

设计模式—

JAVA

语言中的应用

结城浩 著 博硕文化 译



- UML介绍说明
- Pattern名词解释
- 练习问题自我检测学习效果
- GoF的设计Pattern分类
- 范例程序代码到网站

www.tqbook.net/download.asp

处下载

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

设计模式— JAVA 语言中的应用



本书是一本Design Pattern的入门书。书中以浅显易懂的语言逐一说明了GoF的23个Design Pattern，让初次学习面向对象的读者也能立即掌握。在本书中不会看到死板的理论型文章，以Java语言写出书中各Pattern的范例程序，可以直接在计算机上运行，便于读者深入体会和应用。学习设计Pattern不仅是为了将来做准备，而且有助于从另一个全新角度重新审视我们平时编写的程序，提高软件的复用性、扩展性，这才是设计Pattern的真正意义，也是本书理论与实例相结合的目的。

【责任编辑】 苏茜 翟玉峰
【封面制作】 白雪

ISBN 7-113-06456-6



9 787113 064563 >

ISBN 7-113-06456-6/TP·1451

定价：48.00 元



中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

中国铁道出版社·计算机图书批销部

地址：北京市宣武区右安门西街8号

邮编：100054

网址：<http://www.tqbooks.net>

销售服务电话：(010)63556240 (010)63556048

读者热线电话：(010)63533215 (010)63181562

设计模式— Java 语言中的应用

结城 浩 著

博硕文化 译

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

北京市版权局著作权合同登记号：01-2004-0862 号

版 权 声 明

本书中文简体字版经 Japan UNI Agency, Inc., 由 Softbank Publishing, Inc. 授权中国铁道出版社出版（2005），任何单位或个人未经出版者书面允许不得以任何手段复制或抄袭本书内容。

Java 言語で学ぶデザインパターン入門

Copyright©2001 by 结城浩 (Yuki Hiroshi)

Simplified Chinese translation rights arranged with Softbank Publishing, Inc.

through Japan UNI Agency, Inc., Tokyo

图书在版编目 (CIP) 数据

设计模式：Java 语言中的应用/结城 浩著. —北京：中国铁道出版社，2005.3

ISBN 7-113-06456-6

I. 设… II. 结… III. JAVA 语言-程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2005) 第 027348 号

书 名：设计模式---Java 语言中的应用

作 者：结城 浩 著 博硕文化 译

出版发行：中国铁道出版社(100054, 北京市宣武区右安门西街 8 号)

策划编辑：严晓舟 郭毅鹏

责任编辑：苏 苗 瞿玉峰

特邀编辑：贾柳静

封面制作：白 雪

印 刷：北京市彩桥印刷厂

开 本：787×960 1/16 印张：28.5 字数：539 千

版 本：2005 年 6 月第 1 版 2005 年 6 月第 1 次印刷

印 数：1~4000 册

书 号：ISBN 7-113-06456-6/TP · 1461

定 价：48.00 元

版权所有 侵权必究

凡购买铁道版的图书，如有缺页、倒页、脱页者，请与本社计算机图书批销部调换。

出版说明

什么是设计模式（Design Patterns）？在我们的开发生涯中，经常会有这样的情况：某个问题其实在另外一个场合也曾经遇到过，某个解决方法其实在另外一个系统中也曾经使用过。不错，这就是模式产生的根源。所谓模式，有一个比喻非常贴切：武术套路，小孩子打架与武林高手过招的区别就在于此。从没有模式，然后学习模式，到掌握模式，最后真正领悟模式，融会贯通做到“无招胜有招”，这也正是一个程序员成长的过程。所以，设计模式伴随着程序员的成长，是程序员经验的浓缩。

对于很多人而言，真正认识并充分感受设计模式的思想，是从 GoF 的那本《设计模式——可复用的面向对象的软件基础》（Design Patterns: Element of Resuable Object-Oriented Software）一书开始的。这本经典的设计模式教程详细阐述了 23 种在软件开发过程中广为使用的“套路”。在这本书里，对“模式”的定义是：每个模式描述了一个在我们周围不断重复发生的问题，以及该问题解决方案的核心。这样你就能一次又一次使用该方案而不必做重复劳动。

自从 GoF 那本具有跨时代意义的书出版之后，“设计模式”一词渐渐成为每个程序员时常挂在嘴边的词汇之一。的确，阅读大师的每一个模式，犹如在欣赏一件结构精巧的艺术品，总是让人赞叹不已。但是艺术品终归是艺术品，掩卷长思，我们是否从中得到了启发，又是否把每个模式真正融入了我们每一次的软件开发当中。如果仅仅停留在赞叹阶段，相信这并不是大师们的初衷。

如何将 Java 语言的特性与设计模式的普遍思想有机地结合起来，对于 Java 程序员来说，是一个至关重要的转折点，如果能成功地跨过这个坎，我们就能真正达到“无招胜有招”的境界。

结城浩的这本《设计模式——Java 语言中的应用》，以 GoF 所归纳的 23 种设计模式为框架，辅以大量的 Java 语言实例，用精要的语言分析了各个模式的应用场景和相互关联，针对性非常强，为 Java 程序员掌握并领会设计模式的精髓提供了最好的帮助。

本书相关范例可以从天勤书店 www.tqbooks.net 下载专区中下载，以章节来区分子目录内容，书中示例放在 sample 目录，而 Q 和 A 开头的目录分别为问题与解答相关程序文件。

本书由 Softbank Publishing, Inc. 通过 Japan UNI Agency, Inc. 提供版权，并由博硕文化翻译，由中国铁道出版社计算机图书中心审选，颜承、陆小香、汪洋、刘军、陈知新、李莉琴等同志完成本书的整稿工作。

目 录

第 0 章 漫谈 UML	1
UML	2
类图	2
类和层次的关系	2
接口与实现	3
聚合	4
存取控制	5
类间的关联性	5
顺序图	6
处理流程和对象间的协调	6
开始学习 Design Patterns 之前	7
Design Patterns 不等于类函数库	7
不过，类函数库中确实用到 Design Patterns	8
不要认为程序范例已经是完成品	8
图不能只是看，要仔细解读	9
自己仔细琢磨范例	9
了解角色功能——白雪公主给谁演	9

第 1 部分 先熟悉设计 Pattern

第 1 章 Iterator——迭代器	11
Iterator Pattern	12
程序示例	12
Aggregate 接口	13
Iterator 接口	14
Book 类	15
BookShelf 类	15
BookShelfIterator 类	16
Main 类	17
Iterator Pattern 的所有参与者	18
扩展自我视野的提示	19

设计模式 · 110
设计模式 · 110

无论实现结果如何，都能使用 Iterator	19
抽象类、接口实在很难搞.....	20
Aggregate 与 Iterator 的对应.....	20
“下一个”容易搞错.....	21
“最后一个”也容易错.....	21
一个以上的 Iterator	21
各种 iterator	21
不需要 deleteIterator.....	21
相关 Pattern.....	22
重点回顾	22
练习题	22
第 2 章 Adapter (适配器) —— 换个包装再度利用	23
Adapter Pattern	24
程序示例 (1) (继承)	24
Banner 类	25
Print 接口	26
PrintBanner 类.....	26
Main 类	27
程序示例 (2) (委托)	27
Print 类	28
PrintBanner 类.....	29
Adapter Pattern 的所有参与者	29
扩展自我视野的提示	30
什么时候用	30
就算没有源程序也无妨	31
版本更新与兼容性	31
类差别太大	32
相关 Pattern	32
重点回顾	32
练习题	33

第 2 部分 万事交给子类

第 3 章 Template Method (模板方法) —— 实际处理就交给子类	35
Template Method Pattern.....	36

何谓模板 (Template)	36
Template Method Pattern 是什么.....	36
程序示例	37
AbstractDisplay 类.....	38
CharDisplay 类.....	38
StringDisplay 类.....	39
Main 类.....	41
Template Method Pattern 的所有参与者.....	42
扩展自我视野的提示.....	42
逻辑可共享	42
父类跟子类的连续性	43
子类应视同父类	43
相关 Pattern.....	43
进阶说明：类层次与抽象类	44
父类对子类的要求	44
抽象类的意义	44
父类跟子类之间的协调	44
重点回顾	45
练习题	45

第 4 章 Factory Method (工厂方法) ——

建立对象实例交给子类

Factory Method Pattern	48
程序示例	48
Product 类	49
Factory 类	49
IDCard 类	50
IDCardFactory 类	51
Main 类	51
Factory Method Pattern 的所有参与者	52
扩展自我视野的提示	53
框架与内容	53
产生对象实例——方法的实现方式	54
Pattern 利用与程序开发工程师之间的沟通.....	55
相关 Pattern.....	55

重点回顾	56
练习题	56

第 3 部分 建立对象实例

第 5 章 Singleton（单件）——惟一的对象实例 57

Singleton Pattern.....	58
程序示例	58
Singleton 类.....	59
Main 类	59
Singleton Pattern 的所有参与者	60
扩展自我视野的提示	61
设限的理由	61
惟一对象实例产生的时间点	61
相关 Pattern.....	61
重点回顾	61
练习题	62

第 6 章 Prototype（原型）——复制建立对象实例 63

Prototype Pattern	64
程序示例	65
Product 接口	66
Manager 类	66
MessageBox 类	67
UnderlinePen 类	68
Main 类	69
Prototype Pattern 的所有参与者	70
扩展自我视野的提示	71
难道真的不能利用类来建立对象实例吗	71
类名是一种束缚	72
相关 Pattern.....	72
深入说明： clone 方法和 java.lang.Cloneable 接口	73
Java 语言的 clone	73
clone 方法是在哪里定义	73
Cloneable 所要求的方法是什么	73

clone 方法做的是浅拷贝 (shallow copy)	74
重点回顾	74
练习题	74
第 7 章 Builder (生成器) —— 组合复杂的对象实例.....	75
Builder Pattern.....	76
程序示例	76
Builder 类	77
Director 类	77
TextBuilder 类	78
HTMLBuilder 类.....	79
Main 类.....	80
Builder Pattern 的所有参与者	83
相关 Pattern.....	84
扩展自我视野的提示	85
谁知道哪个部分	85
在设计时该决定哪些事、又不该决定哪些事.....	85
源代码的解读与修改	86
重点回顾	86
练习题	86
第 8 章 Abstract Factory (抽象工厂) —— 把相关零件组合成产品.....	89
Abstract Factory Pattern.....	90
程序示例	90
抽象零件: Item 类	94
抽象零件: Link 类.....	95
抽象零件: Tray 类.....	95
抽象产品: Page 类	96
抽象工厂: Factory 类	97
利用工厂把零件组合成产品: Main 类.....	98
具体工厂: ListFactory 类.....	100
具体零件: ListLink 类	100
具体零件: ListTray 类.....	101
具体产品: ListPage 类	102

在程序示例中新增其他的具体工厂	103
具体工厂：TableFactory 类	105
具体零件：TableLink 类	106
具体零件：TableTray 类	106
具体产品：TablePage 类	107
Abstract Factory Pattern 的所有参与者	108
扩展自我视野的提示	110
新增具体工厂尤如探囊取物	110
新增零件很有难度	110
相关 Pattern	110
深入说明：各种建立对象实例的手段	111
重点回顾	112
练习题	112

第 4 部分 切割性思考

第 9 章 Bridge（桥接）——分成功能层次和实现层次	113
Bridge Pattern	114
类层次的两大责任	114
程序示例	116
功能的类层次：Display 类	116
功能的类层次：CountDisplay 类	117
实现的类层次：DisplayImpl 类	118
实现的类层次：StringDisplayImpl 类	118
Main 类	119
Bridge Pattern 的所有参与者	121
扩展自我视野的提示	121
先区分再扩充就很轻松	121
继承是生死不离、委托是潇洒分手	122
相关 Pattern	123
重点回顾	123
练习题	123
第 10 章 Strategy（第略）——把算法整个换掉	125
Strategy Pattern	126

程序示例	126
Hand 类	127
Strategy 接口	129
WinningStrategy 类	129
ProbStrategy 类	130
Player 类	132
Main 类	133
Strategy Pattern 的所有参与者	135
扩展自我视野的提示	136
为什么要特地建立 Strategy 参与者	136
执行时也可切换	136
相关 Pattern	137
重点回顾	137
练习题	137

第 5 部分 一视同仁

第 11 章 Composite（组成）——对容器和内容一视同仁	141
Composite Pattern	142
程序示例	142
Entry 类	143
File 类	144
Directory 类	145
FileTreatmentException 类	147
Main 类	147
Composite Pattern 的所有参与者	150
扩展自我视野的提示	150
单、复数的一视同仁	150
add 放在哪里	151
递归结构随时都会出现	152
相关 Pattern	152
重点回顾	152
练习题	152

第 12 章 Decorator（装饰）——对装饰和内容一视同仁 155

Decorator Pattern.....	156
程序示例	156
Display 类.....	157
StringDisplay 类.....	158
Border 类.....	159
SideBorder 类.....	159
FullBorder 类.....	161
Main 类.....	162
Decorator Pattern 的所有参与者.....	163
扩展自我视野的提示.....	164
可穿透的接口（API）.....	164
可新增功能但内容不变.....	165
可新增动态功能.....	165
即使只有简单的种类规划，也可增加多种功能.....	165
java.io 包和 Decorator Pattern.....	166
小类多了一点.....	167
相关 Pattern.....	167
深入说明：继承和委托的一视同仁.....	167
继承——对子类和父类一视同仁.....	167
委托——对自己和委托对象一视同仁.....	168
重点回顾.....	169
练习题	170

第 6 部分 在结构中穿梭来去

第 13 章 Visitor（访问者）——在结构中穿梭还同时做事 173

Visitor Pattern.....	174
程序示例	174
Visitor 类.....	174
Acceptor 接口	176
Entry 类.....	176
File 类.....	177
Directory 类.....	178

ListVisitor 类	179
FileTreatmentException 类	180
Main 类	180
Visitor 和 Acceptor 两边彼此调用	182
Visitor Pattern 的所有参与者	183
扩展自我视野的提示	184
双重调度 (double dispatch)	184
为什么要搞得这么复杂	185
The Open-Closed Principle——扩充时开放、修改时关闭	185
新增 ConcreteVisitor 参与者？简单	186
新增 ConcreteAcceptor 参与者？复杂	186
Visitor 要怎样做才能进行处理	186
相关 Pattern	186
重点回顾	187
练习题	187

第 14 章 Chain of Responsibility (职责链) —— 责任转送 .. 191

Chain of Responsibility 责任转送 ..	192
程序示例	192
Trouble 类	193
Support 类	193
NoSupport 类	195
LimitSupport 类	195
OddSupport 类	196
SpecialSupport 类	196
Main 类	197
Chain of Responsibility Pattern 的所有参与者 ..	198
扩展自我视野的提示	199
让要求跟处理要求两者间的结合关系不会太紧密	199
机动性改变连锁形态	200
能更专心在自己的工作岗位上	200
转送会不会导致处理变慢	200
相关 Pattern	201
重点回顾	201
练习题	201

第 7 部分 简单最好

第 15 章 Facade (外观) —— 单一窗口	203
Facade 单一窗口	204
程序示例	204
Database 类	206
HtmlWriter 类	206
PageMaker 类	208
Main 类	209
Facade Pattern 的所有参与者	210
扩展自我视野的提示	211
Facade 参与者到底在做什么事	211
Facade Pattern 的递归应用	211
程序设计师拒绝使用 Facade 的理由——敏感的心理因素	211
相关 Pattern	212
重点回顾	212
练习题	212
第 16 章 Mediator (中介者) —— 只要面对一个顾问	215
Mediator Pattern	216
程序示例	216
Mediator 接口	220
Colleague 接口	220
ColleagueButton 类	221
ColleagueTextField 类	222
ColleagueCheckbox 类	223
LoginFrame 类	224
Main 类	227
Mediator Pattern 的所有参与者	227
扩展自我视野的提示	229
当分散变成灾难时	229
通讯路径数量暴增	229
哪些可以再利用	229
相关 Pattern	230

重点回顾	230
练习题	230

第 8 部分 管理状态

第 17 章 Observer (观察者) ——通知状态变化	231
Observer Pattern	232
程序示例	232
Observer 接口	232
NumberGenerator 类	233
RandomNumberGenerator 类	234
DigitObserver 类	235
GraphObserver 类	235
Main 类	236
Observer Pattern 的所有参与者	237
扩展自我视野的提示	238
这里也可对换	238
Observer 的顺序	239
Observer 的行为影响 Subject 时	239
如何处理更新的提示	239
“通知”的意义重于“观察”	240
Model/View/Controller (MVC)	240
补充说明: java.util.Observer 接口	241
相关 Pattern	241
重点回顾	241
练习题	242
第 18 章 Memento (备忘录) ——存储状态	243
Memento Pattern	244
程序示例	244
Memento 类	245
Gamer 类	246
Main 类	248
Memento Pattern 的所有参与者	252
扩展自我视野的提示	253
2 种接口 (API) 与存取控制	253

Memento 可以有几个	255
Memento 有没有使用期限	255
Caretaker 参与者跟 Originator 参与者区隔使用的意义	255
相关 Pattern	255
重点回顾	256
练习题	256
第 19 章 State（状态）——以类表示状态	259
State Pattern	260
程序示例	260
金库保安系统	260
不使用 State Pattern 的虚拟程序代码	261
使用 State Pattern 的虚拟程序代码	262
State 接口	263
DayState 类	265
NightState 类	266
Context 接口	267
SafeFrame 类	267
Main 类	271
State Pattern 的所有参与者	272
扩展自我视野的提示	273
分割统合	273
有此状态才会有的处理	274
状态迁移该由谁管	274
不会有自我矛盾	275
新增状态易如反掌	275
变化万千的对象实例	276
相关 Pattern	276
重点回顾	277
练习题	277
第 9 部分 精简不浪费	

第 20 章 Flyweight（享元）——有相同的部分就共享， 采取精简政策	279
Flyweight Pattern	280

程序示例	280
BigChar 类.....	285
BigCharFactory 类	286
BigString 类.....	287
Main 类.....	289
Flyweight Pattern 的所有参与者.....	289
扩展自我视野的提示	290
影响会传到所有相关位置	290
Intrinsic 和 extrinsic	291
管理中的对象实例不会被视为垃圾收集	291
非内存部分的记录	292
相关 Pattern.....	293
重点回顾	293
练习题	293
第 21 章 Proxy (代理) ——要用再建立	295
Proxy Pattern	296
程序示例	296
Printer 类	298
Printable 接口	299
PrinterProxy 类.....	299
Main 类.....	301
Proxy Pattern 的所有参与者	301
扩展自我视野的提示	302
利用代理提高处理速度	302
不能让代理和本人合二为一吗	303
代理和委托	303
穿透性	303
HTTP proxy	304
各种 Proxy	304
相关 Pattern	304
重点回顾	305
练习题	305

第 10 部分 用类来表示

第 22 章 Command (命令) —— 将命令写成类	307
Command Pattern	308
程序示例	308
Command 接口	309
MacroCommand 类	310
DrawCommand 类	311
Drawable 接口	312
DrawCanvas 类	312
Main 类	314
Command Pattern 的所有参与者	317
扩展自我视野的提示	319
命令应该要有的信息是什么	319
存储记录	319
转换器	320
相关 Pattern	322
重点回顾	322
练习题	322
第 23 章 Interpreter (解释器) —— 以类来表达语法规则	325
Interpreter Pattern	326
迷你语言	326
迷你语言的命令	326
迷你程序示例	327
迷你语言的语法	330
终点表达式与非终点表达式	331
程序示例	331
Node 类	332
ProgramNode 类	333
CommandListNode 类	334
CommandNode 类	335
RepeatCommandNode 类	336
PrimitiveCommandNode 类	337
Context 类	337

ParseException 类	339
Main 类	339
Interpreter Pattern 的所有参与者	341
扩展自我视野的提示	342
还有其他不同的迷你语言吗	342
略过/读入	343
相关 Pattern	343
重点回顾和最后的叮咛	343
练习题	344

附录

附录 A 练习题的解答	345
附录 B GoF 的设计 Pattern 分类	435

第 0 章

漫谈 UML

第二部分

UML

UML

UML 是一种以可视化的观点来看程序系统，将规格和设计重点直接写下来的表现方式。UML 是 Unified Modeling Language 的缩写。中文译成“统一建模语言”。

本书利用 UML 来表现 Design Patterns（设计模式）中类（class）和对象实例（Instance）的关系，所以在此先举出一些阅读本书时所必备的 UML 常识。不过，说明时的名词术语仍然承袭 Java 语言的习惯用法（比如说，用 Java 的“field（字段）”取代 UML 的“attribute（属性）”，用 Java 的“method（方法）”取代 UML 的“operation（操作）”）。

UML 的规格相当庞大，本书中所引用的内容（也就是以下所解说的范围）仅为其中一小部分。如果各位读者有兴趣进一步了解 UML，可到下列网站取得 UML 的详细规格。

- UML Resource Page
<http://www.omg.org/uml/>
- UML Resource Center
<http://www.rational.com/uml/>

类图

UML 的类图（Class Diagram）是一种可表示一组类、对象实例和接口之间静态关系的图。虽然称为类图，但使用到的不只是类。

类和层次的关系

Java 程序与对应类图的范例如图 0-1 所示。

```
abstract class ParentClass {
    int field1;
    static char field2;
    abstract void methodA();
    double methodB() {
        // ...
    }
}

class ChildClass extends ParentClass {
    void methodA() {
        // ...
    }
    static void methodC() {
        // ...
    }
}
```

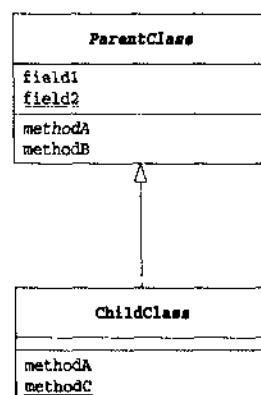


图 0-1 类层次关系的类图

上图所表示的是 ParentClass 和 ChildClass 这两个类之间的关系。白色箭头△方向就是类的层次关系。箭头是从子类 (subclass) 指向父类 (superclass) (所以这是 extends 的箭头)。

ParentClass 是 ChildClass 的父类, ChildClass 是 ParentClass 的子类。父类也称为基类 (Base Class), 而子类又称为派生类或扩充类。

类均以方框表示, 方框内再以横线分割几个部分, 分别是:

- 类名称
- 字段名称
- 方法名称

有时除了名称之外, 还会有其他附属信息 (如存取控制、方法的参数或类型等), 或者是反其道而行省略图中不需要的项目 (故不一定每次都能从类图还原成源程序)。

abstract 类 (抽象类) 名称以斜体表示。如 ParentClass 为抽象类, 故写成斜体。

static 字段 (类字段) 名称加下划线。如 field2 为类字段, 故加上下划线。

abstract 方法 (抽象方法) 以斜体表示。如 ParentClass 的 methodA 是抽象方法, 故写成斜体。

static 方法 (类方法) 名称加下划线。如 ChildClass 的 methodC 为类方法, 故加上下划线。

► 深入说明——Java 和 C++ 的名词术语

Java 跟 C++ 的名词术语稍微有点不一样。Java 的字段 (field) 对应到 C++ 的成员变量 (member variable), Java 的方法 (method) 则对应到 C++ 的成员函数 (member function)。

► 深入说明——箭头方向

在 UML 当中, 箭头是从子类指向父类。也许有人认为反过来应该比较容易理解, 毕竟是先有父类才能生出子类。

其实, 各位可以这么想: 定义子类时, 要用 extends 指定父类。所以子类一定会知道父类是谁; 但是父类就不一定知道子类在哪里。必须知道对方在哪里, 才能指出来, 因此箭头方向是从子类指向父类。

接口与实现

图 0-2 也是类图范例之一。

图中表示一个 **Printable** 接口, 该接口上实现了一个 **PrintClass** 类。接口名称为斜体。

虚线的白色箭头是表示接口与实现类的关系。箭头从实现类指向接口 (所以这是

implements 的箭头)。

UML 中的 Java 接口则以<<interface>>表示。

```
interface Printable {
    abstract void print();
    abstract void newPage();
}

class PrintClass implements Printable {
    void print() {
        // ...
    }
    void newPage() {
        // ...
    }
}
```

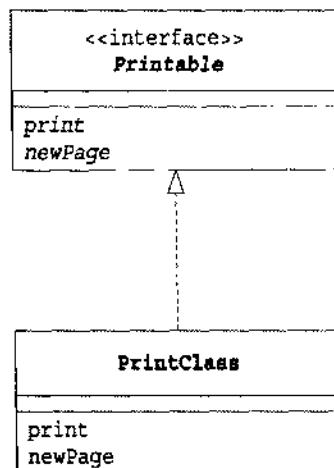


图 0-2 接口和实现类的类图

```
class Color {
    // ...
}

class Fruit {
    Color color;
    // ...
}

class Basket {
    Fruit[] fruits;
    // ...
}
```

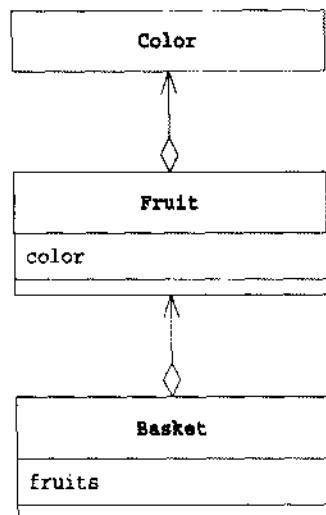


图 0-3 聚合关系的类图

聚合

图 0-3 也是类图范例之一。

图中所表示的是 `Color` (颜色)、`Fruit` (水果)、`Basket` (果篮) 这 3 个类的关系。`Basket` 类中的 `fruits` 字段是 `Fruit` 类的数组, `Basket` 类的对象实例则有多个 `Fruit` 类的对

象实例。而 Fruit 类的 color 字段是 Color 类类型, Fruit 类的对象实例则只有 1 个 Color 类的对象实例。从平面来看, 它们的关系就好比是果篮里面有一些水果, 各种水果有不同的颜色。

这种“结合”的关系就称为“聚合”(aggregation)。只要结合了对象实例, 无论个数多少都有聚合关系。数组、java.util.Vector 类也好, 如何实现也罢, 只要有结合对象实例, 就具有聚合关系。

白色菱形的直线代表聚合关系, 可以把它想成是把东西放在菱形的容器上。

存取控制

图 0-4 还是类图的范例之一。

```
class Something {
    private int privateField;
    protected int protectedField;
    public int publicField;
    private void privateMethod() {
    }
    protected void protectedMethod() {
    }
    public void publicMethod() {
    }
}
```

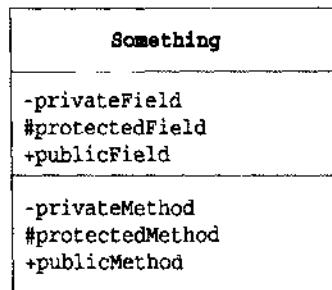


图 0-4 存取控制的类图

图中所表示的是方法和字段的存取控制。UML 只要在方法或字段的名称前面加上符号, 就可以表示存取控制。

如果是 +, 表示这是 public 的方法或字段, 从任何位置都可以存取。

而如果是 -, 则表示这是 private 的方法或字段, 此时从类以外的位置就无法存取。

若是 #, 则表示这是 protected 的方法或字段, 只有同一类、子类或同一 package 内的类才能存取。

类间的关联性

在相关的名称前面加上黑色三角形 (►), 就可以表示类间的关联性。范例如图 0-5 所示。

设计模式的初应用

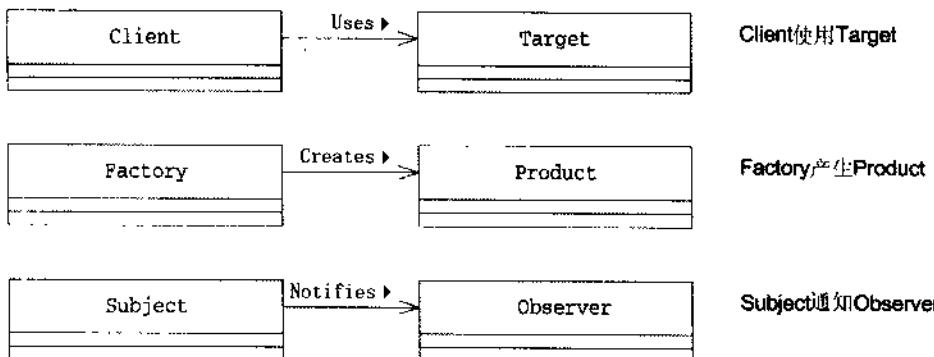


图 0-5 类的关联性

顺序图

UML 的顺序图（Sequence Diagram）是用来表示运行程序时，会按照怎样的顺序执行哪些方法，哪些现象会按什么顺序发生。

类图表示“不因时间而变化的部分（静态关系）”，而顺序图则表示“随时间变化的部分（动态行为）”。

处理流程和对象间的协调

图 0-6 是一种顺序图的范例。

图 0-6 的右边是顺序图，左边则是相对应的 Java 程序（一部分）。

在这个范例中出现了 3 个对象实例，分别对应到图例上方的 3 个方框。在方框中，类名称写在冒号（:）后面，而且还加上下划线，如：`Client`, `Server`, `Device`。意思是指 `Client` 类的对象实例、`Server` 类的对象实例以及 `Device` 类的对象实例。

如果每个对象实例均须命名，则将其名称写在冒号之前，如 `server:Server`。

所有对象实例都有一条往下延伸的虚线，称为 `lifeline`（生命线）。在此请将时间坐标看成由上而下，上而是过去、下面则是未来。只在有对象实例时，才会有 `lifeline`。

`lifeline` 中间有一个长条型的方框，表示该对象处于运行期。

图中有几个箭头指向左右两边。请先看标示为 `open` 的箭头记号。若线段为实心且箭头全黑（→）则表示调用方法。此图是 `client` 调用 `server` 的 `open` 方法。因为已经成功调用出 `open` 方法，所以 `server` 对象实例是运行期的状态，故为长条型的方框。

`open` 箭头所指的 `server` 长条型方框下面，还有一条指向 `client` 的虚线箭头（----->）。这是指 `open` 方法的 `return`（返回）。这个范例把所有方法的 `return` 都画出来，不过也有人宁愿省略以免麻烦。

```

class Client {
    Server server;
    void work() {
        server.open();
        server.print("Hello");
        server.close();
    }
    // ...
}

class Server {
    Device device;
    void open() {
        // ...
    }
    void print(String s) {
        device.write(s);
        // ...
    }
    void close() {
        // ...
    }
    // ...
}

class Device {
    void write(String s) {
        // ...
    }
}

```

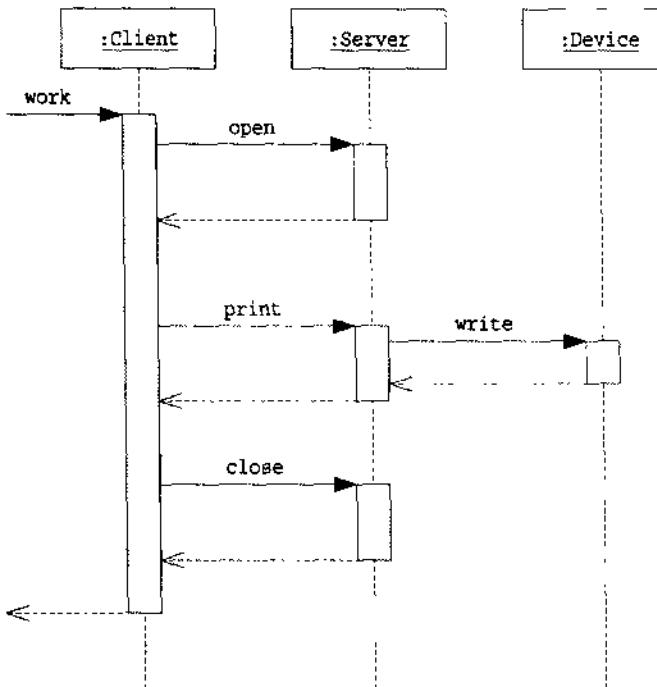


图 0-6 顺序图范例（调用方法）

因为控制又回到了 client，所以 server 对象实例运行期就结束了。

继续以同样的方式调用 print 方法。这次是直接在 print 方法内就调用 device 对象实例的 write 方法。

类似这样两个以上的对象实例之间的动作都可以用图表示出来。顺序图要跟着 lifeline，由上而下依序解读。如果出现箭头时，也要跟着去确认对象实例间的交互操作。

开始学习 Design Patterns 之前

以下介绍几个有助于理解 Design Patterns 的重要常识。

Design Patterns 不等于类函数库

利用 Java 语言写程序时，常常会用到类函数库。但是 Design Patterns 并不是类函数库。

Java语言中的应用

Design Patterns 比类函数库更具有一般性。类函数库是用来当作组件的程序，而 Design Patterns 则是构思要如何组合组件、如何结合不同的组件以发挥最大功能。

假设现在要把白雪公主的童话故事拍成连续剧，在说明白雪公主故事纲要时不必讲到谁演白雪公主、谁演王子等部分，好好解释白雪公主跟王子之间的“关系”要比说明由哪些人演出更重要。最重要的是剧中要出现哪些主人翁，主人翁之间又有什么样的参与者互动。

Design Patterns 也是如此。在回答“Abstract Factory Pattern 是什么？”的问题时，参考实际的程序范例更能加强理解，但只有实际的程序范例并不是真正的 Abstract Factory Pattern。Design Patterns 重点在于描述当中出现的类和接口，以及它们之间的关系。

不过，类函数库中确实用到 Design Patterns

Design Patterns 当然不是类函数库，但是 Java 的标准类函数库却有好几个从 Design Patterns 产生出来的类。如果各位读者能彻底了解 Design Patterns，应该也有助于了解这些类函数库的作用。

后面各个 Design Patterns 的章节也会陆续说明，现在先举几个比较具有代表性的范例给各位参考一下。

- `java.util.Iterator` 是枚举两个以上时的类。应用到 Iterator Pattern（第 1 章）。
- `java.util.Observer` 是观察对象状态变化的接口。应用到 Observer Pattern（第 17 章）。
- 以下的方法则应用到 Factory Method Pattern（第 4 章）。
 - `java.util.Calendar` 类的 `getInstance` 方法
 - `java.security.SecureRandom` 类的 `getInstance` 方法
 - `java.text.NumberFormat` 类的 `getInstance` 方法
- `java.awt.Component` 和 `java.awt.Container` 类则应用到 Composite Pattern（第 11 章）。

不要认为程序范例已经是完成品

Design Patterns 的目标之一就是让程序可重复利用，也就是要如何把程序当作一种“组件”好好重新利用。因此，请不要把程序范例当作是“完成品”，接下来“要继续扩充功能”、“继续修改下去”。

- 是否可继续扩充功能？
- 扩充功能时须修改哪些类？

- 有哪些类并不需要修改？

以这样的观点来看 Design Patterns 的话，应该会加深印象，更容易理解。

图不能只是看，要仔细解读

在解说 Design Patterns 时会需要图解帮忙。本书主要使用类图和顺序图（参见“漫谈 UML”）两种。请不要认为这只是“图解”而已，光凭第一印象下判断通常不太准。

遇到类图时，请先看一个个独立的方框（类），再继续看方框内的方法名称，确认它到底是一般的方法还是抽象方法。然后确认类之间的箭头种类，判断哪个类实现了哪种接口。以此类推，仔细抽丝剥茧找出图中结构元素的真正意义，这样才能了解整个图解所要表达的意思。

顺序图应该比类图容易解读。时间轴是由上而下，所以很轻松就能逐一确认是从哪个对象调用哪个对象。耐着性子按部就班，各个对象在 Pattern 中扮演什么样的角色就慢慢浮上台面了。

自己仔细琢磨范例

请各位读者不要只看完程序范例就算了，思考一下如果是自己会写什么样的程序来用这个 Pattern。

还有，当各位在构思或撰写程序时，也可以随时想想“这部分能不能应用所学的 Design Patterns？”。

了解角色功能 —— 白雪公主给谁演

Design Patterns 就像是一出连续剧，剧中会有很多不同的出场人物（即类和接口），剧情就从他们之间的关系发展出来。每个角色都分配到一个功能，所有参与角色都必须配合自己的参与者功能发挥演技。主角要有主角的气势，演对手戏的另一个角色也要跟主角相抗衡。

Design Patterns 也是如此。每个 Design Patterns 对不同的类或接口都分配了恰如其分的参与角色功能。如果不能了解各个类和接口的功能，不但抓不住相当于剧纲的 Pattern，也很难适当搭配类型。可能反而让主角被敌人牵着跑，或善良可爱的女主角竟然变成冷血大魔女，或者是轻松爆笑剧搞成一把鼻涕一把眼泪的大悲剧，写实纪录片拍成科幻 X 档案。

以下各章将逐一介绍不同的 Design Patterns，还会有 Pattern 中各个功能的重点说明。希望读者在看范例程序代码时，不要只把它当作程序，也要注意类和接口在该 Pattern 中具有何种功能。

Java 指挥者的应用

如果是同一个 Pattern，类名称不小心弄错了，功能对应可不能牛头不对马嘴。只要功能对应正确无误，也就不会太难理解。

好啦，不多说了，我们赶快来学习这些 Pattern 吧！

第1章

Iterator——迭代器

Iterator Pattern

用 Java 语言输出数组 arr 的所有元素时，使用 for 语句的写法是：

```
for (int i = 0; i < arr.length; i++) {  
    System.out.println(arr[i]);  
}
```

请注意上面的循环变量 i。变量 i 一开始被初始化为 0，然后依次递增为 1, 2, 3, ...，同时 arr[i] 的内容也跟着输出到屏幕上。类似这样的 for 语句使用率很高。

数组是多个元素的聚合，只要指定下标（subscript）就可以指定数组元素中的一个元素。

```
arr[0] 最初的元素（第 0 个元素）  
arr[1] 下一个元素（第 1 个元素）  
：  
arr[i] 第 i 个元素  
：  
arr[arr.length - 1] 最后一个元素
```

利用 for 语句的 i++ 递增 i 值，即可依次推移目前的元素。这样递增 i 值的做法可从头依序遍历数组 arr 的所有元素。

在设计 Pattern 时，上述变量 i 的功能抽象化、一般化的结果就称为：

Iterator Pattern

Iterator Pattern 是指依序遍历并处理多个数字或变量。iterate 英文单字本身也是“反复”的意思，iterator 这个名词可译为“迭代器”。

本章将带领各位学习 Iterator Pattern。

程序示例

首先请各位看下面这个 Iterator Pattern 的应用程序示例。这个程序是把书籍（Book）放到书架（BookShelf）上，并依次输出书名（图 1-1）。

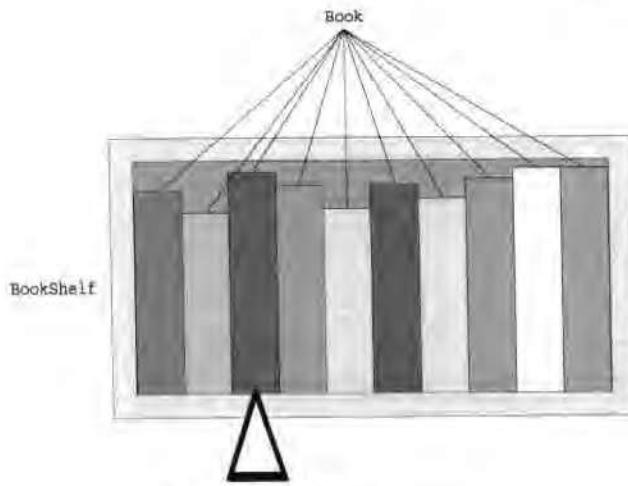


图 1-1 程序示例的示意图

Aggregate 接口

Aggregate 接口（程序 1-1）是一个执行递增的“聚合”。实现此接口的类就变成类似数组的“多个数字（或变量等）的聚合”。aggregate 英文单字本身就是“聚集”、“聚合”的意思。

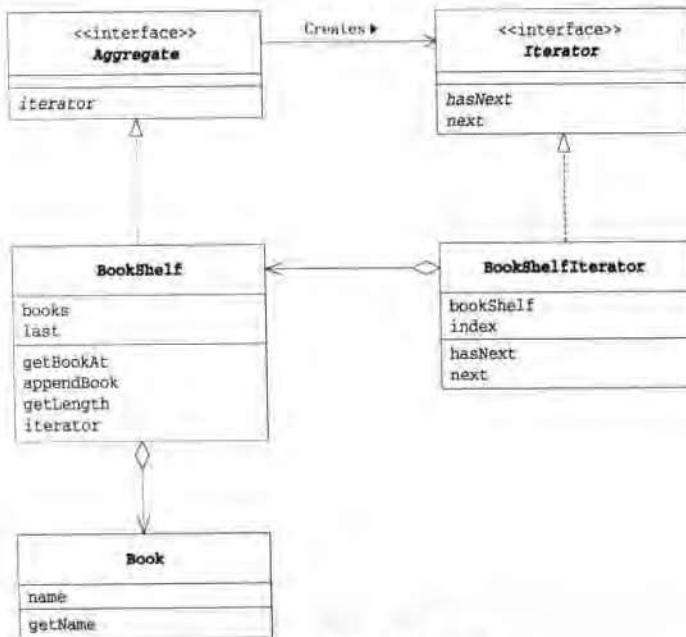


图 1-2 程序示例的类图

表 1-1 类、接口一览表

名称	说明
Aggregate	表示已聚合的接口
Iterator	执行递增、遍历的接口
Book	表示书籍的类
BookShelf	表示书架的类
BookShelfIterator	扫描书架的类
Main	测试用的类

程序 1-1 Aggregate 接口 (Aggregate.java)

```
public interface Aggregate {
    public abstract Iterator iterator();
}
```

Aggregate 接口所声明的方法只有 iterator 方法一个，这是为了建立一个可对应聚合的 Iterator。

如欲递增、遍历或逐一检查某个聚合时，利用 iterator 方法即可建立一个实现 Iterator 接口的类对象实例 (instance)。

Iterator 接口

请继续看 Iterator 接口 (程序 1-2)。Iterator 接口执行元素递增，具有类似循环变量的功能。那么，Iterator 需要使用哪一种方法？当然，方法千变万化随个人喜好而定，这里先介绍最简单的 Iterator 接口。

程序 1-2 Iterator 接口 (Iterator.java)

```
public interface Iterator {
    public abstract boolean hasNext();
    public abstract Object next();
}
```

上面示例中所声明的方法有 2 个。hasNext 方法是用来检查有没有“下一个元素”，next 方法则是取得“下一个元素”。

各位应该都了解 hasNext 方法的返回值为什么是 boolean 类型的理由吧？如果有下一个元素时，hasNext 方法即返回 true；若无下一个元素，也就是已经进行到最后一个元素时，则 hasNext 方法的返回值为 false。在此，hasNext 方法就是循环结束的条件。

Iterator——迭代器

`next` 方法可能需要稍微说明一下。从其返回值类型为 `Object` 可知, `next` 方法是返回聚合当中的 1 个元素。不过, `next` 方法的功能并非仅仅如此。事实上它还会悄悄地先要内部状态进入下一步, 以便下次调用 `next` 方法时能确实返回下一个元素。虽然说这是“台面下的行为”, `Iterator` 接口也只知道方法名称而已, 具体还是要看实现这个 `Iterator` 接口的类 (`BookShelfIterator`) 才比较清楚。

Book 类

`Book` 类 (程序 1-3) 是表示书籍的类。不过作用并不大, 它只能利用 `getName` 方法取得书名而已。书名则是在以构造函数 (`constructor`) 初始化对象实例时就用参数指定。

程序 1-3 Book 类 (Book.java)

```
public class Book {
    private String name = "";
    public Book(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

BookShelf 类

`BookShelf` 类 (程序 1-4) 是表现书架作用的类。要实现 `Aggregate` 接口才能把这个类当作聚合来处理。程序代码中的“`implements Aggregate`”就是实现 `Aggregate` 接口的部分。另外, 也要确认里面含有 `Aggregate` 接口所声明的 `iterator` 方法的实例。

程序 1-4 BookShelf 类 (BookShelf.java)

```
public class BookShelf implements Aggregate {
    private Book[] books;
    private int last = 0;
    public BookShelf(int maxsize) {
        this.books = new Book[maxsize];
    }
    public Book getBookAt(int index) {
        return books[index];
    }
}
```

Java语言中的应用

```

public void appendBook(Book book) {
    this.books[last] = book;
    last++;
}
public int getLength() {
    return last;
}
public Iterator iterator() {
    return new BookShelfIterator(this);
}
}

```

这个书架中有一个叫做 books 的字段，这个字段就是 Book 的数组。数组大小 (maxsize) 在一开始建立 BookShelf 实例时就要设定。把 books 字段 private 起来则是为了避免被其他类修改。

话题转回到 iterator 方法。iterator 方法是产生并返回 BookShelfIterator 类的对象实例，作为对应 BookShelf 类的 Iterator。在遍历此书架上的书籍时，会调用 iterator 方法。

BookShelfIterator 类

接着请看扫描 BookShelf 类的 BookShelfIterator 类（程序 1-5）。

程序 1-5 BookShelfIterator 类 (BookShelfIterator.java)

```

public class BookShelfIterator implements Iterator {
    private BookShelf bookShelf;
    private int index;
    public BookShelfIterator(BookShelf bookShelf) {
        this.bookShelf = bookShelf;
        this.index = 0;
    }
    public boolean hasNext() {
        if (index < bookShelf.getLength()) {
            return true;
        } else {
            return false;
        }
    }
}

```

```

    public Object next() {
        Book book = bookShelf.getBookAt(index);
        index++;
        return book;
    }
}

```

实现 Iterator 接口即可将 BookShelfIterator 视为 Iterator 进行处理。

bookShelf 字段是 BookShelfIterator 所扫描的书架，而 index 字段则是指向目前该书的下标。

构造函数把传过来的 BookShelf 对象实例存储在 bookShelf 字段，将 index 设为 0。

实现 Iterator 接口所声明的方法的是 hasNext 方法。检查是否有“下一本”，如果有则返回 true，若无则返回 false。是否有下一本的判断标准则根据 index 是否小于书架上书籍数量（即表达式 bookShelf.getLength() 的值）来判断。

next 方法则是返回目前该书（Book 的对象实例），并进入到“下一个”方法。这也是在 Iterator 接口所声明的方法。这个部分有点复杂，要先把书籍（即返回值）保留在 book 变量，然后继续把 index 推到下一个位置。

“继续把 index 推到下一个位置”的处理相当于前面 for 语句（第 12 页）所指 i++ 的处理，也就是把循环变量推到“下一个”。

Main 类

到此为止，我们已经完成遍历书架的准备工作。接着就要尝试利用 Main 类（程序 1-6）建立一个小型的书架。

程序 1-6 Main 类（Main.java）

```

public class Main {
    public static void main(String[] args) {
        BookShelf bookShelf = new BookShelf(4);
        bookShelf.appendBook(new Book("Around the World in 80
Days"));
        bookShelf.appendBook(new Book("Bible"));
        bookShelf.appendBook(new Book("Cinderella"));
        bookShelf.appendBook(new Book("Daddy-Long-Legs"));
        Iterator it = bookShelf.iterator();
        while (it.hasNext()) {
            Book book = (Book)it.next();
            System.out.println(" " + book.getName());
        }
    }
}

```

```

    }
}
}

```

一开始野心别太大，先建立一个能放 4 本书的书架就好。依次分别放入以下 4 本书：

- Around the World in 80 Days
- Bible
- Cinderella
- Daddy-Long-Legs

为了让各位更清楚它们的顺序关系，书名按 A, B, C, ... 的英文字母顺序排列。

利用 bookShelf.iterator() 所得到的 it 就是扫描书架用的 Iterator 实例。while 的条件位置写成 it.hasNext()。这样只要有书，while 循环就会进行下去。而且在循环内利用 it.next() 即可逐一检阅所有书籍。

其执行结果如图 1-3 所示。

```

Around the World in 80 Days
Bible
Cinderella
Daddy-Long-Legs

```

图 1-3 执行结果

Iterator Pattern 的所有参与者

在看完程序示例后，请各位综合复习一下 Iterator Pattern 所有曾经出现过的参与者。

◆ Iterator (迭代器) 参与者

定义访问和遍历元素的接口 (API)。在前面的程序示例中，负责这个参与者功能的是 Iterator 接口，它定义能否取得下一个元素相关信息的 hasNext 方法以及下一个元素的 next 方法。

◆ ConcreteIterator (具体迭代器) 参与者

实际上实现 Iterator 所定义的接口 (API)。在前面的程序示例中，负责这个参与者功能的是 BookShelfIterator 类。这个参与者必须掌握遍历时的必要信息。在程序示

Iterator——迭代器

例当中，BookShelf 类的对象实例存储在 bookShelf 字段，目前该书则存储在 index 字段。

◆ Aggregate（聚合）参与者

定义建立 Iterator 参与者的接口（API）。在前面的程序示例中，负责这个参与者功能的是 Aggregate 接口，它决定了 iterator 方法。

◆ ConcreteAggregate（具体聚合）参与者

实际上实现 Aggregate 所定义的接口（API）。它是实际的 Iterator 参与者，也就是 ConcreteIterator 的对象实例。在前面的程序示例中，负责这个参与者功能的是 BookShelf 类，它实现了 iterator 方法。

上述的 Iterator Pattern 可另外用类图来表示，结果如图 1-4。

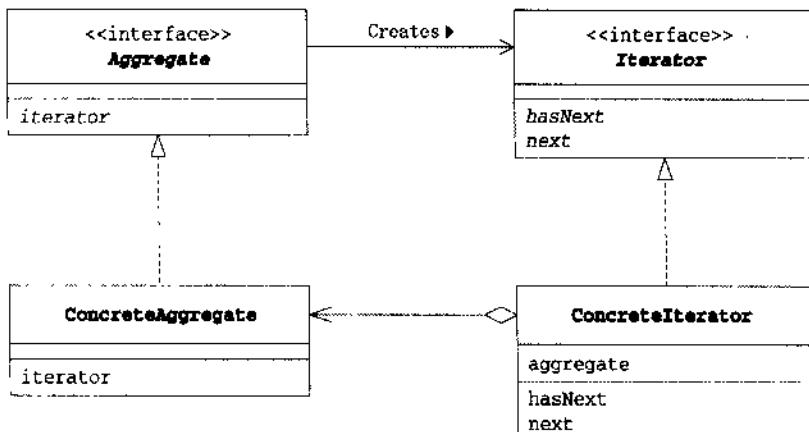


图 1-4 Iterator Pattern 的类图

扩展自我视野的提示

无论实现结果如何，都能使用 Iterator

为什么要在 Iterator Pattern 这么麻烦的东西上花时间？以数组概念的思考 Pattern，用 for 语句让它循环下去不就得了吗？为什么除了聚合之外，还要特别建立像 Iterator 这样的参与者？

最大的理由是因为利用 Iterator 可以跟实现分开，单独进行递增。请看下面的程序代码。

Iterator 模式应用

```

while (it.hasNext()) {
    Book book = (Book)it.next();
    System.out.println(" " + book.getName());
}

```

这里只用到 `hasNext` 和 `next` 这两个 `Iterator` 的方法，并没有调用 `BookShelf` 实现时所使用的方法。换句话说，这里的 `while` 循环不会受到 `BookShelf` 实现的影响。

假设原先实现了 `BookShelf`，但现在不想再利用数组管理书籍，打算把程序修改成能使用 `java.util.Vector`。无论 `BookShelf` 修改成怎样，`BookShelf` 仍然还有 `iterator` 方法，只要能返回正确的 `Iterator`（即返回正常实现 `hasNext`、`next` 方法的类的对象实例），上面的 `while` 循环即使一字不改也能正常执行。

这点在使用 `BookShelf` 上非常有利。设计 Pattern 的目的就是为了提高类的复用率。提高复用率则是指把类当作一个零部件来使用，只要修改某一个零部件，就不需要大费周章去修改其他的零部件。

从这个角度来看，就不难了解为什么前面的程序示例会不把方法 `iterator` 的返回值指定到 `BookShelf Iterator` 类变量，反而指定到 `Iterator` 类变量（程序 1-6）。因为我们不是利用 `BookShelfIterator` 的方法来写程序，而只是打算利用 `Iterator` 的方法来写程序。

抽象类、接口实在很难搞

如果对如何使用抽象类和接口还不太清楚的话，很容易一脚栽进用 `ConcreteAggregate` 或 `Concrete Iterator` 来写程序的泥潭。因为只用具体类就能解决所有问题的感觉会让人不知不觉上瘾。

但是，过度依赖具体类反而会提高类与类的耦合度，增加零部件复用的困难。为了降低耦合度，让类作为零部件再利用，必须引进抽象类和接口的概念。

这也是本书中经常出现的重要概念，即使现在没办法马上适应，相信各位只要耐着性子阅读下去应该就能逐渐体会到其中真味。“不要只用具体类写程序，要利用抽象类和接口才对”，请把这个概念牢记在你的大脑内。

Aggregate 与 Iterator 的对应

请各位回想一下，前面提过把 `BookShelfIterator` 类定义为对应 `BookShelf` 类的 `ConcreteIterator` 参与者。`BookShelfIterator` 非常了解 `BookShelf` 整个实现过程，也因为它了解实现，所以才能调用取得“下一本”书的方法 `getBookAt`。

如果 `BookShelf` 的实现全部改变，包括这个接口的 `getBookAt` 方法（API）也有变动的话，就必须修改 `BookShelfIterator`。

Iterator——迭代器

就像 Aggregate 和 Iterator 这两个接口形影不离的关系一样，BookShelf 跟 BookShelfIterator 两个类也是焦不离孟、孟不离焦。

“下一个”容易搞错

名称中含有 next 的方法都很容易被弄错。到底 next 方法的返回值是当前元素，还是下一个元素？你弄得清楚吗？讲得明白一点，next 方法应该也可以称为：

`returnCurrentElementAndAdvanceToNextPosition`

换句话说，它其实是“返回现在的元素，同时进到下一个位置”的意思。

“最后一个”也容易错

虽然说“下一个”容易搞错，“最后一个”也同样让人搞不清楚。hasNext 方法在取得最后一个元素之前会返回 true，不过当它取得最后一个元素之后却会返回 false。写程序若不稍微注意一点，可能会少返回最后 1 个。

其实只要把 hasNext 方法记成“检查接下来是否可以调用 next 方法”，应该就没问题了。

一个以上的 Iterator

“把递增的架构放在 Aggregate 参与者之外”是 Iterator Pattern 的特征之一。利用这个特点可以对 1 个 ConcreteAggregate 参与者建立出一个以上（即多个）的 ConcreteIterator 参与者。

各种 iterator

前面程序示例所建立的 Iterator 类比较简单，只需从头到尾遍历一次而已。不过，遍历也有以下几种不同的方式。例如：

- 由后往前的反向遍历
- 由前往后、由后往前的双向遍历（不只 next 方法，还有 previous 方法）
- 指定号码，立即跳跃过去遍历

类似这样的 Iterator 类也都能做得出来。

不需要 deleteIterator

在 Java 中，没有使用到的对象实例（instance）会被自动删除（垃圾收集：garbage collection）。所以就不需要有 iterator 所对应的 deleteIterator 方法。

相关 Pattern

◆ *Visitor Pattern* (第 13 章)

Iterator Pattern 是从聚合逐一取出元素并递增上去。虽然递增这个操作是为了对元素进行某种处理，但 Iterator 接口内并没有实现该处理。

Visitor Pattern 则是穿梭在多个元素的聚合内，不断重复同一处理。

相信各位经常会遇到既要计算还要同时做定型处理的例子，Visitor Pattern 简直可以说是专为这个用途而生的 Design Pattern。穿梭在多个元素的聚合内，不断重复同一处理，这就是 Visitor Pattern。

◆ *Composite Pattern* (第 11 章)

Composite Pattern 是一种具有递归结构的 Pattern。Iterator Pattern 不适用于这个部分。

◆ *Factory Method Pattern* (第 4 章)

iterator 方法在建立 Iterator 的对象实例时，有时会使用到 Factory Method Pattern。

重点回顾

本章的学习重点是如何用固定的方法递增计算聚合内元素的 Iterator Pattern。
接下来，请各位动动脑筋做一下后面的练习题。

练习题

解答请见附录 A

问题 1

在程序示例 BookShelf 类（程序 1-4）中，若书籍数量超过最先设定的书架大小，就无法继续把书放上去。请利用 java.util.Vector 取代数组，把程序改成即使已经超过书架容量也能继续新增书籍。

第 2 章

Adapter (适配器) ——换个包装再度 利用

Adapter Pattern

如果现在有一台使用 12V 直流电的笔记本电脑，要改接到 220V 交流电的家用电源时，你需要用到交流电适配器（AC Adapter）。交流电适配器能将家用电源的“220V 交流电”转换成我们所需要的“12V 直流电”。适配器的功能就是介入既有内容和需要结果之间，作为沟通的桥梁。适配器的英文原文是 *adapter*，代表“adapt（适合）”的意思。

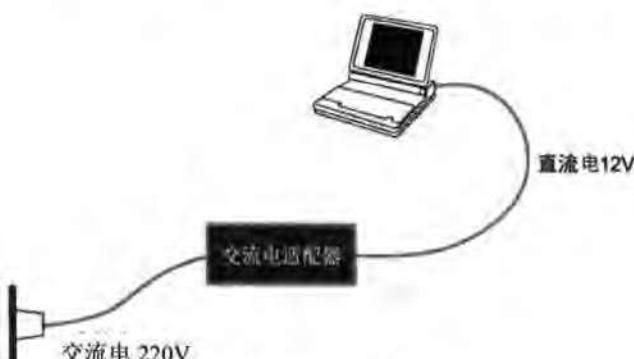


图 2-1 适配器的功能

在程序设计的世界里也是如此，如果既有内容无法直接利用时，通常需要先转换成必要的类型后再使用。具有填平“既有内容”和“需要结果”两者间“落差”的 Design Pattern 就是：

Adapter Pattern

Adapter Pattern 又称为 Wrapper Pattern。*wrapper* 这个字有“包装纸”的意思。就好像一个本来平淡无奇的商品经过美观漂亮的包装纸装扮之后，就能摇身一变作为送人的礼品，某样内容经过包装改头换面后即可使用在其他用途，具有这个转换能力的就可引申为包装器或适配器。

Adapter Pattern 有下列 2 种：

- 类的 Adapter Pattern（继承）
- 对象的 Adapter Pattern（委托）

本章依次介绍这 2 种 Adapter Pattern。

程序示例（1）（继承）

以下程序示例是利用类的 Adapter Pattern，此示例不难，仅把字符串输出成：

`(Hello)`

Adapter (适配器) —— 换个包装再度利用

或

Hello

Banner 类 (banner 是“广告板”的意思) 有两种方法, 一是字符串放在小括号内后再输出的 showWithParen 方法, 另一种是字符串前后各加 1 个*后再输出的 showWithAster 方法。假设这个 Banner 类跟前面交流电 220V 一样都是“既有的内容”。

而 Print 接口则是声明先调弱字符串 (加小括号) 再输出的方法 printWeak (weak 是微弱的意思) 以及先加强字符串 (前后加*表示强调) 再输出的方法 printStrong (strong 是强大的意思)。假设这个接口跟前面的直流电 12V 一样都是“需要的结果”。

接下来要建立一个能使用 Banner 类来满足 Print 接口的类, 换句话说, 就是把交流电 220V 转换成直流电 12V 的适配器。

负责适配器功能的是 PrintBanner 类, 它会继承既有的 Banner 类, 然后实现需要的 Print 接口。Print Banner 类分别利用 showWithParen 方法、showWithAster 方法来实现 printWeak 和 printStrong。如此一来, PrintBanner 类就发挥了它适配器的功能。使用电源与程序示例的对应关系即如表 2-1 所示。

表 2-1 使用电源与程序示例的对应关系

	电源	程序示例
既有的内容	交流电 220V	Banner 类 (showWithParen, showWithAster)
转换装置	适配器	PrintBanner 类
需要的结果	直流电 12V	Print 接口 (printWeak, printStrong)

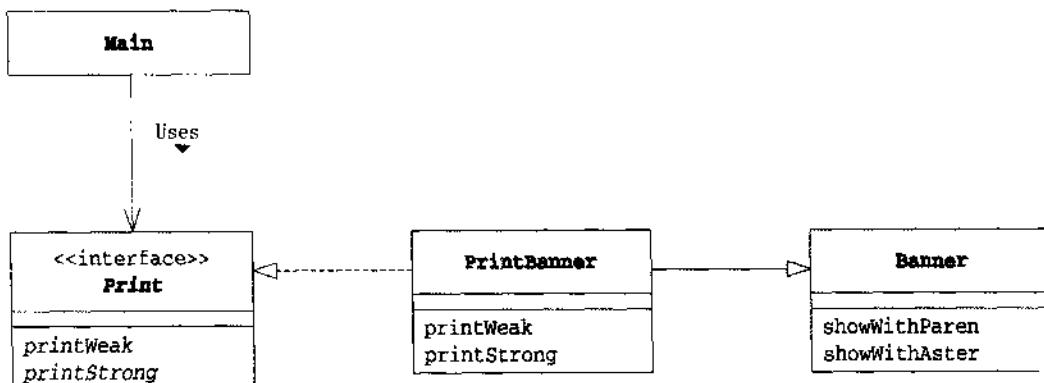


图 2-2 类图——类的程序示例 (继承)

Banner 类

Banner 类 (程序 2-1) 是既有的类。

程序 2-1 Banner 类 (Banner.java)

```
public class Banner {
    private String string;
    public Banner(String string) {
        this.string = string;
    }
    public void showWithParen() {
        System.out.println("(" + string + ")");
    }
    public void showWithAster() {
        System.out.println("*" + string + "*");
    }
}
```

Print 接口

Print 接口（程序 2-2）是需要的接口。

程序 2-2 Print 接口 (Print.java)

```
public interface Print {
    public abstract void printWeak();
    public abstract void printStrong();
}
```

PrintBanner 类

PrintBanner 类（程序 2-3）发挥适配器的功能，它先扩充（extends）既有的 Banner 类，然后继承 showWithParen 方法和 showWithAster 方法。另外还要先实现（implements）需要的 Print 接口，再实现 printWeak 方法和 printStrong 方法。

程序 2-3 PrintBanner 类 (PrintBanner.java)

```
public class PrintBanner extends Banner implements Print {
    public PrintBanner(String string) {
        super(string);
    }
    public void printWeak() {
        showWithParen();
    }
}
```

Adapter (适配器) —— 换个包装再度利用

```

    }
    public void printStrong() {
        showWithAster();
    }
}

```

Main 类

Main 类 (程序 2-4) 则利用前面所建立的 PrintBanner 类 (即适配器功能) 减弱 (小括号)、或加强 (前后各一个*) 字符串 Hello 后再输出。

程序 2-4 Main 类 (Main.java)

```

public class Main {
    public static void main(String[] args) {
        Print p = new PrintBanner("Hello");
        p.printWeak();
        p.printStrong();
    }
}

```

```
(Hello)
*Hello*
```

图 2-3 执行结果

请注意，这里是把 PrintBanner 的对象实例指定到 Print 接口类型的变量。Main 类只不过是利用 Print 这个接口（即利用 printWeak 方法和 printStrong 方法）来写程序而已。从 Main 类的源代码完全看不出有 Banner 类、showWithParen 方法或 showWithAster 方法。就好像是笔记本电脑只要有人提供 12V 直流电的电流就能正常运作，根本不知道它的原形（适配器的另一端）竟然是 220V 交流电。

Main 类并不知道系统会以什么样的方式执行 PrintBanner 类。换种说法，这个不知道也就是指不需要去修改 Main 类，就能修改 PrintBanner 类的实现。

程序示例 (2) (委托)

前面的程序示例是“类”的 Adapter Pattern，这里要继续介绍“对象”的 Adapter Pattern。前面的程序示例利用“继承”让“既有内容”符合 (conformance) “需要结果”，在此则是使用“委托”来完成同样的功能。

▶▶ 补充说明：委托

“委托”这个名词感觉上有点难以理解，主要是指把事情“交给谁”的意思。例如，无法出席重要会议时要写委托书，内容大概是“由于我无法亲自出席，特此委任王大明担任我的代理人”。委托跟委任大致相同。Java语言中的委托是指把某个方法的实际处理交给其他对象的方法进行。

Main类、Banner类同前面的程序示例(1)。但Print是假设为类而非接口(程序2-5)。

也就是说，这里是要利用Banner类来实现另一个跟Print类有相同方法的类。记得吗？Java语言是无法同时继承两个类(单一继承)。因为Java语言无法把PrintBanner类定义成Print和Banner这两者的子类。

PrintBanner类(程序2-6)把Banner类的对象存储在banner字段。这个对象是利用PrintBanner类的构造函数所产生的。而在printWeak和printStrong方法再利用这个banner字段调用出showWithParen和showWithAster方法。

前面的示例是调用从自己的父类继承来的showWithParen和showWithAster方法，这里则是通过字段调用。

这里就出现了委托关系。当系统调用PrintBanner类的printWeak方法时，并不是自己动手处理，而是把工作交给其他对象(Banner的对象)的showWithParen方法。

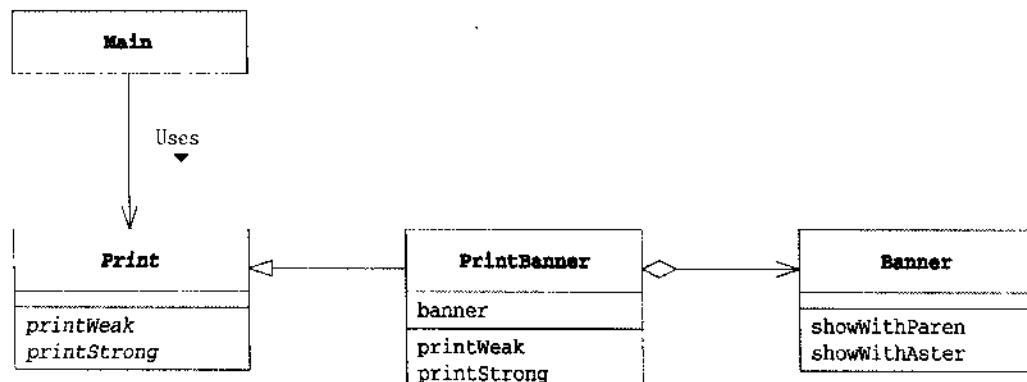


图2-4 类图——对象的程序示例(委托)

Print类

程序2-5 Print类(Print.java)

```

public abstract class Print {
    public abstract void printWeak();
    public abstract void printStrong();
}
  
```

Adapter (适配器) —— 换个包装再度利用

PrintBanner 类

程序 2-6 PrintBanner 类 (PrintBanner.java)

