- 测试代码复制错误：当多分支语句(例如 if-then-else)有许多类似的分支时，程序员就可能复制一个分支，多次粘贴它，然后手工编辑每个分支；在这种情况下，很容易忘记编辑其中一个分支。
- 检查操作符和过程的使用：程序员可能混淆外观类似的操作符，例如& 和 &&，或者对过程作了不正确的假定，如在计算\*之前先计算+。
- 检查等于和标识的正确用法：如第 2 章所述，等于和标识的区别非常微妙，也很重要。例如，两个字符串是相同，还是有相同的字符？
- 测试参数是否以正确的顺序传送：例如，有一个方法 `bar(int,int)`，程序员很容易编写 `bar(34,2)`，而实际上应是 `bar(2,34)`。
- 检查无意编写的无限循环和无限递归：无限循环是永远不会停止的循环，无限递归是调用它本身但没有退出条件的方法。
- 测试无法到达的代码：程序员可能无意中实现了一个不可能到达的分支，例如在 if 语句中的一个分支：在这种情况下，程序员可能在 if 或 else 的条件部分犯错。
- 检查报告的错误：程序偶尔会遇到不能处理的情况；在这种情况下，需要确保错误消息打印到错误日志中，以备以后的研究。

- 测试计算错误：基于分隔符的计算错误是很常见的。这里使用的类比是构建一个栅栏：如果构建一个 16 米的栅栏，使用以栅栏标杆分隔开的 4 米栅栏板，那么需要多个根栅栏标杆？正确答案不是(16/4)，而是(16/4+1)，因为尾部还需要一个标杆，如图 13-4 所示。

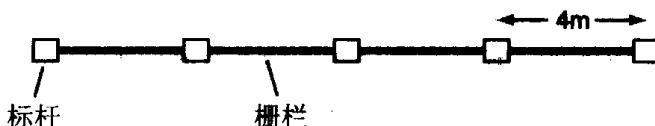


图 13-4 获取正确的栅栏标杆数

- 测试应用程序防火墙的坚固性：在系统中加上错误检查代码是比较浪费的，尤其是系统中没有错误时，浪费就更严重。例如，如果方法 `foo(int)` 需要 1 到 10 之间的参数，而使用 `foo(2)` 调用该方法时检查方法内部的参数，就比较浪费。比较好的方法是添加一个应用程序防火墙——应用程序外围的一层代码，确保无效请求无法进入系统。通过入侵应用程序防火墙，来测试应用程序防火墙是否能保护应用程序是非常重要的。
- 测试正确的初始化：在初始化软件(例如启动系统或创建对象)时，软件应把它自己初始化到适当的状态下。例如，在启动服务器时，服务器应在初始化后立即可以接收来自客户的请求：而不应启动它，再发出一些命令，才能使之可用。
- 测试变量的初始化：在程序中引入变量时，应确保它在使用前有一个适当的值。一些编程语言会自动把变量初始化为合适的默认值，例如 0(数字)或 `null`(指针)。但是，一些语言不会这么做，即使它们这么做了，默认值也不是程序员希望的值。
- 测试对象是否遵循其状态机：软件常常有一组特定的合法状态。例如，表示文件的对象有如下合法状态(在属性中记录)：“在文件系统中不存在”；“存在且打开，可用于读取”；“存在且打开，可用于写入”；“存在且关闭”。可以检查文件对象从来不会处于非法状态下，例如“不存在且打开，可用于读取”。这里 UML 状态机图中的状态模型比较有帮助。
- 测试代码是否使用了数据结构理论：有许多图书都介绍了如何实现和使用数据结构(例如列表、树、合成、集、集合、数组)。例如，如果在按字母顺序排列的列表中搜索一个值，从头开始，一个元素一个元素地搜索会非常浪费时间。应先从中间开始查找：如果要搜索的值比较大，就查找后面一半，如果要搜索的值比较小，就查找前面一半。对于有 32768 个值的列表，线性搜索平均需要进行 16384 次比较(即  $32768/2$ )，才能确定该值是否在列表中。而上述的二叉树搜索(binary chop search)平均只需进行 15 次比较。熟悉数据结构理论，并要求程序员也熟悉该理论(如果读者很性急，可以研究复杂性理论)。
- 测试以前的错误：在发布后，每个检测出的错误都表示一个新的测试案例(因为要确保该错误在修复后不会再出现，它也可能是一个常见的错误)。记录自己和同事发现的所有错误，作为要测试的问题种类的指南。

测试本身该如何检查？还应测试它们吗？如果检查测试，就必须测试这些检查工作，这种测试工作会循环下去，没完没了。幸好，要测试的代码可用于检查测试。只要测试失败，就说明所测试的代码中有错误，或者检查过程中有错误；在检查过程中，可以发现何处有错误(并修复)。另外，测试本身也应进行同行评估。

## 案例分析

iCoot 的测试计划

iCoot 系统的完整测试计划应是一个很大的文档，这里无法全部列出。附录 B.7 包含一个简要的测试计划，用于第一个递增版本生成的系统。

## 13.8 测试驱动的开发

在 20 世纪 90 年代中期，Kent Beck 和 Erich Gamma 用 Smalltalk 给测试驱动的开发设计出了一个框架。该框架称为 SUnit，非常流行，现在它有许多版本：Java、C++、C#、VisualBasic、Eiffel 等。xUnit 用于表示这些版本，但实际上这些版本都不称为 xUnit(Java 版本称为 JUnit，C++ 有一个版本称为 CPPUnit 等)。顾名思义，该框架用于单元测试，其中每个单元可以是类或类的组合。

xUnit 系列是一个简单的系列，设计人员知道，如果它很复杂，就很难使用。其理念是，在编写系统代码的同时编写测试代码，开发人员很容易运行测试代码(所以常常运行测试代码)。在测试代码中，程序员可以编写方法，来测试应用程序代码(样例代码)的各个方面。每个方法都可以创建对象，检查它们是否正确地初始化，把它们连接起来，给它们发送消息，检查结果等。

该框架的核心如图 13-5 所示。这是合成模式的一个例子，合成模式如第 11 章所述。合成模式可以建立测试层次结构，该层次结构以 TestSuite 对象作为节点，以 TestCase 对象作为叶节点。

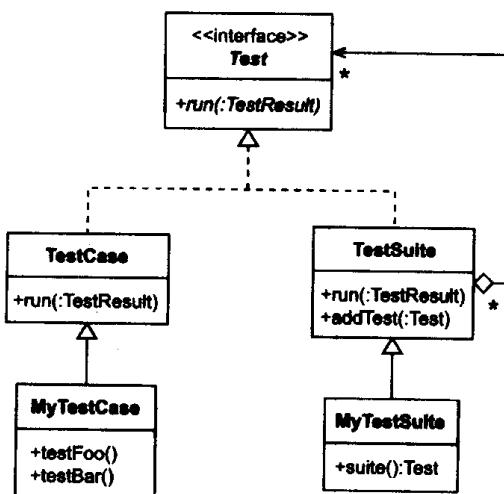


图 13-5 xUnit 的类图

每个测试案例都包含许多测试方法和实际的测试代码。每个测试套件都是测试的逻辑组合，从而建立起测试的层次结构。例如，把对业务对象的所有测试案例放在一个测试套件中，再把该测试套件合并到另一个测试套件中，该测试套件包含所有的服务器层测试案例，一直到系统本身的测试案例。

一旦实现了合成，就可以使用该框架的工具来运行测试了。

## 13.9 使用 JUnit 进行测试驱动的开发示例

要理解测试驱动的开发如何工作，最好的方法是举一个例子。下面的例子将介绍 iCoot 业务代码的开发。为了简单起见，这个讨论只与业务代码相关，与业务层之上或之下的代码无关。可以认为这个例子是人为的。但是，为了遵循最佳实践，业务代码应独立于较高的层次，所以这里介绍的步骤是真实可信的。另外，持久层代码最终会封装在业务对象中，而我们只测试业务对象的接口，所以在这方面也是真实可信的(在最终的系统中，接口中惟一的主要区别是客户使用工厂方法来创建业务对象，而不使用实际的构造函数)。

Java 用作代码段的编程语言。但是，这里没有使用复杂的 Java(只在必要时才使用不常见的、与其他面向对象语言相关的 Java，而且会清楚地标识出来，详细地解释)。代码段是用 Java 编写的，所以测试框架就是 JUnit。

下面先看看要在设计级类图(如图 13-6 所示)中开发的对象：

- Store 存储这个位置的地址和可用于出租的汽车(Store 类来自完整的 Coot 系统，而不是 iCoot，这里引入它是因为它有助于讨论)。
- Address 是一个简单的地址类，用于记录这个分支的位置。
- Car 表示我们拥有的 CarModel 的一个实例(为了简单起见，省略了 CarModelDetails)。
- CarModel 是 Car 的无形补充：它表示可用于出租的某个汽车型号。
- Category 表示汽车的类型(例如跑车、家用车、豪华车)。
- Set 是一个 Java 集合类，可以包含任意数量的独特对象。
- Make 表示汽车厂商，用名称来标识。

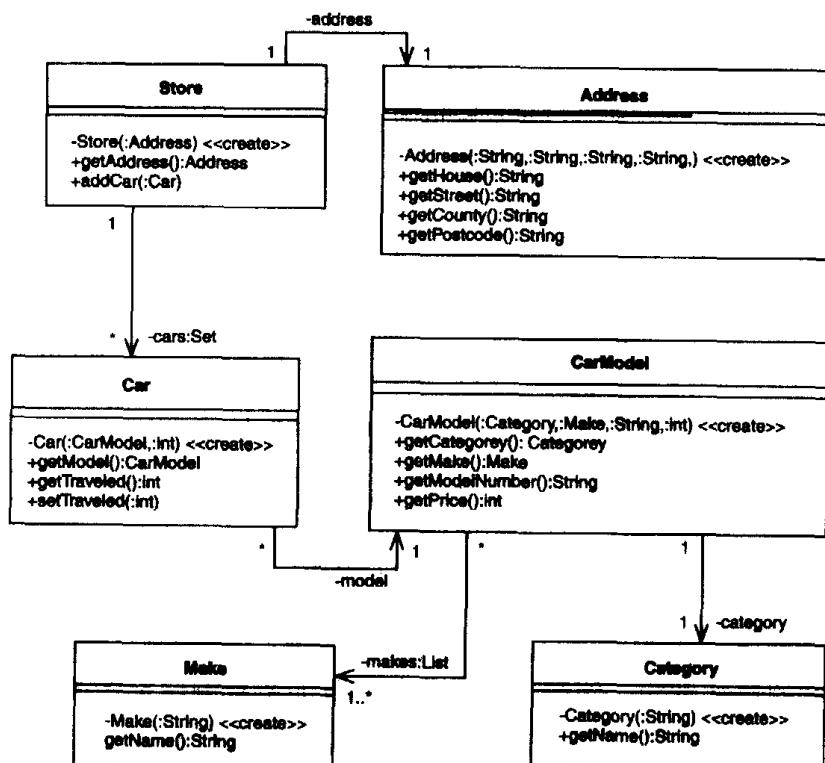


图 13-6 Coot 的部分设计类图

开发和测试好 Address、CarModel、Category 和 Make 后，开发就可以顺利进行下去了。给每个类编写一个测试案例，分别是 AddressTestCase、CarModelTestCase、CategoryTestCase 和 MakeTestCase，以确保该类正常工作。每个类最好有一个测试案例，专门用于检查这个类(因为类本身就是一个单元)。以后就可以添加检查类的测试案例。

类放在一个包 com.nowherecars.business 中。测试案例放在与业务类相同的包中，这样测试就可以使用构造函数，使包可见。

如果要把测试和 JUnit 测试运行程序一起使用，这些测试就必须是公共的，因为 JUnit 类不在包 com.nowherecars.business 中。测试对包的客户来说是可见的，这本身不一定是一件坏事：毕竟，从联合作业和开放的角度来看，应该欢迎小组的其他成员运行测试(尤其是因为要通过所有的测试后才发布代码)。如果希望在部署代码时，测试对其他客户来说是不可见的，就只需在部署版本中省略相应的类。

### 13.9.1 测试 Car 类

在使用编码工具编写 Car 类之前，应知道在测试驱动的开发中，最重要的技巧是在编写代码之前编写测试。因此，下面先实现 CarTestCase。用 JUnit 编写测试案例是很简单的：新类必须继承自 JUnit 类 TestCase，且必须包含一个或多个以 test 开头的方法——测试运行程序会自动查找这类方法，并把它们作为测试案例的一部分来执行。

每个测试方法都应全面测试一个事情，下面第一个测试将检查 Car 的创建。最初的 CarTestCase 如下所示：

```
public class CarTestCase extends TestCase {  
    public void testCreate() {  
    }  
}
```

在 testCreate 方法中，要使用构造函数创建一个 Car，构造函数的参数是 CarModel 和里程数。创建 CarModel，需要创建 Category(有一个名称)和一个 Make 对象列表(每个 Make 对象都有一个名称)。为了方便起见，只给 CarModel 添加一个 Make 对象——毕竟，此时并不是测试 CarModel 的实现。为了清晰起见，把每个属性记录为一个本地变量，因为以后需要访问该值。

编写为创建对象的代码之后，testCreate 方法如下所示：

```
public void testCreate() {  
    Category category = new Category("Saloon");  
    Make make = new Make("Fort");  
    List<Make> makes = new LinkedList<Make>();  
    makes.add(make);  
    String modelNumber = "Blur 1.6";  
    int price = 30;  
    CarModel carModel = new CarModel(category, modelNumber, makes, price);  
    int traveled = 234243;  
    Car car = New Car(carModel, 33445);  
}
```

这段代码使用了一个 Java 模板类(List)。模板类型(<Make>)告诉 Java 编译器，集合中有什么类型的对象，以防止在集合中添加其他类型的对象。

创建了 Car 后，就需要完成测试的实际工作：检查 Car 是否创建正确。为此，使用新 Car 类的获取器，比较其属性和在构造过程中使用的属性。测试案例从 TestCase 类中继承了许多 assertX 方法，它们可用于声明程序的期望状态——如果其中一个方法失败，测试运行程序就会生成一个失败报告，给出发生失败的位置和时间信息。

这里感兴趣的方法是 assertSame，它把两个对象作为其参数，检查这两个对象是否相同，即它们是否指向相同的对象；另一个感兴趣的方法是 assertEquals，它把两个对象作为其参数，检查这两个对象是否相等，即它们是否有类似的属性或者有相同的值。例如：

- assertSame(aCar, aCar)会成功
- assertSame(new Object(), new Object())会失败
- assertEquals(new Category("Sports"), new Category("Sports")) 会成功
- assertEquals(10, 11) 会失败

对于业务层，在创建 Car 时，所传送的 CarModel 应与获取器返回的 CarModel 相同。也就是说，程序会使用内部对象的标识来表示真实世界中的对象标识，而不是使用属性值(另一个方法是让“相同的属性”表示“相同的对象”，但这会导致对象的更多复制操作)。因此，需要使用 assertSame 检查 CarModel 是否正确设置。另一方面，里程数是一个原型值(公里数)，所以应使用 assertEquals 来测试。于是，在 testCreate 方法的最后添加两行：

```
...
assertSame(car.getModel(), carModel);
assertEquals(car.getTraveled(), traveled);
}
```

### 13.9.2 实现 Car 类

给 Car 类实现了第一个测试案例后，就要实现 Car 类本身了。Car 需要：汽车型号字段和里程数；以这两个字段的初始值作为参数的包构造函数、每个字段的获取器、里程数的设置器(必须能改变公里数，以便在顾客还车时更新里程数，但这对修改型号没有意义)。Car 类如下所示(包访问在 Java 中是默认的，所以构造函数不需要可见性指定符)：

```
public class Car {

    private CarModel model;
    private int traveled;

    Car(CarModel m, int t) {
        model = m;
        traveled = t;
    }
    public CarModel getModel() {
        return model;
    }
    public int getTraveled () {
        return traveled;
    }
    public void setTraveled(int t) {
        traveled = t;
    }
}
```

为了尽快开始新的测试，我们启动 JUnit 测试运行程序，选择测试案例，点击 Run 按钮。令人惊讶的是，测试运行程序指出，其中有错误(如图 13-7 所示)，标题栏显示为红色。仔细看看测试运行程序给出的信息，可以看出哪一类断言失败了(234243 和 33445 的数字比较)，失败发生在什么位置。

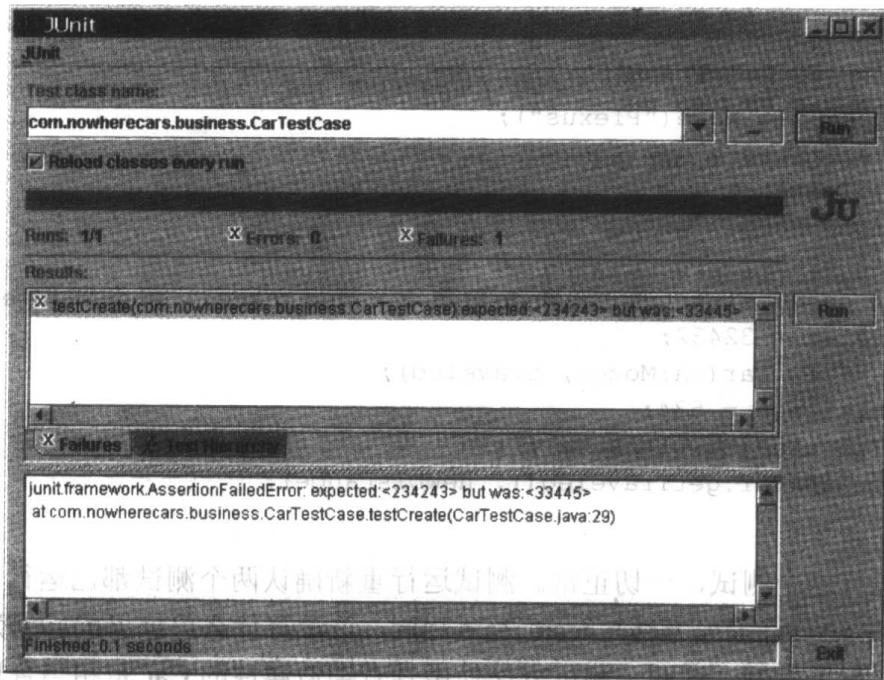


图 13-7 不成功的 JUnit 测试运行

查看 `testCreate` 的代码，会很快发现故障位于测试代码中，而不是在业务代码中：在编写测试代码时，引入了一个本地变量 `traveled`，但在创建 car 时没有使用它，所以 `assertEquals` 比较失败了。用 `traveled` 替换数字 `33445`，重新编译，重新运行测试(这不需要重新启动测试运行程序)。这次，标题栏显示为绿色，测试运行程序指出，一切正常(如图 13-8 所示)。

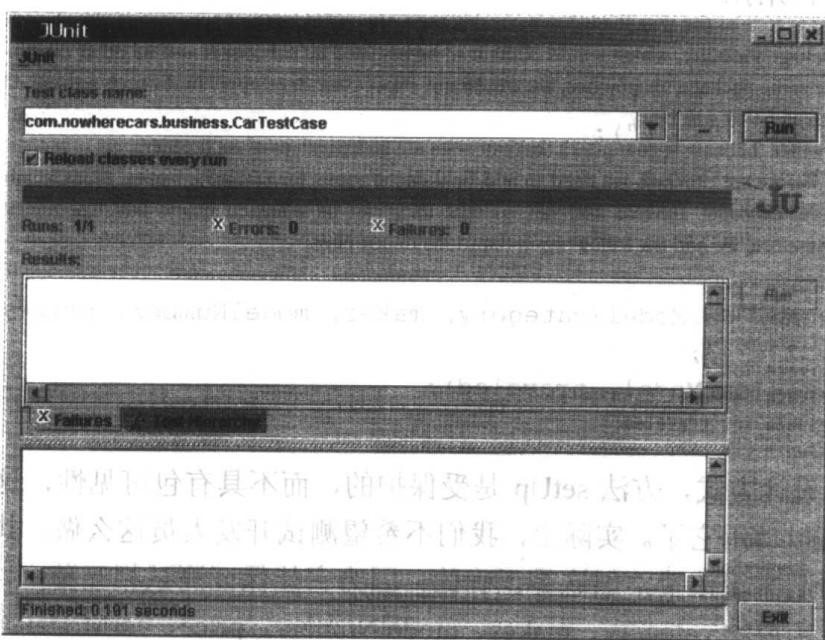


图 13-8 成功的 JUnit 测试运行

### 13.9.3 重新安排测试

现在完成了一个测试驱动的开发循环：编写测试，编写代码，运行测试，修复错误，再次运行测试。为了完成 Car 类的测试，需要检查 setTraveled 方法。与以前一样，先编写测试，给 CarTestCase 添加如下 testSetTraveled 方法：

```
public void testSetTraveled() {
    Category category = new Category("Luxury");
    Make make = new Make("Plexus");
    List<Make> makes = new LinkedList<Make>();
    makes.add(make);
    String modelNumber = "Neo STS 3.0";
    int price = 109;
    CarModel carModel = new CarModel(category, makes, modelNumber, price);
    int traveled = 32432;
    Car car = new Car(carModel, traveled);
    int newDistance = 534;
    car.setTraveled(newDistance);
    assertEquals(car.getTraveled(), newDistance);
}
```

对 Car 类运行这个测试，一切正常。测试运行重新确认两个测试都已运行。但是要注意，testSetTraveled 方法有许多地方都与 testCreate 相同，这没有什么可惊讶的，因为它们都必须从创建 Car 开始。对于本例，这两个测试方法使用具有类似属性的 Car 是相当合理的。因此，应把类似的代码提取出来，放在一个共享方法中。这个过程称为重新安排：重新组织代码，以提高质量。

可以在每个测试的开头显式调用共享方法，但 JUnit 设计人员已经考虑到了这一点：如果添加一个方法 setUp，它就会在运行每个测试方法之前自动调用(如果定义一个 tearDown 方法，它也会在每个测试之后自动调用——可以使用它发布在测试过程中使用的所有外部资源)。

方法 setUp 如下所示：

```
protected void setup() {
    category = new Category("Saloon");
    make = new Make("Fort");
    makes = new LinkedList<Make>();
    makes.add(make);
    modelNumber = "Blur 1.6";
    price = 30;
    carModel = new CarModel(category, makes, modelNumber, price);
    traveled = 234243;
    car = new Car(carModel, traveled);
}
```

由于 JUnit 的设计方式，方法 setUp 是受保护的，而不具有包可见性，这样子类和同一个包中的其他类就可以调用它了。实际上，我们不希望测试开发人员这么做。共享字段在每个测试之前创建；这看起来很浪费，但这是正确的，因为它能保证测试相互独立。

在继续之前，需要给 category、makes、modelNumber、price、carModel、traveled 和 car 添加字段声明，使它们能在任何测试方法内部访问。因此，给类添加如下声明：

```
private Category category;
private List<Make> makes;
private String modelNumber;
private int price;
private CarModel carModel;
private int traveled;
private Car car;
```

**testCreate** 和 **testSetTraveled** 方法现在就简单多了：

```
public void testCreate() {
    assertEquals(car.getModel(), carModel);
    assertEquals(car.getTraveled(), traveled);
}

public void testSetTraveled() {
    int newDistance = 534;
    car.setTraveled(newDistance);
    assertEquals(car.getTraveled(), newDistance);
}
```

进行了这些修改后，重新运行测试，一切仍正常：除了检查业务代码之外，JUnit 框架还帮助确保对测试的修改是正确的，这是衰退测试的一种形式。

完成了 **CarTestCase** 和 **Car** 的实现和测试之后，完成的类如下所示：

```
public class car {

    private int model;
    private int traveled;

    car(CarModel m, int t) {
        model = m;
        traveled = t;
    }
    public CarModel getModel() {
        return model;
    }
    public int getTraveled () {
        return traveled;
    }
    public void setTraveled(int t) {
        traveled = t;
    }
}

public class CarTestCase extends TestCase {

    private Category category;
    private List<Make> makes;
    private String modelNumber;
    private int price;
    private CarModel carModel;
    private int traveled;
    private Car car;
```

```

protected void setUp() {
    category = new Category("Saloon");
    make = new Make("Fort");
    makes = new LinkedList<Make>();
    makes.add(make);
    modelNumber = "Blur 1.6";
    price = 30;
    carModel = new CarModel(category, makes, modelNumber, price);
    traveled = 234243;
    car = New Car(carModel, traveled);
}
public void testCreate() {
    assertEquals(car.getModel(), carModel);
    assertEquals(car.getTraveled(), traveled);
}
public void testSetTraveled() {
    int newDistance = 534;
    car.setTraveled(newDistance);
    assertEquals(car.getTraveled(), newDistance);
}
}

```

#### 13.9.4 为衰退测试创建测试套件

根据前面的类图，最后一个任务是实现和测试 Store 类。与往常一样，先编写测试代码，生成 StoreTestCase 类。下面是实现 testCreate 之后 StoreTestCase 的代码：

```

public class StoreTestCase extends TestCase {

    private Address address;
    private Store store;
    private CarModel carModel;
    private Car car;

    protected void setup() {
        address = New Address("9", "Ash Lane", "Greater Manchester", "SK4 3HJ");
        store = new Store(address);
        category category = new Category("Vintage");
        Make make = new Make("Mostin");
        list<Make> makes = new LinkedList<Make>();
        Makes.add(make);
        string modelNumber = "Wheely 1950";
        int price = 89;
        carModel = new CarModel(category, makes, modelNumber, price);
        int traveled = 435345;
        car = New Car(carModel, traveled);
    }
    public void testCreate() {
        assertEquals(store.getAddress(), address);
    }
}

```

这次，多个测试都需要 Store，所以马上编写一个 setUp 方法。最初的 Store 类如下所示(注意添加 Car 的任务委派给私有的 Set)：

```
public class Store {  
  
    private Address address;  
    private Set<Car> cars;  
  
    store (Address a) {  
        address = a;  
    }  
    public Address getAddress() {  
        return address;  
    }  
    public void addCar(Car c) {  
        cars.add(c);  
    }  
}
```

运行第一个测试，一切正常。但现在已建立了多个测试案例：MakeTestCase、CategoryTestCase、CarModelTestCase 和 CarTestCase，每个测试案例都必须独立运行，这是非常不便的，尤其是在修改了代码后，就应重新运行与该代码相关的所有测试。所以应有一个测试套件——一组测试案例和可以运行为一个测试的其他测试套件。

要在 JUnit 中建立测试套件，可以添加一个继承自 TestSuite 的类，给它提供一个方法 suite，该方法返回套件中的所有测试。下面是该方法的代码：

```
public class CootBusinessTestSuite extends TestSuite {  
    public static Test suite() {  
        TestSuite result = new TestSuite();  
        result.addTestSuite(MakeTestCase.calss);  
        result.addTestSuite(CategoryTestCase.calss);  
        result.addTestSuite(CarModelTestCase.calss);  
        result.addTestSuite(CarTestCase.calss);  
        result.addTestSuite(AddressTestCase.calss);  
        result.addTestSuite(StoreTestCase.calss);  
        return result;  
    }  
}
```

这里首先要注意，suite 方法返回一个 Test 类型的对象，而不是 TestSuite 类型的对象。但是，从图 13-5 可知，TestSuite 和 TestCase 都继承自 Test。声明 suite 方法返回一个 Test 对象，它就可以返回一个 TestCase。这意味着，可以建立测试套件和测试案例的层次结构，而测试运行程序只需考虑单个类型 Test 即可。

suite 方法的一个古怪之处是，把测试案例添加到套件中的方式：result.addTestSuite(MakeTestCase.calss)。这是一个 Java 技巧，它允许测试运行程序从 test 的开头开始，查找出测试案例中的所有方法。

为了运行测试套件中的所有测试，可以像以前一样使用测试运行程序，只是现在要使它指向 CootBusinessTestSuite 类，而不是单个的测试案例，如 CarTestCase。在运行套件时，会再次

看到绿色的标题栏，说明两个新类都没有问题。套件的其他部分也没有问题。这是真正的衰退测试：检查新代码的添加是否使原有的代码崩溃。

### 13.9.5 测试 Across 方法

下面在 `StoreTestCase` 中提供一个 `testAddCar` 方法，来测试 `addCar` 方法：

```
public void testAddCar() {  
    store.addCar(car);  
    assertTrue(store.containsCar(car));  
}
```

`assertTrue` 断言只有在其参数为 `true` 时才成功。因此，如果给 `Store` 添加 `Car` 会使 `Store` 包含该 `Car`，测试就是成功的——这看起来很明显，但这种明确的测试也是需要实现的。

在编写 `assertTrue` 语句时，发现在 `Store` 类上需要一个 `containsCar` 方法。最初需要这个方法，只是为了检查添加操作是否成功，但现在要把它设置为公共的，因为它可能对其他客户有用。`containsCar` 方法没有包含在前面的类图中，因为那时我们并不知道需要它。这说明在设计和编码之间必须有一定的灵活性，这一般是件好事。需要给 `Store` 类添加如下代码：

```
public boolean containsCar(Car c) {  
    return cars.contains(c);  
}
```

新的 `testAddCar` 方法不仅要测试 `addCar`，还要测试 `containsCar`，这对于 JUnit 测试方法来说很常见。

这次在运行测试时，测试会失败，失败消息包含 `NullPointerException`：对于 Java 程序员来说，这是一个确实的信号，说明我们忘记创建对象了。这次问题出在 `Store` 构造函数身上：我们初始化了参数中的 `address` 字段，但忘记创建用于管理 `Car` 对象的 `Set` 字段了。只需给构造函数添加如下代码即可：

```
cars = new HashSet<Car>();
```

(不必担心 `HashSet`，它只是 `Set` 接口的一个 Java 具体实现而已)。

现在 JUnit 帮助实现了对象内部的一些不完整的初始化，这些内容以后还会提及。这是测试驱动的开发的另一个优势。

### 13.9.6 完成 `Store` 类

在完成上述赋值之前，还要给 `Store` 添加一个方法。这个方法是 `containsAlternativeCar`，当且仅当 `Store` 中有另一个用途与参数 `Car` 相同的 `Car` 时，该方法才返回 `true`。在汽车租赁中，这意味着，无论汽车的里程数是多少，它们的型号都必须相同。

首先给 `StoreTestCase` 添加一个测试，确认新的 `Store` 方法正确工作：

```
public void testContainsAlternativeCar() {  
    int traveled2 = 4435;  
    store.addCar(car);  
    Car car2 = new Car(carModel, traveled2);  
    assertTrue(store.containsAlternativeCar(car));  
}
```

最后是需要给 Store 添加的方法：

```
public boolean containsAlternativeCar (Car outer) {  
    for(Car inner : cars) {  
        if(inner.getModel() == outer. getModel()) {  
            return true;  
        }  
    }  
    return false;  
}
```

如果不熟悉 Java，这个方法的内容看起来很古怪。它使用 Java 中 for 循环的一个变体，该变体是专门为 Collection 类设计的，以检查 cars 中的每个 Car。在这个循环中，内部 Car 的型号要与外部 Car 的型号(作为参数传送)相比较；如果型号匹配，就返回 true。如果检查了 Store 中的所有 Car，都没有找到匹配的 Car，就返回 false。

这次运行测试时，标题栏仍是绿色的，这说明代码是正确的(JUnit 的格言是“使标题栏保持为绿色，以使代码正确”）。

Store 和 StoreTestCase 的最终版本如下所示：

```
public class Store {  
  
    private Address address;  
    private Set<Car> cars;  
  
    Store (Address a) {  
        address = a;  
        cars = new HashSet<Car>();  
    }  
    public Address getAddress() {  
        return address;  
    }  
    public void addCar(Car c) {  
        cars.add(c);  
    }  
    public boolean containsCar(Car c) {  
        return cars.contains(c);  
    }  
    public boolean containsAlternativeCar (Car outer) {  
        for(Car inner : cars) {  
            if(inner.getModel() == outer. getModel()) {  
                return true;  
            }  
        }  
        return false;  
    }  
}  
  
public class StoreTestCase extends TestCase {  
  
    private Address address;
```

```

private Store store;
private CarModel carModel;
private Car car;

protected void setUp() {
    address = new Address("9", "Appletree Avenue", "SK5 9PT");
    store = new Store(address);
    category category = new Category("Vintage");
    Make make = new Make("Mostin");
    List<Make> makes = new LinkedList<Make>();
    makes.add(make);
    String modelNumber = "Wheely 1950";
    int price = 89;
    carModel = new CarModel(category, makes, modelNumber, price);
    int traveled = 435345;
    car = new Car(carModel, traveled);
}
public void testCreate() {
    assertEquals(store.getAddress(), address);
}
public void testAddCar() {
    store.addCar(car);
    assertTrue(store.containsCar(car));
}
public void testContainsAlternativeCar() {
    int traveled2 = 4435;
    store.addCar(car);
    Car car2 = new Car(carModel, traveled2);
    assertTrue(store.containsAlternativeCar(car2));
}
}

```

这个例子到此结束，它介绍了测试驱动的开发的一个真实例子(包含错误和迭代)。开发人员下面要添加 home、统一标识符、持久代码和服务器代码。接着，添加一个简单的客户来测试已经开发出来的代码。总之，要创建测试案例，逐层合并到测试套件中，并频繁地运行测试。

## 13.10 小结

本章的主要内容如下：

- 测试人员使用的术语，以总结复杂的概念和任务。常用的术语有几十个，但测试团体使用它们的方式是一致的，所以有必要理解它们的含义。
- 如何测试大型系统。这涉及到测试计划的讨论、不间断的测试、所有制品(不仅仅是代码)的测试，以及不同的人应如何参与。
- 一个测试驱动的开发的有效实例。测试驱动的开发是很有价值的，但在正式的测试阶段，不应把它看做同事和顾客试图攻击系统的替代品。

## 13.11 考外阅读

全面介绍软件测试的一本经久不衰的图书是[Myers 等 04]，许多人都认为这是理解测试理论、学习实际应用测试理论的必备图书。

测试驱动的开发的创始人之一 Kent Beck 在[Beck 02]中描述了测试理论和实践。JUnit 的主页是 [www.junit.org](http://www.junit.org)，其中包含了该框架本身、技术说明、相关文章和实例。

# 附录 A Ripple 小结

这个附录将完整地概述本书使用的简化方法 Ripple。表 A-1 按阶段总结了应生成的制品。为了生成这些制品，应按如下步骤进行(但要螺旋式、递增式地反复地交付)。

表 A-1 按阶段生成的制品

阶段	制 品		UML
起源		任务陈述或非正式的需求 任务 责任 项目计划 工作本 术语表(全面更新) 测试计划	否 否 否 否 否 否 否 否
需求	业务	参与者列表(带有描述) 用例列表(带有描述) 用例细节 活动图(可选) 通信图(可选)	否 否 否 是 是
	系统	参与者列表(带有描述) 用例列表(带有描述) 用例细节 用例图 用例调查 用户接口的框架	否 否 否 是 否 否
分析		类图 通信图	是 是
设计	系统	部署图 层图	是 否
	子系统	类图 顺序图 数据库模式	是 是 否
类规范		注释	否
实现		源代码	否

(续表)

阶 段	制 品	UML
测试	测试报表	否
部署	压缩打包的解决方案	否
	手册	否
	培训材料	否
维护	错误报告 递增计划	否 否

1. 项目来源(与顾客一起进行)
  - (a) 了解顾客的需求，或者告诉他们需要什么
  - (b) 从顾客处获取需求文档，作为任务陈述或较长的文档；如果没有这些文档，就和顾客一起编写非正式的需求文档。
2. 分配任务
  - (a) 确定开发任务(例如规划、管理、计时、开发、测试、系统管理)。
  - (b) 确定每个任务由谁负责。
3. 编写业务手册：在纸上或者在线业务手册上包含所有的项目制品。
4. 编写术语表
  - (a) 编写一个术语表，以记录项目术语的定义。
  - (b) 在整个开发过程中更新术语表。
5. 制订项目计划
  - (a) 为螺旋式递增阶段进行初步的规划，并制订一个进度表。
  - (b) 在整个开发过程中，定期检查、调整项目计划。
6. 制订测试计划：考虑不间断的测试、测试阶段、顾客评估、开发和维护。
7. 业务需求(与顾客一起)：
  - (a) 制订业务参与者列表(及其描述)
  - (b) 制订业务用例列表(及其描述)
  - (c) (可选)用活动图描述业务用例
  - (d) (可选)用通信图描述业务用例
  - (e) 确定业务用例的细节
8. 系统需求(与顾客一起)：
  - (a) 使用用户界面草图描述系统交互
  - (b) 制订系统参与者列表(及其描述)
  - (c) 制订系统用例列表(及其描述)
  - (d) 绘制系统用例图
  - (e) 制订系统用例调查表
  - (f) 确定系统用例的细节
  - (g) 给系统生成辅助需求
  - (h) 确定系统用例的优先级

9. 分析:

- (a) 绘制分析类图
- (b) 制订属性列表(及其描述)
- (c) (可选)使用状态机给复杂的生命周期建模, 在状态机图上记录结果
- (d) 实现用例, 使用通信图描述结果
- (e) 制订操作列表(及其描述)

10. 系统设计

- (a) 选择技术
- (b) 确定重用的可能性(库、模式和框架)
- (c) 绘制层图
- (d) 编写层交互策略
- (e) 设计包结构, 在包图上记录它
- (f) 绘制部署图
- (g) 制订安全策略
- (h) 制订并发策略

11. 子系统设计

- (a) 定义业务服务
- (b) 确定更多重用的可能性(库、模式和框架)
- (c) 把分析类映射到业务层类上: 类列表(及其描述)、类图、字段表(及其描述)。
- (d) 生成数据库模式
- (e) 为其他层设计类(例如服务器和协议、服务小程序、控制、持久)
- (f) 实现业务服务, 在顺序图上记录结果
- (g) 制订消息列表(及其描述)
- (h) 最后确定用户界面的设计

12. 类的规范

- (a) 为每个类生成非正式规范
- (b) 在设计和源代码中记录非正式规范

13. 实现

- (a) 编写单元测试
- (b) 编写实现代码

14. 测试

- (a) 让测试小组测试系统
- (b) 修改错误

15. 部署

- (a) 生成手册和课件
- (b) 在客户的系统上安装代码制品
- (c) 培训客户

16. 维护

- (a) 修改错误
- (b) 把顾客的反馈、改进建议和市场变化合并到新的递增版本中

# 附录 B iCoot 案例分析

## B.1 业务需求

本节介绍在 iCoot 开发的需求阶段根据项目任务陈述和业务用例模型进行的业务需求建模。业务用例模型也应用于整个 Coot 系统。

### B.1.1 顾客的任务陈述

下面是 Nowhere Cars 在 Coot 项目的一开始交付的任务陈述:

商店将汽车的跟踪自动化了——使用条形码、柜台终端和激光阅读器，这有许多优点：租赁助手的效率提高了 20%，汽车很少失踪，客户群很快变大(根据市场调查，其部分原因至少是专业化和效率的显著提高)。

管理层认为，Internet 会提供进一步提高效率、降低成本的机会。例如，现在不是打印可用汽车的目录，而可以让每个 Internet 冲浪人员在线浏览这些目录。对于有特权的客户，可以提供额外的服务，例如通过鼠标点击进行预约。这个领域的目标是每个商店的运营成本降低 15%。

在两年内，使用电子商务的所有功能，通过 Web 浏览器提供所有的服务，在客户家中完成汽车的交付和收回，以达到虚拟租赁公司的最终目标，将未预约业务的运营成本降到最低。

与顾客一起把这个任务陈述扩展为业务用例。

### B.1.2 参与者列表

- 助手：商店的一个员工，帮助顾客租用其汽车和预约汽车型号。
- 顾客：为获得一个标准服务而付费的人。
- 会员：其身份和信用状况已得到验证的顾客，因此，可以访问特定的服务(例如电话预约或通过 Internet 预约)。
- 非会员：其身份和信用状况没有验证的顾客，因此，他要预约必须交押金，要租用汽车必须提供一份驾照副本。
- Auk：处理顾客信息、预约、出租和可用汽车型号目录的已有系统。
- 债务部门：处理未付费用的 Nowhere Cars 部门。
- 法律部门：处理涉及租用汽车的事故的 Nowhere Cars 部门。

### B.1.3 用例列表

- B1: 顾客租用汽车: 顾客租用从可用汽车中选择出来的汽车。
- B2: 会员预约汽车型号: 当有该型号的汽车时, 会员应得到通知。
- B3: 非会员预约汽车型号: 当有该型号的汽车时, 非会员交纳了押金, 就应得到通知。
- B4: 顾客取消预约: 顾客通过电话或亲自取消未结束的预约。
- B5: 顾客交还汽车: 顾客交还租用的汽车。
- B6: 顾客获知有某型号的汽车: 当有该型号的汽车时, 助手会与顾客联系。
- B7: 报告失踪: 顾客或助手发现汽车失踪了。
- B8: 顾客重新预约: 超过一星期后, 顾客可以重新预约。
- B9: 顾客访问目录: 顾客在店内或在家中浏览目录。
- B10: 顾客因没有取预约的车而接受罚款: 顾客没有取预约好的车。
- B11: 顾客取预约好的车: 顾客取预约好的车。
- B12: 顾客成为会员: 顾客提供信用卡信息和地址证明, 成为会员。
- B13: 通知顾客汽车已超过租用期限: 助手与顾客联系, 警告顾客他租用的汽车已超过租用期限一星期了。
- B14: 顾客丢了钥匙: 为丢钥匙的顾客提供备用钥匙。
- B15: 更新会员卡: 当会员卡过期时, 助手与顾客联系, 更新会员卡。
- B16: 汽车不能还回来: 汽车出事或坏了。

### B.1.4 用例的通信图

通信图在业务需求建模过程中用得不是很多(但在收集系统需求的过程中用得较多)。但还是绘制出图 B-1, 来说明“B3:非会员预约汽车型号”中涉及的外部和内部参与者。

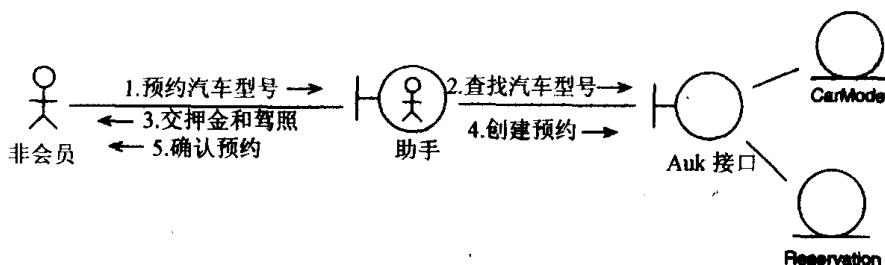


图 B-1 “B3:非会员预约汽车型号”的通信图

### B.1.5 用例的活动图

活动图在业务需求建模过程中用得不是很多, 但还是绘制出图 B-2, 来说明“B3:非会员预约汽车型号”用例的细节。

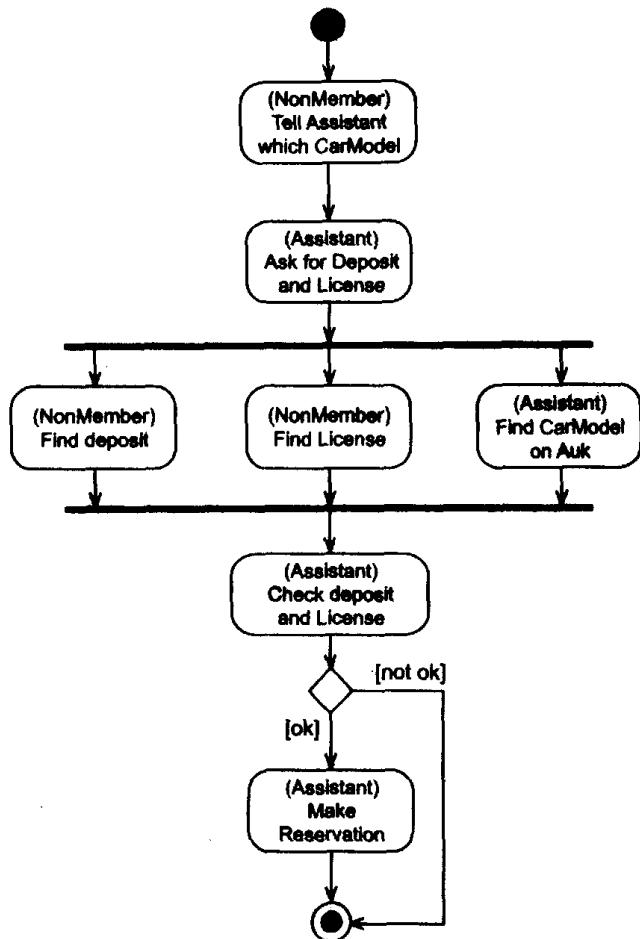


图 B-2 “B3:非会员预约汽车型号”的活动图

### B.1.6 用例的细节

**B1:顾客租用汽车:**

1. 顾客告诉助手要租用什么汽车型号。
2. 如果 Auk 发现没有这种汽车，就给顾客提供一种备用汽车型号。
3. 如果有这种汽车，助手就在 Auk 中把该汽车标记为已租用。
4. 助手要求顾客提供驾照，以确认他们的身份。
5. 对于会员，助手从会员卡中提取出会员号，检查他们是否欠费，是否被禁止租用汽车。
6. 对于非会员，助手检查顾客是否已在 Auk 中；如果不在，助手就把顾客驾照的副本扫描到 Auk 中，记录顾客的姓名、电话号码和驾照号。
7. 如果顾客的信息是令人满意的，且没有欠费，就要为租用汽车付费。
8. 如果付费失败，就在 Auk 中取消该汽车。
9. 如果付费成功，就给顾客提供钥匙，并进入显示区域。

**B2:会员预约汽车型号:**

1. 会员告诉助手他们的会员号(通过电话或亲自来)。
2. 会员告诉助手要预约的汽车型号。
3. 如果会员没有被禁止租用汽车，他们的信用卡没有过期，且没有欠费，就在 Auk 上进

行预约。

4. 如果通过电话进行预约，会员就可以通过确认信用卡信息来付费，这些信用卡信息必须匹配存储在 Auk 上的信息，且不能是过期的。

5. 告诉会员预约号。

B3:非会员预约汽车型号：

1. 非会员告诉助手要预约的汽车型号。
2. 助手在 Auk 中查找该汽车型号。
3. 助手请求非会员为预约交纳押金。
4. 助手请求非会员提供驾照和电话号码。
5. 助手检查非会员的驾照。

6. 如果驾照没有问题，助手就会创建新的预约，并记录驾照号码、电话号码，在 Auk 中扫描驾照。

7. 助手给非会员一个预约卡，其中包含唯一的预约号。

B4:顾客取消预约：

1. 顾客可以随时取消预约。
2. 会员可以通过电话或亲自来提供会员号，以取消预约。
3. 非会员必须亲自来取消预约：他们要给助手出示驾照，助手应检查该驾照是否匹配在 Auk 中扫描的驾照，并退回押金。
4. 如果汽车已开到保护区域，就把对应的 Car 对象返回到显示区域。

B5:顾客交还汽车：

1. 当汽车返还到检查区域时，助手扫描条形码，以确认还车，检查油箱是否已满。
2. 助手把汽车返还到显示区域。
3. 如果顾客返还过期的汽车，或者返还的汽车油箱不满，顾客就必须付相应的费用——会员可以使用已有的、未过期的信用卡信息来付费。
4. 如果顾客拒绝付费，他们的信息就会传送到债务部门。

B6:顾客获知有某型号的汽车：

1. 在返还汽车时，Auk 告诉助手它是否匹配某个预约对象。
2. 如果匹配，助手就把该汽车移动到保留区域。
3. 在先到先服务的准则下，助手应试图通过电话联系有预约的顾客。
4. 如果顾客在两天内没有来，就取消他们的预约，把汽车从保留区域移动到显示区域。

B7:报告失踪：

1. 如果 Auk 表示应在显示区域中的汽车在需要时找不到了，或在盘点时找不到了，就向警察报告汽车失窃。
2. 如果顾客报告汽车失踪，就向警察报告汽车失窃，并报告顾客(已知最后一个开走汽车的人)的信息。
3. 在上述两种情况下，都在 Auk 中记录丢失数据。

B8:顾客重新预约：

1. 如果七天内都没有预约到汽车，就必须重新预约。
2. 助手有两天时间电话联系顾客，看看他们是否要在接着的七天内重新预约。
3. 如果顾客不打算重新预约，就取消预约；顾客必须返回商店，出示驾照，才能取回押金。

**B9:顾客访问目录：**

1. 顾客可以到商店来查看目录。
2. 顾客还可以复印目录，带回家去，但这需要付费。
3. 如果顾客选择加入邮件列表，就会每六个月收到一个目录的免费副本。

**B10:顾客因没有取预约的车而接受罚款：**

1. 如果有预约好的汽车型号，助手也通过电话告诉顾客了，顾客就必须在两天内取车。
2. 如果顾客没有取车，就结束预约，助手把匹配的汽车从保留区域移动到显示区域中。
3. 对于非会员，会没收其押金。
4. 对于会员，会在 Auk 中记录一次罚款，并把顾客的信息传送给债务部门。

**B11:顾客取预约好的车：**

1. 顾客来商店从保留区域取车。
2. 顾客出示驾照。
3. 如果驾照匹配 Auk 上的信息，就把预约标记为已结束。
4. 助手给顾客提供钥匙，并把他们带到保留区域。

**B12:顾客成为会员：**

1. 为了成为会员，顾客必须提供驾照、地址证明和信用卡信息。
2. 助手检查驾照和地址证明。
3. 助手通过信用卡公司检查信用卡。
4. 如果没有问题，助手就在 Auk 中记录驾照号、地址、电话号码和信用卡信息。
5. Auk 发放新会员卡和惟一的会员号。
6. 如果信用卡过期，就不允许执行其他的会员动作，除非会员返回商店，提供新的信用卡。

**B13:通知顾客汽车已超过租用期限：**

1. 因为租用业务是事先付费的，所以顾客如果忘记返还汽车，就会受到警告。
2. 如果汽车超过了一个星期的期限，助手就试图与顾客电话联系。
3. 如果顾客在两个星期内联系不上，就报告汽车失踪(参见 B7)。

**B14:顾客丢了钥匙：**

1. 如果顾客告诉助手他丢了钥匙，就让送快信的人为顾客提供备用钥匙。
2. 在 Auk 中，把备用钥匙的费用加到该顾客的信息中。

**B15:更新会员卡：**

1. Auk 记录信用卡过期的会员处于不好的状态下。
2. Auk 通知助手，该会员的信用卡已过期。
3. 助手电话联系会员，告诉他们必须更新其会员卡。
4. 会员带着新的信用卡到商店来，把信用卡信息输入 Auk 中。

5. Auk 记录会员处于良好状态下。

B16: 汽车不能还回来: 汽车出事或坏了。

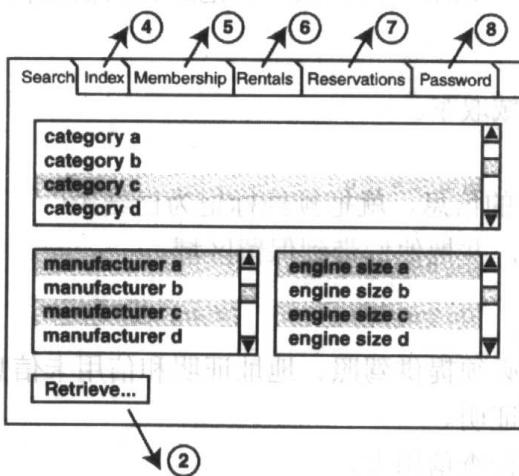
1. 如果顾客告诉助手, 汽车出事或坏了, 助手就安排修车。
2. 如果汽车出事了, 就把信息传送给法律部门。

## B.2 系统需求

本节根据用户界面草图和系统用例模型, 介绍在 iCoot 开发的需求阶段进行系统建模的结果。

### B.2.1 用户界面草图

iCoot 的用户界面草图在顾客的帮助下建立, 如图 B-3~B-10 所示。



② 非会员只能查看 search 和 index 页面

会员需要登录和注销机制

图 B-3 用户界面草图 1(创建查询)

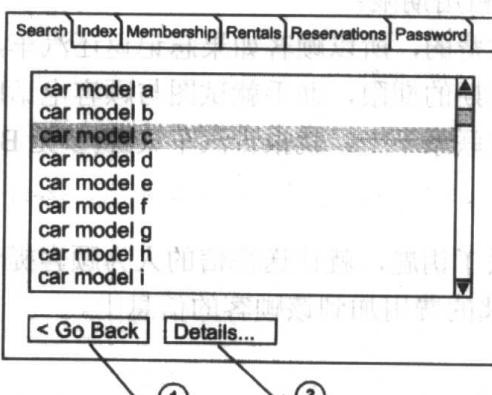


图 B-4 用户界面草图 2(查看结果)

Search Index Membership Rentals Reservations Password

Category Sports

Make(s) Abc, Def

Model 358

Engine Size 3.6

Description Waffle

Daily Price €89.50

< Go Back Reserve... Advert... Poster...

图 B-5 用户界面草图 3(查看汽车型号信息)

Search Index Membership Rentals Reservations Password

index entry a  
index entry b  
index entry c  
index entry d  
index entry e  
index entry f  
index entry g  
index entry h  
index entry i

Retrieve... → 进入 Search 页面

图 B-6 用户界面草图 4(选择一个索引标题)

Search Index Membership Rentals Reservations Password

Personal Details

Address

Credit Card

< Go Back Reserve... Advert... Poster...

图 B-7 用户界面草图 5(查看会员信息)

Search Index Membership Rentals Reservations Password

rental a  
rental b  
rental c  
rental d

图 B-8 用户界面草图 6(查看租赁信息)

Search Index Membership Rentals Reservations Password

reservation a  
reservation b  
reservation c

Cancel...