```
public class PrintBanner extends Print {
    private Banner banner;

    public PrintBanner(String string) {
        this.banner = new Banner(string);
    }

    public void printWeak() {
        banner.showWithParen();
    }

    public void printStrong() {
        banner.showWithAster();
    }
}
```

Adapter Pattern 的所有参与者

Adapter Pattern 中所出现过的参与者可整理如下。

◆ Target (对象) 参与者

决定现在需要什么方法的参与者，例如笔记本电脑必须要有 12V 直流电才能工作。在程序示例中，负责这个参与者的是 Print 接口（继承）和 Print 类（委托）。

◆ Client (委托人) 参与者

利用 Target 参与者的方法来做事的参与者，例如有 12V 直流电才能用的笔记本电脑。在程序示例中，负责这个参与者的是 Main 类。

◆ Adaptee (被动符合) 参与者

是 Adapt-ee (被动符合) 而非 Adapt-er (主动符合)。具有既有方法的参与者，也就是 220V 交流电的交流电源。在程序示例中，负责这个参与者的是 Banner 类。

如果担任 Adaptee 参与者的方法与 Target 参与者的方法相符时（即若家用电源为 12V 直流电），那就不需要后面的 Adapter 参与者了。

◆ Adapter 参与者

Adapter Pattern 的核心人物。利用 Adaptee 参与者的方法努力满足 Target 参与者的要求是 Adapter Pattern 的天赋使命，这是 Adapter 参与者的工作。例如，把 220V 交流电转换成 12V 直流电的适配器。在程序示例中，负责这个参与者的是 PrintBanner 类。

如果是类的 Adapter Pattern，Adapter 参与者要通过“继承”的方式来利用 Adaptee 参与者。

而换到对象的 Adapter Pattern 时，就要改以“委托”来利用 Adaptee 参与者。

以上 2 个 Adapter Pattern 的类图如图 2-5、图 2-6 所示。

扩展自我视野的提示

什么时候用

有些读者可能会问“为什么不等到需要用那个方法时，再动手编写？花时间写 Adapter Pattern 真的有必要吗？”。到底什么时候会需要用到 Adapter Pattern 呢？

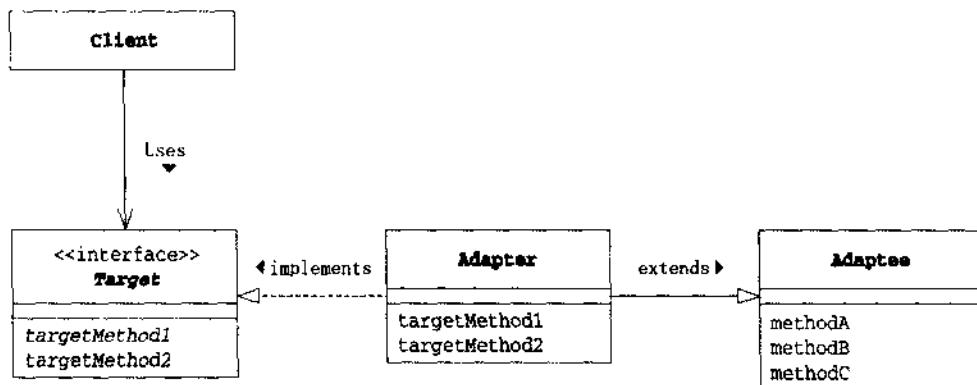


图 2-5 类图 - 类的 Adapter Pattern (继承)

写程序时不一定每次都要从零出发，我们经常会利用一些既有的类。尤其是那些已经做过太多测试知道 bug 不多，或是曾经用过觉得不错的类，既然好用，当然想尽量多利用这些类。

Adapter Pattern 可以把过去使用过的类换个包装重新建立出需要的类，这个 Pattern 可以帮你节省建立必要方法组群的时间。万一程序写着写着发现有 bug 时，因为可以确定既有类（Adaptee 参与者）没有 bug，所以只要重点检查 Adapter 参与者的类即可，减轻程序检查的负担。

Adapter（适配器）——换个包装再度利用

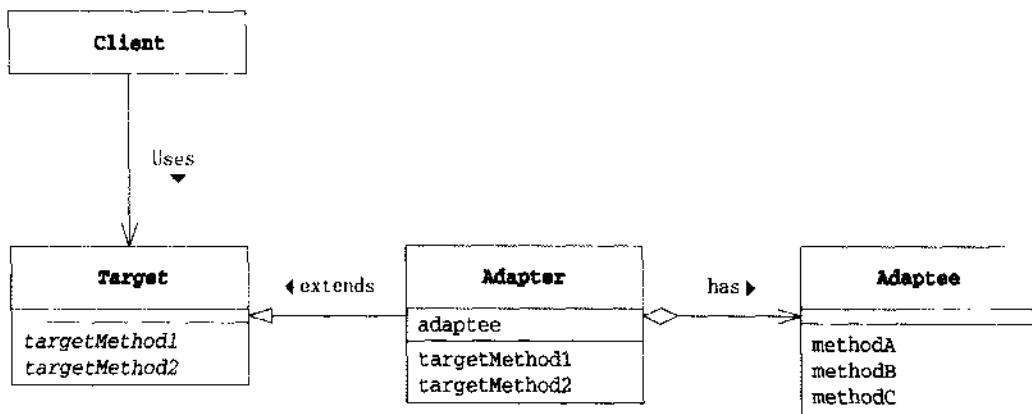


图 2-6 类图 – 对象的 Adapter Pattern（委托）

就算没有源程序也无妨

如果想要让已经建立好的类符合新建的接口（API），Adapter Pattern 应该就是最好的做法。但是我们经常会不自觉地陷入一个误解的圈套，误认为必须修改既有类的程序代码才能让它符合新建的接口（API）。事情可没那么简单，一旦执行这个修改操作，原本已经完成功能测试的现有类就得再重新测试！

Adapter Pattern 的目的是不必改动现有类就能让它符合该接口（API）。而且 Adapter Pattern 也不一定要取得既有类的源程序。Adapter Pattern 只要知道现有类的规格，就能建立其他新的类。

版本更新与兼容性

凡是软件都会需要做版本更新，在做软件的版本更新时最常碰到的问题应该是“与旧版本的兼容性”。如果干脆把旧版本整个删除，软件维护当然就轻松许多，不过这个手段可不是万灵丹。有时候利用 Adapter Pattern 能让新旧版本共存，而且维护也很容易。

假设现在完成版本更新后，就只着重在新版本而不想再维护旧版本的部分。此时，把新版本视为 Adapter 参与者，旧版本则为 Target 参与者，接下来只要建立一个担任 Adapter 参与者的类，让它利用新版本的类来实现旧版本的方法即可。

这个关系的概念图如图 2-7 所示（请注意，这不是 UML 类图）。

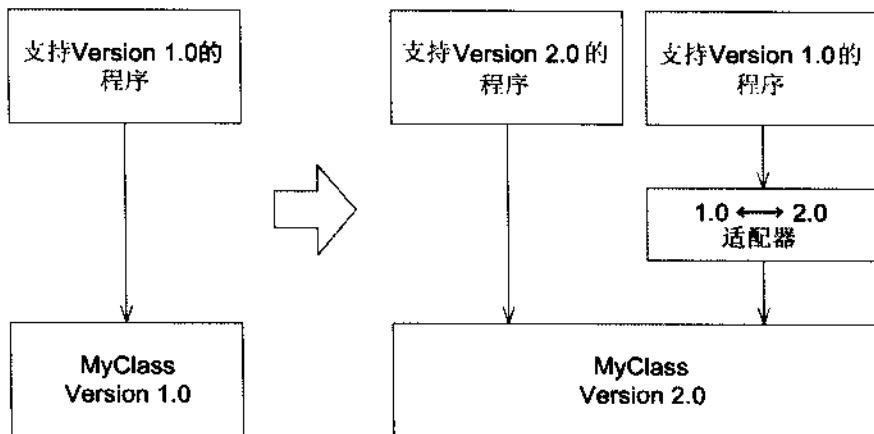


图 2-7 与旧版本取得兼容性的 Adapter Pattern

类差别太大

如果 Adapter 参与者跟 Target 参与者的功能相差太多的话，当然就不能使用 Adapter Pattern。想想，你可能从 220V 交流电的电源流出自来水吗？

相关 Pattern

◆ Bridge Pattern (第 9 章)

Adapter Pattern 是用来连接相异接口 (API) 的类时所使用的 Pattern。Bridge Pattern 是连接功能阶段和实现阶段的 Pattern。

◆ Decorator Pattern (第 12 章)

Adapter Pattern 是填补接口 (API) 间差距的 Pattern。Decorator Pattern 是不需更改接口 (API) 即可新增功能的 Pattern。

重点回顾

在本章所学习到的是连接两个接口 (API) 相异的类并填平其差距的 Adapter Pattern。分别介绍利用继承关系、委托关系的做法，同时说明特征。

各位觉得如何，是不是开始习惯设计 Pattern 了？别忘了要做练习题喔。

练习题

解答请见附录 A

问题 1

本章的程序示例中，在建立 PrintBanner 类的对象时，把它指定到 Print 类变量（程序 2-4），即：

```
Print p = new PrintBanner("Hello");
```

为什么不采取下面这个指定到 PrintBanner 类变量的做法呢？

```
PrintBanner p = new PrintBanner("Hello");
```

问题 2

`java.util.Properties` 类是用来管理键值和相对应数值（即内容）的，如：

```
year=2000
```

```
month=11
```

```
day=20
```

`java.util.Properties` 类有下面几个方法可从数据流（stream）读取内容或反向写入到数据流。

从 `InputStream` 读取内容的集合。

```
void load(InputStream in) throws IOException
```

把内容的集合写入到 `OutputStream`。header 是注释字符串。

```
void store(OutputStream out, String header) throws IOException
```

请利用 Adapter Pattern 建立一个把内容的集合存储成文件的 `FileProperties` 类。

假设在此是以程序 2-7 的 `FileIO` 接口（Target 参与者）声明把内容的集合存储成文件的方法，`FileProperties` 类则是实现这个 `FileIO` 接口。

执行前和执行后的 `file.txt`、`newfile.txt` 如程序 2-9 和程序 2-10 所示（若该行前缀是#，则为 `java.util.Properties` 类自动产生的注释）。

如果有 `FileProperties` 类，就算不知道 `java.util.Properties` 类的方法，只要知道 `FileIO` 接口的方法就能处理内容。

以电源来比喻的话，`java.util.Properties` 类是既有的 220V 交流电、`FileIO` 接口是现在需要的 12V 直流电，而 `FileProperties` 类则为适配器。

程序 2-7 FileIO 接口（FileIO.java）

```
import java.io.*;

public interface FileIO {
    public void readFromFile(String filename) throws IOException;
    public void writeToFile(String filename) throws IOException;
```

Java语言基础应用

```
public void setValue(String key, String value);  
public String getValue(String key);  
}
```

程序 2-8 Main 类 (Main.java)

```
import java.io.*;  
  
public class Main {  
    public static void main(String[] args) {  
        FileIO f = new FileProperties();  
        try {  
            f.readFromFile("file.txt");  
            f.setValue("year", "2000");  
            f.setValue("month", "11");  
            f.setValue("day", "20");  
            f.writeToFile("newfile.txt");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

程序 2-9 读取文件 (file.txt)

year=1999

程序 2-10 输出文件 (newfile.txt)

```
#written by FileProperties  
#Thu Jan 24 10:27:07 CST 2002 ←显示文件建立的日期  
day=20  
year=2000  
month=11
```

第3章

Template Method (模板方法) — 实际处理交给子类

Template Method Pattern

何谓模板 (Template)

模板 (Template) 是指在薄片塑料板上面写字 (或印字) 后挖空，再使用毛笔或色笔涂满挖空部分，就能看到纯手工而又不失工整的字样。看到模板上的挖空形状，马上就知道最后会变出什么样的字，不过实际上所显现出来的字样还是要依使用的画笔种类而定。拿黑色签字笔当画笔，成果当然就是签字笔风的字样；光用铅笔来画，得到的也只会是灰黑色的铅笔字；如果用五颜六色的彩色笔，自然能创造出让人眼花撩乱的多色字样。但是，无论使用哪种文具，制作出来的字样都还是脱不了模板上已经固定的形状。

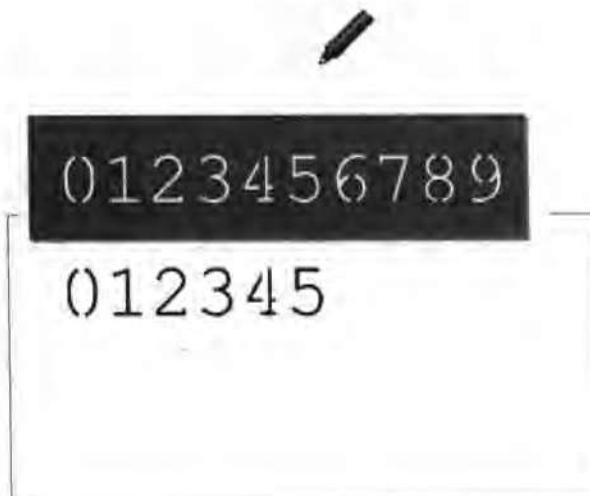


图 3-1 用签字笔描画模板的结果

Template Method Pattern 是什么

本章所要介绍的 Template Method Pattern 是一种具有模板功能的 Pattern。作为模板的方法要定义在父类，而方法的定义中使用到抽象方法。因此如果只看父类部分的程序，根本不知道到底会是怎样的处理内容，最多只能了解该如何调用抽象方法而已。

实际实现抽象方法的是子类。要在子类实现方法，才能决定具体的操作。理论上，如果在不同的子类执行不同的实现，应该就能发展出不同的处理内容。不过，无论在哪个子类执行任何一种实现，处理的大致流程都还是要依照父类所制定的方式。

Template Method (模板方法) —— 实际处理交给子类

像这样在父类指定处理大纲、在子类规定具体内容的 Design Pattern 就称为：

Template Method Pattern

本章将带领各位探寻 Template Method Pattern 的奥妙精髓。

程序示例

请看下面应用 Template Method Pattern 的程序示例。这个程序示例比较简单，只是反复输出 5 次同一字符或字符串。

在这里出现了 AbstractDisplay、CharDisplay、StringDisplay 和 Main 四个类。

AbstractDisplay 类定义 display 方法，而 display 方法之内又使用到 open、print、close 这 3 个方法。AbstractDisplay 类声明 open、print、close 这 3 个方法，但却是没有实例的抽象方法。在此，使用了抽象方法的 display 方法就是 Template Method。

实际上实现 open、print、close 方法的是 AbstractDisplay 类的子类：CharDisplay 类和 StringDisplay 类。

Main 类则是测试用的类。

表 3-1 类一览表

名称	说明
AbstractDisplay	只实现方法 display 的抽象类
CharDisplay	实现方法 open、print、close 的类
StringDisplay	实现方法 open、print、close 的类
Main	测试用的类

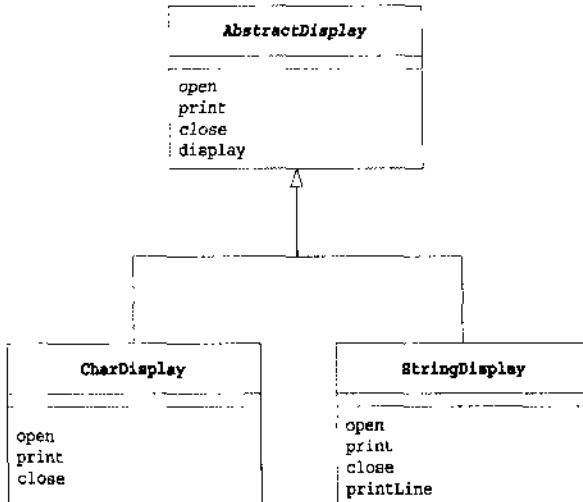


图 3-2 程序示例的类图

AbstractDisplay 类

AbstractDisplay 类（程序 3-1）是具有 open、print、close、display 等方法的类。当中的 open、print、close 都是抽象方法，只有 display 方法被实现。只要查到 AbstractDisplay 类里对 display 方法的定义，就会知道它所进行的处理为：

- 调用 open 方法
- 调用 5 次 print 方法
- 调用 close 方法

而 open、print、close 这三个方法究竟有什么作用？其实从 AbstractDisplay 类就可以看出这三者都是抽象方法；光看 AbstractDisplay 类是无法知道 AbstractDisplay 类中的 display “实际上” 在做什么。因为 display 的实际行为还放在实现 open、print、close 的子类里。

程序 3-1 AbstractDisplay 类（AbstractDisplay.java）

```
public abstract class AbstractDisplay { //抽象类 AbstractDisplay
    public abstract void open();      // 由子类实现的抽象方法(1)open
    public abstract void print();     // 由子类实现的抽象方法(2)print
    public abstract void close();     // 由子类实现的抽象方法(3)close
    public final void display() {
        open(); // 先 open...
        for (int i = 0; i < 5; i++) { //反复 5 次 print...
            print();
        }
        close(); // ...最后 close 起来。这就是实现 display 方法时的内容
    }
}
```

CharDisplay 类

如果各位对前面的说明都能理解的话，就可以继续来看 CharDisplay 子类（程序 3-2）。由于在父类 AbstractDisplay 类中被列为抽象方法的 open、print、close 都已经被实现过了，所以 CharDisplay 类并不是抽象类。

CharDisplay 类的 open、print、close 会进行如表 3-2 所示的处理。

表 3-2 在 CharDisplay 类中 open、print、close 方法的处理内容

方法名称	处理内容
open	输出字符串“<<”

Template Method（模板方法）——实际处理交给子类

续上表

方法名称	处理内容
print	输出1个以构造函数传递的字符
close	输出字符串">>"

在这个情形下，如果调用 `display` 方法的话会变成怎样？举例来说，如果把字符'H'传递给构造函数，则画面上会输出如下的字符串：

<<HHHH>>

程序 3-2 CharDisplay 类 (CharDisplay.java)

```
public class CharDisplay extends AbstractDisplay {
    // CharDisplay 是 AbstractDisplay 的子类。
    private char ch; // 应输出的字符。
    public CharDisplay(char ch) { // 把构造函数传递过来的字符 ch
        this.ch = ch; // 存储在字段内。
    }
    public void open() { // 本来在父类时是抽象方法。
        // 这里重载 (override) 后实现。
        System.out.print("<<"); // 输出"<<"作为开始字符串。
    }
    public void print() { // print 方法也是在此实现。它是从 display
        // 被重载后实现。
        System.out.print(ch); // 输出1个存储在字段的字符。
    }
    public void close() { // close 方法也是在此实现。
        System.out.println(">>"); // 输出结束字符串">>"。
    }
}
```

StringDisplay 类

再来看另一个子类——`StringDisplay` 类（程序 3-3）。当然 `open`、`print`、`close` 在这里也被实现。那么，这个部分是做什么样的处理呢？

`StringDisplay` 的 `open`、`print`、`close` 所进行的处理如表 3-3 所示。

在这个情形下，如果调用 `display` 方法的话会变成怎样？当字符串"Hello, world."被传递给构造函数之后，画面上会输出有外框的字符串：

```
+-----+
|Hello, world.|
+-----+
```

Java语言中的应用

```
|Hello, world.|  
|Hello, world.|  
|Hello, world.|  
|Hello, world.|  
+-----+
```

表 3-3 在 StringDisplay 类中 open、print、close 方法的处理内容

方法名称	处理内容
open	输出字符串" +----+"
print	先在构造函数所传递的字符串前后加上" "和" ", 再输出
close	输出字符串" +----+"

程序 3-3 StringDisplay 类 (StringDisplay.java)

```
public class StringDisplay extends AbstractDisplay {  
    //StringDisplay 也是 AbstractDisplay 的子类。  
    private String string; // 应输出的字符串。  
    private int width; // 以 byte 为单位所求出的字符串“宽度”。  
    public StringDisplay(String string) { // 把构造函数传递过来的  
        // 字符串 string 存储在字段内。  
        this.string = string;  
        this.width = string.getBytes().length;  
        // 接着把以 byte 为单位的字符宽度也先存储在字段，后面还会使用到。  
    }  
    public void open() { // 先重载原先定义的 open 方法。  
        printLine(); // 以此类的方法 printLine 画出线段。  
    }  
    public void print() { // print 方法在字段所存储的字符串前后加上  
        // "||", 然后输出到画面上。  
        System.out.println("||" + string + "||");  
    }  
    public void close() { // close 方法则同 open，利用 printLine 方  
        // 法画出线段。  
        printLine();  
    }  
    private void printLine() { // 这是 open 和 close 所调用的 printLine  
        // 方法。因为这是 private，所以只能在此类内部使用。  
        System.out.print("+"); // 输出 "+" 号表示边框位置。  
        for (int i = 0; i < width; i++) { // 输出 width 个 "-",  
            System.out.print("-"); // 当作线段。  
        }  
        System.out.println("+"); // 输出 "+" 号表示边框位置。  
    }  
}
```

Template Method (模板方法) —— 实际处理交给子类

Main 类

Main 类执行测试，它先建立前面所建的 CharDisplay 类和 StringDisplay 类的对象实例，再调用 display 方法。

程序 3-4 Main 类 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        // 建立 1 个有'H'的 CharDisplay 的对象。
        AbstractDisplay d1 = new CharDisplay('H');
        // 建立 1 个有"Hello, world."的 StringDisplay 的对象。
        AbstractDisplay d2 = new StringDisplay("Hello, world.");
        // 建立 1 个有"你好。"的 StringDisplay 的对象。
        AbstractDisplay d3 = new StringDisplay("你好。");
        d1.display();      // d1,d2,d3 都是 AbstractDisplay 的子类的对象,
                           // 因此,
        d2.display();      // 可以调用继承到的 display 方法。
        d3.display(); // 实际动作则规定在类 CharDisplay、StringDisplay。
    }
}
```

<<HHHHHH>>	← d1 的输出结果 (CharDisplay)
+-----+	← d2 的输出结果 (CharDisplay)
Hello, world.	
+-----+	
+-----+	← d3 的输出结果 (CharDisplay)
你 好。	
+-----+	

图 3-3 执行结果

Template Method Pattern 的所有参与者

Template Method Pattern 中所出现过的参与者可整理如下。

◆ AbstractClass (抽象类) 参与者

AbstractClass 参与者实现模板，声明 Template Method 所使用的抽象方法。这个抽象方法由子类 ConcreteClass 参与者负责实现。在程序示例中，负责这个参与者的是 AbstractDisplay 类。

◆ ConcreteClass (具体类) 参与者

具体实现 AbstractClass 参与者所定义的抽象方法。这里所实现的方法从 AbstractClass 参与者的模板调用出来。在程序示例中，负责这个参与者的是 CharDisplay 类和 StringDisplay 类。

Template Method Pattern 的类图如图 3-4 所示。

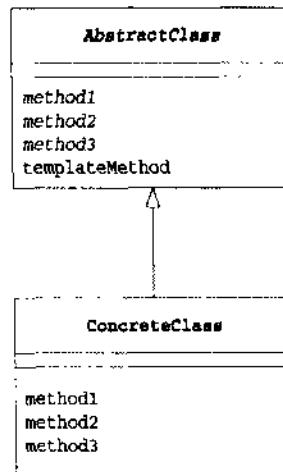


图 3-4 Template Method Pattern 的类图

扩展自我视野的提示

逻辑可共享

Template Method Pattern 究竟有什么好处？因为父类的模板已经实现了算法，所以

Template Method (模板方法) —— 实际处理交给子类

子类就不需要重新逐一实现算法。

假设现在故意不用 Template Method Pattern, 只利用编辑器的剪贴功能来建立多个 ConcreteClass 参与者。ConcreteClass1、ConcreteClass2、ConcreteClass3 等都是一些似是而非的类。刚写完还觉得很满意, 可是要是后来发现 ConcreteClass1 有 1 个 bug, 该怎么办? 只修改 1 个 bug 并没有办法反映到所有的 ConcreteClass 参与者上。

从这个角度来看, 如果利用 Template Method Pattern 来写程序的话, 当你发现模板里面有错误时, 只要修改这个模板就行了。

父类跟子类的连续性

在 Template Method Pattern 当中, 父类跟子类之间的联络互动相当紧密。因此如果要在子类实现一个已经在父类声明过的抽象方法时, 必须先了解应该在哪个时间点调用这个抽象方法。要是没有父类的程序源代码, 恐怕子类的实现会是一个高难度的挑战。

子类应视同父类

在前面的程序示例中, CharDisplay 的对象实例和 StringDisplay 的对象实例都先指定到 AbstractDisplay 类型的变量, 然后才调用 display 方法。

假设现在有一个父类类型的变量, 子类的对象实例也被指定到该变量。此时, 程序最好利用 instanceof 等写成无论子类类型为什么, 程序都能正常执行。

无论父类类型的变量指定哪一个子类的对象实例都能正常执行的原则称为 The Liskov Substitution Principle (LSP) (注)¹。LSP 是继承相关的一般性原则, 并不只限于 Template Method Pattern 才可适用。

相关 Pattern

◆ Factory Method Pattern (第 4 章)

将 Template Method Pattern 应用于产生对象实例的典型示例就是 Factory Method Pattern。

◆ Strategy Pattern (第 10 章)

Template Method Pattern 可以利用“继承”来更改程序, 先以父类规定程序类型的大纲, 然后再用子类规定较具体的行为。

¹ Robert C. Martin C++ Report, March 1996 <http://www.objectmentor.com/publications/lsp.pdf>

相比之下，Strategy Pattern 则可利用“委托”来更改程序。不过，Strategy Pattern 不是修改部分程序，而是切换整个算法。

进阶说明：类层次与抽象类

父类对子类的要求

我们在学习类层次时，通常都是从子类的角度来思考。换句话说，我们会比较着重在：

- 在子类可利用父类中所定义的方法
- 子类只要写一点点方法就可以新增其他功能
- 在子类重载（override）方法就能修改程序行为

这里希望各位尝试改变一下观点，从父类的立场来想想看。假设现在父类已经声明了抽象方法。此时，这个方法的实现要“全权交给子类”。也就是说，当你在声明抽象方法时，事实上是在程序中表示：

- 期待子类会实现这个方法
- 要求子类要实现这个方法

你也可以把它解释成子类产生一个须实现在父类所声明的抽象方法的责任。这称为 subclass responsibility（子类的责任）。

抽象类的意义

抽象类无法建立对象。刚开始学习抽象类时，或许有人会觉得很奇怪：“一个无法建立对象的类究竟扮演什么样的参与者”。当你确实理解本章的 Template Method Pattern 之后，应该就不会再有这个疑问了。因为抽象方法里面没有方法的主体，所以无法得知具体的处理内容。但是，它可以决定方法名称、应用该方法的模板来叙述处理的内容。实际的处理内容当然要等到执行到子类才会确定，不过在抽象类的阶段就先抓出处理流程的类型也很重要。

父类跟子类之间的协调

在设计一个程序时，必须不断协调父类和子类。如果增加父类的代码，就可以减轻子类的负担，但是相对地也会降低子类自由发挥的空间。反过来，如果减少父类的代码，子类的代码实现就会比较棘手，可能会发生不同子类出现同样处理的代码。

Template Method Pattern 是父类实现处理的骨架，把具体的留给子类来做。至于应

Template Method (模板方法) —— 实际处理交给子类

该在什么阶段把处理分开来、哪些处理交给父类、哪些处理该给子类，并没有实战手册可供参考。这个部分就是程序员自行发挥的空间了。

重点回顾

在父类规定处理的骨架，在子类具体描述处理的内容，这个 Template Method Pattern 就是本章所学习的内容。另外也请各位跟着思考一下抽象类的意义和子类的责任。

下一章将继续介绍在产生对象实例时该如何导入 Template Method Pattern 的 Factory Method Pattern。

练习题

解答请见附录 A

问题 1

java.io.InputStream 类用到 Template Method Pattern。请翻阅规范说明（JDK 的 API 参考事项），找出要求在 java.io.InputStream 的子类实现的方法是哪一个。

问题 2

程序示例的 AbstractDisplay 类（程序 3-1）当中的 display 方法会实现成：

```
public final void display() {  
    ...  
}
```

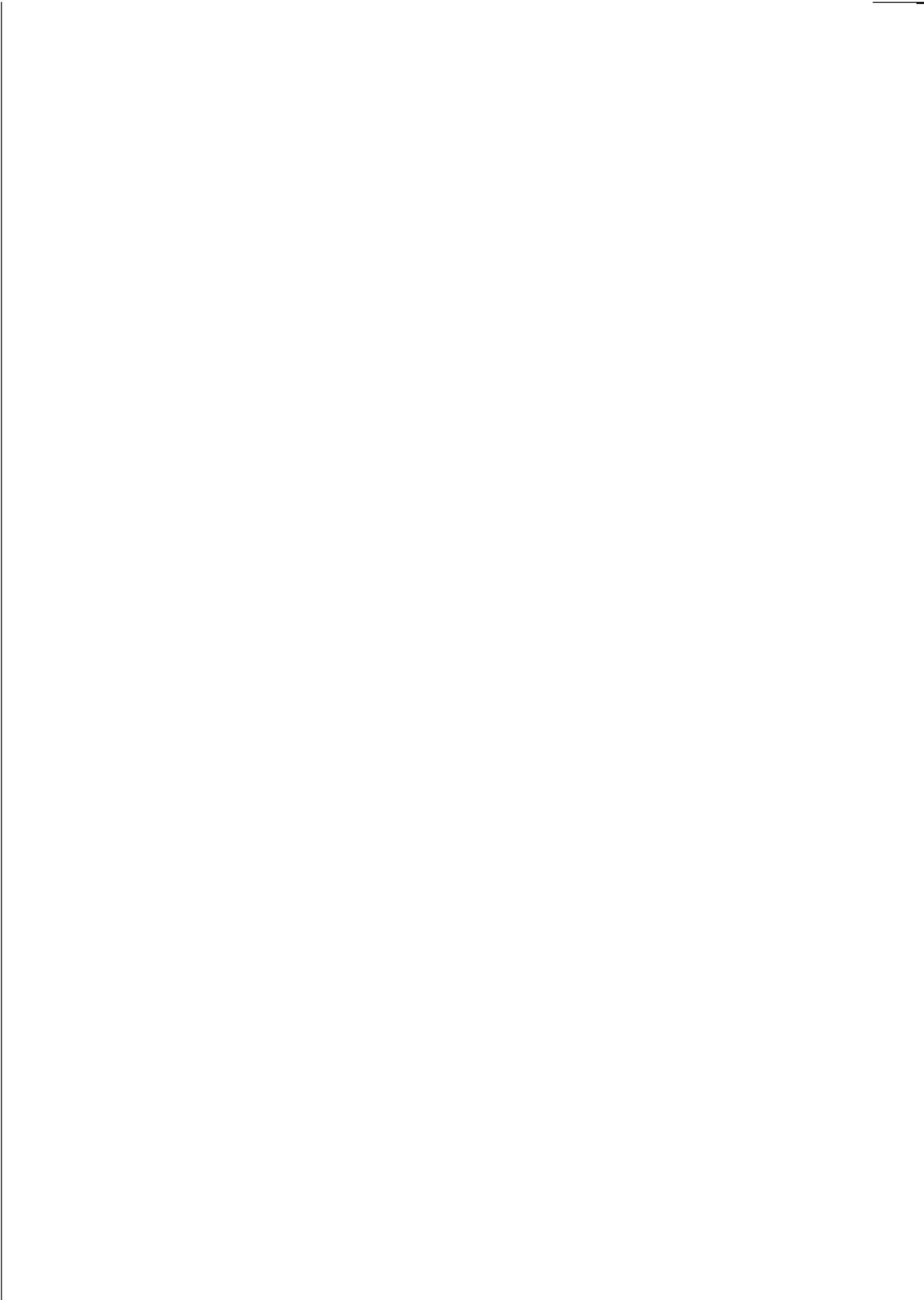
这里的结果是 final，请问有什么意义？

问题 3

如果想把程序示例改成只能从父类调用 open、print、close 方法，不能从其他无关的类任意调用时，应该如何重载？

问题 4

Java 的接口跟抽象类很类似。接口也是抽象方法的集合，不过 Template Method Pattern 可不容许在 AbstractClass 参与者使用接口。为什么？



第 4 章

Factory Method (工厂方法) — 建立对象实例交 给子类

Factory Method Pattern

Template Method Pattern（第3章）是在父类建立处理逻辑的大纲骨架，而在子类补充具体的处理内容。如果把这个Pattern应用在产生对象实例方面，就是本章所要学习的Factory Method Pattern。

factory这个英文单词的意思是“工厂”。以Template Method Pattern架构获取产生对象实例的工厂就是Factory Method Pattern。

Factory Method Pattern在父类规定对象的创建方法，但并没有深入到较具体的类名。所有具体的完整内容都放在子类。根据这个原则，我们可以大致分成产生对象实例的大纲（框架）和实际产生对象实例的类两方面。

程序示例

请看下面应用Factory Method Pattern的程序示例。这个程序示例主要是建立一个生产身份证（ID卡）的工厂，其中共出现5个不同的类。

Product类和Factory类属于“framework”包。这2个类负责建立产生对象实例大纲（框架）的功能。

IDCard类和IDCardFactory类则处理实际的内容，属于“idcard”包的一部分。

Main类是测试用的类。

在阅读本章的程序示例时，请随时注意自己是在读framework包的部分还是idcard包的部分。

- 建立产生对象实例的框架（framework包）
- 实际处理内容的部分（idcard包）

注意：建议各位在开发公开的程序包时，最好直接利用域名来命名程序包，而且排列顺序要反过来，这样就可以保证您的包名是独一无二的。例如，若域名为hyuki.com时，则包名就以com.hyuki为前缀。不过本书为了避免命名过于复杂，所以并没有依据前述的原则命名。

表4-1 类一览表

程序包	名称	说明
framework	Product	仅定义抽象方法use的抽象类
framework	Factory	实现方法create的抽象类
idcard	IDCard	实现方法use的抽象类
idcard	IDCardFactory	实现方法createProduct、registerProduct的类
未命名	Main	测试用的类

Factory Method (工厂方法) —— 建立对象实例交给子类

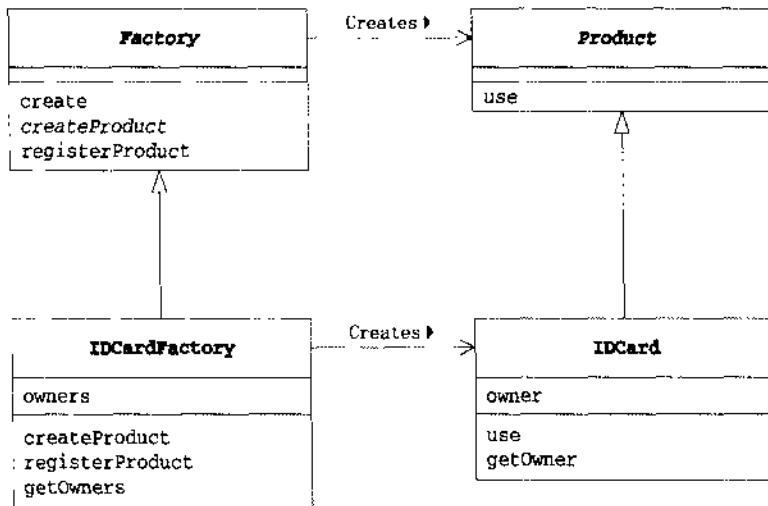


图 4-1 程序示例的类图

Product 类

framework 程序包的 Product 类（程序 4-1）是表示“产品”的类。这个类只声明抽象方法 use 而已，use 的具体实现则由 Product 的子类负责。

在此框架中，产品定义为“不管内容如何都能 use（可用）的东西”。

程序 4-1 Product 类 (Product.java)

```

package framework;

public abstract class Product {
    public abstract void use();
}

```

Factory 类

framework 程序包的 Factory 类（程序 4-2）用到 Template Method Pattern。以抽象方法 createProduct “制作产品”，再利用抽象方法 registerProduct “注册”成品。“制作产品”和“注册”的实现均由子类进行。

在此框架中，工厂定义为“以 create 方法产生 Product 的对象实例的物体”。而 create 方法的实现步骤则为“先以 createProduct 制作产品，再以 registerProduct 完成注册”。

实现的具体内容会因应用 Factory Method Pattern 的程序而不同。不过只要它是 Factory Method Pattern，产生对象实例就会使用到 Template Method Pattern。

程序 4-2 Factory 类 (Factory.java)

```
package framework;

public abstract class Factory {
    public final Product create(String owner) {
        Product p = createProduct(owner);
        registerProduct(p);
        return p;
    }
    protected abstract Product createProduct(String owner);
    protected abstract void registerProduct(Product product);
}
```

IDCard 类

前面说明的是框架部分 (framework 包)，接下来要换到详细处理内容的部分 (idcard 包)。举个例子来说，假设现在要建立一个表示识别号码卡的类 IDCard。我们把这个包命名为 idcard 包，以便跟框架划清界线。IDCard 类 (程序 4-3) 定义为产品 Product 类的子类。

程序 4-3 IDCard 类 (IDCard.java)

```
package idcard;
import framework.*;

public class IDCard extends Product {
    private String owner;
    IDCard(String owner) {
        System.out.println("建立" + owner + "的卡。");
        this.owner = owner;
    }
    public void use () {
        System.out.println("使用" + owner + "的卡。");
    }
    public String getOwner() {
        return owner;
    }
}
```

Factory Method (工厂方法) —— 建立对象实例交给子类

IDCardFactory 类

IDCardFactory 类（程序 4-4）实现了 createProduct 和 registerProduct 这两个方法。createProduct 是利用产生 IDCard 的对象实例来表示有“制造产品”；而 registerProduct 则是把 IDCard 的 owner（持有人）新增到 owners 字段，表示“注册”的功能。

程序 4-4 ICardFactory 类 (IDCardFactory.java)

```
package idcard;
import framework.*;
import java.util.*;

public class IDCardFactory extends Factory {
    private Vector owners = new Vector();
    protected Product createProduct (String owner) {
        return new IDCard(owner);
    }
    protected void registerProduct (Product product) {
        owners.add(((IDCard)product).getOwner());
    }
    public Vector getOwners() {
        return owners;
    }
}
```

Main 类

接下来，在 Main 类（程序 4-5）中利用 framework 包和 idcard 包来建立实际的 IDCard，并加以应用。

程序 4-5 Main 类 (Main.java)

```
import framework.*;
import idcard.*;

public class Main {
    public static void main(String[] args) {
        Factory factory = new IDCardFactory();
        Product card1 = factory.create("结城浩");
```

Factory 模式的作用

```

Product card2 = factory.create("户村");
Product card3 = factory.create("佐藤花子");
card1.use();
card2.use();
card3.use();
}
}

```

建立结城浩的卡。
 建立户村的卡。
 建立佐藤花子的卡。
 使用结城浩的卡。
 使用户村的卡。
 使用佐藤花子的卡。

图 4-2 执行结果

Factory Method Pattern 的所有参与者

Factory Method Pattern 的所有参与者可整理如下。

从类图（图 4-3）可以清楚看出父类（抽象的骨架大纲、框架）的 Creator 参与者和 Product 参与者的关系，跟子类（具体的处理内容）的 ConcreteCreator 参与者和 ConcreteProduct 角色的关系，两者是平行共同存在的。

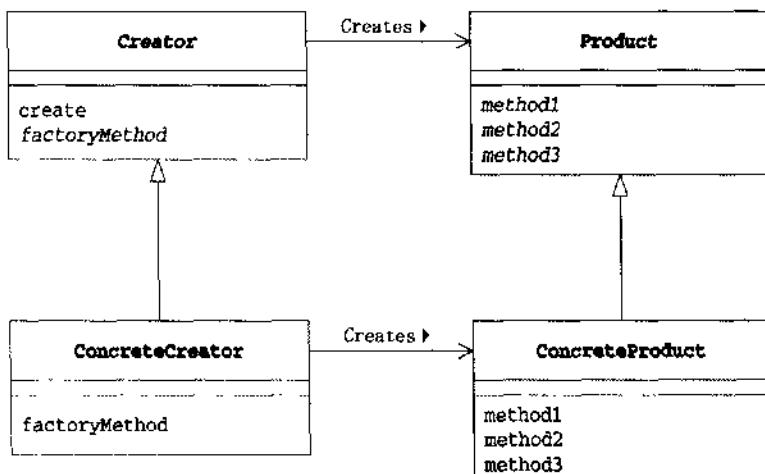


图 4-3 Factory Method Pattern 的类图

Factory Method (工厂方法) —— 建立对象实例交给子类

◆ Product (产品) 参与者

框架的部分。这个抽象类规定此 Pattern 所产生的对象实例应有的接口 (API)，具体内容则由子类的 ConcreteProduct 参与者规定。在前面的程序示例中，负责这个参与者的是 Product 类。

◆ Creator (生产者) 参与者

框架的部分。这是产生 Product 参与者的抽象类。具体的内容由子类的 ConcreteCreator 参与者决定。在前面的程序示例中，负责这个参与者的是 Factory 类。

Creator 参与者对于实际产生的 ConcreteProduct 参与者完全一无所知。Creator 参与者唯一知道的是只要调用 Product 参与者和产生对象的方法 (在图 4-3 中为 factoryMethod)，就能产生 Product。在前面的程序示例中，createProduct 方法是产生对象实例的方法。如果用 new 的实际产生对象实例来取代产生对象实例的方法调用，则可解除实际类名称对父类的约束。

◆ ConcreteProduct (实际产品) 参与者

实际处理内容的部分。规定具体的产品样式。在前面的程序示例中，负责这个参与者的是 IDCard 类。

◆ ConcreteCreator (实际生产者) 参与者

实际处理内容的部分。规定制造实际产品的类。在前面的程序示例中，负责这个参与者的是 IDCardFactory 类。

扩展自我视野的提示

框架与内容

前面谈到“框架”和“内容”这两个部分，分别属于 framework 包和 idcard 包。

假设现在想要利用同一个框架建立不同的“产品”和“工厂”，好比说电视机 Television 类和电视机工厂 TelevisionFactory 类。这时候当然要建立另一个 television 包，并导入 (import) framework 包。

请特别注意一下不必修改 framework 包就能建立完全不同的“产品”和“工厂”。你不需要去修改 framework 包中的内容。

不知道各位还有没有印象，framework 包并没有导入 idcard 包。Product 类和 Factory

类内也没有任何具体的类名（如 `IDCard`、`IDCardFactory`）。因此，如果要用同一框架产生新类时，完全不需要做任何修改，像是把 `television` 包导入到 `framework` 包的行为等。

产生对象实例——方法的实现方式

在程序示例当中，`Factory` 类的 `createProduct` 方法是抽象方法。也就是说，这个方法需要在子类中实现。

`createProduct` 方法的叙述方式有下列 3 种：

◆ 写成抽象方法

写成抽象方法的方式。如果当作抽象方法，子类就一定要实现这个方法。要是没有实现的话，编译时一定会检查出来。本章的程序示例就是使用这种方式。

```
abstract class Factory {
    public abstract Product createProduct(String name);
    ...
}
```

◆ 另外准备缺省的实现

另外准备一个缺省的实现方式。如果另外准备了一个缺省实现，万一遇到子类没有进行实现的时候，就会使用这个实现。

```
class Factory {
    public Product createProduct(String name) {
        return new Product(name);
    }
    ...
}
```

不过这时候它是直接对 `Product` 类用 `new` 分配一个新对象，所以不能把 `Product` 类设为抽象类。

◆ 抛出异常

设成抛出错误的方式。如果预先把缺省实现的内容设为抛出错误的话，当子类没有实现时，程序一执行就会发生错误（如果程序有错误时，就表示没有进行实现）。

```
class Factory {
    public Product createProduct(String name) {
        throw new FactoryMethodRuntimeException();
    }
}
```

Factory Method (工厂方法) —— 建立对象实例交给子类

...

但是这种方式只限于另外还有一个 FactoryMethodRuntimeException 的情形。

Pattern 利用与程序开发工程师之间的沟通

不管是第 3 章的 Template Method Pattern 还是本章的 Factory Method Pattern, 感觉上都比实际的操作步骤复杂。除了深入理解父类的框架、找出其中的抽象方法之外，还要去看实际实现该抽象方法的类的源代码。

一般说来，利用设计 Pattern 设计类群组时，一定要完完整整地把程序设计师所规划的设计 Pattern 和理念传达给那些后续要负责维护类群组的人，否则可能以后修改程序时会跟源程序设计师的规划蓝图渐行渐远。

建议最好把实际使用到的设计 Pattern 名称和设计理念写在程序内的注释或相关文档上，避免发生此类问题。

相关 Pattern

◆ *Template Method Pattern* (第 3 章)

Factory Method Pattern 是最典型的 Template Method Pattern 应用。程序示例中的 create 方法就是模板方法。

◆ *Singleton Pattern* (第 5 章)

负责 Creator 参与者（或 ConcreteCreator 参与者）的类通常都可以用 Singleton Pattern 来做，因为在同一程序中不太需要有两个以上的对象实例。不过，本章的程序示例并没有使用到 Singleton Pattern。

◆ *Composite Pattern* (第 11 章)

在某些情况下，有时可以提供 Composite Pattern 给 Product 参与者（或 ConcreteProduct 参与者）。

◆ *Iterator Pattern* (第 1 章)

如果在 Iterator Pattern 利用 iterator 方法产生 Iterator 的对象实例时，可能会用到 Factory Method Pattern。

重点回顾

在本章学习到把 Template Method Pattern 应用在产生对象实例的 Factory Method Pattern 上。

相信各位一定渐渐习惯了设计 Pattern 的思维模式。在 1 个 Pattern 当中，不同的类和接口负责不同的功能，运行时又保持彼此间的互动关系。当你抓出 1 个类时，不是要追究它是哪种 Pattern，而是要把眼光放在类跟接口之间的互动关系上。毕竟只靠白雪公主一个“角色”，不可能演出白雪公主整出戏。

当然，这世界上还是有舞台上只有 1 个人独撑大局的“独角戏”，下一章就要介绍很类似“独角戏”的 Pattern。

练习题

解答请见附录 A

问题 1

在前面的程序示例中，IDCard 类（程序 4-3）的构造函数并不是 public。这有什么特殊意义？

```
public class IDCard extends Product {  
    ...  
    IDCard(String owner) {  
        ...  
        this.owner = owner;  
    }  
    ...  
}
```

问题 2

请把程序示例的 IDCard 类（程序 4-3）加上卡片的流水号码，让 IDCardFactory 类有一个持有人和流水编号的对照表。

问题 3

假设现在想在 Product 类（程序 4-1）的子类强迫一定要用构造函数把“产品名称”传递出来作为参数，因此重新定义了 Product 类。但是程序编译时却出现错误，为什么？

```
public abstract class Product {  
    public abstract Product(String name);  
    public abstract void use();  
}
```

第 5 章

Singleton (单件) ——唯一的对象实例

Singleton Pattern

通常我们在启动程序时，会产生许多对象实例（Instance）。拿代表字符串的 `java.lang.String` 类的对象实例来说，每个字符串都会相对产生 1 个，所以如果某个程序内含 1,000 个字符串，就会产生 1,000 个对象实例。

不过，有时候难免会有“让这个类的对象实例只产生 1 个”的需要，像是用程序来表现在程序中绝对是独一无二的某个部分。比如说，表现计算机的类，表现目前系统设定的类或是表现窗口系统的类等等就是最典型的例子。

如果写程序时特别注意让 `new MyClass()` 只执行 1 次的话，当然就只会产生 1 个 `MyClass` 的对象实例。但是如果不是“因为程序员有特别处理所以只会产生 1 个对象实例”的情形之下，遇到以下状况时该怎么办？

- 想保证某个特定类的对象实例绝对只有 1 个时
- 想在程序上表达出对象实例只会有 1 个时

能确实保证对象实例只有 1 个的 Pattern 就称为：

Singleton Pattern

`Singleton` 是指只有 1 个元素的集合。就是因为它只会有 1 个对象实例，故而得名。本章将介绍 `Singleton Pattern`。

程序示例

请看下面应用 `Singleton Pattern` 的程序示例。

表 5-1 类一览表

名称	说明
Singleton	只有 1 个对象实例的类
Main	测试用的类

图 5-1 是程序示例的类图。为了表示 `Singleton` 是 `private` 类型，所以在构造函数 `Singleton` 的前面加了“-”。而方法 `getInstance` 的下划线是因为它是 `static` 方法（类方法）（以上是 UML 的限制事项。詳請見本書最前面 UML 部分）。

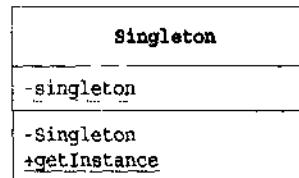


图 5-1 程序示例的类图

Singleton (单件) —— 唯一的对象实例

Singleton 类

Singleton 类（程序 5-1）只会产生 1 个对象实例。Singleton 类把 singleton 定义为 static 字段（类变量），再以 Singleton 类的对象实例进行初始化。这个初始化的操作仅在加载 Singleton 类时进行一次。

Singleton 类的构造函数是 private 的，主要是为了禁止从非 Singleton 类调用构造函数。即使下面这个表达式不在此类之内，编译时仍然会出现错误。

```
new Singleton()
```

其实只要程序员小心不要让程序进行 new 操作的话，就不需要特地把构造函数设成 private 类型。只不过，如此一来 Singleton Pattern 就没有存在的意义了。请容笔者再提醒各位一次，Singleton Pattern 的目的是为了保证即使程序员出错时也只会产生 1 个对象实例。基于这个保证责任，才会需要把构造函数设成 private。

在本章的程序示例中，构造函数内有一个“已产生对象实例。”的文字信息，以便确认操作进行正常。

想要取得 Singleton 类的唯一对象实例的方法就是 getInstance。本章的程序示例直接将方法名称设成 getInstance，不过并不是非这个名称不可，请不要误会。但是这个取得唯一对象实例的方法还得稍微加工一下。

程序 5-1 Singleton 类 (Singleton.java)

```
public class Singleton {
    private static Singleton singleton = new Singleton();
    private Singleton() {
        System.out.println("已产生对象实例。");
    }
    public static Singleton getInstance() {
        return singleton;
    }
}
```

Main 类

Main 类（程序 5-2）是利用 Singleton 的类。在本章的程序示例中，它利用 Singleton 类的 getInstance 方法来取得 Singleton 的对象实例。getInstance 方法共被调用两次。返回值分别指定到 obj1 和 obj2。利用 obj1 == obj2 这个表达式则可检查所取得的对象实例是否相同。

设计模式——单例模式

程序 5-2 利用 Singleton 类的类 (Main.java)

```

public class Main {
    public static void main(String[] args) {
        System.out.println("Start.");
        Singleton obj1 = Singleton.getInstance();
        Singleton obj2 = Singleton.getInstance();
        if (obj1 == obj2) {
            System.out.println("obj1 和 obj2 是同一对象实例。");
        } else {
            System.out.println("obj1 跟 obj2 并非同一对象实例。");
        }
        System.out.println("End.");
    }
}

```

程序示例的执行结果如图 5-2 所示。

```

Start.
已产生对象实例。
obj1 和 obj2 是同一对象实例。
End.

```

图 5-2 执行结果

Singleton Pattern 的所有参与者

Singleton Pattern 的所有参与者可整理如下。

◆ Singleton 参与者

Singleton Pattern 只出现一个 Singleton 参与者。Singleton 的参与者具有 static 方法可取得惟一对象实例。这个方法永远都会返回同一对象实例。

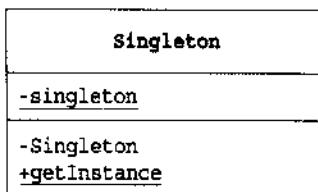


图 5-3 Singleton Pattern 的类图

扩展自我视野的提示

设限的理由

Singleton Pattern 对对象实例数目有限制。为什么程序要特地对这个部分设限制？没限制的意思就等于增加前提条件。

如果有 1 个以上的对象实例时，由于对象实例彼此之间的影响，可能会发展成出乎意料的 bug。

而如果能保证对象实例只有 1 个的话，程序员就可以在这个大前提的保护下放手去编写程序。

惟一对象实例产生的时间点

请仔细看一下程序示例的执行结果（图 5-2），你会发现是在出现“Start.”之后才有“已产生对象实例。”的消息。

当程序开始执行后，第一次调用 getInstance 方法时会初始化 Singleton 类。同时也将初始化 static 字段，产生惟一的对象实例。

相关 Pattern

下列几个 Pattern 通常都只有 1 个对象实例。

- Abstract Factory Pattern (第 8 章)
- Builder Pattern (第 7 章)
- Facade Pattern (第 15 章)
- Prototype Pattern (第 6 章)

重点回顾

本章所介绍的是可保证只产生 1 个对象实例的 Singleton Pattern。另外要写 static 方法以取得对象实例，而且构造函数也要设成 private 类型以免被其他类实例化。

除了利用类产生对象实例的做法之外，下一章将继续介绍利用现有对象实例产生其他对象实例的 Pattern。

练习题

解答请见附录 A

问题 1

下面的 TicketMaker 类（程序 5-3）每次调用 `getNextTicketNumber` 方法时，就会依次返回 1000, 1001, 1002, ...。通常用来产生门票号码、产品序号等。如果按照现在的方式，TicketMaker 类可以产生无限个对象实例。请把 Singleton Pattern 应用到此类，让它只会产生 1 个对象实例。

程序 5-3 成为 Singleton Pattern 之前的 TicketMaker 类（TicketMaker.java）

```
public class TicketMaker {
    private int ticket = 1000;
    public int getNextTicketNumber() {
        return ticket++;
    }
}
```

问题 2

请自制一个对象实例数目只能有 3 个的类 Triple。假设所有对象实例都要加上编号（0, 1, 2, ），且利用 `getInstance(int id)` 可取得编号 id 的对象实例（P127 有此类的示例）。

问题 3

假设现在有一个如程序 5-4 的 Singleton 类，但严格说来它并不能算是 Singleton Pattern。为什么？

程序 5-4 为什么严格说来不能算是 Singleton Pattern（Singleton.java）

```
public class Singleton {
    private static Singleton singleton = null;
    private Singleton() {
        System.out.println("已产生对象实例。");
    }
    public static Singleton getInstance() {
        if (singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

提示：这个示例跟多线程结构（multithread）有关。

第 6 章

Prototype (原型) ——复制建立对象 实例

Prototype Pattern

当我们在建立 Something 类的对象实例时，会使用下面这个表达式：

```
new Something()
```

利用 new (Java 语言的关键字) 指定类名就可以产生对象实例。以 new 建立对象实例时，都必须像这样指定类名称。但有时候不靠指定类名的方式就能产生对象实例，此时不是利用类建立对象实例，而是复制 (copy) 对象实例另建一个新对象实例。通常这个需求发生在以下几种情形：

(1) 种类过多无法整合成类时

第一种情形：若须处理的对象种类太多，如果要一个个设成不同的类，须产生大量的源程序文件。

(2) 不容易利用类产生对象实例时

第二种情形：该对象实例的产生过程太复杂，很难利用类来建立。假设现在有一个对象实例是用户在图形编辑软件下利用鼠标操作而产生。要想以程序来产生这样一个原本是用户操作所建立的对象实例，其实并不容易。如果是在已经有一个以用户操作的方式产生的对象实例，还想再建立一个相同对象实例的情形时，应该先把已产生的对象实例暂时存储起来，等到需要产生时再复制即可。

(3) 希望把框架和所产生的对象实例分开时

第三种情形：希望让产生对象实例时的框架不会只存在于某些特殊类时。遇到这种情形，就不能沿用指定类名产生对象实例的做法，应该预先把用来当作“雏形”的对象实例注册进去，再以复制已注册对象实例的方式来产生对象实例。

从一个现有的对象实例建立另一个新对象实例有点像是到面包店指着玻璃柜里的某种甜点，跟老板说“我要这个”的感觉。虽然不知道甜点叫什么名字，也不知道这种甜点的做法，但是只要一指眼前的甜点，还是可以买到跟“这个”相同的东西。

本章即将介绍的 Pattern 不是利用类产生对象实例，而是从一个对象实例产生出另一个新对象实例，也就是 Prototype Pattern。

Prototype 这个英文单词有“原型”的意思，命名由来是因为这个 Pattern 是根据被视为原型的对象实例，才能建立另一个新对象实例。

Java 语言把复制的操作称为“clone”。读者在本章也会学到 Java 语言的 clone 方法和 Cloneable 接口的使用方法。

Prototype (原型) —— 复制建立对象实例

程序示例

请看下面应用 Prototype Pattern 的程序示例。程序示例中有外框或下划线的部分就是字符串。

表 6-1 是此 Pattern 中出现的类和接口的一览表。Product 接口和 Manager 类属于 framework 包，处理复制对象实例的部分。Manager 类会调用 createClone，不过基本上它并不知道要复制哪个类的对象实例。只要该类实现 Product 接口，就能复制它的对象实例。

MessageBox 类和 UnderlinePen 类两者都是实现 Product 接口的类。如果先把对象实例做起来注册到 Manager 类上，随时都能复制另一个对象实例。

表 6-1 类一览表

程序包	名称	说明
framework	Product	已声明抽象方法 use 和 createClone 的接口
framework	Manager	利用 createClone 复制对象实例的类
未命名	MessageBox	把字符串框起来 use 的类。实现 use 和 createClone
未命名	UnderlinePen	把字符串加上下划线的类。实现 use 和 createClone
未命名	Main	测试用的类

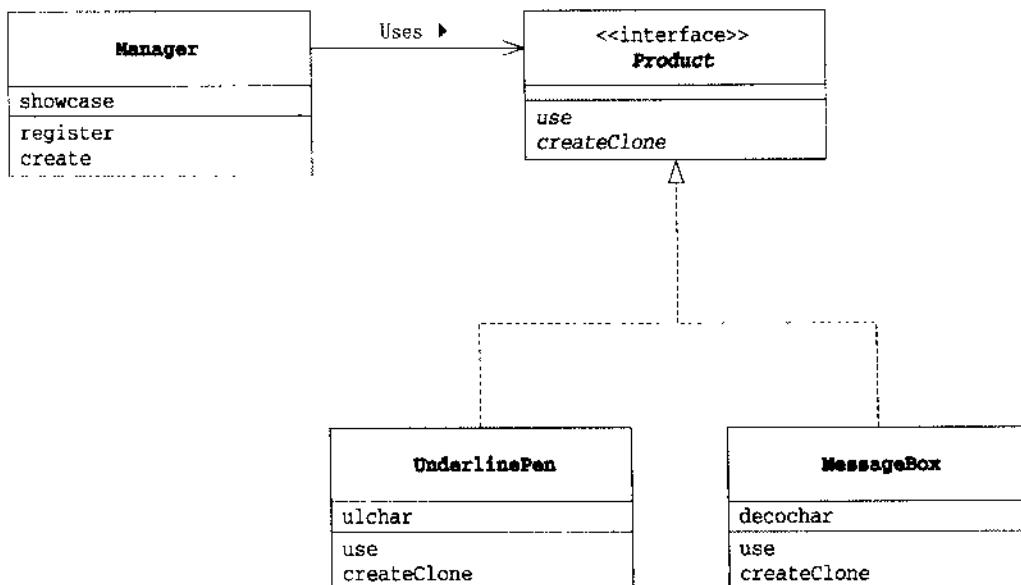


图 6-1 程序示例的类图

Product 接口

有了 Product 接口（程序 6-1）才可能进行复制。这个接口继承 java.lang.Cloneable 接口。Cloneable 接口的部分在第 73 页的深入说明再做更详细的解说，利用 clone 方法可自动对实现此接口的类进行复制。

use 方法是“使用”的部分，“使用”代表何种意义则需看子类的实现。

createClone 方法则负责复制对象实例的部分。

程序 6-1 Product 接口（Product.java）

```
package framework;

public interface Product extends Cloneable {
    public abstract void use(String s);
    public abstract Product createClone();
}
```

Manager 类

Manager 类（程序 6-2）是利用 Product 接口进行对象实例复制的类。

showcase 字段以 java.util.Hashtable 表示对象实例“名称”跟“对象实例”两者间的对应关系。

当系统以 register 方法把产品名称和 Product 接口传递过来之后，其中 1 组数据会注册到 showcase。在此传递给参数的 Product 类型 proto 是什么？虽然不了解它的实际类，不过它一定是实现了 Product 接口的类的对象实例（也就是可以调用 use 方法和 createClone 方法的对象实例）。

请注意：在 Product 接口和 Manager 类的源代码中完全没有出现 MessageBox 类或 UnderlinePen 类的类名。没有类名代表着 Product 跟 Manager 可以跟这些类分开单独修改，这点非常重要。如果类名被写在源代码内，它就会跟该类发生密切的关系。Manager 类没有一一写出所有的类名，只用到 Product 这个接口名称。所以这个接口是 Manager 类跟其他类沟通的唯一桥梁。

程序 6-2 Manager 类（Manager.java）

```
package framework;
import java.util.*;

public class Manager {
    private Hashtable showcase = new Hashtable();
```

Prototype (原型) —— 复制建立对象实例

```

public void register(String name, Product proto) {
    showcase.put(name, proto);
}

public Product create(String protoname) {
    Product p = (Product)showcase.get(protoname);
    return p.createClone();
}
}

```

MessageBox 类

接着，我们来看比较具体的子类。MessageBox 类（程序 6-3）实现（implements）Product 接口。

decochar 字段是把字符串设成边框的字符。use 方法利用 decochar 把字符串框在里面。举例来说，假设 decochar 是'*'，则当字符串“Hello”被传递给 use 方法时便会输出：

```

*****
* Hello *
*****

```

createClone 方法是复制本身的方法。这里所调用的 clone 方法规定在 Java 语言规格内，它会产生另一个分身（本身的复制品）。在产生复制品时，若该字段包含对象实例，则其值也会跟着复制到新的对象实例。能利用 clone 方法进行复制的只限于实现 java.lang.Cloneable 接口的类。如果该类没有实现此接口，则会抛出异常 CloneNotSupportedException，所以必须要先用 try ... catch 捕捉。MessageBox 类只实现 Product 接口而已，不过因为 Product 接口本身就是从 java.lang.Cloneable 接口扩充出来（如前所述）的，所以不会有 CloneNotSupportedException。另外，java.lang.Cloneable 接口只有标识的作用，没有声明此接口的方法。

Java 语言的 clone 方法只能从本身的类（含子类）调用，所以如果是因其他类的要求而进行复制时，则须另以其他方法（如 createClone）把 clone 抓进来。

程序 6-3 MessageBox 类（MessageBox.java）

```

import framework.*;

public class MessageBox implements Product {
    private char decochar;
    public MessageBox(char decochar) {
        this.decochar = decochar;
    }
}

```

Java 程序设计基础

```

        public void use(String s) {
            int length = s.getBytes().length;
            for (int i = 0; i < length + 4; i++) {
                System.out.print(decochar);
            }
            System.out.println("");
            System.out.println(decochar + " " + s + " " + decochar);
            for (int i = 0; i < length + 4; i++) {
                System.out.print(decochar);
            }
            System.out.println("");
        }
        public Product createClone() {
            Product p = null;
            try {
                p = (Product)clone();
            } catch (CloneNotSupportedException e) {
                e.printStackTrace();
            }
            return p;
        }
    }
}

```

UnderlinePen 类

UnderlinePen 类（程序 6-4）的操作大致与 MessageBox 相同，但 uchar 字段是下划线的作用。use 方法是在字符串前后加上双引号，字符串部分还有下划线。如果 uchar 是'~'，则当字符串“Hello”被传递给 use 方法时便会输出：

"Hello"

~~~~~

**程序 6-4 UnderlinePen 类 (UnderlinePen.java)**

```

import framework.*;

public class UnderlinePen implements Product {
    private char uchar;
    public UnderlinePen(char uchar) {
        this uchar = uchar;
    }
}

```

## Prototype (原型) —— 复制建立对象实例

```

public void use(String s) {
    int length = s.getBytes().length;
    System.out.println("'" + s + "'");
    System.out.print(" ");
    for (int i = 0; i < length; i++) {
        System.out.print(ulchar);
    }
    System.out.println("");
}

public Product createClone() {
    Product p = null;
    try {
        p = (Product)clone();
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
    return p;
}
}

```

---

**Main 类**

Main 类（程序 6-5）先产生 Manager 的对象实例，然后再把 UnderlinePen 的对象实例和 MessageBox 的对象实例（含名称）注册到 Manager 的对象实例中（见表 6-2）。

表 6-2 注册在 Manager 的内容

| 名称               | 类和对象实例的内容                     |
|------------------|-------------------------------|
| "strong message" | 使用 UnderlinePen，则 uchar 是'~'  |
| "warning box"    | 使用 MessageBox，则 decochar 是'*' |
| "slash box"      | 使用 MessageBox，则 decochar 是'/' |

程序 6-5 Main 类 (Main.java)

```

import framework.*;

public class Main {
    public static void main(String[] args) {
        // 预备阶段
        Manager manager = new Manager();
        UnderlinePen upen = new UnderlinePen('~');

```

## Java语言特例应用

```

    MessageBox mbox = new MessageBox('*');
    MessageBox sbox = new MessageBox('/');
    manager.register("strong message", upen);
    manager.register("warning box", mbox);
    manager.register("slash box", sbox);

    // 实现产生
    Product p1 = manager.create("strong message");
    p1.use("Hello, world.");
    Product p2 = manager.create("warning box");
    p2.use("Hello, world.");
    Product p3 = manager.create("slash box");
    p3.use("Hello, world.");
}

```

|                   |                |
|-------------------|----------------|
| "Hello, world."   | ← p1.use 的输出结果 |
| ~~~~~             |                |
| *****             | ← p2.use 的输出结果 |
| * Hello, world. * |                |
| *****             |                |
| //////////        | ← p3.use 的输出结果 |
| / Hello, world. / |                |
| //////////        |                |

图 6-2 执行结果

## Prototype Pattern 的所有参与者

Prototype Pattern 中所有出现过的参与者可整理如下。

### ◆ Prototype (原型) 参与者

Prototype 参与者规定复制 (copy) 对象实例再建立新对象实例的方法。在前面的程序示例中，担任这个参与者的是 Product 接口。

### ◆ ConcretePrototype (具体原型) 参与者

ConcretePrototype 参与者是实际上实现先复制对象实例再建立新对象实例的方法。在前面的程序示例中，担任这个参与者的是 MessageBox 类和 UnderlinePen 类。

## Prototype (原型) —— 复制建立对象实例

### ◆ Client (客户) 参与者

Client 参与者利用复制对象实例的方法以产生另一个新对象实例。在前面的程序示例中，担任这个参与者的是 Manager 类。

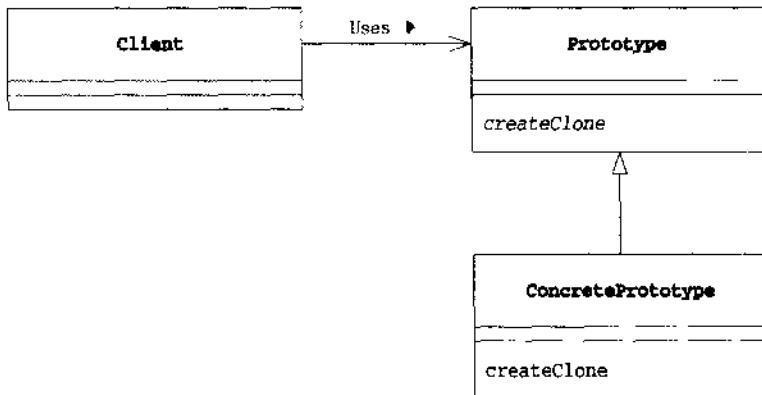


图 6-3 Prototype Pattern 的类图

## 扩展自我视野的提示

### 难道真的不能利用类来建立对象实例吗

还记得笔者当初在学习 Prototype Pattern 时，刚开始就在想：如果要建立对象实例的话，为什么不干脆写成这样呢？