图 B-9 用户界面草图 7(查看预约信息)

Search Index Membership Rentals Reservations Password

Old Password \*\*\*\*\*

New Password \*\*\*\*\*

Repeat New Password \*\*\*\*\*

Change... Clear Fields

图 B-10 用户界面草图 8(修改密码)

## B.2.2 参与者列表

- 顾客：使用 Web 浏览器访问 iCoot 的人。
- 会员：在一家商店提供了姓名、地址和信用卡信息的顾客；每个会员都有一个 Internet 密码和一个会员号。(特殊化顾客)
- 非会员：不是会员的顾客。(特殊化顾客)
- 助手：商店的一个员工，他与会员联系，告诉他们预约的进展情况。

## B.2.3 用例列表

- U1: 浏览索引：顾客浏览汽车型号的索引(特殊化 U13, 包含 U2)。

- U2: 查看结果：给顾客显示检索到的汽车型号子集(被 U1 和 U4 包含，被 U3 扩展)。
- U3: 查看汽车型号的细节：给顾客显示检索到的汽车型号细节，例如描述和广告(扩展 U2，被 U7 扩展)。
- U4: 搜索：顾客通过指定类别、构造和引擎规格，搜索汽车型号(特殊化 U13，包含 U2)。
- U5: 登录：会员使用会员号和当前密码登录 iCoot(由 U6、U8、U9、U10 和 U12 扩展)。
- U6: 查看会员信息：会员查看 iCoot 存储的会员信息，例如姓名、地址和信用卡细节(扩展 U5)。
- U7: 进行预约：会员在查看汽车型号的细节时，预约一种汽车型号(扩展 U3)。
- U8: 查看租用情况：会员查看当前租用的汽车汇总信息(扩展 U5)。
- U9: 修改密码：会员修改用于登录的密码(扩展 U5)。
- U10: 查看预约情况：会员查看还没有结束的预约汇总信息，例如日期、时间和汽车型号(扩展 U5，被 U11 扩展)。
- U11: 取消预约：会员取消还没有结束的预约(扩展 U10)。
- U12: 注销：会员从 iCoot 中注销(扩展 U5)。
- U13: 查找汽车型号：顾客从类别表中检索汽车型号的子集(抽象，被 U1 和 U4 一般化)。

#### B.2.4 用例图

iCoot 的用例图如图 B-11 所示。

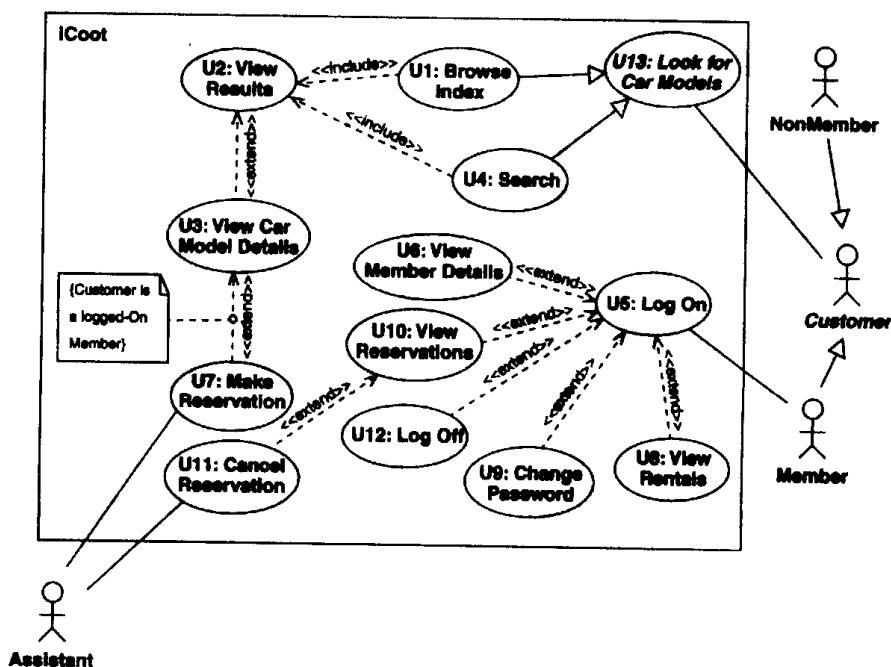


图 B-11 iCoot 的用例图

#### B.2.5 用例调查

iCoot 的用例调查描述了用例如何相互关联，如下所示：

任何顾客都可以浏览汽车模型索引(U1)或通过搜索(U4)，在目录中查找汽车型号。在后一种情况下，顾客要指定他们感兴趣的类别、构造和引擎规格。无论采用哪种方式，在每次检索

后，都会给顾客显示匹配汽车型号的集合(U2)，以及基本信息，例如汽车型号的名称。然后，顾客就可以选择查看特定汽车型号的其他信息，例如描述和广告(U3)。

已成为会员的顾客可以登录(U5)，访问额外的服务。额外的服务有进行预约(U7)，取消预约(U11)，检查会员信息(U6)，查看已有的预约(U10)，修改登录密码(U9)，查看已有的租用记录(U8)和注销(U12)。

助手涉及到预约的整个过程，例如把汽车开进开出保留区域。

顾客有两种情况：会员和非会员。

浏览索引和搜索汽车型号是查找汽车型号(U13)的两种不同方式。为了查看汽车型号的细节，顾客必须查看搜索型号的结果(通过浏览或搜索路径)。

为了预约汽车型号，会员必须查看其细节(非会员不能预约，即使他们在查看细节也不行)。要取消预约，会员必须查看已有的预约。

## B.2.6 用例细节

U1: 浏览索引(特殊化 U13，包含 U2)

前提条件：无

1. 顾客选择一个索引标题。
2. 顾客选择查看选中索引标题的汽车型号
3. 包含 U2

后置条件：无

U2: 查看结果(被 U1 和 U4 包含，被 U3 扩展)。

前提条件：无

1. iCoot 给顾客显示每个检索出来的汽车型号，包括型号和价格。
2. 用 U3 扩展。

后置条件：无

U3: 查看汽车型号的细节(扩展 U2，被 U7 扩展)

前提条件：无

1. 顾客选择一个匹配的汽车型号。
2. 顾客请求选中汽车型号的细节。
3. iCoot 显示选中汽车型号的细节(构造、引擎规格、价格、描述、广告和海报)。
4. 如果顾客是一个已登录的会员，就用 U7 扩展。

后置条件：iCoot 显示选中汽车型号的细节。

非功能需求：

- r1. 广告应使用流协议显示，而不应要求下载。

U4: 搜索：(特殊化 U13，包含 U2)

前提条件：无

1. 顾客选择需要的类别。
2. 顾客选择需要的构造。
3. 顾客选择需要的引擎规格。

4. 顾客开始搜索。

5. 包含 U2。

后置条件：无

异常路径：

a1. 如果顾客没有指定类别、构造或引擎规格，而是检索整个目录，iCoot 就不允许启动搜索。

U5:登录(由 U6、U8、U9、U10 和 U12 扩展)

前提条件：会员从本地商店获得一个密码。

1. 会员输入会员号。

2. 会员输入密码。

3. iCoot 强制会员必须登录，所以会员可以选择盗取(验证无效，所以盗取)已有的会话。

4. 会员选择登录。

5. 用 U6、U8、U9、U10 和 U12 扩展。

后置条件：会员登录。

异常路径：

a1. 如果会员号和密码组合是不正确的，iCoot 会通知会员，这两个中的一个不正确(为了安全起见，不会说明是哪一个不正确)。

a2. 如果会员号和密码组合是正确的，但会员已经登录，且没有选择盗取会话，iCoot 会通知会员。

U6:查看会员信息： (扩展 U5)

前提条件：无。

1. 会员选择查看会员信息。

2. 给会员显示会员信息(姓名、地址、状态、拥有的账户、信用卡信息)。

3. 为了安全起见，iCoot 只能显示会员信用卡号的最后四位数字。

4. iCoot 给会员显示正确的信息，会员必须联系本地商店。

后置条件：已给会员显示了会员信息。

U7:进行预约： (扩展 U3)

前提条件：顾客是已登录的会员。

1. 会员在显示区域中选择保留汽车型号的细节。

2. iCoot 要求会员确认，并发出警告：如果没有及时取走已预约的车，就会被罚款。

3. 会员确认预约。

4. iCoot 给会员显示预约号，指示助手当有预约的汽车时联系顾客。

5. 助手登录到 Coot 上时，Coot 会显示需要执行的预约列表。

6. 助手采取必要的行动来推进预约(如果有预约的汽车，就切换到可取车状态，并把汽车放在保留区域)。

后置条件：完成所有已请求的预约。

异常路径：

a1. 如果会员限定了预约条件，就不进行预约。

U8:查看租用情况： (扩展 U5)

前提条件：无。关系： U5。

1. 会员选择查看他们租用的信息。
  2. iCoot 给会员显示他们当前租用的汽车汇总信息(包括号码牌和租用期限)。
- 后置条件： iCoot 给会员显示了当前租用的汽车汇总信息。

#### U9:修改密码： (扩展 U5)

前提条件： 无。

1. 会员选择修改密码。
2. 会员输入旧密码(在屏幕上会被屏蔽)。
3. 会员输入新密码(屏蔽)。
4. 会员再次输入新密码(用于确认，也是被屏蔽的)。
5. 会员启动修改。
6. iCoot 请求确认(警告必须记住新密码)。
7. 如果会员确认，就修改密码。

后置条件： 密码已修改。

异常路径：

a1. 如果旧密码不正确或新密码不匹配，就告诉会员(为了安全起见，不给出错误的细节)，不修改密码。

a2. 如果旧密码匹配，但新密码没有遵守密码规则(至少是 6 个字母和数字)，就通知会员，不修改密码。

#### U10:查看预约对象： (扩展 U5， 被 U11 扩展)

前提条件： 无。

1. 会员选择查看预约信息。
2. iCoot 显示还没有结束的预约汇总信息，包括预约号、状态、时间戳和汽车型号。
3. 用 U11 扩展。

后置条件： 已给会员显示了已有的预约汇总信息。

#### U11:取消预约： (扩展 U10)

前提条件： 无。

1. 会员选择一个预约。
2. 会员选择取消预约。
3. iCoot 请求确认。
4. 会员确认要取消预约。
5. iCoot 把预约标记为已结束，更新助手的终端。

后置条件： 已确认要取消的预约都标记为已结束。

异常路径： a1. 如果会员没有确认取消， iCoot 就不采取任何行动。

#### U12:注销： 会员从 iCoot 中注销(扩展 U5)。

前提条件： 无。

1. 会员选择注销。
2. iCoot 结束当前会话。
3. iCoot 关闭只支持会员的功能。

后置条件：会员已注销。

异常路径：a1. 为了安全起见，如果登录的会员在十分钟内没有与 iCoot 交互，就会自动注销。

U13:查找汽车型号(抽象，由 U1 和 U4 特殊化)

前提条件：无

后置条件：给顾客显示检索到的汽车型号汇总信息。

### B.2.7 辅助需求

s1. 客户小程序必须运行在 Java PlugIn 1.2(和更新的版本)上。

s2. iCoot 必须能处理 100 000 种汽车型号。

s3. iCoot 必须能同时给一百万顾客服务，且性能没有明显的降低。

### B.2.8 用例的优先级

下面列出了 iCoot 用例的优先级，其中的分数用于第一个递增版本。

- 绿色：

- U1: 浏览索引
- U4: 搜索
- U2: 查看结果
- U3: 查看汽车型号的细节
- U5: 登录

- 黄色：

- U12: 注销
- U6: 查看会员信息
- U7: 进行预约
- U10: 查看预约情况

- 红色：

- U11: 取消预约
- U8: 查看租用情况
- U9: 修改密码

在第一个递增版本中，也完成了 U6。其他用例在第二个递增版本中完成。

## B.3 分析

本节根据分析类模型、预约的状态机和用例的实现(通信图)，来介绍 iCoot 开发的分析阶段的成果。预约状态机也应用于完整的 Coot 系统。类模型包括几个来自 Coot 系统的部分，例如非会员和丢失日期。

### B.3.1 类图

iCoot 的分析类图如图 B-12 所示。大多数类也出现在设计类模型(B.5 节)中，所以它们的描述放在术语表(B.8 节)中，以避免重复。

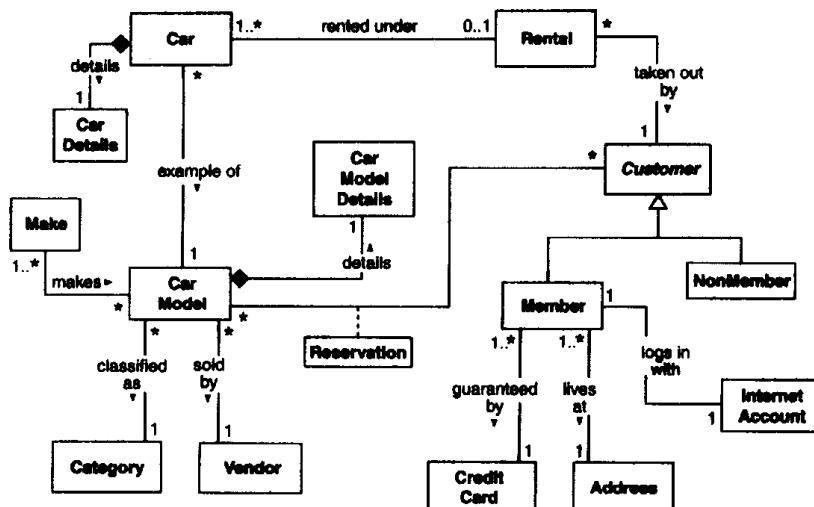


图 B-12 分析类图

### B.3.2 属性

iCoot 的类属性如图 B-13 所示。这些属性在设计类模型中显示为字段，并在设计类图中给出它们的类型和描述，细节请参阅设计说明(B.5 节)。

<b>Car</b>	<b>CarModel</b>	<b>CarModelDetails</b>	<b>Rental</b>
travelled dateLost[0..1]	name price	engineSize description advert poster	number startDate dueDate totalAmount
<b>CarDetails</b> barCode:String numberPlate vin	<b>Category</b> name		
<b>Make</b> name	<b>Member</b> inGoodStanding: boolean number:String	<b>CreditCard</b> number:String type:String expiryDate	<b>Reservation</b> number:String timestamp state
<b>Address</b> house street county postCode	<b>Vendor</b> name	<b>Customer</b> name phoneNumber: String amountDue:int	<b>NonMember</b> driversLicense: String
		<b>Internet Account</b> password:String	

图 B-13 分析属性

### B.3.3 操作列表

- **CarModel:**
  - `getSummary()`——获取接收者的汇总信息，包括型号和价格。
  - `getDetails()`——获取接收者的信息，包括构造、引擎规格、价格、描述、广告和海报。
- **CarModelHome:**
  - `findByIndexHeading(h:String)`——在索引标题 h 下搜索 CarModel 对象。
  - `findByQuery(categories, makes, sizes)`——搜索 CarModel 对象，它的类别为指定的 Category，构造为 Make，引擎规格为 Size。

- LogonController:
  - logon(n:String, p:String, s:Boolean) ——用会员号 n 和密码 p 登录为会员，指定是否用 s 盗取已有的会话。
  - changePassword(m:Member, o:String, n1:String, n2:String) ——只要 n2 匹配，且当前密码为 o，就把 m 的密码改为 n1。
  - logoff() ——注销已登录的会员。
- Member:
  - getPassword():String ——获取接收者的密码。
  - isLoggedIn():boolean ——如果接收者已登录，就返回 true。
  - logon() ——使接收者登录。
  - logoff() ——注销接收者。
  - getDetails() ——获取接收者的信息，包括姓名、地址、状态、已有的账户和(隐藏的)信用卡信息。
  - setPassword(p:String) ——把接收者的密码设置为 p。
    - MemberHome: findByMembershipNumber(n:String):Member ——用会员号 m 查找会员。
    - MemberUI:
  - search(categories, makes, sizes) ——搜索 CarModel 对象，它的类别为指定的 Category，构造为 Make，引擎规格为 Size。
  - index(h:String) ——在索引标题 h 下搜索 CarModel 对象。
  - logon(n:String, p:String, s:boolean) ——用会员号 n 和密码 p 登录为会员，指定是否用 s 盗取已有的会话。
  - setMember(m:Member) ——把已登录的会员设置为 m。
  - showMemberDetails() ——显示已登录会员的信息。
  - showRentals() ——显示已登录会员的 Rental 对象。
  - showReservations() ——显示已登录会员的未结束 Reservation 对象。
  - changePassword(o:String, n1:String, n2:String) ——只要 n2 匹配，且当前密码为 o，就把登录会员的密码改为 n1。
  - confirmChange() ——确认密码已修改。
  - reserve(c:CarModel) ——给已登录的会员预约 c。
  - confirmReserve() ——确认已进行预约。
  - cancel(r: Reservation) ——取消 r。
  - confirmCancel() ——确认预约已取消。
  - showDetails(c:CarModel) ——显示 c 的细节。
  - logoff() ——注销已登录的会员。
- NonMemberUI:
  - search(categories, makes, sizes) ——搜索 CarModel 对象，它的类别为指定的 Category，构造为 Make，引擎规格为 Size。
  - index(h:String) ——在索引标题 h 下搜索 CarModel 对象。
- Rental: getSummary() ——获取接收者的汇总信息，包括车牌号和到期日。
- RentalHome: findByMember(m:Member) ——获取会员 m 的 Rental 对象。
- Reservation:
  - getSummary() ——获取接收者的汇总信息，包括号码、时间戳、状态和汽车型号。

- `getNumber()` —— 获取接收者的号码。
- `setState(s)` —— 把接收者的状态设置为 s。
- `ReservationHome`:
  - `findByMember(m:Member)` —— 获取 m 的预约信息。
  - `create(c:CarModel, m:Member)` —— 为 m 预约 c, 包括当前日期和时间。

### B.3.4 预约的状态机

图 B-14 显示了预约的状态机图, 用于建立复杂的生命周期模型。相关状态机调查如下。

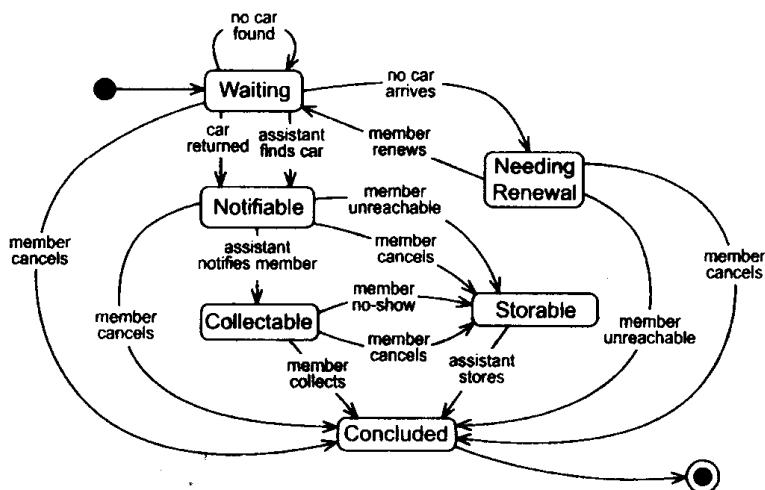


图 B-14 预约的状态机图

当会员通过 Internet 预约汽车型号时, `Reservation` 就等待助手来处理(所以顾客可以在没有助手的帮助下进行预约)。如果以后的某个时刻助手发现在停车场的显示区域中有一辆合适的、未预约的汽车, 或者顾客返还了一辆汽车, `Reservation` 就变成 `Notifiable`。此时, 汽车移动到保留区域。

如果在一星期内都没有给某个 `Reservation` 提供汽车, 该 `Reservation` 就变成 `NeedingRenewal`: 必须电话或亲自通知会员, 让他们取消预约, 或在下周重新预约。如果会员取消了预约, 或在五天内联系不上, `Reservation` 就变成 `Concluded`。

一旦 `Reservation` 变成 `Notifiable`, 助手就必须在三天内电话或亲自通知会员。如果联系上顾客, `Reservation` 就变成 `Collectable`, 否则就变成 `Displayable`(移动到保留区域的汽车必须返还到显示区域)。

一旦 `Reservation` 变成 `Collectable`, 会员就必须在三天内取车, 如果会员取了车, `Reservation` 就变成 `Concluded`, 否则就变成 `Displayable`。

一旦 `Displayable Reservation` 的汽车返还到显示区域, `Reservation` 就变成 `Concluded`。

会员可以在任何时候通过 Internet 电话或亲自取消预约。

系统会告诉助手当前预约的状态(还未结束), 这样助手就可以采取适当的措施。

### B.3.5 用例的实现

iCoot 的通信图验证了分析类模型, 如图 B-15~B-26 所示, 每个系统用例都有一个通信图。注意使用了 `guard`(括号中的任意条件), 以指定条件消息, 用`*`指定迭代(迭代条件用于控制迭代, 但这些都会使图更复杂)。