```
new Something()
```

为什么还要蹦出一个 Prototype Pattern？答案就是本章最前面所提到的 3 种情形。这里再跟各位一起回顾程序示例，简单说明一下。

#### (1) 种类过多无法整合成类时

程序示例出现过下面 3 种雏形：

- 以`~`把字符串加上下划线
- 以`\*\*`把字符串加上边框
- 以`\`把字符串加上边框

这些例子都属于简单型，虽然雏形只列出 3 种，只要有心的话还可以多做几种。但是如果全部都要做成不同的类，类数量会很可观，增加程序源代码管理的困难度。

#### (2) 不容易利用类产生对象实例时

这一点在本章程序示例中可能不太明显，不过如果把这个操作跟几乎全以鼠标辅

助操作的绘图软件联想起来，应该会比较容易理解。假设有一个表示原来以人工操作方式绘制图形的对象实例，而现在要建立一个完全相同的新对象实例。在这种情形下，利用对象实例复制的方式当然会比从头用类产生对象实例要简单容易得多。

### (3) 希望把框架和所产生的对象实例分开时

程序示例把执行对象实例复制(`clone`)的部分放在 `framework` 程序包内。

产生对象实例时当然要传递类名给 `Manager` 类的 `create` 方法，不过这里以“`strong message`”和“`slash box`”取代。这可以说是加强广泛应用 Java 语言原本就有的对象实例产生架构 `new Something()` 的格式，而让框架脱离类名称的束缚。

## 类名是一种束缚

话说回来，如果硬把类名塞到程序源代码内会发生什么问题？把程序中利用到的类名写在里面不是理所当然的吗？

在这里，要请各位回忆一下面向对象程序设计的几个使用目的，“零件化复用”正是其中之一。

事先把源代码用到的类名写下来这个操作，也不见得就是一定做错。但是当你写下源代码内所用到的类名的那一刻起，它就无法跟该类分离供你复用了。

当然只要动手修改源代码就可以改掉类名称，不过从“零件化复用”的角度来说，根本不考虑修改源代码。谈到 Java，“即使手中只有类文件(`.class`)，可否复用该类？”这个概念相当重要。说到重点了，对，就是“即使没有源文件(`.java`)，可否予以复用？”。

必须紧密结合在一起的类名写到源代码内是理所当然的，根本不成问题；真正的问题是应该分离独立成零件的类名却被写到源代码里面。

## 相关 Pattern

### ◆ Flyweight Pattern (第 20 章)

`Prototype Pattern` 是建立一个跟目前对象实例同样状态的新对象实例，再加以应用。

`Flyweight Pattern` 则是在不同位置共享同一个对象实例。

### ◆ Memento Pattern (第 18 章)

`Prototype Pattern` 是建立一个跟目前对象实例同样状态的新对象实例。

`Memento Pattern` 则是存储目前对象实例的状态，才能进行 `snapshot(快照)` 和 `undo(恢复)`。

## Prototype (原型) —— 复制建立对象实例

### ◆ Composite Pattern (第 11 章) 和 Decorator Pattern (第 12 章)

大量使用 Composite Pattern 和 Decorator Pattern 时, 有时需要能机动建立一些构造较复杂的对象实例。此时, 利用 Prototype Pattern 就很方便。

### ◆ Command Pattern (第 22 章)

如果想复制 Command Pattern 中的命令时, 可能会需要使用 Prototype Pattern。

## 深入说明: clone 方法和 java.lang.Cloneable 接口

### Java 语言的 clone

Java 语言中的 clone 方法是一个进行对象实例复制的结构。请注意, 在执行 clone 方法时, 欲复制的类必须实现 `java.lang.Cloneable` 接口。欲复制的类可直接实现 `java.lang.Cloneable` 接口, 也可在父类实现 `Cloneable` 接口。另外, 要实现 `Cloneable` 接口的子接口也没问题。本章的程序示例是以 `MessageBox` 类和 `UnderlinePen` 类实现 `Product` 接口, 这个 `Product` 接口就是 `Cloneable` 接口的子接口。

当实现过 `Cloneable` 接口的类的对象实例调用 `clone` 方法时, 就会开始复制。而 `clone` 方法的返回值是新复制成的对象实例(内部操作——先保留与原始对象实例同样大小的内存空间, 再复制原始对象实例的字段内容)。

如果是并未实现过 `Cloneable` 接口的类的对象实例在调用 `clone` 方法, 便会发生异常 `CloneNotSupportedException` (不支持 `clone` 的异常)。

上述重点可整理如下。

- 实现 `Cloneable` 接口的类的对象实例
  - 进行复制
- 尚未实现 `Cloneable` 接口的类的对象实例
  - 发生 `CloneNotSupportedException` 异常

另外, `java.lang` 包不经声明就已经被 `import` 进来, 所以源代码中可以写成简单的 `Cloneable`, 不需要写成 `java.lang.Cloneable`。

### clone 方法是在哪里定义的

`clone` 方法是由 `java.lang.Object` 类定义的。`Object` 类是 Java 类层次的基类, 因此所有类都会继承 `clone` 方法。

### Cloneable 所要求的方法是什么

“`Cloneable` 接口”这个名词乍听之下, 很容易被误导以为里面有声明了 `clone` 方

法。Cloneable 接口并没有声明任何方法。这个接口只是用来表示“可使用 clone 进行复制”而已。

## clone 方法做的是浅拷贝（shallow copy）

利用 clone 方法所做的操作是直接复制字段内容，换句话说，这个操作并不管该字段对应的对象实例内容。假设现在有个字段对应到一个数组，当你使用 clone 方法进行复制时，复制的结果只对应到该数组的参照而已，并不会一个个复制该数组的元素。

像这样字段对字段的复制（field-for-field copy）就称为“浅拷贝（shallow copy）”。clone 方法所做的正是“浅拷贝”。

如果只做“浅拷贝”的 clone 无法满足你的需求，设计类的程序员也可以重写 clone 方法，自行定义想要的“复制结果”（若有重载 clone 方法时，要记得改用 super.clone() 调用父类的 clone 方法）。

另外请注意一点，clone 只会进行复制而已，没有调用构造函数。如果该类在产生对象实例时需要做特殊的初始化，则应将处理操作语句放在 clone 方法之内。

JDK 的 API 参考在 java.lang.Object 类的 clone 方法以及 java.lang.Cloneable 接口这两项有更详细的解说内容，如有需要请自行参考。

## 重点回顾

本章所介绍的是利用对象实例建立新对象实例的 Prototype Pattern，不需要从类去建立对象实例。此外，对于 clone 方法和 Cloneable 接口的使用方法也略有着墨。

## 练习题

解答请见附录 A

### 问题 1

在前面的程序示例中，MessageBox 类（程序 6-3）和 UnderlinePen 类（程序 6-4）都定义同样操作的 createClone 方法。同一程序内若定义多个相同操作的方法会造成管理上的困扰，最好能让它们共享一个方法。应该如何改写才能达到目的？

### 问题 2

java.lang.Object 类有 clone 方法。那么，java.lang.Object 实现了 java.lang.Cloneable 接口吗？

# 第7章

## Builder(生成器) ——组合复杂的 对象实例

## Builder Pattern

大都市里到处可见高楼耸立，高楼大厦（building）当然是有结实构造的大型建筑物。在英文中，通常建筑或建造有相当构造的大型建筑物就称为“build”。

所谓万丈高楼平地起，大楼首先得打稳地基、搭建骨架，再由下往上一层层盖上去。一般说来，如果搭建的结构越复杂就越难一气呵成。你得先把整个架构分成几个部分，等到各个部分都完成了之后再依次组合起来。

本章所要介绍的正是采取循序渐进的方式组合较复杂对象实例的 Builder Pattern。

## 程序示例

我们来试做一个能利用 Builder Pattern 建立“文件”的程序吧，这个文件的构造如下：

- 含有 1 个标题
- 含有一些字符串
- 含有一些有项目符号的项目

Builder 类规定组成文件的方法，而 Director 类利用这个方法才能产生 1 份具体的文件。

Builder 类是抽象类，不含实际的处理内容，仅声明抽象方法而已。决定产生文件的具体处理内容的是 Builder 类的子类。

程序示例定义以下几个类作为 Builder 类的子类。

- TextBuilder 类 —— 产生一般纯文本（plaintext）格式的文件
- HTMLBuilder 类 —— 产生 HTML 格式的文件

Director 若使用 TextBuilder 可产生一般格式的文件，如果使用 HTMLBuilder 则产生 HTML 格式的文件。

章末的练习题（问题 3）也建立了 Builder 类的子类。

表 7-1 类一览表

| 名称          | 说明                        |
|-------------|---------------------------|
| Builder     | 规定建立文件时的方法的类              |
| Director    | 产生 1 个文件的类                |
| TextBuilder | 产生 plaintext 格式（一般文本格式）的类 |
| HTMLBuilder | 产生 HTML 格式的类              |
| Main        | 测试用的类                     |

## Builder (生成器) —— 组合复杂的对象实例

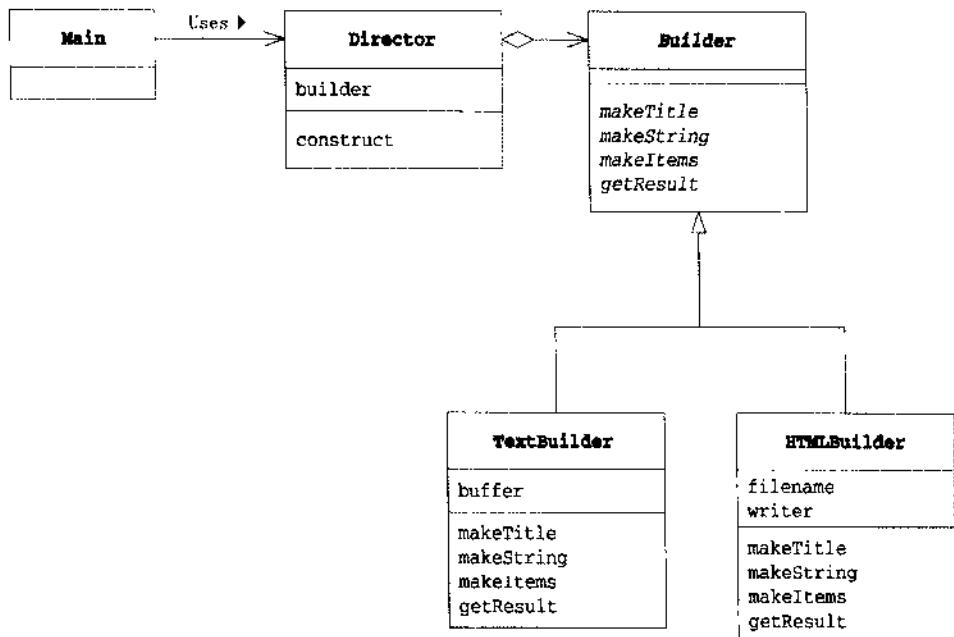


图 7-1 程序示例的类图

## Builder 类

Builder 类（程序 7-1）是声明产生“文件”的所有方法的抽象类。MakeTitle、makeString、makeItems 分别是产生文件中的标题、字符串和项目的方法。getResult 方法是在最后取得完成文件的方法。

### 程序 7-1 Builder 类 (Builder.java)

---

```

public abstract class Builder {
    public abstract void makeTitle(String title);
    public abstract void makeString(String str);
    public abstract void makeItems(String[] items);
    public abstract Object getResult();
}
  
```

---

## Director 类

Director 类（程序 7-2）利用在 Builder 类所声明的方法来产生文件。

Director 类的构造函数的参数是 Builder 类型，但实际上并没有任何 Builder 类的对象实例被传递出去作为参数。因为 Builder 类是抽象类，根本无法产生对象实例。实

## Java语言中的应用

际上传递给 Director 的构造函数的是 Builder 类的子类（即后述 TextBuilder 类、HTMLBuilder 类等）的对象实例。根据传递过来的 Builder 类的子类型，才能决定 Director 类所产生的具体文件的格式。

construct 方法是产生文件的方法。调用这个方法，就会产生文件并返回结果。construct 方法只能使用于 Builder 声明过的方法（construct 的英文语意为“构造”）。

## 程序 7-1 Director 类 (Director.java)

```
public class Director {
    private Builder builder;
    public Director(Builder builder) { // 先把 Builder 的子类的对象
        // 实例，存储在 builder 字段
        this.builder = builder;
    }
    public Object construct() { // 建立文件
        builder.makeTitle("Greeting"); // 标题
        builder.makeText("从早上到白天结束"); // 字符串
        builder.makeItems(new String[]{ // 项目
            "早安。",
            "午安。",
        });
        builder.makeText("到了晚上"); // 另一个字符串
        builder.makeItems(new String[]{ // 另一个项目
            "晚安。",
            "祝你有个好梦。",
            "再见。",
        });
        return builder.getResult(); // 完成的文件就是返回值
    }
}
```

## TextBuilder 类

TextBuilder 类（程序 7-3）是 Builder 类的子类。以一般文本格式建立文件。以 String 返回结果。

## 程序 7-3 TextBuilder 类 (TextBuilder.java)

```
public class TextBuilder extends Builder {
    private StringBuffer buffer = new StringBuffer();
    // 开始在此字段建立文件
```

## Builder（生成器）——组合复杂的对象实例

```

public void makeTitle(String title) { // 一般文本格式的标题
    buffer.append("=====\n"); //花边
    buffer.append("!" + title + "!\n"); // 有!的标题
    buffer.append("\n"); // 空行
}
public void makeString(String str) { // 一般文本格式的字符串
    buffer.append('■' + str + "\n"); // 有■的字符串
    buffer.append("\n"); // 空行
}
public void makeItems(String[] items) { // 一般文本格式的项目
    for (int i = 0; i < items.length; i++) {
        buffer.append("-" + items[i] + "\n"); // 有-的项目
    }
    buffer.append("\n"); // 空行
}
public Object getResult() { // 完成的文件
    buffer.append("=====\n"); //花边
    return buffer.toString(); //把 StringBuffer 转换成 String
}
}

```

---

## HTMLBuilder 类

HTMLBuilder 类（程序 7-4）也是 Builder 类的子类。HTMLBuilder 类是把文件建成 HTML 格式，最后返回 HTML 文件的文件名作为建立的结果。

**程序 7-4 HTMLBuilder 类（HTMLBuilder.java）**

```

import java.io.*;

public class HTMLBuilder extends Builder {
    private String filename; // 产生的文件名
    private PrintWriter writer; // 写入到文件的 PrintWriter
    public void makeTitle(String title) { // HTML 文件的标题
        filename = title + ".html"; // 根据标题决定文件名
        try {
            writer = new PrintWriter(new FileWriter(filename));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

```

    }
    writer.println("<html><head><title>" + title + "</title>
</head><body>");
    // 输出标题
    writer.println("<h1>" + title + "</h1>");
}
public void makeString(String str) { // HTML文件的字符串
    writer.println("<p>" + str + "</p>"); // 以<p>标签输出
}
public void makeItems(String[] items) { // HTML文件的项目
    writer.println("<ul>"); // 以<ul>和<li>输出
    for (int i = 0; i < items.length; i++) {
        writer.println("<li>" + items[i] + "</li>");
    }
    writer.println("</ul>");
}
public Object getResult() { // 完成的文件
    writer.println("</body></html>"); // 关闭标签
    writer.close(); // 关闭文件
    return filename; // 返回文件名
}
}

```

## Main 类

Main类(程序 7-5)是Builder Pattern的测试程序,它能产生如下符合命令行所指定格式的文件。

```

java Main plain ——以一般格式产生文件
java Main html ——以HTML格式产生文件

```

如果命令行指定的是plain时,则将TextBuilder类的对象实例传递给Director类的构造函数;若命令行指定为html时,则将HTMLBuilder类的对象实例传递给Director类的构造函数。

TextBuilder 和 HTMLBuilder 都是 Builder 的子类, Director 只会利用 Builder 的方法来产生文件。Director之所以只利用 Builder 的方法,主要是因为它并不会去管实际调用的是TextBuilder还是HTMLBuilder。

因此如果想要达到产生文件的目的,Builder 必须声明一些必要而且面面俱到的方法群组。不过,Builder 不可以连固有的方法都提供给一般文本格式或 HTML 格式。

## Builder (生成器) —— 组合复杂的对象实例

程序 7-5 Main 类 (Main.java)

```

public class Main {
    public static void main(String[] args) {
        if (args.length != 1) {
            usage();
            System.exit(0);
        }
        if (args[0].equals("plain")) {
            Director director = new Director(new TextBuilder());
            String result = (String)director.construct();
            System.out.println(result);
        } else if (args[0].equals("html")) {
            Director director = new Director(new HTMLBuilder());
            String filename = (String)director.construct();
            System.out.println("已产生" + filename + "。");
        } else {
            usage();
            System.exit(0);
        }
    }
    public static void usage() {
        System.out.println("Usage: java Main plain 产生一般格式的文件");
        System.out.println("Usage: java Main html 产生 HTML 格式的文
件");
    }
}

```

```

java Main plain
=====
「Greeting」
■从早上到白天结束
·早安。
·午安。
■到了晚上
·晚安。
·祝你有个好梦。
·再见。
=====
```

图 7-2 执行结果 (一般文本格式)

```
java Main html  
已产生 Greeting.html。  
  
type Greeting.html  
<html><head><title>Greeting</title></head><body>  
<h1>Greeting</h1>  
<p>从早上到白天结束</p>  
<ul>  
<li>早安。</li>  
<li>午安。</li>  
</ul>  
<p>到了晚上</p>  
<ul>  
<li>晚安。</li>  
<li>祝你有个好梦。</li>  
<li>再见。</li>  
</ul>  
</body></html>
```

图 7-3 执行结果 (HTML 格式)



图 7-4 HTMLBuilder 所产生的 Greeting.html 文件在浏览器下的效果

## Builder Pattern 的所有参与者

Builder Pattern 中出现过的所有参与者可整理如下。

### ◆ Builder(工人) 参与者

Builder 参与者是规定产生对象实例的接口 (API)。包括产生对象实例各个部分的方法和取得最后结果的方法。在前面的程序示例中，扮演这个参与者的是 Builder 类。

### ◆ ConcreteBuilder(具体工人) 参与者

ConcreteBuilder 参与者是实现 Builder 参与者的接口 (API) 的类。在实际产生对象实例时所调用的方法就是在这里定义的。在前面的程序示例中，扮演这个参与者的是 TextBuilder 类和 HTMLBuilder 类。

### ◆ Director(监工) 参与者

Director 参与者利用 Builder 参与者的接口 (API) 产生对象实例。设计程序时必须注意不要被 ConcreteBuilder 参与者牵着鼻子走。为了让 ConcreteBuilder 参与者无论在什么情形之下都能正常发挥功能，所以只使用 Builder 参与者的方法。在前面的程序示例中，扮演这个参与者的是 Director 类。

### ◆ Client(客户) 参与者

利用 Builder Pattern 的参与者。在前面的程序示例中，扮演这个参与者的是 Main 类。

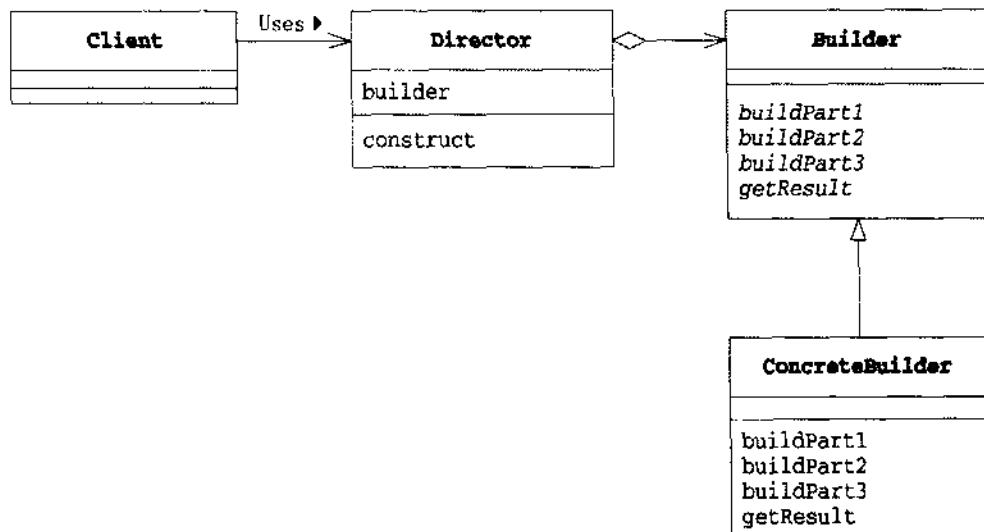


图 7-5 Builder Pattern 的类图

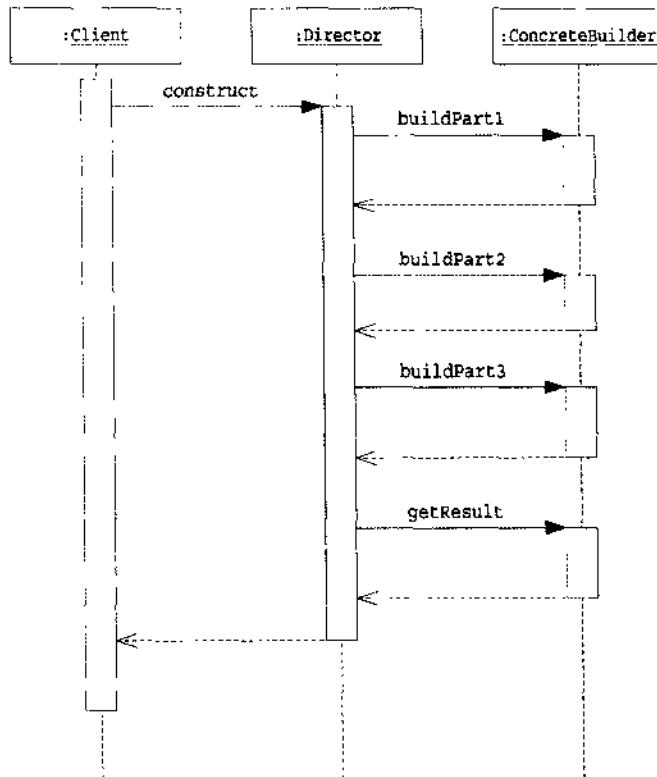


图 7-6 Builder Pattern 的流程图

## 相关 Pattern

### ◆ Template Method Pattern (第 3 章)

在 Builder Pattern 时，Director 参与者控制 Builder 参与者。

而在 Template Method Pattern，则是父类控制子类。

### ◆ Composite Pattern (第 11 章)

利用 Builder Pattern 所产生的结果可能会变成 Composite Pattern。

### ◆ Abstract Factory Pattern (第 8 章)

Builder Pattern 和 Abstract Factory Pattern 都会产生较复杂的对象实例。

### ◆ Facade Pattern (第 15 章)

Builder Pattern 的 Director 参与者是组合 Builder 参与者所提供的复杂方法，再把构成对象实例的简单接口（API）提供给外部（程序示例中的 construct 方法就相当于

## Builder（生成器）——组合复杂的对象实例

这个部分)。

Facade Pattern 的 Facade 参与者则是组合内部模块，再把执行操作的简单接口 (API) 提供给外部。

### 扩展自我视野的提示

#### 谁知道哪个部分

在面向对象程序设计中，“谁知道哪个部分”相当重要。也就是说，在设计程序时要注意哪个类能使用（或可以使用）什么方法。

请回想一下前面的程序示例。

Main 类并不知道 Builder 类的方法（没有调用）。Main 类只调用 Director 类的 construct 方法。所有工作在 Director 类内如火如荼地进行（但 Main 类不管这部分），最后从 construct 方法的返回值可取得已完成的文件。

从另一个角度来看，Director 类唯一知道的就是 Builder 类。Director 类利用 Builder 类的方法建立文件。但是，Director 类并不知道自己实际在利用的类“究竟是何方神圣”，它不会知道这是 TextBuilder、HTMLBuilder 还是 Builder 的其他子类。其实不知道也没什么大不了，因为 Director 类只有利用 Builder 类的方法，其实是由 Builder 类的子类来实现该方法。

说老实话，Director 类对于自己利用的 Builder 类的子类一无所知还真是值得高兴。因为“就是因为不知道才能拿其他东西来取代”。不管传递给 Director 类的是 TextBuilder 的对象实例还是 HTMLBuilder 的对象实例，Director 类仍然能正常发挥作用，当然就是因为 Director 类“不知道”Builder 类的具体子类。

就是因为不知道才能拿其他东西来取代，而由于它具有可被取代的特性，也等于提高了它的零件价值。程序员在设计类时要无时不刻想到这个“可替代性”。

#### 在设计时该决定哪些事、又不该决定哪些事

Builder 类必须声明建立文件时（达成目的）必要而且包罗万象的方法群。因为传递给 Director 类的工具都是由 Builder 类提供，所以 Builder 类要准备哪些方法显得十分重要。

更何况，Builder 类也需要能对应将来可能会有的 Builder 类的子类所提出的要求。本章的程序示例只产生一般文本格式和 HTML 格式两种。假设以后想建立其他格式（例如 XXXX 格式）的文件时，是否真能够建立 XXXX 格式用的 XXXXBuilder 类？难道不会需要其他的新方法吗？

设计类的程序员又不是神仙，哪能预料到未来会发生什么事。不过设计时还是得注意能顺应未来可能发生的变化，这才是专业人士应有的素养。

## 源代码的解读与修改

有时候写程序得从零开始，不过通常为了节省开发时间，都会直接拿现有的源代码来修改或增加一些内容。

首先你要解读现有的源代码。但是如果只看抽象类的部分，也得不到什么有用的信息（从方法名称倒是可以获得一些灵感）。

就拿前面的程序示例为例，光看 `Builder` 这个抽象类的部分，恐怕只会徒劳无功。至少该去看 `Director` 的源代码，了解 `Builder` 这个类的用法（方法的调用方式）吧。最好连 `TextBuilder`、`HTMLBuilder` 几个类也一起拿来看，这样就能知道 `Builder` 的抽象方法在等待哪些操作。

要是还没了解类的功能就急着去修改或新增其他内容的话，很容易搞不清楚该修改哪个类才正确。例如，如果要修改 `Builder` 类，应该是修改 `Director` 所调用的方法，或者是修改结果要影响到 `Builder` 类的所有子类。万一把程序修改成让 `Director` 类去调用 `TextBuilder` 类的方法，就会失去零件的独立性，在切换到 `HTMLBuilder` 时可能会无法正常运作。

## 重点回顾

本章所介绍的是循序渐进组合较复杂的对象实例的 `Builder Pattern`，组合过程的详细处理则隐藏在 `Director` 参与者里。

## 练习题

解答请见附录 A

### 问题 1

请将程序示例的 `Builder` 类（程序 7-1）修改成接口，同时配合修改其他的类。

### 问题 2

程序示例的 `HTMLBuilder` 类（程序 7-4）一开始就要调用 `makeTitle` 方法，但 `TextBuilder` 类（程序 7-3）却没有方法调用顺序的限制。

请修改 `Builder` 类（程序 7-1）、`TextBuilder` 类（程序 7-3）和 `HTMLBuilder` 类（程序 7-4），确定“在调用出 `makeString` 方法、`makeItems` 方法或 `getResult` 方法之前，一定会先调用一次 `makeTitle` 方法”。

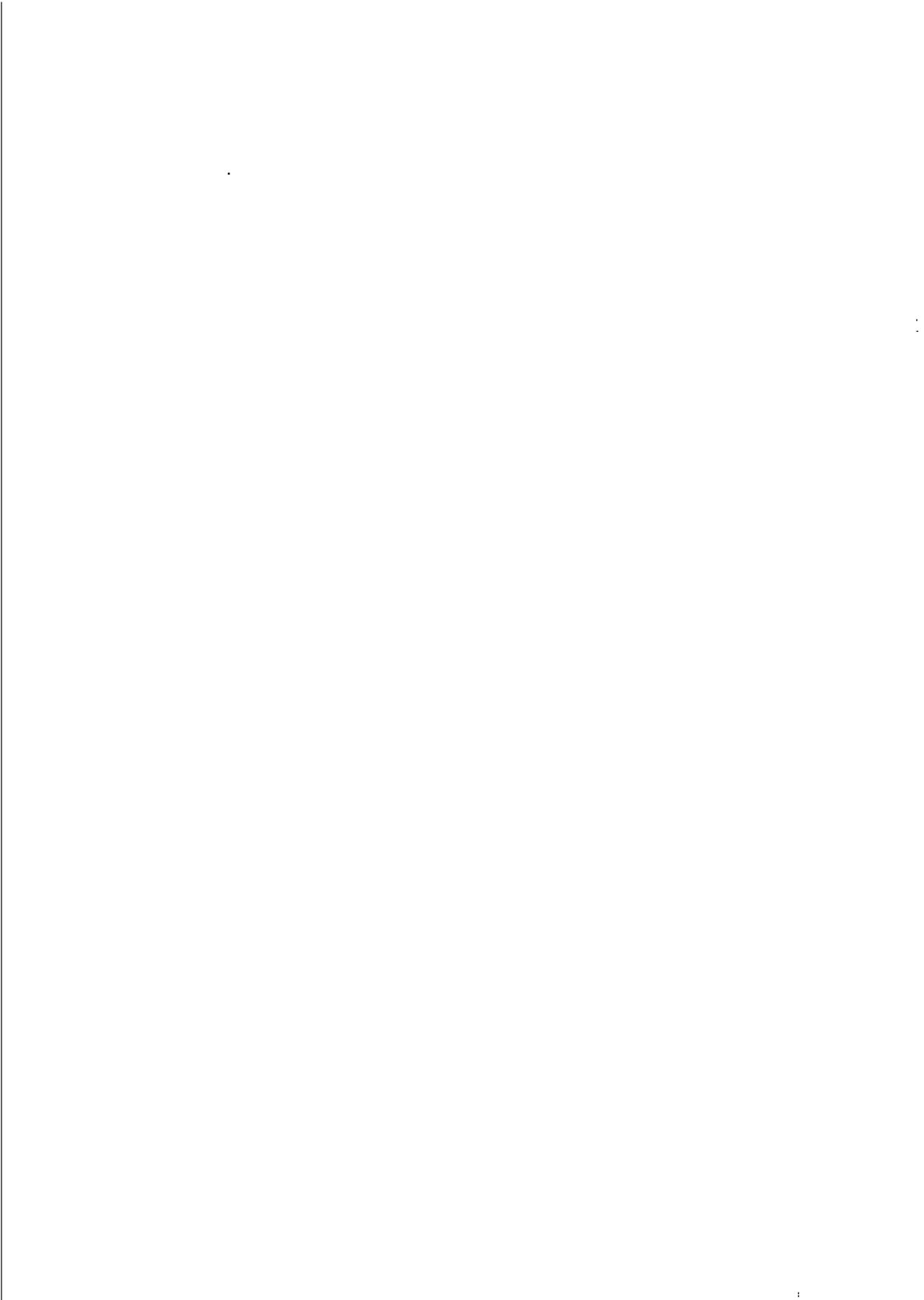
## Builder（生成器）——组合复杂的对象实例

### 问题 3

请建立一个程序示例中 `Builder` 类（程序 7-1）的子类，让它扮演 `ConcreteBuilder` 参与者的类。内容不限，但不可使用已经出现过的一般文本格式和 HTML 格式。

### 问题 4

在 `TextBuilder` 类（程序 7-3），为什么建立文件的字段 `buffer` 是 `StringBuffer` 类而非 `String` 类？如果使用 `String` 的话，会发生什么问题？



# 第 8 章

## Abstract Factory (抽象工厂) — 把相关零件组合 成产品



## Abstract Factory Pattern

本章的学习主题是

Abstract Factory Pattern

abstract 是“抽象”，factory 是“工厂”，所以合起来 abstract factory 就是“抽象工厂”的意思。

从基本常识的角度来说，我们通常不会有“抽象” + “工厂”的说法。工厂本来就是把各种零件组合成产品出货的地点，因此所作所为应该都是再具体不过的行为，现在居然出现“抽象工厂”，这不是违反常理吗？

请各位不要听到“抽象工厂”就大吃一惊，因为在 Abstract Factory Pattern 当中除了“抽象工厂”之外，还有“抽象零件”和“抽象产品”。这个抽象工厂把各种抽象零件组合起来成为抽象产品。

或许这样的说法会让部分读者觉得越听越迷糊。请各位再回忆一下面向对象对于“抽象的”的定义。“抽象的”是指“不考虑要如何具体进行实现，只注意接口（API）部分”的状态。例如，抽象方法（abstract method）就是一个不写方法的主体，而仅规定名称和 signature（参数类型与数值）的方法。

在 Template Method Pattern（第 3 章）和 Builder Pattern（第 7 章）提到需要忽略方法的具体内容（假装没看到），利用抽象方法编写程序。

Abstract Factory Pattern 中的抽象工厂则是把各种抽象零件组合成抽象产品。换句话说，处理的重点是在接口（API）而不是零件的具体实现。只利用接口（API）就能把零件组合成产品。

Template Method Pattern 和 Builder Pattern 的情形是在子类进行具体的实现，而 Abstract Factory Pattern 也同样是由子类来处理具体的实现。在子类就会出现具体工厂利用具体零件组合成具体产品。

以下就开始解读抽象工厂的具体程序示例。

## 程序示例

本章的程序示例是“将一个层次结构的相关网站链接做成 HTML 文件”。最后完工的 HTML 文件如图 8-1 所示，若以 Web 浏览器开始时，则如图 8-2 所示。

## Abstract Factory (抽象工厂) —— 把相关零件组合成产品

```
<html><head><title>LinkPage</title></head>
<body>
<h1>LinkPage</h1>
<li>
新闻
<ul>
<li><a href="http://www.asahi.com/">朝日新闻</a></li>
<li><a href="http://www.yomiuri.co.jp/">读卖新闻</a>
</li>
</ul>
</li>
<li>
搜索引擎
<ul>
<li>
Yahoo!
<ul>
<li><a href="http://www.yahoo.com/">Yahoo!</a></li>
<li><a href="http://www.yahoo.co.jp/">Yahoo!Japan</a>
</li>
</ul>
</li>
<li><a href="http://www.excite.com/">Excite</a></li>
<li><a href="http://www.google.com/">Google</a></li>
</ul>
</li>
</ul>
<hr><address>结城 浩</address></body></html>
```

图 8-1 层次结构的相关网站链接 (HTML)



图 8-2 层次结构的相关网站链接（在浏览器下的样子）

这个程序示例包括以下 3 个包的类群组：

- factory 包——含抽象工厂、零件和产品的包
- 未命名包——含 Main 类的包
- listfactory 包——含具体工厂、零件和产品的包（在此使用<ul>标签实现）

类一览表和 UML 类图分别如表 8-1、图 8-3 所示。位于类图上方的是抽象工厂，下方则是具体工厂。类图中并没有列出 Main 类。

表 8-1 类一览表

| 包           | 名称          | 说明                                        |
|-------------|-------------|-------------------------------------------|
| factory     | Factory     | 表示抽象工厂的类（产生 Link, Tray, Page）             |
| factory     | Item        | 用来统一处理 Link 和 Tray 的类                     |
| factory     | Link        | 抽象零件：表示 HTML 链接的类                         |
| factory     | Tray        | 抽象零件：抽取 Link 和 Page 的类                    |
| factory     | Page        | 抽象零件：表示 HTML 网页的类                         |
| 未命名         | Main        | 测试用的类                                     |
| listfactory | ListFactory | 表示具体工厂的类（产生 ListLink, ListTray, ListPage） |
| listfactory | ListLink    | 具体零件：表示 HTML 链接的类                         |
| listfactory | ListTray    | 具体零件：抽取 Link 和 Page 的类                    |
| listfactory | ListPage    | 具体零件：表示 HTML 网页的类                         |

## Abstract Factory (抽象工厂) —— 把相关零件组合成产品

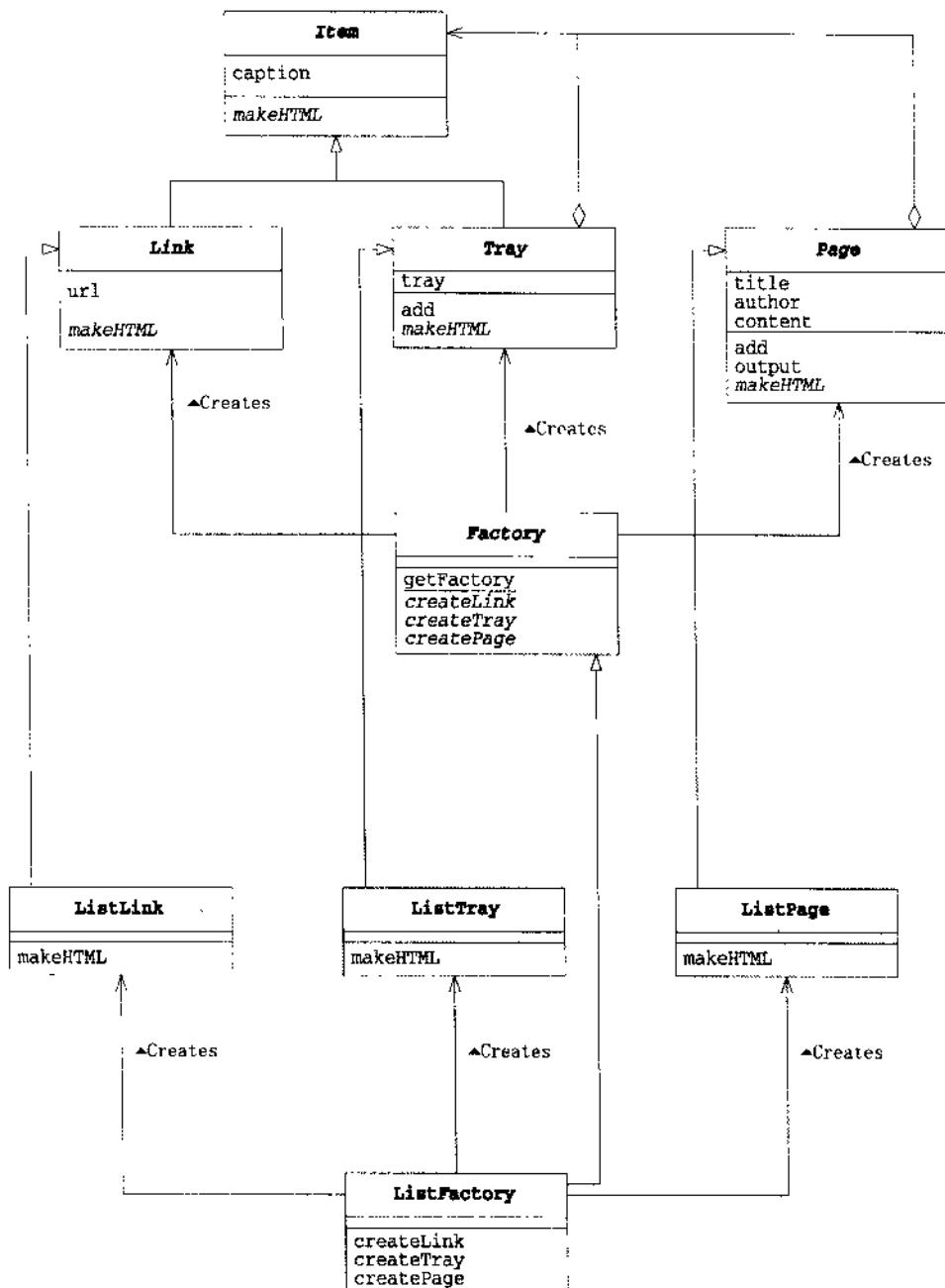


图 8-3 程序示例的类图

各类的源程序代码就在图 8-4 的目录内。

## 工厂方法模式的应用

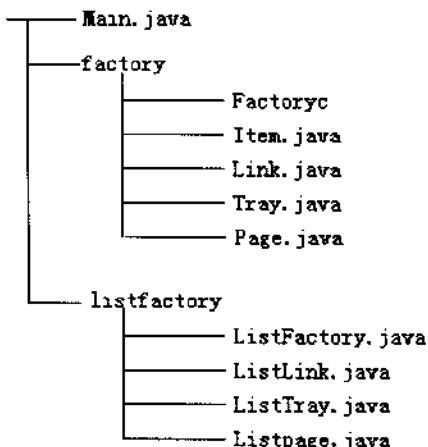


图 8-4 源程序文件的所在目录

编译操作为：

```
javac Main.java listfactory/ListFactory.java
```

前面几章的程序示例只编译 Main.java 就能对所有必要的类进行编译，但若本章的程序示例也只编译 Main.java 的话，虽然能顺利编译 Factory.java、Item.java、Link.java、Tray.java 和 Page.java，但没有编译到 ListFactory.java、ListLink.java、ListTray.java、ListPage.java。因为 Main 类只使用 factory 包，并没有直接使用 listfactory 包。因此必须把 listfactory/ListFactory.java 加入到参数内才能一起编译（这样就能让 ListFactory.java、ListLink.java、ListTray.java、ListPage.java 都加入编译操作的范围内）。

```

javac Main.java listfactory/ListFactory.java
java Main listfactory.ListFactory
已产生 LinkPage.html。
(若以 Web 浏览器读取 LinkPage.html，则如图 8-2 所示)
  
```

图 8-5 编译与执行结果

## 抽象零件：Item 类

Item 类（程序 8-1）是 Link 和 Tray 的父类（item="项目"）。它是让 Link 跟 Tray 两者一视同仁的类。

caption 字段是此项目的“标题”。

makeHTML 方法是抽象方法，须等待子类进行实现。一旦调用此方法时，其返回

**Abstract Factory (抽象工厂) —— 把相关零件组合成产品**

值即为 HTML 的字符串 (子类要有这样的执行功能)。

**程序 8-1 Item 类 (Item.java)**

```
package factory;

public abstract class Item {
    protected String caption;
    public Item(String caption) {
        this.caption = caption;
    }
    public abstract String makeHTML();
}
```

**抽象零件: Link 类**

Link 类 (程序 8-2) 是以抽象方式表示 HTML 超链接的类。

url 字段是用来存储链接网站的 URL。乍看之下, Link 类好像没有任何抽象方法, 但可别被骗了。因为 Link 类没有实现父类 (Item) 的抽象方法 (makeHTML), 所以 Link 类也是抽象类。

**程序 8-2 Link 类 (Link.java)**

```
package factory;

public abstract class Link extends Item {
    protected String url;
    public Link(String caption, String url) {
        super(caption);
        this.url = url;
    }
}
```

**抽象零件: Tray 类**

Tray 类 (程序 8-3) 是表示收集一个以上的 Link 或 Tray 的类 (tray="浅盘", 请把它想成是将一个个标有项目符号的项目放在盘子上)。

Link 和 Tray 则利用 add 方法把它们找出来。add 方法对于“Link 和 Tray”这部分的表现方式则是将 Link 和 Tray 的父类 Item 设为参数。

Tray 类继承了 Item 类的抽象方法 makeHTML, 但并未实现。所以 Tray 类是抽象类。

**程序 8-3 Tray 类 (Tray.java)**

```

package factory;
import java.util.Vector;

public abstract class Tray extends Item {
    protected Vector tray = new Vector();
    public Tray(String caption) {
        super(caption);
    }
    public void add(Item item) {
        tray.add(item);
    }
}

```

**抽象产品：Page 类**

Page 类（程序 8-4）则是以抽象的方式表现整个 HTML 网页的类。如果说 Link 和 Tray 是抽象的“零件”，那么 Page 类就可以说是抽象的“产品”。Title 字段是网页的标题，author 字段则是网页作者。作者姓名则以传递给构造函数的参数指定。

利用 add 方法可以在网页上新增其他 Item（即 Link 或 Tray）。新增的部分会显示在该网页上。

output 方法则是根据标题决定文件名，利用 makeHTML 方法把本身 HTML 的内容写进文件内。

```
writer.write(this.makeHTML());
```

这个语句就算把 this 拿掉，改成

```
writer.write(makeHTML());
```

也不会影响原来的意思，只不过为了强调“这是本身的 makeHTML 方法”，所以才特地加上 this。这里所调用的 makeHTML 方法是抽象方法。output 方法就成为一个简单的 Template Method Pattern。

**程序 8-4 Page 类 (Page.java)**

```

package factory;
import java.io.*;
import java.util.Vector;

public abstract class Page {
    protected String title;

```

## Abstract Factory (抽象工厂) —— 把相关零件组合成产品

```

protected String author;
protected Vector content = new Vector();
public Page(String title, String author) {
    this.title = title;
    this.author = author;
}
public void add(Item item) {
    content.add(item);
}
public void output() {
    try {
        String filename = title + ".html";
        Writer writer = new FileWriter(filename);
        writer.write(this.makeHTML());
        writer.close();
        System.out.println("已产生" + filename + "。");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
public abstract String makeHTML();
}

```

---

## 抽象工厂：Factory 类

前面所看到的源代码都是抽象零件和抽象产品的部分。接下来要进入抽象工厂的部分。

程序 8-5 的 getFactory 方法是以指定类名的方式产生具体工厂的对象实例。以字符串把欲产生的具体工厂的类名称指定到变量 classname，例如

"listfactory.ListFactory"

getFactory 则利用 Class 类的 forName 方法动态地读取该类。接着再利用 newInstance 方法产生一个该类的对象实例。这个对象实例就是 getFactory 的返回值。

Class 类属于 java.lang 包，是“表示类的类”。Class 类就在 Java 的标准类函数库内。forName 是 java.lang.Class 的类方法（static 方法），newInstance 是 java.lang.Class 的对象实例方法。

请注意，即使是在 getFactory 这个方法中建立具体工厂的对象实例，返回值仍然是抽象工厂。

在抽象工厂制作零件或产品时，会使用到 createLink、createTray、createPage 这几

**Java语言中的应用**

个方法。这些方法都是抽象方法，实际上的具体零件和产品制作过程则交给 Factory 的子类处理（subclass responsibility）。但是方法的名称和 signature 一定要在这里规定才行。

**程序 8-5 Factory 类 (Factory.java)**

```
package factory;

public abstract class Factory {
    public static Factory getFactory(String classname) {
        Factory factory = null;
        try {
            factory = (Factory)Class.forName(classname).newInstance();
        } catch (ClassNotFoundException e) {
            System.err.println("找不到类" + classname + ". ");
        } catch (Exception e) {
            e.printStackTrace();
        }
        return factory;
    }
    public abstract Link createLink(String caption, String url);
    public abstract Tray createTray(String caption);
    public abstract Page createPage(String title, String author);
}
```

**利用工厂把零件组合成产品：Main 类**

前面已经看过了抽象零件和产品工厂的源代码，接下来要进入到 Main 类（程序 8-6）。这里是利用抽象工厂生产抽象零件。然后再组装成抽象产品。从 import 进来的只有 factory 包就可以知道，这个类没有利用到任何具体零件、产品或工厂。

具体工厂的类名称要在命令行指定。例如，如果要使用 listfactory 包的 ListFactory 类，就要在命令行输入：

```
java Main listfactory.ListFactory
```

接着就可以根据这个参数变量（args[0]）在 getFactory 建立工厂，指定到变量 factory。

然后利用 factory 建立 Link、Tray，把 Link 和 Tray 放入到 Tray 内，最后再建立 Page，进行 output。

## Abstract Factory (抽象工厂) —— 把相关零件组合成产品

程序 8-6 Main 类 (Main.java)

```
import factory.*;

public class Main {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java Main class.name.of.Concrete
Factory");
            System.out.println("Example 1: java Main listfactory.List
Factory");
            System.out.println("Example 2: java Main tablefactory.Table
Factory");
            System.exit(0);
        }
        Factory factory = Factory.getFactory(args[0]);
        Link asahi = factory.createLink("朝日新闻", "http://www.asahi.
com/");
        Link yomiuri = factory.createLink("读卖新闻", "http://www.yomiuri.co.jp/");
        Link us_yahoo = factory.createLink("Yahoo!", "http://www.yahoo.com/");
        Link jp_yahoo = factory.createLink("Yahoo!Japan", "http://www.yahoo.co.jp/");
        Link excite = factory.createLink("Excite", "http://www.excite.com/");
        Link google = factory.createLink("Google", "http://www.google.com/");
        Tray traynews = factory.createTray("新闻");
        traynews.add(asahi);
        traynews.add(yomiuri);
        Tray trayyahoo = factory.createTray("Yahoo!");
        trayyahoo.add(us_yahoo);
        trayyahoo.add(jp_yahoo);
```

## 设计模式：工厂方法模式

```

        Tray traysearch = factory.createTray("搜索引擎");
        traysearch.add(trayyahoo);
        traysearch.add(excite);
        traysearch.add(google);

        Page page = factory.createPage("LinkPage", "结城 浩");
        page.add(traynews);
        page.add(traysearch);
        page.output();
    }
}

```

---

**具体工厂：ListFactory 类**

好了，看过那么多跟抽象有关的程序代码之后，请各位换换口味改看具体部分的程序代码。首先是 listfactory 包的工厂——ListFactory 类。

ListFactory 类（程序 8-7）实现 Factory 类的抽象方法 createLink、createTray 和 createPage。这里的实现工具调用 new 函数把 ListLink、ListTray、ListPage 创建出来（有些程序不是做 new 的操作，或许会利用 Prototype Pattern 进行 clone）。

**程序 8-7 ListFactory 类（程序 Factory.java）**

```

package listfactory;
import factory.*;

public class ListFactory extends Factory {
    public Link createLink(String caption, String url) {
        return new ListLink(caption, url);
    }
    public Tray createTray(String caption) {
        return new ListTray(caption);
    }
    public Page createPage(String title, String author) {
        return new ListPage(title, author);
    }
}

```

---

**具体零件：ListLink 类**

ListLink 类（程序 8-8）是 Link 类的子类。它是不是非实现不可的方法？嗯，好

## Abstract Factory (抽象工厂) —— 把相关零件组合成产品

聪明，它在父类的时候是抽象方法 (makeHTML)。ListLink 利用标签<li>和<a>来产生 HTML 的片段。就好比形影不离的螺栓和螺帽能搭配得天衣无缝一样，我们也能把程序写成让这个 HTML 的片段跟其他的 ListTray 或 ListPage 成为绝配。

**程序 8-8 ListLink 类 (程序 Link.java)**

```
package listfactory;
import factory.*;

public class ListLink extends Link {
    public ListLink(String caption, String url) {
        super(caption, url);
    }
    public String makeHTML() {
        return "<li><a href=\"" + url + "\">" + caption +
"</a></li>\n";
    }
}
```

## 具体零件：ListTray 类

ListTray 类 (程序 8-9) 是 Tray 类的子类。这里也要注意 makeHTML 是如何被实现。tray 字段是要输出成 HTML 的一些 Item，让这些 Item 能表示成 HTML 的标签就是 makeHTML 方法的使命。那么，到底是怎么做的呢？

先利用<li>输出项目符号 (caption)，然后再以<ul>和<li>一个个输出 Item。输出结果要先集中到 StringBuffer，最后才用 toString 转换成 String。

这些一个个的 Item 要怎样才能变身成 HTML 格式？当然得调用各个 Item 的 makeHTML。不必管变量 item 内容到底是 ListLink 还是 ListTray 的对象实例，只要用下面这个简单的表达式即可：

```
item.makeHTML()
```

这部分的程序可不能写成先检查变量 item 的实际内容后再利用 switch 语句或 if 语句来处理。因为这样会让程序完全脱离面向对象的原则。变量 item 是 Item 类型，Item 类已经声明了 makeHTML 方法；再加上 ListLink 和 ListTray 两者都是 Item 类的子类，所以可以放心去调用 makeHTML 方法。剩下的就等 item 选择最佳时机来处理 makeHTML 方法即可。至于如何处理，也只有 item 这个 instance (对象) 才知道。这就是面向对象的优点。

这里所使用的 java.util.Iterator 类在功能上与前面 Iterator Pattern 的所学都一样，唯

有一点不同：这里是作为 Java 的类函数库。利用 java.util.Vector 类建立 java.util.Iterator 类时要使用 iterator 方法。

#### 程序 8-9 ListTray 类（程序 Tray.java）

```
package listfactory;
import factory.*;
import java.util.Iterator;

public class ListTray extends Tray {
    public ListTray(String caption) {
        super(caption);
    }
    public String makeHTML() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("<li>\n");
        buffer.append(caption + "\n");
        buffer.append("<ul>\n");
        Iterator it = tray.iterator();
        while (it.hasNext()) {
            Item item = (Item)it.next();
            buffer.append(item.makeHTML());
        }
        buffer.append("</ul>\n");
        buffer.append("</li>\n");
        return buffer.toString();
    }
}
```

### 具体产品：ListPage 类

ListPage（程序 8-10）是 Page 类的子类。相信各位对 makeHTML 方法已经很了解了。ListPage 利用字段的内容来产生网页。网页作者姓名（author）以<address>标签来表示。

各位是否了解 while 语句要放在<ul>…</ul>中间的理由？原因是：while 语句内被 append 的 item.makeHTML() 的输出结果要放在<ul>里面。请翻回前面复习一下 ListLink 和 ListTray 的 makeHTML 方法，<li>标签是不是都在最外面？这里就等于是“螺栓”跟“螺帽”的连接点。

while 前面的 content 则是继承自 Page 类的字段。

## Abstract Factory (抽象工厂) —— 把相关零件组合成产品

**程序 8-10 ListPage 类 (程序 Page.java)**

```

package listfactory;
import factory.*;
import java.util.Iterator;

public class ListPage extends Page {
    public ListPage(String title, String author) {
        super(title, author);
    }
    public String makeHTML() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("<html><head><title>" + title + "</title>
</head> \n");
        buffer.append("<body>\n");
        buffer.append("<h1>" + title + "</h1>\n");
        buffer.append("<ul>\n");
        Iterator it = content.iterator();
        while (it.hasNext()) {
            Item item = (Item)it.next();
            buffer.append(item.makeHTML());
        }
        buffer.append("</ul>\n");
        buffer.append("<hr><address>" + author + "</address>");
        buffer.append("</body></html>\n");
        return buffer.toString();
    }
}

```

### 在程序示例中新增其他的具体工厂

虽然篇幅有点长，不过请各位再耐心看下去。前节已经介绍过抽象工厂和具体工厂。但是如果只想建立一个 HTML 格式的相关网页链接，好像有点小题大作。况且若具体工厂只有 1 个，根本不需要去区分抽象或具体。下面就试着在原来的程序示例中新增其他的具体工厂（即产生其他格式的 HTML 文件）。

前面的 listfactory 包设计成含有项目符号，假设现在要利用 tablefactory 包来设计一个窗体 (table)。图 8-6 是编译及执行结果，图 8-7 则为完成的 LinkPage.html。在 Web 浏览器下所见的画面则如图 8-8 所示（请跟图 8-2 比较看看）。

## 8.1.2 表驱动应用

```
javac Main.java tablefactory/TableFactory.java
java Main tablefactory.TableFactory
已产生 LinkPage.html.
```

图 8-6 编译与执行结果

```
<html><head><title>LinkPage</title></head>
<body>
<h1>LinkPage</h1>
<table width="80%" border="3">
<tr><td><table width="100%" border="1"><tr><td bgcolor="#cccccc" align="center" colspan="2"><b>新闻</b></td></tr>
<tr>
<td><a href="http://www.asahi.com/">朝日新闻</a></td>
<td><a href="http://www.yomiuri.co.jp/">读卖新闻</a></td>
</tr></table></td></tr><tr><td><table width="100%" border="1">
<tr>
<td bgcolor="#cccccc" align="center" colspan="3"><b>搜索引擎</b></td>
</tr>
<tr><td><table width="100%" border="1"><tr><td bgcolor="#cccccc" align="center" colspan="2"><b>Yahoo!</b></td></tr>
<tr>
<td><a href="http://www.yahoo.com/">Yahoo!</a></td>
<td><a href="http://www.yahoo.co.jp/">Yahoo!Japan</a></td>
</tr></table></td><td><a href="http://www.excite.com/">Excite</a></td>
<td><a href="http://www.google.com/">Google</a></td>
</tr></table></td></tr></table>
<hr><address>结城 浩</address></body></html>
```

图 8-7 利用 tablefactory 写成的相关网站链接 (HTML)

## Abstract Factory (抽象工厂) —— 把相关零件组合成产品



图 8-8 利用 tablefactory 写成的相关网站链接（在浏览器下的样子）

表 8-2 类一览表（请跟表 8-1 比较）

| 包           | 名称           | 说明                                         |
|-------------|--------------|--------------------------------------------|
| listfactory | TableFactory | 表示具体工厂的类（产生 TableLink、TableTray、TablePage） |
| listfactory | TableLink    | 具体零件：表示 HTML 链接的类                          |
| listfactory | TableTray    | 具体零件：抽取 TableLink 和 TableTray 的类           |
| listfactory | TablePage    | 具体零件：表示 HTML 网页的类                          |

## 具体工厂：TableFactory 类

TableFactory 类(程序 8-11)是 Factory 类的子类。createLink、createTray、createPage 分别产生 TableLink、TableTray、TablePage 等类的对象实例。

## 程序 8-11 TableFactory 类 (TableFactory.java)

```

package tablefactory;
import factory.*;

public class TableFactory extends Factory {
    public Link createLink(String caption, String url) {
        return new TableLink(caption, url);
    }
}

```

## 第八章 特别的应用

```

public Tray createTray(String caption) {
    return new TableTray(caption);
}
public Page createPage(String title, String author) {
    return new TablePage(title, author);
}
}

```

---

**具体零件：TableLink 类**

TableLink 类（程序 8-12）是 Link 类的子类。MakeHTML 使用表示表格 1 行的<td>卷标。还记得吗，ListLink 类（程序 8-8）是用<li>标签。这次我们用的是<td>标签。

**程序 8-12 TableLink 类 (TableLink.java)**

```

package tablefactory;
import factory.*;

public class TableLink extends Link {
    public TableLink(String caption, String url) {
        super(caption, url);
    }
    public String makeHTML() {
        return "<td><a href=\"" + url + "\">" + caption + "</a></td>\n";
    }
}

```

---

**具体零件：TableTray 类**

TableTray 类（程序 8-13）是 Tray 类的子类。makeHTML 利用<td>和<table>标签输出各个 Item。

**程序 8-13 TableTray 类 (TableTray.java)**

```

package tablefactory;
import factory.*;
import java.util.Iterator;

public class TableTray extends Tray {
    public TableTray(String caption) {

```

## Abstract Factory (抽象工厂) —— 把相关零件组合成产品

```

        super(caption);
    }

    public String makeHTML() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("<td>");
        buffer.append("<table width=\"100%\" border=\"1\"><tr>");
        buffer.append("<td bgcolor=\"#cccccc\" align=\"center\" colspan=\"" + tray.size() + "\"><b>" + caption + "</b></td>");
        buffer.append("</tr>\n");
        buffer.append("<tr>\n");
        Iterator it = tray.iterator();
        while (it.hasNext()) {
            Item item = (Item)it.next();
            buffer.append(item.makeHTML());
        }
        buffer.append("</tr></table>");
        buffer.append("</td>");
        return buffer.toString();
    }
}

```

---

### 具体产品：TablePage 类

TablePage 类（程序 8-14）是 Page 类的子类。我想各位应该不需要再听太琐碎的解释说明，跟 ListPage 类（程序 8-10）对比就可以清楚它们之间的对应关系了。

**程序 8-14 TablePage 类 (TablePage.java)**

```

package tablefactory;
import factory.*;
import java.util.Iterator;

public class TablePage extends Page {
    public TablePage(String title, String author) {
        super(title, author);
    }
    public String makeHTML() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("<html><head><title>" + title + "</title>
</head>\n");

```

## Java 模板方法模式

```

        buffer.append("<body>\n");
        buffer.append("<h1>" + title + "</h1>\n");
        buffer.append("<table width=\"80%\" border=\"3\">\n");
        Iterator it = content.iterator();
        while (it.hasNext()) {
            Item item = (Item)it.next();
            buffer.append("<tr>" + item.makeHTML() + "</tr>");
        }
        buffer.append("</table>\n");
        buffer.append("<hr><address>" + author + "</address>");
        buffer.append("</body></html>\n");
        return buffer.toString();
    }
}

```

---

## Abstract Factory Pattern 的所有参与者

Abstract Factory Pattern 出现过的所有参与者整理如下。

### ◆ AbstractProduct (抽象产品) 参与者

AbstractProduct 参与者规定由 AbstractFactory 参与者所产生的抽象零件和产品的接口 (API)。在前面的程序示例中，扮演这个参与者的是 Link 类、Tray 类和 Page 类。

### ◆ AbstractFactory (抽象工厂) 参与者

AbstractFactory 参与者则规定用来产生 AbstractProduct 参与者的对象实例的接口 (API)。在前面的程序示例中，扮演这个参与者的是 Factory 类。

### ◆ Client (客户) 参与者

Client 参与者是一个只使用 AbstractFactory 参与者和 AbstractProduct 参与者的接口 (API) 来完成工作的参与者。Client 参与者并不知道具体零件、产品或工厂。在前面的程序示例中，扮演这个参与者的是 Main 类。对了，图 8-9 省略掉 Client 参与者。

### ◆ ConcreteProduct (具体产品) 参与者

ConcreteProduct 参与者是实现 AbstractProduct 参与者的界面 (API)。在前面的程序示例当中，各个包里扮演这个参与者的是：

- listfactory 包——ListLink 类、ListTray 类、ListPage 类
- tablefactory 包——TableLink 类、TableTray 类、TablePage 类

## Abstract Factory (抽象工厂) —— 把相关零件组合成产品

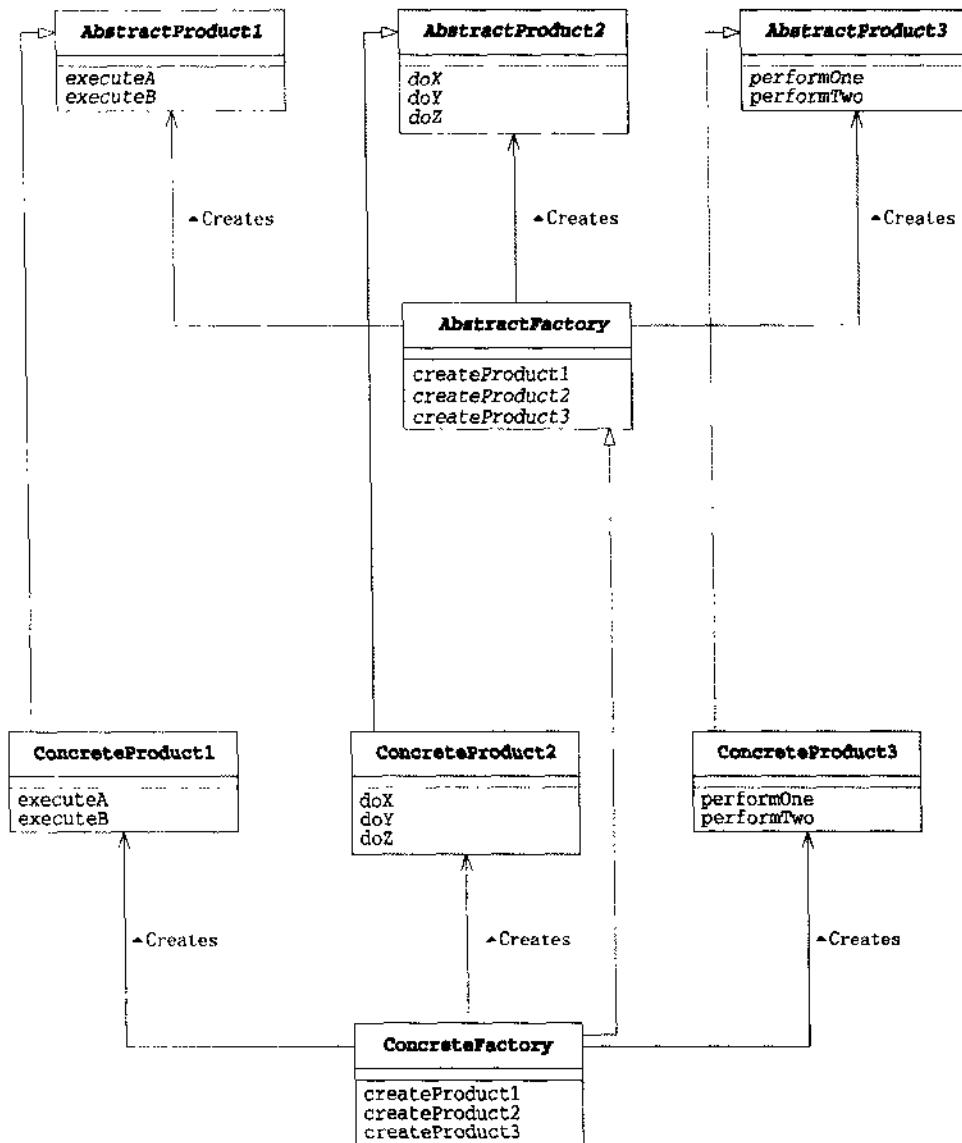


图 8-9 Abstract Factory Pattern 的类图

### ◆ ConcreteFactory (具体工厂) 参与者

ConcreteFactory 参与者是实现 AbstractFactory 参与者的界面 (API)。在前面的程序示例当中，各个包里扮演这个参与者的是：

- `listfactory` 包——`ListFactory` 类
- `tablefactory` 包——`TableFactory` 类

## 扩展自我视野的提示

### 新增具体工厂尤如探囊取物

如果想在 Abstract Factory Pattern 里新增具体工厂，还真是一件简单的任务。因为要建立哪些类、该实现哪些方法都一清二楚，当然就简单啰。

假设现在想在程序示例中新增其他具体工厂。固定操作不外乎就是建立 Factory、Link、Tray、Page 的子类，实现各个抽象方法。换句话说，就是要让含有 factory 包的类的抽象部分全部具体化。

此时，无论新增几个具体工厂（或修改具体工厂的 bug），都不需要再去修改抽象工厂或 Main 的部分。

### 新增零件很有难度

如果想在 Abstract Factory Pattern 新增其他零件时，该怎么做？假设现在想在 factory 包新增一个显示影像的 Picture 零件。此时，所有已经存在的具体工厂都要修改成支持 Picture。以 listfactory 包为例，这里要做的修改操作包括：

- 在 ListFactory 类新增 createPicture 方法
- 新增一个 ListPicture 类

已经完工的具体工厂越多，修改操作就越多。

## 相关 Pattern

### ◆ *Builder Pattern* (第 7 章)

Abstract Factory Pattern 是接口 (API) 规定的抽象零件，产生构造较复杂的对象实例。

Builder Pattern 循序渐进产生较大规模的对象实例。

### ◆ *Factory Method Pattern* (第 4 章)

Abstract Factory Pattern 产生产品和零件的部分可能会变成 Factory Method Pattern。

### ◆ *Composite Pattern* (第 11 章)

以 Abstract Factory Pattern 所产生的产品可能会变成 Composite Pattern。

### ◆ *Singleton Pattern* (第 5 章)

Abstract Factory Pattern 的具体工厂可能会变成 Singleton Pattern。

## Abstract Factory (抽象工厂) —— 把相关零件组合成产品

## 深入说明：各种建立对象实例的手段

Java 有以下几种方法可建立对象实例。

### ◆ new

通常是利用 Java 的关键字 `new` 来建立对象实例。

若程序写成如下，则可建立 `Something` 类的对象实例后再指定到变量 `obj`。

```
Something obj = new Something();
```

此时，必须把类名（在此为 `Something`）写在源代码之内。

### ◆ clone

若利用 Prototype Pattern（第 6 章）所介绍的 `clone` 方法，则可根据现有的对象实例建立另一个新对象实例。

若程序写成如下，则可根据当前对象本身（`this`）建立另一个新对象实例（但并未调用构造函数）。

```
class Something {
    ...
    public Something createClone() {
        Something obj = null;
        try {
            obj = (Something)clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return obj;
    }
}
```

### ◆ newInstance

利用本章所介绍的 `java.lang.Class` 类的 `newInstance` 方法，则可根据 `Class` 的对象实例，建立该 `Class` 所表示的类的对象实例（调用无参数的构造函数）。

本章的程序示例也使用到 `newInstance`（程序 8-5）。下面是其他的 `newInstance` 示例。假设有一个 `Something` 类的对象实例 `someobj`，则以下面的表达式可另建一个 `Something` 类的对象实例。

```
someobj.getClass().newInstance()
```

事实上 `newInstance` 方法会抛出 `InstantiationException` 或 `IllegalAccessException` 异常，所以前后要分别加上 `try ... catch` 或在方法里声明 `throws`。