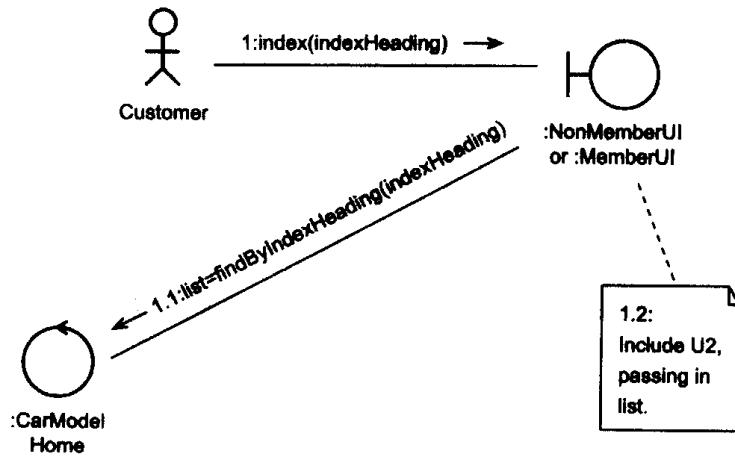


图 B-15 “U1:浏览索引”的通信图

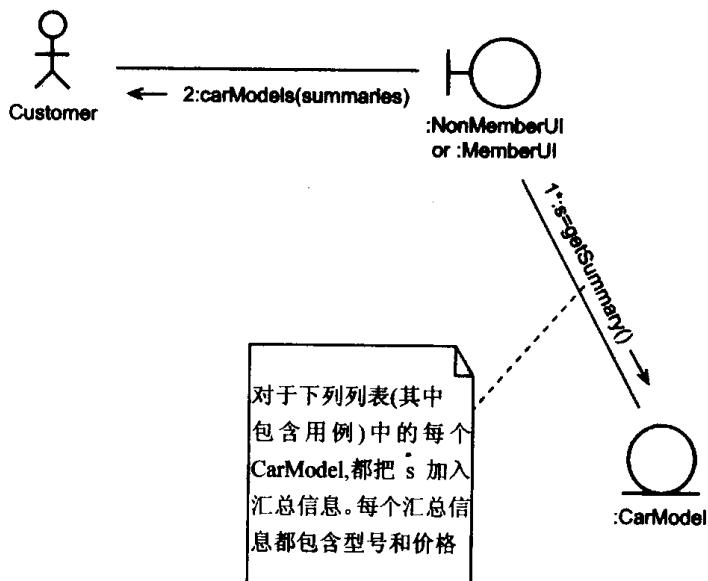


图 B-16 “U2:查看结果”的通信图

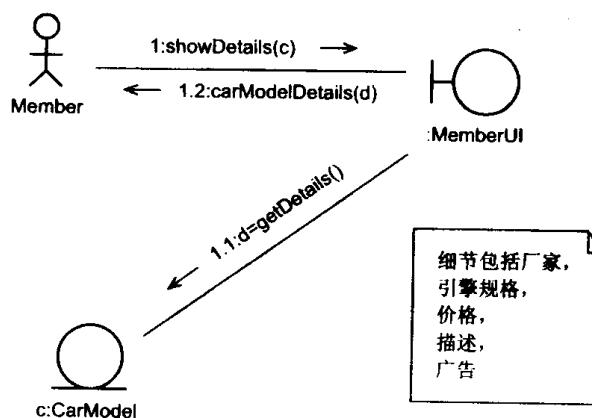


图 B-17 “U3:查看汽车型号的细节”的通信图

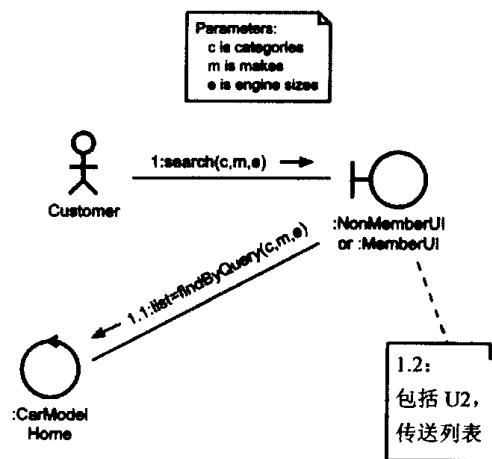


图 B-18 “U4:搜索”的通信图

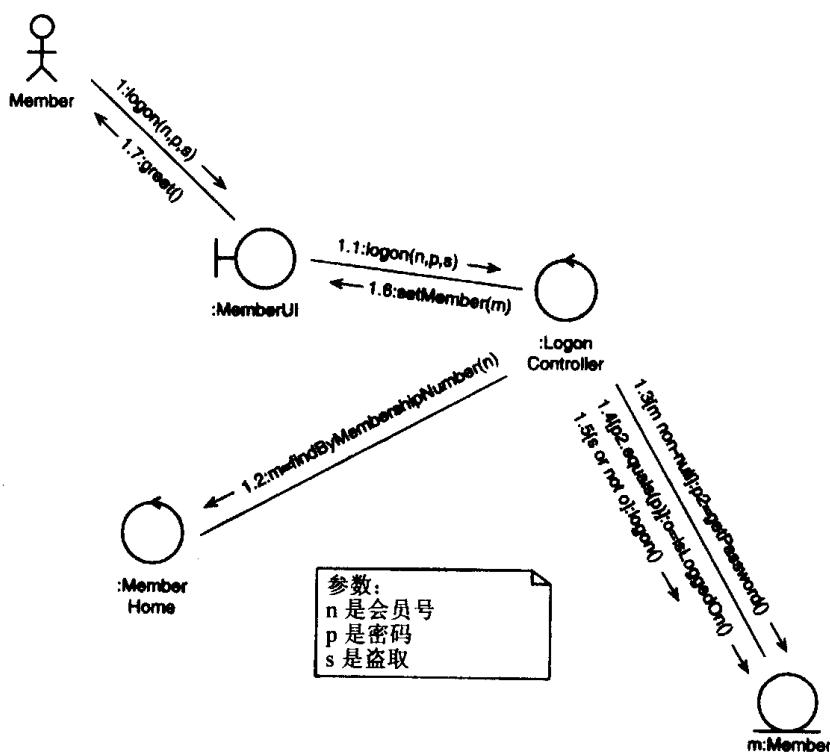


图 B-19 “U5:登录”的通信图

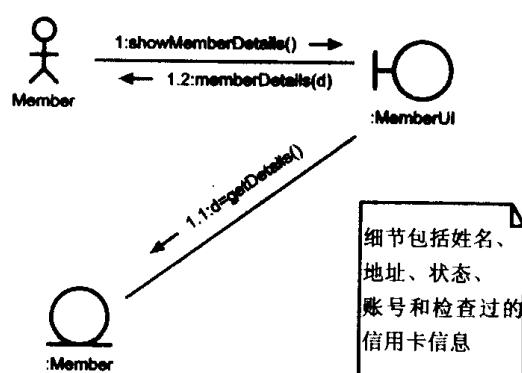


图 B-20 “U6:查看会员信息”的通信图

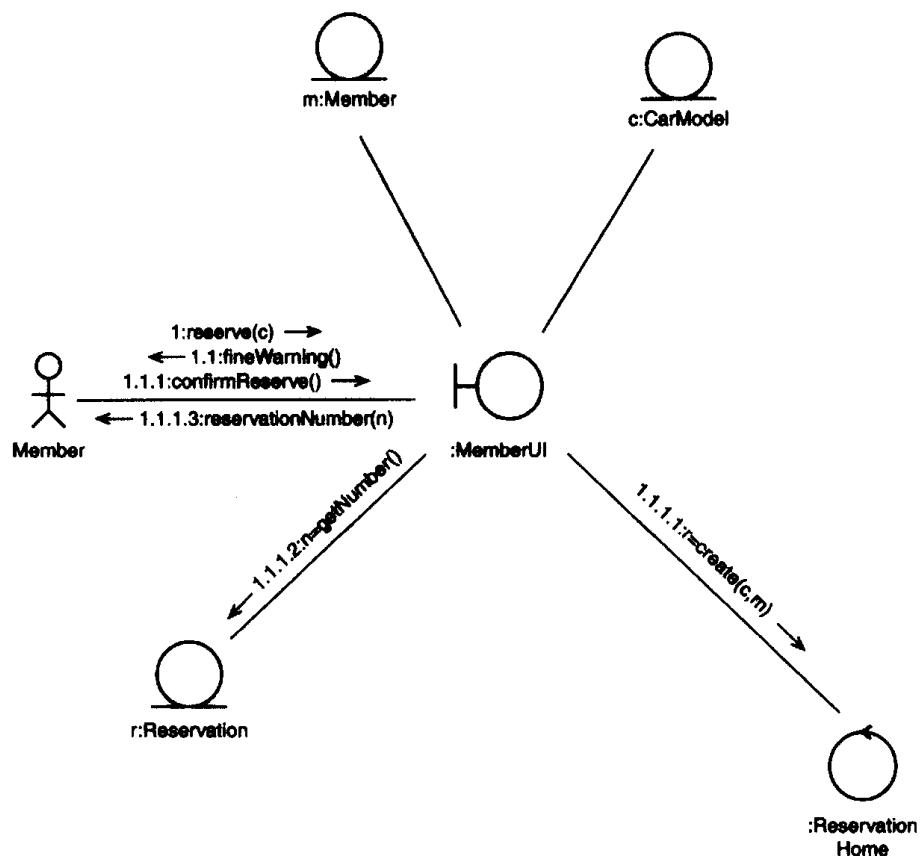


图 B-21 “U7:进行预约”的通信图

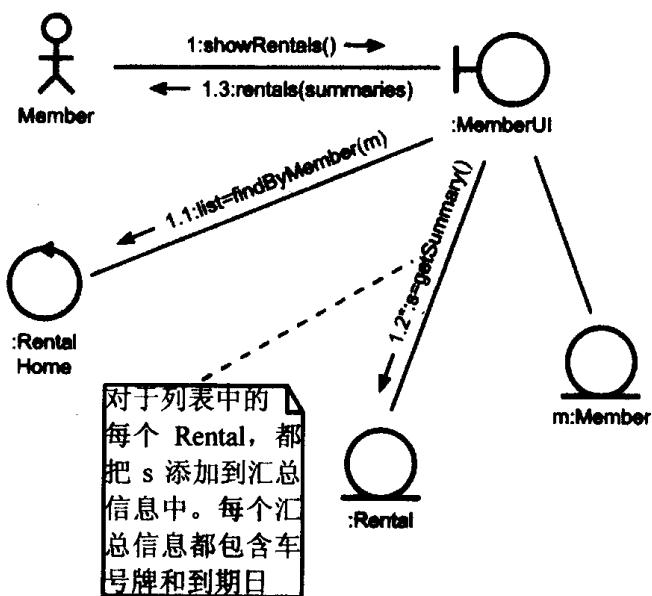


图 B-22 “U8:查看租用情况”的通信图

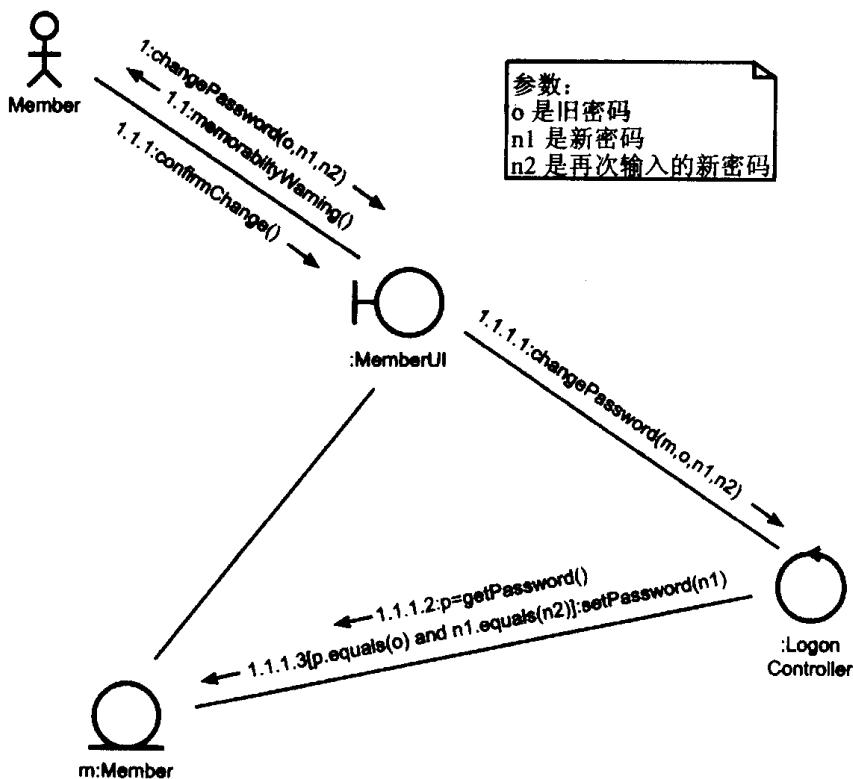


图 B-23 “U9:修改密码”的通信图

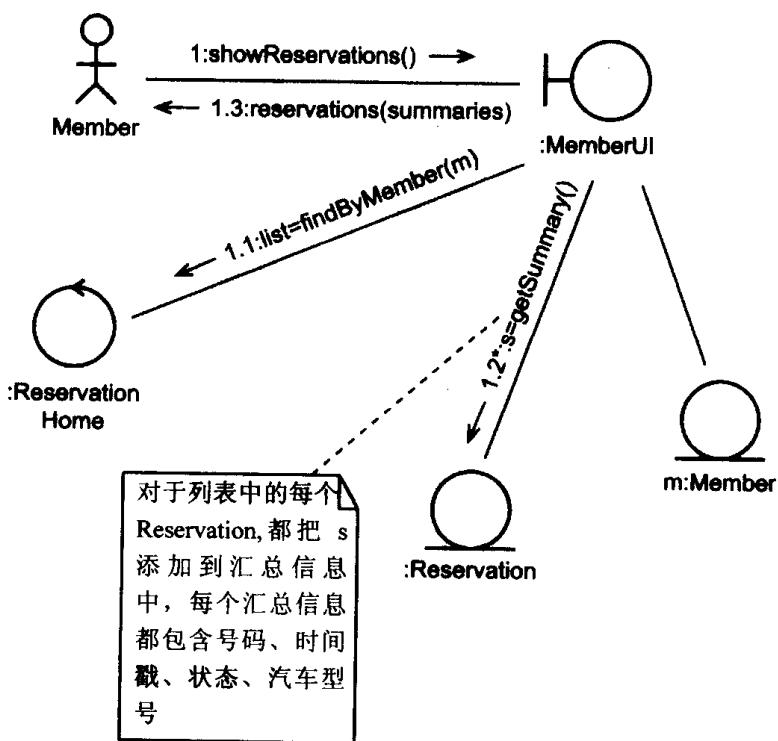


图 B-24 “U10:查看预约情况”的通信图

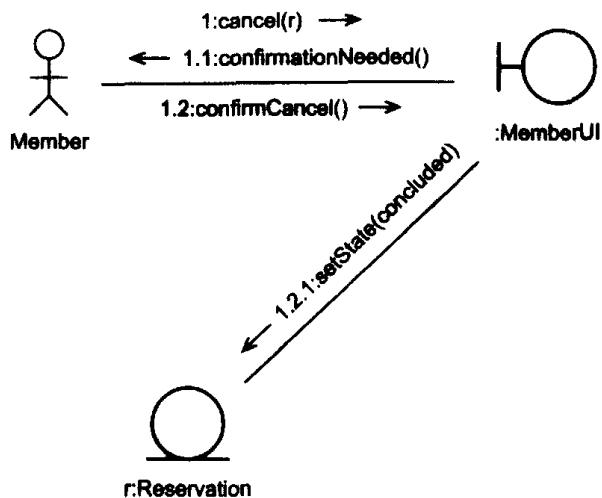


图 B-25 “U11:取消预约”的通信图

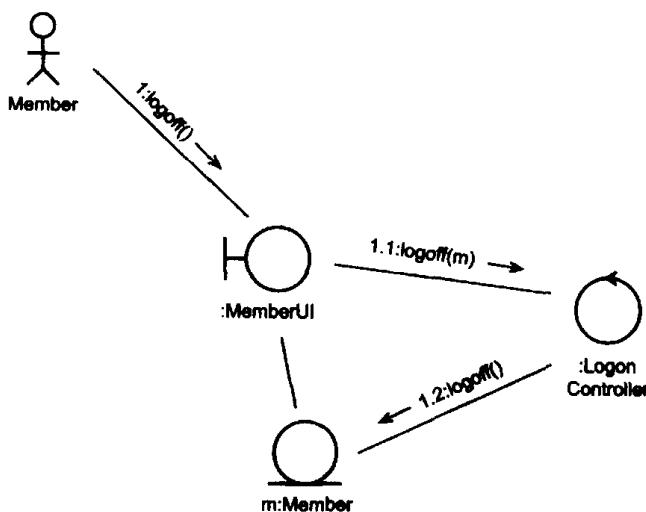


图 B-26 “U12:注销”的通信图

## B.4 系统设计

本节根据技术选择、层、包、部署图、安全策略和并发策略，介绍在 iCoot 开发的设计阶段执行的系统设计的结果。

### B.4.1 选择技术

在客户端，技术的选择由顾客的便利性决定——我们不希望顾客必须安装软件，才能访问服务。另外，顾客应能使用任意桌面机器，无论该机器安装了什么操作系统。显然，因为客户环境选择的应是 Web 浏览器，用户界面必须是能交互操作的，才能进行预约，所以必须选择 HTML/CGI、Java 小程序、ActiveX 控件或者 Flash。由于需要移植(还要考虑客户的安全性)，因此可以排除 ActiveX 控件。在浏览站点时，顾客最好能即时访问用户界面，这就排除了 Java 小程序和 Flash，它们在下载界面时一般都会有延迟。因此，最初的用户界面是 HTML/CGI。

在服务器端，服务小程序是处理 CGI 请求的一个好选择，因为它们是可以移植的、比较有效，还可以访问 J2EE 的所有功能，J2EE 提供了服务小程序需要的所有功能(例如访问分布式事务管理)。传统的脚本在可移植性、表达能力和性能方面有一些问题，.NET 技术在移植性方面有问题。一旦把服务小程序用作服务器的入口点，生成动态 Web 页面就显然应选择 JSP 机制。

最初，使用 J2EE 的开放源代码版本(免费)来开发和部署，但要小心地避免专用的锁定。该版本必须支持把请求传送给运行在独立进程中的服务小程序和 JSP，使 JSP 能直接通过 GUI 客户来访问。如果开放源代码的版本不够用，就可以购买商业产品，重新部署代码。

由于服务小程序可以移植，因此可以把它们部署在任意硬件和操作系统的组合中，以后如果需要，再重新部署。开始时，每个商店都把系统软件部署在两个有预算的 Linux 服务器上，以确保自动防止故障，通信量比较大，避免购买高容量服务器或高级硬件。

对于业务数据，最初使用开放源代码的数据库，以后如果需要，还可以改为使用商业产品。使用关系数据库是因为，其技术比较成熟，应用程序是面向业务的，有大量的数据，但没有特别复杂的逻辑。数据库也部署在每个商店的一对 Linux 服务器上。

将来，要提供可用于移动电话或 PDA 的用户界面。最好避免在设备上使用不能管理 HTML/CGI 界面的 WAP，因为这会很笨拙、也不好推广。应使用 J2ME，以提供丰富的交互操作，这些操作可以根据屏幕的大小自动缩放。对于喜欢在客户机上安装 J2SE 的顾客，应手工或使用 Web 浏览器的插拔机制，并提供方便的图形化用户界面，作为一个应用小程序。这个应用小程序部署在每个商店的触摸屏上，在顾客查看广告时，可以改进顾客的操作便利性。

J2ME 和 J2SE 会绕过 JSP 和服务小程序，提高性能。

#### B.4.2 层图

iCoot 的层如图 B-27 所示。

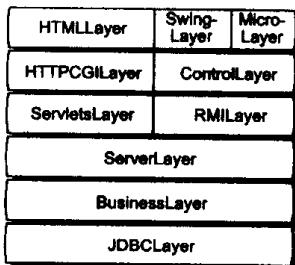


图 B-27 iCoot 的层图

持久由 JDBCCLayer 提供，使用标准的 JDBC 库访问关系数据库。没有单独的持久层，因为关系数据库就可以满足系统生命周期内的要求。

BusinessLayer 包含分析类图中实体对象以及各种支持对象的实现。这些对象包括数据传入传出数据库的 JDBC 代码。

ServerLayer 把 BusinessLayer 中的对象和消息以服务器对象上的消息的形式传送给业务服务。ServerLayer 中的对象是 EJB 会话对象，这有两个优点：第一，可以访问 J2EE 事务管理；第二，可以通过 RMI 直接访问 GUI 客户机，绕过 Web 服务器。

为了使 ServerLayer 保持关闭，业务服务返回的所有信息都采用协议对象的形式，即业务对象的轻型副本。

ServletsLayer 是 HTML/CGI 客户的一个控制层。每个服务小程序都把 ServerLayer 上的一

个或多个对象转换为可以从客户机发出的简单命令和问题。为了响应每个命令或问题，服务小程序要执行必要的动作，再把下一个 HTML 页面传回客户机。这样页面设计和源代码就是分开的，每个回应页面都由 JSP 建立，根据顾客的交互操作动态生成其内容。JSP 把动态数据接收为协议对象，由服务小程序传送。HTMLLayer 的网络通信由标准的 HTTPCGILayer 提供。

Rmilayer 是一个网络层，允许从 GUI 上进行远程访问(Java 应用程序和使用 J2ME 的任意设备)。这个层上的对象只是 ServerLayer 上 EJB 会话对象的修饰器：每个服务器对象都用一个 RMI servant 来修饰，每个 RMI servant 则通过客户机上的 RMI 代理来访问。在与 ControlLayer 通信时，Rmilayer 使用的协议对象与调用 JSP 时 ServletsLayer 使用的协议对象相同。

ControlLayer 位于 GUI 对象和 RMI 代理之间。它用于简化与服务器对象的交互，隐藏 RMI 的细节。RMILayer、ControlLayer、SwingLayer 和 MicroLayer 都不详细介绍，因为图形化用户界面不是 iCoot 第一个递增版本的一部分。

### B.4.3 层交互策略

在服务器上，为了便于简化，所有的层通信都是向下进行的。换言之，消息只能从一层传送到下面的层上。客户端使用事件，是为了 SwingLayer 和 MicroLayer 的方便，这样与应用程序相关的信息就可以从用户接口组件向下传送到 ControlLayer(HTML/CGI 前端不需要事件，因为显示给用户的所有信息都由服务小程序计算，直接传送给 JSP，进行显示)。

层是关闭的，以便于实现和维护：每个对象都可以访问其下一层中的对象，但不能访问其上一层中的对象。

### B.4.4 包

Coot 的包图(包括第一个递增版本中没有实现的图形化用户界面/RMI 包)。如图 B-28 所示。

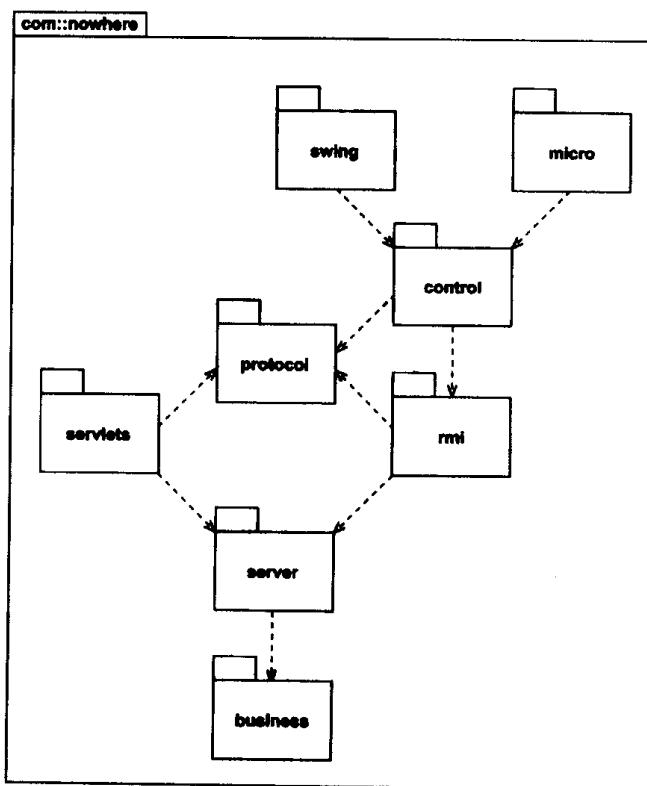


图 B-28 iCoot 的包图

## B.4.5 部署图

iCoot 的部署图如图 B-29 所示。

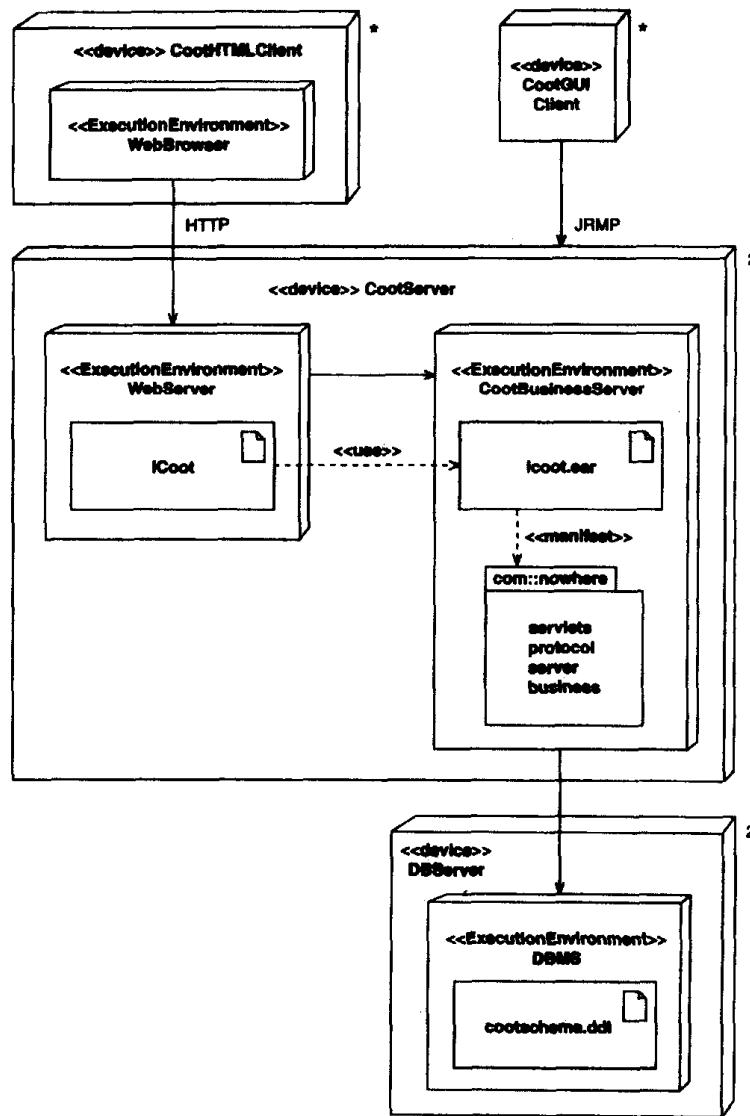


图 B-29 iCoot 的部署图

iCoot 数据层有两个数据库服务器(称为 DBServer)。有这样两个节点可以提高通过量和可靠性。每个 DBServer 都拥有一个 DBMS 过程，以管理对数据的访问。

cootschema.ddl 制品包含了创建数据库表的命令，其格式为该数据库所特有的。这会使用与数据库相关的工具(这里没有给出细节)，部署到每个 DBMS 过程中。注意 cootschema.ddl 包含整个 Coot 系统的模式，因为 iCoot 和 Coot 使用相同的数据。

中间层与数据层通信，它包含两个服务器(CootServer)，也是为了提高通过量和可靠性。每个 CootServer 都拥有一个 CootBusinessServer(用于处理业务请求)和一个 WebServer(用于处理静态 HTML 内容，把业务请求传送给 CootBusinessServer)。CootBusinessServer 的数据访问由 DBMS 提供。它们都专用于所选的产品，所以不指定 WebServer 与 CootBusinessServer 之间的通信协议和 CootBusinessServer 与 DBMS 之间的通信协议。

在每个 CootServer 中, iCoot 文件夹包含静态的 HTML 页面, 部署到 WebServer 上, 而 icoot.ear 文档部署到 CootBusinessServer 上。icoot.ear 文档包含 com::nowhere 包中的服务小程序、JSP、业务对象和(最终)RMI 修饰工具。

每个 CootServer 都可以由任意多个 CootHTMLClient 节点同时访问。每个 CootHTMLClient 都拥有一个 WebBrowser, 以使用 HTTP 访问某个 WebServer 节点。不需要把制品部署到 CootHTMLClient 节点上。

最后, 还可以提供从 CootGUIClient 节点的访问。每个 CootGUIClient 都使用 JRMP 访问一个 CootServer 节点。因为未来递增的一个主题是机制允许这种请求进入 CootBusinessServer, 所以不给出细节。也不给出 CootGUIClient 的任何细节。没有指定部署到 CootGUIClient 节点上的制品。

#### B.4.6 安全策略

所有的访问者都可以搜索和浏览服务, 不需要登录。另外, 对于只支持会员的服务, 每个会员都必须先亲自到本地商店获得一个密码, 然后使用该密码从他们所选的客户机上登录到会员区域。用于登录的会员号和密码在一个中心目录中管理, 以便于使用标准的 Java 集成机制进行维护。

为了保护会员活动的私密性, 在客户机上对会员服务的所有访问都要通过 SSL 进行, 而不是一般的 TCP/IP。SSL 还在服务器之间使用, 进行内部保护。

服务器部署在 Internet 防火墙后面, 强力控制外部访问。

#### B.4.7 并发策略

BusinessLayer 中的对象使用分布式事务来管理。在每个业务请求的开始(如 ServerLayer 对象上的每个方法), 都会创建一个 Java 事务——这个事务与业务对象在该请求中进行的每个数据库访问关联起来。在每个请求的最后, 都会提交 Java 事务, 使更新的结果可用于其他请求。

为了把事务冲突降到最低, 所有的 RMI servant、服务小程序和服务器对象都是无状态的。

对于 GUI 客户, 对本地数据的访问(协议对象的副本)都是单线程的。对于 HTML 客户, 每个 JSP 都可以对其协议对象进行独占访问, 这种访问也是高效的单线程访问。对于业务数据, 低级并发控制由 EJB 框架自动管理: 对业务服务的每次使用都封装在一个事务中, 以正确地传送给数据库管理系统。

为了简化业务层上的并发控制, 可以采用两种策略: 第一, 迫使成员只能登录一次。第二, 在脱线状态下更新可用汽车型号的类别, 在凌晨切换到可用的类别上。这样就不需要向顾客和助手报告像“试图预约的汽车型号已被撤销”这样的错误了(这个错误偶尔会发生, 因为在通过并发路径修改服务器上的数据时, 客户机的显示可能没有显式更新。尽管把所有相关的更新内容都传送给客户机在技术上是可行的, 但效率太低)。

### B.5 子系统设计

本节将根据业务服务、设计类模型(每一层都有一个类模型和协议对象)、数据库模式、用

户界面设计和业务服务的实现(顺序图)，介绍在 iCoot 开发的设计阶段中子系统设计的结果。本节基本上描述了支持业务服务的实现所需要的子系统制品和 CootHTMLClient。

### B.5.1 业务服务

1. 从汽车型号索引中读取标题。
2. 读取消给定索引标题的汽车型号。
3. 读取汽车型号的所有类别。
4. 读取汽车型号的所有引擎规格。
5. 读取汽车型号的所有厂家。
6. 读取消给定类别、引擎规格和厂家的汽车型号。
7. 读取消给定汽车型号的细节。
8. 预约一种汽车型号。
9. 读取消给定会员的细节。
10. 修改会员的密码。
11. 读取消给定会员租赁的汽车。
12. 读取消给定会员的预约信息。
13. 取消预约。

### B.5.2 ServletsLayer 类图

ServletsLayer 的类图如图 B-30 所示。

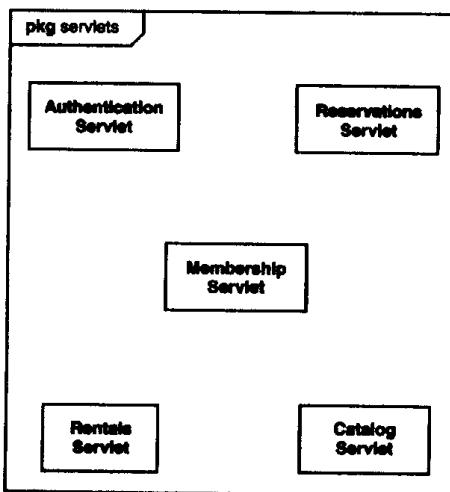


图 B-30 ServletsLayer 类

### B.5.3 ServletsLayer 的字段列表

所有的服务小程序都是无状态的，因此都没有字段。顾客交互状态的信息记录在结果页面中，也记录在浏览器的 HTTP 会话中存储的一个状态对象中。对于登录的会员，会话标识符也存储在 HTTP 会话中。服务器层上的对象使用其 home 来定位(用单一模式实现)。

#### B.5.4 ServletsLayer 的消息列表

使用标准的 Java 机制，每个客户请求都通过一个消息 `doGet (:HttpServletRequest, :HttpServletResponse)` 传送给选中的服务小程序。客户的问题和命令传送为 `HttpServletRequest` 的一部分。下面列出了客户可以传送的问题和命令，以及作为结果调用的 JSP。

##### **AuthenticationServlet**

- `logon`: 在主页上；其参数是会员号、密码和 `steal`，如果成功登录，就返回会员页面(`steal`参数表示会员是否要盗取已有的会话)。
- `logoff`: 在会员页面上；注销当前的会员。

##### **CatalogServlet**

- `index`: 在会员或非会员页面上；返回索引页面，其中包含可供选择的索引标题。
- `browse`: 在索引页面上；把索引标题作为参数，返回结果页面，其中包含匹配的汽车型号。
- `search`: 在会员或非会员页面上；返回搜索页面，其中包含可供选择的类别、厂家和引擎规格。
- `query`: 在搜索页面上；其参数是类别 ID、厂家 ID 和引擎规格，返回结果页面，其中包含匹配的汽车型号。
- `detail`: 在结果页面上；把汽车型号 ID 作为参数，返回细目页面，其中包含该汽车型号的细节。

##### **MembershipServlet**

- `membership`: 在会员页面上；返回会员页面，其中包含当前会员的细节。
- `password`: 在会员页面上；返回密码页面。
- `changePassword`: 在会员页面上；其参数是旧密码和新密码，返回确认修改页面。
- `confirmChange`: 在确认修改页面上；如果给出的旧密码正确，就设置当前会员的新密码。

##### **RentalsServlet**

- `rentals`: 在会员页面上；返回租用页面，其中包含当前会员的租用信息。

##### **ReservationsServlet**

- `reserve`: 在细目页面上；把汽车型号 `id` 作为参数，返回确认预约页面。
- `confirmReserve`: 在确认预约页面上；预约已经标识的汽车型号，返回确认页面。
- `ok`: 在确认页面上，返回细目页面。
- `reservations`: 在会员页面上；返回预约页面，其中包含当前会员的预约信息。
- `cancel`: 在预约页面上；以预约 `id` 作为参数，返回确认取消页面。
- `confirmCancel`: 在确认取消页面上；取消以前选择的预约，返回预约页面。

#### B.5.5 ServerLayer 类图

ServerLayer 的类图如图 B-31 所示。每个类还有用单一模式实现的 `home`(没有显示)。

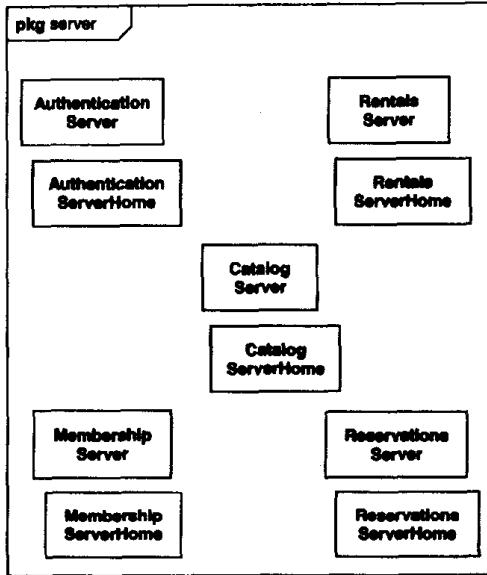


图 B-31 ServerLayer 类

### B.5.6 ServerLayer 的字段列表

服务器对象是无状态的，所以没有字段(所有的 BusinessLayer 类都通过各自的 home 来访问)。

### B.5.7 ServerLayer 的消息列表

对于服务器对象，下面的消息对应于业务服务(这个列表省略了 home，因为它们只是创建服务器对象，且没有参数。)

#### AuthenticationServer

- +logon(n:String, p:String, s:Boolean):long——会员用会员号 n 和密码 p 登录，并指定是否用 s 盗取已有的会话。
- +logoff(i:int)——用会话标识符 i 注销会员。

#### CatalogServer

- +readCategoryNames():String[]——读取每个类别的名称。
- +readMakeNames():String[]——读取所有厂家的名称。
- +readEngineSizes():int[]——读取所有汽车型号的引擎规格(不重复)。
- +readIndexHeadings():String[]——读取派生于所有汽车型号和厂家名称的索引标题。
- +readCarModels(h:String):PCarModelDetails——读取匹配索引标题 h 的所有汽车型号。
- +readCarModelDetails(i:int):PCarModel——读取标识号为 i 的汽车型号的细节。
- +readCarModels(q:PCatalogQuery):PCarModel[]——读取所有匹查询 q 的汽车型号。

#### MembershipServer

- +readMember(i:int):PMember——读取会话标识符为 i 的会员。

- `+changePassword(i:int, o:String, n:String)` —— 使用旧密码 o 和新密码 n, 修改会话标识符为 i 的会员的密码。

### RentalsServer

- `+readRentals(i:int):PRental[]` —— 读取会话标识符为 i 的会员的所有租用信息。

### ReservationsServer

- `+readReservations(i:int):Preservation[]` —— 读取会话标识符为 i 的会员的所有预约信息。
- `+createReservation(i:int, c:int)` —— 为会话标识符为 i 的会员、标识符为 c 的汽车型号创建一个预约。
- `+cancelReservations(i:int, r:int)` —— 为会话标识符为 i 的会员取消标识符为 r 的预约, 只要预约匹配会员即可。

## B.5.8 BusinessLayer 类图

BusinessLayer 的类图如图 B-32 所示。其中的大多数类也显示在分析类模型中(参见 B.3 节), 所以它们的描述放在术语表中(参见 B.8 节), 以避免重复。

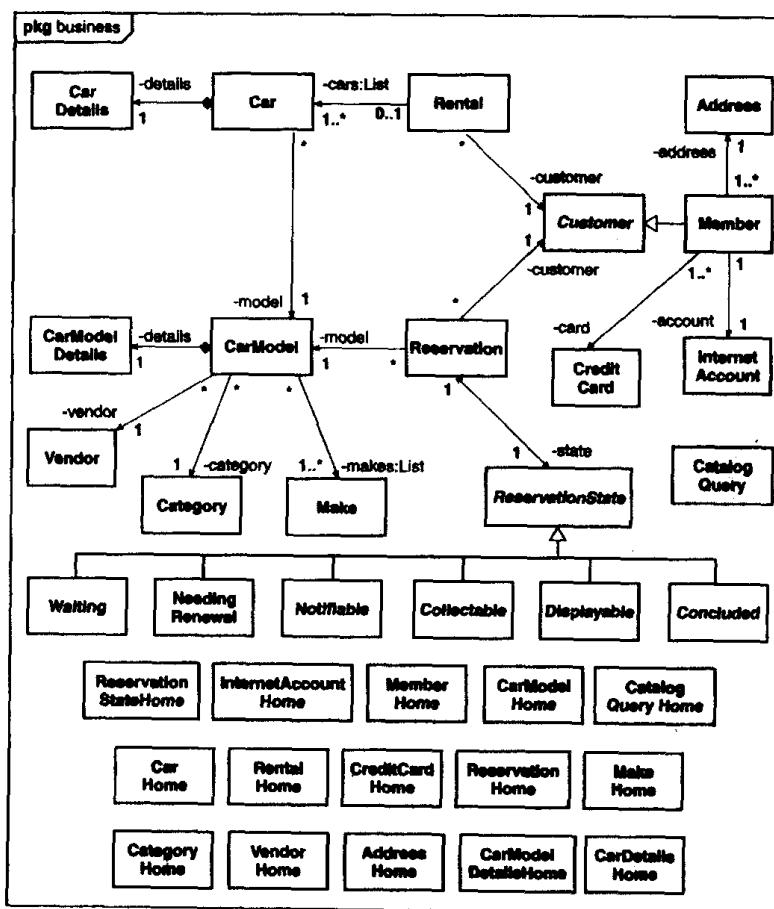


图 B-32 BusinessLayer 类

每个实体类除了 Customer 之外, 都有一个 home(因为它是抽象的)。ReservationStateHome 是创建其子类实例的抽象工厂方法。

Store 类在第 13 章用于演示, 本附录没有介绍它, 因为它在业务服务的实现中不起作用。

## B.5.9 BusinessLayer 的字段列表

下面只列出了用于存储属性的字段，存储链接的字段显示在图 B-32 中。除了状态对象之外，其他的对象都有一个-id:int，用于存储统一标识符，但这个列表省略了它。除了链接之外没有其他字段的类也省略了。

### Address

- -house:String——存储数字和/或名称(在邮政编码中是惟一的)。
- -street:String——房子所在的街道。
- -country:String——街道所在的国家。
- -postCode:String——给部门和地区排序的邮政编码。

### Car

- -traveled:int——汽车的里程数(以千米为单位，从里程表中读取)。
- -dateLost:Date——汽车报失的日期，如果没有丢，就设置为空。

### CarDetails

- -barCode:String——条形码，贴在汽车的挡风玻璃上。
- -numberPlate:String——贴在金属板上的汽车许可号。
- -vin:String——汽车的惟一车辆标识号，固定在车身的一个金属板上。

### CarModel

- -name:String——汽车型号的厂家名称。
- -price:int——租用汽车的日租费(以美分为单位)。

### CarModelDetails

- -engineSize:int——引擎的容量(立方厘米)。
- -description:String——汽车型号的一个描述句子。
- -advert:String——汽车型号的流广告的文件名。
- -poster:String——汽车型号的海报文件名。

### CatalogQuery

- -makeIds>List<Integer>——相关厂家的统一标识符(在建立数据库查询时，使用标识符可避免检索厂家)。
- -carModelIds>List<Integer>——相关汽车型号的统一标识符(使用标识符可避免检索汽车型号)。
- -engineSizes>List<Integer>——引擎规格(立方厘米)。

### Catalog

- -name:String——类别的名称。

**Collectable**

- `-dateNotified:Date`——通知顾客可取车的日期。

**Concluded**

- `-reason:String`——预约结束的原因：“成功租用”、“被顾客取消”、“没有更新”或者“没有取车”。

**CreditCard**

- `-type:String`——信用卡的类型(如“Annex”)。
- `-number:String`——信用卡上的号码。
- `-expiryDate:Date`——信用卡过期的日期。

**Customer**

- `-name:String`——顾客的姓名。
- `-phone:String`——顾客的电话号码。
- `-amountDue:int`——顾客的预期费用(美分，例如超时租用的费用)

**Displayable**

- `-reason:String`——汽车必须放回显示区域的原因：“顾客联系不上”或者“没有取车”。

**InternetAccount**

- `-password:String`——相关会员的密码，至少有 6 个混合了字母和数字的字符。
- `-sessionId:long`——相关会员的会话标识符，如果没有登录，就是 0；如果登录，就是随机的、惟一的非 0 数字。

**Make**

- `-name:String`——厂家的名称。

**Member**

- `-number:String`——会员的惟一号码。
- `-inGoodStanding:boolean`——会员是否有问题，例如怀疑会员迟交费用。

**NeedingRenewal**

- `-dateRenewalNeeded:Date`——预约必须更新以避免自动结束的日期。

**Notifiable**

- `-datePutAside:Date`——汽车移动到保留区域的日期。

### Rental

- `-number:String`——租用信息的惟一号码。
- `-startDate:Date`——开始租用的日期。
- `-dueDate:Date`——租用结束的日期。
- `-totalAmount:int`——租用的费用(以美分为单位)。

### Reservation

- `-number:String`——预约信息的惟一号码。
- `-timestamp:Timestamp`——进行预约的日期和时间。

### Vendor

- `-name:String`——供应商的名称。

### Waiting

- `-lastRenewedDate:Date`——预约最后更新的日期(最初与其创建的日期相同)。

## BusinessLayer 消息列表

BusinessLayer 类上的大多数公共消息都只是属性、派生属性或链接的访问器，所以不给出它们的细节。

对于 ReservationState 层次结构和 Reservation 类，每个类为如图 B-14 所示的每个事件都提供了一个消息，还为每个状态属性提供了一个获取器。为了简短一些，这些消息的细节也省略了。另外，Reservation 类还有测试消息，用于让客户确定接收者处于什么状态，例如 `isConcluded` 和 `isWaiting`。

对于 home，没有给出从单一模式获得的细节。为了支持 home，home 创建的每个类都有一个包构造函数，它把所有的属性和链接作为其参数，这些构造函数没有给出进一步的细节。

除了 `ReservationStateHome` 和 `CatalogQueryHome` 之外，每个 home 都有下述消息(其中 X 表示相应的 BusinessLayer 类)：

- `+findByPrimaryKey(id:int):X`——用统一标识符 id 返回 X 的实例。
- `+create(...):X`——把每个属性和链接作为参数，返回 X 的一个新实例。

`ReservationStateHome` 的每一个子类都有一个 `create` 消息，将所有子类的属性作为参数。

`CatalogQueryHome` 的 `create` 消息有三个 `List<Integer>` 参数：`makeIds`、`categoryIds` 和 `engineSizes`。下面列出在业务服务的实现中使用的其他 home 消息。

### CarModelHome

- `+findByIndexHeading(h:String):List<CarModel>`——返回所有把 h 作为型号或厂家名称的汽车型号，按照型号名称排序。
- `+findByQuery(q:CatalogQuery):List<CarModel>`——根据在 q 中指定的厂家、类别和引擎规格返回汽车型号，按照型号名称排序。

### CarModelDetailsHome

- `+findByCarModelId(id:int):CarModelDetails`——返回匹配统一标识符为 id 的汽车型号的实例。
- `+findEngineSizes():List<Integer>`——按升序返回所有的引擎规格。

### CategoryHome

- `+findCategoryNames():List<String>`——按字母顺序返回所有的类别名称。

### MakeHome

- `+findMakeNames():List<String[]>`——按字母顺序返回所有的厂家名称。

### MemberHome

- `+findByMembershipNumber(n:String):Member`——返回会员号为 n 的会员。
- `+findBySessionId(id:long):Member`——返回会话标识符为 id 的会员。

### RentalHome

- `+findByMember(m:Member):List<Rental>`——返回 m 的所有租用信息，按开始日期排序。

### ReservationHome

- `+findUnconcludedByMember(m:Member):List<Rental>`——返回 m 的所有未结束的预约，按照创建日期排序。

## B.5.10 协议对象的类图

协议对象的类图用于服务器层和服务小程序层之间的通信，如图 B-33 所示。

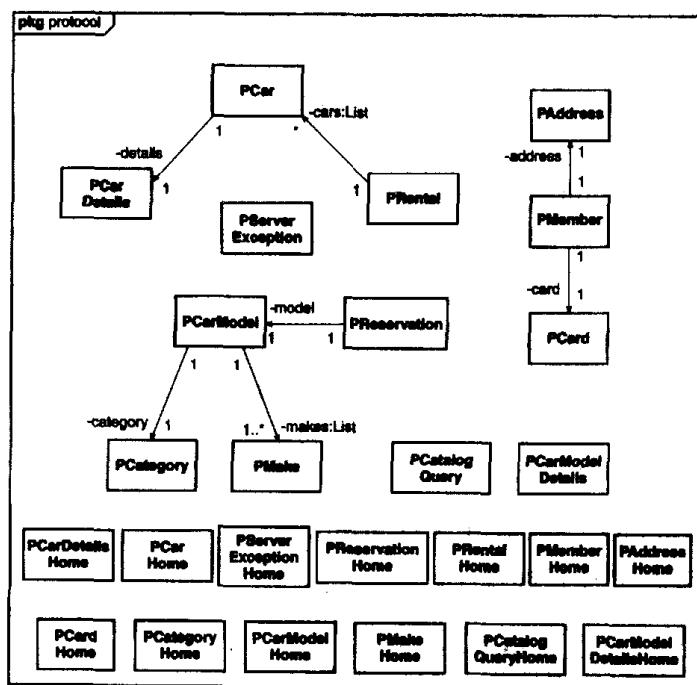


图 B-33 协议类

## 1. 协议对象的字段列表

下面只列出了存储属性的字段，存储链接的字段显示在图 B-33 中。这些字段的含义与 BusinessLayer 类的字段一样，但 cardNumber 只包含最后四个实数。PServerException 的 message 变量存储了异常的解释。

- PAddress -house:String、-street:String、-country:String、-postCode:String
- PCar -traveled:int、-numberPlate:String
- PCarModel -id:int、-name:String、-price:int
- PCarModelDetails -engineSize:int、-description:String、-advert:String、-poster:String
- PCatalogQuery -makeIds:int[]、-carModelIds:int[]、-engineSizes:int[]
- PCategory -name:String
- PCreditCard -type:String、-number:String
- PMake -name:String
- PMember -name:String、-phone:String、-amountDue:int、-inGoodStanding:boolean
- PRental -startData:Date、-dueData:Date
- PReservation -id:int、-number:String、-timestamp:Timestamp
- PServerException -message:String

## 2. 协议对象的消息列表

协议类上几乎所有的消息都是属性或链接的访问器。另外，每个协议类都有一个 `toString` 消息，用于返回人类可理解的接收者汇总，这些消息用于在用户界面上显示对象。

每个 `home` 都有带一个参数的 `create` 消息，它初始化每个属性和链接，除此之外，`home` 上的所有消息都派生于单一模式。为了支持 `home`，每个协议类都有一个包构造函数，它把每个属性和链接作为参数。这些构造函数不进一步讨论。

除了基本的创建消息之外，`PCatalogQueryHome` 还有一个 `createCatalogQuery` 消息，它带三个 `int[]` 参数(`makeIds`、`categoryIds` 和 `engineSizes`)，返回一个新的 `CatalogQuery`。

协议类或 `home` 都没有特别的消息，所以不列出详细的列表。

## B.5.11 数据库模式

数据库模式如图 B-34 所示。在这个图中，主键列的名称显示为黑体，外键列的名称显示为斜体。属性列的含义与 BusinessLayer 类中的字段的属性列相同。在所有的列中，只有 CAR 表中的 DATELOST 列可以为空。

```

ADDRESS (ID:INTEGER, HOUSE:VARCHAR(99), STREET: VARCHAR(99), COUNTRY: VARCHAR(99), POSTCODE:VARCHAR(99))

CAR (ID:INTEGER, TRAVELED: INTEGER, DATELOST:DATE, CARDETAILSID: INTEGER)

CARD (ID:INTEGER, TYPE: VARCHAR(99), NUMBER: VARCHAR(99))

CARDETAILS (ID:INTEGER, BARCODE: VARCHAR(99), NUMBERPLATE: VARCHAR(99), VIN: VARCHAR(99))

CARMPDEL (ID:INTEGER, NAME: VARCHAR(99), PRICE: INTEGER,
CARMODELDETAILSID:INTEGER, CATEGORYID:INTEGER, VENDORID: INTEGER)

CARMPDELDETAILS (ID:INTEGER, ENGINESIZE: VARCHAR(99), DESCRIPTION: VARCHAR(256), ADVERT: VARCHAR(99), POSTER: VARCHAR(99))

CATEGORY(ID:INTEGER, NAME: VARCHAR(99))

COLLECTABLERESERVATION(RESERVATIONID: INTEGER, DATENOTIFIED:DATE)

CONCLUDEDRESERVATION(RESERVATIONID: INTEGER, REASON: VARCHAR(99))

CUSTOMER(ID:INTEGER, NAME: VARCHAR(99), PHONE: VARCHAR(99), AMOUNTDUE: INTEGER)

DISPLAYABLERESERVATION(RESERVATIONID: INTEGER, REASON: VARCHAR(99))

INTERNETACCOUNT(ID:INTEGER, PASSWORD: VARCHAR(99), SESSIONID:INTEGER)

MAKE(ID:INTEGER, NAME: VARCHAR(99))

MAKECARMODEL(CARMODELID:INTEGER, MAKEID:INTEGER)

MEMBER(ID:INTEGER, NUMBER: VARCHAR(99), INDOODSTANDING:BOOLEAN,
CARDID:INTEGER, ADDRESSID: INTEGER)

NEEDINGRENEWALRESERVATION(RESERVATIONID: INTEGER,
DATERENEWALNEEDED:DATE)

NONMEMBER(ID:INTEGER, DRIVERSLICENSE: VARCHAR(99))

NOTIFIABLERESERVATION(RESERVATIONID: INTEGER, DATEPUTASIDE:DATE)

RENTAL(ID:INTEGER, NUMBER: VARCHAR(99), STARTDATE:DATE, DUEDATE:DATE,
TOTALAMOUNT: INTEGER)

RENTALCAR(RENTALID: INTEGER, CARID: INTEGER)

RESERVATION(ID:INTEGER, NUMBER: VARCHAR(99), TIMESTAMP: TIMESTAMP, TIMESTAMP,
CUSTOMERID: INTEGER, CARMODELID: INTEGER)

VENDOR(ID:INTEGER, NAME: VARCHAR(99))

WAITINGRESERVATION(RESERVATIONID: INTEGER, LASTRENEWEDDATE:DATE)

```

图 B-34 数据库模式

## B.5.12 用户界面设计

用户界面的设计这里不给出，因为它类似于 B.3 节中的用户界面草案。

## B.5.13 业务服务的实现

图 B-35 到 B-46 给每个系统用例显示了一个顺序图，说明了业务服务的实现。其中几个图使用图框来表示循环。loop 操作符用伪代码来控制迭代次数。还使用方框把生命线包围起来，使用 ref 操作符表示包含的用例——在这种情况下，包含的用例名显示在方框的中间，并列出了所传送的参数。