## 重点回顾

本章所介绍的是将抽象零件组合成抽象产品的抽象工厂——Abstract Factory Pattern。

笔者当初刚刚开始学习设计 Pattern 时，觉得其中最困难的就是这个 Abstract Factory Pattern。不仅结构复杂，而且也出现很多种类。这跟从头到尾只有 1 个类的 Singleton Pattern 差很多，如果把 Singleton Pattern 比喻成独角戏的话，那么 Abstract Factory Pattern 就是群魔乱舞了。

## 练习题

解答请见附录 A

### 问题 1

在 Tray 类（程序 8-3）当中，tray 字段被定义为 `protected`，可以从子类直接引用。请说明若 tray 字段改定义为 `private` 时的优缺点。

### 问题 2

假设现在想在程序示例的 Factory 类（程序 8-5）定义一个“可产生只有 Yahoo! (<http://www.yahoo.com/>) 链接的网页的具体方法”，即：

```
public Page createYahooPage()
```

（网页作者和标题均为“Yahoo!”）此时，具体工厂和具体零件的部分必须配合做哪些修改操作？

### 问题 3

请看 ListLink 类（程序 8-8）的构造函数：

```
public ListLink(String caption, String url) {  
    super(caption, url);  
}
```

也就是说，它的作用只是调用父类的构造函数而已。如果不需做什么特殊处理的话，为什么要特地定义 ListLink 的构造函数呢？

### 问题 4

Page 类（程序 8-4）的作用跟 Tray 类（程序 8-3）很像，为什么不干脆把 Page 类写成是 Tray 类的子类呢？

# 第9章

## Bridge（桥接） ——分成功能层次和实现层次

# Bridge Pattern

本章要学习的是

## Bridge Pattern

bridge 是“桥梁”的意思。就像现实社会里的桥梁担负着连接两岸的重大责任一样，Bridge Pattern 也是负责连接 2 个不同位置的参与者。

Bridge Pattern 沟通着“功能的类层次”和“实现的类层次”

沟通“功能的类层次”和“实现的类层次”？听起来有点天马行空，很难在脑海中勾勒出具体的形象。

在开始研究 Bridge Pattern 的程序示例之前，笔者要先跟各位说明：

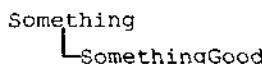
- 功能的类层次
- 实现的类层次

因为如果不了解两岸是什么样的地理环境，就不容易了解桥梁的意义。

## 类层次的两大责任

### ◆ 故新增“功能”时

假设现在有一个类 Something，如果想要对 Something 新增其他功能（例如，想多加一个新方法），首先要建立 SomethingGood 类作为 Something 的子类（子类、衍生类或扩充类）。这就是一个小型的类层次。



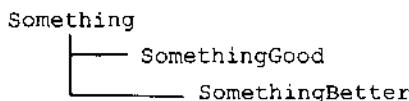
这个层次是为了新增功能而建立。

- 基本功能放在父类
- 新功能则新增到子类

这个类层次就称为：

“功能的类层次”

假设现在又想对 SomethingGood 类新增另一个功能。此时，要建立 SomethingBetter 类作为 SomethingGood 类的子类。如此一来，功能的类层次又往下延伸一层。



如欲追加新功能时，找出类层次中最接近的目的类，然后建立一个子类、建立一个有该功能的新类。这就是一个功能的类层次。

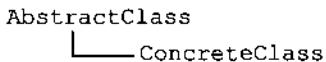
## Bridge (桥接) —— 分成功能层次和实现层次

注意：基本上，类层次最好不要建立太多。

### ◆ 欲新增“实现”时

在前面的 Template Method Pattern (第 3 章) 等部分已经学过了抽象类的作用。抽象类把一连串的方法声明成抽象方法，再规定接口 (API)。然后由子类实现这个抽象方法。父类的作用是利用抽象方法来规定接口 (API)，子类的作用则是进行实现。这样把父类和子类扮演的参与者分工，可以建立起高价值零组件 (可更换性) 的类。

这里也有类层次的影子，假设实现父类 AbstractClass 的抽象方法的子类是 ConcreteClass，则可建立起下面这个小型类层次。



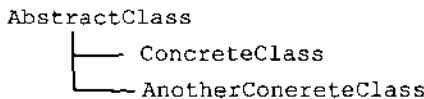
不过这里的类层次并不是为了新增功能，因为类层次没有追加新方法的目的。这个类层次有下列的参与者分工：

- 父类使用抽象方法来规定接口 (API)
- 子类使用具体方法来实现此接口 (API)

这样的类层次就称为：

“实现的类层次”

假设现在要建立另一个 AbstractClass 的实现，若其子类为 AnotherConcreteClass，则实现的类层次会有一点变化。



如果要建立一个新的实现，必须建立 AbstractClass 的子类，然后实现抽象方法。这就是实现的类层次。

### ◆ 类层次的固处一室和独立分离

看过前面的说明之后，相信各位应该了解什么是功能的类层次、什么是实现的类层次。当我们有一个念头“好，现在要做一个子类”的时候，请先确认自己想要完成的内容：“这个行为是要新增功能？还是要进行实现？”。如果类层次只有一个的话，功能的类层次和实现的类层次就会放在同一个层次构造里。这样可能会让类层次变得太复杂，而且不容易预测后面的发展。因为自己在建立子类时，常常会搞不清楚应该放在类层次的哪个位置。

既然如此，那就把“功能的类层次”和“实现的类层次”分成 2 个独立的类层次吧。但是一分为二可能会弄得支离破碎，所以必须在 2 个类层次之间建一座沟通的桥梁。本章所要介绍的 Bridge Pattern 就是扮演沟通桥梁的 Design Pattern。

现在就来研究 Bridge Pattern，阅读程序示例时一定要有“2 个类层次”的概念。

## 程序示例

请看下面应用 Bridge Pattern 的程序示例。这个程序是用来“打印内容”，别怪笔者现在说得太抽象，耐心看下去就能慢慢建立起具体概念了。

表 9-1 类一览表

| 桥的哪一边  | 名称                | 说明             |
|--------|-------------------|----------------|
| 功能的类层次 | Display           | “打印内容”的类       |
| 功能的类层次 | CountDisplay      | 新增“打印指定次数”功能的类 |
| 实现的类层次 | DisplayImpl       | “打印内容”的类       |
| 实现的类层次 | StringDisplayImpl | “以字符串进行打印”的类   |
|        | Main              | 测试用的类          |

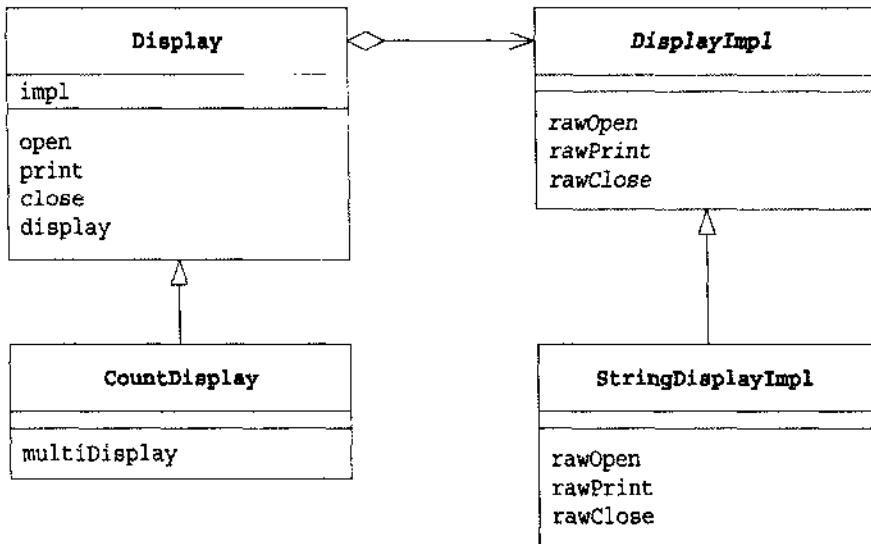


图 9-1 程序示例的类图

### 功能的类层次：Display 类

Display 类（程序 9-1）是抽象的“打印内容”。这个类位于“功能的类层次”的最上层。

impl 字段是表示 Display 类的“实现”的对象实例（impl 是 implementation（实现）的缩写）。

把表示实现的类的对象实例传递给构造函数。参数所传递的对象实例会存储在

## Bridge（桥接）——分成功能层次和实现层次

`impl` 字段，供后面的处理使用（这个字段就是 2 个类层次的“桥梁”）。

`open`, `print`, `close` 这 3 个方法是 `Display` 类所提供的接口（API），表示打印的程序。其详细分工为：

- `open` → 打印的前处理
- `print` → 打印行为
- `close` → 打印的后处理

请看这 3 个方法的内容。使用 `impl` 字段的实现的方法来执行这个方法，就可以让 `Display` 的接口（API）转换成 `DisplayImpl` 的接口（API）。

`display` 方法则使用 `open`, `print`, `close` 这几个 `Display` 的接口（API）来进行“打印”的处理操作。

程序 9-1 `Display` 类 (`Display.java`)

---

```
public class Display {
    private DisplayImpl impl;
    public Display(DisplayImpl impl) {
        this.impl = impl;
    }
    public void open() {
        impl.rawOpen();
    }
    public void print() {
        impl.rawPrint();
    }
    public void close() {
        impl.rawClose();
    }
    public final void display() {
        open();
        print();
        close();
    }
}
```

---

### 功能的类层次：`CountDisplay` 类

把功能新增到 `Display` 类的是 `CountDisplay` 类（程序 9-2）。虽然 `Display` 类只有“打印”这个功能，但是我们可以在 `CountDisplay` 类新增一个“打印指定次数”的功能，即 `multiDisplay` 方法。

CountDisplay 类继承了 Display 类 (open, print, close) 的方法并追加新的方法。以上是“功能的类层次”。

#### 程序 9-2 CountDisplay 类 (CountDisplay.java)

```
public class CountDisplay extends Display {
    public CountDisplay(DisplayImpl impl) {
        super(impl);
    }
    public void multiDisplay(int times) { // 反复 times 次再打印
        open();
        for (int i = 0; i < times; i++) {
            print();
        }
        close();
    }
}
```

## 实现的类层次：DisplayImpl 类

DisplayImpl 类（程序 9-3）位于“实现的类层次”的最上层。

DisplayImpl 类是抽象类，有 rawOpen, rawPrint, rawClose 这 3 个方法，分别对应到 Display 类的 open、print、close 方法，执行前处理、打印和后处理。

#### 程序 9-3 DisplayImpl 类 (DisplayImpl.java)

```
public abstract class DisplayImpl {
    public abstract void rawOpen();
    public abstract void rawPrint();
    public abstract void rawClose();
}
```

## 实现的类层次：StringDisplayImpl 类

接下来要进入真正的“实现”。StringDisplayImpl 类（程序 9-4）是打印字符串的类。但它不只有打印的功能，DisplayImpl 类的子类还可以使用 rawOpen, rawPrint, rawClose 等方法进行打印。

## Bridge (桥接) —— 分成功能层次和实现层次

程序 9-4 StringDisplayImpl 类 (StringDisplayImpl.java)

```

public class StringDisplayImpl extends DisplayImpl {
    private String string; // 应打印的字符串。
    private int width; // 以 byte 为单位所求出的字符串“宽度”。
    public StringDisplayImpl(String string) {
        // 构造函数传递过来的字符串 string,
        this.string = string; // 先存储在字段。
        this.width = string.getBytes().length;
        // 接着把 byte 单位的宽度也存储到字段，等到后面再利用。
    }
    public void rawOpen() {
        printLine();
    }
    public void rawPrint() {
        System.out.println("|" + string + "|"); // 打印时前后加上
        // "|.
    }
    public void rawClose() {
        printLine();
    }
    private void printLine() {
        System.out.print("+"); // 输出框角的 "+" 记号。
        for (int i = 0; i < width; i++) { // 输出 width 个 "-",
            System.out.print("-"); // 作为框线。
        }
        System.out.println("+"); // 输出框角的 "+" 记号。
    }
}

```

DisplayImpl 和 StringDisplayImpl 这 2 个类相当于“实现的类层次”。

## Main 类

Main 类则组合上述 4 个类以输出字符串。先把 Display 类的对象实例指定到变量 d1，再把 CountDisplay 类的对象实例指定到变量 d2、d3。两者都是由 StringDisplayImpl 类的对象实例来进行实现。

执行结果即如图 9-2 所示。d1、d2、d3 都是 Display 类的对象实例，可以调用 display

## Java语言中的应用

方法。d3 还可以调用 multiDisplay 方法。

## 程序 9-5 Main 类 (Main.java)

```
public class Main {  
    public static void main(String[] args) {  
        Display d1 = new Display(new StringDisplayImpl("Hello,  
Japan."));  
        Display d2 = new CountDisplay(new StringDisplayImpl("Hello,  
World."));  
        CountDisplay d3 = new CountDisplay(new StringDisplayImpl  
("Hello, Universe."));  
        d1.display();  
        d2.display();  
        d3.display();  
        d3.multiDisplay(5);  
    }  
}
```

|                  |                         |
|------------------|-------------------------|
| +-----+          | ← 打印 d1.display()       |
| Hello, Japan.    |                         |
| +-----+          |                         |
| +-----+          | ← 打印 d2.display()       |
| Hello, World.    |                         |
| +-----+          |                         |
| +-----+          | ← 打印 d3.display()       |
| Hello, Universe. |                         |
| +-----+          |                         |
| +-----+          | ← 打印 d3.multiDisplay(5) |
| Hello, Universe. |                         |
| +-----+          |                         |

图 9-2 执行结果

## Bridge Pattern 的所有参与者

Bridge Pattern 中出现的所有参与者可整理如下。

### ◆ Abstraction (抽象化) 参与者

位于“功能的类层次”最上层的类，利用 Implementor 参与者的方法只记载基本功能的类。这个对象实例是保持住 Implementor 参与者。在前面的程序示例中，扮演这个参与者的是 Display 类。

### ◆ RefinedAbstraction (改良后的抽象化) 参与者

对 Abstraction 参与者新增功能的参与者。在前面的程序示例中，扮演这个参与者的是 CountDisplay 类。

### ◆ Implementor (实现者) 参与者

位于“实现的类层次”最上层的类，规定要实现 Abstraction 参与者的接口（API）的方法。在前面的程序示例中，扮演这个参与者的是 DisplayImpl 类。

### ◆ ConcreteImplementor (具体实现者) 参与者

具体实现 Implementor 参与者的接口（API）。在前面的程序示例中，扮演这个参与者的是 StringDisplayImpl 类。

若以类图来画出 Bridge Pattern，即可得到图 9-3。左边 2 个是功能的类层次，右边 2 个则是实现的类层次。看得出来吧，连接 2 个层次的就是 impl 字段。

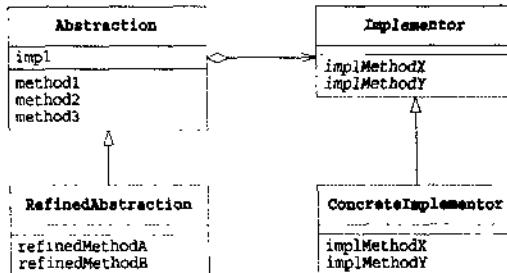


图 9-3 Bridge Pattern 的类图

## 扩展自我视野的提示

### 先区分再扩充就很轻松

Bridge Pattern 的最大特征就在区分成“功能的类层次”和“实现的类层次”。如

果先把这 2 个类层次区分清楚，就可以分别扩充两边的类层次（具体的扩充示例请见练习题）。

如果想新增功能的话，就在功能的类层次追加类。这时候根本不需要修改实现的类层次，而且所有新增加的功能都可以利用“所有实现”来使用。

举例来说，假设现在想把实现的类层次嵌入到 OS。程序里会有因 OS 不同而变的部分，姑且分成 Windows 版、Macintosh 版和 UNIX 版三种类型。这个因 OS 不同而变的部分就要用 Bridge Pattern 的“实现的类层次”来表现。也就是说，先决定所有 OS 共享的接口（API）设成 Implementor 参与者，再建立 3 个类（Windows 版、Macintosh 版、UNIX 版）作为 ConcreteImplementor 参与者。如此一来，无论在“功能的类层次”增加多少功能，都能同时对应 3 种不同的 OS。

## 继承是生死不离、委托是潇洒分手

在扩充类时，“继承”是相当好用的方法，但是它会让类间的关系牢不可分。如果源代码写成：

```
class SomethingGood extends Something {
    ...
}
```

则 SomethingGood 类会变成 Something 类的子类。而且只要你不修改源程序，这个关系就永远不变。如果想根据需要更改类间的关系时，继承就有点不适合，毕竟每次更动都要修改源代码，哪有那么多闲功夫！遇到这种情形时，就该使用“委托”而非“继承”。

前面的程序示例在 Display 类中就使用了委托。实现的对象实例被保留在 Display 类的 impl 字段内，所以会“轮流”：

- 若执行 open 时，调用 impl.rawOpen()
- 若执行 print 时，调用 impl.rawPrint()
- 若执行 close 时，调用 impl.rawClose()

只要一声“该做事了”，就会“交给 impl 办”。这就是委托。

继承是一种牢不可分的关系，但委托却是说分手就分手。因为它的对象是在产生 Display 类的对象实例时传递给参数的内容。在程序示例中，当 Main 类产生 Display 和 CountDisplay 的对象实例时，就把 StringDisplayImpl 的对象实例传递给参数。

如果还有一个非 StringDisplayImpl 类的 ConcreteImplementor 参与者，而传递给 Display 和 CountDisplay 的是这个对象实例，则实现就会很干脆的切换过去。这个切换中被修改的只有 Main 类而已，不需要动到 Display 或 DisplayImpl 等的源程序。

继承是生死不离，委托则是潇洒分手。在设计多项类时，必须确实搞清这个概念。

## Bridge (桥接) —— 分功能层次和实现层次

继承和委托的关系在 Template Method Pattern (第 3 章) 也提到过, 可翻到前面参考看看。

## 相关 Pattern

### ◆ *Template Method Pattern (第 3 章)*

Template Method Pattern 是利用实现的类层次。在父类使用抽象方法设计程序, 到了类再实现该抽象方法。

### ◆ *Abstract Factory Pattern (第 8 章)*

为了让 Bridge Pattern 中的 ConcreteImplementor 参与者能设计得更配合环境条件, 有时会需要用到 Abstract Factory Pattern。

### ◆ *Adapter Pattern (第 2 章)*

Bridge Pattern 是把功能的类层次和实现的类层次两者区分清楚后再结合使用的 Pattern。

Adapter Pattern 的功能很类似, 但接口 (API) 是结合不同类的 Pattern。

## 重点回顾

本章所介绍的是在 2 种类层次间扮演沟通桥梁的 Bridge Pattern。把 2 种不同的类层次区隔开来, 会更容易看清楚类该往哪个方向扩充。另外还学到不紧密连接类的委托。

好啦, 请各位利用下面的练习题检查一下学习的吸收程度如何。

## 练习题

解答请见附录 A

### 问题 1

请在本章的程序示例新增一个类, 让它具有“打印随机次数”的处理能力。此时应该要扩充哪个类?

提示: 假设打印的方法是 void randomDisplay(int times), 只有遇到大于 0 但小于 times 的时候才打印随机次数。

### 问题 2

请在本章的程序示例新增一个类, 让它具有“输出文本文件内容”的处理能力。

此时应该要扩充哪个类？

### 问题 3

假设现在要在本章的程序示例新增一个打印结果如图 9-4 或图 9-5 格式的类。

```
<>  
<*>  
<**>  
<***>
```

图 9-4 格式示例 1

```
| -  
| # #-  
| ##### -  
| ##### #-  
| ##### #-  
| ##### #-
```

图 9-5 格式示例 2

每行的格式设定如下：

打印第一个字符 → 打印多个装饰字符 → 打印结束字符并换行

反复执行多行，装饰字符数量会随着反复次数而增加。

在程序示例中新增上述动作的类时，是应该放到功能的类层次？还是应该放在实现的类层次？

因为追加的是不同打印结果的方法，所以应该是功能的类层次？或者因为追加的是字符显示的打印结果，所以应该是实现的类层次？请想想看该怎样放进这个 Bridge Pattern。

# 第 10 章

Strategy (策略)

——把算法整个  
换掉

## Strategy Pattern

本章要学习的是

### Strategy Pattern

**strategy** 是“战略”的意思，泛指与敌方对峙时的作战策略、指挥军队的教战守策以及解决问题的方法等。在程序设计里，不妨把它视为“算法”。

任何一种程序都是为了解决问题而撰写出来，解决问题时需要实现一些特定的运算法则。在 Strategy Pattern 之下，可以更换实现算法的部分而且不留痕迹。切换整个算法，简化改为采用其他方法来解决同样的问题，这就是 Strategy Pattern。

## 程序示例

请看下面应用 Strategy Pattern 的程序示例。这里的程序是计算机游戏“剪刀石头布”。

猜拳时的“策略”有 2 种方法。第一种方法有点笨，“猜赢之后继续出同样的招式”（WinningStrategy），第二种方法则是“从上一次出的招式，以概率分配方式求出下一个招式的机率”（ProbStrategy）。

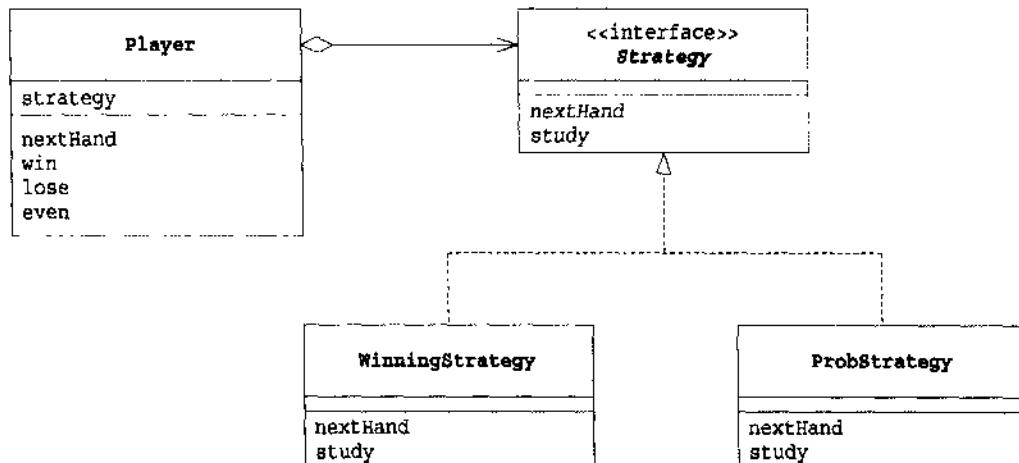


图 10-1 程序示例的类图

表 10-1 类一览表

| 名称       | 说明          |
|----------|-------------|
| Hand     | 表示猜拳“手势”的类  |
| Strategy | 表示猜拳“战略”的接口 |

## Strategy（策略）——把算法整个换掉

续上表

| 名称              | 说明                            |
|-----------------|-------------------------------|
| WinningStrategy | 表示猜赢之后继续出同样招式的战略的类            |
| ProbStrategy    | 表示从上一次出的招式，以概率分配方式求出下一个招式机率的类 |
| Player          | 表示玩猜拳的游戏者的类                   |
| Main            | 测试用的类                         |

**Hand 类**

Hand 类（程序 10-1）表示猜拳时“手势”的类。类内部以 0（石头）、1（剪刀）和 2（布）来表示。这部分存储在表示手势的值的字段（handvalue）里。

Hand 类的对象实例只会产生 3 个。一开始先产生 3 个对象实例，然后存储到数组 hand 内。

利用类方法 getHand 即可取得对象实例。若以参数传递手势的值，则其返回值为对象实例。这是 Singleton Pattern（第 5 章）的一种应用。

IsStrongerThan 和 isWeakerThan 则比较是哪个手势赢。例如，现在有 2 个手势 hand1 和 hand2，则写成

```
hand1.isStrongerThan(hand2)
和
```

```
hand1.isWeakerThan(hand2)
```

在此类内部实际判断手势谁赢的是 fight 方法。判断谁赢谁输时要使用到手势的值。

这里所使用的表达式看起来不太容易懂：

```
(this.handvalue + 1) % 3 == h.handvalue
```

它的意思是：若 this 手势的值加 1 等于 h 手势的值（即若 this 为石头，则 h 为剪刀；若 this 为剪刀，则 h 为布；若 this 为布，则 h 为石头），则 this 赢过 h。利用运算符%取 3 的余数是因为若布（2）加 1 时，要让结果变成石头（0）。

这个表达式特地写成 this.handvalue，其实只是要让它跟 h.handvalue 有比较明显的对比性。程序上写成像下面这样也无所谓：

```
(handvalue + 1) % 3 == h.handvalue
```

这里是从其他类（Player, WinningStrategy, ProbStrategy）来使用 Hand 类，Hand 类并不包含在 Strategy Pattern 的众多参与者之内。

**程序 10-1 Hand 类 (Hand.java)**

```
public class Hand {
    public static final int HANDVALUE_GUU = 0; // 表示石头的值
```

## 观察者模式

```

public static final int HANDVALUE_CHO = 1; // 表示剪刀的值
public static final int HANDVALUE_PAA = 2; // 表示布的值
public static final Hand[] hand = {           // 表示猜拳手势的 3 个对象
//象实例
    new Hand(HANDVALUE_GUU),
    new Hand(HANDVALUE_CHO),
    new Hand(HANDVALUE_PAA),
};
private static final String[] name = {         // 猜拳手势的字符串
    "石头", "剪刀", "布",
};
private int handvalue;                         // 猜拳手势的值
private Hand(int handvalue) {
    this.handvalue = handvalue;
}
public static Hand getHand(int handvalue) { // 从值取得对象实例
    return hand[handvalue];
}
public boolean isStrongerThan(Hand h) {        // 若 this 赢过 h, 则为
//true
    return fight(h) == 1;
}
public boolean isWeakerThan(Hand h) {          // 若 this 输给 h,
//则为 true
    return fight(h) == -1;
}

private int fight(Hand h) { // 平手为 0, 若 this 胜出为 1, 若
//h 胜出则为 -1
    if (this == h) {
        return 0;
    } else if ((this.handvalue + 1) % 3 == h.handvalue) {
        return 1;
    } else {
        return -1;
    }
}
public String toString() {                      // 转换成字符串
}

```

**Strategy (策略) —— 把算法整个换掉**

```

    return name[handvalue];
}
}

```

---

## Strategy 接口

Strategy 接口（程序 10-2）是抓出所有猜拳“策略”的抽象方法。

`nextHand` 是“取得下一次出的招式”的方法。只要调用出这个方法，实现 Strategy 接口的类就要动动脑决定“下一次的招式”。

`study` 是学习“前一次出的手势有没有赢”的方法。如果前一次调用的 `nextHand` 方法赢了的话，就调用 `study(true)`。如果输了就调用 `study(false)`。如此一来，实现 Strategy 接口的类就会修改自己的内部状态，作为决定接下来 `nextHand` 方法的返回值的参考。

**程序 10-2 Strategy 类 (Strategy.java)**

```

public interface Strategy {
    public abstract Hand nextHand();
    public abstract void study(boolean win);
}

```

---

## WinningStrategy 类

WinningStrategy 类（程序 10-3）是实现 Strategy 接口的类之一。实现 Strategy 接口是指实现 `nextHand` 和 `study` 这 2 个方法。

这个类采取的策略比较笨，如果上一次赢了的话，接下来就继续出同样的手势（石头→石头、布→布）。如果上一次输了的话，接下来的手势就要利用随机数决定。

`random` 字段是存储 `java.util.Random` 的对象实例，当此类需要使用随机数时就要用到。简单讲，很像是这个类的随机数产生器。

`won` 字段存储上一次的比拳结果。如果赢了就是 `true`、输了就是 `false`。

`prevHand` 字段存储上一次猜拳时所出的手势。

**程序 10-3 WinningStrategy 类 (WinningStrategy.java)**

```

import java.util.Random;

public class WinningStrategy implements Strategy {
    private Random random;
    private boolean won = false;
    private Hand prevHand;
    public WinningStrategy(int seed) {

```

```

        random = new Random(seed);
    }

    public Hand nextHand() {
        if (!won) {
            prevHand = Hand.getHand(random.nextInt(3));
        }
        return prevHand;
    }

    public void study(boolean win) {
        won = win;
    }
}

```

---

## ProbStrategy 类

ProbStrategy 类（程序 10-4）则是比较象样的“策略”，稍微用一点脑筋。虽然下一次的手势都是由随机数决定，不过它会参考之前的输赢记录，机动性更改手势的出现机率。

history 字段是一份列出过去输赢记录的表格，用来计算机率。history 是一个 int 类型的 2 维数组，各该下标的代表意义如下：

history[上一次的手势][这次的手势]

这个表达式的值越大，则代表之前获胜的机率越高。用实际的数据可能会更清楚一点：

|               |                |
|---------------|----------------|
| history[0][0] | 石头、之前出石头时的获胜次数 |
| history[0][1] | 石头、之前出剪刀时的获胜次数 |
| history[0][2] | 石头、之前出布时的获胜次数  |

假设上一次是出石头。这时候就会利用前面的 history[0][0], history[0][1], history[0][2] 之值以概率的方式求出这次应该出哪个手势。换句话说，就是把这两个表达式的值加起来（getSum 方法），求出一个 0 到该数值之间的随机数，然后再根据随机数决定下一个手势（nextHand 方法）。

假设，

|                     |
|---------------------|
| history[0][0] 的值为 3 |
| history[0][1] 的值为 5 |
| history[0][2] 的值为 7 |

此时，就以出石头、剪刀、布为 3: 5: 7 的比例来决定下一个手势。所得随机数值为大于 0、小于 15 (15 是 3+5+7 的和)，故

大于 0、小于 3，出石头

## Strategy (策略) —— 把算法整个换掉

大于 3、小于 8 ( $=3+5$ )，出剪刀

大于 8、小于 15 ( $=3+5+7$ )，则出布

study 方法根据 nextHand 方法所返回的手势输赢，随时更新 history 字段的内容。这个作战策略的前提是对手的出拳方式也有另一种 Pattern。

程序 10-4 ProbStrategy 类 (ProbStrategy.java)

```
import java.util.Random;

public class ProbStrategy implements Strategy {
    private Random random;
    private int prevHandValue = 0;
    private int currentHandValue = 0;
    private int[][] history = {
        { 1, 1, 1, },
        { 1, 1, 1, },
        { 1, 1, 1, },
    };
    public ProbStrategy(int seed) {
        random = new Random(seed);
    }
    public Hand nextHand() {
        int bet = random.nextInt(getSum(currentHandValue));
        int handvalue = 0;
        if (bet < history[currentHandValue][0]) {
            handvalue = 0;
        }
        else if (bet < history[currentHandValue][0] + history
            [currentHandValue][1]) {
            handvalue = 1;
        }
        else {
            handvalue = 2;
        }
        prevHandValue = currentHandValue;
        currentHandValue = handvalue;
        return Hand.getHand(handvalue);
    }
    private int getSum(int hv) {
        int sum = 0;
```

## 策略模式的应用

```

        for (int i = 0; i < 3; i++) {
            sum += history[hv][i];
        }
        return sum;
    }

    public void study(boolean win) {
        if (win) {
            history[prevHandValue][currentHandValue]++;
        } else {
            history[prevHandValue][(currentHandValue + 1) % 3]++;
            history[prevHandValue][(currentHandValue + 2) % 3]++;
        }
    }
}

```

---

**Player 类**

Player 类（程序 10-5）是表示玩猜拳的游戏者的类。

Player 类取得“姓名”和“战略”后产生对象实例。

nextHand 方法取得下一次手势，但实际上决定这个手势的还是自己本身的“战略”。战略的 nextHand 方法的返回值就是 Player 的 nextHand 方法的返回值。nextHand 方法把自己应该进行的处理交给 Strategy，两者间是“委托”的关系。

因为这里要让猜赢（win）、猜输（lose）或平手（even）的胜负结果影响到下一次的猜拳手势，所以 Player 类要经由 strategy 字段调用 study 方法。这样才能使用 study 方法来更改战略的内部状态。wincount、losecount、gamecount 则是记录猜拳者的获胜、失败和游戏点次数。

**程序 10-5 Player 类（Player.java）**

```

public class Player {
    private String name;
    private Strategy strategy;
    private int wincount;
    private int losecount;
    private int gamecount;
    public Player(String name, Strategy strategy) { // 得到姓名和战略
        this.name = name;
        this.strategy = strategy;
    }
}

```

## Strategy (策略) —— 把算法整个换掉

```

    }

    public Hand nextHand() {                                // 向战略请示手势
        return strategy.nextHand();
    }

    public void win() {                                    // 猜赢
        strategy.study(true);
        wincount++;
        gamecount++;
    }

    public void lose() {                                 // 猜输
        strategy.study(false);
        losecount++;
        gamecount++;
    }

    public void even() {                               // 平手
        gamecount++;
    }

    public String toString() {
        return "[" + name + ":" + gamecount + " games, " + wincount
+ " win, " + losecount + " lose" + "]";
    }
}

```

---

**Main 类**

Main 类（程序 10-6）则是利用以上类实际在计算机上进行猜拳的类，这里是一个只有 2 个人的猜拳比赛。

- 姓名: "Taro"、战略: WinningStrategy
- 姓名: "Hana"、战略: ProbStrategy

要他们猜拳 10000 次后，输出比赛结果。

另外，

"Winner:" + player1

这部分跟下面的写法同义：

"Winner:" + player1.toString()

**程序 10-6 Main 类 (Main.java)**


---

```

public class Main {
    public static void main(String[] args) {

```

## Java语言的初步应用

```

        if (args.length != 2) {
            System.out.println("Usage:    java    Main    randomseed1
randomseed2");
            System.out.println("Example: java Main 314 15");
            System.exit(0);
        }
        int seed1 = Integer.parseInt(args[0]);
        int seed2 = Integer.parseInt(args[1]);
        Player player1 = new Player("Taro", new WinningStrategy
(seed1));
        Player player2 = new Player("Hana", new ProbStrategy
(seed2));
        for (int i = 0; i < 10000; i++) {
            Hand nextHand1 = player1.nextHand();
            Hand nextHand2 = player2.nextHand();
            if (nextHand1.isStrongerThan(nextHand2)) {
                System.out.println("Winner:" + player1);
                player1.win();
                player2.lose();
            }
            else if (nextHand2.isStrongerThan(nextHand1)) {
                System.out.println("Winner:" + player2);
                player1.lose();
                player2.win();
            }
            else {
                System.out.println("Even...");
                player1.even();
                player2.even();
            }
        }
        System.out.println("Total result:");
        System.out.println("'" + player1);
        System.out.println("'" + player2);
    }
}

```

|                                      |         |
|--------------------------------------|---------|
| java Main 314 15                     |         |
| Even...                              | ←平手     |
| Winner:[Hana:1 games, 0 win, 0 lose] | ←Hana ★ |
| Winner:[Taro:2 games, 0 win, 1 lose] | ←Taro ★ |

## Strategy (策略) —— 把算法整个换掉

```

Even...                                ←平手
Winner:[Hana:4 games, 1 win, 1 lose]   ←Hana 赢
Winner:[Taro:5 games, 1 win, 2 lose]   ←Taro 赢
Even...                                ←平手
Even...                                ←平手
Even...                                ←平手
Winner:[Taro:8 games, 2 win, 2 lose]   ←Hana 赢
Winner:[Taro:9 games, 3 win, 2 lose]   ←Taro 赢
Winner:[Taro:10 games, 4 win, 2 lose]  ←Taro 赢
Even...                                ←平手
~(中间省略)~
Even...                                ←平手
Winner:[Taro:9992 games, 3164 win, 3488 lose] ←Taro 赢
Winner:[Hana:9993 games, 3488 win, 3165 lose]   ←Hana 赢
Winner:[Taro:9994 games, 3165 win, 3489 lose]   ←Taro 赢
Winner:[Taro:9995 games, 3166 win, 3489 lose]   ←Taro 赢
Winner:[Hana:9996 games, 3489 win, 3167 lose]   ←Hana 赢
Even...                                ←平手
Even...                                ←平手
Even...                                ←平手
Total result:
[Taro:10000 games, 3167 win, 3490 lose]    ←Taro 赢 3167 次、输 3490 次
[Hana:10000 games, 3490 win, 3167 lose]    ←Hana 赢 3490 次、输 3167 次

```

图 10-2 执行结果

## Strategy Pattern 的所有参与者

Strategy Pattern 中出现的所有参与者可整理如下。

### ◆ Strategy (策略) 参与者

规定使用策略的接口 (API) 的参与者。在前面的程序示例中，扮演这个参与者的是 Strategy。

### ◆ ConcreteStrategy (具体策略) 参与者

实际上实际 Strategy 参与者的接口 (API) 的参与者。这里会实际编写有关具体战  
略 (作战、策略、方法或算法) 的程序。在前面的程序示例中，扮演这个参与者的是  
WinningStrategy 类和 ProbStrategy 类。

### ◆ Context (上下文) 参与者

利用 Strategy 参与者的参与者。它有 ConcreteStrategy 参与者的对象实例，如有必要时则可使用（最多只能调用 Strategy 参与者的接口（API））。在前面的程序示例中，扮演这个参与者的是 Player 类。

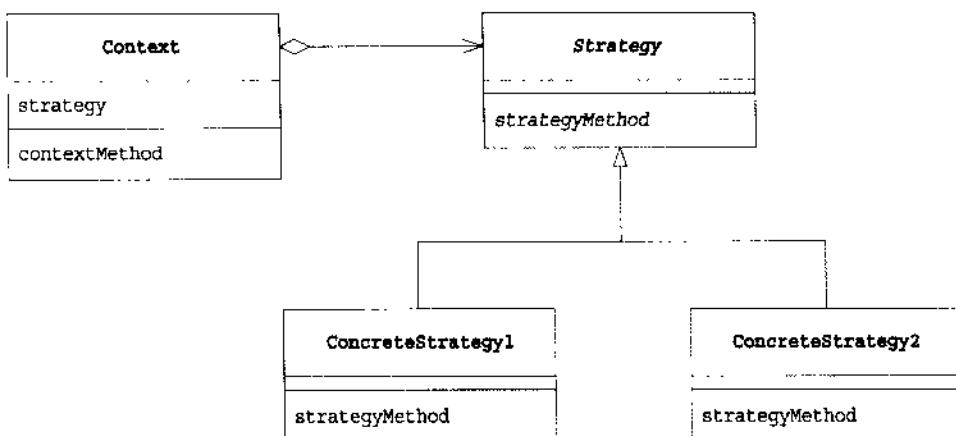


图 10-3 Strategy Pattern 的类图

## 扩展自我视野的提示

### 为什么要特地建立 Strategy 参与者

一般在设计程序时，比较习惯把实现算法这部分结合到方法里面。但是 Strategy Pattern 则故意把算法的部分跟其他部分分开。只规定跟算法有关的接口（API）部分，然后再从程序这边以委托的方式来利用运算法则。

这样看起来好像让整个程序变得很复杂，其实并不尽然。假设现在我们想改良算法提高速度，如果使用 Strategy Pattern 的话，只要注意不要去修改到 Strategy 参与者的接口（API），然后再改 ConcreteStrategy 参与者就行。而且它所使用的委托又是说分就分的关系，所以很容易就可以切换运算法则。举例来说，比较原始运算法则跟改良后的运算法则两者的速度时，只要动手切换一下就能测得出来。

即使这个程序是一个象棋游戏，只要使用 Strategy Pattern，还能配合用户选择的参与者切换思考模式等级。

### 执行时也可切换

利用 Strategy Pattern，还可在程序执行中直接切换 ConcreteStrategy 参与者。例如

## Strategy (策略) —— 把算法整个换掉

若计算机的内存稍嫌不足，就使用 SlowButLessMemoryStrategy（速度稍慢，但可节省内存），若有足够的内存则使用 FastButMoreMemoryStrategy（速度较快，但很占用内存）。

当然也可以用甲方的算法来“验算”乙方的算法。假设现在想用一个电子表格软件的测试版，进行高复杂度的计算。此时，只要手边同时有“可能还有 bug 的高速算法”和“可保证计算正确的低速算法”，就可能利用后者去验算前者。

## 相关 Pattern

### ◆ Flyweight Pattern (第 20 章)

ConcreteStrategy 参与者有时会利用 Flyweight Pattern，让一个以上的位置能共享。

### ◆ Abstract Factory Pattern (第 8 章)

在 Strategy Pattern 可以彻底切换算法而不留痕迹。

而 Abstract Factory Pattern 则可完整切换具体工厂、零件或产品。

### ◆ State Pattern (第 19 章)

Strategy Pattern 和 State Pattern 两者都是切换委托对象的 Pattern，类间的关系也很类似。但两 Pattern 的目的并不相同。

Strategy Pattern 是建立表示“算法”的类，把它设为 Concrete Strategy 参与者。在 Strategy Pattern 可以切换不同的类，不过如果没有必要的话，不切换也没关系。

而 State Pattern 则是建立表示“状态”的类，把它设为 Concrete State 参与者。State Pattern 一遇到状态发生变化时，就一定会切换委托对象的类。

## 重点回顾

本章所介绍的是可简单切换算法的 Strategy Pattern。因为有了委托关系，所以才能切换算法，尤其是机动性的切换动作。

## 练习题

解答请见附录 A

### 问题 1

请建立一个随机决定下一个猜拳手式的 RandomStrategy 类。

### 问题 2

在本章的程序示例中，Hand 类（程序 10-1）的 fight 方法在判断平手并没有用第

**设计语言中的应用**

一个表达式，而是第二个表达式：

```
this.handvalue == h.handvalue  
this == h
```

为什么这个写法不会有问题呢？

**问题 3**

假设现在某甲在编写 WinningStrategy 类（程序 10-3）的程序时，won 字段没有写成

```
private boolean won = false;
```

而写成

```
private boolean won;
```

但是这样的结果跟有= false 的时候的操作也没什么两样，为什么？

**问题 4**

以下程序 10-7 到程序 10-10 的程序是进行排序的类与接口。这个程序的执行结果如图 10-4 所示。这个程序使用的算法是插入排序法（insertion sort）。请配合 Sorter 的接口，建立一个可表示其他算法（无限制）的类。

**程序 10-7 Sorter 接口（Sorter.java）**


---

```
import java.lang.Comparator;  
  
public interface Sorter {  
    public abstract void sort(Comparable[] data);  
}
```

---

**程序 10-8 InsertSorter 类（InsertSorter.java）**


---

```
public class InsertSorter implements Sorter {  
    public void sort(Comparable[] data) {  
        for (int i = 0; i < data.length - 1; i++) {  
            for (int j = i + 1; j < data.length; j++) {  
                if (data[i].compareTo(data[j]) > 0) {  
                    Comparable passingplace = data[i];  
                    data[i] = data[j];  
                    data[j] = passingplace;  
                }  
            }  
        }  
    }  
}
```

---

## Strategy(策略)——把算法整个换掉

## 程序 10-9 SorterAndPrint 类 (SorterAndPrint.java)

```

public class SortAndPrint {
    Comparable[] data;
    Sorter sorter;
    public SortAndPrint(Comparable[] data, Sorter sorter) {
        this.data = data;
        this.sorter = sorter;
    }
    public void execute() {
        print();
        sorter.sort(data);
        print();
    }
    public void print() {
        for (int i = 0; i < data.length; i++) {
            System.out.print(data[i] + ", ");
        }
        System.out.println("");
    }
}

```

## 程序 10-10 Main 类 (Main.java)

```

public class Main {
    public static void main(String[] args) {
        String[] data = {
            "Dumpty", "Bowman", "Carroll", "Elfland", "Alice",
        };
        SortAndPrint sap = new SortAndPrint(data, new Insert
Sorter());
        sap.execute();
    }
}

```

插入排序的实现

|                                         |      |
|-----------------------------------------|------|
| Dumpty, Bowman, Carroll, Elfland, Alice | ←排序前 |
| Alice, Bowman, Carroll, Dumpty, Elfland | ←排序后 |

图 10-4 执行结果

# 第 11 章

## Composite (组成) ——对容器和内容 — 视同仁

## Composite Pattern

在计算机的文件系统中有“目录”（有些 OS 称为文件夹）。目录之下有文件或者其他目录（子目录）；在子目录下还会有其他文件或子目录。目录就是类似这样“画中有画”的结构和递归结构。

虽然目录和文件实际上是两种截然不同的东西，但两者都可放在目录底下。目录和文件两者合起来也可称为“目录进入点”。以目录进入点这个名称来说，目录和文件被视为同一种类（一视同仁）。

举个例子来说，假设现在想依次检查某目录底下有哪些东西。按照顺序排列起来，第一个得到的可能是子目录、也可能是文件。简单一句话，就是依次来看“目录进入点”而已。

像这样把目录和文件兜起来变成一个目录进入点一样，有时候把容器和内容当作是同类来处理会比较好下手。容器里面可以是内容，也可以是更小一号的容器；而这个小一号的容器里还可以再放更小一号的容器，可以建立出像这样大套小的结构和递归结构。

### 本章要学习的 Composite Pattern

就是建立类似上述的结构。对容器和内容一视同仁，建立递归结构的 Design Pattern 就是 Composite Pattern。Composite 的英文原意是“混合物”“复合物”的意思。

## 程序示例

程序示例在此用一个以模式方式表示文件和目录的程序来当作 Composite Pattern 的程序示例。表示文件的类是 File 类、表示目录的类是 Directory 类，两者合并起来就是父类 Entry 类。Entry 类是表示目录进入点的类，对 File 和 Directory 一视同仁的类。

表 11-1 类一览表

| 名称                     | 说明                          |
|------------------------|-----------------------------|
| Entry                  | 对 File 和 Directory 一视同仁的抽象类 |
| File                   | 表示文件的类                      |
| Directory              | 表示目录的类                      |
| FileTreatmentException | 欲在文件内新增 Entry 时所发生的异常类      |
| Main                   | 测试用的类                       |

## Composite (组成) —— 对容器和内容一视同仁

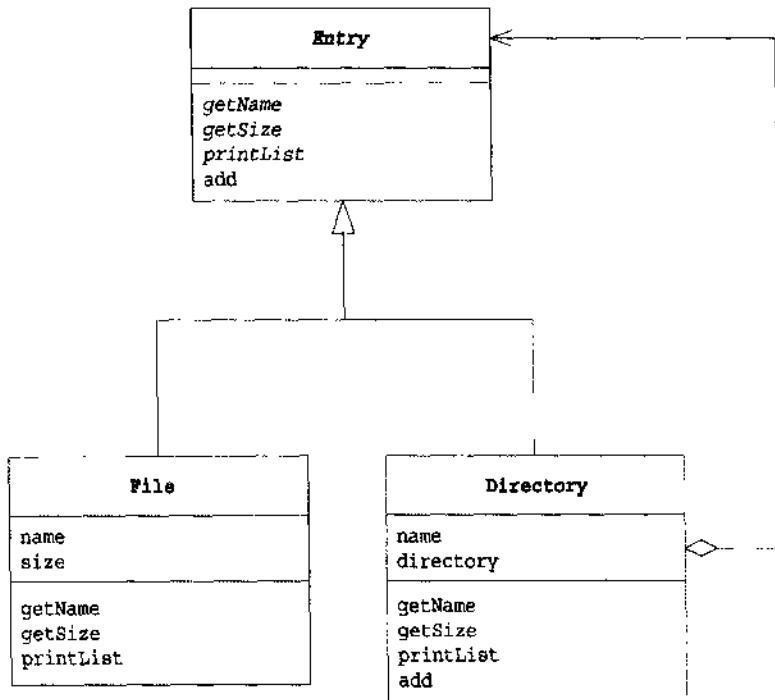


图 11-1 程序示例的类图

## Entry 类

Entry 类（程序 11-1）是抽象类，表示目录进入点。子类有 File 类和 Directory 类。

目录进入点有一个名称，可以用 getName 方法来取得名称。实现部分则交给子类（*subclass responsibility*）。

此外，假设目录进入点有容量大小。可以用 getSize 方法来取得这个容量。这部分的实现也是交给子类。

把文件和目录（即目录进入点）放入目录底下的方法是 add。不过实现 add 的是表示目录的子类——Directory 类的层次。如果在 Entry 类的层次要抛出异常视为程序错误。这里的程序示例设计成会抛出异常，但是 add 的实现方法还有其他很多种变形，详细说明请参见本章的“扩展自我视野的提示”一节。

printList 方法是输出“总览”的方法，有不带参数的 printList() 和带参数的 printList(String) 两种方法。这称为 printList 方法的 overload（重载）。根据调用时的参数类型，从有 overload 的多个方法中选择并执行最适当的方法。在此，printList() 公开为 public，printList(String) 则为 protected，仅可在 Entry 的子类使用。

## ·如何提高编程效率

`toString` 方法是定义对象实例的标准字符串输出格式，这里采用文件名跟文件大小并列的输出格式。`getName` 和 `getSize` 是抽象方法，不过子类是要等待它们被实现，才从 `toString` 把它们调用出来（Template Method Pattern）。

程序 11-1 Entry 类 (Entry.java)

```
public abstract class Entry {
    public abstract String getName(); // 取得文件名
    public abstract int getSize(); // 取得文件容量
    public Entry add(Entry entry) throws FileTreatmentException {
        //新增进入点
        throw new FileTreatmentException();
    }
    public void printList() { // 输出总览
        printList("");
    }
    protected abstract void printList(String prefix); // 在前面加
    //上prefix后输出总览
    public String toString() { // 字符串输出格式
        return getName() + " (" + getSize() + ")";
    }
}
```

## File 类

`File` 类（程序 11-2）是表示文件的类。声明为 `Entry` 类的子类。

字段有 2 个，分别是表示文件名 `name` 和文件容量 `size`。

以构造函数传递文件名和文件大小后，产生 `File` 的对象实例。例如写成：

```
new File("readme.txt", 1000)
```

表示有一个文件名为 `readme.txt`，文件大小为 1000 的文件。当然，这只是虚拟文件，并不是在实际的文件系统上建立新文件。

`getName` 方法和 `getSize` 方法分别是返回文件名和文件大小的方法。

从父类继承而来的 `printList(String)` 就在这里进行实现。`printList(String)` 在输出 `prefix` 和本身字符串时，会在中间加上“`/`”。这里有一个“`/` + `this`”的运算动作，但是只要加上字符串和对象之后，就能自动的调用该对象的 `toString` 方法。这是 Java 语言的规格。换句话说，下面几个表达式的意思都是一样。

```
prefix + "/" + this
prefix + "/" + this.toString()
prefix + "/" + toString()
```

## Composite (组成) ——对容器和内容一视同仁

在父类 Entry 设为 abstract 的方法都会在此进行实现，所以 File 类并不是抽象类。

**程序 11-2 File 类 (File.java)**

```
public class File extends Entry {
    private String name;
    private int size;
    public File(String name, int size) {
        this.name = name;
        this.size = size;
    }
    public String getName() {
        return name;
    }
    public int getSize() {
        return size;
    }
    protected void printList(String prefix) {
        System.out.println(prefix + "/" + this);
    }
}
```

## Directory 类

Directory 类（程序 11-3）是表示目录的类。它也是定义为 Entry 类的子类。

字段有 2 个，第 1 个字段 name 是目录名称，跟 File 类一样。不过 Directory 并没有表示容量大小的字段，因为目录容量的大小要动态求出。

另一个字段 directory 则定义为 Vector。这个 directory 是存储目录进入点的字段。

getName 方法只有返回 name 字段而已，而 getSize 方法则执行计算。逐一取出 directory 的元素并合计所求出的总容量大小，就是 getSize 方法的返回值。请注意其中的这个叙述：

```
size += entry.getSize();
```

entry 的容量大小会加到变量 size，但这个 entry 可能是 File 的对象实例或 Directory 的对象实例，并不一定。无论是哪一边的对象实例，都可以用同一个方法 getSize 求出容量大小，这就是 Composite Pattern 的特征——“对容器与内容一视同仁”的精髓所在。不管是 File 的对象实例也好、Directory 的对象实例也罢，因为两者都是 Entry 的子类的对象实例，所以 entry 可以放心的调用 getSize。即使以后又增加更多个 Entry 的子类，因为它实现 getSize 方法，所以没有要修改 Directory 类这部分的问题。

如果 entry 是 Directory 的对象实例，则一开始评估表达式 entry.getSize 时，就会把目录里所有插入点的容量一个个加进去。如果刚好该目录底下还有目录时，就继续调用子目录的 getSize，以递归的方式调用 getSize 方法。相信各位已经明白，Composite Pattern 的递归结构就是直接对应到 getSize 方法的递归调用。

add 方法则是把文件和目录新增到目录底下。被当作参数传递出来的 entry 不是为了检查到底是 Directory 类的对象实例、还是 File 类的对象实例，而是要新增到字段 directory：

```
directory.add(entry);
```

“新增”这个动作其实也可以委托给 Vector。

printList 方法输出目录总览。printList 方法也跟 getSize 方法一样会以递归的方式调用 printList。此时，当然也和 getSize 方法的时候一样，不会去检查变量 entry 是 File 的对象实例、还是 Directory 的对象实例。因为容器和内容已经是一视同仁。

### 程序 11-3 Directory 类 (Directory.java)

```
import java.util.Iterator;
import java.util.Vector;

public class Directory extends Entry {
    private String name; // 目录名称
    private Vector directory = new Vector(); // 目录进入点的集合
    public Directory(String name) { // 构造函数
        this.name = name;
    }
    public String getName() { // 取得目录名称
        return name;
    }
    public int getSize() { // 取得目录容量
        int size = 0;
        Iterator it = directory.iterator();
        while (it.hasNext()) {
            Entry entry = (Entry)it.next();
            size += entry.getSize();
        }
        return size;
    }
    public Entry add(Entry entry) { // 新增进入点
        directory.add(entry);
        return this;
    }
}
```

## Composite (组成) —— 对容器和内容一视同仁

```

    }

protected void printList(String prefix) { // 进入点的总览
    System.out.println(prefix + "/" + this);
    Iterator it = directory.iterator();
    while (it.hasNext()) {
        Entry entry = (Entry)it.next();
        entry.printList(prefix + "/" + name);
    }
}
}

```

---

**FileTreatmentException 类**

FileTreatmentException 类（程序 11-4）是对文件调用 add 方法时所抛出的异常。这个异常并不在 Java 的类函数库之内，而是为了这个程序示例而特别定义的类。

**程序 11-4 FileTreatmentException 类（FileTreatmentException.java）**

```

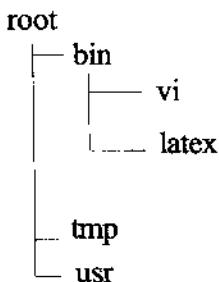
public class FileTreatmentException extends RuntimeException {
    public FileTreatmentException() {
    }
    public FileTreatmentException(String msg) {
        super(msg);
    }
}

```

---

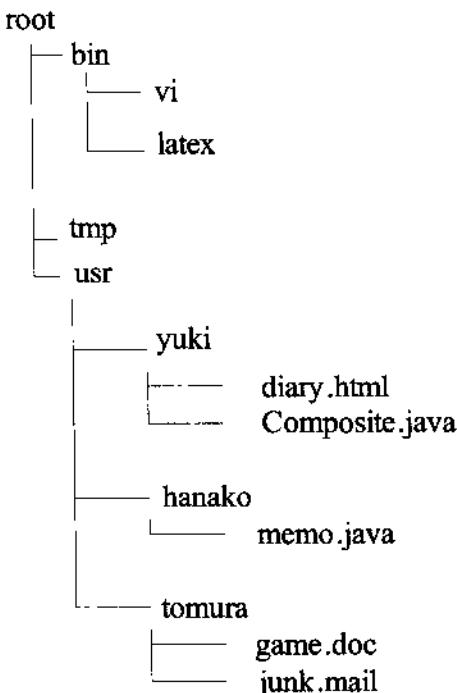
**Main 类**

Main 类（程序 11-5）则建立如下的目录层次。首先要建立 root, bin, tmp, usr 等目录，然后把 vi 文件和 latex 文件放到 bin 底下。



接着在 usr 目录底下另建 yuki, hanako, tomura 等目录，把每个人的文件建立在该

目录之下。



执行结果即如图 11-2 所示。请注意：当每个人的文件完成之后，root 目录的容量就会变得很大。

**程序 11-5 Main 类 (Main.java)**

---

```

public class Main {
    public static void main(String[] args) {
        try {
            System.out.println("Making root entries...");
            Directory rootdir = new Directory("root");
            Directory bindir = new Directory("bin");
            Directory tmpdir = new Directory("tmp");
            Directory usrdir = new Directory("usr");
            rootdir.add(bindir);
            rootdir.add(tmpdir);
            rootdir.add(usrdir);
            bindir.add(new File("vi", 10000));
            bindir.add(new File("latex", 20000));
            rootdir.printList();
        }
    }
}
  
```

## Composite (组成) ——对容器和内容一视同仁

```

System.out.println("");
System.out.println("Making user entries...");
Directory yuki = new Directory("yuki");
Directory hanako = new Directory("hanako");
Directory tomura = new Directory("tomura");
usrdir.add(yuki);
usrdir.add(hanako);
usrdir.add(tomura);
yuki.add(new File("diary.html", 100));
yuki.add(new File("Composite.java", 200));
hanako.add(new File("memo.tex", 300));
tomura.add(new File("game.doc", 400));
tomura.add(new File("junk.mail", 500));
rootdir.printList();
} catch (FileTreatmentException e) {
    e.printStackTrace();
}
}

```

---

```

Making root entries...
/root (30000)
/root/bin (30000)
/root/bin/vi (10000)
/root/bin/latex (20000)
/root/tmp (0)
/root/usr (0)

Making user entries...
/root (31500)
/root/bin (30000)
/root/bin/vi (10000)
/root/bin/latex (20000)
/root/tmp (0)
/root/usr (1500)
/root/usr/yuki (300)
/root/usr/yuki/diary.html (100)
/root/usr/yuki/Composite.java (200)
/root/usr/hanako (300)
/root/usr/hanako/memo.tex (300)
/root/usr/tomura (900)
/root/usr/tomura/game.doc (400)
/root/usr/tomura/junk.mail (500)

```

图 11-2 执行结果

## Composite Pattern 的所有参与者

Composite Pattern 中出现的所有参与者可整理如下。

### ◆ Leaf (叶子) 参与者

表示“内容”的参与者。这个参与者不可放入其他东西。在前面的程序示例中，扮演这个角色的是 File 类。

### ◆ Composite (复合体) 参与者

表示“容器”的参与者。可放入 Leaf 参与者或 Composite 参与者。在前面的程序示例中，扮演这个角色的是 Directory 类。

### ◆ Component 参与者

对 Leaf 参与者和 Composite 参与者一视同仁的参与者。Component 参与者是 Leaf 参与者和 Composite 参与者共享的父类。在前面的程序示例中，扮演这个角色的是 Entry 类。

### ◆ Client (客户) 参与者

利用 Composite Pattern 的人。在前面的程序示例中，扮演这个角色的是 Main 类。

Composite Pattern 的类图即如图 11-3 所示。图中，包含在 Composite 参与者里面的 Component 参与者（即 Leaf 参与者或 Composite 参与者）就像在父母亲保护伞底下的“子女”。getChild 方法则是从 Component 参与者取得“子女”的方法。

## 扩展自我视野的提示

### 单、复数的一视同仁

Composite Pattern 对容器和内容一视同仁，这也可以称为单、复数的一视同仁。也就是说：把复数个物体集中在一起，可像是在处理 1 个整体。

就以程序的行为测试为例，假设 Test1 是键盘输入的输入测试，Test2 是文件导入的输入测试，Test3 则是网络导入的输入测试。如果现在想把 Test1, Test2, Test3 这三个整合成一个“输入测试”，利用 Composite Pattern 就能办到。先把复数个测试集中起来设为“输入测试”，再建立一个包含其他测试的“输出测试”，最后两者合起来便成为“输出 / 输入测试”。

下页所示案例的测试程序就利用了 Composite Pattern。

## Composite (组成) —— 对容器和内容一视同仁

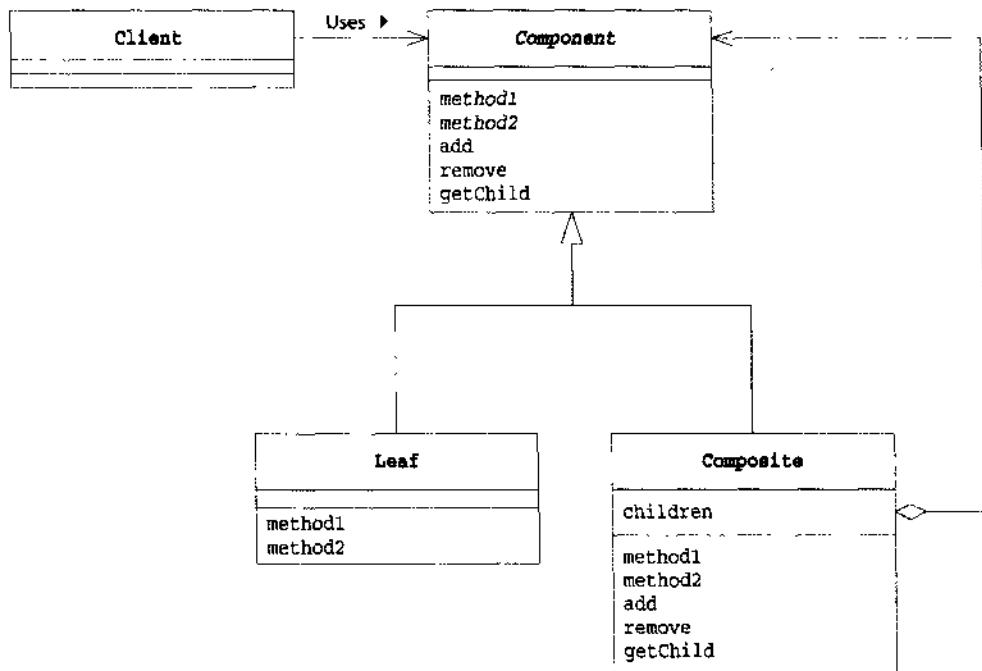


图 11-3 Composite Pattern 的类图

## add 放在哪里

本章的程序示例是在 Entry 类定义 add 方法，然后再抛出异常。实际上能使用 add 方法的也只有 Directory 类而已。add 方法的摆放、实现会有几种可能：

### ◆ 做法 1：在 Entry 类进行实现、设置程序错误（error）

把 add 方法放在 Entry 类进行实现、设为程序错误正是本章程序示例的做法。在实际能使用 add 方法的 Directory 类重新写入 Entry 类的 add，置换成有意义的实现。

由于 File 类从 Entry 类继承 add 方法，所以可以做 add，但会抛出异常。

### ◆ 做法 2：在 Entry 类进行实现，没有任何行为

add 方法是在 Entry 类进行实现，但也可以不设为程序错误（即什么都不执行）。

### ◆ 做法 3：在 Entry 类会有声明，但不实现

在 Entry 类把 add 方法设为抽象方法，如果有必要则另在子类加以定义，无此必要则设为程序错误，这也是一种做法。采用这种做法的好处是可以统一处理，而且没有必要时的动作可以在子类再决定。不过，如此一来就必须在 File 这边定义原本已经不要的 add（或可能是 remove、getChild 等）。

### ◆ 做法 4：只放在 Directory 类

这种做法是不把 add 方法放在 Entry 类，只有真正有必要的 Directory 类才放。但是采取这种做法时，如果 add 到 Entry 类型变量（其实际内容为 Directory 的对象实例），就必须一个个强迫转型成 Directory 类型。

## 递归结构随时都会出现

在本章的程序示例中，提到目录进入点的时候已经说过：一旦开始设计程序，很多情形下都会遇到递归结构和 Composite Pattern。以窗口系统为例，在已开启的窗口中还可再开启另一个子窗口，这就是典型的 Composite Pattern。项目下还可继续建立其他项目，也算是递归结构之一。当你在设计一个整合所有给计算机的 command（命令）的宏命令时，如果把宏命令设计成递归结构，甚至还可以写出宏命令中的宏命令。一般来说，本身是树状结构的数据结构就相当于 Composite Pattern。

## 相关 Pattern

### ◆ Command Pattern（第 22 章）

在 Command Pattern 建立“宏命令”时，会使用到 Composite Pattern。

### ◆ Visitor Pattern（第 13 章）

Visitor Pattern 是在绕着 Composite 进行处理时使用。

### ◆ Decorator Pattern（第 12 章）

Composite Pattern 对容器（Composite 参与者）和内容（Leaf 参与者）一视同仁，两者都是 Component 参与者。

Decorator Pattern 则是对装饰和内容一视同仁。

## 重点回顾

本章所介绍的是对容器和内容一视同仁、并建立递归结构的 Composite Pattern。

## 练习题

解答请见附录 A

### 问题 1

请提出除了文件系统以外，还有哪些例子相当于 Composite Pattern？

## Composite (组成) —— 对容器和内容一视同仁

### 问题 2

如果现在想在程序示例中新增一个能从 Entry 的对象实例（其子类的对象实例）取得“全路径（full path）”的功能，例如想从 File 的对象实例取得以下字符串：

"/root/usr/yuki/Composite.java"

此时，应该修改程序示例的哪个类？又该如何修改？

# 第 12 章

## Decorator（装饰） ——对装饰和内容 一视同仁

## Decorator Pattern

假设现在有一块蛋糕，如果涂上鲜奶油，就变成一块鲜奶油蛋糕。上面加上草莓，又变成草莓奶油蛋糕；再加上巧克力片、用白巧克力写成名字、插上相当于年龄数的蜡烛，又摇身一变成为生日蛋糕。

蛋糕、鲜奶油蛋糕、草莓奶油蛋糕或生日蛋糕的原始类型其实都是蛋糕，只不过运用了各种装饰手法（涂鲜奶油、加上草莓等），让这块海绵蛋糕变得更诱人、能符合不同目的而改变的蛋糕。

对象也很像是蛋糕的多种变化。首先建立一个像是蛋糕的核心对象，再一层层加上装饰用的功能，就可以完成符合所需的对象。

像这样在对象上不断披上 decoration（装饰）的 Design Pattern 就称为：

**Decorator Pattern**

decorator 是“进行 decorate（装饰）的主体”的意思。本章要学习的就是 Decorator Pattern。

## 程序示例

本章的程序示例是在字符串四周加上装饰外框后再打印出来的程序。装饰外框是指以-，+，|等字符组成的框线。图 12-1 为其中一例。



图 12-1 Hello, world. 加上装饰外框后的示例

表 12-1 类一览表

| 名称            | 说明              |
|---------------|-----------------|
| Display       | 打印字符串用的抽象类      |
| StringDisplay | 只有 1 行的打印字符串用的类 |
| Border        | 表示“装饰外框”的抽象类    |
| SideBorder    | 只在左右加上装饰外框的类    |
| FullBorder    | 在上下左右加上装饰外框的类   |
| Main          | 测试用的类           |