为了便于简化，顾客浏览器和服务小程序之间的每次交互都显示为从参与者直接发送给服

务小程序的一个消息。实际上，其实现方式是把一个命令及其参数传送给服务小程序的 doGet 消息。

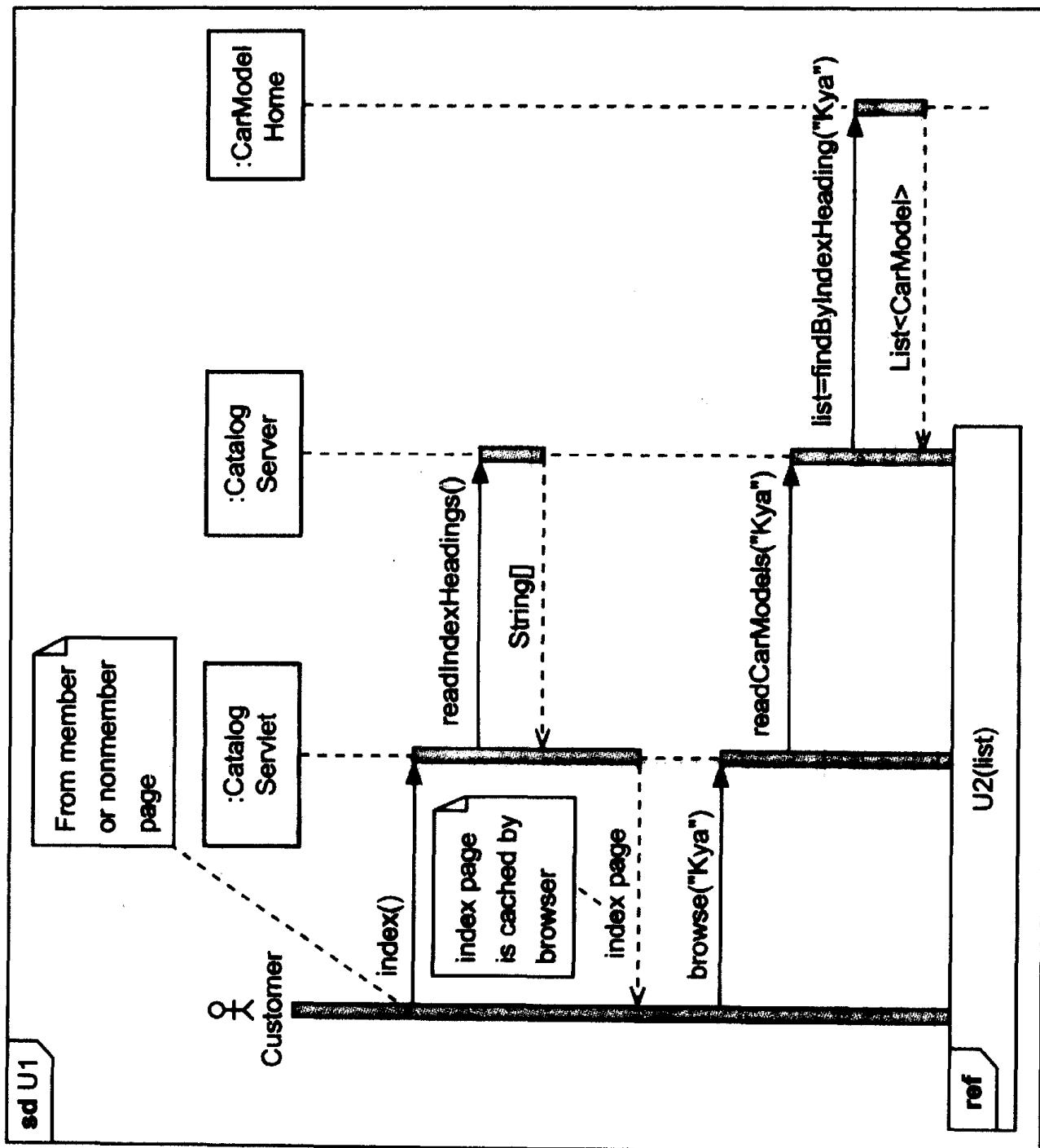


图 B-35 “U1:浏览索引”的顺序图

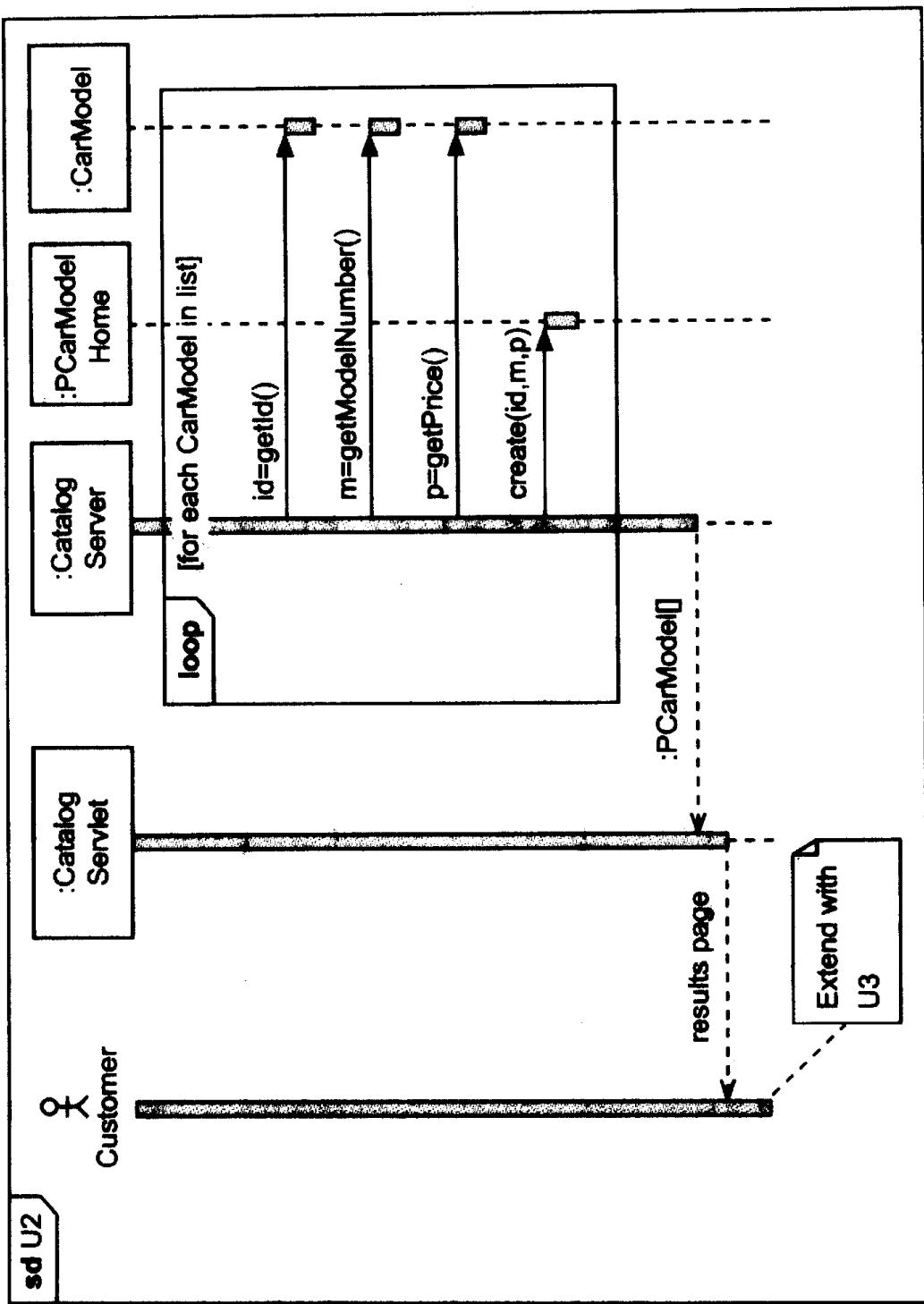


图 B-36 “U2:查看结果”的顺序图

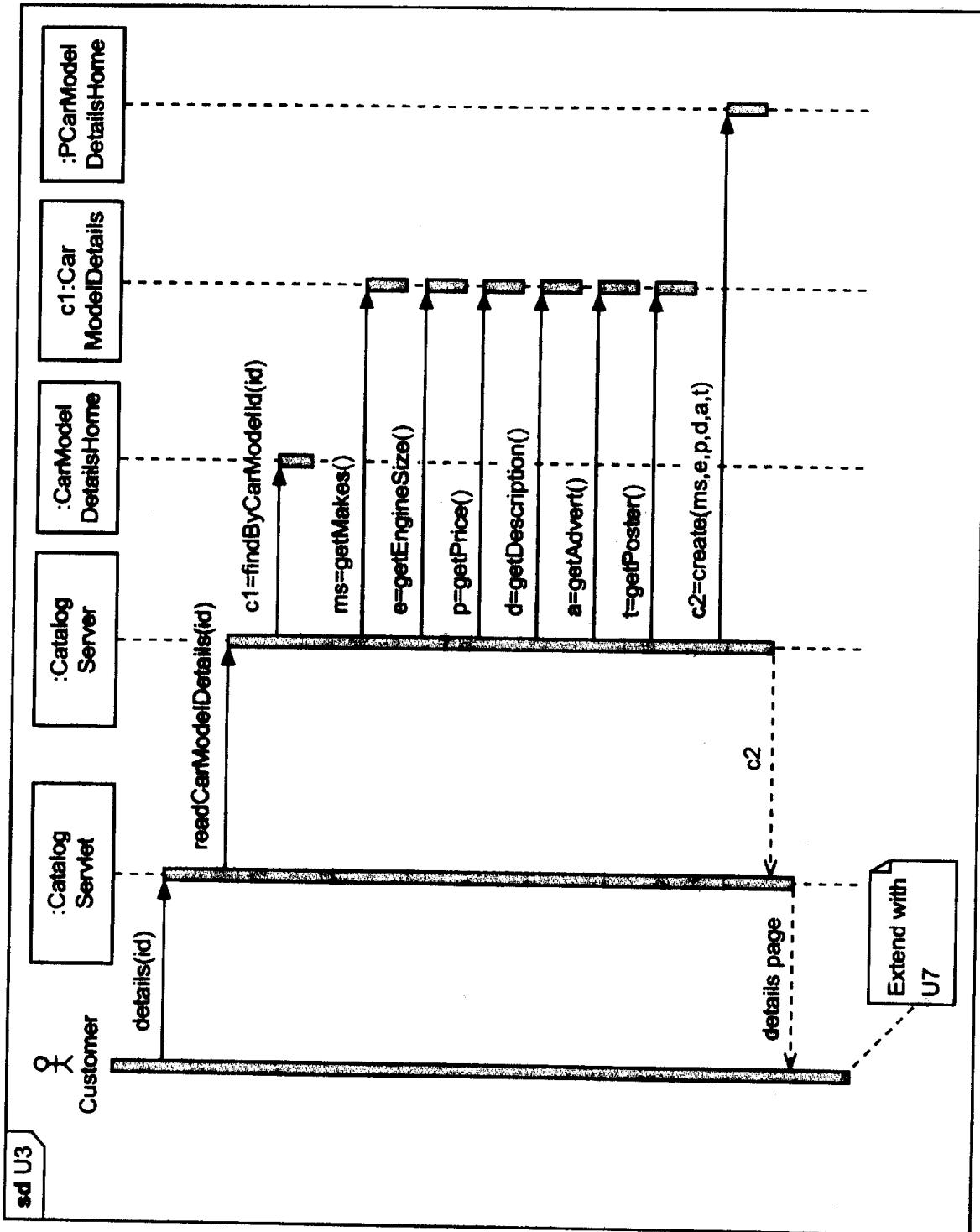


图 B-37 “U3:查看汽车型号细节”的顺序图

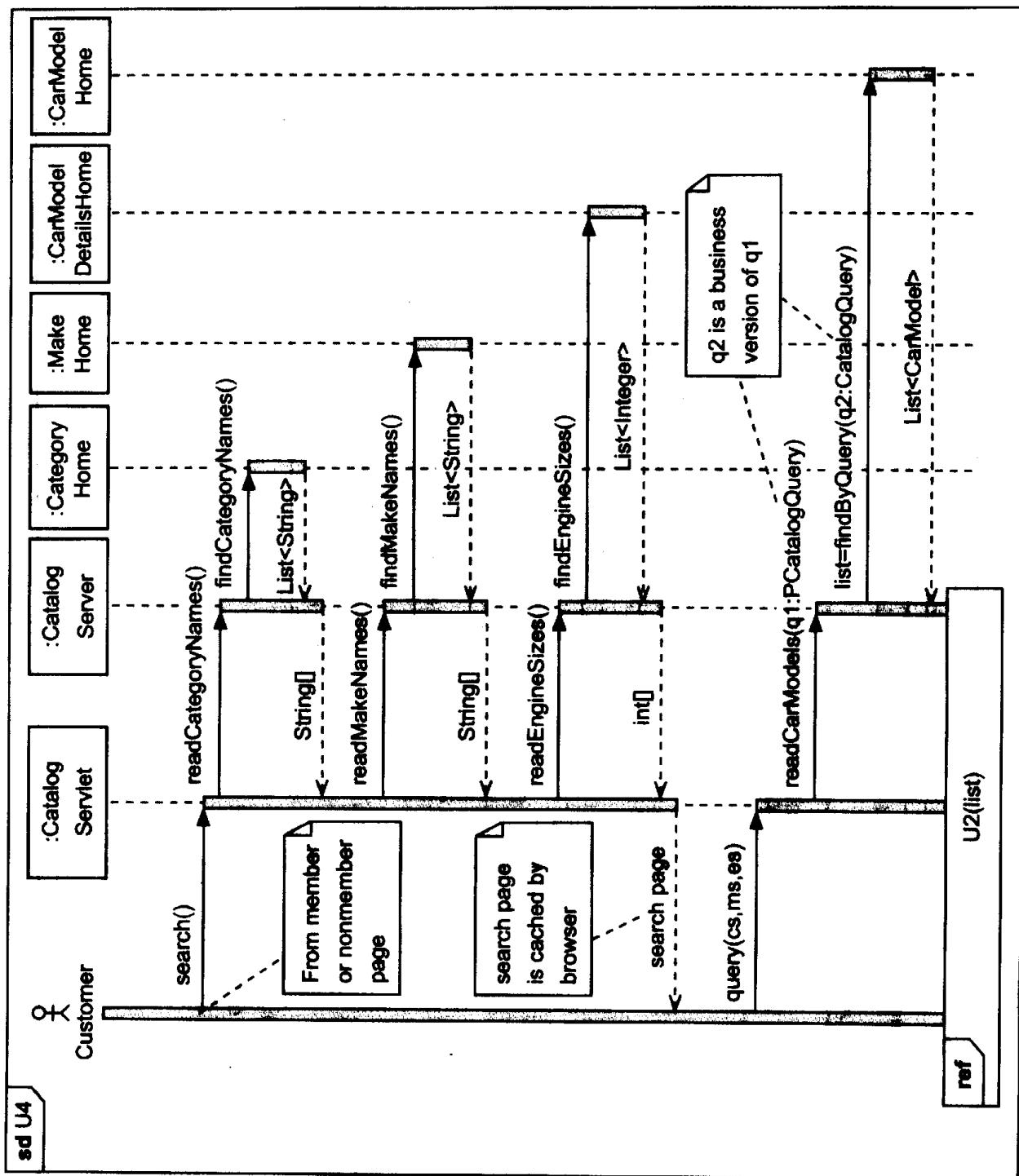


图 B-38 “U4:搜索”的顺序图

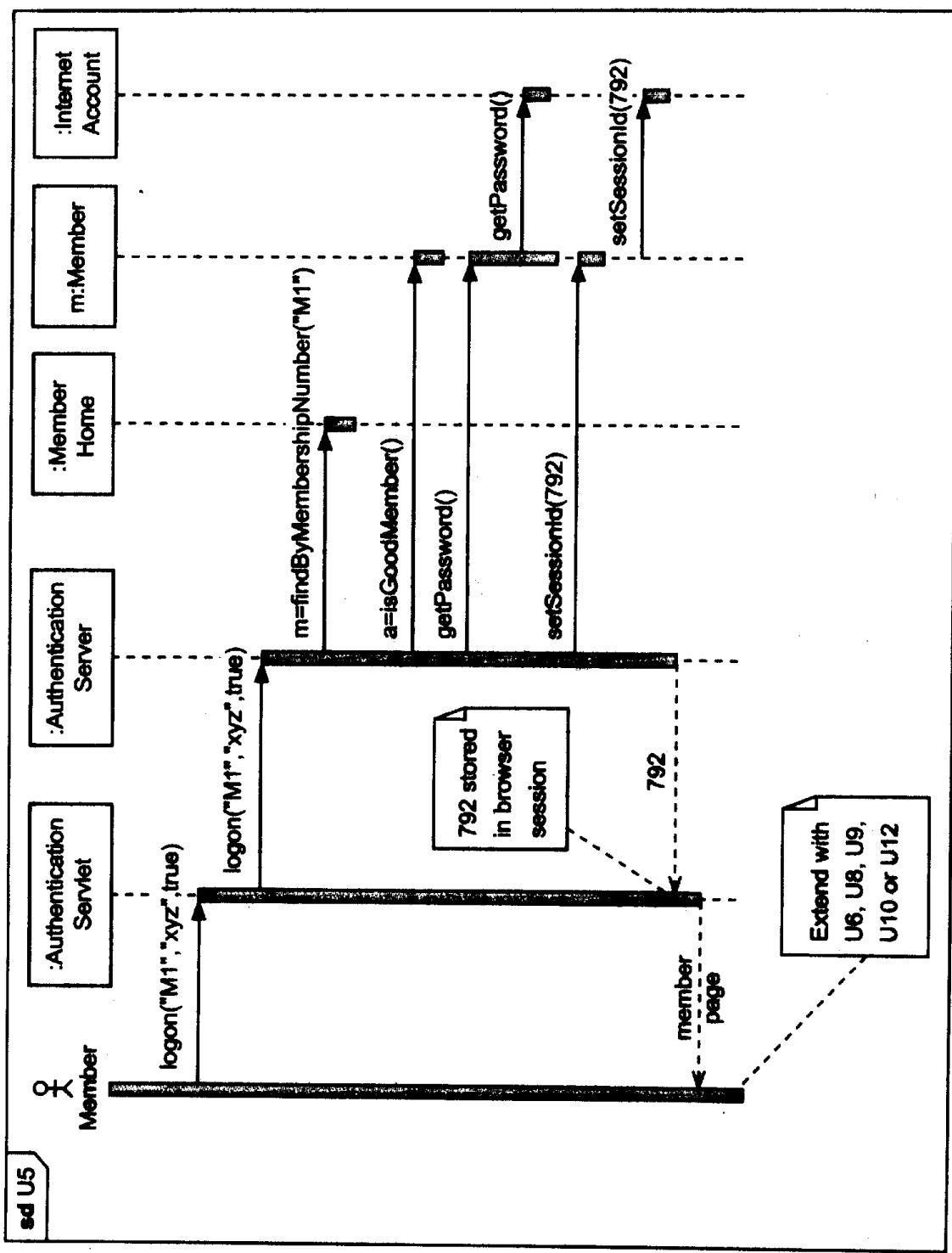


图 B-39 “U5:登录”的顺序图

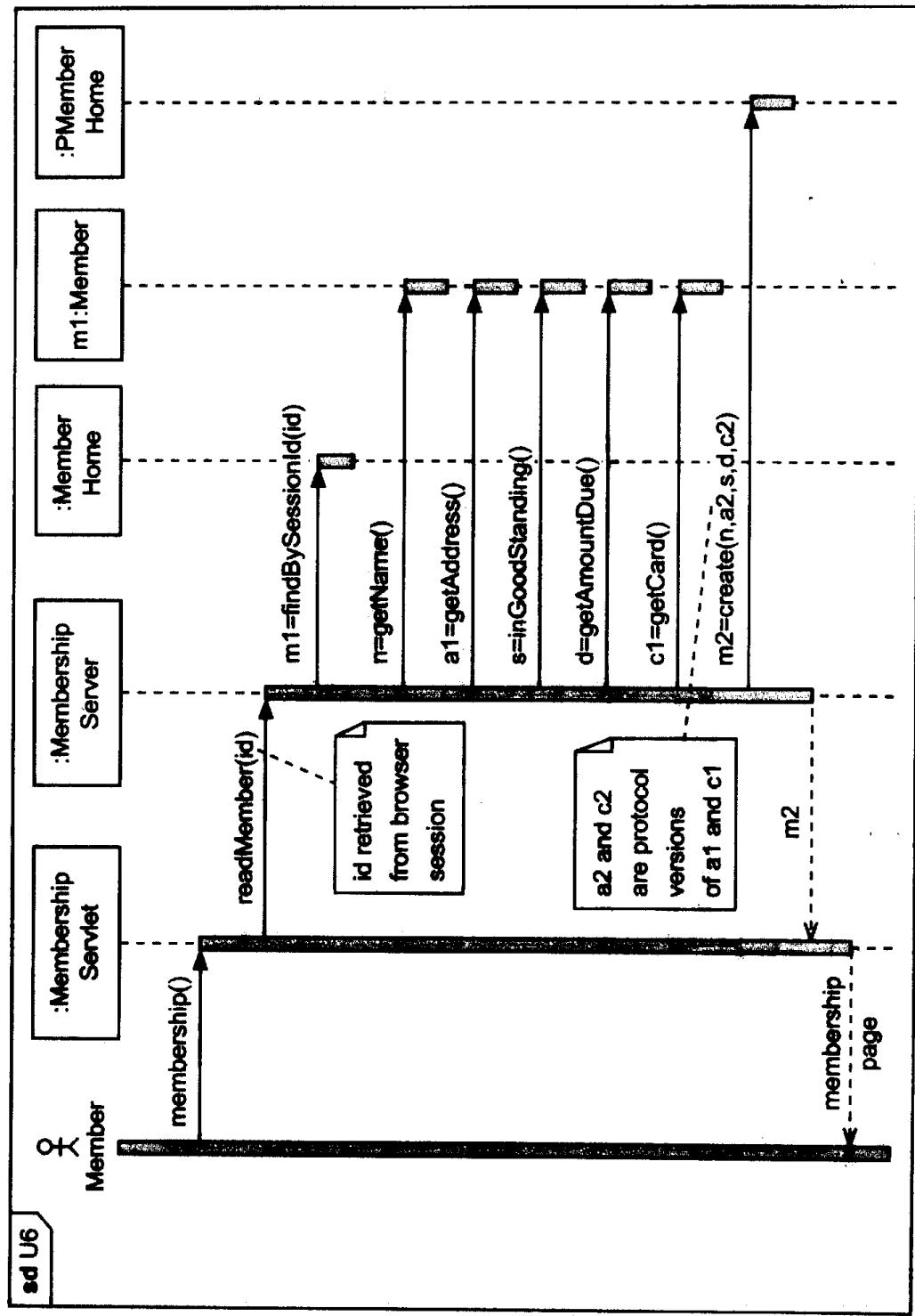


图 B-40 “U6:查看会员信息”的顺序图

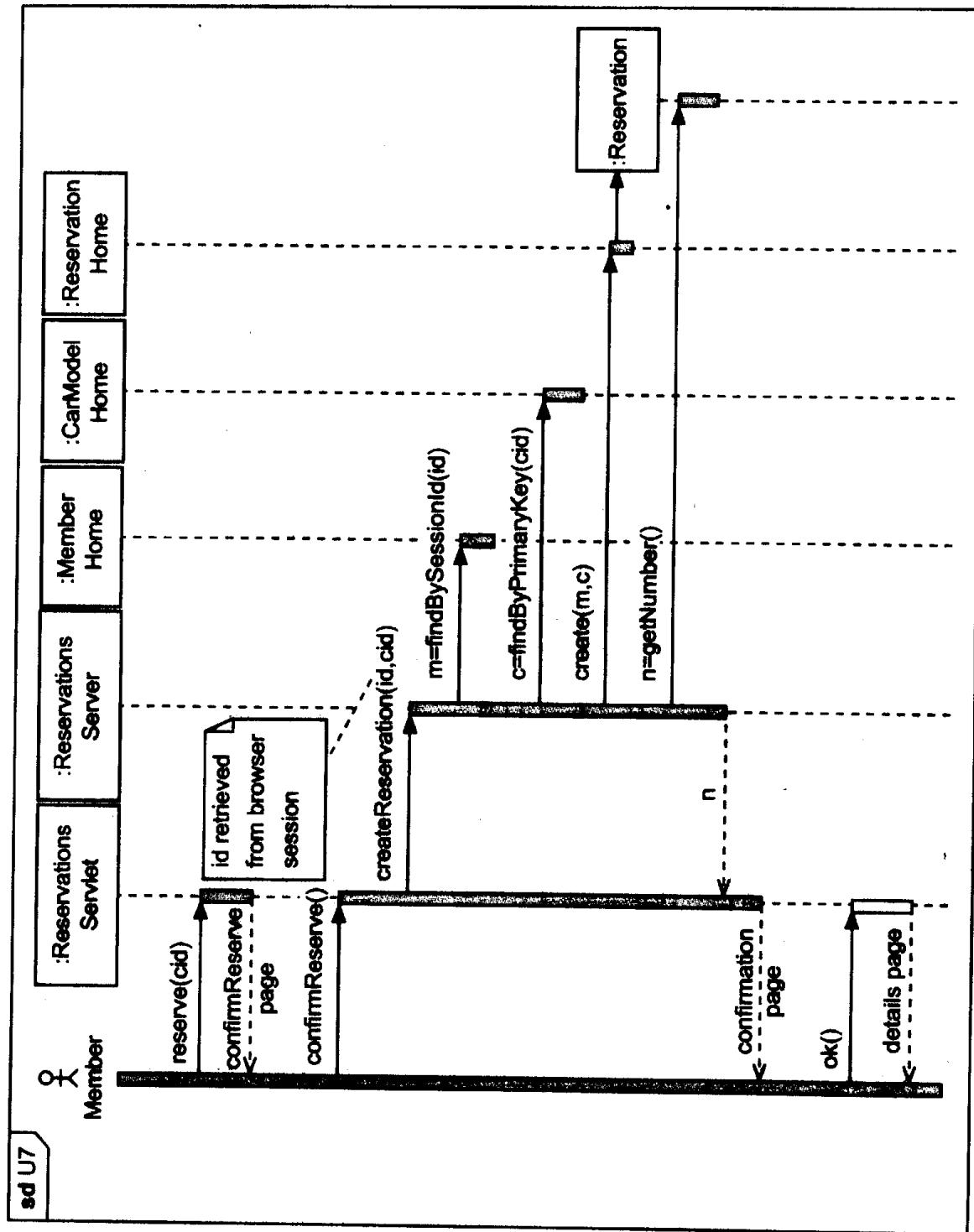


图 B-41 “U7:进行预约”的顺序图

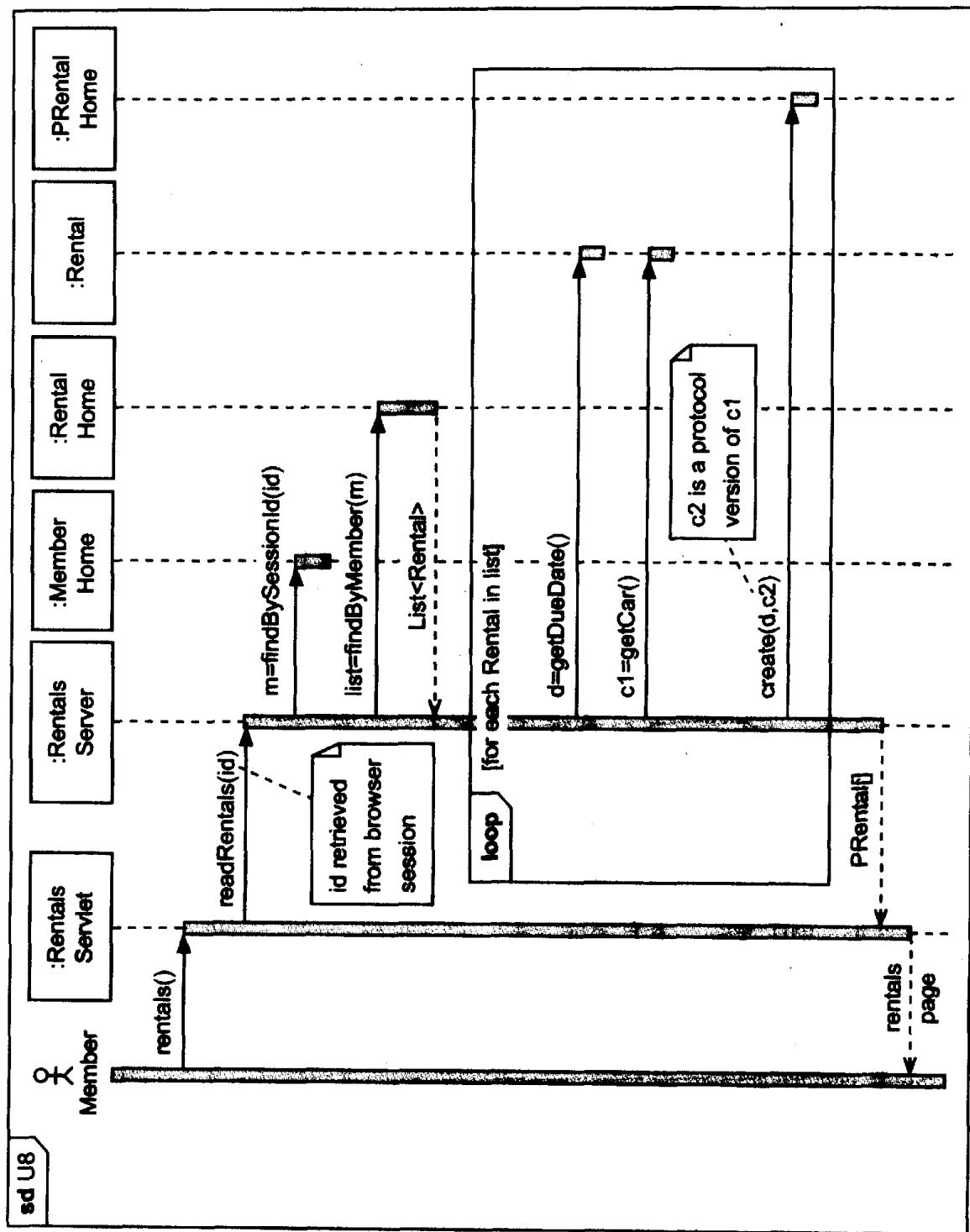


图 B-42 “U8:查看租用情况”的顺序图

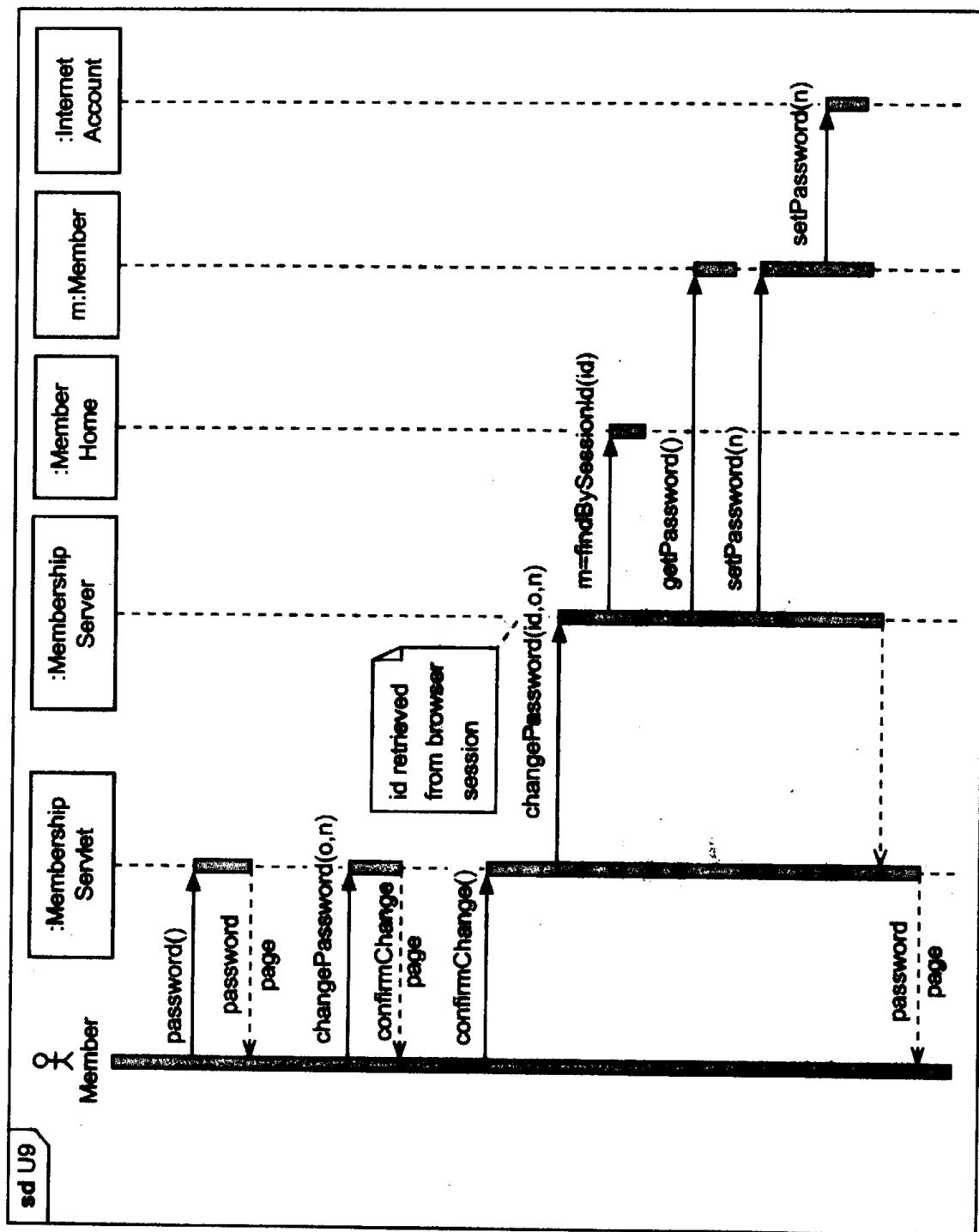


图 B-43 “U9:修改密码”的顺序图

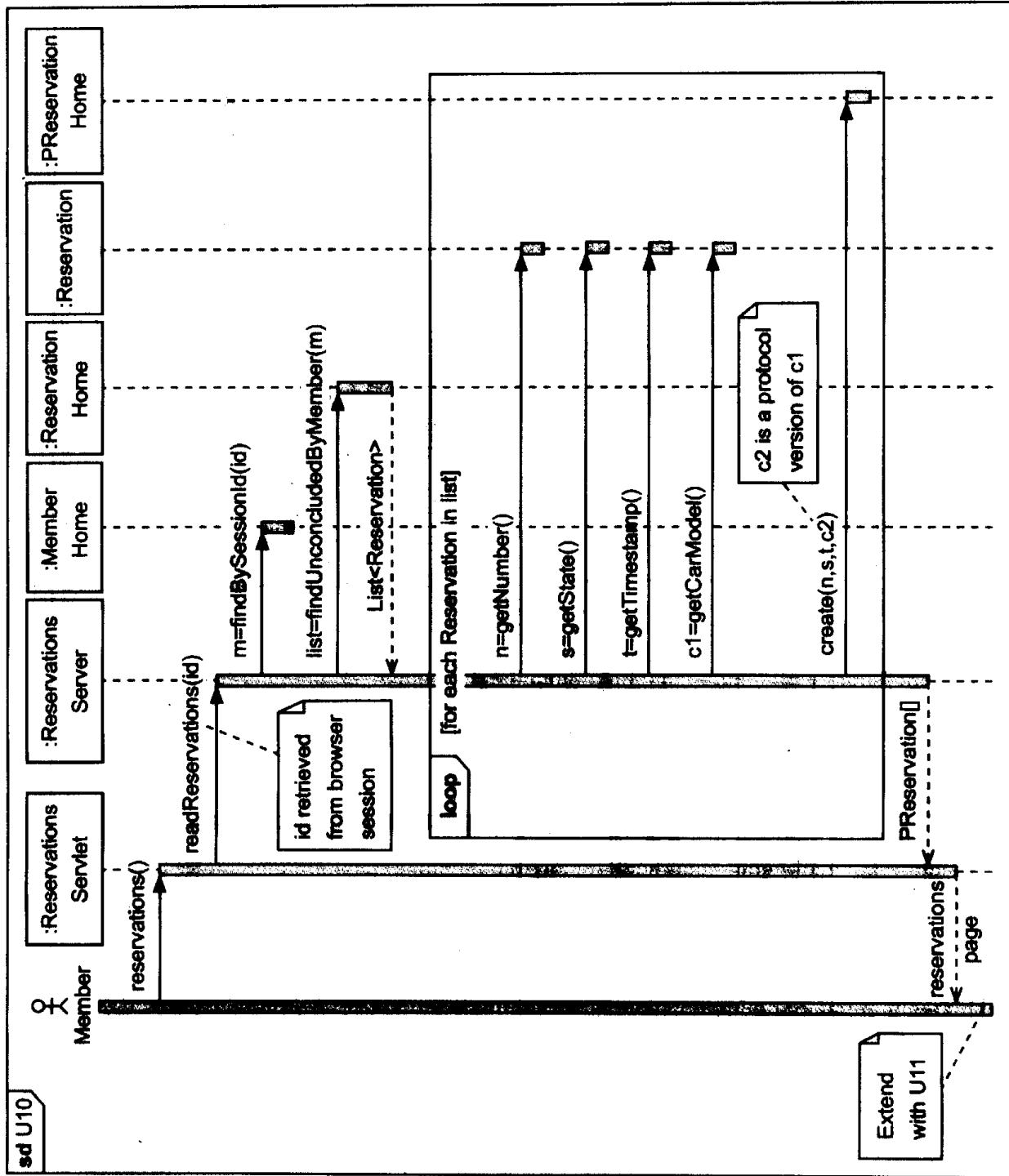
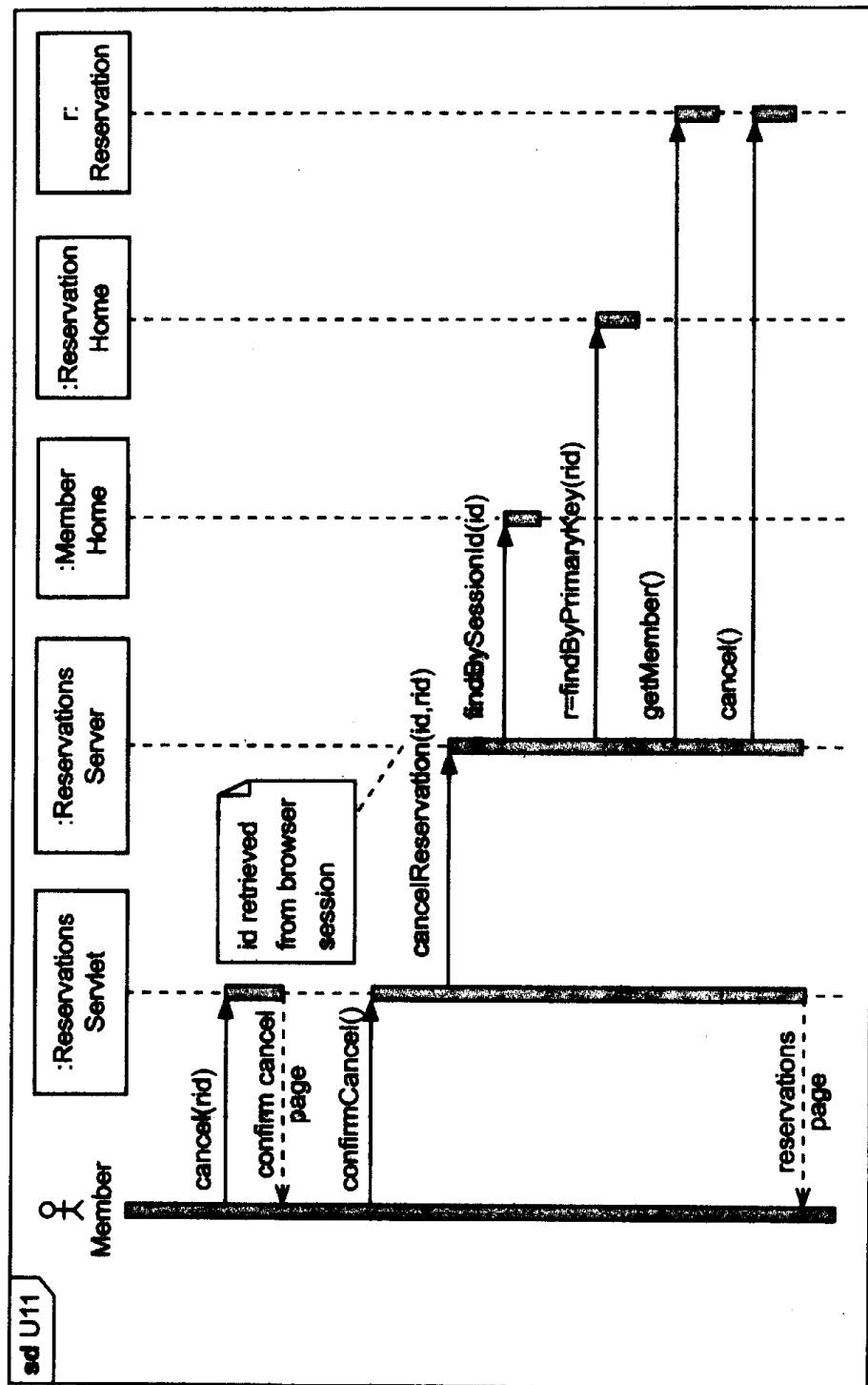