## Decorator (装饰) —— 对装饰和内容一视同仁

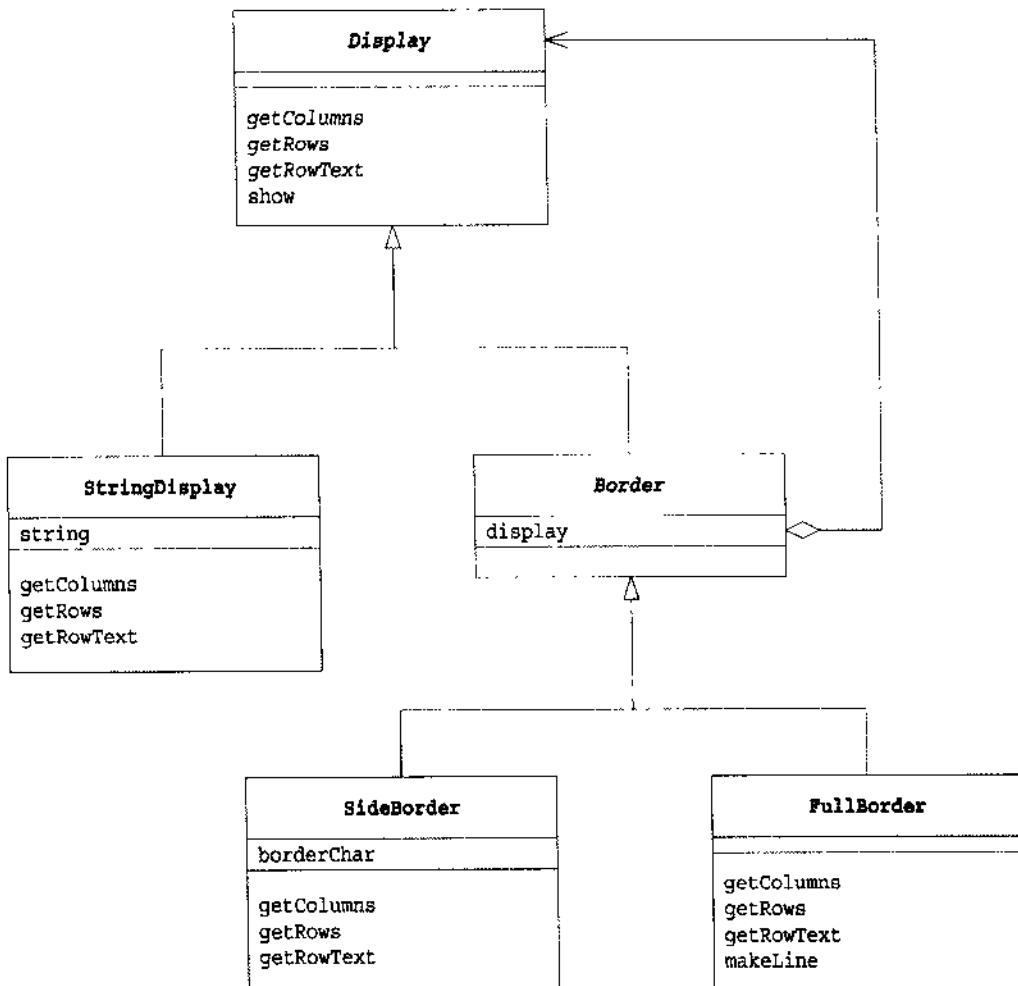


图 12-2 程序示例的类图

**Display** 类

**Display** 类（程序 12-1）是打印多行字符串的抽象类。

`getColumns` 和 `getRows` 分别是取得横向字数和纵向行数的方法。它是一个抽象方法，要等待在子类进行实现（*subclass responsibility*）。`getRowText` 是取得被指定该行的字符串的方法。这也是抽象方法，要等待在子类进行实现。

`show` 是打印所有行数的方法。一连串的行为包括：以 `getRows` 方法取得行数，以 `getRowText` 方法取得应打印的字符串，再以 `for` 循环打印所有行数。`show` 是一个应用了 `getRows` 和 `getRowText` 这两个抽象方法的 *Template Method Pattern*（第 3 章）。

## 程序 12-1 Display 类 (Display.java)

```

public abstract class Display {
    public abstract int getColumns();           // 取得横向的字数
    public abstract int getRows();              // 取得纵向的行数
    public abstract String getRowText(int row); // 取得第 row 个字符串
    public final void show() {                  // 打印所有内容
        for (int i = 0; i < getRows(); i++) {
            System.out.println(getRowText(i));
        }
    }
}

```

## StringDisplay 类

如果只看 Display 类会抓不到意思，所以我们直接来看子类的 StringDisplay 类。

StringDisplay 类（程序 12-2）是打印整行字符串的类。StringDisplay 类是 Display 类的子类，因此有责任要实现 Display 类所声明的抽象方法。

string 字段则存储打印的字符串。StringDisplay 类所打印的只有 string 字段的一整行而已，所以 getColumns 返回 string.getBytes().length 的值，getRows 则返回 1。

另外，getRowText 只有在取第 0 行的值时才会返回 string 字段。如果套用前面的蛋糕示例来比喻的话，StringDisplay 类就相当于生日蛋糕中最里面的蛋糕。

## 程序 12-2 StringDisplay 类 (StringDisplay.java)

```

public class StringDisplay extends Display {
    private String string;                      // 打印的字符串
    public StringDisplay(String string) {        // 以参数指定打印的字符串
        this.string = string;
    }
    public int getColumns() {                   // 字数
        return string.getBytes().length;
    }
    public int getRows() {                     // 行数为 1
        return 1;
    }
    public String getRowText(int row) {         // 仅在 row 为 0 时才返回
        if (row == 0) {
            return string;
        } else {

```

## Decorator (装饰) —— 对装饰和内容一视同仁

```

        return null;
    }
}
}

```

---

## Border 类

Border 类 (程序 12-3) 是表示“装饰外框”的抽象类。虽然是表示装饰外框的类，但仍定义为打印字符串的 Display 类的子类。

换句话说，因为继承的关系，装饰外框会有同样的内容和方法。讲得更具体一点，Border 类会继承 getColumns, getRows, getRowText, show 的方法。从接口 (API) 的立场来看，“装饰外框 (Border) 跟内容 (Display) 有同样方法”的意思就等于“可以对装饰外框和内容一视同仁”等等。好像讲得太快了，各位也许还没会过意来。先继续看下面的说明吧。

现在装饰外框的 Border 类已经有 Display 类型的 display 字段。这是指在装饰外框里面的“内容”。但是，display 的内容并不只限于 StringDisplay 的对象实例而已。毕竟 Border 也是 Display 的子类，所以 display 字段的内容也有可能是另一个装饰外框 (Border 类的子类的对象实例)。这个新的装饰外框也有 display 字段，嗯，是不是已经开始浮现出 Decorator Pattern 的样子了呢？

**程序 12-3 Border 类 (Border.java)**

```

public abstract class Border extends Display {
    protected Display display;           // 指装饰外框里面的“内容”
    protected Border(Display display) { // 在产生对象实例时，以参数指
// 定“内容”
        this.display = display;
    }
}

```

---

## SideBorder 类

SideBorder 类 (程序 12-4) 是一种具体的装饰外框，也是 Border 类的子类。SideBorder 类是在字符串左右两边加上指定装饰字符(borderChar)。例如，若 borderChar 字段的值为 '|'，则在“内容”的左右两边加上指定字符，然后以 show 打印出来，即：

| 内容 |

borderChar 字段以构造函数指定。

SideBorder 不是抽象类，因为父类所声明的抽象方法都在这里进行实现。

getColumns 是取得打印内容的横向字数的方法。字数如何计算出来？其实很简单，

## 如何设置装饰字符

只要在装饰外框里面的“内容”字数加上左右两个装饰字符即可。内容的字数又要怎么取得？当然是用 `display.getColumns()` 这个表达式。因为 `display` 字段在 `Border` 类被设为 `protected`，所以可以从子类直接使用。加上左右两边的装饰字符之后，返回值即为：

```
1 + display.getColumns() + 1
```

当然也可以写成 `display.getColumns() + 2`，毕竟这个表达式很清楚就是左右各加 1 个。

了解了 `getColumns` 方法的构造之后，`getRows` 方法就可以如法炮制。由于 `SideBorder` 类并没有处理上下的部分，所以表达式 `display.getRows()` 的值即为 `getRows` 方法的值。

`getRowText` 是取得参数所指定该行的字符串。在 `display.getRowText(row)` 的内容两边加上 `borderChar` 的装饰字符，即可得到：

```
borderChar + display.getRowText(row) + borderChar
```

这就是 `getRowText` 的返回值（原来这就是 `SideBorder` 的“装饰”）。

---

**程序 12-4 SideBorder 类 (SideBorder.java)**


---

```
public class SideBorder extends Border {
    private char borderChar; // 装饰字符
    public SideBorder(Display display, char ch) { // 以构造函数指定
        // Display 和装饰字符
        super(display);
        this.borderChar = ch;
    }
    public int getColumns() { // 字数要再加上内容
        // 两边的装饰字符
        return 1 + display.getColumns() + 1;
    }
    public int getRows() { // 行数同内容的行数
        return display.getRows();
    }
    public String getRowText(int row) { // 指定该行的内容即为在内容
        // 指定行的两边加上装饰字符
        return borderChar + display.getRowText(row) + borderChar;
    }
}
```

---

## Decorator (装饰) —— 对装饰和内容一视同仁

**FullBorder 类**

FullBorder 类 (程序 12-5) 也跟 SideBorder 类一样, 都是 Border 的子类之一。SideBorder 类只在内容的左右两边加上装饰字符, 而 FullBorder 类则是在上下、左右加入装饰字符。不过, 虽然 SideBorder 类可以自行指定装饰字符, 但 FullBorder 类的装饰字符是固定不变的。

makeLine 方法是补充的方法, 可以用指定字符建立连续字符串 (设为 private, 以免有非此类的部分跑进来用)。

程序 12-5 FullBorder 类 (FullBorder.java)

```
public class FullBorder extends Border {
    public FullBorder(Display display) {
        super(display);
    }
    public int getColumns() { // 字数 = 内容字数 + 内容两边的装饰字符
        return 1 + display.getColumns() + 1;
    }
    public int getRows() { // 行数 = 内容行数 + 上下的装饰字符
        return 1 + display.getRows() + 1;
    }
    public String getRowText(int row) { // 指定该行的内容
        if (row == 0) { // 外框顶端
            return "+" + makeLine('-', display.getColumns()) + "+";
        } else if (row == display.getRows() + 1) { // 外框底部
            return "+" + makeLine('-', display.getColumns()) + "+";
        } else { // 其他部分
            return "|" + display.getRowText(row - 1) + "|";
        }
    }
    private String makeLine(char ch, int count) { // 以字符 ch, 建立
// 重复 count 次的连续字符串
        StringBuffer buf = new StringBuffer();
        for (int i = 0; i < count; i++) {
            buf.append(ch);
        }
        return buf.toString();
    }
}
```

## Main 类

Main 类（程序 12-6）是测试用的类。这里对象实例 b1~b4 的作用分别为：

- b1——打印“Hello, world.”，没有任何装饰
- b2——把装饰字符‘#’加在 b1 的左右两边
- b3——把 b2 全部加上装饰外框
- b4——在“你好。”外面加上多重的装饰外框

程序 12-6 Main 类 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        Display b1 = new StringDisplay("Hello, world.");
        Display b2 = new SideBorder(b1, '#');
        Display b3 = new FullBorder(b2);
        b1.show();
        b2.show();
        b3.show();
        Display b4 =
            new SideBorder(
                new FullBorder(
                    new FullBorder(
                        new SideBorder(
                            new FullBorder(
                                new StringDisplay("您好。")
                            )
                        )
                    )
                )
            );
        b4.show();
    }
}
```

|                 |                  |
|-----------------|------------------|
| Hello, world.   | ←1. 打印 b1.show() |
| #Hello, world.# | ←2. 打印 b2.show() |
| +-----+         | ←3. 打印 b3.show() |

## Decorator (装饰) —— 对装饰和内容一视同仁

```
|#Hello, world.#
+-----+
/+-----+/-> b4.print()
|+-----+/
|| *-----+* || | |
|| *| 您好。 | * ||
|| *-----+* ||
|+-----+/
/+-----+/

```

图 12-3 执行结果

请利用类图（图 12-4）弄清楚对象实例 b1, b2, b3 之间的关系，可以看得出来 b1 的装饰外框是 b2、而 b2 的装饰外框则是 b3 吧？



图 12-4 b3, b2, b1 的类图

## Decorator Pattern 的所有参与者

Decorator Pattern 中出现的所有参与者可整理如下。

### ◆ Component 参与者

新增功能时的核心参与者。在蛋糕的比喻说明中，相当于装饰前的蛋糕。Component 参与者只规定蛋糕的接口（API）。在前面的程序示例中，扮演这个角色的是 Display 类。

### ◆ ConcreteComponent 参与者

实现 Component 参与者接口（API）的具体蛋糕。在前面的程序示例中，扮演这个角色的是 StringDisplay 类。

### ◆ Decorator (装饰) 参与者

具有跟 Component 参与者相同的接口（API），另外还有 Decorator 参与者要装饰的 Component 参与者。这个参与者是“早就知道”自己要去装饰的对象。在前面的程序示例中，扮演这个角色的是 Border 类。

### ◆ ConcreteDecorator (具体装饰) 参与者

具体的 Decorator 参与者。在前面的程序示例中，扮演这个角色的是 SideBorder 类和 FullBorder 类。

Decorator Pattern 的类图即如图 12-5 所示。

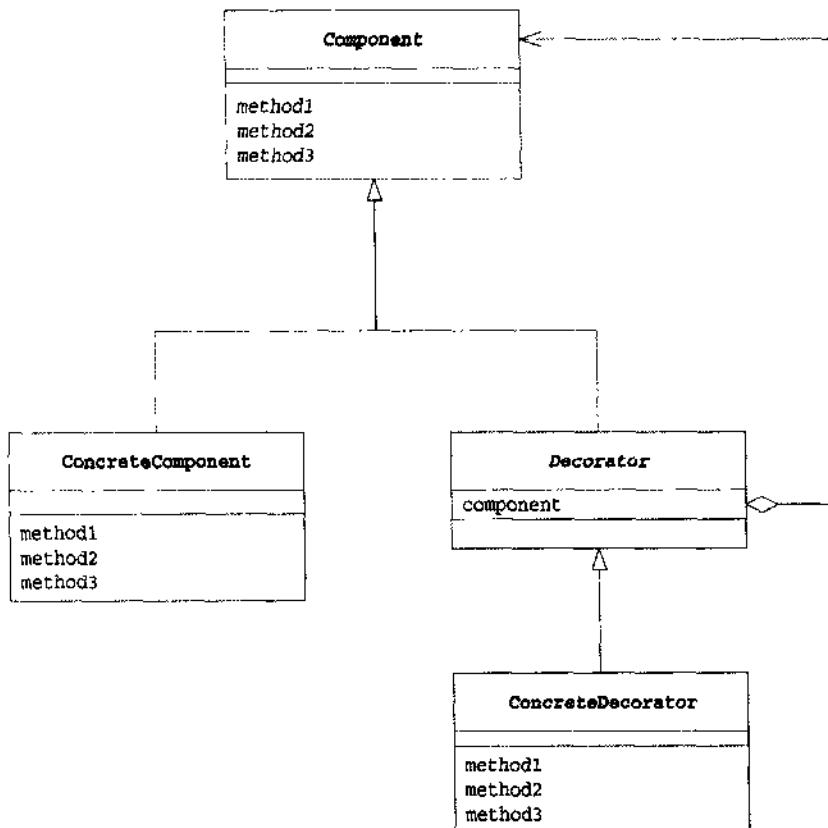


图 12-5 Decorator Pattern 的类图

## 扩展自我视野的提示

### 可穿透的接口 (API)

Decorator Pattern 对装饰外框和内容是一视同仁。基本上，在程序示例中表示装饰外框的 Border 类是表示内容的 Display 类的子类，这就是一视同仁。换句话说，Border 类（以及其子类）具有跟表示内容的 Display 类相同的接口（API）。

## Decorator（装饰）——对装饰和内容一视同仁

即使使用装饰外框把内容包起来，也不能隐藏接口（API）。`getColumns`, `getRows`, `getRowText`, 以及 `show` 这几个方法并没有被隐藏起来，还是可以从其他类看得到。这个特性就称为“可穿透的”接口（API）。

如程序示例中的对象实例 `b4` 所示，即使用了再多的装饰外框，接口（API）本身仍然没有受到影响。

因为接口（API）具有可穿透性，所以 Decorator Pattern 就有跟 Composite Pattern 很像的递归架构。也就是说，这个架构是：装饰外框所存储的“内容”事实上是另一个“装饰外框”构造。就好像在剥洋葱，当你剥掉一层洋葱皮，以为已经到了内容，结果看到的还是另一层洋葱皮。Decorator Pattern 和 Composite Pattern 两者在处理递归架构的部分很像，但目的不尽相同。因为 Decorator Pattern 使用多重外框的原因是为了新增功能。

## 可新增功能但内容不变

在 Decorator Pattern 当中，装饰外框和内容具有共享的接口（API）。虽然接口（API）是共享，但功能逐渐一一新增进去的时候，外面就一层层越包越多。如果是用 `SideBorder` 来包 `Display` 的话，打印时可以设定在左右两边加入新的装饰字符；如果是用 `FullBorder` 来包，整个外围都要加上装饰外框。你并不需要去修改它的包装方式，因为它不需要换掉被包起来的主体，就能新增其他功能。

Decorator Pattern 使用到委托。所有对“装饰外框”的要求（调用方法）都会推到（委托到）“内容”去。本章的程序示例是在 `SideBorder` 的 `getColumns` 方法当中调用 `display.getColumns()`，而在 `getRows` 方法调用 `display.getRows()`。

## 可新增动态功能

Decorator Pattern 所使用的委托并没有紧密结合类，所以不需要修改框架的源代码就能建立一个改变对象关系的新对象。

## 即使只有简单的种类规划，也可增加多种功能

利用 Decorator Pattern 可以新增多种不同的功能，因为只要准备好多种具体装饰外框（ConcreteDecorator 参与者），就能随意组合成新对象。各个装饰外框就算很简单也无所谓。

这就像是一个可供客人自行选择香草、巧克力、草莓、奇异果等多种组合的冰淇淋店。如果想要把客人可能要求的所有冰淇淋口味组合都准备好，时间上、空间上都是一大负担。店里只要先准备各种口味的大冰淇淋桶，再根据客人的要求组合，就能完成客人想要的特殊口味冰淇淋。有些客人只要香草口味、有人要咖啡兰姆加开心果、

## Decorator 模式的应用

小朋友可能贪心的要香草、草莓加奇异果等。Decorator Pattern 非常适合用来适应多种不同的要求。

## java.io 包和 Decorator Pattern

接下来要谈谈 java.io 包里的类。java.io 是一个管理输出 / 输入 (Input/Output, 简称 I/O) 的包，里面也用到 Decorator Pattern。

首先，可建立一个从文件读入数据的对象实例，如：

```
Reader reader = new FileReader("datafile.txt");
```

而从文件读入数据时，会需要作缓冲处理 (buffering)。

```
Reader reader = new BufferedReader(
    new FileReader("datafile.txt")
);
```

如此一来，如果要建立 BufferedReader 类的对象实例时，就会指定 FileReader 类的对象实例作为实际上数据被读入的位置。

接着，还有行号管理的部分。

```
Reader reader = new LineNumberReader(
    new BufferedReader(
        new FileReader("datafile.txt")
    )
);
```

Reader 类的对象实例（其子类的对象实例）可以传递给 LineNumberReader 的构造函数或 BufferedReader 的构造函数，因此可以有像上面这样的组合。

而行号虽然有管理，但有时不一定会做缓冲处理。

```
Reader reader = new LineNumberReader(
    new FileReader("datafile.txt")
);
```

下面是有做行号管理和缓冲处理，但是从网络端读入数据而非从文件（已省略掉细节和例外处理）。

```
java.net.Socket socket = new Socket(hostname, portnumber);
...
Reader reader = new LineNumberReader(
    new BufferedReader(
        new InputStreamReader(
            socket.getInputStream()
        )
    )
);
```

## Decorator (装饰) —— 对装饰和内容一视同仁

这里的 `InputStreamReader` 类是根据 `InputStream` 类的对象实例 (`getInputStream` 的返回值) 提供 `Reader` 类的接口 (API) (记得吗, 这是 Adapter Pattern (第 2 章))。

除了 `java.io` 包之外, `javax.swing.border` 包也用到 Decorator Pattern。`javax.swing.border` 包里是一些打印出来的组件外围的装饰外框用的类。

### 小类多了一点

Decorator Pattern 不是没有缺点, 彼此类似的小类一不小心就愈建愈多。

## 相关 Pattern

### ◆ Adapter Pattern (第 2 章)

Decorator Pattern 不改变内容的接口 (API) 就能建立外框 (穿透作用)。

Adapter Pattern 则是用来连接两个有距离的接口 (API)。

### ◆ Strategy Pattern (第 10 章)

Decorator Pattern 可利用更换外框或增加其他外框的方式来新增功能。

Strategy Pattern 是以切换运算法则的方式变换功能。

## 深入说明：继承和委托的一视同仁

这个部分想让各位更进一步思考一下“一视同仁”, 也就是“看成同等物体”。

## 继承——对子类和父类一视同仁

子类可以跟父类一视同仁, 举个简单的例子来说明:

```
class Parent {
    ...
    void parentMethod() {
        ...
    }
}

class Child extends Parent {
    ...
    void childMethod() {
        ...
    }
}
```

## 父类类型的引用

在这个示例中，Child 的对象实例可以直接指定到 Parent 类型的变量。然后就可以调用从 Parent 继承而来的方法。

```
Parent obj = new Child();
obj.parentMethod();
```

换句话说，Child 的对象实例会被视为 Parent 的对象实例来处理。这就是把子类视为父类的例子。

附带说明一下，如果要把这个例子反过来，也就是把父类视为子类时，则须做强迫转型。

```
Parent obj = new Child();
((Child)obj).childMethod();
```

## 委托——对自己和委托对象一视同仁

当接口利用委托而具有穿透性时，可以让自己跟委托对象一视同仁。

请先看这个还不成熟的例子。

```
class Rose {
    Violet obj = ...
    void method() {
        obj.method();
    }
}

class Violet {
    void method() {
        ...
    }
}
```

Rose 和 Violet 具有相同的 method，而 Rose 委托给 Violet。看起来有点怪，这个类好像连起来，但又好像没有连起来。

会让人觉得怪怪的地方就在双方有一个共享的方法 method，但这个“共享”的信息又没有写在源代码里面。当然，这里还是有 method 的定义没错。如果有一个像下面这样的共享抽象类 Flower 的话，整个结构就会比较紧密。

```
abstract class Flower {
    abstract void method();
}
```

## Decorator (装饰) —— 对装饰和内容一视同仁

```

class Rose extends Flower {
    Violet obj = ...
    void method() {
        obj.method();
    }
}
class Violet extends Flower {
    void method() {
        ...
    }
}

```

或者，Flower 也可能是接口。

```

interface Flower {
    abstract void method();
}

class Rose implements Flower {
    Violet obj = ...
    void method() {
        obj.method();
    }
}

class Violet implements Flower {
    void method() {
        ...
    }
}

```

看到这里，或许有些读者会心生疑问：“Rose 里的字段 obj 被特定成 Violet 类型不会有问题是吗”、“为什么不改用较广的 Flower 类型？不是会比较好吗”，其实应该怎么处理最好，还是要看实际的程序而定。

## 重点回顾

本章介绍了在不影响到穿透性接口（API）的前提下，逐一新增功能到对象上的 Decorator Pattern。另外，还有继承和委托。相信各位已经差不多得到了抽象类和接口的感觉了。

## 小结与综合应用

请各位别忘了试做一下练习题。

## 练习题

解答请见附录 A

### 问题 1

请在本章的程序示例新增一个 UpDownBorder 类, 让它能打印出上下的装饰字符。UpDownBorder 类的用法如程序 12-7, 其执行结果如图 12-6 所示。

程序 12-7 Main 类 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        Display b1 = new StringDisplay("Hello, world.");
        Display b2 = new UpDownBorder(b1, '-');
        Display b3 = new SideBorder(b2, '*');
        b1.show();
        b2.show();
        b3.show();
        Display b4 =
            new FullBorder(
                new UpDownBorder(
                    new SideBorder(
                        new UpDownBorder(
                            new SideBorder(
                                new StringDisplay("您好。"),
                                '*'
                            ),
                            '='
                        ),
                        '|'
                    ),
                    '/'
                )
            );
        b4.show();
    }
}
```

## Decorator (装饰) ——对装饰和内容一视同仁

```

Hello, world.           ← 1. 打印 b1.snow()
-----
Hello, world.           ← 2. 打印 b2.snow()

-----
+-----+           ← 3. 打印 b3.snow()
|#Hello, world.|#
+-----+
+-----+           ← 4. 打印 b4.snow()
|//////////|
|=====|
||*您好。 * ||
|=====|
|//////////|
+-----+

```

图 12-6 执行结果

## 问题 2

请建立一个 ConcreteComponent 参与者，假设它是可打印多个字符串的 MultiStringDisplay 类。MultiStringDisplay 的用法如程序 12-8，其执行结果如图 12-7 所示。

## 程序 12-8 Main 类 (Main.java)

```

public class Main {
    public static void main(String[] args) {
        MultiStringDisplay md = new MultiStringDisplay();
        md.add("早安。");
        md.add("午安。");
        md.add("晚安，明天见。");
        md.show();

        Display d1 = new SideBorder(md, '#');
        d1.show();

        Display d2 = new FullBorder(md);
        d2.show();
    }
}

```

```
早安。          ← 打印 md.snow()  
午安。  
晚安，明天见。  
#早安。      #          ← 打印 d1.snow()  
#午安。      #  
#晚安，明天见。 #  
+-----+          ← 打印 d2.snow()  
|早安。      |  
|午安。      |  
|晚安，明天见。 |  
+-----+
```

图 12-7 执行结果

# 第 13 章

## Visitor（访问者） ——在结构中穿梭 还同时做事

## Visitor Pattern

不知道各位有没有听过圣诞节的故事。约瑟夫带着即将临盆的玛莉亚寻找旅社，因为伯利恒街上有很多家旅社，所以两人挨家挨户去敲旅社的门。

本章要学习的是

Visitor Pattern

visitor 是“旅客、访客”的意思。

数据结构里存储了很多个元素，假设现在要对所有元素进行一项“处理”。那么，这项“处理”的程序代码应该写在哪里？以常理来判断，应该要写在表示数据结构的类里面，不过如果这项“处理”的行为不只一个的话，该怎么办？每次要做新处理的时候，就必须修改数据结构的类。

Visitor Pattern 把数据结构和处理两者分开，另外写一个表示在数据结构内穿梭来去的主体“访客”的类，然后把处理交给这个类来进行。如此一来，如果想追加新的处理行为时，只要再建立一个新的“访客”即可。而在数据结构这边，也只要能接受来敲门的“访客”就能完成动作。

## 程序示例

请看下面 Visitor Pattern 的程序示例。这里借用 Composite Pattern (第 11 章) 的文件与目录的示例，来当作访客在其中穿梭来去的数据结构。这个程序是访客穿梭在由文件和目录组成的数据结构内，以打印文件信息。

### Visitor 类

Visitor 类 (程序 13-1) 是表示“访客”的抽象类。这个访客必须要有可访问的数据结构 (即 File 和 Directory) 才能继续存在。

在 Visitor 类声明了两个跟 visit 同名的方法。虽然名称相同，但参数并不一样。一个是 File，另一个则是 Directory。即使名称相同，不过它会根据调用方法时的参数类型，自动辨识是哪一个方法。这称为方法的重载 (overload)。

表 13-1 类与接口一览表

| 名称          | 说明                                        |
|-------------|-------------------------------------------|
| Visitor     | 表示访问文件或目录的访客的抽象类                          |
| Acceptor    | 表示接受 Visitor 类的对象实例的数据结构的接口               |
| ListVisitor | Visitor 类的子类，打印文件或目录信息的类                  |
| Entry       | File 和 Directory 的父类的抽象类 (实现 Acceptor 接口) |

## Visitor (访问者) —— 在结构中穿梭还同时做事

续上表

| 名称                     | 说明                     |
|------------------------|------------------------|
| File                   | 表示文件的类                 |
| Directory              | 表示目录的类                 |
| FileTreatmentException | 发生在对 File 进行 add 时的例外类 |
| Main                   | 测试用的类                  |

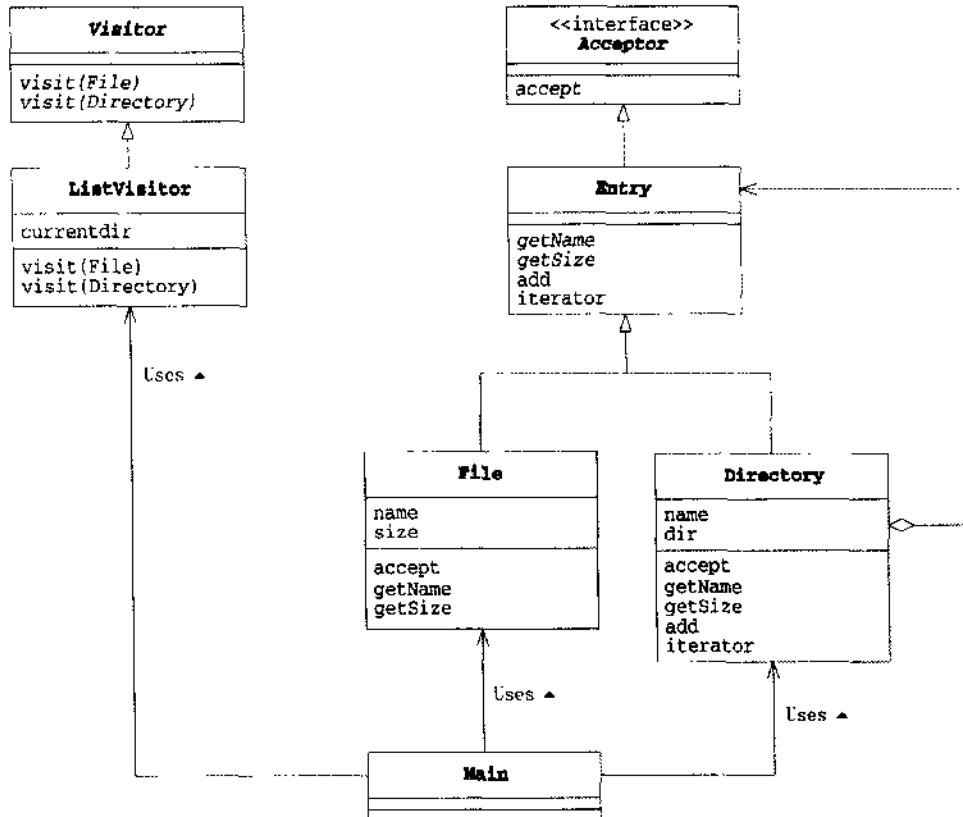


图 13-1 程序示例的类图

`visit(File)`是 `File` 类去找 `File` 时所调用的方法, `visit(Directory)`则是 `Directory` 类去找 `Directory` 时所调用的方法。因为 Visitor Pattern 在类间彼此调用的部分比较复杂, 因此如果只看 Visitor 类对程序理解毫无帮助。请各位先了解 Visitor 类有 2 个同样名为 `visit` 的方法, 再进入下一个类。

## 程序 13-1 Visitor 类 (Visitor.java)

```
public abstract class Visitor {
```

## Java语言中的应用

```

public abstract void visit(File file);
public abstract void visit(Directory directory);
}

```

## Acceptor 接口

Visitor 类是表示“访客”的类，相对地，Acceptor 接口（程序 13-2）则是接受访客的接口。如果说 Visitor 类是玛莉亚，那么 Acceptor 接口就相当于旅社（实际的旅社是实现 Acceptor 接口的类的对象实例）。

在 Acceptor 接口所声明的方法是 accept（accept 的英文原意是“接受”）。参数是访客 Visitor 类。

**程序 13-2 Acceptor 接口（Acceptor.java）**

```

public interface Acceptor {
    public abstract void accept(Visitor v);
}

```

## Entry 类

基本上，Entry 类（程序 13-3）就是 Composite Pattern 的程序示例（程序 11-1）里的 Entry 类，只不过在本章它是实现（implements）Acceptor 接口。这是为了让 Entry 类能符合 Visitor Pattern 的需求。实际上实现在 Acceptor 接口所声明的抽象方法 accept 的是 File 类和 Directory 类（Entry 的子类）。

add 方法只有在 Directory 类才有效，所以在 Entry 类要设为异常。用来取得元素的 Iterator 所得到的 iterator 方法也同样只在 Directory 类才有效，因此也跟 add 方法同样在 Entry 类要设成异常。

**程序 13-3 Entry 类（Entry.java）**

```

import java.util.Iterator;

public abstract class Entry implements Acceptor {
    public abstract String getName(); // 取得文件名
    public abstract int getSize(); // 取得文件容量
    public Entry add(Entry entry) throws FileTreatmentException {
        // 新增进入点
        throw new FileTreatmentException();
    }
    public Iterator iterator() throws FileTreatmentException {

```

## Visitor(访问者)——在结构中穿梭还同时做事

```

// 产生 Iterator
    throw new FileTreatmentException();
}

public String toString() { // 印出字符串
    return getName() + " (" + getSize() + ")";
}
}

```

---

## File类

File类(程序13-4)也跟Composite Pattern的程序示例中的File类差不多。Visitor Pattern的重点当然是要怎么去实现accept方法。accept方法的参数是Visitor类，利用v.visit(this);这个语句从accept方法中调用出Visitor的visit方法。虽然visit方法有重载(overload)，不过这里所调用的是visit(File)。因为这里的this是File类的对象实例。

调用visit方法的行为会把来访的File的对象实例(this)告诉Visitor(光是简短几段话可能不太容易弄清楚visit跟accept的关系，在本章的“Visitor和Acceptor两边彼此调用”小节中会重新再配合流程图一起解释这个关系)。

程序13-4 File类(File.java)

```

public class File extends Entry {
    private String name;
    private int size;
    public File(String name, int size) {
        this.name = name;
        this.size = size;
    }
    public String getName() {
        return name;
    }
    public int getSize() {
        return size;
    }
    public void accept(Visitor v) {
        v.visit(this);
    }
}

```

---

## Directory 类

Directory 类（程序 13-5）是表示目录的类。跟 Composite Pattern 的时候比起来，这里多了 2 个方法。

第 1 个是 iterator 方法。iterator 方法会传回 Iterator，利用它可以取得目录下所有目录进入点（文件和目录）的信息。

第 2 个当然就是 accept 方法。如 File 类的 accept 调用 visit(File)方法，Directory 类的 accept 会调用 visit(Directory)方法。这样才能告诉访客：“你现在来的地方是 Directory 的对象实例”。

程序 13-5 | Directory 类 (Directory.java)

```
import java.util.Iterator;
import java.util.Vector;

public class Directory extends Entry {
    private String name;                      // 目录名称
    private Vector dir = new Vector();         // 目录进入点的集合
    public Directory(String name) {           // 构造函数
        this.name = name;
    }
    public String getName() {                  // 取得名称
        return name;
    }
    public int getSize() {                    // 取得目录容量
        int size = 0;
        Iterator it = dir.iterator();
        while (it.hasNext()) {
            Entry entry = (Entry)it.next();
            size += entry.getSize();
        }
        return size;
    }
    public Entry add(Entry entry) {           // 新增进入点
        dir.add(entry);
        return this;
    }
    public Iterator iterator() {             // 产生 Iterator
        return dir.iterator();
    }
}
```

## Visitor (访问者) —— 在结构中穿梭还同时做事

```

    }

    public void accept(Visitor v) {      // 接受访客
        v.visit(this);
    }
}

```

---

**ListVisitor 类**

ListVisitor 类（程序 13-6）是 Visitor 类的子类，这个类可以穿梭在数据结构之间并打印信息。因为它是 Visitor 类的子类，所以要实现 visit(File)方法和 visit(Directory)方法。

currentdir 字段是存储目前的目录名称。如果访问的对象是文件，就从 File 类的 accept 方法当中调用出 visit(File)方法。参数 file 是访问的 File 类的对象实例。换句话说，这个 visit(File)方法是语句“应对 File 类的对象实例进行的处理”的位置。这里先在目前目录名称（currentdir）的后面加上斜线符号（/），然后再打印出文件名称。

如果访问的对象是目录，就从 Directory 类的 accept 方法当中调用出 visit(Directory)方法。参数 directory 是 Directory 类的对象实例。

“应对 Directory 类的对象实例进行的处理”写在 visit(Directory)方法里面。

“处理”到底是指什么？跟 visit(File)一样，在打印出目录之后，以 iterator 方法取得目录的 Iterator，对该目录的所有进入点调用 accept 方法。因为目录下有很多目录进入点，所以要一个个去访问。

accept 方法调用 visit 方法、visit 方法则调用 accept 方法，这是一个相当复杂的递归式调用方法。一般的递归式调用方法是方法自己调用自己，但在 Visitor Pattern 则是 visit 方法和 accept 方法两者互相调用对方。

**程序 13-6 ListVisitor 类（程序 Visitor.java）**

```

import java.util.Iterator;

public class ListVisitor extends Visitor {
    private String currentdir = "";           // 目前所在的目录名称
    public void visit(File file) {            // 访问文件时调用
        System.out.println(currentdir + "/" + file);
    }
    public void visit(Directory directory) { // 访问目录时调用
        System.out.println(currentdir + "/" + directory);
        String savedir = currentdir;
        currentdir = currentdir + "/" + directory.getName();
        Iterator it = directory.iterator();

```

```

        while (it.hasNext()) {
            Entry entry = (Entry)it.next();
            entry.accept(this);
        }
        currentdir = savedir;
    }
}

```

---

## FileTreatmentException 类

FileTreatmentException 类（程序 13-7）与 Composite Pattern 的部分相同。

**程序 13-7 FileTreatmentException 类 (FileTreatmentException.java)**

```

public class FileTreatmentException extends RuntimeException {
    public FileTreatmentException() {
    }
    public FileTreatmentException(String msg) {
        super(msg);
    }
}

```

---

## Main 类

Main 类（程序 13-8）几乎跟 Composite Pattern 的 Main 类一模一样，惟一不同的地方是要使用访客 ListVisitor（执行打印）的对象实例来打印 Directory 的内容。

前面的 Composite Pattern 是使用 printList 方法来打印目录，这个方法在 Directory 类（即数据结构这边）进行过实现。而 Visitor Pattern 的打印目录的部分也是在 Visitor 这边进行，因为打印目录也是一种对数据结构内所有元素的处理。

**程序 13-8 Main 类 (Main.java)**

```

public class Main {
    public static void main(String[] args) {
        try {
            System.out.println("Making root entries...");
            Directory rootdir = new Directory("root");
            Directory bindir = new Directory("bin");
            Directory tmpdir = new Directory("tmp");
            Directory usrdir = new Directory("usr");
            rootdir.add(bindir);

```

## Visitor(访问者)——在结构中穿梭还同时做事

```

rootdir.add(tmpdir);
rootdir.add(usrdir);
bindir.add(new File("vi", 10000));
bindir.add(new File("latex", 20000));
rootdir.accept(new ListVisitor());


System.out.println("");
System.out.println("Making user entries...");
Directory yuki = new Directory("yuki");
Directory hanako = new Directory("hanako");
Directory tomura = new Directory("tomura");
usrdir.add(yuki);
usrdir.add(hanako);
usrdir.add(tomura);
yuki.add(new File("diary.html", 100));
yuki.add(new File("Composite.java", 200));
hanako.add(new File("memo.tex", 300));
tomura.add(new File("game.doc", 400));
tomura.add(new File("junk.mail", 500));
rootdir.accept(new ListVisitor());
} catch (FileTreatmentException e) {
    e.printStackTrace();
}
}
}

```

Making root entries...

```

/root (30000)
/root/bin (30000)
/root/bin/vi (10000)
/root/bin/latex (20000)
/root/tmp (0)
/root/usr (0)

```

Making user entries...

```

/root (31500)
/root/bin (30000)
/root/bin/vi (10000)

```

## Java语言中的应用

```
/root/bin/latex (20000)
/root/tmp (0)
/root/usr (1500)
/root/usr/yuki (300)
/root/usr/yuki/diary.html (100)
/root/usr/yuki/Composite.java (200)
/root/usr/hanako (300)
/root/usr/hanako/memo.tex (300)
/root/usr/tomura (900)
/root/usr/tomura/game.doc (400)
/root/usr/tomura/junk.mail (500)
```

图 13-2 执行结果

## Visitor 和 Acceptor 两边彼此调用

看到这里，各位是不是已经很清楚 Visitor Pattern 了？笔者当初刚在学习 Visitor Pattern 的时候，说老实话，简直就像鸭子听雷一雾沙沙，因为脑海中 visit 方法跟 accept 方法的行为已经全部搅和在一起，搞不清楚谁是谁。这里特别利用流程图（图 13-3）来说明处理的整个流向。

为了避免复杂而不易整理，图 13-3 的处理流程图示是假设在 1 个目录下只有 2 个文件。

首先，Main 类先产生 ListVisitor 的对象实例。在程序示例中，Main 类还有产生其他的 Directory 或 File 的对象实例，但在这个流程图中省略。

Main 对 Directory 的对象实例调用 accept 方法。此时，ListVisitor 的对象实例会传递给参数，不过在此省略。

Directory 的对象实例则调用参数所传递过来的 ListVisitor 的 visit(Directory)方法。于是 ListVisitor 的对象实例便开始在目录中寻找并调用第一个文件的 accept 方法。再把自己本身 (this) 传递给参数。

File 的对象实例则调用参数所传递过来的 ListVisitor 的 visit(File)方法。请注意此时 ListVisitor 还在执行 visit(Directory)中（不是多线架构执行中的意思，而是指在调用记录 (call stack) 里有 visit(Directory)的意思。在本图中以长方形重迭在右边的方式表示）。

等到从 visit(File)、accept 传回之后，接着再调用另一个 File 的对象实例（同一目录底下的第 2 个文件）的 accept 方法。然后把 ListVisitor 的对象实例 (this) 传递给参数。

重复前面的动作，File 的对象实例调用 visit(File)的方法。当所有方法的处理结束之后，便一步步往上走到调用位置，最后 Main 的调用 accept 方法也传回来。

## Visitor (访问者) —— 在结构中穿梭还同时做事

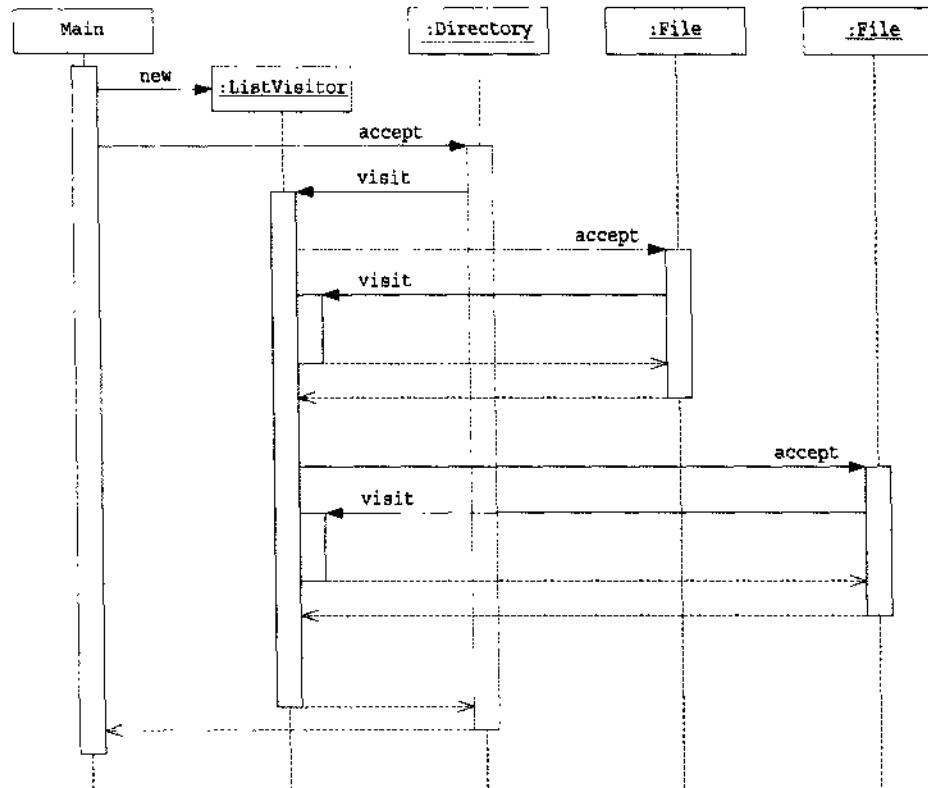


图 13-3 程序示例的流程图（1个目录下只有 2 个文件）

请找出图中符合下列语句的重点。

- 分别对 Directory 的对象实例和 File 的对象实例调用 accept 方法
- 每个对象实例只会调用 accept 方法 1 次
- 对 ListVisitor 的对象实例调用 visit(Directory) 和 visit(File) 方法
- 处理 visit(Directory) 和 visit(File) 的是只有这个 ListVisitor 的对象实例看出来了么？visit 的“处理”都集中在 ListVisitor。

## Visitor Pattern 的所有参与者

Visitor Pattern 中出现的所有参与者可整理如下。

### ◆ Visitor (访问者) 参与者

Visitor 参与者是对每个数据结构中的具体元素 (ConcreteAcceptor 参与者) 声明“已经去找过 XXXX”的 visit(XXXX)方法。visit(XXXX)是处理 XXXX 的方法，实际源代码则写在 ConcreteVisitor 参与者那里。在前面的程序示例中，扮演这个角色的是 Visitor 类。

### ◆ ConcreteVisitor (具体访问者) 参与者

ConcreteVisitor 参与者是实现 Visitor 参与者的接口 (API)。它实现 visit(XXXX) 格式的方法，然后语句各个 ConcreteAcceptor 参与者的处理。在前面的程序示例中，扮演这个角色的是 ListVisitor 类。就像 ListVisitor 的 currentdir 字段的值会发生变化一样，在处理 visit(XXXX) 的过程中，ConcreteVisitor 参与者的内部状态也会有变化。

### ◆ Acceptor (接受者) 参与者

Acceptor 参与者是表示 Visitor 参与者访问对象的参与者。声明接受访客的 accept 方法。Visitor 参与者则被传递给 accept 方法的参数。在前面的程序示例中，扮演这个角色的是 Acceptor 接口。

### ◆ ConcreteAcceptor (具体接受者) 参与者

ConcreteAcceptor 参与者是实现 Acceptor 参与者接口 (API) 的参与者。在前面的程序示例中，扮演这个角色的是 File 类和 Directory 类。

### ◆ ObjectStructure (对象结构) 参与者

ObjectStructure 参与者是处理 Acceptor 参与者的集合的参与者。ConcreteVisitor 参与者有可处理所有 Acceptor 参与者的方法。在前面的程序示例中，扮演这个角色的是 Directory 类 (一人饰两角)。程序示例的 Directory 类里有 iterator 方法的原因是为了让 ConcreteVisitor 参与者能处理所有的 Acceptor 参与者。

对了，在 GoF 一书中并没有用 Acceptor, ConcreteAcceptor 这两个名词，而是用 Element, ConcreteElement。

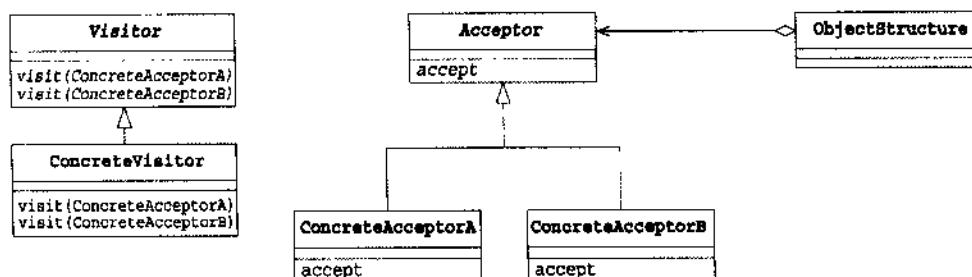


图 13-4 Visitor Pattern 的类图

## 扩展自我视野的提示

### 双重调度 (double dispatch)

Visitor Pattern 的方法调用可整理如下。

## Visitor (访问者) —— 在结构中穿梭还同时做事

accept (接受) 方法的调用为: acceptor.accept(visitor)

而 visit (访问) 方法的调用为: visitor.visit(acceptor)

两者刚好站在相反的立场。Acceptor 会 accept visitor、visitor 则 visit acceptor。

Visitor Pattern 是由 ConcreteAcceptor 参与者和 ConcreteVisitor 参与者这一组来决定实际的处理。一般称为双重调度 (double dispatch)。

## 为什么要搞得这么复杂

或许有人看到处理的流程时会觉得：“Visitor Pattern 不是越弄越复杂了吗？真是帮倒忙！”、“如果要做反复处理，为什么不直接在数据结构里写个循环就好？还搞个什么 accept、visit 的，难道一定得把方法调用弄得这么复杂吗？”

Visitor Pattern 的目的是要把处理从数据结构分出来。在集合元素或元素间的连结时，数据结构的地位非常重要。但是维持这个结构跟写一个以这个结构为基础的处理完全是两码事。

程序示例中有一个 ListVisitor 类是作为进行打印信息的处理的 ConcreteVisitor 参与者。另外在练习题里也有一个进行其他处理的 ConcreteVisitor 参与者。通常 ConcreteVisitor 参与者可以单独开发，不必跟 File 类或 Directory 类写在一起；换句话说，Visitor Pattern 能提高 File 类和 Directory 类的零件独立性。假设现在把一个处理动作设计成 File 类或 Directory 类的方法，每次想新增“处理”以扩充功能时就得去修改 File 类（程序 13-4）和 Directory 类（程序 13-5）。

## The Open-Closed Principle——扩充时开放、修改时关闭

既然都已经讲到扩充功能和修改的部分，那就来谈谈 The Open-Closed Principle (OCP) 原则吧。这个原则最先由 Bertrand Meyer 提出，整理在 Robert C. Martin 在所写的 C++ Report (Jan. 1996) 中的专栏 Engineering Notebook 里（注）。

这个原则是主张类等应该：

- 扩充 (extension) 时要打开 (open)；
- 但修改 (modification) 时要关闭 (closed)

除非有特殊理由，否则程序员在设计类时都应该容许以后继续扩充该程序。若无正当理由，就不应该禁止后人扩充程序。这就是“扩充时要打开”。

但是，如果每次扩充程序时还要去修改现有类的话，那又未免太麻烦。所以，在扩充程序时没有修改现有类的需要正是“修改时要关闭”的真正意义。

欢迎大家来扩充程序，但不准修改现有类；在不修改现有类的原则下就可以扩充。这就是 The Open-Closed Principle 原则。

对类的要求实在是变化繁杂，而且这些要求大部分都是“要求扩充功能”。此时，

如果类不能配合功能扩充的话，还真是很麻烦。然而如果换个方向想去修改已经完成、甚至已经测试完毕的类，又可能会降低软件程序的品质。

具有扩充时开放、修改时关闭这个特性的类属于零件可再用性较高的类。而所谓 Design Patterns 的目的、对象导向的目的，其实就是提供建立这种类的结构而已。

## 新增 ConcreteVisitor 参与者？简单

要想追加一个新的 ConcreteVisitor 参与者很简单。因为具体的处理可以直接丢给 ConcreteVisitor 参与者去做，不需要为了这个处理就去修改 ConcreteAcceptor 参与者。

## 新增 ConcreteAcceptor 参与者？复杂

虽然新增 ConcreteVisitor 参与者很简单，但新增 ConcreteAcceptor 参与者可就不容易了。

假设想在本章的程序示例新增一个 Entry 类的子类，姑且命名为 Device 类。也就是说，Device 类跟 File 类、Directory 类是兄弟关系。此时，必须先在 Visitor 类建立一个 visit(Device)方法，然后还要在 Visitor 类的所有子类都要实现这个 visit(Device)方法。

## Visitor 要怎样做才能进行处理

Visitor Pattern 把对数据结构的元素的处理切出来交给 Visitor 参与者，这样就能分开数据结构和对元素的处理。这样讲起来好像很冠冕堂皇，不过 Acceptor 参与者必须公开足够的信息给 Visitor 参与者。

举例来说，本章的程序示例在 visit(Directory)里对每个目录进入点都执行 accept。如果想要做这样的处理，Directory 必须提供“能取得所有目录进入点”的 iterator 方法。

访客很努力从数据结构取得必要的信息。如果不能取得必要的信息情报，访客就不能百分之百发挥功能。万一把另外不应该公开的信息都开放出来，会增加以后改良数据结构的困难度。

## 相关 Pattern

### ◆ Iterator Pattern（第 1 章）

Iterator Pattern 和 Visitor Pattern 都是在数据结构上进行处理。

Iterator Pattern 是用在想逐一取得数据结构内的元素之时。

Visitor Pattern 则是用在想对数据结构内的元素进行特殊处理之时。

## Visitor (访问者) —— 在结构中穿梭还同时做事

## ◆ Composite Pattern (第 11 章)

被访问的数据结构有可能是 Composite Pattern。

## ◆ Interpreter Pattern (第 23 章)

Interpreter Pattern 有时候会用到 Visitor Pattern。例如，在完成树状剖析(parser tree)后，穿梭在树状剖析的各个节点之间同时进行处理时。

**重点回顾**

本章所介绍的是穿梭在数据结构间并同时进行处理的 Visitor Pattern。

**练习题**

解答请见附录 A

**问题 1**

请在本章的程序示例新增一个类，该新类命名为 FileFindVisitor 类。这个类是找出符合指定扩展名的文件。假设 FileFindVisitor 类的用法如程序 13-9，FileFindVisitor 类的执行结果则如图 13-5 所示。这个问题是抓出所有扩展名为.html 的文件。

**程序 13-9 Main 类 (Main.java)**

```
import java.util.Iterator;

public class Main {
    public static void main(String[] args) {
        try {
            Directory rootdir = new Directory("root");
            Directory bindir = new Directory("bin");
            Directory tmpdir = new Directory("tmp");
            Directory usrdir = new Directory("usr");
            rootdir.add(bindir);
            rootdir.add(tmpdir);
            rootdir.add(usrdir);
            bindir.add(new File("vi", 10000));
            bindir.add(new File("latex", 20000));

            Directory yuki = new Directory("yuki");
        }
    }
}
```

## Java设计模式的应用

```

Directory hanako = new Directory("hanako");
Directory tomura = new Directory("tomura");
usrdir.add(yuki);
usrdir.add(hanako);
usrdir.add(tomura);
yuki.add(new File("diary.html", 100));
yuki.add(new File("Composite.java", 200));
hanako.add(new File("memo.tex", 300));
hanako.add(new File("index.html", 350));
tomura.add(new File("game.doc", 400));
tomura.add(new File("junk.mail", 500));

FileFindVisitor ffv = new FileFindVisitor(".html");
rootdir.accept(ffv);

System.out.println("HTML files are:");
Iterator it = ffv.getFoundFiles();
while (it.hasNext()) {
    File file = (File)it.next();
    System.out.println(file);
}
} catch (FileTreatmentException e) {
    e.printStackTrace();
}
}
}
}

```

```

HTML files are:
diary.html (100)
index.html (350)

```

图 13-5 执行结果

## 问题 2

程序示例的 Directory 类（程序 13-5）的 getSize 方法是进行“取得目录容量”的处理。请把 Directory 类的 getSize 方法改写成进行“取得容量大小”的处理的 SizeVisitor 类。

## Visitor (访问者) —— 在结构中穿梭还同时做事

## 问题 3

请在 `java.util.Vector` 建立一个具有 `Acceptor` 接口功能的 `AcceptorVector` 类。让它能对 `AcceptorVector` 的对象实例 add `Directory` 和 `File` 的对象实例，而且也能 accept `ListVisitor` 的对象实例。假设 `AcceptorVector` 的用法如程序 13-10，执行结果则如图 13-6 所示。

程序 13-10 Main 接口 (Main.java)

```
import java.util.Iterator;

public class Main {
    public static void main(String[] args) {
        try {
            Directory root1 = new Directory("root1");

            root1.add(new File("diary.html", 10));
            root1.add(new File("index.html", 20));

            Directory root2 = new Directory("root2");
            root2.add(new File("diary.html", 1000));
            root2.add(new File("index.html", 2000));

            AcceptorVector vec = new AcceptorVector();
            vec.add(root1);
            vec.add(root2);
            vec.add(new File("etc.html", 1234));

            vec.accept(new ListVisitor());
        } catch (FileTreatmentException e) {
            e.printStackTrace();
        }
    }
}
```

|                          |              |
|--------------------------|--------------|
| /root1 (30)              | ← 打印目录 root1 |
| /root1/diary.html (10)   |              |
| /root1/index.html (20)   |              |
| /root2 (3000)            | ← 打印目录 root2 |
| /root2/diary.html (1000) |              |

## JAVA语言中的应用

```
/root2/index.html (2000)  
/etc.html (1234)           - 打印文件 etc.html
```

图 13-6 执行结果

### 问题 4

被 final 声明的类是不能建立子类。例如，`java.lang.String` 类被声明为 final 类，所以不能定义如下的 `MyString` 类：

× 编译错误

```
class MyString extends String {  
    ...  
}
```

虽然在程序扩充上，`String` 类已经违背了 The Open-Closed Principle，但其实它有可容许违背原则的正当理由。你知道理由是什么吗？

# 第 14 章

## Chain of Responsibility (职责链)——责任转 送

## Chain of Responsibility 责任转送

请各位先想一下“转送”这个动作的意思。假设现在要到某家公司去取文件，跟柜台小姐说明来意后，她会告诉你应该到“业务窗口”去处理。当你走到业务窗口后，那边又告诉你这份文件目前由“客户服务部”负责；因此你得再到客户服务部去一趟，找到客户服务部后，小姐还是很客气的说文件应该要到“文管中心”去拿。虽然觉得怎么绕了一大圈，还是得去找“文管中心”。像这样子把自己的要求传送下去，一直找到适当的人或地点就是“转送”。

虽然转送这个名词听起来有点刺耳，带有负面的意义，但有时候还是需要有能表现“转送”的处理。例如，现在产生一个要求，但无法直接决定处理该要求的对象。这时候，可以把一个以上的对象串联成链（chain），依次走过这个链对象再决定最后目的地的对象。

这种 Pattern 就称为

**Chain of Responsibility Pattern**

**responsibility** 是责任的意思，整个词组直译过来应该是“职责链”，不过把它想象成是一个“转送”结构应该会比较容易记忆。

如果使用这个 Pattern，可以降低“要求端”和“处理端”之间的耦合度，让它们个别成为独立的零件。另外，还可支持需要根据发生状况改变处理要求的对象的程序。

先对人产生一个要求，如果这个人有处理的能力就处理掉；如果不能处理的话，就把要求转送给“第二个人”。同样的，如果第二个人有处理的能力时就处理掉，不能处理的话，继续转送给“第三个人”，以此类推。这就是 **Chain of Responsibility Pattern**。

## 程序示例

就以下面应用 **Chain of Responsibility Pattern** 的程序示例，来研究看看在不同状况下究竟应该由谁来处理发生的问题。所有会使用到的类如表 14-1 所示。

表 14-1 类一览表

| 名称             | 说明                       |
|----------------|--------------------------|
| Trouble        | 表示发生问题的类。内有问题编号（number）  |
| Support        | 解决问题的抽象类                 |
| Nosupport      | 解决问题的具体类（永远“不处理”）        |
| LimitSupport   | 解决问题的具体类（解决小于指定号码的问题）    |
| OddSupport     | 解决问题的具体类（解决奇数号码的问题）      |
| SpecialSupport | 解决问题的具体类（解决特殊号码的问题）      |
| Main           | 建立 Support 的连锁、产生问题的测试用类 |

## Chain of Responsibility (职责链) —— 责任转送

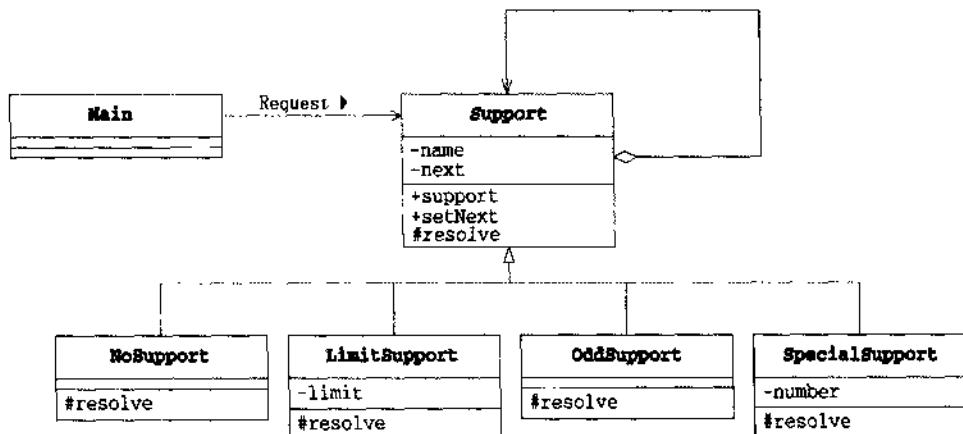


图 14-1 程序示例的类图

## Trouble 类

Trouble 类（程序 14-1）是表示该发生问题的类。number 是问题编号，利用 getNumber 方法即可取得问题编号。

程序 14-1 Trouble 类 (Trouble.java)

---

```

public class Trouble {
    private int number;           // 问题编号
    public Trouble(int number) {   // 发生问题
        this.number = number;
    }
    public int getNumber() {       // 取得问题编号
        return number;
    }
    public String toString() {    // 打印显示问题的字符串
        return "[Trouble " + number + "]";
    }
}
  
```

---

## Support 类

Support 类（程序 14-2）是建立解决问题的抽象类。

next 字段是指转送位置， setNext 方法则是设定该转送位置。

resolve 方法是预设会在子类进行实现的抽象方法。若返回值为 true，则表示要求

## Java语言中的应用

已经被处理，若为 false，则表示要求尚未处理（即转送到下一人）。

support 方法会调用 resolve 方法，若返回值为 false，则转送给“下一个人”。如果没有“下一个人”，则表示本身就是连锁处理的最后一棒，没有任何人能处理。本章示例在此情形下只将结果打印出来，但有些程序可能要抛出异常才比较妥当。

不知道各位有没有发现到：support 方法已经是利用抽象方法 resolve 的 Template Method Pattern（第 3 章）。

程序 14-2 Support 类（Support.java）

```
public abstract class Support {
    private String name;                      // 问题解决者的名称
    private Support next;                     // 转送位置
    public Support(String name) {             // 产生问题解决者
        this.name = name;
    }
    public Support setNext(Support next) {    // 设定转送位置
        this.next = next;
        return next;
    }
    public final void support(Trouble trouble) { // 解决问题的步骤
        if (resolve(trouble)) {
            done(trouble);
        } else if (next != null) {
            next.support(trouble);
        } else {
            fail(trouble);
        }
    }
    public String toString() {                // 打印字符串
        return "[" + name + "]";
    }
    protected abstract boolean resolve(Trouble trouble); // 解决的
    //方法
    protected void done(Trouble trouble) { // 已解决
        System.out.println(trouble + " is resolved by " + this + ".");
    }
    protected void fail(Trouble trouble) { // 尚未解决
        System.out.println(trouble + " cannot be resolved.");
    }
}
```

## Chain of Responsibility (职责链) —— 责任转送

```

    }
}
```

---

**NoSupport 类**

NoSupport 类（程序 14-3）是 Support 类的子类。NoSupport 类的 resolve 方法永远返回 false，也就是“自己不处理任何问题”的类。

**程序 14-3 NoSupport 类 (NoSupport.java)**

```

public class NoSupport extends Support {
    public NoSupport(String name) {
        super(name);
    }
    protected boolean resolve(Trouble trouble) { // 解决的方法
        return false; // 自己不做处理
    }
}
```

---

**LimitSupport 类**

LimitSupport 类（程序 14-4）是解决小于 limit 指定号码的问题的类。虽然 resolve 方法在判断后只返回 true 而已，但它本来应该要“解决”问题。

**程序 14-4 LimitSupport 类 (LimitSupport.java)**

```

public class LimitSupport extends Support {
    private int limit; // 若小于此号码则可解决问题
    public LimitSupport(String name, int limit) { // 构造函数
        super(name);
        this.limit = limit;
    }
    protected boolean resolve(Trouble trouble) { // 解决的方法
        if (trouble.getNumber() < limit) {
            return true;
        } else {
            return false;
        }
    }
}
```

---

## OddSupport 类

OddSupport 类（程序 14-5）是处理奇数编号的问题的类。

程序 14-5 OddSupport 类 (OddSupport.java)

```
public class OddSupport extends Support {
    public OddSupport(String name) {                      // 构造函数
        super(name);
    }
    protected boolean resolve(Trouble trouble) {           // 解决的方法
        if (trouble.getNumber() % 2 == 1) {
            return true;
        } else {
            return false;
        }
    }
}
```

## SpecialSupport 类

SpecialSupport 类（程序 14-6）是只处理指定号码的问题的类。

程序 14-6 SpecialSupport 类 (SpecialSupport.java)

```
public class SpecialSupport extends Support {
    private int number;                                // 只能解决此号码的问题
    public SpecialSupport(String name, int number) {   // 构造函数
        super(name);
        this.number = number;
    }
    protected boolean resolve(Trouble trouble) {         // 解决的方法
        if (trouble.getNumber() == number) {
            return true;
        } else {
            return false;
        }
    }
}
```

## Chain of Responsibility (职责链) —— 责任转送

## Main 类

Main 类（程序 14-7）建立了 Alice~Fred 六个问题解决者。这里全部都是使用 Support 类类型的变量，不过实际所指定的是 NoSupport、LimitSupport、SpecialSupport、OddSupport 各类的对象实例。

接下来利用 setNext 方法让 Alice~Fred 排成一列，然后把问题一个个传递给 alice，打印出是由谁来解决这个问题。程序示例中的问题编号从 0 开始，总共有 33 个，不过 33 这个数字并没有特别意义。

程序 14-7 Main 类 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        Support alice = new NoSupport("Alice");
        Support bob = new LimitSupport("Bob", 100);
        Support charlie = new SpecialSupport("Charlie", 429);
        Support diana = new LimitSupport("Diana", 200);
        Support elmo = new OddSupport("Elmo");
        Support fred = new LimitSupport("Fred", 300);
        // 形成链
        alice.setNext(bob).setNext(charlie).setNext(diana).
        setNext(elmo).setNext(fred);
        // 发生各种问题
        for (int i = 0; i < 500; i += 33) {
            alice.support(new Trouble(i));
        }
    }
}
```

```
[Trouble 0] is resolved by [Bob].
[Trouble 33] is resolved by [Bob].
[Trouble 66] is resolved by [Bob].
[Trouble 99] is resolved by [Bob].
[Trouble 132] is resolved by [Diana].
[Trouble 165] is resolved by [Diana].
[Trouble 198] is resolved by [Diana].
[Trouble 231] is resolved by [Elmo].
[Trouble 264] is resolved by [Fred].
[Trouble 297] is resolved by [Elmo].
```

## Handler 模型中的应用

```
[Trouble 330] cannot be resolved.  

[Trouble 363] is resolved by [Elmo].  

[Trouble 396] cannot be resolved.  

[Trouble 429] is resolved by [Charlie].  

[Trouble 462] cannot be resolved.  

[Trouble 495] is resolved by [Elmo].
```

图 14-2 执行结果

请看到执行结果（图 14-2）。一开始先是 Bob 在努力解决问题，但无法解决时就换成 Diana 接手。在这个执行结果当中，都没有发现 Alice 的踪影，因为 Alice 只负责转送所有问题。当问题编号超过 300 时，就没有人能处理这个程序中 LimitSupport 的对象实例。不过只有奇数编号的问题会有 OddSupport 类的 Elmo 来帮忙处理。SpecialSupport 类的 Charlie 则是专门解决问题编号 429，在整个执行结果中只出现过一次。

图 14-3 是处理 363 号问题时的流程图。这个流程图把焦点集中在 support 方法调用的部分。事实上在调用下一个 support 方法之前，它会调用本身的 resolve 方法。

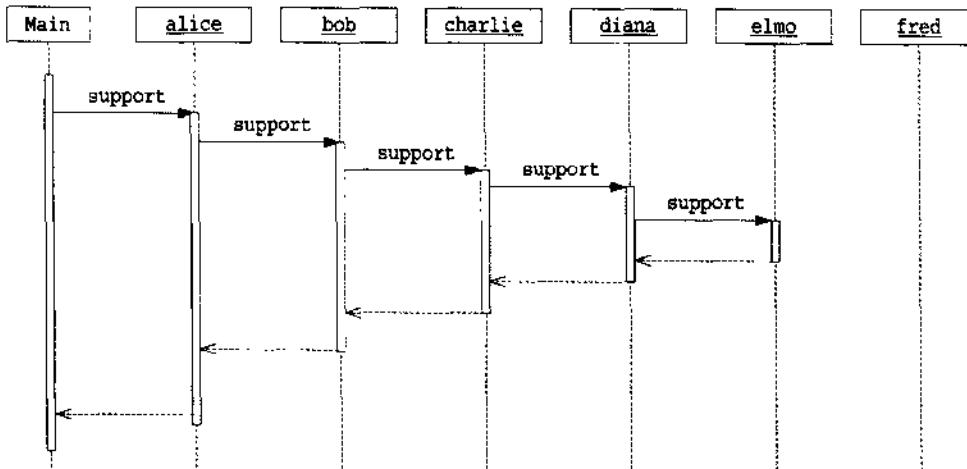


图 14-3 程序示例处理 (Trouble 363) 的流程图

## Chain of Responsibility Pattern 的所有参与者

Chain of Responsibility Pattern 中出现的所有参与者可整理如下。

### ◆ Handler (处理器) 参与者

Handler 是规定处理要求的接口 (API) 的参与者。它会维持住“下一个人”，万

## Chain of Responsibility (职责链) —— 责任转送

一出现自己无法处理的要求时，再转送过去。当然，“下一个人”也是 Handler 参与者。在前面的程序示例中，扮演这个角色的是 Support 类。处理要求的方法则是 support 方法。

### ◆ ConcreteHandler (具体处理器) 参与者

ConcreteHandler 是具体处理要求的参与者。在前面的程序示例中，扮演这个角色的是 NoSupport, LimitSupport, OddSupport, SpecialSupport 这几个类。

### ◆ Client (客户) 参与者

Client 是对第一个 ConcreteHandler 参与者发出要求的参与者。在前面的程序示例中，扮演这个角色的是 Main 类。

Chain of Responsibility Pattern 的类图即如图 14-4 所示。

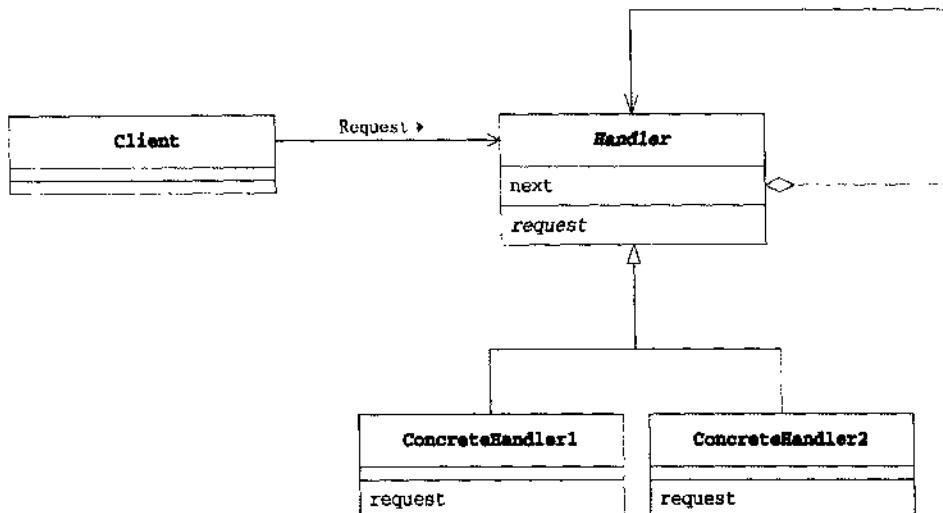


图 14-4 Chain of Responsibility Pattern 的类图

## 扩展自我视野的提示

### 让要求跟处理要求两者间的耦合度降低

Chain of Responsibility Pattern 的重点就在让要求的人 (Client 参与者) 跟处理要求的人 (ConcreteHandler 参与者) 两者间的耦合度降低。Client 参与者直接把要求丢给第一个人，接下来这个要求就会被送往进行连锁处理，由适合的处理器来处理要求。

如果不使用这个 Pattern 的话，就必须采取中央集权式的管理，也就是要由其

## 小结与语言中的应用

中一个人掌握“这个要求该由哪个人处理”的消息。这个消息最好不要交给“发出要求的人”，因为如果发出要求的人还要一五一十了解处理者个别作用能力的话，反而会降低零件的独立性。

为了简洁明了起见，本章的程序示例把最初形成连锁关系的工作交给 Client 参与者的 Main 类。

## 机动性改变链形态

在前面的程序示例中，支持小组从 Alice 到 Fred 的排列顺序都没有改变。但是，程序也有可能需要机动性改变对象（处理要求的 ConcreteHandler 参与者）间的关系。此时，只要像 Chain of Responsibility Pattern 这样利用委让来转送，就能根据状况变化重组 ConcreteHandler 参与者。

万一程序没有利用 Chain of Responsibility Pattern，把“如果是 A 要求，则由甲处理者处理”的对应关系写死了的话，要想在运行程序的时候改成其他的处理者就很困难。

有些窗口系统允许用户自行在窗口上新增组件（按钮或文本输入字段），这时候 Chain of Responsibility Pattern 就能发挥其功效。

## 能更专心在自己的工作岗位上

虽然“转送”这个名词含有浓浓的否定意味，但换个角度来想，这种方式反而让对象“能更专心在自己的工作岗位上”。所有 ConcreteHandler 参与者都只看着自己有能力处理的工作，如果自己处理不了，就当机立断抛出去：“下一关，拜托你了”。如此一来，应该写在各个 ConcreteHandler 参与者里的处理就会锁定是在该 ConcreteHandler 参与者的固定内容。

假设现在故意不使用 Chain of Responsibility，那就要采取“找 1 个出来带头，让它决定要由谁来处理不同的要求”的方法；或者是让所有 ConcreteHandler 参与者都要负责“工作分配”，也就是“如果自己无法处理时，要交给某甲。如果某甲也不行，就交给某乙。如果系统出现乙状况，则交给某丙”的方法。

## 转送会不会导致处理变慢

利用 Chain of Responsibility Pattern 转送要求，寻找能妥善适当处理的参与者，这种做法确实很有弹性，但会不会降低处理的速度呢？

这个顾虑很有道理。有预先确定由谁处理哪个要求，可以让处理者立即动手处理，相比之下，Chain of Responsibility Pattern 的处理速度本来就会稍慢。

不过，这是二选一的取舍问题。如果说要求跟处理者的关系很固定，而且这个程序又非常讲究处理速度的话，可能不使用 Chain of Responsibility Pattern 会比较有利。

## 相关 Pattern

### ◆ Composite Pattern (第 11 章)

Handler 参与者经常会用到 Composite Pattern。

### ◆ Command Pattern (第 22 章)

丢给 Handler 参与者的“要求”有时会用到 Command Pattern。

## 重点回顾

本章所介绍的是将处理要求的对象实例排列成链状，依次检查是否能处理要求的“转送” Pattern——Chain of Responsibility Pattern。

Chain of Responsibility Pattern 经常应用在窗口系统中。各位可以利用练习题思考一下相关的具体示例。

## 练习题

解答请见附录 A

### 问题 1

窗口系统使用 Chain of Responsibility Pattern 的频率很高。

窗口上有按钮、文本框、复选框等多种组件（又称 widget 或控件等）。按下鼠标左键时所产生的事件是如何转送出去？Chain of Responsibility Pattern 中的“next”（转送位置）又出现在哪里呢？

### 问题 2

这里还是窗口系统下的 Chain of Responsibility Pattern 问题。

以图 14-5 的小型对话框为例。如果当前活动组件是在上面的（字体）列表内时，利用↑↓键即可选择列表内的字体。如果活动组件是在（显示调和字体）这个复选框时，按一次↑键，当前活动组件就会跳到（字体）列表，即使再按↓键也不会回到复选框。请试以 Chain of Responsibility Pattern 的概念来说明整个动作关系。



图 14-5 小型对话窗口

### 问题 3

在程序示例的 Support 类（程序 14-2）当中，support 方法设为 public，但 resolve 方法却是 protected。从这个差别可以看出程序员想要表达什么意义吗？

### 问题 4

请试将程序示例中原本以递归方式调用的 support 方法（程序 14-2）改成循环方式。

# 第 15 章

## Facade ( 外观 ) ——单一窗口

## Facade 单一窗口

程序这个东西往往越做越大，许多类彼此间的影响让关系更加错综复杂。因此在使用类时，要确实了解类之间的关系，正确依次调用方法。

利用大型程序进行数据处理时，必须精确控制相关的类。既然如此，就干脆设个处理专用的“窗口”，如此一来不需要个别控制类，只要把要求留给“窗口”即可。

这个“窗口”就是本章要学习的

### Facade Pattern

Facade Pattern 能整理错综复杂的来龙去脉，提供较高级的接口（API）。Facade 参与者让系统外部看到较简单的接口（API）。而且 Facade 参与者还会兼顾系统内部各类型功能和互动关系，以最正确的顺序利用类。

本章以下将介绍这个单一窗口—Facade Pattern（外观）。

## 程序示例

本章的程序示例是一个产生用户 Web 网页的程序。

如果要做出一个 Facade Pattern 的示例，本来应该有许多组合复杂的类。不过为了缩短程序示例的篇幅起见，在此只以 3 个类的简单型系统为例。这个系统中包括利用邮件信箱取得姓名的数据库（Database）、产生 HTML 文件的类（HtmlWriter）以及提供较高级接口（API）的类（PageMaker）（即 Facade 参与者）。

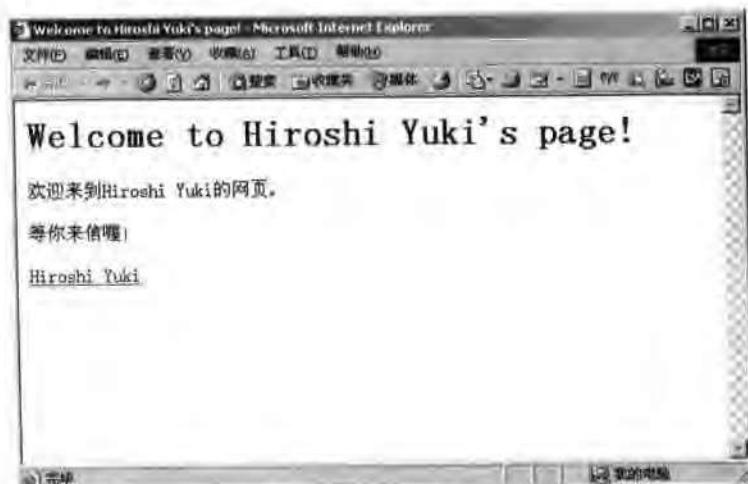


图 15-1 程序示例所产生的 Web 网页。

## Facade (外观) —— 单一窗口

类一览表如表 15-1、UML 类图如图 15-2 所示。

各类的原始文件在图 15-3 所示目录之下。

表 15-1 类一览表

| 程序包       | 名称         | 说明             |
|-----------|------------|----------------|
| pagemaker | Database   | 从邮件信箱取得用户名的类   |
| pagemaker | HtmlWriter | 产生 HTML 文件的类   |
| pagemaker | PageMaker  | 根据邮件信箱产生用户网页的类 |
| 未命名       | Main       | 测试用的类          |

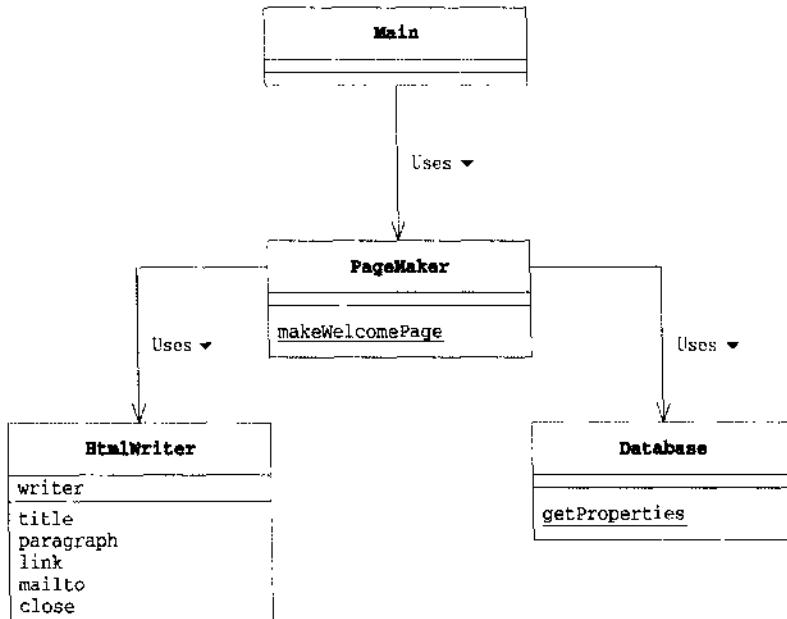


图 15-2 程序示例的类图

Main.java

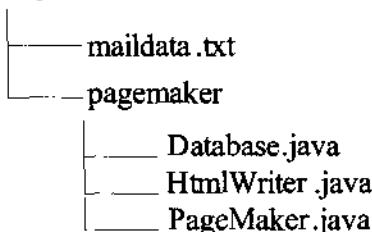


图 15-3 原始文件所在的目录位置

## Database 类

Database 类（程序 15-1）是指定数据库名称（例如：“maildata”）并产生对应 Properties 的类。这个类并不产生对象实例，而是利用 getProperties 这个 static 方法来取得 Properties 的对象实例。数据库示例如程序 15-2 所示。

程序 15-1 Database 类 (Database.java)

```
package pagemaker;

import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

public class Database {
    private Database() { // 声明 private 以免 new 时会产生对象实例
    }
    public static Properties getProperties(String dbname) { // 从数
//据库名称取得 Properties
        String filename = dbname + ".txt";
        Properties prop = new Properties();
        try {
            prop.load(new FileInputStream(filename));
        } catch (IOException e) {
            System.out.println("Warning: " + filename + " is not
found.");
        }
        return prop;
    }
}
```

程序 15-2 数据文件 (maildata.txt)

```
hyuki@hyuki.com=Hiroshi Yuki
hanako@hyuki.com=Hanako Sato
tomura@hyuki.com=Tomura
mamoru@hyuki.com=Mamoru Takahashi
```

## HtmlWriter 类

HtmlWriter 类（程序 15-3）可产生简单的 Web 网页。在产生对象实例时就先把

## Facade(外观)——单一窗口

Writer传递过去，即可将HTML输出到该Writer。

title方法输出标题，paragraph方法输出段落，link方法输出链接，mailto方法输出邮件信箱的链接，而close方法结束输出HTML。

这个类有一个隐性限制：第一个调用的必须是title方法。负责窗口的PageMaker类就写到须遵守此限制。

程序15-3 HtmlWriter类(HtmlWriter.java)

```
package pagemaker;

import java.io.Writer;
import java.io.IOException;

public class HtmlWriter {
    private Writer writer;
    public HtmlWriter(Writer writer) { //构造函数
        this.writer = writer;
    }
    public void title(String title) throws IOException { //输出标题
        writer.write("<html>");
        writer.write("<head>");
        writer.write("<title>" + title + "</title>");
        writer.write("</head>");
        writer.write("<body>\n");
        writer.write("<h1>" + title + "</h1>\n");
    }
    public void paragraph(String msg) throws IOException { //输出段落
        writer.write("<p>" + msg + "</p>\n");
    }
    public void link(String href, String caption) throws IOException {
        //输出链接
        paragraph("<a href=\"" + href + "\">" + caption + "</a>");
    }
    public void mailto(String mailaddr, String username) throws
IOException { //输出邮件信箱
        link("mailto:" + mailaddr, username);
    }
    public void close() throws IOException { //关闭文件
        writer.write("</body>");
        writer.write("</html>\n");
        writer.close();
    }
}
```

```
    )  
}
```

## PageMaker 类

PageMaker 类（程序 15-4）结合 Database 类和 HtmlWriter 类，产生依用户而变的 Web 网页。

此类中只有 makeWelcomePage 定义为 public 的方法。只要把邮件信箱、输出文件名设定到这个方法里，就可以产生 Web 网页。

如果有哪个地方在调用 HtmlWriter 类的方法，就由 PageMaker 类承接过来，让外部只看到 1 个 makeWelcomePage 方法。这就是单一窗口。

程序 15-4 PageMaker 类（PageMaker.java）

```
package pagemaker;  
  
import java.io.FileWriter;  
import java.io.IOException;  
import java.util.Properties;  
  
public class PageMaker {  
    private PageMaker() { // 因为不产生对象实例，故声明 private  
    }  
    public static void makeWelcomePage(String mailaddr, String  
filename) {  
        try {  
            Properties mailprop = Database.getProperties("maildata");  
            String username = mailprop.getProperty(mailaddr);  
            HtmlWriter writer = new HtmlWriter(new FileWriter(filename));  
            writer.title("Welcome to " + username + "'s page!");  
            writer.paragraph("欢迎来到" + username + "的网页。");  
            writer.paragraph("等你来信喔！");  
            writer.mailto(mailaddr, username);  
            writer.close();  
            System.out.println(filename + " is created for " + mailaddr +  
" (" + username + ")");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

## Main 类

Main 类 (程序 15-5) 则利用 pagemaker 包的 PageMaker 类。其内容其实只有这么简短一段叙述:

```
PageMaker.makeWelcomePage("hyuki@hyuki.com", "welcome.html");
```

意思是检查 hyuki@hyuki.com 访客的姓名, 以文件名 welcome.html 产生 Web 网页。

**程序 15-5 Main 类 (Main.java)**

```
import pagemaker.PageMaker;

public class Main {
    public static void main(String[] args) {
        PageMaker.makeWelcomePage("hyuki@hyuki.com",
        "welcome.html");
    }
}
```

```
javac Main.java
java Main
welcome.html is created for hyuki@hyuki.com (Hiroshi Yuki)
```

图 15-4 编译与执行结果

(以下继续以浏览器开启 welcome.html, 即如图 15-5 所示)

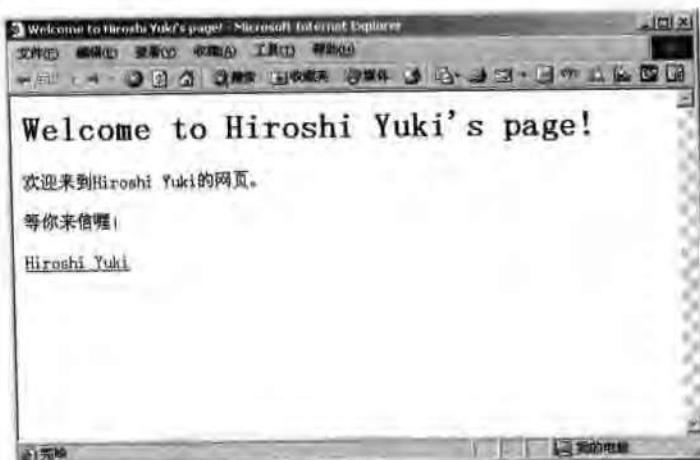


图 15-5 以浏览器开启 welcome.html 的结果

## Facade Pattern 的所有参与者

Facade Pattern 中出现的所有参与者可整理如下。

### ◆ Facade (外观) 参与者

Facade 是构筑成系统的其他参与者的“单一窗口”。Facade 参与者对系统外部提供较高级且单一的接口（API）。在前面的程序示例中，扮演这个角色的是 PageMaker 类。

### ◆ 构筑成系统的其他参与者

其他林林种种的参与者则各司其职，Facade 参与者的存在并不会有任何影响。它们听从 Facade 参与者的调用，但不会反过来调用 Facade 参与者。在前面的程序示例中，扮演这个角色的是 Database 类和 HtmlWriter 类。

### ◆ Client (客户) 参与者

利用 Facade Pattern 的参与者（在 GoF 一书中，Facade Pattern 内并没有 Client 参与者）。在前面的程序示例中，扮演这个角色的是 Main 类。

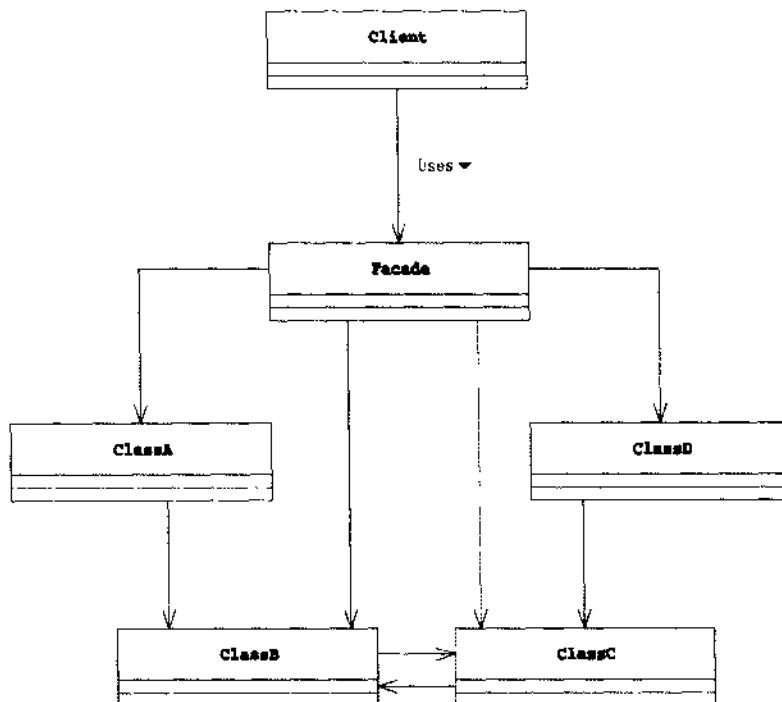


图 15-6 Facade Pattern 的类图

## 扩展自我视野的提示

### Facade 参与者到底在做什么事

Facade 参与者让复杂的内容看起来很简单。什么是“复杂的内容”？躲在背后工作的那些类间的关系和用法相当复杂，Facade 参与者会让你完全遗忘这个令人头痛的复杂问题。

这部分的重点就在于减少接口（API）。看了太多的类和方法，只会让程序员犹豫不知该使用哪一个才对，而且还得注意调用顺序不能搞错。要注意的事情越多，就越容易弄错，这是不变的真理。所以不如把精力放在界面（API）较少的 Facade 参与者，反而比较有效率。

“接口（API）数量较少”这句话也可以解释成“跟外部的耦合度较低”。跟外部的耦合度较低、低度耦合或不紧密的耦合都能提高包（类的集合）的零件重用性。

在设计类时都要考虑到把哪些方法设为 public。如果 public 的方法太多，会造成修改类内部时的负担。字段也是如此，太轻易就把字段设成 public 的话，会让其他类很容易就能参照或修改该字段的内容，间接造成修改类时的负担。

在设计包时也跟类设计一样，需要考虑哪些类应设为 public。类公开过度（即让包之外的部分容易看到），会提高包内部修改的困难度（练习题 15-1 会进一步研究这个部分）。

### Facade Pattern 的递归应用

再来看看比类单位更大的包，应该可以拓宽各位读者的想象空间。

假设现在有数个含有 Facade 参与者的类集合，这时候当然可以新增一个包含所有集合的 Facade 参与者。换句话说，就是递归应用 Facade Pattern 的意思。

如果面对的大型系统有大量的类和包时，在适当的位置应用 Facade Pattern 会让系统使用起来更加便利。

### 程序员拒绝使用 Facade 的理由——敏感的心理因素

业界有一个很微妙的现象：程序员只要了解到程序内部相当复杂，就不想碰 Facade 参与者；或者该说程序员会下意识地避免建立 Facade 参与者。

为什么？或许是资深程序员已经把整个系统都勾勒在脑海里，就算是类之间的关系再纠结复杂也都能乱中有序。也许有心想炫耀自己的能力，等于向其他程序员下战书：“你看得出来吗？”

如果一个程序还需要程序员来指导：“喔，在调用这个类之前要先调用这里。在调用这里的方法之前要先登录到那个类”，就表示程序真的该建立 Facade 参与者才行了。

程度深到要口头说明才能表达清楚的 know how 就应该详实注明在程序代码上，而不是紧握在程序员的手里不放。

## 相关 Pattern

### ◆ Abstract Factory Pattern (第 8 章)

Abstract Factory Pattern 可以视为跟产生对象的复杂作业有关的 Facade Pattern。因为它是提供“只要调用即可产生对象”的接口 (API)。

### ◆ Singleton Pattern (第 5 章)

Facade 参与者有时会设计成 Singleton Pattern。

### ◆ Mediator Pattern (第 16 章)

Facade Pattern 所产生的是让 Facade 参与者单方面利用其他参与者的较高等级接口 (API)。

而在 Mediator Pattern 当中，Mediator 参与者的处理行为比较类似 Colleague 参与者的中间人。可以说 Facade Pattern 是单行道，而 Mediator Pattern 则是双向通车。

## 重点回顾

本章所介绍的是面对复杂型系统扮演单一窗口参与者的 Facade Pattern。

## 练习题

解答请见附录 A

### 问题 1

假设有位程序员在设计类时，为了预留未来扩充和程序改良的空间，决定设成能从 pagemaker 包之外利用类的只有 PageMaker 类。请问：本章的程序示例应如何修改，才能让 pagemaker 包以外的部分不能使用 Database 类和 HtmlWriter 类呢？

### 问题 2

请在 PageMaker 类新增一个 makeLinkPage 方法，这个 makeLinkPage 方法可产生 maildata.txt (程序 15-2) 内用户邮件信箱的相关链接。调用方式即如程序 15-6，

## Facade (外观) —— 单一窗口

产生的相关链接如图 15-8 所示 (输出画面如图 15-9)。

## 程序 15-6 Main 类 (Main.java)

```
import pagemaker.PageMaker;

public class Main {
    public static void main(String[] args) {
        PageMaker.makeLinkPage("linkpage.html");
    }
}
```

```
javac Main.java
java Main
linkpage.html is created
```

图 15-7 编译与执行结果

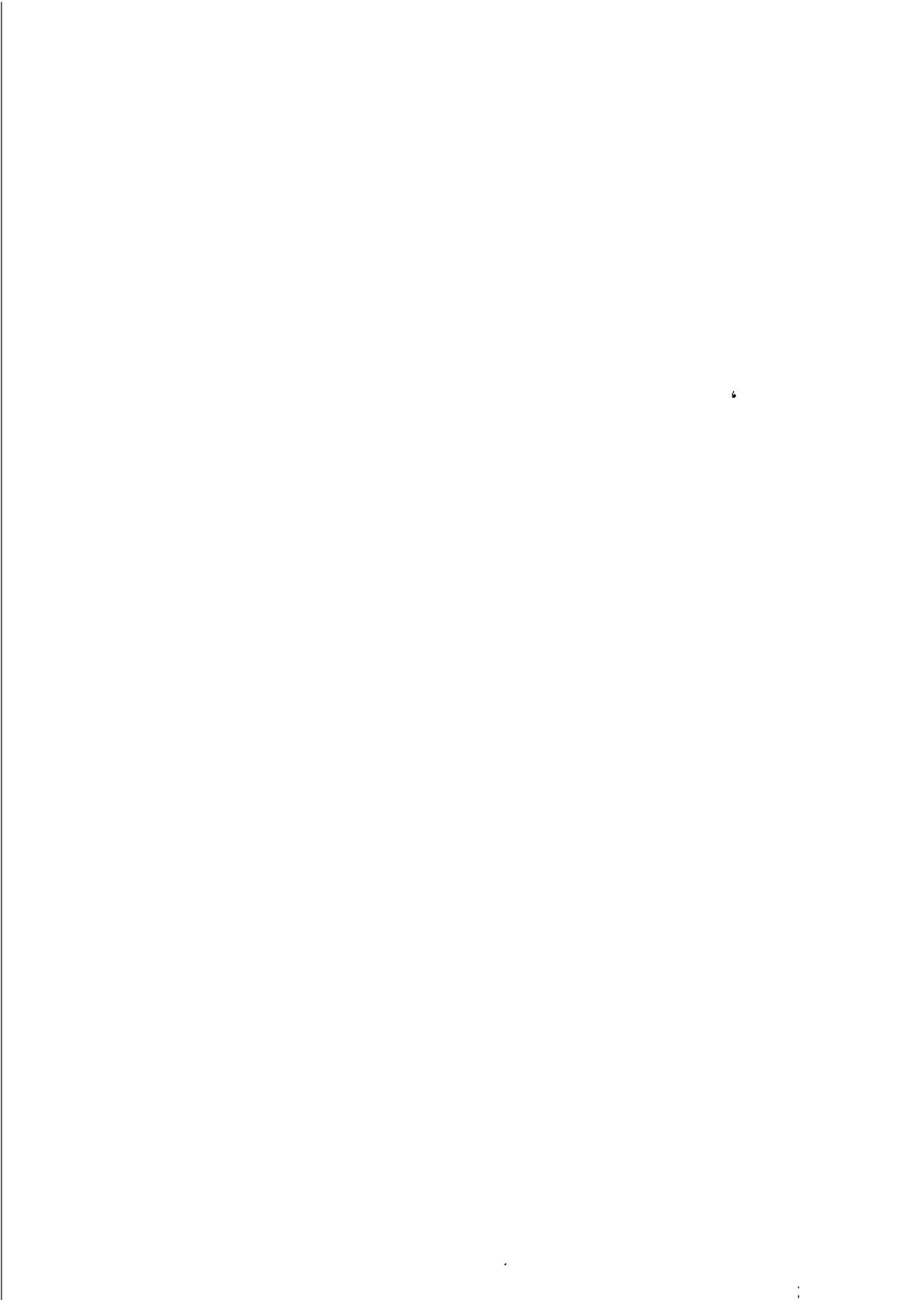
(以下继续以浏览器开启 linkpage.html, 即如图 15-9 所示)

```
<html><head><title>Link page</title></head><body>
<h1>Link page</h1>
<p><a href="mailto:hanako@hyuki.com">Hanako Sato</a></p>
<p><a href="mailto:mamoru@hyuki.com">Mamoru Takahashi</a></p>
<p><a href="mailto:hyuki@hyuki.com">Hiroshi Yuki</a></p>
<p><a href="mailto:tomura@hyuki.com">Tomura</a></p>
</body></html>
```

图 15-8 所产生的 linkpage.html



图 15-9 以浏览器开启 linkpage.html 的结果



# 第 16 章

## Mediator (中介者) ——只要面对一个 顾问

## Mediator Pattern

各位能想象看一个闹哄哄的工作小组吗？一个小组里有 10 个同样立场的成员在共同进行作业，但却各自为政。每个成员都对别人发出指令，让整个作业乱成一团，而且还有干扰其他成员的作业方式，不断发生指令相左的情形。这时候如果有一个重要的“顾问”站出来说：“请各位成员把所有状况汇报给我这个顾问，我会做出整体考虑后发给各位适当的指示。但是我不会插手管各位手上工作的处理细节”。所有成员同意顾问的提议，于是达成共识。

每个成员都只对顾问提出报告，也只有顾问会发出指示给各个成员；成员们彼此也不会去探问目前状况如何，或乱发指示给其他成员。

本章要学习的就是：

### Mediator Pattern

mediator 的英文原意是“调停者”“斡旋人”，但光从中文翻译“调停者”这三个字来看，有点不太容易联想起来。各位不妨把它想象成帮一个小组做事的“顾问”，如果有困难时就告诉顾问，发生什么会影响到整个小组的事也要告诉顾问。对于顾问所提出的要求事项要确实执行。所有小组成员都不可以擅自跟其他成员沟通意见做判断，必须通过顾问进行。而顾问则根据小组成员所提出的报告做整体性判断，对各个成员发出指示。Mediator Pattern 就是这样的设计 Pattern。

在 Mediator Pattern 当中，“顾问”称为 mediator(中介者)、“各个成员”称为 colleague(同事)。colleague 这个单字很容易拼错，而且平时也不常用，在此我们直接采用 GoF 所用的名词。

## 程序示例

请看下面应用 Mediator Pattern 的程序示例，本章的程序是一个“要求输入姓名和密码的系统登录对话框”的 GUI 应用软件。图 16-1 为执行此程序示例时的画面。

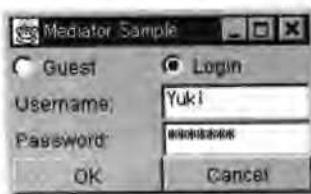


图 16-1 登录对话框

这个对话框的使用规则如下：

- 选择访客登录（Guest）、或用户登录（Login）

## Mediator (中介者) —— 只要面对一个顾问

- 若为用户登录，则输入用户名（Username）和密码（Password）
- 选择登录按（OK）键、放弃登录按（Cancel）键（但本程序示例无法登录任何系统。只要按下任一按键即结束程序）

这个程序看起来很简单嘛，你确定吗？

如果只看上面的语句说明，这个程序好像很简单，不过如果是要让程序能像下面这样运行的话，就有点复杂了。

- 若是选择访客登录，用户名和密码设为“不可使用”，无法输入字符串
- 若是选择用户登录，用户名设为“可供使用”，可输入字符串
- 若用户名的位置未输入任何字符时，密码为“不可使用”
- 只要用户名的位置输入字符，密码为“可供使用”（若为访客登录时，密码当然就是“不可使用”）
- 若用户名和密码这两个位置都输入字符时，则 OK 键为“可供使用”的可按下状态，但若用户名或密码任一个位置是空白时，则 OK 键为“不可使用”的无法按下状态（若为访客登录时，OK 键当然就永远是“可供使用”）
- Cancel 键永远为“可供使用”，是随时均可按下的状态

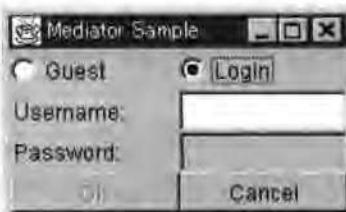


图 16-2 若是选择用户登录，用户名为“可供使用”，但密码为“不可使用”

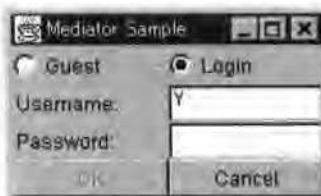


图 16-3 只要用户名的位置输入字符，密码为“可供使用”，但 OK 键为“不可使用”

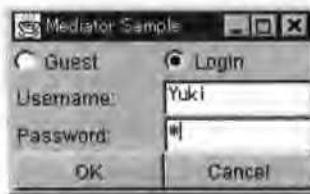


图 16-4 只要密码的位置输入字符，则 OK 键亦为“可供使用”

## 设计模式 —— Java语言中的应用

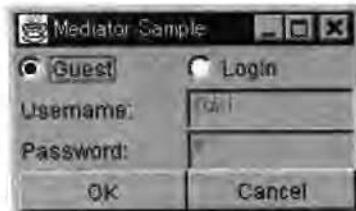


图 16-5 若是选择访客登录，则用户名和密码均为“不可使用”

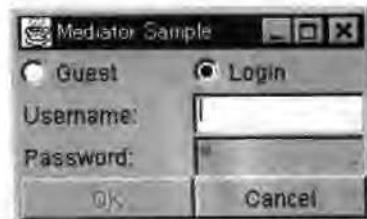


图 16-6 即使已输入密码，若删除用户名时则 OK 键和密码均为“不可使用”

事情看来没那么简单喔。以上这些条件用文字来说明的话，看起来好像很复杂，不过实际操作过对话框应该就能了解原始设计者想要表达的意思。但是，这样的程序应该怎么写比较好呢？

对话框上的选择按钮（选择访客登录或用户登录）、文本框字段（输入用户名和密码）以及按键（OK、Cancel）分属不同的类。如果要把前面所列出的逻辑分散到各个类，这个程序会很不好写。

因为所有对象之间都互有连带关系，这样会变成互相控制对方，很难收拾残局。

“若选择用户登录，则用户名、密码均为有效，但若用户名为空时，则密码不能设为有效。接着，若用户名和密码两部分都输入字符串时，则 OK 键为有效”这个程序代码到底应该写在哪里？写在选择按钮的位置吗？

如果写在选择按钮的位置，用来控制画面输出的那些似是而非的程序代码就要拆开写到各个类去。这样一来，不仅是写程序麻烦，程序出错时也很让人头痛。而且如果将来想修改规格，“新增可供输入邮件信箱的字段”时就会非常麻烦。

像这样需要协调多个对象的时候，就是 Mediator Pattern 大展身手的好时机。不要让对象彼此直接沟通，另设一个“出面帮忙的顾问”，每个对象都只跟这个顾问沟通联络。当然，画面输出控制的逻辑就只要写在顾问里面即可。

虽然花了好几页的篇幅来说明，不过相信各位应该已经掌握到重点。在看后面的程序时，请不要忘记上述的 Mediator 相关重点。

## Mediator（中介者）——只要面对一个顾问

类和接口的一览表如表 16-1，类图和流程图则分别如图 16-7、图 16-8 所示。

表 16-1 类与接口一览表

| 名称                 | 说明                                 |
|--------------------|------------------------------------|
| Mediator           | 决定“顾问”接口（API）的接口                   |
| Colleague          | 决定“成员”接口（API）的接口                   |
| ColleagueButton    | 实现 Colleagues 接口。表示按键的类            |
| ColleagueTextField | 实现 Colleagues 接口。输入文本的类            |
| ColleagueCheckbox  | 实现 Colleagues 接口。表示选择项目（在此为选择按钮）的类 |
| LoginFrame         | 实现 Mediator 接口。表示登录对话框的类           |
| Main               | 测试用的类                              |

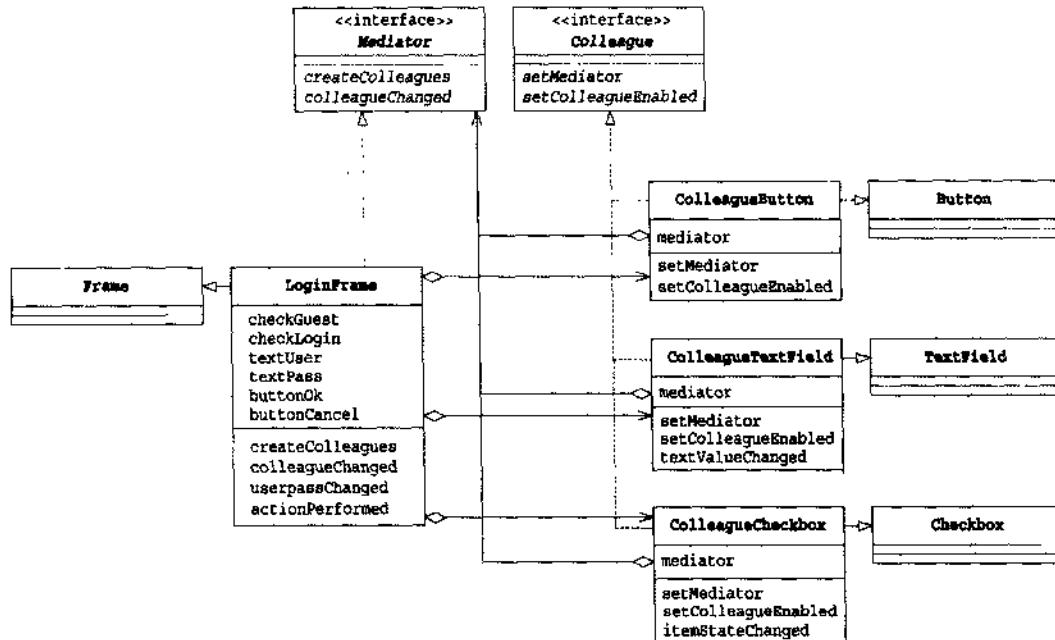


图 16-7 程序示例的类图

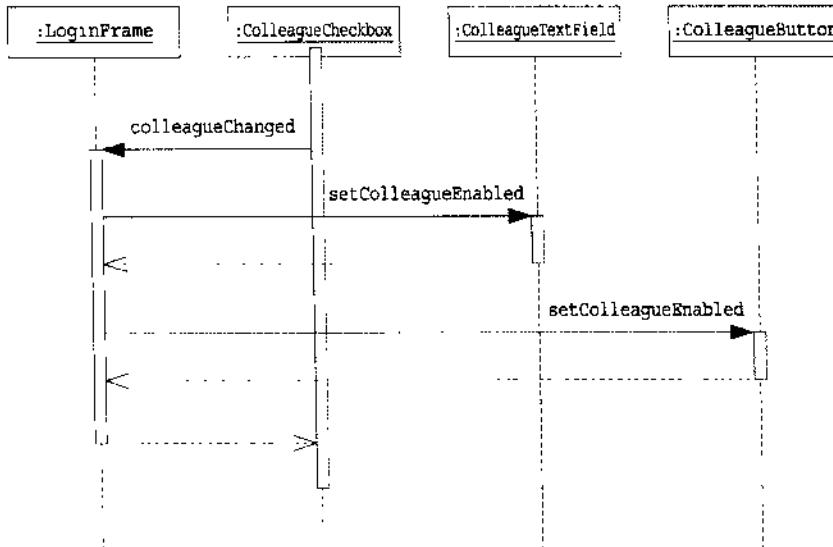


图 16-8 程序示例的流程图

## Mediator 接口

Mediator 接口(程序 16-1)是表示“顾问”的接口。具体的顾问(后面的 `LoginFrame` 类)再实现这个接口。

`createColleagues` 方法就是产生 Mediator 所要管理的所有成员的方法。在本章的程序示例中是以 `createColleagues` 方法产生所有按键、文本框字段等。

`colleagueChanged` 方法是各个成员 `Colleague` 所调用的方法。这个方法相当于向顾问“提出咨询”。参数 `Colleague` 则表示有事要咨询的成员。在程序示例中，当选择按钮或文本框字段的状态发生变化时，都会调用这个方法。

### 程序 16-1 Mediator 接口 (`Mediator.java`)

---

```

public interface Mediator {
    public abstract void createColleagues();
    public abstract void colleagueChanged(Colleague colleague);
}
  
```

---

## Colleague 接口

Colleague 接口(程序 16-2)表示要跟顾问咨询的成员的接口。具体的成员(`ColleagueButton`, `ColleagueTextField`, `ColleagueCheckbox`)实现这个接口。

`setMediator` 方法是实现 Mediator 接口的 `LoginFrame` 类一开始就调用的方法。这

## Mediator (中介者) —— 只要面对一个顾问

个方法是用来表示“记住喔，顾问就是我，我就是顾问”。这个方法的参数所传递的对象实例在以后需要咨询时（即调用 `colleagueChanged` 时）就会用到。

`setColleagueEnabled` 方法相当于顾问发出来的“指示”。若参数 `enabled` 为 `true`，则本身设为“可供使用”；若为 `false`，则本身设为“不可使用”。这个方法是表示它不能自己判断自己应该是“可供使用”还是“不可使用”，必须根据顾问的判断来决定。

另外，`Mediator` 接口和 `Colleague` 接口所使用的方法则因设计的应用软件而异。本章的程序示例是把 `colleagueChanged` 方法放在 `Mediator` 中，`setColleagueEnabled` 方法放在 `Colleague` 中；如果要做更进一步的沟通，需要用到 `Mediator` 参与者和 `Colleague` 参与者时，就需要增加方法的数量。即使同样使用 `Mediator Pattern`，实际的方法还是会因应用软件而有所不同。

程序 16-2 `Colleague` 接口 (`Colleague.java`)

---

```
public interface Colleague {
    public abstract void setMediator(Mediator mediator);
    public abstract void setColleagueEnabled(boolean enabled);
}
```

---

## ColleagueButton 类

`ColleagueButton` 类(程序 16-3)是 `java.awt.Button` 的子类，不过它是实现 `Colleague` 接口后，再跟 `LoginFrame` 类 (`Mediator` 接口) 进行协调。

`mediator` 字段保留 `setMediator` 所传递过来的 `Mediator` 对象(`LoginFrame` 类的对象实例)。`setColleagueEnabled` 方法调用以 Java 的 GUI 所定义的 `setEnabled` 方法，设定有效 / 无效。若为 `setEnabled(true)`，则按键为可按下的状态，若为 `setEnabled(false)` 则为不可按下的状态。

程序 16-3 `ColleagueButton` 类 (`ColleagueButton.java`)

---

```
import java.awt.Button;

public class ColleagueButton extends Button implements Colleague {
    private Mediator mediator;
    public ColleagueButton(String caption) {
        super(caption);
    }
    public void setMediator(Mediator mediator) { // 保留 Mediator
        this.mediator = mediator;
    }
    public void setColleagueEnabled(boolean enabled) { //
```

---

```

Mediator // 指示有效 / 无效
    setEnabled(enabled);
}
}

```

---

## ColleagueTextField 类

ColleagueTextField 类（程序 16-4）是 java.awt.TextField 的子类，实现 Colleague 接口。这个类也实现 java.awt.event.TextListener 接口。因为这样才能用 textValueChanged 方法来捕捉到输入内容是否有变化。

Java 的类无法从两个以上的类再扩充（extends），但可以实现（implements）两个以上的接口。setColleagueEnabled 方法除了调用 setEnabled 方法之外，也会调用 setBackground 方法。这是为了让有效时的背景色彩为白色（white）、无效时变为亮灰色（light gray）。

textValueChanged 方法是 TextListener 接口的方法。当输入内容有变化时就会从 AWT 的框架调用这个方法。程序示例中的 textValueChanged 方法调用 colleagueChanged 方法，对顾问说“对不起，向您报告我的字符串内容发生变化了。”。参数所传递出去的 this 就是有变化的 ColleagueTextField 的对象实例。顾问根据这个参数即可判断“喔，原来有变化的是这个成员”。

**程序 16-4 ColleagueTextField 类（ColleagueTextField.java）**

```

import java.awt.TextField;
import java.awt.Color;
import java.awt.event.TextListener;
import java.awt.event.TextEvent;

public class ColleagueTextField extends TextField implements
TextListener, Colleague{
    private Mediator mediator;
    public ColleagueTextField(String text, int columns) { // 构
//造函数
        super(text, columns);
    }
    public void setMediator(Mediator mediator) { // 保留 Mediator
        this.mediator = mediator;
    }
    public void setColleagueEnabled(boolean enabled) { // Mediator
//指示有效 / 无效
}
}

```

## Mediator（中介者）——只要面对一个顾问

```

       setEnabled(enabled);
       setBackground(enabled ? Color.white : Color.lightGray);
    }

    public void textValueChanged(TextEvent e) {      // 若字符串有变
//化，则通知 Mediator
        mediator.colleagueChanged(this);
    }
}

```

---

**ColleagueCheckbox 类**

ColleagueCheckbox 类（程序 16-5）是 java.awt.Checkbox 类的子类。在本章的程序示例中，被用来作为选择按钮而非复选框（利用 CheckboxGroup）。

这个类也会实现 java.awt.event.ItemListener 接口，因为要用 itemStateChanged 方法来捕捉选择按钮的状态变化。

**程序 16-5 ColleagueCheckbox 类（ColleagueCheckbox.java）**

```

import java.awt.Checkbox;
import java.awt.CheckboxGroup;
import java.awt.event.ItemListener;
import java.awt.event.ItemEvent;

public class ColleagueCheckbox extends Checkbox implements
ItemListener, Colleague {
    private Mediator mediator;
    public ColleagueCheckbox(String caption, CheckboxGroup group,
boolean state) {
        // 构造函数
        super(caption, group, state);
    }
    public void setMediator(Mediator mediator) { // 保留 Mediator
        this.mediator = mediator;
    }
    public void setColleagueEnabled(boolean enabled) { // Mediator
// 指示有效 / 无效
        setEnabled(enabled);
    }
    public void itemStateChanged(ItemEvent e) { // 若状态有变，通知
Mediator

```

```

        mediator.colleagueChanged(this);
    }
}

```

## LoginFrame 类

转了这么久，终于到了“顾问”的类。LoginFrame 类（程序 16-6）是 java.awt.Frame（建立 GUI 的类）的子类，实现 Mediator 接口。

因为 Java 的 AWT 框架解说已经超过本书设定的范围，因此这里只抓几个重点稍做说明。

构造函数中会进行下列几项工作：

- 设定背景色彩
- 设定页面编排（子窗口为竖 4 个 × 横 2 个）
- 以 createColleagues 方法产生 Colleague
- Colleague 的配置
- 设定初始状态
- 输出到画面上

利用 createColleagues 方法产生这个对话框的 Colleague，并保留在字段。然后对各个 Colleague 调用 setMediator，告诉它们“你们的顾问就是我，有什么问题要告诉我”。另外还要利用 createColleagues 方法设定各个 Listener。这是为了能适当的从 AWT 的框架调用出各个 Listener。

这个程序中最重要的方法是 LoginFrame 类的 colleagueChanged 方法。在这个方法中会进行“设定输出到画面上为有效 / 无效的复杂处理”。请回想一下前面所学过的 ColleagueButton、ColleagueTextField、ColleagueCheckbox 这几个类。它们都可设定本身有效 / 无效的方法，但是并没有逻辑指定什么时候要设有效 / 无效。如果每个类只要一声“顾问，麻烦你了”就能调用 colleagueChanged 方法的话，那不就糟了？换句话说，所有 Colleague 的咨询都要集合到这个 colleagueChanged 方法来。

实际上判断传递给 colleagueChanged 的参数的 Colleague 究竟是谁，只要用 == 去比较 LoginFrame 所保留的字段和参数即可，不需要再用 instanceof 等一个个去检查类。就是为了这个操作，前面才特地用 createColleagues 来掌握所有 Colleague。接下来只要再利用 getState 方法（取得选择按钮状态）、getText 方法（取得文字方块字段的字符串）等，把前面那些复杂条件一一写成程序即可。这里调用 userpassChanged 方法作为共享的处理。userpassChanged 方法则是只在内部使用的 private 方法。

**程序 16-6 LoginFrame 类（LoginFrame.java）**

```

import java.awt.Frame;
import java.awt.Label;

```

## Mediator（中介者）——只要面对一个顾问

```
import java.awt.Color;
import java.awt.CheckboxGroup;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class LoginFrame extends Frame implements ActionListener,
Mediator {
    private ColleagueCheckbox checkGuest;
    private ColleagueCheckbox checkLogin;
    private ColleagueTextField textUser;
    private ColleagueTextField textPass;
    private ColleagueButton buttonOk;
    private ColleagueButton buttonCancel;

    // 构造函数
    // 产生并配置好 Colleague 之后再输出到画面上
    public LoginFrame(String title) {
        super(title);
        setBackground(Color.lightGray);
        // 使用页面编排，产生 4×2 的画面
        setLayout(new GridLayout(4, 2));
        // 产生 Colleague
        createColleagues();
        // 配置
        add(checkGuest);
        add(checkLogin);
        add(new Label("Username:"));
        add(textUser);
        add(new Label("Password:"));
        add(textPass);
        add(buttonOk);
        add(buttonCancel);
        // 有效 / 无效的初始设定
        colleagueChanged(checkGuest);
        // 输出到画面上
        pack();
        show();
    }

    // 产生 Colleague
    public void createColleagues() {
        // 产生
```

## 观察者模式实现

```

CheckboxGroup g = new CheckboxGroup();
checkGuest = new ColleagueCheckbox("Guest", g, true);
checkLogin = new ColleagueCheckbox("Login", g, false);
textUser = new ColleagueTextField("", 10);
textPass = new ColleagueTextField("", 10);
textPass.setEchoChar('*');
buttonOk = new ColleagueButton("OK");
buttonCancel = new ColleagueButton("Cancel");
// Mediator 的部分
checkGuest.setMediator(this);
checkLogin.setMediator(this);
textUser.setMediator(this);
textPass.setMediator(this);
buttonOk.setMediator(this);
buttonCancel.setMediator(this);
// Listener 的部分
checkGuest.addItemListener(checkGuest);
checkLogin.addItemListener(checkLogin);
textUser.addTextListener(textUser);
textPass.addTextListener(textPass);
buttonOk.addActionListener(this);
buttonCancel.addActionListener(this);
}

// 根据 Colleague 的消息，判断各 Colleague 的有效 / 无效
public void colleagueChanged(Colleague c) {
    if (c == checkGuest || c == checkLogin) {
        if (checkGuest.getState()) {           // Guest 模式
            textUser.setColleagueEnabled(false);
            textPass.setColleagueEnabled(false);
            buttonOk.setColleagueEnabled(true);
        } else {                            // Login 模式
            textUser.setColleagueEnabled(true);
            userpassChanged();
        }
    } else if (c == textUser || c == textPass) {
        userpassChanged();
    } else {
        System.out.println("colleagueChanged:unknown colleague
= " + c);
    }
}

```

## Mediator (中介者) —— 只要面对一个顾问

```
// textUser 或 textPass 有变动
// 判断各 Colleague 的有效 / 无效
private void userpassChanged() {
    if (textUser.getText().length() > 0) {
        textPass.setColleagueEnabled(true);
        if (textPass.getText().length() > 0) {
            buttonOk.setColleagueEnabled(true);
        } else {
            buttonOk.setColleagueEnabled(false);
        }
    } else {
        textPass.setColleagueEnabled(false);
        buttonOk.setColleagueEnabled(false);
    }
}
public void actionPerformed(ActionEvent e) {
    System.out.println(" " + e);
    System.exit(0);
}
```

---

## Main 类

Main 类（程序 16-7）产生 LoginFrame 的对象实例。虽然 main 方法已经结束，但 LoginFrame 的对象实例还保留在 AWT 的框架这边。

**程序 16-7 Main 类 (Main.java)**

```
public class Main {
    static public void main(String args[]) {
        new LoginFrame("Mediator Sample");
    }
}
```

---

## Mediator Pattern 的所有参与者

Mediator Pattern 的所有参与者可整理如下。

### ◆ Mediator (中介者) 参与者

Mediator 参与者是跟 Colleague 参与者进行通信，规定调整的接口（API）。在前面的程序示例中，扮演这个角色的是 Mediator 接口。

### ◆ ConcreteMediator (具体中介者) 参与者

ConcreteMediator 参与者是实现 Mediator 参与者的接口 (API)，进行实际的调整。在前面的程序示例中，扮演这个角色的是 LoginFrame 类。

### ◆ Colleague (同事) 参与者

Colleague 参与者是规定与 Mediator 参与者沟通的接口 (API)。在前面的程序示例中，扮演这个角色的是 Colleague 接口。

### ◆ ConcreteColleague (具体同事) 的参与者

ConcreteColleague 参与者是实现 Colleague 参与者的接口 (API)。在前面的程序示例中，扮演这个角色的是 ColleagueButton、ColleagueTextField、ColleagueCheckbox 等几个类。

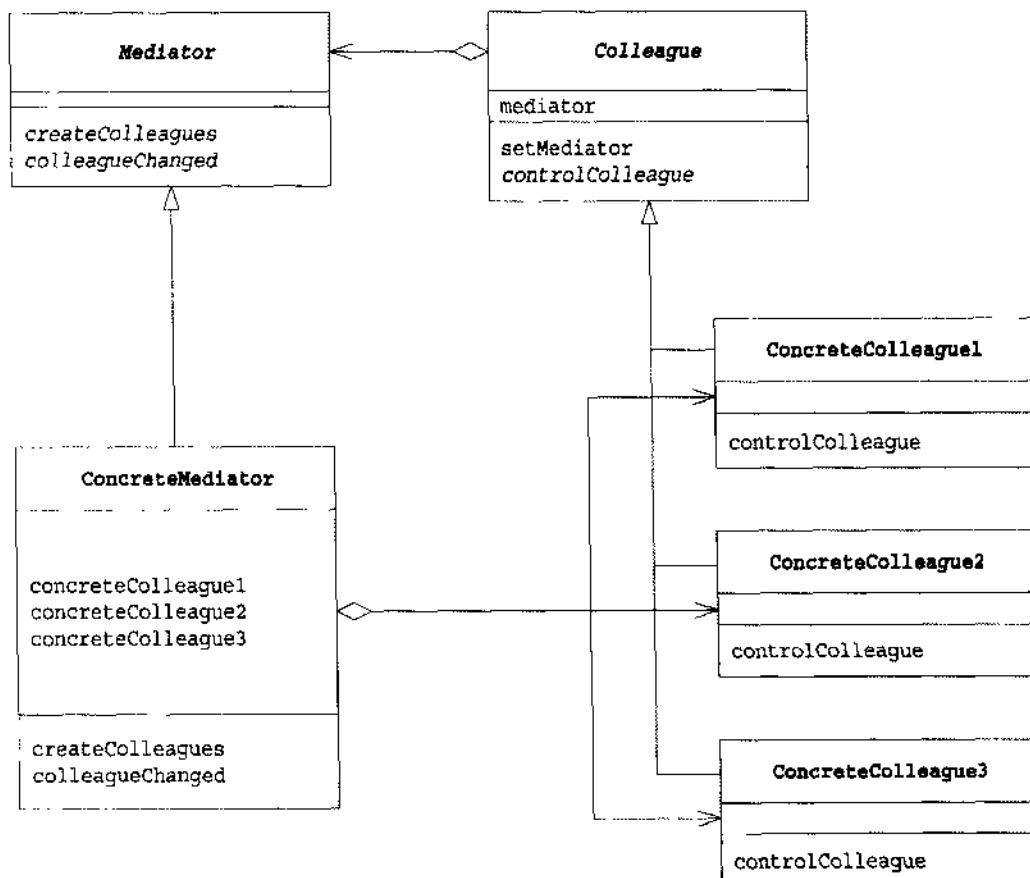


图 16-9 Mediator Pattern 的类图

## 扩展自我视野的提示

### 当分散变成灾难时

程序示例的 LoginFrame 当中的 colleagueChanged 方法（p.226）稍微有点复杂。万一要修改规格的时候，这么复杂的方法中不是会产生 bug 吗？不敢说没有这个可能，不过它并不会是个问题。即使 colleagueChanged 方法里面有 bug，因为跟画面输出的有效 / 无效有关的逻辑只有这里有而已，所以只要对这个部分进行除错操作即可。

请各位想想看，如果逻辑被分散在 ColleagueButton、ColleagueTextField、ColleagueCheckbox 的话，对于写程序、除错、甚至修改程序不都是一大负担吗？

对象导向通常都会把处理分散出去，避免集中在同一个位置；换句话说，就是要达到“分割统合”问题的目的。但单就本章的程序示例而言，硬把处理分散到各个类并不见得有多聪明。如果把该分散的部分分散到各个类，却连该集中的也分散出去的话，费尽心力所做的类区分反倒成了灾难之源。

### 通讯路径数量暴增

假设现在有 A、B 两个对象实例，两者间互相有通信（彼此调用方法）。此时，通讯路径有 2 条：A→B 和 A←B。如果有三个对象实例 A、B、C 时，则路径数量会增加到 6 条：A→B, A←B, B→C, B←C, C→A, C←A。同理可推，4 个对象实例会有 12 条通讯路径、5 个对象实例就有 20 条、6 个对象实例甚至高达 30 条。当一个程序内有太多相同立场的对象实例时，它们彼此间的通信会提高程序的复杂度。

如果对象实例数量不多，这个问题还不大。但如果只知沿用最初的设计方式，对象实例逐渐增加时，就会漏洞百出。

### 哪些可以再利用

虽然 ConcreteColleague 参与者很容易再利用，但 ConcreteMediator 参与者却不太容易再利用。

举例来说，假设现在除了登录对话框之外，还要另建一个新的对话框。此时，这个新的对话框也可以再利用 ColleagueButton、ColleagueTextField、ColleagueCheckbox（即 ConcreteColleague 参与者）。因为 ConcreteColleague 参与者内并没有必须在特殊对话框下才会有效的程序代码。

与应用软件信赖性强的程序代码都放在 LoginFrame 类（即 ConcreteMediator 参与者）内。与应用软件的信赖性强就等于再利用性较低的意思，所以 LoginFrame 类比较不容易直接再利用到其他对话框。

## 相关 Pattern

### ◆ *Facade Pattern* (第 15 章)

在 Mediator Pattern 中，Mediator 参与者扮演 Colleague 参与者的中介者。

在 Facade Pattern 中，Facade 参与者是单向径行利用其他参与者产生较高等级的接口（API）。Mediator 属于双向通信，而 Facade 则可算是单向传达。

### ◆ *Observer Pattern* (第 17 章)

Mediator 参与者和 Colleague 参与者的通信有时会使用 Observer Pattern。

## 重点回顾

本章的学习重点是 Mediator Pattern，这里出现一个让众人依赖的顾问参与者。Mediator Pattern 舍弃关系错综复杂的对象间通信，而改以把信息集中到 Mediator 参与者来整理所有的处理。这个 Pattern 对 GUI 所产生的效果更好。

## 练习题

解答请见附录 A

### 问题 1

请将程序示例修改成满足“当用户登录时，须用户名和密码两者输入字符均超过 4 个字符，OK 键开始为可供使用”的规格。请想想该修改哪些类。

### 问题 2

请注意程序示例中的 ColleagueButton（程序 16-3）、ColleagueTextField（程序 16-4）和 ColleagueCheckbox（程序 16-5），这几个类都有 mediator 字段。而且 setMediator 的内容也相同。为了让整个程序看起来更简单一点，可不可以把 Colleague 接口内加入 mediator 字段，让它实现 setMediator 方法？

# 第 17 章

## Observer (观察者) ——通知状态变化

## Observer Pattern

本章的学习内容是

Observer Pattern

observer 是观察 (observe) 的人，也就是“观察者”的意思。

当被 Observer Pattern 列入观察名单的状态发生变化，就会通知观察者。在写一些跟状态变化有关的处理时，Observer Pattern 是很好用的工具。

## 程序示例

请看下面应用 Observer Pattern 的程序示例。这个程序示例比较简单，只是要观察者观察产生多个数值的对象，然后输出该值。不过输出方式因观察者而异，DigitObserver 以数字表示数值，GraphObserver 则以简易长条图表示数值。

表 17-1 类与接口一览表

| 名称                    | 说明           |
|-----------------------|--------------|
| Observer              | 表示观察者的接口     |
| NumberGenerator       | 表示产生数值对象的抽象类 |
| RandomNumberGenerator | 产生随机数的类      |
| DigitObserver         | 以数字表示数值的类    |
| GraphObserver         | 以简易长条图表示数值的类 |
| Main                  | 测试用的类        |

## Observer 接口

Observer 接口（程序 17-1）是表示“观察者”的接口。具体的观察者会实现此接口。请注意，这里的 Observer 接口是专为本章的程序示例所设计，跟 JDK 的类函式库中的 java.util.Observer 截然不同。两者的差异请参见本章的“补充说明：java.util.Observer 接口”一节。

调用 update 方法的是产生数值的 NumberGenerator（generator 是“产生器”“产生设备”的意思）。update 方法是 NumberGenerator 用来告诉 Observer 说“我的内容已经更新过了，请你也更新你的输出内容”的方法。

## Observer (观察者) —— 通知状态变化

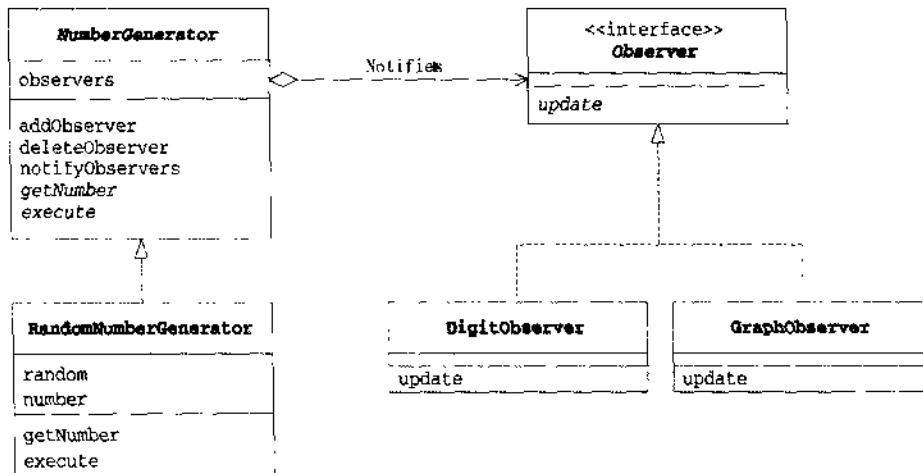


图 17-1 程序示例的类图

## 程序 17-1 Observer 接口 (Observer.java)

```

public interface Observer {
    public abstract void update(NumberGenerator generator);
}
  
```

## NumberGenerator 类

NumberGenerator 类（程序 17-2）是产生数值的抽象类。实际产生数值（execute 方法）和取得数值的部分（getNumber 方法）则要等子类进行实现，属于抽象方法（subclass responsibility）。

observers 字段是存储观察 NumberGenerator 的 Observer 的字段。

addObserver 是新增 Observer 的方法，deleteObserver 是删除 Observer 的方法。

notifyObservers 方法是告诉所有 Observer 说“我的内容已经更新过了，请你也更新你的输出内容”。这个方法是一个调用 observers 内 Observer 的 update 方法。

## 程序 17-2 NumberGenerator 类 (NumberGenerator.java)

```

import java.util.Vector;
import java.util.Iterator;

public abstract class NumberGenerator {
    private Vector observers = new Vector(); // 存储 Observer
    public void addObserver(Observer observer) { // 新增 Observer
  
```

## 观察者模式设计

```

        observers.add(observer);
    }
    public void deleteObserver(Observer observer) { // 删除 Observer
        observers.remove(observer);
    }
    public void notifyObservers() { // 通知 Observer
        Iterator it = observers.iterator();
        while (it.hasNext()) {
            Observer o = (Observer)it.next();
            o.update(this);
        }
    }
    public abstract int getNumber(); // 取得数值
    public abstract void execute(); // 产生数值
}

```

---

**RandomNumberGenerator 类**

RandomNumberGenerator 类（程序 17-3）是 NumberGenerator 的子类，用来产生随机数。

random 字段是存储 java.util.Random 类的对象实例（即随机数产生器），number 字段存储目前的随机数值。

getNumber 方法返回 number 字段值。

execute 方法是产生 20 个随机数（0~49 之间的整数），每次产生时都会利用 notifyObservers 通知观察者。这里的 nextInt 方法是 java.util.Random 类的方法，以随机数的方式将大于 0、但小于参数指定数值的整数设为返回值。

**程序 17-3 RandomNumberGenerator 类 (RandomNumberGenerator.java)**

```

import java.util.Random;

public class RandomNumberGenerator extends NumberGenerator {
    private Random random = new Random(); // 随机数产生器
    private int number; // 目前数值
    public int getNumber() { // 取得数值
        return number;
    }
    public void execute() {
        for (int i = 0; i < 20; i++) {

```

**Observer (观察者) —— 通知状态变化**

```

        number = random.nextInt(50);
        notifyObservers();
    }
}
}
}

```

---

**DigitObserver 类**

**DigitObserver** 类（程序 17-4）是实现 **Observer** 接口的类，以“数字”来表示观察到的数值。它利用 **update** 方法中，参数 **NumberGenerator** 的 **getNumber** 方法取得并输出数值。为了让各位可以更清楚看出画面输出的样子，故特地使用 **Thread.sleep** 降低速度。

**程序 17-4 DigitObserver 类 (DigitObserver.java)**

```

public class DigitObserver implements Observer {
    public void update(NumberGenerator generator) {
        System.out.println("DigitObserver:" + generator.getNumber());
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
        }
    }
}

```

---

**GraphObserver 类**

**GraphObserver** 类（程序 17-5）也是实现 **Observer** 接口的类。这个类是以“简易长条图”（如\*\*\*\*\*）来表示观察到的数值。

**程序 17-5 GraphObserver 类 (GraphObserver.java)**

```

public class GraphObserver implements Observer {
    public void update(NumberGenerator generator) {
        System.out.print("GraphObserver:");
        int count = generator.getNumber();
        for (int i = 0; i < count; i++) {
            System.out.print("*");
        }
        System.out.println("");
    }
}

```

```

        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
        }
    }
}

```

---

## Main 类

Main 类（程序 17-6）产生 1 个 RandomNumberGenerator 的对象实例和 2 个观察者。observer1 是 DigitObserver 的对象实例，observer2 是 GraphObserver 的对象实例。先利用 addObserver 方法登录观察者之后，再使用 generator.execute 产生数值。

**程序 17-6 Main 类 (Main.java)**

```

public class Main {
    public static void main(String[] args) {
        NumberGenerator generator = new RandomNumberGenerator();
        Observer observer1 = new DigitObserver();
        Observer observer2 = new GraphObserver();
        generator.addObserver(observer1);
        generator.addObserver(observer2);
        generator.execute();
    }
}

```

---

```

DigitObserver:44
GraphObserver:*****
DigitObserver:45
GraphObserver:*****
DigitObserver:43
GraphObserver:*****
DigitObserver:9
GraphObserver:*****
DigitObserver:44
GraphObserver:*****
DigitObserver:39
GraphObserver:*****
DigitObserver:1

```

## Observer (观察者) —— 通知状态变化

```

GraphObserver:*
DigitObserver:17
GraphObserver:*****
DigitObserver:41
GraphObserver:*****
DigitObserver:5
GraphObserver:****
DigitObserver:1
GraphObserver:*
DigitObserver:44
GraphObserver:*****
DigitObserver:15
GraphObserver:*****
(以下省略)...