图 B-44 “U10:查看预约情况”的顺序图



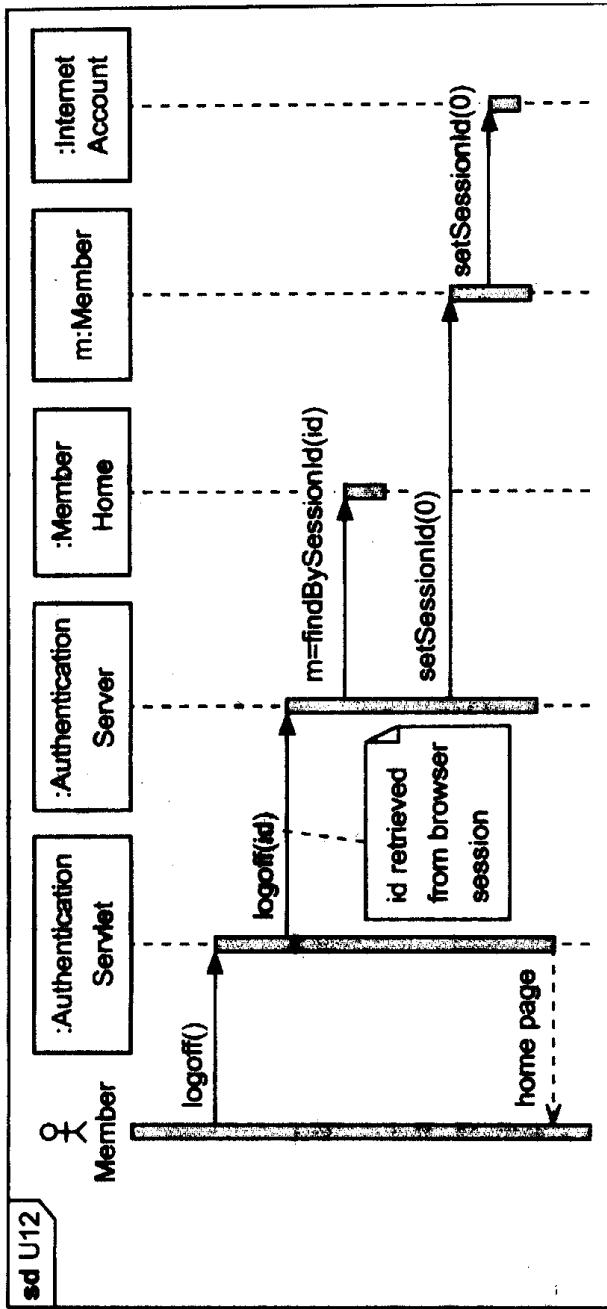


图 B-46 “U12:注销”的顺序图

## B.6 类的规范

本节介绍 iCoot 开发中规范阶段的结果，iCoot 中有许多类，所以下面只介绍一个 ServerLayer 类和一个 BusinessLayer 类的规范，作为例子。完整的规范在源代码中显示为注释。

### B.6.1 服务器类的规范

Invariants: NONE

```
// No invariants, because this is a stateless object
```

Methods:

```
/*
 * Create a reservation for the member with session identifier i
 * and the car model with identifier c
 *
 * Preconditions:
 *   i != 0
 *   for mem = MemberHome.getInstance().findBySessionId(i),
 *     mem != null
 *     mem.isInGoodStanding()
 *     mem.getAmountDue() == 0
 *   c != 0
 *   CarModelHome.getInstance().findByPrimaryKey(i) != null
 *
 * Postconditions:
 *   A new Reservation has been created for Member with session identifier i
 *   and CarModel with identifier c
 *
 * Exceptions:
 *   PServerException (checked) thrown if the server has a problem
 *   IllegalArgumentException (unchecked) thrown if parameters are invalid
 */
public void createReservation(int i, int c)
    throws PServerException;

/*
 * Read all reservations for the member with session identifier i
 *
 * Preconditions:
 *   i != 0
 *   MemberHome.getInstance().findBySessionId(i) != null
 *
 * Postconditions:
 *   result != null
 *   result contains all unconcluded reservations for Member
 *     with session identifier i
 *   result is a new array, exclusive to the client
 *
 * Exceptions:
 *   PServerException (checked) thrown if the server has a problem
 *   IllegalArgumentException (unchecked) thrown if parameters are invalid
 */
public PRReservation[] readReservation (int i)
    throws PServerException;

/*
 * Cancel the reservations with identifier r for the member
 * with session identifier i
 *
 * Preconditions:
 *   i != 0
```

```

* For mem = MemberHome.getInstance().findBySessionId(i),
*   mem != null
* r != 0
* for res = ReservationHome.getInstance().findByPrimaryKey(r)
*   res != null
*   res.getMember() ==mem
*
* Postconditions:
* For res = ReservationHome.getInstance().findByPrimaryKey(r)
*   res.isConcluded()
*   res.getReason().equals("Canceled by customer");
*
* Exceptions:
* PServerException (checked) thrown if the server has a problem
* IllegalArgumentException (unchecked) thrown if parameters are invalid
*/
public void cancelReservation(int i, int r)
    Throws PServerException;

```

## B.6.2 业务逻辑类的规范

Member Class Specification

Invariants:

```

/*
 * Values named in invariants, preconditions and postconditions
 * are attributes, with a getter and an optional setter. Each
 * invariant is an extra precondition for the corresponding setter
 * and an extra postcondition for the corresponding getter.
 */
id is fixed after creation
id != 0
number != null
number.size() != 0
internetAccount != null
address != null

```

Methods:

```

/*
 * Fetch the receiver's id
 *
 * Preconditions: NONE
 * Postconditions:
 *   result == id
 * Exceptions: NONE
 */
Public int getId();

/*
 * Fetch the receiver's number
 *
 * Preconditions: NONE
 * Postconditions:

```

```
* result == number
* Exceptions: NONE
*/
public String getNumber();

/*
 * Set the receiver's number to n
 *
 * Preconditions: NONE
 * Postconditions:
 *   number == n
 * Exceptions: NONE
 */
public void setNumber(String n);

/*
 * Fetch the receiver's internetAccount
 *
 * Preconditions: NONE
 * Postconditions:
 *   result == internetAccount
 * Exceptions: NONE
 */
public InternetAccount getInternetAccount();

/*
 * Set the receiver's internetAccount to ia
 *
 * Preconditions: NONE
 * Postconditions:
 *   internetAccount == ia
 * Exceptions: NONE
 */
public void setInternetAccount(InternetAccount ia);

/*
 * Fetch the receiver's address
 *
 * Preconditions: NONE
 * Postconditions:
 *   result == address
 * Exceptions: NONE
 */
public String getAddress();

/*
 * Set the receiver's address to a
 *
 * Preconditions: NONE
 * Postconditions:
 *   address == a
 * Exceptions: NONE
*/
```

```

        */
    public void setAddress(Address a);

    /*
     * Fetch the session identifier of the receiver's internetAccount
     *
     * Preconditions: NONE
     * Postconditions:
     *   result == internetAccount.getSessionId()
     * Exceptions: NONE
     */
    public int getSessionId();

    /*
     * Set the session identifier of the receiver's internetAccount to i
     *
     * Preconditions: NONE
     * Postconditions:
     *   internetAccount.getSessionId() == i
     * Exceptions: NONE
     */
    public void setSessionId(int i);

```

## B.7 测试计划概述

### B.7.1 引言

iCoot 的测试是不间断的，开发人员、同行、顾客、构建小组和测试小组都要参与进来。

- 开发人员在开发过程中要测试他们的结果。
- 顾客要参与高级产品的验证、接受测试和 beta 测试。
- 同行要评估开发人员开发出来的产品。
- 构建小组负责第一个递增版本之后的构建测试。
- 测试小组负责协调测试过程，包括这个计划的制订和维护、测试阶段本身和系统测试。

这个测试计划概述了要进行的测试，然后指定开发各个阶段的具体测试任务。它不考虑 iCoot 原型的实现。原型的开发在项目经理的监控下，使用快速、非正式的方法进行。

### B.7.2 螺旋式递增方式的作用

在开发的每次螺旋式上升过程中，都由开发人员进行测试。每个制品都要进行同行评估，其附带条件是，正式的同行评估对于第一次螺旋式上升过程是没有必要的。对于第二次之后的螺旋式上升过程，正式的同行评估主要考虑对制品进行的修改，以避免重复。同样，顾客评估也应主要考虑上一次评估之后进行的修改。

在完成了完整的螺旋式上升过程后，测试小组就接管测试阶段，直到发布下一个递增版本为止。

在第一个递增版本之后，衰退测试应确保 iCoot 至少与上一个递增版本有相同的功能。

### B.7.3 非代码制品的测试

用例和 UML 图是由开发小组利用顾客的输入生成的。在早期的螺旋式开发过程中，开发小组的成员会相互评估彼此的工作，并立即修改。

在递增版本发布之前的螺旋式开发过程中，要与不直接参与项目的相关专家进行正式的同行评估。这些评估用于验证制品，从用例到基于类的规范，都要进行测试。

开发人员、同行和项目经理要负责确保制品是相容的。

### B.7.4 代码的评估

在每个递增版本的最后编码阶段之后，要与不参与项目的同行一起进行正式的代码评估。在这些评估过程中，将使用手工的白盒子测试和衡量工具来判断需要进行的调整。

### B.7.5 测试驱动的开发

在设计的实现过程中，程序员将在 JUnit 的帮助下对自己的工作进行不间断的测试。这些程序员开发的测试包括类级别的单元测试和对开发人员的所有类进行的完整性测试。每个开发人员在公开其代码之前，都必须修改代码中的错误。

在每个螺旋式开发过程的最后，开发人员都要使用 JUnit，进行非正式的完整性测试和子系统测试，其目标是在正式的测试阶段之前，尽可能地减少错误。此时，开发小组要在下一个螺旋式过程或测试阶段之前，修改查找出的错误。

### B.7.6 断言

如这个测试计划后面所述，程序员要在代码中添加断言，并在开发过程中使这些断言起作用。在测试阶段，断言最初用于使错误的查找更容易。所有的测试都成功通过后，这些测试就要在禁用断言的情况下再次运行，来检查是否已没有断言在起作用了。它还可以比较代码在有断言和没有断言时的性能。在发布时，断言是禁用的，但仍保留在代码中，这样就可以再次激活它们，以帮助诊断故障。所以，实时系统不会因禁用断言而受损失，并可以采取进一步的措施来保护 iCoot。

为了在禁用断言的状态下保护服务器不被偶然或恶意攻击，要在服务器层上实现应用程序防火墙。这个防火墙一般会强制满足服务器层的前提条件，使检查强制进行。为了减少到达服务器层的不正确的信息量，要在用户界面的下面(在控制层中)放置另一个应用程序防火墙，以拒绝用户的无效请求。这个客户防火墙一般基于控制层的前提条件，但实现了它，就会使检查强制进行。另外，Web 服务器放在非军事区，位于两个常规应用程序防火墙的中间，以拒绝一般的 Internet 攻击。

客户防火墙有两种样式。对于基于 HTML 的客户，防火墙在服务小程序内部实现，并使用防止无效请求的标准 Web 技术(例如，不会在 Web 页面上进行无效选择，使用 JavaScript 检查输入数据)。对于基于 GUI 的客户，防火墙应在控制层上实现，使用标准的 GUI 技术来防止无效请求(例如禁用不可用的按钮，用下拉列表和数字按钮替代文本输入项)。

### B.7.7 测试阶段

每个递增版本的螺旋式开发结束后，就把代码转交给测试小组。在正式的测试开始之前，测试小组会再次进行开发小组已进行过的单元测试和完整性测试，以确保没有已知错误。测试

小组使用类和子系统规范来生成子系统测试案例和相关的测试过程。系统测试案例应建立在系统用例的基础上。每个测试案例都包含许多单个的测试，每个测试都有测试名称、测试描述、测试过程和期望的结果。

测试小组要进行如下测试：

- 负载测试：平均负载和最大负载。
- 渗透测试：验证资源是否会随时间的推移而崩溃或用尽。
- 应力测试：确认 iCoot 会优雅地失败。
- 安全测试。

只要可能，就使测试自动化，降低测试的成本。

同时，测试小组要在室内志愿者的帮助下，小心地选择顾客，组织接受测试。

接着使用测试台进行安装测试，该测试台应包括目标平台的一个重要子集。在发布前，在选中的顾客站点上进行 beta 测试。只要可行，开发小组就在发布前修改所有已知的错误。

收集性能衡量标准(例如平均事务时间)和样式衡量标准(例如方法的大小)。在发布前必须修改不能接受的性能缺陷。样式问题应记录下来，作为下一个递增版本的输入。

## B.7.8 说明文档的测试

在第一个螺旋式开发过程中，说明文档小组就开始编写手册和培训教材。这些都要进行同行评估。在几次螺旋式开发之后，每个递增版本发布之前的测试阶段要再次测试说明文档。这些测试包括同行评估、接受测试和 beta 测试。

## B.7.9 构建测试

程序员使用源代码管理工具来确保，所有的代码都保存在一个中心位置。他们要检查用于工作的代码，以避免重复或重叠。项目经理负责确定谁完成什么工作，构建小组负责管理代码库。为了处理开发人员缺席的情况，代码管理工具要配置为允许在受控条件下检查已通过测试的制品。

在发布第一个递增版本之后，后续的开发都要由构建小组进行每天的构建测试。这涉及到构建整个系统，运行系统测试的重要子集。

## B.7.10 测试建档和记录日志

所有的测试案例都要建档，在测试小组的控制下保存在测试库中。在进行测试时，测试结果都要记录，并添加到库中。

为了鼓励开发人员在测试阶段之前测试他们的代码，开发人员不必在测试库中记录测试失败的情况。因为开发人员的测试案例是用 Java(使用 JUnit)编写的，所以可以存储在代码库中；于是，不需要把这些测试案例添加到测试库中。

在测试阶段，测试失败的情况要添加到库中。测试小组要与项目经理合作，确保把补丁分配给开发小组的成员，并完成测试。

项目经理负责确保开发足够的 JUnit 测试案例，并常常运行它们。为了便于报告，开发小组必须在每个螺旋式开发过程的最后记录已进行的完整性测试，并把相应的项添加到测试库中。

## B.7.11 分阶段的测试活动

下面是开发的各个阶段进行的测试活动。第一个螺旋式开发过程可以放松正式同行评估的

要求。同行应从专家中选择。

- 需求阶段：业务用例、用户界面草图、用例图、系统用例、用例优先级、辅助需求和活动图由开发人员、同行和顾客来评估。
- 分析阶段：分析类图、状态机图、通信图由开发人员、同行和顾客来评估。
- 系统设计阶段：部署图、技术选择、层图、层交互策略、并发策略和安全策略由开发人员和同行来评估。
- 子系统设计阶段：设计类图、数据库模式、用户界面设计(用于对服务小程序和 JSP、应用小程序、应用程序和移动设备的接口的 HTML 访问)和顺序图由开发人员和同行来评估。
- 规范阶段：对于每个类，都要指定前提条件、后置条件和类不变式。这些断言由开发人员和同行来评估。子系统规范可以在适当的时候使用，其测试方式与类规范类似。
- 实现阶段：这个阶段的测试包括三个部分：
  - 给方法添加断言(每个类规范都添加)。其他类型的断言是可选的，但推荐使用，例如检查循环结束，避免不可能的条件。
  - 创建 JUnit 测试案例和 JUnit 测试套件。每个类至少有一个测试案例(这可能涉及一些完整性测试)，每个包至少有一个测试套件。
  - 由开发人员和同行进行代码评估。
- 测试阶段：测试阶段由测试小组负责。测试小组开发的测试案例和测试过程要经过同行评估。测试包括：
  - JUnit 测试，确保修改所有已知的错误。
  - 子系统测试，基于在设计和规范阶段指定的子系统接口。
  - 系统测试，基于用例(功能测试、平均负载和最大负载的负载测试)
  - 应力测试：确认系统会优雅地失败(像在设计和规范阶段定义的那样失败)
  - 安全测试(未经授权，入侵 iCoot)
  - 接受测试，一般基于生产率标准
  - 衡量标准，基于系统性能和编码样式
  - 说明文档测试，由最终用户和系统管理员帮助完成
  - 安装测试，使用目标环境的重要子集
  - beta 测试，在选中的顾客站点上
- 维护阶段：测试小组在项目经理和开发小组的帮助下，负责管理报告、修改发布后发现的错误。在递增版本中，由测试小组和项目经理决定是否修改错误，进行衰退测试，最后发布。衰退测试包括：
  - JUnit 测试
  - 子系统测试
  - 系统测试
  - 安装测试

顾客对改进 iCoot 的反馈将传送给项目经理库，并合并到下一个递增版本中。

## B.8 术语表

术 语	定 义
Address(业务对象, 系统对象, 分析对象, 设计对象)	会员的地址
AddressHome(设计对象)	创建和查找 Address 对象的函数
Assistant(业务参与者, 系统参与者)	商店中的雇员, 帮助顾客租用 Car 对象, 预约汽车型号
Auk(业务参与者)	已有的系统, 处理顾客的信息、预约信息、租用信息和可用汽车型号的类别
AukInterface(分析对象)	访问 Auk 的边界
AuthenticationServer(设计对象)	控制会员在 iCoot 上的登录和注销
AuthenticationServerHome(设计对象)	创建 AuthenticationServer 的函数
AuthenticationServlet(设计对象)	使 AuthenticationServer 可以在 Web 浏览器的 HTML 页面上访问
BusinessLayer(设计层)	包含把 PersistenceLayer 转换为面向对象应用程序对象的对象
Car(业务对象, 系统对象, 分析对象, 设计对象)	商店用于出租的 CarModel 的实例
CarDetails(分析对象, 设计对象)	Car 的额外信息, 例如号牌和 VIN
CarDetailsHome(设计对象)	创建和查找 CarDetails 的函数
CarHome(设计对象)	创建和查找 Cars 的函数
CarModel(业务对象, 系统对象, 分析对象, 设计对象)	Catalog 中的一个型号, 可用于预约
CarModelDetails(分析对象, 设计对象)	CarModel 的额外信息, 例如广告和海报
CarModelDetailsHome(设计对象)	创建和查找 CarModelDetails 的函数
CarModelHome(分析对象, 设计对象)	创建和查找 CarModel 的函数
Catalog(业务对象)	描述可用于租用的 CarModel 的文档
CatalogQuery(设计对象)	会员在线搜索 iCoot 上的 Catalog 时感兴趣的汽车型号规范; 包括类别、厂家或引擎规格
CatalogQueryHome(设计对象)	创建 CatalogQuery 对象的函数
CatalogServer(设计对象)	控制对可通过 iCoot 浏览或预约(只限于会员)的 CarModel 的访问
CatalogServerHome(设计对象)	创建 CatalogServer 对象的函数
CatalogServlet(设计对象)	使 CatalogServer 可在 Web 浏览器的 HTML 页面上访问
Category(分析对象, 设计对象)	汽车的分类, 有助于顾客查找信息, 例如 Sports 或 Luxury
CategoryHome(设计对象)	创建和查找 Category 对象的函数

(续表)

术 语	定 义
Collectable(设计对象)	一个预约状态, 表示已通知顾客有一个匹配的预约, 但还没有取车
com::nowhere::business(设计包)	包含 BusinessLayer 类的包
com::nowhere::control(设计包)	包含 ControlLayer 类的包
com::nowhere::micro(设计包)	包含 MicroLayer 类的包
com::nowhere::persistence(设计包)	包含 PersistenceLayer 类的包
com::nowhere::protocol(设计包)	包含协议类的包, 这些类由 ServletsLayer、RMILayer 和 ControlLayer 用于与 ServerLayer 通信
com::nowhere::rmi(设计包)	包含 RMILayer 类的包
com::nowhere::server(设计包)	包含 ServerLayer 类的包
com::nowhere::servlets(设计包)	包含 ServletsLayer 类的包
com::nowhere::swing(设计包)	包含 SwingLayer 类的包
Concluded(设计对象)	一个预约状态, 表示预约已结束, 因为顾客已取走了车, 或者预约超时了, 或取消了
CootBusinessServer(设计节点)	包含 CootServer 上的 iCoot 服务的过程
CootGUIClient(设计节点)	顾客的机器, 其中安装了 J2SE 或 J2ME GUI, 用于通过 RMI 访问 iCoot
CootHTMLClient(设计节点)	顾客的机器, 其中安装了 Web 浏览器, 用于访问 iCoot
cootschema.ddl(部署制品)	用于为 Coot 生成数据库表的脚本
CootServer(设计节点)	安装了 WebServer 和 CootBusinessServer 的机器
CreditCard(业务对象, 系统对象, 分析对象, 设计对象)	用于确认会员的信用卡是否可用; 该卡不能过期
CreditCardCompany(业务参与者)	发放信用卡和确认其有效性的公司
CreditCardHome(设计对象)	创建和查找 CreditCard 的函数
Customer(业务参与者, 业务对象, 系统参与者, 系统对象, 分析对象, 设计对象)	给一个标准服务付费的人
DBMS	包含关系数据库管理系统的过程
DBServer	安装了 DBMS 的机器
DebtDepartment(业务对象)	处理欠费的部门
Displayable(设计对象)	一个预约状态, 表示状态为 Collectable 的预约超时或取消了; 即位于保留区域的汽车必须移动到显示区域
EJB	表示 Enterprise JavaBean, 它是标准 Java 框架中的一个对象, 可以处理事务、网络访问和数据库访问, 该对象在 Internet 防火墙的后面。它有两个变体: 会话 Bean(用于远程访问业务服务)和实体 Bean(用于自动映射数据库中的数据)

(续表)

术 语	定 义
HTMLLayer(设计层)	访问 ServletsLayer 的客户端代码, 由标准的 HTML Web 浏览器提供
HTTPCGILayer(设计层)	标准的网络层, 位于 HTMLLayer 和 ServletsLayer 之间
iCoot.ear(部署制品)	Java 企业归档, 包含 CootHTMLClient(最终是 CootGUIClient)使用的服务小程序、JSP 和 EJB
iCoot(部署制品)	包含 iCoot 站点中静态 HTML 的文件夹
IllegalArgumentException	标准 Java 异常, 表示试图用无效的参数发送消息
InternetAccount(设计对象)	会员登录到 iCoot 上所需要的信息以及其登录状态的记录
InternetAccountHome(设计对象)	创建和查找 InternetAccount 对象的函数
J2EE	Java 2 平台的企业版
J2ME	Java 2 平台的 Micro 版
J2SE	Java 2 平台的标准版
JDBC	标准 Java 库, 提供以统一的方式对所有关系数据库的访问
JDBCLayer(设计层)	从 Java 中访问关系数据库(在 EJB 框架内)的层
JRMP	RMI 使用的通信协议
JSP	Java Server Page, 动态的 Web 页面, 其中包含由服务器执行的 Java 代码和静态的 HTML
Keys(业务对象)	用于操作 Car, 顾客在租用汽车时有备用钥匙; 商店保存可用汽车的钥匙, 在所有汽车的钥匙; 如果所有的备用钥匙都丢失或损坏了, 商店还有厂家提供的复制系列号
LegalDepartment(业务参与者)	处理涉及所租出汽车的事故的部门
License(业务对象)	租用汽车必须有的一个文档, 该文档还可以作为非会员进行预约时的身份证件
LogonController(分析对象)	iCoot 控制器, 控制会员的登录和注销
Make(分析对象, 设计对象)	每辆汽车都有一个或多个制造它的厂家
MakeHome(设计对象)	创建和查找 Make 的函数
MemberHome(分析对象, 设计对象)	创建和查找 Member 的函数
MembershipCard(业务对象)	商店给会员发放的卡, 作为会员的凭证
MembershipServer(设计对象)	控制通过 iCoot 对会员信息的访问, 例如地址和信用卡
MembershipServerHome(设计对象)	创建 MembershipServer 的函数

(续表)

术 语	定 义
MembershipServlet(设计对象)	使 MembershipServer 可以在 Web 浏览器的 HTML 页面中访问
MemberUI(分析对象)	会员用于访问系统的 iCoot 边界
MicroLayer	使用 Java 2 Micro Edition 在外围设备(如移动电话或机顶盒)上访问 RMILayer 的对象, 为 iCoot 的未来版本保留的对象
NeedingRenewal(设计对象)	一个预约状态, 表示预约在一星期内未找到匹配, 要使该预约不过期, 就必须更新
NonMember(业务参与者, 业务对象, 系统参与者, 分析对象, 设计对象)	未检查身份和信用卡有效性的顾客, 所以该顾客必须交押金, 才能进行预约, 或者提供驾照的副本, 才能租用汽车
NonMemberUI(分析对象)	非会员用于访问系统的 iCoot 边界
Notifiable(设计对象)	一个预约状态, 表示预约匹配一个可用的汽车, 但还没有通知会员
PAddress(设计对象)	Address 的协议版本
PAddressHome(设计对象)	创建 InternetAccount 的函数
PCar(设计对象)	Car 的协议版本
PCarHome(设计对象)	创建 PCar 对象的函数
PCarModel(设计对象)	CarModel 的协议版本
PCarModelDetails(设计对象)	CarModelDetails 的协议版本
PCarModelDetailsHome(设计对象)	创建 PCarModelDetails 的函数
PCarModelHome(设计对象)	创建 PCarModel 的函数
PCatalogQuery(设计对象)	CatalogQuery 的协议版本
PCatalogQueryHome (设计对象)	创建 PCatalogQuery 对象的函数
PCategory(设计对象)	Category 的协议版本
PCategoryHome (设计对象)	创建 PCategory 的函数
PCreditCard(设计对象)	CreditCard 的协议版本
PCreditCardHome(设计对象)	创建 PCreditCard 的函数
PMake(设计对象)	Make 的协议版本
PMakeHome (设计对象)	创建 PMake 的函数
PMember(设计对象)	Member 的协议版本
PMemberHome (设计对象)	创建 PMember 的函数
PRental(设计对象)	Rental 的协议版本
PRentalHome (设计对象)	创建 PRental 的函数
PReservation (设计对象)	Reservation 的协议版本

(续表)

术 语	定 义
PReservationHome(设计对象)	创建 PReservation 的函数
PServerException(设计对象)	表示 ServerLayer 中的一个对象不能完成请求, 因为(例如)有一个数据库问题
PServerExceptionHome(设计对象)	创建 PServerException 的函数
Rental(业务对象, 系统对象, 分析对象, 设计对象)	Nowhere Cars 和顾客之间的协议, 在某个阶段保有一辆或多辆汽车, 如果汽车没有及时返还, 就要进行罚款
RentalHome(分析对象, 设计对象)	查找和创建 Rental 的函数
RentalServerHome(设计对象)	创建 RentalServer 的函数
RentalsServer(设计对象)	控制会员通过 iCoot 对租用信息的访问
RentalsServlet(设计对象)	使 RentalsServer 可以在 Web 浏览器的 HTML 页面中访问
Reservation(业务对象, 系统对象, 分析对象, 设计对象)	顾客预约一种汽车型号
ReservationHome(分析对象)	查找和创建 Reservation 的函数
ReservationServerHome(设计对象)	创建 ReservationServer 的函数
ReservationServer(设计对象)	控制会员通过 iCoot 对预约信息的访问
ReservationServlet(设计对象)	使 ReservationServer 可以在 Web 浏览器的 HTML 页面中访问
ReservationsSlip(业务对象)	详细记录会员号、汽车型号、时间戳和预约号的卡片
ReservationState(分析对象, 设计对象)	预约的状态, 例如 Waiting 或 Concluded
ReservationStateHome(设计对象)	创建 ReservationState 的函数
RMI	通过网络把消息发送给对象的标准 Java 机制
RMILayer(设计层)	把 ServerLayer 转换为可由 SwingLayer 或 MicroLayer 访问的简单对象; 为 iCoot 的未来版本而保留
ServerLayer	包含通过网络访问 iCoot 的对象
ServletsLayer(设计层)	服务器端对象, 可以从 HTML Web 浏览器上访问 iCoot
Store(业务参与者, 设计对象)	Nowhere Cars 站点, 可以在该站点上租用汽车、预约汽车型号, 浏览或获得目录
SwingLayer(设计层)	使用标准 Java 库从 Java GUI 上访问 RMILayer 的对象, 为 iCoot 的未来版本而保留
Vendor(分析对象, 设计对象)	提供一辆或多辆汽车的公司
VendorHome(设计对象)	创建和查找 Vendor 对象的函数
VIN	车辆标识号, 由许可权威机构发放的惟一号码, 贴在车身的一个牌子上
Waiting(设计对象)	一个预约状态, 表示预约已通过 Internet 完成, 但还没有满足顾客的需要、取消或过期