```

图 17-2 执行结果(每次都会因随机数产生而有不同结果)

## Observer Pattern 的所有参与者

Observer Pattern 的所有参与者可整理如下。

### ◆ Subject (目标) 参与者

Subject 参与者表示“被观察的一方（目标）”。Subject 参与者具有登录或删除 Observer 参与者（观察者）的方法。另外也有宣告了“取得目前状态”的方法。在本章的程序示例中，扮演这个角色的是 NumberGenerator 类。

### ◆ ConcreteSubject (具体目标) 参与者

ConcreteSubject 参与者是表示实际“被观察的一方（目标）”的参与者。一旦状态有变化，就会立即通知已登录的 Observer 参与者。在本章的程序示例中，扮演这个角色的是 RandomNumberGenerator 类。

### ◆ Observer (观察者) 参与者

Observer 参与者是被 Subject 参与者通知“状态有变化”的参与者。通知的方法是 update。在本章的程序示例中，扮演这个角色的是 Observer 接口。

### ◆ ConcreteObserver (具体观察者) 参与者

ConcreteObserver 参与者是实际的 Observer。一调用 update 方法时，即可从该方法取得 Subject 参与者的当前状态。在本章的程序示例中，扮演这个角色的是

DigitObserver 类和 GraphObserver 类。

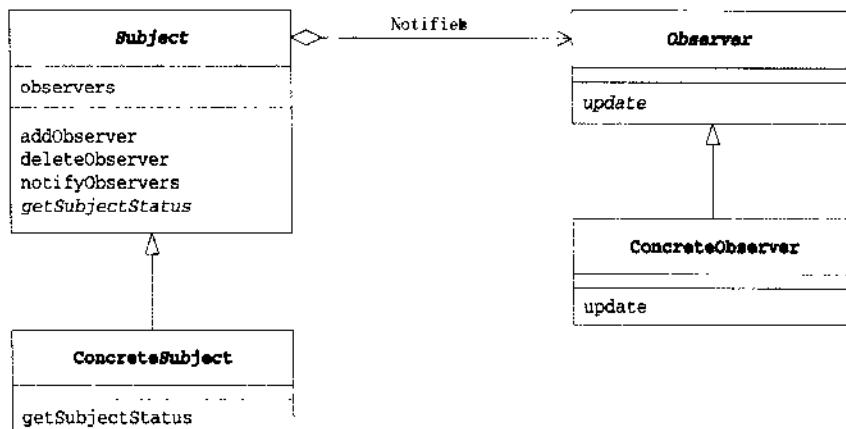


图 17-3 Observer Pattern 图

## 扩展自我视野的提示

### 这里也可对换

让类变成可供再利用的零件也是设计 Pattern 的目的之一。

在 Observer Pattern 里有具有状态的 ConcreteSubject 参与者和接收状态变化通知的 ConcreteObserver 参与者。而居中连接这两个参与者的则是 Subject 参与者和 Observer 参与者（接口（API））。

RandomNumberGenerator 类并不知道，甚至也不在乎现在正在观察自己的（通知的对象）究竟是 DigitObserver 类的对象，还是 GraphObserver 类的对象。但是存储在 observers 字段的对象实例知道实现 Observer 接口。因为这些对象实例是以 addObserver 新增而来，所以一定会实现 Observer 接口，也绝对能调用 update 方法。

而 DigitObserver 类也不会去注意自己所观察的是 RandomNumberGenerator 类的对象实例还是其他 XXXXNumberGenerator 类的对象实例。不过仍然知道这是 NumberGenerator 的子类的对象实例，而且具有 getNumber 方法。

相信从头看到这里的读者们一定已经发现到上述现象在不同的 Pattern 里也出现过好几次。

只要设计如下，就能直接把实际类的部分对换过来。

- 先利用抽象类和接口从实际类抽出抽象方法
- 当以参数传递对象实例或将对象实例存储在字段时，把它设成抽象类或接口的类型，不要写成实际类的类型

## Observer 的顺序

Subject 参与者里登录了好几个 Observer 参与者。本章程序示例的 `notifyObservers` 方法是优先调用最早登录的 Observer 的 `update` 方法。

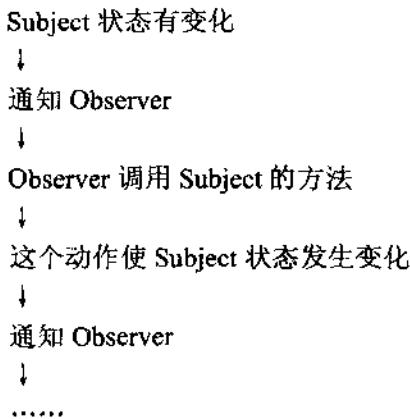
通常在设计 `ConcreteObserver` 参与者的类时，必须注意当 `update` 方法调用顺序改变时也不会发生问题。例如 `GraphObserver` 的 `update` 必须排在调用 `DigitObserver` 的 `update` 之后，否则就不能正常动作，这种事情是绝对不可以发生。其实只要确保各个类的独立性，就不会发生依赖关系的混乱问题。

但是，遇到下节的情况时要特别注意。

## Observer 的行为影响 Subject 时

本章的程序示例是让 `RandomNumberGenerator` 在自己的位置产生数据并调用 `update` 方法，不过一般的 Observer Pattern 可能会把 Subject 参与者调用 `update` 方法的触发点放在其他类。以图形用户接口（GUI）为例，它就是以用户按下按钮的事件作为调用 `update` 方法的触发点。

话又说回来，Subject 参与者调用 `update` 方法的触发点有时就是 Observer 参与者本身。在这种情形之下，设计时要特别注意，否则可能会变成永远在调用方法；也就是说：



如果想要预防这个问题，只要在 Observer 参与者加一个表示“目前是否有 Subject 参与者的通知”的标志变量即可。

## 如何处理更新的提示

`NumberGenerator` 利用 `update` 方法对 Observer 通知“更新了”。`update` 方法的参数

## Observer 模型中的应用

也只有执行调用的 NumberGenerator 的对象实例。Observer 必须在 update 方法当中调用 getNumber，取得必要的信息。

不过如果光就本章的程序示例而言，update 方法的参数也可以是已经更新过的数值。即不要写成：

```
void update(NumberGenerator generator);          (1)
```

而写成：

```
void update(NumberGenerator generator, int number);    (2)
```

或再简化成：

```
void update(int number);                         (3)
```

(1) 是只传递 Subject 参与者。如有需要时，Observer 才从 Subject 参与者得到信息。

(2) 除了 Subject 参与者之外还传递了提示（在此为所有更新信息）。如此一来，Observer 就不需要再从 Subject 参与者取得信息。但是，传递这种提示会让 Subject 参与者去了解 Observer 参与者的处理内容。

如果你将来遇到的程序比本章的程序示例更复杂时，要让 Subject 参与者知道 Observer 参与者真正需要的信息反而会是一大负担。就以传送上次调用 update 方法之后到现在为止所更新的信息为例，此时 Subject 参与者会需要做差分计算。因此，程序员必须思考程序本身究竟有多复杂，才能判断到底该给 update 方法多少提示。

(3) 则是(2)的简化版，连 Subject 参与者都省略掉。(3)的写法在本章的程序示例还能正常运行，但若遇到用 1 个 Observer 参与者观察两个以上的 Subject 参与者时就会失灵，因为它不知道传递过来的数值究竟是哪个 Subject 参与者的信息。

## “通知”的意义重于“观察”

observer 这个名词的原意是“观察者”，但实际上 Observer 参与者并不是主动的“观察”，而是被动等待 Subject 参与者的“通知”。Observer Pattern 又称为 Publish-Subscribe Pattern。publish（发行）和 subscribe（订阅）的表达可能比较贴切一点吧。

## Model/View/Controller (MVC)

各位听过 Model/View/Controller (MVC) 这个名词吗？在 MVC 中 Model 跟 View 的关系正好与 Observer Pattern 的 Subject 参与者和 Observer 参与者两者的关系相呼应。Model 是指操作“不因输出格式而改变的内部模型”的部分，而 View 则是管理 Model “看起来的模样”的部分。通常 1 个 Model 会对应两个以上的 View。Smalltalk 也使用到 MVC。Windows 的 MFC (Microsoft Foundation Class) 也应用它来做文件 / 预览打印的切换。

## Observer（观察者）——通知状态变化

**补充说明：java.util.Observer 接口**

JDK 中的 `java.util.Observer` 接口和 `java.util.Observable` 类是一种 Observer Pattern。`java.util.Observer` 接口具有以下的方法：

```
public void update(Observable obj, Object arg)
```

`update` 方法传递给参数的是：

- `Observable` 类的对象实例——被观察的一方（Subject 参与者）
- `Object` 类的对象实例——附加信息

这个格式跟“如何处理更新的提示”一节中的（2）很像。

也许有些读者会很高兴：“喔，原来 JDK 也有 Observer Pattern。那干脆就用这个好了”。

可是 `java.util.Observer` 接口和 `java.util.Observable` 类不见得有那么好用。理由很简单，因为 `java.util.Observer` 接口所传递的 Subject 参与者必须使用 `java.util.Observable` 类（或其子类）。各位应该没有忘记，Java 的类都是单一继承。如果想用来作为 Subject 参与者的类（即列入观察名单的类）已经是别人的子类时，它就不能再设为 `java.util.Observable` 类的子类。

Coad 一书就提到这个问题。所以才特别介绍能把 Subject 参与者或 Observer 参与者作为 Java 的接口、而且比较容易处理的 Observer Pattern。

**相关 Pattern****◆ Mediator Pattern（第 16 章）**

Mediator Pattern 在进行 Mediator 参与者和 Colleague 参与者的通信时可能会使用 Observer Pattern。

Mediator Pattern 跟 Observer Pattern 两者在“通知状态变化”这点相当类似，不过 Pattern 目的和观点角度并不相同。

Mediator Pattern 也会通知状态变化，不过充其量也只是 Mediator Pattern 的一部分，而 Mediator Pattern 的主要目的还是在 Colleague 参与者的协调。

Observer Pattern 的重点则是放在对 Observer 参与者（可能不只一个）通知 Subject 参与者的变化，而且在通知后取得同步。

**重点回顾**

本章所学习到的是对其他对象通知该对象状态变化的 Observer Pattern。

## 练习题

解答请见附录 A

### 问题 1

请建立一个子类 IncrementalNumberGenerator，让它具有扩充 Number Generator 类（程序 17-2）并逐一加计数值的功能。假设这个子类会传递下列 3 个 int 类型变量作为构造函数的参数：

- 开始的数值
- 结束的数值（但不含此数）
- 增加幅度

另外，请写一个程序让 DigitObserver 类和 GraphObserver 类观察其行为。

IncrementalNumberGenerator 类的用法如程序 17-7 所示，其执行结果如图 17-4。画面上所输出的是大于 10 但小于 50 的数值，且前后两数均相差 5。

程序 17-7 应用 IncrementalNumberGenerator 的 Main 类 (Main.java)

```
public class Main {  
    public static void main(String[] args) {  
        NumberGenerator generator = new IncrementalNumberGenerator  
            (10, 50, 5);  
        Observer observer1 = new DigitObserver();  
        Observer observer2 = new GraphObserver();  
        generator.addObserver(observer1);  
        generator.addObserver(observer2);  
        generator.execute();  
    }  
}
```

### 问题 2

请在本章的程序示例中另外新增一个 ConcreteObserver 参与者，并修改 Main 类（程序 17-6）让它利用这个 ConcreteObserver 参与者。

# 第 18 章

## Memento（备忘录） ——存储状态

## Memento Pattern

在文本软件的操作环境下，即使不小心误删某些文字内容，只要利用恢复（undo）功能就能救回删除前的内容。有些文本软件甚至支持一次以上的恢复行为。

面向对象程序如果要执行复原，必须预先存储对象实例的状态。但是又不能只做存储的行为，否则无法以存储的状态将对象实例复原到原始状态。

必须要能自由存取对象实例内部的状态，才能还原对象实例。不过如果对存取行为毫不设限，又会让高度依赖类内部结构的程序代码分散到程序各处，增加修改类时的困难。这称为封装性被破坏。

加入表示对象实例状态的功能，而能在执行存储和恢复时不破坏封装性，这就是本章要介绍的：

### Memento Pattern

利用 Memento Pattern 可对程序进行以下操作：

- undo（恢复）
- redo（重复）
- history（产生操作记录）
- snapshot（存储目前状态）

Memento Pattern 就是这样的 Design Patterns。它会把某个时间点的对象实例状态记录存储起来，等到以后再让对象实例复原到当时的状态。

## 程序示例

请看下面使用 Memento Pattern 的程序示例。

本章的程序示例是一个“收集水果的骰子游戏”。游戏规则很简单：

- 游戏会自动进行。
- 游戏的主人翁丢骰子，根据骰子的结果决定下一个状态。
- 出现好的点数，主人翁的金钱会增加。
- 出现不好的点数，金钱减少。
- 出现的点数很好时，主人翁可以额外得到一个水果。
- 主人翁玩到没钱时，则游戏结束。

为了后面能不受影响继续进行，程序中存储金钱的位置有建立一个 Memento 类的对象实例，用来存储“目前的状态”。里面存储的是现阶段手边有的金钱和水果。利用预先存储起来的 Memento 的对象实例可以回复到原先的状态，避免如果一直输钱输得精光时会结束程序。

就让我们利用这个小程序来了解 Memento Pattern 的真面目吧！

## Memento (备忘录) —— 存储状态

表 18-1 类一览表

| 包    | 名称      | 说明                                             |
|------|---------|------------------------------------------------|
| game | Memento | 表示 Gamer 状态的类                                  |
| game | Gamer   | 玩游戏的主人翁的类。产生 Memento 的对象实例                     |
| 未命名  | Main    | 进行游戏的类。先把 Memento 的对象实例存储起来，如有必要时再复原 Gamer 的状态 |

## Memento 类

Memento 类（程序 18-1）是表示主人翁（Gamer）状态的类。

Memento 类和 Gamer 类要放在同一个 game 包里。

Memento 类里的字段有 money 和 fruits 两个。Money 是目前手边的金钱总额，fruits 则是目前手边的水果数量。为了让同一包内的 Gamer 类也能自由存取 money 和 fruits 字段，所以这两个字段没有设为 private。

getMoney 方法是取得目前手边金钱总额的方法。

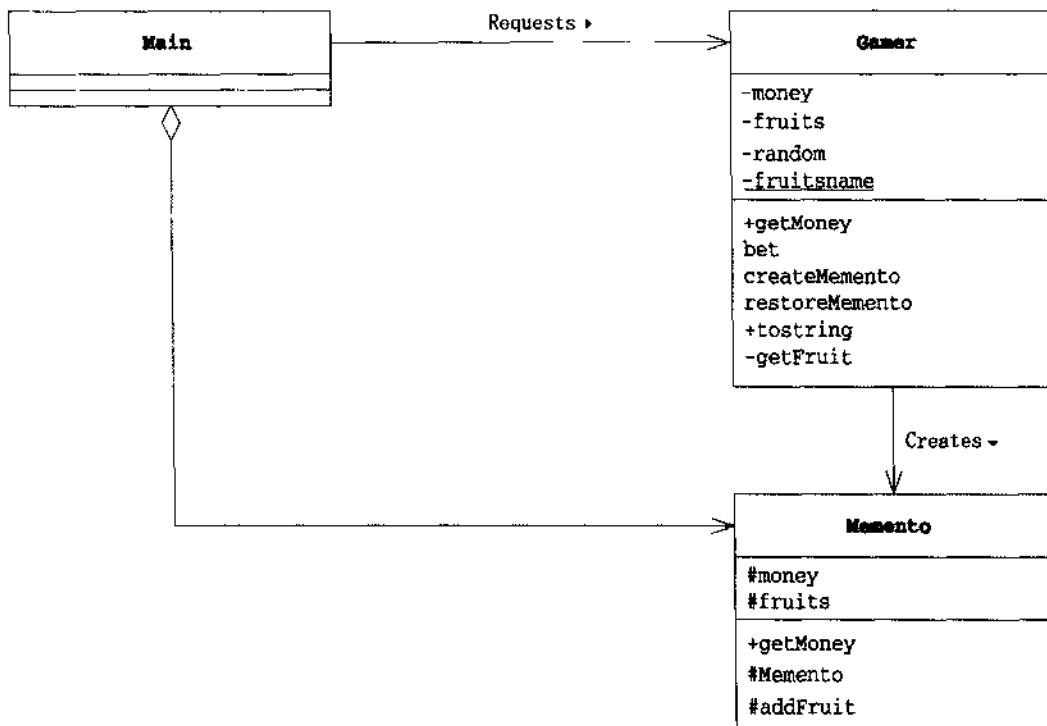


图 18-1 程序示例的类图

**Memento 模式中的应用**

Memento 类的构造函数并没有加上 public。所以 Memento 类的对象实例不是谁都能做，必须是属于同一包（在此为 game 包）的类才能使用。以程序示例为例，产生 Memento 类的对象实例的是 game 包的 Gamer 类。

addFruit 方法是新增水果的方法。这个方法也没有设成 public。只有同一包的类才能新增水果，所以不能从 game 包以外的地方更改 Memento 内部。

Memento 类中有两个批注“narrow interface”和“wide interface”，其相关意义说明请见 p.252。

**程序 18-1 Memento 类 (Memento.java)**

```
package game;
import java.util.Vector;

public class Memento {
    int money;           // 手边金钱总额
    Vector fruits;       // 水果
    public int getMoney() { // 取得手边金钱总额 (narrow interface)
        return money;
    }
    Memento(int money) { // 构造函数 (wide interface)
        this.money = money;
        this.fruits = new Vector();
    }
    void addFruit(String fruit) { // 新增水果 (wide interface)
        fruits.add(fruit);
    }
    Vector getFruits() {
        return (Vector)fruitsclone();
    }
}
```

**Gamer 类**

Gamer 类（程序 18-2）是表示进行游戏的主人翁的类。这个类有手边金钱总额（money）、水果（fruits）和随机数产生器（random）。而类字段则有水果名称（fruitsname）。

整个游戏的核心方法就是 bet（下注）。只要主人翁还没有破产，这个方法就会丢出骰子，根据骰子点数更改手边金钱总额和水果数量。

## Memento（备忘录）——存储状态

`createMemento` 是存储目前状态（快照，snapshot）的方法。先以 `createMemento` 方法产生 `Memento`，此时便依照手边金钱总额和水果产生一个 `Memento` 的对象实例。以这种方式产生的 `Memento` 对象实例即可表示“目前 `Gamer` 对象实例的状态”，这就是 `createMemento` 方法的返回值。有如拍照存证一样，把目前状态封存到 `Memento` 对象实例。水果的部分则只存储“好吃”的水果。

`restoreMemento` 则是复原的方法，恰恰与 `createMemento` 方法相反。它会根据得到的 `Memento` 对象实例还原自己的状态。很像是游戏中“复活的咒语”。

程序 18-2 `Gamer` 类 (`Gamer.java`)

```
package game;
import java.util.Vector;
import java.util.Iterator;
import java.util.Random;

public class Gamer {
    private int money; // 手边金钱总额
    private Vector fruits = new Vector(); // 水果
    private Random random = new Random(); // 随机数产生器
    private static String[] fruitsname = { // 水果名称一览表
        "苹果", "葡萄", "香蕉", "橘子",
    };
    public Gamer(int money) { // 构造函数
        this.money = money;
    }
    public int getMoney() { // 取得目前的手边金钱总额
        return money;
    }
    public void bet() { // 下注...开始游戏
        int dice = random.nextInt(6) + 1; // 掷骰子
        if (dice == 1) { // 第1次...手边金钱有变多
            money += 100;
            System.out.println("手边金钱增加了。");
        } else if (dice == 2) { // 第2次...手边金钱剩一半
            money /= 2;
            System.out.println("手边金钱剩一半。");
        } else if (dice == 6) { // 第6次...得到水果
            String f = getFruit();
            System.out.println("得到水果(" + f + ")。");
        }
    }
    private String getFruit() {
        int index = random.nextInt(fruitsname.length);
        return fruitsname[index];
    }
}
```

## Memento 模式的应用

```

        fruits.add(f);
    } else { // 其他...没有发生什么事
        System.out.println("没有发生什么事。");
    }
}

public Memento createMemento() { // 拍照存证 (snapshot)
    Memento m = new Memento(money);
    Iterator it = fruits.iterator();
    while (it.hasNext()) {
        String f = (String)it.next();
        if (f.startsWith("好吃的")) { // 只存储好吃的水果
            m.addFruit(f);
        }
    }
    return m;
}

public void restoreMemento(Memento memento) { // 进行恢复
    this.money = memento.money;
    this.fruits = memento.getFruits();
}

public String toString() { // 输出成字符串
    return "[money = " + money + ", fruits = " + fruits + "]";
}

private String getFruit() { // 得到 1 个水果
    String prefix = "";
    if (random.nextBoolean()) {
        prefix = "好吃的";
    }
    return prefix + fruitsname[random.nextInt(fruitsname.length)];
}
}

```

## Main 类

Main 类（程序 18-3）产生 Gamer 的对象实例，利用这个对象实例来进行游戏。它会反复调用 Gamer 的 bet 方法，而且每次调用时都会显示手边金钱总额。

光是这样的话，整个游戏就只是掷骰子游戏而已，不过在这里有一个 Memento

## Memento（备忘录）——存储状态

Pattern。变量 memento 里存储“某个时间点的 Gamer 状态”。如果运气好、手边金钱总额有增加时，就用 createMemento 把目前状态存储起来。

当手边金钱快输光时，再把 memento 丢给 restoreMemento 方法即可还原手边金钱总额。

图 18-2 为 Main 调用 Gamer 的 createMememto、restoreMememto 的流程图。

程序 18-3 Main 类 (Main.java)

```
import game.Memento;
import game.Gamer;

public class Main {
    public static void main(String[] args) {
        Gamer gamer = new Gamer(100);      //一开始手边金钱总额为 100
        Memento memento = gamer.createMemento(); //预先存储最初状态
        for (int i = 0; i < 100; i++) {
            System.out.println("==== " + i);    // 输出次数
            System.out.println("现况：" + gamer);   // 输出主人翁的目前状态

            gamer.bet();    // 进行游戏

            System.out.println("手边金钱总额为" + gamer.getMoney() + "元。");

            // 决定如何处理 Memento
            if (gamer.getMoney() > memento.getMoney()) {
                System.out.println("（因为已经赢了不少，故先存储目前状态）");
                memento = gamer.createMemento();
            } else if (gamer.getMoney() < memento.getMoney() / 2) {
                System.out.println("（因为已经输了很多，故恢复到前次状态）");
                gamer.restoreMemento(memento);
            }

            // 等候
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
            System.out.println("");
        }
    }
}
```

```

    }
}

```

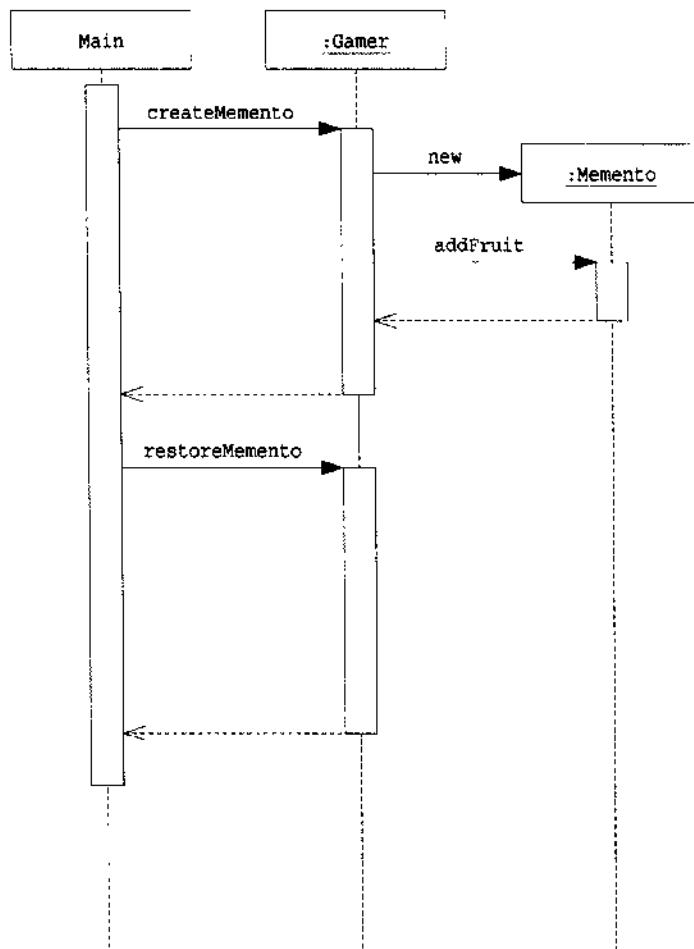


图 18-2 程序示例的流程图

```
===== 0
```

现况: [momey = 100, fruits = [] ]

没有发生什么事。

手边金钱总额为 100 元。

```
===== 1
```

现况: [momey = 100, fruits = [] ]

没有发生什么事。

## Memento(备忘录)——存储状态

手边金钱总额为 100 元。

===== 2

现况: [momey = 100, fruits = [] ]

得到水果(好吃的葡萄)。

手边金钱总额为 100 元。

===== 3

现况: [momey = 100, fruits = [好吃的葡萄] ]

没有发生什么事。

手边金钱总额为 100 元。

===== 4

现况: [momey = 100, fruits = [好吃的葡萄] ]

得到水果(葡萄)。

手边金钱总额为 100 元。

===== 5

现况: [momey = 100, fruits = [好吃的葡萄, 葡萄] ]

手边金钱增加了。

手边金钱总额为 200 元。

(因为已经赢了不少, 故先存储目前状态) ← 产生 Memento

(中间省略) ...

===== 13

现况: [momey = 400, fruits = [好吃的葡萄, 葡萄, 好吃的橘子] ]

手边金钱增加了。

手边金钱总额为 400 元。

(因为已经赢了不少, 故先存储目前状态) ← 产生 Memento

(中间省略) ...

===== 21

现况: [momey = 200, fruits = [好吃的葡萄, 葡萄, 好吃的橘子] ]

没有发生什么事。

手边金钱总额为 200 元。

==== 22

现况: [momey = 200, fruits = [好吃的葡萄, 葡萄, 好吃的橘子] ]

手边金钱剩一半。

手边金钱总额为 100 元。

(因为已经输了很多, 故恢复到前次状态) ← 利用 Memento 恢复状态

==== 23

现况: [momey = 400, fruits = [好吃的葡萄, 好吃的橘子] ]

没有发生什么事。

手边金钱总额为 400 元。

(以下省略) ...

图 18-3 执行结果显示例 (部分)

## Memento Pattern 的所有参与者

Memento Pattern 中出现的所有参与者可整理如下。

### ◆ Originator (原发器) 参与者

Originator 参与者是在欲存储本身目前状态时产生一个 Memento 参与者。当 Originator 参与者又接收到以前的 Memento 参与者时, 便进行恢复到产生 Memento 参与者时状态的处理。在前面的程序示例中, 扮演这个角色的是 Gamer 类。

### ◆ Memento (备忘录) 参与者

Memento 参与者是整合 Originator 参与者的内部信息。Memento 参与者虽然有 Originator 参与者的内部信息, 但并不会随便把信息公开出去。

Memento 参与者有下列两种类接口 (API)。

- wide interface——广义接口 (API)

Memento 参与者所提供的“广义接口 (API)”是一个可取得对象状态恢复原状时的必要信息的所有方法的集合。广义接口 (API) 会泄漏 Memento 参与者的内部状态, 所以只有 Originator 参与者能使用它。

- narrow interface——狭义接口 (API)

Memento 参与者所提供的“狭义接口 (API)”是给外部 Caretaker 参与者看的。狭义接口 (API) 的能力有限, 可预防内部状态公开给外部的危险。

视情况使用这两种不同的接口 (API) 可以避免对象封装性遭破坏。

在前面的程序示例中, 扮演这个角色的是 Memento 类。

## Memento (备忘录) —— 存储状态

Originator 参与者和 Memento 参与者两者的关系非常密切。

### ◆ Caretaker (负责人) 参与者

如欲存储目前 Originator 参与者的状态时, Caretaker 参与者会把这个情形告诉 Originator 参与者。Originator 参与者接收到这个消息后就产生 Memento 参与者, 然后传递给 Caretaker 参与者。Caretaker 参与者为了将来可能会需要使用, 因此要预先存储这个 Memento 参与者。在前面的程序示例中, 扮演 Caretaker 参与者的是 Main 类。

但是 Caretaker 参与者只能使用 Memento 参与者的 2 种接口 (API) 中的“狭义接口 (API)”, 所以不能存取 Memento 参与者的内部信息。它只会把别人产生出来的 Memento 参与者照单全收存储起来, 当作是一块未知区域 (black box)。

Originator 参与者和 Memento 参与者的结合相当紧密, 但 Caretaker 参与者跟 Memento 参与者的结合比较松散。Memento 参与者会对 Caretaker 参与者隐藏信息。

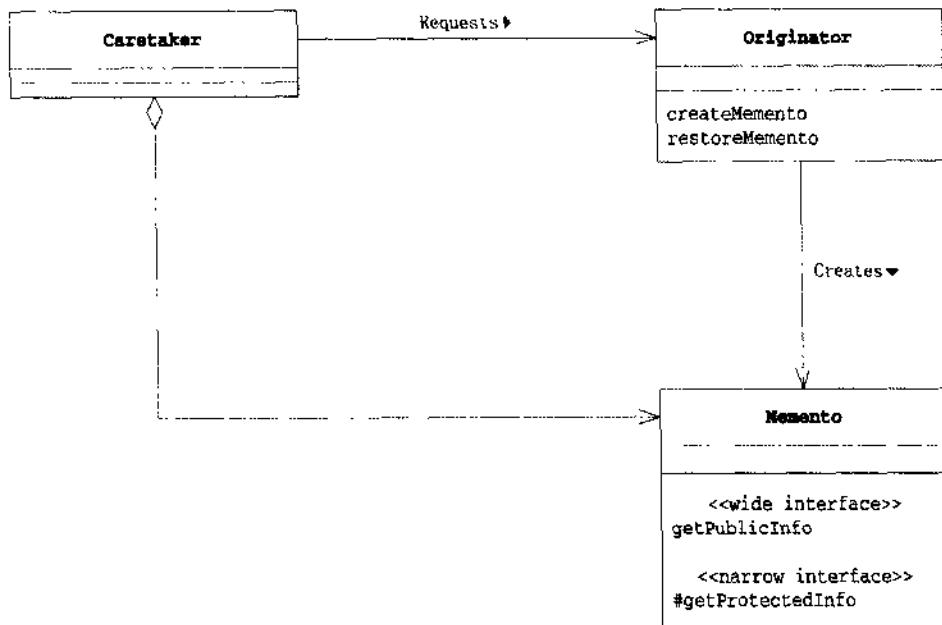


图 18-4 Memento Pattern 的类图

## 扩展自我视野的提示

### 2 种接口 (API) 与存取控制

本章的程序示例利用 Java 的存取控制功能来操作 Memento Pattern 的 2 个接口

(API)。Java 语言中有表 18-2 所示的 4 种不同的存取控制。

表 18-2 Java 的存取控制

| 存取控制      | 说明          |
|-----------|-------------|
| Public    | 从任何类均能看到    |
| protected | 从当前包和子类均能看到 |
| 无         | 只有从当前包才能看到  |
| private   | 只有从当前类才能看到  |

Memento 类的方法和字段分成 public 和什么都没有这两种，用意是为了方便在程序中表现哪个方法（或字段）要给哪个类看、又不给哪个类看（表 18-3）。

表 18-3 Memento 类中所使用到的存取控制

| 存取控制   | 字段/方法/构造函数 | 谁看得到                     |
|--------|------------|--------------------------|
| 无      | money      | Memento 类、Gamer 类        |
| 无      | fruit      | Memento 类、Gamer 类        |
| public | getMoney   | Memento 类、Gamer 类、Main 类 |
| 无      | Memento    | Memento 类、Gamer 类        |
| 无      | addFruit   | Memento 类、Gamer 类        |

Memento 类中只有 getMoney 方法是 public，设为狭义接口 (API)。Caretaker 参与者的 Main 类也可以利用这个方法。

稍微再补充一点，明明是 public 却还说它是“狭义”，可能会让人觉得前后矛盾。不过这里所指的“狭义”是“可操作内部状态的比例较少”的意思。Memento 类的方法当中只有 getMoney 加上 public，换句话说，它只能取得手边金钱总额。我们把像这样“能做的不多”的状态称为“狭义”。

Caretaker 参与者的 Main 类并不在 game 包之内，故只能使用 Memento 类的 public 方法 getMoney。因此这个 Main 类不能任意修改 Memento 类的内容。

Main 类就只能以 getMoney 方法取得手边金钱总额、以及预先将 Memento 类的对象实例保留在变量等位置。

请注意：Main 类内看不到任何 Memento 的构造函数。没有构造函数的意思是它不能在 Main 类中建立 Memento 的对象实例，例如：

```
new Memento(100)
```

这种写法会使程序编译时出错。因为如果 Main 类需要有 Memento 的对象实例时，就会调用 createMemento 方法，跟 Gamer 类要求：“我想存储目前状态，请产生一个 Memento 的对象实例”。

利用存取控制就能在程序中表达出“A 类可以看得到这个方法，但 B 类却看不到”。

## Memento 可以有几个

在本章的程序示例中，Main 类的 Memento 只有 1 个。不过，如果 Main 类利用数组等方式而拥有一个以上的 Memento 对象实例，就可以存储对象实例任何一个时间点的状态。

## Memento 有没有使用期限

本章的程序示例只把 Memento 保留在内存上，这还没有什么大问题，如果是像问题 4 那样把 Memento 永久存储成文件格式的话，Memento 的“使用期限”会是一个问题。

想想，即使你把这个时间点的 Memento 存储成文件，如果以后程序有版本更新的话，很有可能无法与此时所存储的文件 Memento 顺利整合。

## Caretaker 参与者跟 Originator 参与者区隔使用的意义

或许有些读者会想问：“如果想要做复原操作，把功能直接做到 Originator 参与者里不是比较好吗？为什么要特地用到 Design Patterns 呢？”。

Caretaker 参与者的功能是决定要在哪个时间点做快照（snapshot）、什么时候复原以及保留 Memento 参与者。

而 Originator 参与者是负责产生 Memento 参与者以及利用传递过来的 Memento 参与者复原自己的状态。

Caretaker 参与者跟 Originator 参与者两者间有这样的工作内容分配，好处是如果要做以下修正时，就不需要去修改 Originator 参与者。

- 将复原操作修改为需要有一个步骤以上
- 除了复原之外，还要将目前状态存储成文件

## 相关 Pattern

### ◆ Command Pattern（第 22 章）

若利用 Command Pattern 处理命令时，可使用 Memento Pattern 进行复原（undo）或重复（redo）。

### ◆ Prototype Pattern（第 6 章）

Memento Pattern 会存储目前的对象实例状态，以便进行快照或复原。只会存储还原时所需要的信息。

Prototype Pattern 则是另外产生一个跟目前对象实例同样状态的对象实例。完全复制对象实例的内容。

### ◆ State Pattern (第 19 章)

在 Memento Pattern, 以对象实例表示“状态”。

而在 State Pattern, 则以类表示“状态”。

## 重点回顾

本章所介绍的是用来记录、存储对象目前状态的 Memento Pattern。这是尽可能在不公开对象内部状态的前提下，而能存储对象状态的方法。

Caretaker 参与者去找 Originator 参与者产生表示“目前状态”的 Memento 参与者（拍照存证）。Caretaker 参与者并不知道 Memento 参与者的内部信息，而且也不会过问。Caretaker 参与者为了以后可能要还原，所以预先将 Memento 参与者存储起来。有这个需求时，再从抽屉取出 Memento 参与者交给 Originator 参与者，即可顺利还原。这就是 Memento Pattern。

另外，也确认了利用 public、protected、private 等的存取控制信息可控制能见 / 不能见的方法。

## 练习题

解答请见附录 A

### 问题 1

Caretaker 参与者只能利用狭义接口 (API) 来操作 Memento 参与者。请问：如果 Caretaker 参与者能随意操作 Memento 参与者时，会发生什么问题？

### 问题 2

在本章的程序示例中，决定状态的因素只有手边金钱总额和水果，如果应存储的信息量较多时，必须有更多的内存空间或磁盘空间才能存储 Memento 的对象实例。请想想该如何处理这个情形（此问题引用自 GoF 一书）。

### 问题 3

假设现在想在 Memento 类（程序 18-1）加入一个新字段：

`int number;`

且能进行以下的存取控制。请问应该怎么做？

- Memento 类可取得或修改 number 的值
- Gamer 类可取得但无法修改 number 的值

## Memento（备忘录）——存储状态

- Main 类无法取得或修改 number 的值

**问题 4**

利用串行化（serialization）功能可以将 Memento 的对象实例存储成文件。请修改本章的程序示例使其具有下列功能。

- (1) 若无文件 game.dat 时，手边金钱总额从 100 元开始
- (2) 当手边金钱总额增加到一定额度时，将 Memento 的对象实例存储成文件 game.dat
- (3) 如欲启动程序时已经有文件 game.dat，则从该文件所存储的状态开始  
各位可翻到以下部分参考串行化的相关信息。
  - (a) 存储的 Memento 类是实现 java.io.Serializable 接口
  - (b) 存储时使用 ObjectOutputStream 类的 writeObject 方法
  - (c) 复原时则使用 ObjectInputStream 类的 readObject 方法
  - (d) 详细内容可参考 Java 的 API 参照的各项：
    - java.io.ObjectOutputStream 类
    - java.io.ObjectInputStream 类
    - java.io.ObjectOutput 接口
    - java.io.ObjectInput 接口



# 第 19 章

## State（状态）—— 以类表示状态

## State Pattern

在面向对象程序语言中，欲写成程序的部分都会以“类”来表示，而哪些部分该用类来表示则是程序员个人的设计了。有时候，类所对应的具体“东西”可能就在实际生活的社会里，有时则是虚无缥缈；或许你还会大吃一惊“什么，这也能用类来表示？”。

而在本章所要学习的

### State Pattern

当中，就以类来表示“状态”。state 的英文原意是“状态（物体状况）”。在现实世界中，我们会考虑到很多种“状态”，但很少会把状态视为一个“物体”。因此，“以类来表示状态”这个概念可能不太容易体会得出来。

各位在本章将会学到如何以类表现“状态”的方法。以类来表示状态之后，只要切换类就能表现“状态变化”，而且在必须新增其他状态时，也很清楚该编写哪个部分。

## 程序示例

请看下面这个应用了 State Pattern 的程序示例。

### 金库保安系统

假设现在有一个会随时间改变警备状态的金库保安系统。金库保安系统其实很简单，基本概念如表 19-1 所示。此系统的架构图如图 19-1。

我们来想想看该如何把这个金库保安系统写成程序。时间如果要实际计时很麻烦，因此在此程序中的 1 秒等于实际生活的 1 小时。图 19-2 即为程序执行经过的一部分。

表 19-1 金库保安系统

- 有 1 个金库
- 金库跟保安中心联机
- 金库有警铃和一般通话用的电话
- 金库有时钟，监视目前的时间
- 白天是 9:00~16:59，晚间为 17:00~23:59 以及 0:00~8:59
- 只有白天才能使用金库
- 在白天使用金库时，保安中心会保留使用记录
- 若晚间使用金库，保安中心会接到发生异常现象的通知

## State(状态)——以类表示状态

续上表

- 警铃是 24 小时均可使用
- 一旦使用警铃，保安中心就会接到警铃通知
- 一般通话用的电话是 24 小时均可使用（但晚间只有录音机服务）
- 在白天使用电话时，就会调用保安中心
- 若晚间使用电话时，则会调用保安中心的录音机

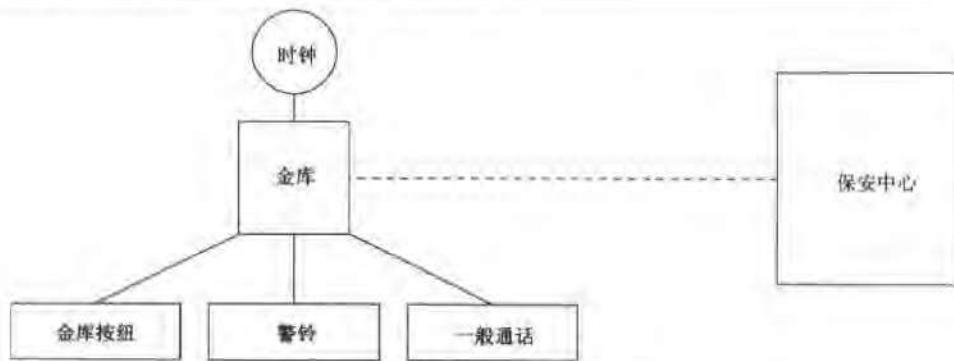


图 19-2 程序示例执行的情形

## 不使用 State Pattern 的虚拟程序代码

在正式开始说明 State Pattern 的程序示例之前，请先稍微想一想自己会怎么来编写这个程序。

如果是笔者来写的话，看完前面的细节说明后会这么想：

“嗯，基本上就是让系统行为根据现在时间而变化。只要发生了金库使用中、警铃或一般通话三者之一的话，保安中心就会接到相关的联络报告。而联络事项又会因时间而改变。”

根据这样的想法，应该就会勾勒出如程序 19-1 的虚拟程序代码。接下来就是该如何用程序语言来表达这个虚拟程序代码的问题了。

#### 程序 19-1 不使用 State Pattern 的金库保安系统虚拟程序代码（1）

```

保安系统的类 {
    使用金库时所调用的方法 () {
        if (白天) {
            保安中心会有使用记录
        } else if (晚间) {
            保安中心会有发生异常状态的通知
        }
    }

    使用警铃时所调用的方法 () {
        保安中心会有警铃通知
    }

    一般通话时所调用的方法 () {
        if (白天) {
            调用保安中心
        } else if (晚间) {
            调用保安中心的录音机
        }
    }
}

```

## 使用 State Pattern 的虚拟程序代码

其实程序 19-1 的做法并没有任何问题，只不过本章所要介绍的 State Pattern 是从另一个不同的角度来处理同一件事，程序编辑如程序 19-2 所示。

#### 程序 19-2 使用 State Pattern 的金库保安系统虚拟程序代码（2）

```

表示白天这个状态的类 {
    使用金库时所调用的方法 () {
        保安中心会有使用记录
    }

    使用警铃时所调用的方法 () {
        保安中心会有警铃通知
    }

    一般通话时所调用的方法 () {
        调用保安中心
    }
}

表示晚间这个状态的类 {
    使用金库时所调用的方法 () {
        保安中心会有发生异常状态的通知
}

```

## State（状态）——以类表示状态

```

    }
    使用警铃时所调用的方法 () {
        保安中心会有警铃通知
    }
    一般通话时所调用的方法 () {
        调用保安中心的录音机
    }
}

```

---

各位看得出来这 2 个虚拟程序代码有什么不一样吗？

在没有使用 State Pattern 的（1），白天和晚间的状态出现在各个方法的 if 语句，就在各个方法当中调查现在的状态。

而使用 State Pattern 的（2），白天、晚间的状态则是以类来表示。因为状态已经用类来表示，所以方法里并没有任何调查状态的 if 语句。

（1）是把“状态”写在方法里面，（2）则是把它写成类。相信各位已经掌握到这里的特征就是把深藏在方法内部的“状态”挖出来并拉到最外面。

请各位在阅读程序示例时，脑海中要保持住（1）和（2）的核心概念。

表 19-2 类与接口一览表

| 名称         | 说明                            |
|------------|-------------------------------|
| State      | 表示金库状态的接口                     |
| DayState   | 实现 State 的类。表示白天的状态           |
| NightState | 实现 State 的类。表示夜间的状态           |
| Context    | 管理金库的状态变化，跟保安中心联络的接口          |
| SafeFrame  | 实现 Context 的类。含有按钮、画面显示等的用户接口 |
| Main       | 测试用的类                         |

## State 接口

State 接口（程序 19-3）是表示金库的状态。State 接口规定了遇到下列情形时所调用的接口（API）：

- 设定时刻时
- 使用金库时
- 有人按下警铃时
- 进行一般通话时

各位刚才所见的虚拟程序代码当中的“使用金库时所调用的方法”等就是这个接口（API）。这里所规定的方法都是根据状态而改变处理内容；换句话说，State 接口就是一个依状态而变的方法的集合。

参数所传递的 Context 是管理状态的接口，后面会另有详细说明。

## 程序 19-3 State 接口 (State.java)

```
public interface State {
    public abstract void doClock(Context context, int hour);
    // 设定时间
    public abstract void doUse(Context context);      // 使用金库
    public abstract void doAlarm(Context context);     // 警铃
    public abstract void doPhone(Context context);     // 一般通话
}
```

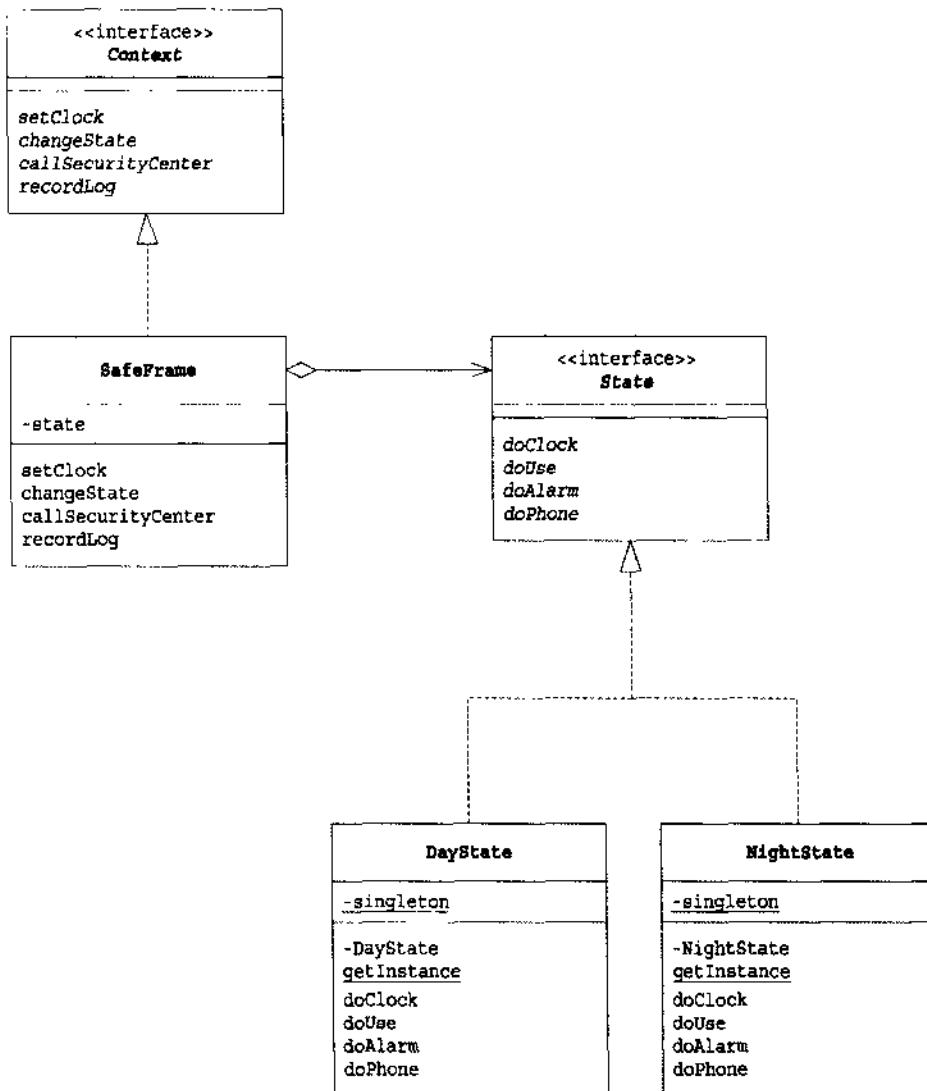


图 19-3 程序示例的类图

## DayState 类

DayState 类(程序 19-4)是表示白天状态的类。这个类是实现 State 接口，所以它会实现 State 接口中所声明的方法。

表示状态的类只会一个个产生对象实例，因为如果每当状态一有变化就要产生新的对象实例，反而会浪费内存和处理时间。基于这个理由，这里使用到 Singleton Pattern(第 5 章)。

doClock 方法是设定时间的方法。如果参数所传递的时间是晚间的话，系统便移转为晚间的状态。这个地方发生状态的变化(状态迁移)，所以利用 Context 接口的 changeState 方法来表示状态的变化。晚间的状态则作成 NightState 类，对象实例则以 getInstance 方法取得(这里使用到 Singleton Pattern，请注意不要写成 new NightState())。

doUse、doAlarm、doPhone 则是分别对应使用金库、警铃和一般通话等不同情形的方法。它们的工作只有调用 Context 的方法去作该做的事而已。请注意，这些方法中都没有“调查现在状态”之类的 if 语句。程序员在写这些方法时要不断在心里重复着：“现在是白天的状态、现在是白天的状态”。State Pattern 用不同的类来表现状态变化，所以就不需要另外用 if 语句或 switch 语句去做不同状态时的条件判断。

程序 19-4 DayState 类(DayState.java)

```
public class DayState implements State {
    private static DayState singleton = new DayState();
    private DayState() { // 构造函数为 private
    }
    public static State getInstance() { // 取得唯一的对象实例
        return singleton;
    }
    public void doClock(Context context, int hour) { // 设定时间
        if (hour < 9 || 17 <= hour) {
            context.changeState(NightState.getInstance());
        }
    }
    public void doUse(Context context) { // 使用金库
        context.recordLog("使用金库(白天)");
    }
    public void doAlarm(Context context) { // 警铃
        context.callSecurityCenter("警铃(白天)");
    }
    public void doPhone(Context context) { // 一般通话
    }
}
```

## Java语言中的应用

```

        context.callSecurityCenter("一般的通话(白天)");
    }
    public String toString() { // 输出字符串
        return "[白天]";
    }
}

```

---

**NightState 类**

NightState 类（程序 19-5）是表示晚间这个状态的类。它跟 DayState 类一样都使用 Singleton Pattern。架构亦同 DayState 类，故不另赘述。

**程序 19-5 NightState 类（NightState.java）**

```

public class NightState implements State {
    private static NightState singleton = new NightState();
    private NightState() { // 构造函数为 private
    }
    public static State getInstance() { // 取得唯一的对象实例
        return singleton;
    }
    public void doClock(Context context, int hour) { // 设定时间
        if (9 <= hour && hour < 17) {
            context.changeState(DayState.getInstance());
        }
    }
    public void doUse(Context context) { // 使用金库
        context.callSecurityCenter("异常：晚间使用金库！");
    }
    public void doAlarm(Context context) { // 警铃
        context.callSecurityCenter("警铃(晚间)");
    }
    public void doPhone(Context context) { // 一般通话
        context.recordLog("晚间的通话录音");
    }
    public String toString() { // 输出字符串
        return "[晚间]";
    }
}

```

---

## Context 接口

Context 接口（程序 19-6）负责管理状态和调用保安中心。它的实际行为则留在后面的 SafeFrame 类（程序 19-7）一起说明。

程序 19-6 Context 接口（Context.java）

```
public interface Context {
    public abstract void setClock(int hour);      // 设定时间

    public abstract void changeState(State state); // 状态变化
    public abstract void callSecurityCenter(String msg); // 调用
    // 保安中心
    public abstract void recordLog(String msg); // 保安中心保留记录
}
```

## SafeFrame 类

SafeFrame 类（程序 19-7）是利用 GUI 设计出一个金库保安系统（safe 有“金库”的意思）。SafeFrame 类会实现 Context 接口。

SafeFrame 类的字段是画面上的文字字段、文字区以及按钮等。不过只有 state 字段是表示金库现在状态，而不是输出到画面上的零件。程序一开始先初始化成“白天”的状态。

构造函数所执行的处理如下：

- 设定背景颜色
- 设定版面编排管理员（layout manager）
- 输出零件的位置
- 设定接受命令者（listener）

设定接受命令者是一大重点，请容笔者稍微说明一下。先调用各个按钮的 addActionListener 方法以设定接受命令者。此时，把“按钮按下时所调用的对象”赋给 addActionListener 方法的参数。addActionListener 方法的参数须实现 ActionListener 接口，这里所指定的是 this，也就是 SafeFrame 的对象实例（SafeFrame 确实有 implements ActionListener）。在按钮被按下时便调用接受命令者（listener（聆听的人））这个构造有点类似 Observer Pattern（第 17 章）。

actionPerformed 方法是按钮被按下时所调用的方法。这个方法名称是 ActionListener 接口（java.awt.event.ActionListener 接口）所规定的名称，不可任意更名。这个方法中会调查是哪个按钮被按下，并执行相对应的处理动作。

请各位注意一下，这里是用到了 if 语句，不过这个 if 语句的条件判断所调用的对

## Java语言中的应用

象其实是“按扭种类”，跟“现在状态”毫无关系。请不要误解说“明明说 State Pattern 不会用到 if 语句，这里怎么还用呢？”。

接下来，处理内容是整个 State Pattern 中最重要的一部分。举例来说，当使用金库的按扭被按下时，系统就会执行下面这个语句：

```
state.doUse(this);
```

它不去调查现在时间或金库的现在状态，反而立刻调用 doUse。这就是 State Pattern 的特征。如果你的程序决定不用 State Pattern 的话，那就应该做“按照现在时间判断来进行不同的处理”，而不必立即调用 doUse。

setClock 方法则是将现在时间设定为指定时间。利用下面的语句即可将现在时间输出成标准输出的格式。

```
System.out.println(clockstring);
```

下面的语句是把现在时间输出到 textClock 这个文字字段（画面的最上面）。

```
textClock.setText(clockstring);
```

接着再利用下面这个语句根据现在状态执行不同的处理（这里面还有发生状态迁移）。

```
state.doClock(this, hour);
```

changeState 方法是被 DayState 类或 NightState 类调用出来，每当发生状态迁移时就会调用这个方法。实际的状态迁移则是用下面这行语句来进行。

```
this.state = state;
```

把表示状态的类的对象实例指定到表示现在状态的字段，就相当于状态迁移。

callSecurityCenter 方法是表示调用保安中心，recordLog 方法则是表示保安中心所留存的记录。这里是把字符串新增到 textScreen 这个文字区内。如果是真正的保安系统程序，这里应该是要用网络或其他方式去调用实际生活中的保安中心。

**程序 19-7 SafeFrame 类 (SafeFrame.java)**

```
import java.awt.Frame;
import java.awt.Label;
import java.awt.Color;
import java.awt.Button;
import java.awt.TextField;
import java.awt.TextArea;
import java.awt.Panel;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class SafeFrame extends Frame implements ActionListener,
```

## State（状态）——以类表示状态

```
Context {
    private TextField textClock = new TextField(60); //输出现在
    //时间
    private TextArea textScreen = new TextArea(10, 60); //输出保安
    //中心
    private Button buttonUse = new Button("使用金库"); //使用金库
    //的按钮
    private Button buttonAlarm = new Button("警铃"); //警铃的按
    //钮
    private Button buttonPhone = new Button("一般通话"); //一般通话
    //的按钮
    private Button buttonExit = new Button("结束"); //结束的按钮

    private State state = DayState.getInstance(); //现在状态

    //构造函数
    public SafeFrame(String title) {
        super(title);
        setBackground(Color.lightGray);
        setLayout(new BorderLayout());
        // 输出 textClock
        add(textClock, BorderLayout.NORTH);
        textClock.setEditable(false);
        // 输出 textScreen
        add(textScreen, BorderLayout.CENTER);
        textScreen.setEditable(false);
        // 把按钮放到面板上
        Panel panel = new Panel();
        panel.add(buttonUse);
        panel.add(buttonAlarm);
        panel.add(buttonPhone);
        panel.add(buttonExit);
        // 输出面板
        add(panel, BorderLayout.SOUTH);
        // 输出到画面上
        pack();
        show();
        // 设定听命令者
    }
}
```

## 命令模式的应用

```

        buttonUse.addActionListener(this);
        buttonAlarm.addActionListener(this);
        buttonPhone.addActionListener(this);
        buttonExit.addActionListener(this);
    }
    // 若有人按下按钮，则跳到这里
    public void actionPerformed(ActionEvent e) {
        System.out.println("") + e);
        if (e.getSource() == buttonUse) { // 使用金库的按钮
            state.doUse(this);
        } else if (e.getSource() == buttonAlarm) { // 警铃的按钮
            state.doAlarm(this);
        } else if (e.getSource() == buttonPhone) { // 一般通话的按钮
            state.doPhone(this);
        } else if (e.getSource() == buttonExit) { // 结束的按钮
            System.exit(0);
        } else {
            System.out.println("?");
        }
    }
    // 设定时间
    public void setClock(int hour) {
        String clockstring = "现在时间是";
        if (hour < 10) {
            clockstring += "0" + hour + ":00";
        } else {
            clockstring += hour + ":00";
        }
        System.out.println(clockstring);
        textClock.setText(clockstring);
        state.doClock(this, hour);
    }
    // 状态变化
    public void changeState(State state) {
        System.out.println("状态已经从" + this.state + "变成" + state
+ ". ");
        this.state = state;
    }
}

```

## State(状态)——以类表示状态

```

// 调用保安中心
public void callSecurityCenter(String msg) {
    textScreen.append("call! " + msg + "\n");
}

// 保安中心保留记录
public void recordLog(String msg) {
    textScreen.append("record ... " + msg + "\n");
}
}

```

状态迁移的前后进行 doUse 的情形如流程图(图 19-4)所示。一开始调用 DayState 端的 doUse，但在 changeState 之后就改调用 NightState 端的 doUse。

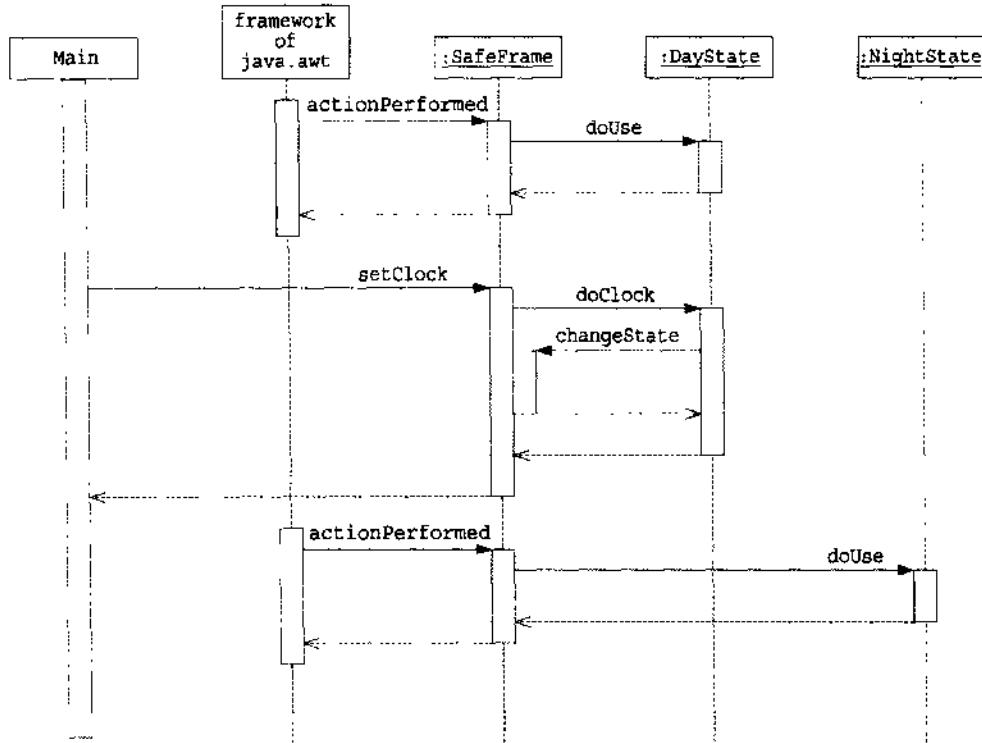


图 19-4 程序示例的流程图

## Main 类

Main 类(程序 19-8)先产生 1 个 SafeFrame 的对象实例，再定期以 setClock 对

## 状态模式的应用

该对象实例设定时间。设定时间的操作每秒就会进行一次，不过在程序中它是代表 1个小时。

程序 19-8 Main 类 (Main.java)

```
public class Main extends Thread {
    public static void main(String[] args) {
        SafeFrame frame = new SafeFrame("State Sample");
        while (true) {
            for (int hour = 0; hour < 24; hour++) {
                frame.setClock(hour); // 设定时间
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                }
            }
        }
    }
}
```

## State Pattern 的所有参与者

State Pattern 的所有参与者可整理如下。

### ◆ State (状态) 参与者

State 参与者是表示状态。规定不同状态下做不同处理的接口 (API)。这个接口 (API) 等于是一个不同状态所做处理的所有方法的集合。在本章的程序示例中，扮演这个角色的是 State 接口。

### ◆ ConcreteState (具体状态) 参与者

ConcreteState 参与者是表示具体的不同状态，具体实现在 State 参与者所规定的接口 (API)。在本章的程序示例中，扮演这个角色的是 DayState 类和 NightState 类。

### ◆ Context (状况、前后关系、语法) 参与者

Context 参与者具有表示现在状态的 ConcreteState 参与者，而且还规定 State Pattern 的利用者所需要的接口 (API)。在本章的程序示例中，扮演这个角色的是 Context 接口和 SafeFrame 类。

再补充一点，在本章的程序示例中，Context 接口负责规定接口 (API) 的部分，SafeFrame 类负责具有 ConcreteState 参与者的部分。Context 参与者为什么不也写成类

## State（状态）——以类表示状态

的理由就留在练习题 19-1 再考验各位看看。

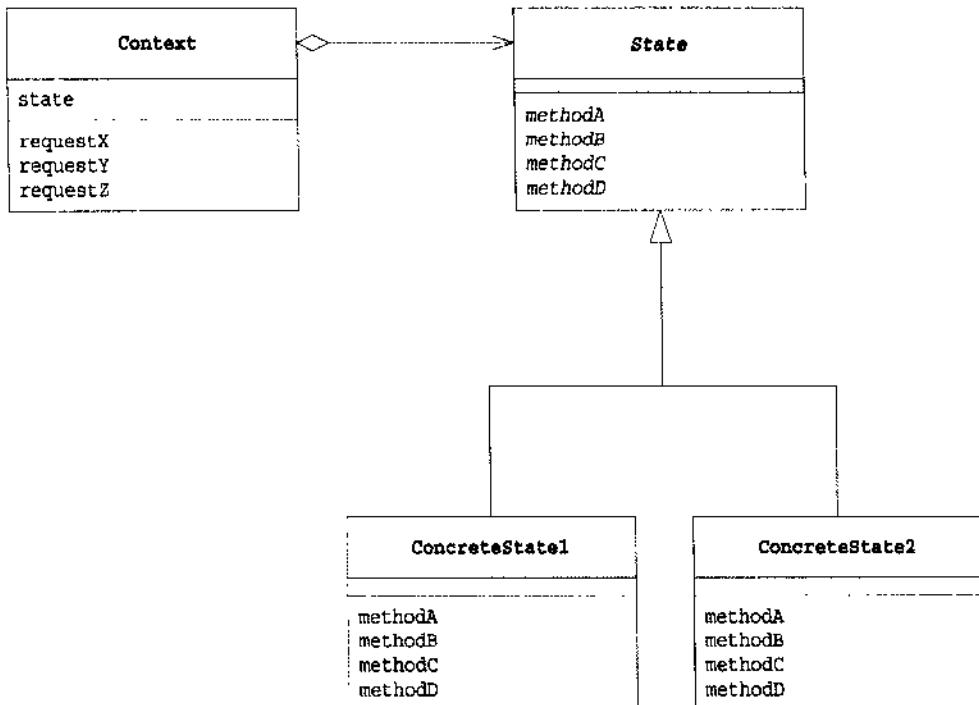


图 19-5 State Pattern 的类图

## 扩展自我视野的提示

### 分割统合

在编写程序时经常会看到分割（divide and conquer）这个方针，它主要用来处理大型且复杂的程序。如果问题牵扯过广又难以处理的话，千万不要硬碰硬想去解。应该先把大问题分割成几个小问题；如果还是不容易求解的话，就再分割成更小的问题。一个大问题要先分解成容易解决的小问题后，再来一个个慢慢解决。与其费尽力解决一个难解的大问题，还不如把它分解成多个容易处理的小问题。“分割”这个名词，简单来说就有点像是这样的概念。

State Pattern 是以类来表示“状态”。利用不同的类分别表示各种具体状态的动作就是在分割问题。当程序员正在写某一个 ConcreteState 参与者的类时，可以让自己的脑海空白一片没有其他类（嗯，不一定是 100% 但至少要有这样的感觉）。本章所提出的金库保安系统只有 2 种状态，可能比较不容易感觉 State Pattern 的优点，不过如果

状态一多起来，State Pattern 就能发挥它的威力。

请回想一下 p.262 的虚拟程序代码（1）和（2）。如果坚持不使用 State Pattern 的话，使用金库时所调用的方法里就会需要根据现在状态进行条件判断的处理。而状态种类越多，条件判断处理就越多；况且发生时所调用的所有方法都要写同样的条件判断处理。

因为 State Pattern 是用类来表示系统“状态”，所以才能细细分割一个庞大复杂的程序。

## 有此状态才会有的处理

我们来探讨一下 SafeFrame 类的 setClock 方法（p.270）跟 State 接口的 doClock 方法（p.264）两者之间的关系。

SafeFrame 类的 setClock 方法是被 Main 类调用的。

Main 类调用 setClock 方法说“请设定时间”。

在 setClock 方法中，这个处理委托给 state，即：

```
state.doClock(this, hour);
```

也就是说，设定时间被视为一个“有现在的状态才会有的处理”。

不只是 doClock 方法，State 接口所声明的方法都是“有该状态才会有的处理”，其实就是“因状态而异的处理”。这是理所当然的常识，同时也是应该确实掌握的重点。