(续表)

术 语	定 义
WebBrowser(设计节点)	提供对 CootHTMLClient 的 HTML 访问的过程
WebServer(设计节点)	提供从 Web 浏览器上对 CootBusinessServer 的服务器端访问的过程

# 附录 C UML 表示法小结

本附录总结本书使用的 UML 表示法，按图的类型来排列：

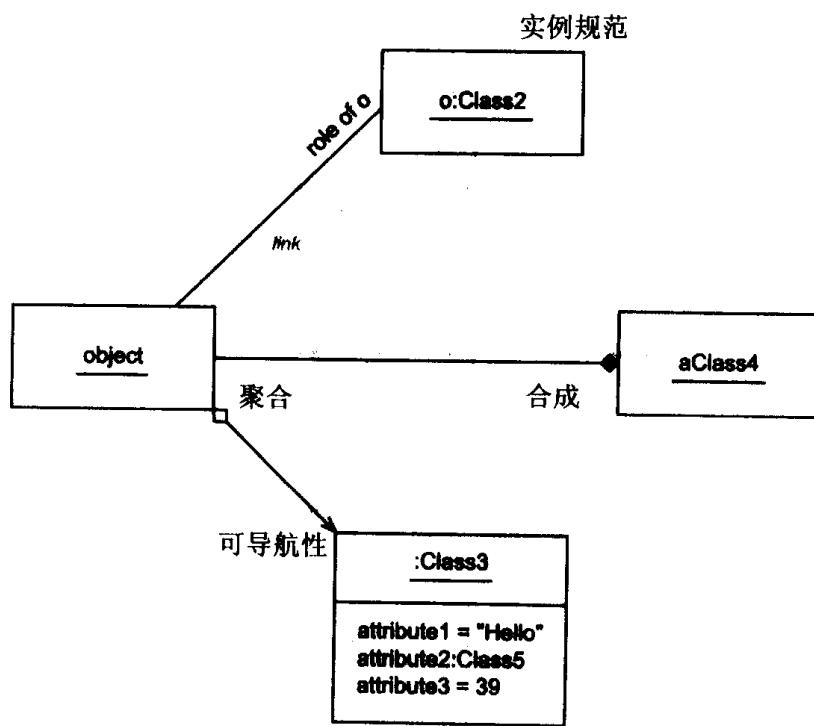
- 对象图
- 通信图(业务级)
- 活动图
- 用例图
- 类图(分析级)
- 通信图(分析级)
- 状态机图
- 部署图(网络拓扑)
- 包图
- 部署图(带有过程、制品和放大)
- 类图(设计级)
- 顺序图

这个顺序对应于本书讨论的图的顺序。按顺序介绍它们是因为，要避免重复和混乱，一些表示法在前面的图中是突出显示的，而在后面没有突出显示。当然，UML 注释可以出现在任何图中——在许多情况下，最好使用注释，而不是使用复杂的 UML 表示法，尤其是在要查看规范的情况下。

UML 使类图的设计级版本和分析级版本没有区别，但这里仍做出区分，因为前者的风格比较简单，可用的表示法也较少。

为了避免与 Java 代码段的混淆，本书中用于图的原型类型都是 Java 原型，例如 int 和 boolean，而不是 UML 原型，如 Integer 和 Boolean。对于风格，一般避免使用数组，而使用集合类(如 List)。

在 iCoot 的开发过程中不需要 UML 图的一些类型，这里也没有介绍，包括组件图、合成结构图、交互操作概图和计时图。



图C-1 对象图

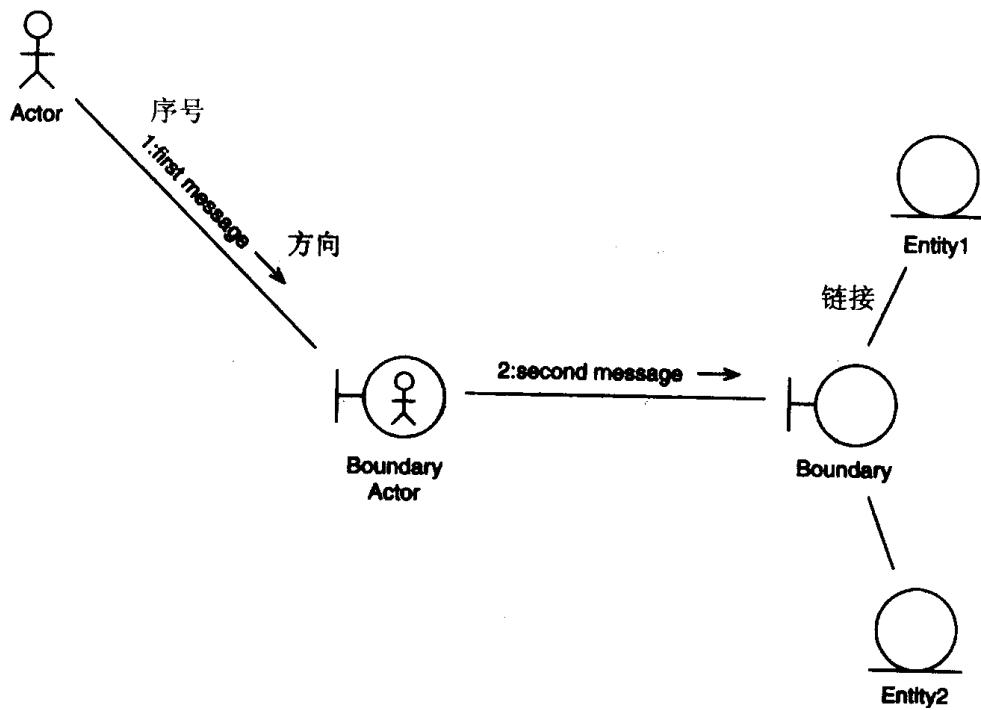


图 C-2 通信图(业务级)

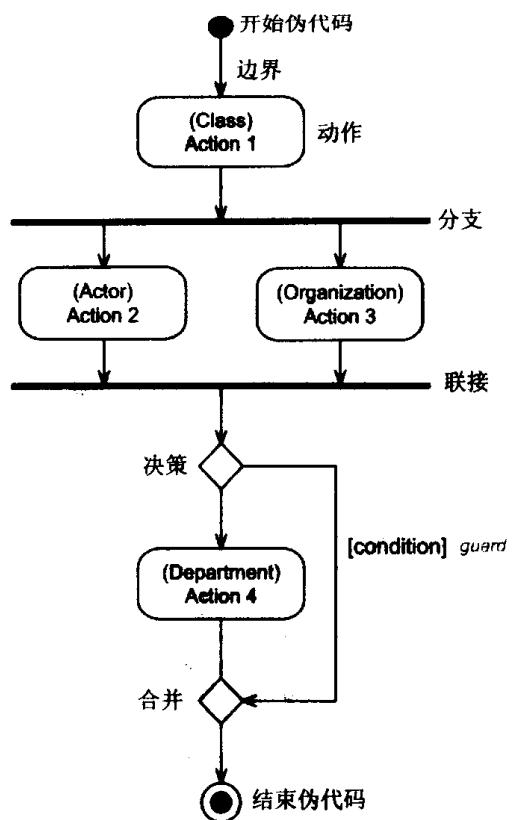


图 C-3 活动图

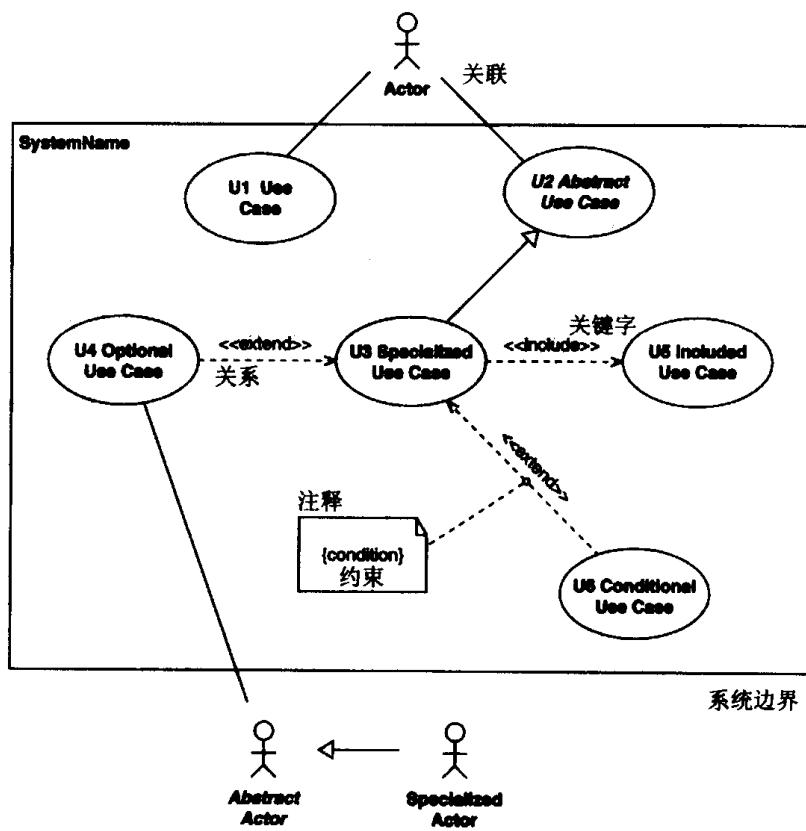


图 C-4 用例图

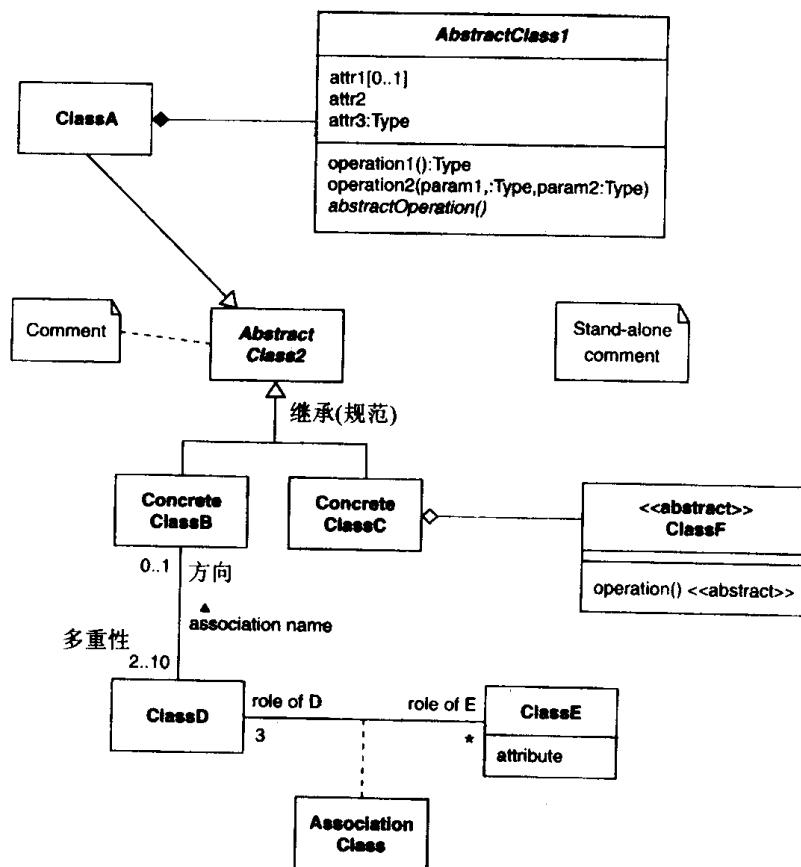


图 C-5 类图(分析级)

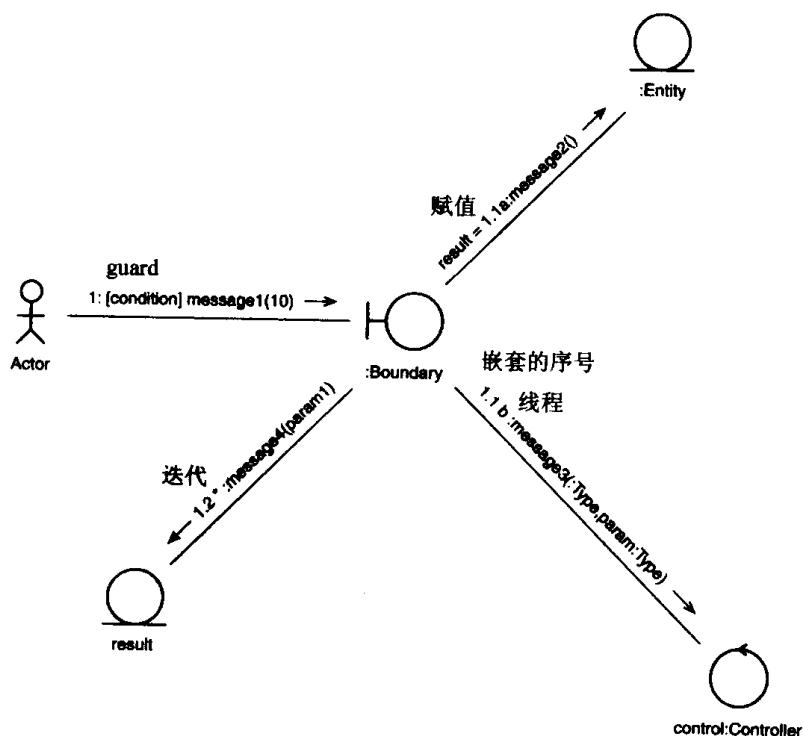


图 C-6 通信图(分析级)

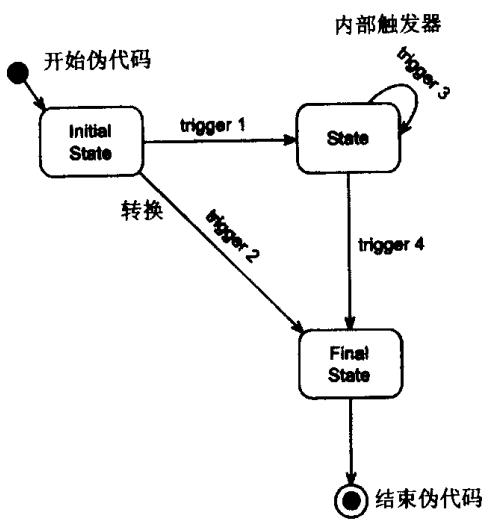


图 C-7 状态机图

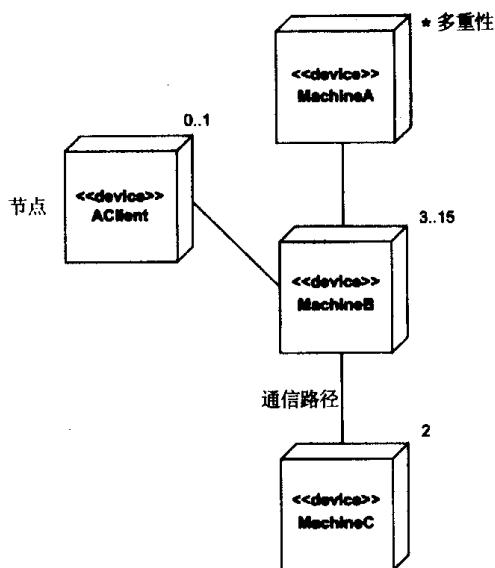


图 C-8 部署图(网络拓扑)

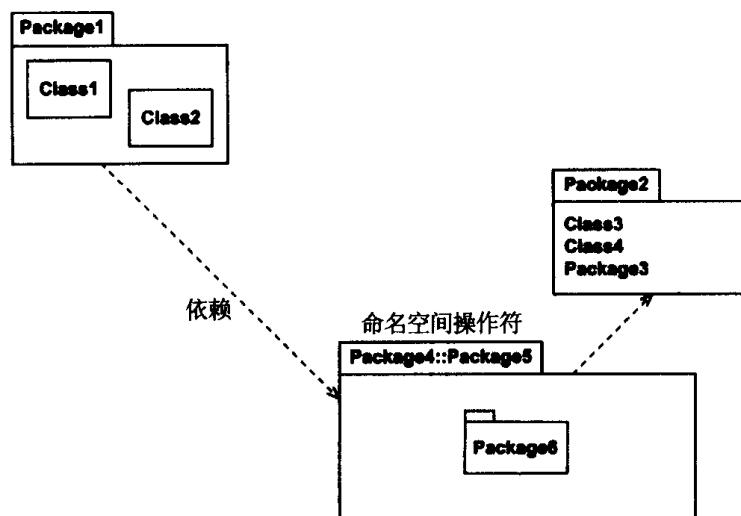


图 C-9 包图

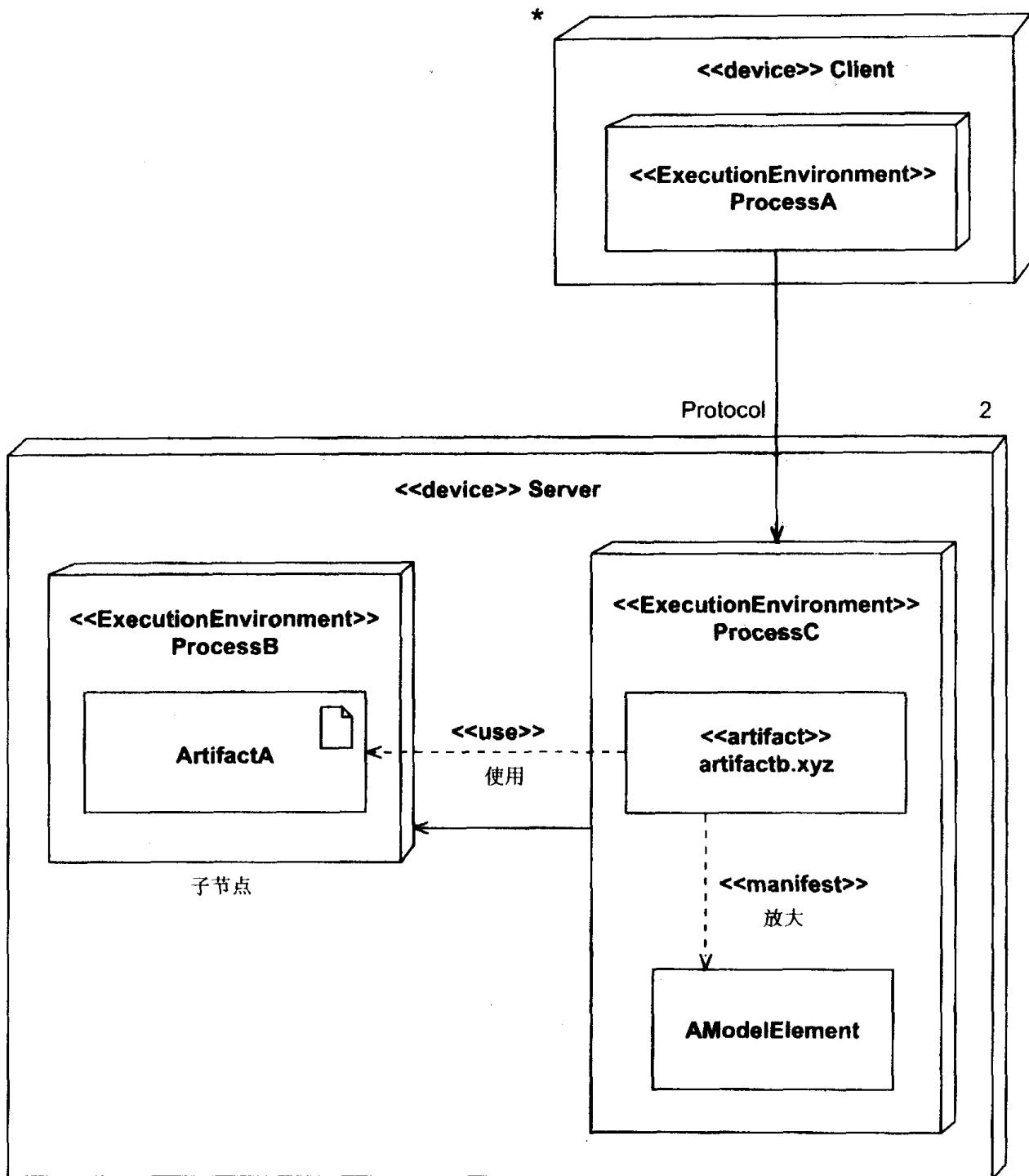


图 C-10 部署图(带有过程、制品和放大)

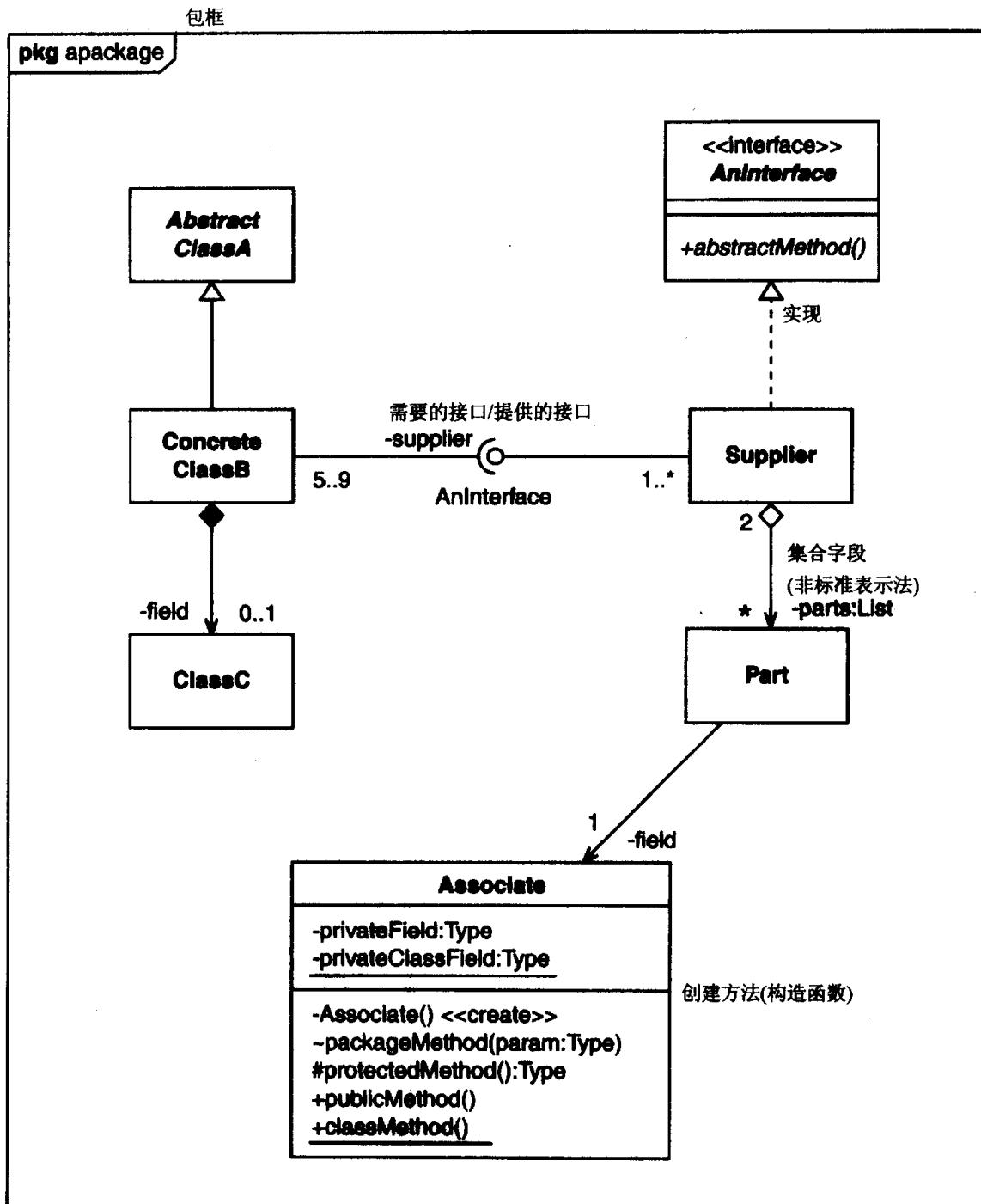


图 C-11 类图(设计级)

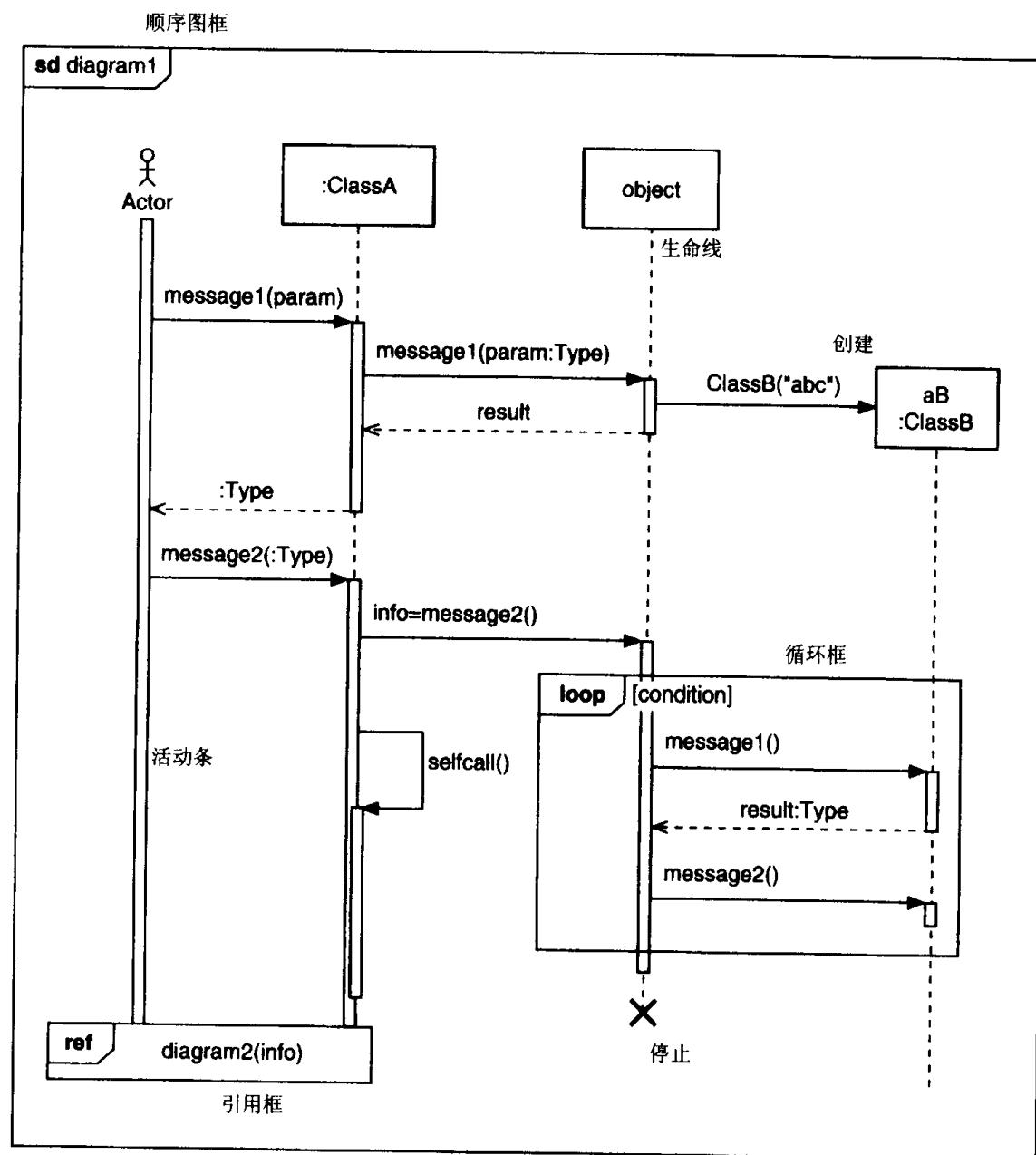


图 C-12 顺序图