State Pattern 究竟是如何在程序中表示“有该状态才会有的处理”？这部分可浓缩成下面 2 个要点：

- 声明成抽象方法，作为接口
- 实现成具体方法，作为不同的类

这就是 State Pattern 中对“有此状态才会有的处理”的表现方式。

上面两个要点写得比较抽象一点，但如果各位读者看了能心领神会而点头称是的话，那么就恭喜你了，这表示你已经相当清楚 State Pattern 以及接口和类的关系。

## 状态迁移该由谁管

以类来表示状态，然后把各个状态的处理内容分配出去给不同的 ConcreteState 参与者，这确实是相当高明的做法。

不过在使用 State Pattern 时，要注意状态迁移应该由谁来管理。

在本章的程序示例中，Context 参与者的 SafeFrame 类实现了实际执行状态迁移的 changeState 方法。但实际调用 changeState 方法的却是 ConcreteState 参与者的 DayState 类和 NightState 类；换句话说，程序示例是把“状态迁移”视为“某种状态下的处理内容”。这种做法虽好，但也有缺点。

## State（状态）——以类表示状态

优点是可以把“何时迁移到其他状态”的信息整合在同一个类内。也就是说如果想了解“DayState 类是在何时迁移到其他状态？”时，只要检查 DayState 类的程序代码就能知道。

缺点是“某 1 个 ConcreteState 参与者必须了解其他的 ConcreteState 参与者”。假设 DayState 类在 doClock 方法当中使用了 NightState 类，如果将来想要删除 NightState 类时，就必须同时修改 DayState 类。把状态迁移交给 ConcreteState 参与者是指这个 ConcreteState 参与者至少必须了解其他的 ConcreteState 参与者。也就是说，如果把状态迁移交给 ConcreteState 参与者的话，会加深类之间的依存关系。

你也可以不采用程序示例的做法，把所有状态迁移都交给 Context 参与者的 SafeFrame 类。有时这样可以提高各个 ConcreteState 参与者的独立性，容易看出整个程序的走向。不过，现在 Context 参与者变成必须了解“所有 ConcreteState 参与者”。也许这里改用 Mediator Pattern（第 16 章）会比较合适。

或者是改用状态表（table）来取代 State Pattern 也行，在此不另详细说明，不过这个状态表要是能根据“输入内容和内部状态”而取得“输出结果和下一个内部状态”的一览表。如果状态迁移有固定的规则可循，利用这个状态表来写程序应该也很有效的吧。

## 不会有自我矛盾

如果现在故意不用 State Pattern，改以多个变量之值的集合来表示系统状态，这时候变量值之间不能有自我矛盾或不一致的情形。

State Pattern 是以类来表示状态，而表示现在状态的变量只有 1 个。在本章的程序示例中，这个变量是 SafeFrame 类的 state 字段。state 字段的值就会直接决定系统的状态，所以不会有自我矛盾的问题。

## 新增状态易如反掌

利用 State Pattern 时，新增状态非常简单。以本章的程序示例为例，只要建立一个实现 State 接口的 XXXState 类，让它去实现必要的方法即可。当然状态迁移这个部分的程序代码也要小心，因为状态迁移正是与其他 ConcreteState 参与者的接点。

不过，如果想在已经完成的 State Pattern 中新增一个“有状态才会有处理”就没那么容易。因为这个新增的动作代表了要在 State 参与者的接口新增其他方法的意思，而且所有 ConcreteState 参与者都要新增其他的处理。

虽然这个动作有点困难，但却不必担心一不留神就会忘掉要新增方法。例如，现在要在本章程式示例的 State 接口新增一个 doYYY 的方法。如果忘记在 DayState 类或 NightState 类实现这个方法，当编译程序时，这里会发生“未实现方法”的程序错误。

如果没有使用 State Pattern，会发生什么情况？如果不只想用 State Pattern 的话，考虑状态就要交给 if 语句里的条件表达式。如此一来，就很难在编译程序时检测出类似“这个状态没有完全实现”的程序错误。（程序执行时检测出问题并不困难，不过不怕一万、只怕万一，预先设计成在各个方法里，若检测到意外的状态时就会抛出程序异常会比较保险。）

## 变化万千的对象实例

SafeFrame 类中出现了下面 2 种语句（p.368）。

- 在 SafeFrame 的构造函数内：

```
buttonUse.addActionListener(this);
```

- 在 actionPerformed 方法内：

```
state.doUse(this);
```

这两个语句里都有用到 this。这个 this 是指什么？当然两者都是 SafeFrame 类的对象实例。因为本章程式示例只会产生 1 个 SafeFrame 的对象实例，所以这两个 this 为同值。

只不过在传递给 actionPerformed 和 doUse 时的处理稍微有点不同。

在传递给 actionPerformed 方法时，这个对象实例被视为“实现 ActionListener 接口的类的对象实例”。因为 actionPerformed 方法的参数是 ActionListener 类型。在 actionPerformed 方法中，参数的使用范围是在“ActionListener 接口所规定方法的范围之内”。传递过来作为参数的是不是 SafeFrame 的对象实例根本不重要。

而在传递给 doUse 方法的时候，同样的对象实例却被视为“实现 Context 接口的类的对象实例”。因为 doUse 方法的参数是 Context 类型。在 doUse 方法中，参数的使用范围是在“Context 接口所规定方法的范围之内”（请重新翻回去看 DayState 类和 NightState 类的 doUse 方法就不难了解）。

请牢记：同一个对象实例会有多个不同的面貌。

## 相关 Pattern

### ◆ Singleton Pattern（第 5 章）

ConcreteState 参与者有时会被实现成 Singleton Pattern。本章的程序示例也利用到 Singleton Pattern，因为表示状态的类没有对象实例字段（即对象实例的状态）。

### ◆ Flyweight Pattern（第 20 章）

表示状态的类没有对象实例字段。因此有时会利用 Flyweight Pattern 让两个以上的 Context 参与者共享 ConcreteState 参与者。

## 重点回顾

本章所学习的是以不同的类来表示系统各种状态的 State Pattern。状态迁移则以切换表示状态之类的对象实例来表现。

## 练习题

解答请见附录 A

### 问题 1

理论上，Context 应该要设成抽象类而非接口、state 字段应该要归到 Context 类才符合 Pattern 的主旨，为什么本章的程序示例要用 Context 接口来表示 Context 参与者、state 字段甚至是归到 SafeFrame 类？试说明其理由。

### 问题 2

如果想把本章程序示例的白天 / 晚间时间范围修改如下表，请问应该修改源代码程序中的哪个部分？

|     | 白天         | 晚间                      |
|-----|------------|-------------------------|
| 修改前 | 9:00~16:59 | 17:00~23:59 和 0:00~8:59 |
| 修改后 | 8:00~20:59 | 21:00~23:59 和 0:00~7:59 |

### 问题 3

试在程序示例中加入“午餐时间”（12:00~12:59）这个新状态。

- 若午餐时间使用金库，则保安中心会接到发生异常状态的通知
- 若午餐时间使用警铃，则保安中心会接到警铃通知
- 若午餐时间使用电话，则会调用保安中心的录音机

### 问题 4

试在程序示例中加入如下的“紧急”这个新状态。让系统一旦发生紧急状况，无论任何时间都会通知保安中心。

- 按下警铃时，状态变为紧急状态
- 若紧急状态时使用金库，则保安中心会接到紧急状况的通知（无论任何时间）
- 若紧急状态时使用警铃，则保安中心会接到警铃通知（无论任何时间）
- 若紧急状态时使用电话，则会调用保安中心（无论任何时间）

但是这个程序设计有很大的问题，请问是什么样的问题？



# 第 20 章

## Flyweight (享元) ——有相同的部分 就共享，采取精简 政策

## Flyweight Pattern

本章要学习的是

### Flyweight Pattern

flyweight 是“蝇量级”的意思，它是拳击赛中体重最轻的等级。这个 Design Patterns 就是用来“减轻”对象。

在计算机的世界里，对象属于虚拟的存在，所以当我们说到“重”或“轻”并不是指实际的重量。这里所指的重其实是用来描述“内存使用量”。使用海量存储器的对象称为“较重”，使用内存不多的对象则属于“较轻”。

在 Java 当中，利用下面这个表达式可以创建 Something 类的对象。

```
new Something()
```

此时，系统会保留部分内存以存储这个对象实例。当你需要多个 Something 类的对象实例时，会需要做很多次 new，而这也会提高内存的使用量。

如果要用一句话来形容 Flyweight Pattern 所使用的技巧，应该就是：

“尽量共享对象实例，不做无谓的 new”

不是一需要对象实例就马上去 new，如果可以利用其他现有的对象实例，就让它们共享。这就是 Flyweight Pattern 的核心概念。废话不多说，马上就开始学习 Flyweight Pattern 吧。

## 程序示例

请看下面应用 Flyweight Pattern 的程序示例。假设现在有一个输出“大型字符”的类会产生较重的对象实例，需要输出多个字符产生一个“大型字符”。这里准备了数字 0'~'9' 和 '-' 作为测试用文件（表 20-1～表 20-11）。

表 20-1 数字 0 (big0.txt)

```
....#####
...##.....##
...##.....##
...##.....##
...##.....##
...##.....##
....###....
```

## Flyweight (享元) ——有相同的部分就共享，采取精简政策

表 20-2 数字 1 (big1.txt)

```
.....##.....
..##&##&##.....
.....##.....
.....##.....
.....##.....
.....##.....
.....##.....
..##&##&##&##&##.....
.....
```

表 20-3 数字 2 (big2.txt)

```
....##&##&##.....
..##.....##.....
.....##.....
.....##&##.....
.....##.....
..##.....##.....
..##&##&##&##&##.....
.....
```

表 20-4 数字 3 (big3.txt)

```
....##&##&##.....
..##.....##.....
.....##.....
.....##&##.....
.....##.....
..##.....##.....
.....##&##&##&##.....
.....
```

表 20-5 数字 4 (big4.txt)

```
.....##.....
.....##&##.....
.....##&##.....
..##....##.....
..##&##&##&##&##.....
```

## Java 布局设计的应用

```
.....##.....  
....######....  
.....
```

---

表 20-6 数字 5 (big5.txt)

```
....##########....  
....##.....  
....##.....  
....########....  
.....##....  
....##.....  
....##.....  
.....
```

---

表 20-7 数字 6 (big6.txt)

```
....######....  
....##.....##....  
....##.....  
....########....  
....##.....##....  
....##.....##....  
....######....  
.....
```

---

表 20-8 数字 7 (big7.txt)

```
....##########....  
....##.....##....  
.....##....  
.....##....  
.....##....  
.....##....  
.....##....  
.....
```

---

表 20-9 数字 8 (big8.txt)

```
....######....
```

## Flyweight (享元) ——有相同的部分就共享，采取精简政策

```
...##.....##....  
..##.....##....  
....###+#+##....  
..##.....##....  
..##.....##....  
....###+#+##....  
.....  
.....
```

---

表 20-10 数字 9 (big9.txt)

```
....#+#+##....  
..##.....##....  
..##.....##....  
....###+#+##....  
.....##....  
..##.....##....  
....###+#+##....  
.....
```

---

表 20-11 字符- (big-.txt)

```
.....  
.....  
.....  
.....  
..###+#+###+#+##....  
.....  
.....  
.....
```

---

上面程序示例所使用的类一览即如表 20-12 所示。

表 20-12 类一览表

| 名称             | 说明                        |
|----------------|---------------------------|
| BigChar        | 表示“大型字符”的类                |
| BigCharFactory | 共享并产生 BigChar 的对象实例的类     |
| BigString      | 表示多个 BigChar 所产生的“大型文本”的类 |
| Main           | 测试用的类                     |

## Java语言中的应用

`BigChar` 是表示“大型文本”的类。从文件读入大型文本的字符并存储在内存里，再以 `print` 方法输出。这种大型文本相当占用内存，所以我们要想办法该如何共享 `BigChar` 的对象实例。

`BigCharFactory` 类会产生 `BigChar` 类的对象实例。但若已经有一个对应同样字符的 `BigChar` 类的对象实例时就直接拿来用，不再产生新的对象实例。在这之前所产生的对象实例都存储在 `pool` 字段。利用 `java.util.Hashtable` 可快速判断出是否已经有对应指定字符的对象实例。

`BigString` 类是一个结合多个 `BigChar` 所产生的“大型字符串”的类。

`Main` 类是测试用的类。

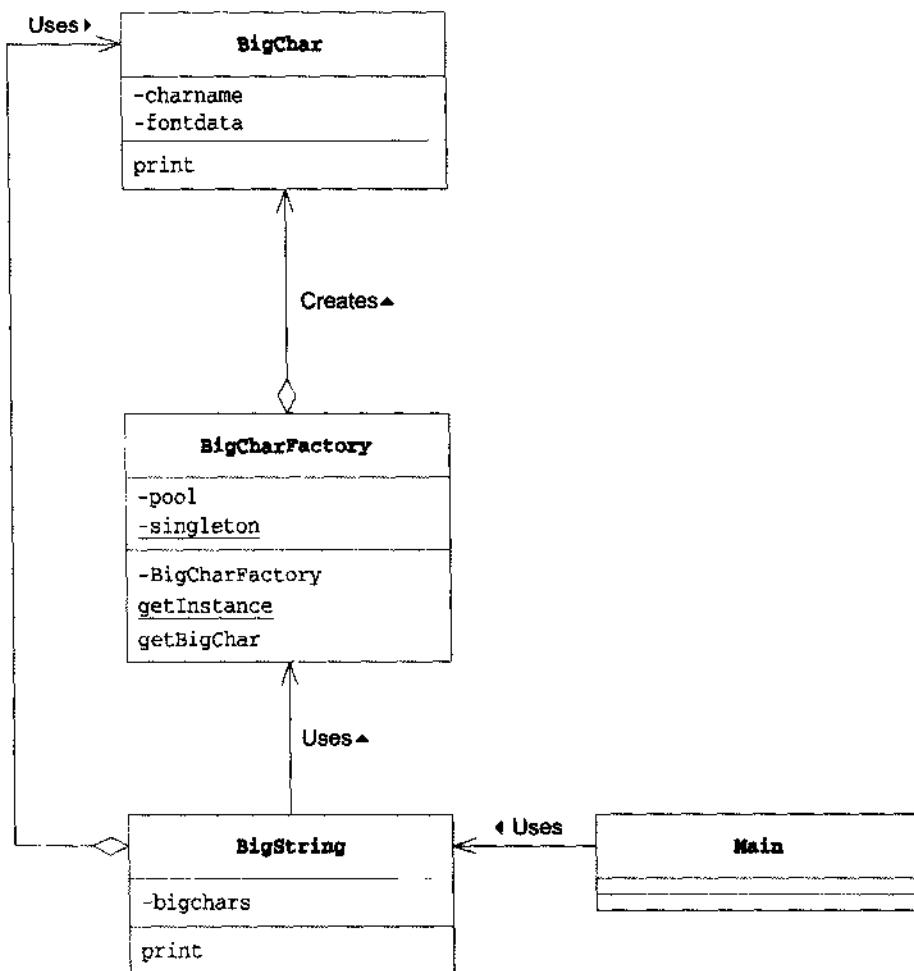


图 20-1 程序示例的类图

## Flyweight (享元) ——有相同的部分就共享，采取精简政策

**BigChar 类**

BigChar 类（程序 20-12）是表示“大型字符”的类。

在构造函数建立参数所接收到的指定字符的“大型字符”版本。所有产生的字符串都存储在 fontdata 字段。例如若参数接收到的是'3'，则存储在 fontdata 字段的会是下面的字符串：（由于可能看起来不太方便，所以\n 后面都多加了一个换行）

```
....##...#....\n
..##.....##....\n
.....##....\n
.....##...#....\n
.....##....\n
..##.....##....\n
....##...##....\n
.....\n
```

但是，所有组成“大型字符”的数据（即字符型数据）都要存储成文件格式（图表 20-1~表 20-11）。我们在此先订一个规则，文件的命名方式是在指定字符前附加字符串"big"，而以".txt"为扩展名。举例来说，'3'这个字符的字符型数据会存储在名为"big3.txt"的文件内。如果没有以上述命名规则的文件时，则字符型数据应为字符加"?"。

这个类还没用到 Flyweight Pattern 中跟“共享”有关的程序代码，控制共享的是程序 20-13 的 BigCharFactory 类。

**程序 20-1 BigChar 类 (BigChar.java)**

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class BigChar {
    // 字符名称
    private char chardata;
    // 输出成大型字符的字符串（有'#' '.' '\n'的行）
    private String fontdata;
    // 构造函数
    public BigChar(char chardata) {
        this.chardata = chardata;
        try {
            BufferedReader reader = new BufferedReader(
```

```

        new FileReader("big" + charname + ".txt")
    );
    String line;
    StringBuffer buf = new StringBuffer();
    while ((line = reader.readLine()) != null) {
        buf.append(line);
        buf.append("\n");
    }
    reader.close();
    this.fontdata = buf.toString();
} catch (IOException e) {
    this.fontdata = charname + "?";
}
}

// 输出大型字符
public void print() {
    System.out.print(fontdata);
}
}

```

---

## BigCharFactory 类

BigCharFactory 类（程序 20-2）是产生 BigChar 的对象实例的工厂（factory）。共享处理就在这里进行。

pool 字段管理现有的 BigChar 的对象实例。pool 的英文原意是游泳池，只要是存储用的位置就称为 pool。现实生活中的游泳池里面是水，BigCharFactory 的 pool 里面则是所有已经产生的 BigChar 的对象实例。

这里的 java.util.Hashtable 是管理“字符串→对象实例”这个对应关系的类。利用 java.util.Hashtable 类的 put 方法，让 1 个字符串（按钮）对应 1 个对象实例（值）。利用 get 方法可取得按钮所对应的值。本章的程序示例是根据 1 个指定字符（如'3'），去对应输出 3 的 BigChar 的对象实例。

在此，BigCharFactory 类利用到 Singleton Pattern（第 5 章），因为只要有 1 个 BigCharFactory 的对象实例即可。getInstance 方法是取得 BigCharFactory 的对象实例（不是 BigChar 的对象实例）的方法。

接下来的 getBigChar 方法是整个 Flyweight Pattern 最主要的方法。这个方法会产生参数所对应的 BigChar 的对象实例。但如果已经有一个对应同样字符的对象实例时，则拿现有的对象实例作为返回值，而不另外产生新的对象实例。

## Flyweight (享元) ——有相同的部分就共享，采取精简政策

我们再复习一下这里的逻辑概念。首先，以 pool.get 检查是否已经存在与指定字符 (charname) 相对应的 BigChar 的对象实例。如果返回值为 null，则表示目前尚无该对象实例，故再以 new 产生 BigChar 的对象实例。接下来利用 pool.put 把该对象实例注册进去。而如果 pool.get 的返回值非 null，则返回值为现有的对象实例。

各位从上面的处理过程，应该能发现这样可以巧妙地共享 BigChar 的对象实例。为什么 getBigChar 方法会变成 synchronized，这就留在练习题 20-3 中再来动动脑筋。

**程序 20-2** BigCharFactory 类 (BigCharFactory.java)

```
import java.util.Hashtable;

public class BigCharFactory {
    // 管理现有的 BigChar 的对象实例
    private Hash 表 pool = new Hashtable();
    // Singleton Pattern
    private static BigCharFactory singleton = new BigCharFactory();
    // 构造函数
    private BigCharFactory() {
    }
    // 取得惟一的对象实例
    public static BigCharFactory getInstance() {
        return singleton;
    }
    // 产生 (共享) BigChar 的对象实例
    public synchronized BigChar getBigChar(char charname) {
        BigChar bc = (BigChar)pool.get("'" + charname);
        if (bc == null) {
            bc = new BigChar(charname); // 在此产生 BigChar 的对象实例
            pool.put("'" + charname, bc);
        }
        return bc;
    }
}
```

## BigString 类

BigString 类（程序 20-3）是结合多个 BigChar 形成“大型字符串”的类。

bigchars 字段是 BigChar 的数组，存储 BigChar 的对象实例。请看构造函数的 for 语句，它利用 getBigChar 写成：

## Java语言中的应用

```
BigCharFactory factory = BigCharFactory.getInstance();
for (int i = 0; i < bigchars.length; i++) {
    bigchars[i] = factory.getBigChar(string.charAt(i)); ←共享
}
```

如此一来，new 就不能像这样做了。

```
for (int i = 0; i < bigchars.length; i++) {
    bigchars[i] = new BigChar(string.charAt(i));           ←不共享
}
```

因为使用到 BigCharFactory，所以同样的字符所产生的 BigChar 对象实例应该要能够共享才对。举例来说，欲产生对应字符串"1212123"的 BigString 的对象实例时，其 bigchars 字段应如图 20-2。

程序 20-3 BigString 类 (BigString.java)

---

```
public class BigString {
    // “大型字符”的数组
    private BigChar[] bigchars;
    // 构造函数
    public BigString(String string) {
        bigchars = new BigChar[string.length()];
        BigCharFactory factory = BigCharFactory.getInstance();
        for (int i = 0; i < bigchars.length; i++) {
            bigchars[i] = factory.getBigChar(string.charAt(i));
        }
    }
    // 显示
    public void print() {
        for (int i = 0; i < bigchars.length; i++) {
            bigchars[i].print();
        }
    }
}
```

---

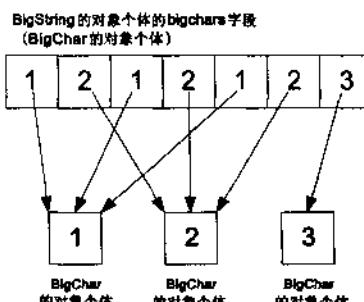


图 20-2 对应字符串"1212123"的 BigString 的对象实例

## Flyweight (享元) ——有相同的部分就共享，采取精简政策

## Main 类

Main 类（程序 20-4）比较简单。它只是根据参数所接收到的字符串，产生并输出 BigString 的对象实例。

程序 20-4 Main 类 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Usage: java Main digits");
            System.out.println("Example: java Main 1212123");
            System.exit(0);
        }

        BigString bs = new BigString(args[0]);
        bs.print();
    }
}
```

## Flyweight Pattern 的所有参与者

Flyweight Pattern 的所有参与者可整理如下。类图请见下页。

## ◆ Flyweight (享元) 参与者

表示以一般处理会让程序变复杂，因此选择共享较好的参与者。在本章的程序示例，扮演这个角色的是 BigChar 类。

## ◆ FlyweightFactory (享元工厂) 参与者

产生 Flyweight 参与者的工厂。利用这个工厂来产生 Flyweight 参与者，即可共享对象实例。在本章的程序示例，扮演这个角色的是 BigCharFactory 类。

## ◆ Client (客户) 参与者

利用 FlyweightFactory 参与者产生并使用 Flyweight 参与者。在本章的程序示例，扮演这个角色的是 BigString 类。

注意：这里的参与者分配跟 GoF 一书中所指的参与者不尽相同。GoF 一书所提到的是 ConcreteFlyweight 参与者和 UnsharedConcreteFlyweight 参与者。GoF 的 ConcreteFlyweight 参与者相当于本章的 Flyweight 参与者；而本章的程序示例并没有使用到 UnsharedConcrete Flyweight 参与者。

## 扩展自我视野的提示

### 影响会传到所有相关位置

Flyweight Pattern 的焦点还是在于“共享”对象实例，那么在共享对象实例时要注意哪些问题呢？

第一个马上浮现脑海的就是“共享的部分一旦被修改，就会影响到有关联的全体”。换句话说，只要修改 1 个对象实例就能同时把修改内容反映到用到该对象实例的所有地方。假设现在需要修改程序示例的 BigChar 类中‘3’的 fontdata，那么所有 BigString 所使用的‘3’的字型都会跟着改变。像这样影响到所有相关位置的效果并不一定每次都不好，还是要看程序所处理的内容而定，所以请各位千万记得：共享具有这样的特性“只要修改 1 个，影响就会传到所有相关位置”。

因为这个特性，我们必须小心翼翼选择 Flyweight 参与者的信息。只有真正需要让多个位置共享的信息才交给 Flyweight 参与者，这样就万事 OK 了。

举个例子来说，假设现在想在本章的程序示例中新增其他功能，例如输出“有文字颜色的大型字符串”。此时，“文字颜色”的信息应该要交给哪个类？

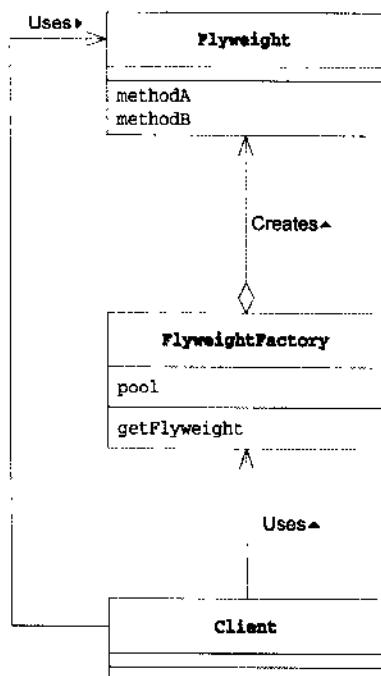


图 20-3 Flyweight Pattern 的类图

## Flyweight (享元) —— 有相同的部分就共享，采取精简政策

假设先交给 BigChar。由于 BigChar 的对象实例属于共享，所以文字颜色的信息也会变成共享。换句话说，BigString 当中有相同 BigChar 的对象实例都会是同样的文字颜色。

那么，如果“文字颜色”的信息是交给 BigString 而不是 BigChar 时，又会怎样？变成 BigString 管理“第 3 个字符是红色”的文字颜色信息。在这个情况之下，即使是相同的 BigChar 的对象实例也可能会有不同的文字颜色。

这两种做法无法分出哪个才是最正确的，毕竟要共享、或不共享哪些信息都要视该类的使用目的而定。如果你想利用 Flyweight Pattern 设计类时，一定要搞清楚应该要让哪些部分共享！

## Intrinsic 和 extrinsic

前面所提到的“要共享的信息 / 不要共享的信息”都有特殊的正式名称。

要共享的信息称为 intrinsic 型信息。intrinsic 的英文原意是“原本具有的”“本质上的”。换句话说，它就是无论该对象实例放在哪里，在任何情况下都不会改变的信息，或不受状态影响的信息。BigChar 的字符型数据无论是放在 BigString 的哪个位置都不会有变化，所以 BigChar 的字符型数据属于 intrinsic 型信息。

不要共享的信息则称为 extrinsic 型信息。extrinsic 的英文原意是“从外部而来的”“非本质的”；指会因对象实例所在位置或情况而变的信息，或会受状态影响的信息。会因所在位置而变的信息当然就不能共享了。例如，“BigChar 的对象实例是 BigString 的第几个字符”的信息会因 BigChar 所在位置而改变，所以当然不能放到 BigChar。这种信息就称为 extrinsic 型信息。

前面所提到的“文字颜色”信息究竟应不应该放在 BigChar 的问题其实可以换另一个方向来思考：文字颜色应该视为 intrinsic 型信息还是 extrinsic 型信息？

表 20-2 intrinsic 型信息与 extrinsic 型信息

|               |                  |
|---------------|------------------|
| intrinsic 型信息 | 不受地点或状况的影响，故可共享  |
| extrinsic 型信息 | 不受地点或状况的影响，故不可共享 |

## 管理中的对象实例不会被视为垃圾

BigCharFactory 利用 java.util.Hashtable 来管理已产生的 BigChar 的对象实例。如果想在 Java 里也能有这种“管理对象实例”的功能，一定要注意“管理中的对象实例不会被视为垃圾”。

简单说明一下垃圾收集（garbage collection）。Java 程序会利用 new 来确保足够的内存空间。当内存空间保留越多，就会渐渐发生内存空间不足的问题。内存空间不足时，执行 Java 程序的虚拟机器便会开始进行垃圾收集的处理动作。垃圾收集是检查本

身的内存空间（堆积领域），释放未使用的对象实例以增加可用内存空间的处理。简单的说，就是像垃圾车那样收集不用的内存空间再做资源回收的意思。

这个垃圾收集的功能让 Java 语言的程序员能安心地对前面 new 过的对象实例置之不理（如果是 C++，对于以 new 的方式所保留的内存空间，程序员必须确实使用 delete 才能释放掉。相比之下，Java 不需要做 delete，而且 Java 本来就没有 delete 函数）。

这里的重点是垃圾收集中“释放不使用的对象实例”的部分。进行垃圾收集时当然会对所有对象实例判断其是否为垃圾，但是有被其他部分参照的对象实例则视为“在使用”，判断它不是垃圾。

再回到本章的程序示例，pool 字段是管理现有 BigChar 的对象实例。因此 pool 字段所管理的对象实例不会被视为垃圾。举例来说，假设事实上已经不会再从 BigString 的对象实例去参照 BigChar 的对象实例。但是，这些产生出来的 BigChar 类的对象实例还是一直残留在内存里面。本章的程序示例在输出字符串后就立即结束了，所以不至于有内存空间不足的情形，但如果是设计需要长时间连续执行的程序，或只能以少量内存空间执行的程序时，仍然要记得：“管理中的对象实例不会变成垃圾”。

虽然我们不能直接删除对象实例，但可以删除对对象实例的引用操作。如果想要让管理中的对象实例变成垃圾，必须删除引用操作，让它不再受人管理。例如，只要删除 Hashtable 中含有该对象实例的进入点，即可删除对对象实例的引用操作。

## 非内存部分的记录

在本章的程序示例中，只要让对象实例共享，就能降低内存的使用量。一般说来，只要共享对象实例，确实可以减少产生对象实例时的记录量。记录是一种计算机资源，内存就是记录之一。

时间也是记录之一。假设现在要 new 对象实例时，会花掉一段时间。但如果利用 Flyweight Pattern 让对象实例能共享的话，就可以减少 new 对象实例的次数。这样能提高程序的执行速度。

文件处理程序（file handler；file descriptor）和窗口处理程序（window handler）等也都是一种记录。有些 OS 对于可同时使用的文件处理程序或窗口处理程序数量有限，遇到这种情形，如果不先设好共享对象实例，当超过限制数量时就会发生程序停止无法继续执行的问题。

Flyweight (享元) —— 有相同的部分就共享，采取精简政策

## 相关 Pattern

### ◆ Proxy Pattern (第 21 章)

Flyweight Pattern 如果遇到产生对象实例较费时，利用对象实例共享的方式即可提高处理速度。

Proxy Pattern 则是利用建立代理人的方式来提高处理速度。

### ◆ Composite Pattern (第 11 章)

有时可利用 Flyweight Pattern 让 Composite Pattern 的 Leaf 参与者能共享。

### ◆ Singleton Pattern (第 5 章)

FlyweightFactory 参与者、Singleton Pattern。

此外，Singleton Pattern 的 Singleton 参与者只能产生 1 个对象实例，因此使用到该对象实例的地方都要变成共享。Singleton 参与者的对象实例只有 intrinsic 型信息。

## 重点回顾

本章所学习的是利用对象实例共享 (share) 的方式以降低使用内存空间的 Flyweight Pattern。共享的对象实例一被修改，就会影响到所有相关位置。因此必须确实区分哪些是应该共享的 intrinsic 型信息，哪些是不可设为共享的 extrinsic 型信息。

## 练习题

解答请见附录 A

### 问题 1

请在程序示例的 BigString 类（程序 20-3）新增如下的构造函数。

`BigString(String string, boolean shared)`

若 shared 为 true 则共享 BigChar；若为 false 则不共享。

### 问题 2

请利用问题 20-1 所修改的 BigString 类（程序 A20-1），比较共享和不共享 BigChar 的对象实例时的内存使用量。

**注意：**请参考下面的步骤，取得目前实际使用的内存使用量。为了能更精确地取得内存使用量的数据，执行前要用 `gc` 方法强迫进行垃圾收集。

```
Runtime.getRuntime().gc();
```

## Java语言中的应用

```
long used = Runtime.getRuntime().totalMemory() - Runtime.getRuntime()
().freeMemory();
System.out.println("已使用内存 = " + used);
```

### 问题 3

在程序示例的 BigCharFactory 类（程序 20-2）中，getBigChar 方法是 synchronized 方法。如果不设为 synchronized 方法，会出现什么样的问题？

# 第 21 章

## Proxy (代理) —— 要用再建立

# Proxy Pattern

本章要学习的是

## Proxy Pattern

proxy 这个名词有“代理”的意思。望文生义，代理就是那个代替（代理）本来应该自己动手做事的本人的人。因为要把不需要本人亲自处理的工作交代出去，所以才会建立一个代理。由于代理纯粹只是代理工作而已，因此能力范围也有限。如果遇到超出代理能力所及的范围，代理就应该去找本人商量才对。

在面向对象中，“本人”和“代理”都是对象。代理对象是代替忙到无法自己动手的本人对象去处理工作（有一定范围）。

## 程序示例

请看下面应用了 Proxy Pattern 的程序示例。

本章的程序示例是一个“有命名的打印机”。虽然说是打印机，不过实际上也只是把字符串输出到画面上而已。由 Main 类产生 PrinterProxy 类的对象实例（即“代理”）。该对象实例命名为“Alice”，并将此名称输出到画面上。然后改名为“Bob”，且输出该新名称。在命名和取得名称的阶段都还没有产生真正的 Printer 类的对象实例（即“本人”），命名和取得名称的部分由 PrinterProxy 类代理执行。最后调用 print 方法进入真正执行打印的阶段，才由 PrinterProxy 类产生 Printer 类的对象实例。类图和流程图分别如图 21-1、图 21-2 所示。

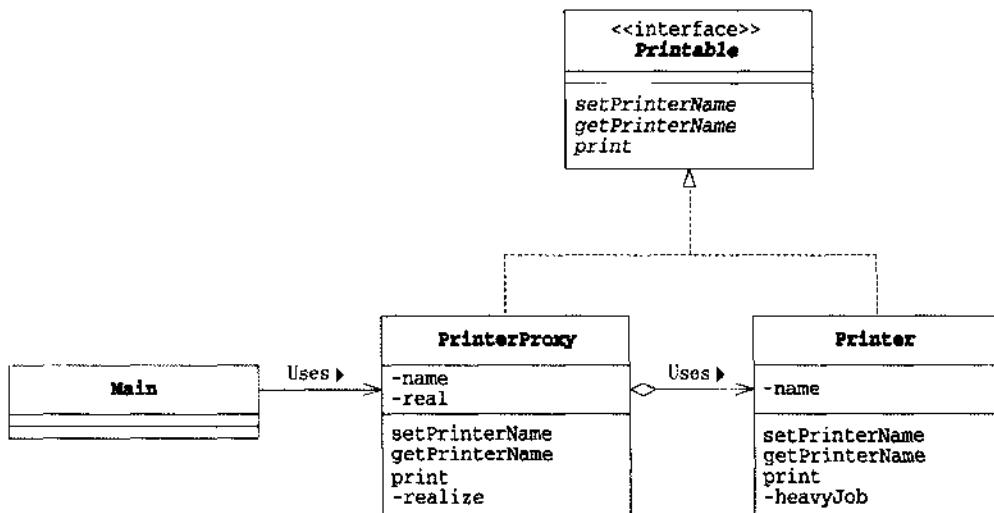


图 21-1 程序示例的类图

## Proxy (代理) —— 要用再建立

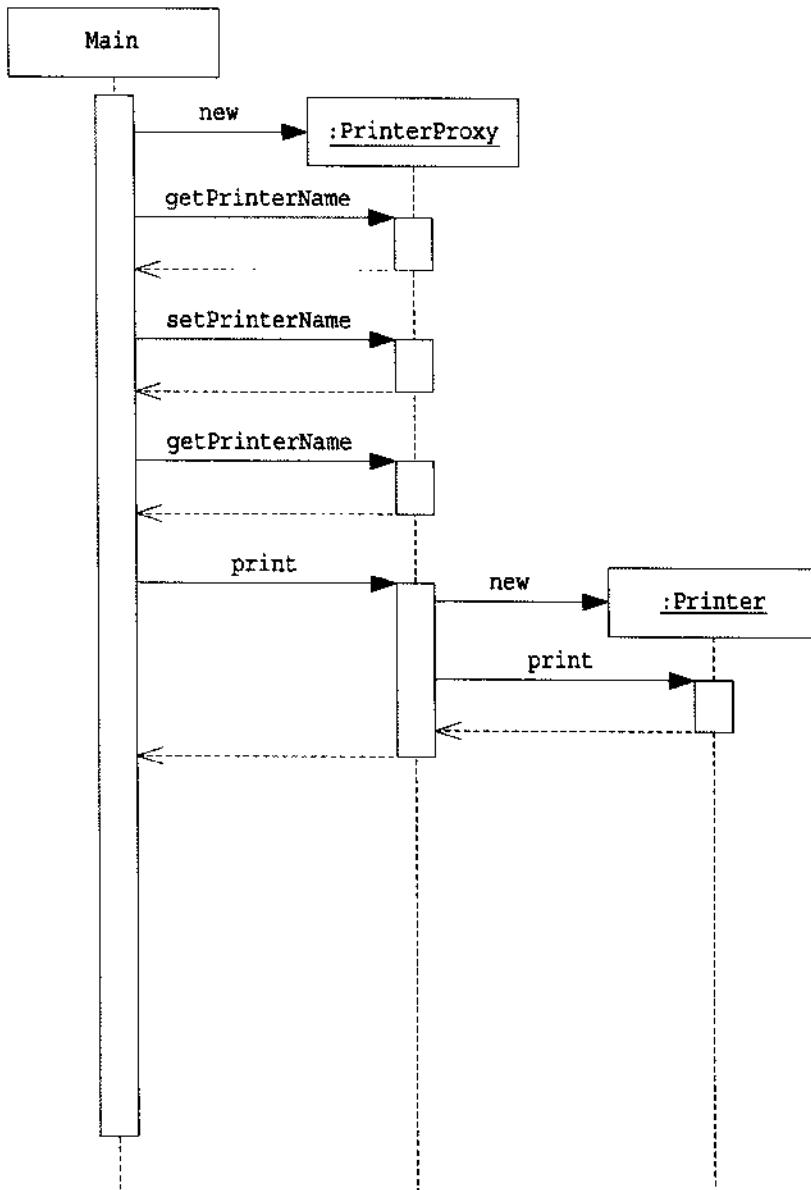


图 21-2 程序示例的流程图

**PrinterProxy** 类和 **Printer** 类要一视同仁，故须另定义 **Printable** 接口。

这个程序示例的前提条件是产生 **Printer** 类的对象实例会花很多时间。为了表现出相当费时的感觉，所以从构造函数故意调用 **heavyJob** 方法去做“较重的工作”（需要 **heavyJob** 方法运行 5 秒左右）。

表 21-1 类与接口一览表

| 名称           | 说明                           |
|--------------|------------------------------|
| Printer      | 表示命名的打印机的类（本人）               |
| Printable    | Printer 和 PrinterProxy 共享的接口 |
| PrinterProxy | 表示命名的打印机的类（代理）               |
| Main         | 测试用的类                        |

## Printer 类

Printer 类（程序 21-1）是表示“本人”的类。

如前所述，在构造函数的部分会执行虚拟的“较重的工作”heavyJob。

setPrinterName 是命名的方法，getPrinterName 则是取得名称的方法。

print 方法是输出字符串（打印机的名称）。

heavyJob 方法则是表示执行时会花上 5 秒的较重工作。以每秒（1000 毫秒）输出 1 个句点（.）的方式显示。

Proxy Pattern 的核心在 PrinterProxy 类的部分，Printer 类没有什么困难的地方。

程序 21-1 Printer 类 (Printer.java)

```
public class Printer implements Printable {
    private String name;
    public Printer() {
        heavyJob("正在产生 Printer 的对象实例");
    }
    public Printer(String name) {           // 构造函数
        this.name = name;
        heavyJob("正在产生 Printer 的对象实例(" + name + ")");
    }
    public void setPrinterName(String name) { // 命名
        this.name = name;
    }
    public String getPrinterName() {         // 取得名称
        return name;
    }
    public void print(String string) {       // 输出名称
        System.out.println("==" + name + " ==");
        System.out.println(string);
    }
    private void heavyJob(String msg) {      // 较重的工作（假设）
    }
}
```

## Proxy (代理) —— 要用再建立

```

        System.out.print(msg);
        for (int i = 0; i < 5; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
            }
            System.out.print(".");
        }
        System.out.println("完成。");
    }
}

```

---

## Printable 接口

Printable 接口（程序 21-2）是对 PrinterProxy 类和 Printer 类一视同仁的接口。setPrinterName 方法是命名，getPrinterName 方法为取得名称，而 print 方法是打印（输出字符串）。

**程序 21-2** Printable 接口（Printable.java）

```

public interface Printable {
    public abstract void setPrinterName(String name); // 命名
    public abstract String getPrinterName(); // 取得名称
    public abstract void print(String string); // 输出字符串（打印）
}

```

---

## PrinterProxy 类

PrinterProxy 类（程序 21-3）扮演代理的角色，实现 Printable 接口。

name 字段存储名称，real 字段则存储“本人”。

在构造函数的部分进行命名（此时还没有建立“本人”）。

setPrinterName 方法是重新命名。如果 real 不为 null（即“本人”已经存在），则是命名本人。但若 real 为 null（即尚未有“本人”时），则只命名自己（PrinterProxy 的对象实例）的 name 字段。

getPrinterName 方法只传回自己的 name 字段的值。

print 方法是非代理能力范围的处理，故调用 realize 方法以产生本人。realize 有“成为现实”的意思（让它变成实际的存在）。执行 realize 方法之后，本人（Printer 类的对象实例）会存储在 real 字段，并调用 real.print。记得吗，这是“委托”。

无论调用多少次 setPrinterName 或 getPrinterName，都不会产生 Printer 的对象实

## Java语言中的应用

例。只有在真正需要用到“本人”时，才会产生 Printer 的对象实例（使用 PrinterProxy 的人并不知道是否有产生本人，而且也没必要知道）。

realize 方法比较简单。若 real 字段为 null，以 new Printer 产生 Printer 的对象实例。如果 real 字段不为 null（即已经产生），则不做任何操作。

这里要请各位记住一件事：Printer 类并不知道 PrinterProxy 的存在。Printer 类不会知道自己是直接被调用，还是辗转通过 PrinterProxy 被调用。

另一方面，PrinterProxy 类却知道 Printer 类的事。因为 PrinterProxy 类的 real 字段是 Printer 类型。PrinterProxy 类的源代码里本来就有 Printer 这个类名称，所以 PrinterProxy 类才会变成跟 Printer 类有密切关系的零件（练习题 21-1 会继续研究分割的方法）。

相信细心的读者们已经发现 setPrinterName 方法和 realize 方法都是 synchronized 方法。理由留到练习题 21-2，再好好想想吧。

程序 21-3 PrinterProxy 类 (PrinterProxy.java)

```
public class PrinterProxy implements Printable {
    private String name;           // 命名
    private Printer real;          // “本人”
    public PrinterProxy() {
    }
    public PrinterProxy(String name) { // 构造函数
        this.name = name;
    }
    public synchronized void setPrinterName(String name) { // 命名
        if (real != null) {
            real.setPrinterName(name); // “本人”也要命名
        }
        this.name = name;
    }
    public String getPrinterName() { // 取得名称
        return name;
    }
    public void print(String string) { // 输出
        realize();
        real.print(string);
    }
    private synchronized void realize() { // 产生“本人”
        if (real == null) {
            real = new Printer(name);
        }
    }
}
```

## Proxy (代理) —— 要用再建立

```

    }
}

}

```

**Main 类**

Main 类（程序 21-4）是通过 PrinterProxy 再去利用 Printer 类。这个类一开始先产生 PrinterProxy，再利用 getPrinterName 输出名称。接着以 setPrinterName 命名，最后才用 print 输出“Hello, world.”。

请看一下执行结果（图 21-3），验证 Printer 的对象实例（本人）并不是在命名和输出到画面之间产生，而是要等到调用 print 方法之后才会产生。

**程序 21-4 Main 类 (Main.java)**

```

public class Main {
    public static void main(String[] args) {
        Printable p = new PrinterProxy("Alice");
        System.out.println("现在的名称是" + p.getPrintName() + "。");
        p.setPrinterName("Bob");
        System.out.println("现在的名称是" + p.getPrintName() + "。");
        p.print("Hello, world.");
    }
}

```

```

现在的名称是 Alice。
现在的名称是 Bob。
正在产生 Printer 的对象实例 (Bob) ..... 完成。
==== Bob ===
Hello, world.

```

图 21-3 执行结果

**Proxy Pattern 的所有参与者**

Proxy Pattern 的所有参与者可整理如下。

◆ **Subject (主体) 参与者**

规定对 Proxy 参与者和 RealSubject 参与者一视同仁的接口 (API)。因为已经有 Subject 参与者，所以 Client 参与者不需要去注意 Proxy 参与者跟 RealSubject 参与者有

## Proxy 模式中的应用

什么差异。在本章的程序示例中，扮演这个角色的是 `Printable` 接口。

### ◆ Proxy (代理) 参与者

Proxy 参与者会尽量处理 Client 参与者的要求。当自己无法单独处理时，Proxy 参与者便会把工作交给 RealSubject 参与者。要等到真正需要用到 RealSubject 参与者时，Proxy 参与者才会产生 RealSubject 参与者。Proxy 参与者是实现 Subject 参与者规定的接口（API）。在本章的程序示例中，扮演这个角色的是 `PrinterProxy` 类。

### ◆ RealSubject (实际主体) 参与者

当“代理人”Proxy 参与者束手无策时，就轮到“本人”RealSubject 参与者自己上场。这个参与者跟 Proxy 参与者同样都要实现 Subject 参与者规定的接口（API）。在本章的程序示例中，扮演这个角色的是 `Printer` 类。

### ◆ Client (客户) 参与者

利用 Proxy Pattern 的参与者（在 GoF 一书中，Proxy Pattern 里并没有 Client 参与者）。在本章的程序示例中，扮演这个角色的是 `Main` 类。

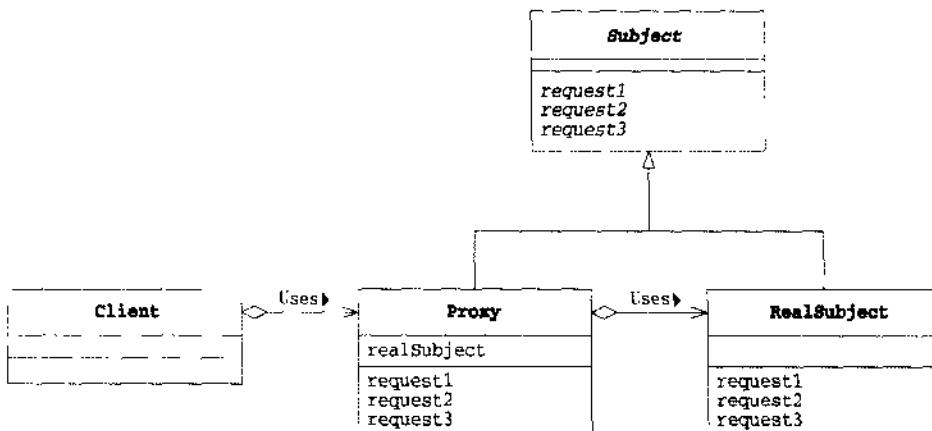


图 21-4 Proxy Pattern 的类图

## 扩展自我视野的提示

### 利用代理提高处理速度

在 Proxy Pattern 当中，Proxy 参与者发挥代理的功能尽力处理代理的工作。例如，本章的程序示例利用 Proxy 参与者可以把较重的处理（产生对象实例）往后延迟到实际 print 为止。

## Proxy (代理) —— 要用再建立

程序示例中这个较重的处理只有文字上的说明，不太容易有实际的延迟停滞感。各位不妨想象一下假设这是一个大型系统，里面大多数的功能在初始化时都要花很多时间。在启动的时候，所有功能（连不用的功能）都要全面初始化的话，当然会连带影响到应用软件也要花很多时间才能启动，这样只会让用户抱怨连连。把程序设计到了实际要使用某种功能的阶段才初始化该功能，比较有助于节省用户的时间。

GoF 一书所举的 Proxy Pattern 示例是一个内嵌图形对象（如图文件等）的文件编辑器。因为欲产生图形对象时，读取图文件会很花时间，所以每当开启文件都要产生图形对象就会浪费时间。其实等到图形对象要输出到画面上时再去产生会比较好，而 Proxy Pattern 在此时就能适时发挥它的作用。

### 不能让代理和本人合二为一吗

你也可以选择另一种做法：不分成 PrinterProxy 类和 Printer 类，直接一开始就在 Printer 类里加入延迟评估的功能（如果需要才产生对象实例的功能）。不过把 Proxy 参与者和 RealSubject 参与者两者分开来才能提高程序零件化的程度，允许个别修改（就是要分割统合！）。

当你一改变 PrinterProxy 类的实现，就可以修改 Printable 接口所声明的方法，哪些由代理处理、哪些要由本人处理。而且不管你想改多少次、多少地方，都不必再去修改 Printer 类。如果完全不想执行延迟评估的话，Main 类就不要去 new PrinterProxy 类的对象实例，改成 new Printer 类的对象实例即可。PrinterProxy 类和 Printer 类都是实现 Printable 接口，所以 Main 类能放心大胆切换使用 PrinterProxy 或 Printer。

PrinterProxy 这个类是表示“Proxy 参与者”的功能。因此要不要利用这个功能就是选择要不要使用 PrinterProxy。

### 代理和委托

代理只能处理它有能力处理的工作，若代理无法处理时，就要“交给”有能力处理的本人去做。这个“交给”就是本书中多次提到的“委托”。从 PrinterProxy 类的 print 调用 real.print 的地方就是“委托”。

现实世界里原本是本人将责任委托给代理，设计 Pattern 中所指的委托刚好跟这个原则相反。

### 穿透性

PrinterProxy 类跟 Printer 类是实现同一个 Printable 接口。Main 类并不在乎实际调用到的是 PrinterProxy 类、还是 Printer 类。直接用 Printer 或中间多隔一个 PrinterProxy 都同样能正常使用。

这时候我们可以说 PrinterProxy 类具有“穿透性”。就好像即使墙上挂的画跟自己站的位置中间有一片透明玻璃，我们还是能看得见那幅画；Main 类跟 Printer 类中间即使多了 PrinterProxy 类，也不会有问题。

## HTTP proxy

一看到“proxy”，有些读者可能会马上联想到 HTTP proxy。HTTP proxy 是指介于 HTTP 服务器（Web 服务器）和 HTTP 客户端（Web 浏览器）之间、有 Web 网页缓存等功能的软件程序。这个观念也可以拿来说明 Proxy Pattern。

HTTP proxy 的功能种类繁多，在此仅举网页缓存（caching）为例做说明。

欲利用 Web 浏览器浏览 Web 网页时，不需要每次存取远程 Web 服务器才能取得网页，HTTP proxy 即可代替取得已经缓存到的网页。只有在需要取得最新信息或网页已过有效期限时，才要到 Web 服务器去撷取 Web 网页。

在以上的关系中，Web 浏览器是 Client 参与者，HTTP proxy 是 Proxy 参与者，而 Web 服务器则是 RealSubject 参与者。

## 各种 Proxy

Proxy Pattern 常见情况：

### ◆ Virtual Proxy (虚拟 proxy)

Virtual Proxy 就是本章所介绍的 Proxy Pattern。真正需要用到对象实例时，才产生或进行初始化。

### ◆ Remote Proxy (远程 proxy)

即使 RealSubject 参与者在网络的另一端，Remote Proxy 仍然可以自由调用方法，就好像伸手拿起身旁物品一样（穿透性）。例如：Java 的 RMI（Remote Method Invocation：远程方法调用）等。

### ◆ Access Proxy

Access Proxy 是用来对 RealSubject 参与者的功能设定存取限制。这个 proxy 只限已经登录的用户才能调用方法，未登录的用户则视为程序错误。

## 相关 Pattern

### ◆ Adapter Pattern (第 2 章)

Adapter Pattern 的功能是连接两个相异接口（API）的对象。而 Proxy Pattern 的 Proxy 参与者和 RealSubject 参与者两者的接口（API）并没有不一样（穿透性）。

## Proxy (代理) —— 要用再建立

### ◆ *Decorator Pattern* (第 12 章)

Decorator Pattern 跟 Proxy Pattern 的实现很相似，但目的却不尽相同。Decorator Pattern 的目的是为了新增其他功能，而 Proxy Pattern 则比较着重在代替本人进行作业，以减少对本人的存取操作，新增其他功能反而没那么重要。

## 重点回顾

本章所学习到的 Proxy Pattern 是指定代理执行工作、直到真的需要找本人出来解决为止。

## 练习题

解答请见附录 A

### 问题 1

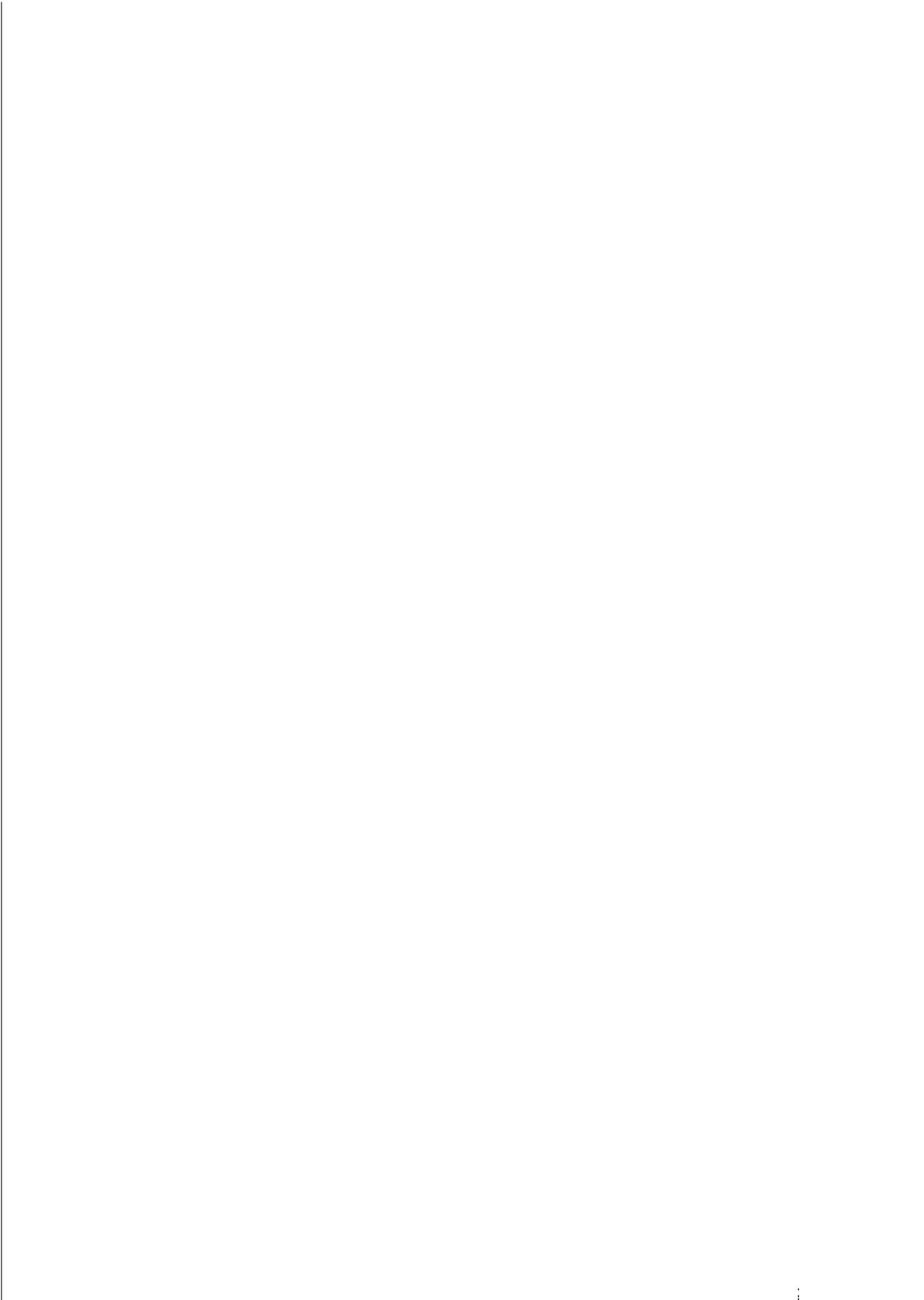
在本章的程序示例中，PrinterProxy 类（程序 21-3）知道 Printer 类（程序 21-1）；也就是说 Printer 这个类名称就直接写在 PrinterProxy 类内。

请把 PrinterProxy 类修改成：即使 PrinterProxy 类“不知道 Printer 类也没有影响”。

**提示：**方法不只一种，例如以字符串格式把 RealSubject 参与者的类名称传递给 PrinterProxy 类的构造函数。

### 问题 2

在 PrinterProxy 类（程序 21-3）中，把 setPrinterName 方法和 realize 方法设为 synchronized 方法。请举例说明若不设为 synchronized 方法时会发生什么样的问题。



# 第 22 章

## Command (命令) ——将命令写成类



## Command Pattern

当类在执行操作时，会调用自己类或其他类的方法。调用方法之后的结果会反映在对象状态上，但却不会留下任何操作记录。

遇到这种情形时，要是有一个类能表现“请执行这项作业”的“命令”就方便多了。因为如此一来便可用1个“表示命令的类的对象实例”来代表欲执行的操作，而不需采用“调用方法”的类的动态处理。如欲管理相关记录，只须管理该对象实例的集合即可。而若预先将命令的集合存储起来，还可再执行同一命令；或者是把多个命令结合成一个新命令供再利用。

设计Pattern将这样的“命令”命名为：

Command Pattern

Command有“命令”的意思。

Command有时也称为Event (event是“事件”的意思)，跟“事件驱动程序语言”中所使用的“事件”同义。一旦发生事件（如按下鼠标左键、或按下按键）时，则先将该事件变成对象实例，按照发生顺序排入等候序列内。接着，再依次处理所有排列等待的事件。像是GUI (graphical user interface) 相关的程序语言中，经常会使用到这种“事件”。

本章要学习的就是处理“命令”的Command Pattern。

## 程序示例

请看下面应用Command Pattern的程序示例，这个程序示例是一个简单的绘图软件。移动鼠标时，便会自动绘制一个个红点，按下[clear]键即可清除所有图样。



图 22-1 执行结果

## Command（命令）——将命令写成类

当用户移动鼠标时，就会产生“在这里画一个点”的命令（DrawCommand 类的对象实例）。把这个对象实例存储起来，需要用到的时候就可以继续绘制红点。

程序示例中所有类和接口的一览表如表 22-1 所示。程序示例共分成 3 个不同的包，

表 22-1 类与接口的一览表

| 包       | 名称           | 说明              |
|---------|--------------|-----------------|
| command | Command      | 表示“命令”的接口       |
| command | MacroCommand | 表示“结合多个命令的命令”的类 |
| drawer  | DrawCommand  | 表示“点的绘制命令”的类    |
| drawer  | Drawable     | 表示“绘制对象”的接口     |
| drawer  | DrawCanvas   | 表示“绘制对象”的类      |
| 未命名     | Main         | 测试用的类           |

command 包和 drawer 包分别处理“命令”、“绘制”的动作。Main 类则放在未命名包里。

### Command 接口

Command 接口（程序 22-1）是表示“命令”。这个接口只有 1 个方法—execute（execute 是“执行”的意思）。规定调用 execute 方法时，实际执行的是实现 Command 接口的类，不过“执行”的还是 Command 接口。

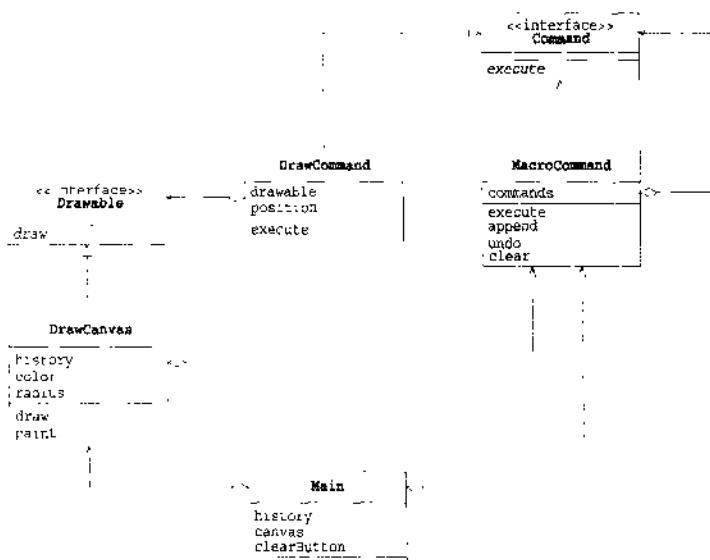


图 22-2 程序示例的类图

**程序 22-1 Command 接口 (Command.java)**

```
package command;

public interface Command {
    public abstract void execute();
}
```

## MacroCommand 类

MacroCommand 类（程序 22-2）是表示“结合多个命令的命令”。此类实现 Command 接口。MacroCommand 当中的 macro 本来是“大”的意思，但在程序语言中通常代表结合多个复合命令的意思。

MacroCommand 类的 commands 字段是 java.util.Stack 类型，用来聚集多个（实现）Command（类的对象实例）。当然也可以用 java.util.Vector，不过为了让后面的 undo 方法比较容易实现，所以这里才用 java.util.Stack。

MacroCommand 类会实现 Command 接口，因此定义 execute 方法。那 execute 方法要怎么处理才好？它要执行多个命令的话，只要把存储在 commands 字段的所有对象实例的 execute 方法都调用出来即可。这样就能执行手边所有的 Command。万一有些 Command 是要在这个 while 循环里执行的话，可能还会是另一个新的 MacroCommand 的实例对象。不过，只要再调用该对象实例的 execute，最后所有 Command 仍然都会执行到。

append 方法是在 MacroCommand 类中新增 Command 的方法（“新增 Command”是指“新增实现（implements）Command 的类的对象实例”）。这个新增的 Command 也可能是其他 MacroCommand 的对象实例。虽然这里有 if 语句，但它只是用来检查避免误将自己（this）也 add 进去，否则 execute 方法会永远结束不了。这里 java.util.Stack 类的 push 方法是在 java.util.Stack 类的对象实例的最后面新增元素（跟 java.util.Vector 的 append 很像）。

undo 方法是删除 commands 最后面的命令。这里 java.util.Stack 类的 pop 方法是取出前面以 push 方法新增的最后一个元素。被取出的元素会从 Stack 类的对象实例中删除掉。

clear 方法删除所有命令。

**程序 22-2 MacroCommand 类 (MacroCommand.java)**

```
package command;

import java.util.Stack;
import java.util.Iterator;
```

## Command（命令）——将命令写成类

```

public class MacroCommand implements Command {
    // 命令的集合
    private Stack commands = new Stack();
    // 执行
    public void execute() {
        Iterator it = commands.iterator();
        while (it.hasNext()) {
            ((Command)it.next()).execute();
        }
    }
    // 新增
    public void append(Command cmd) {
        if (cmd != this) {
            commands.push(cmd);
        }
    }
    // 删除最后一个命令
    public void undo() {
        if (!commands.empty()) {
            commands.pop();
        }
    }
    // 全部删除
    public void clear() {
        commands.clear();
    }
}

```

---

## DrawCommand 类

DrawCommand 类（程序 22-3）是实现 Command 接口的类，表示“点的绘制命令”。有 drawable 和 position 这两个字段。drawable 字段是存储绘制的对象（Drawable 接口要到后面才会出现）。position 字段是表示绘制的位置。Point 类是 java.awt 包所规定的类，表示 X、Y 坐标二维空间上（即平面）的位置。

在构造函数中，将实现 Drawable 接口的类的对象实例和 Point 类的对象传递给参数，然后赋值到字段。这个地方就是产生“在这里画点”的命令部分。

execute 方法调用 drawable 字段的 draw 方法，这是执行命令的部分。

Java 编程基础

**程序 22-3** DrawCommand 类 (DrawCommand.java)

```

package drawer;

import command.Command;
import java.awt.Point;

public class DrawCommand implements Command {
    // 绘制对象
    protected Drawable drawable;
    // 绘制位置
    private Point position;
    // 构造函数
    public DrawCommand(Drawable drawable, Point position) {
        this.drawable = drawable;
        this.position = position;
    }
    // 执行
    public void execute() {
        drawable.draw(position.x, position.y);
    }
}

```

## Drawable 接口

Drawable 接口（程序 22-4）表示“绘制对象”。draw 是绘制的方法。为了简化程序，本章的程序示例故意设计成无法设定颜色和点的大小，设定颜色的部分，可参见练习题 22-1。

**程序 22-4** Drawable 接口 (Drawable.java)

```

package drawer;

public interface Drawable {
    public abstract void draw(int x, int y);
}

```

## DrawCanvas 类

DrawCanvas 类（程序 22-5）是实现 Drawable 接口的类，也是 java.awt.Canvas 类

## Command(命令)——将命令写成类

的子类。

自己应该绘制的命令的集合会存储在 history 字段，该字段的类型当然是 command.MacroCommand。

在构造函数中使用宽(width)、高(height)和绘制内容(history)对 DrawCanvas 的对象实例进行初始化。在这个过程所调用的 setSize、setBackground 方法都是 java.awt.Canvas 类的方法，分别用来指定大小和背景颜色。

paint 方法是需要再度绘制 DrawCanvas 时，从 Java 处理(java.awt 的框架)调用出来的方法。它所执行的处理只调用 history.execute 而已，虽然行为简单，但这样就能再执行 history 里所记录的命令的集合。

draw 方法是为了实现 Drawable 接口而定义的方法，利用 g.setColor 设定颜色，利用 g.fillOval 输出钱币单位(元)。

### DrawCanvas 类 (DrawCanvas.java)

```
package drawer;

import command.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DrawCanvas extends Canvas implements Drawable {
    // 设置颜色
    private Color color = Color.red;
    // 绘制点的半径
    private int radius = 6;
    // 记录
    private MacroCommand history;
    // 构造函数
    public DrawCanvas(int width, int height, MacroCommand history)
    {
        setSize(width, height);
        setBackground(Color.white);
        this.history = history;
    }
    // 再度绘制整个记录
    public void paint(Graphics g) {
```

## 面向对象的运用

```

        history.execute();
    }

    // 绘制
    public void draw(int x, int y) {
        Graphics g = getGraphics();
        g.setColor(color);
        g.fillOval(x - radius, y - radius, radius * 2, radius * 2);
    }
}

```

---

**Main 类**

Main 类（程序 22-6）用来测试程序示例。

history 字段存储绘制的记录，其内容同后面要传递给 DrawCanvas 的对象实例的内容。也就是说，Main 的对象实例跟 DrawCanvas 的对象实例两者共享同一个绘制记录。

canvas 字段是绘制区域，程序示例设定以  $400 \times 400$  为默认值。

clearButton 字段是删除绘制红点的删除键。JButton 类为 javax.swing 包的类，表示按键。

在构造函数设定接收事件（如按下鼠标左键等）的监听者（listener），并编排绘制组件（零件）的版面。

因为要配合后面衍伸出来的练习题，所以按键的版面设计稍微有点复杂。首先，建立把零件排成一列的方块 buttonBox。传递 BoxLayout.X\_AXIS 给构造函数，就能让零件排得整整齐齐。接着把 clearButton 放在 buttonBox 上面。下一步，再建立一个可让零件由上而下排成一列的方块

mainBox，把 buttonBox 和 canvas 放到 mainBox 上面。

最后把 mainBox 放在 Jframe 上面。你也可以直接把组件放在 java.awt.Frame 上，这里的 javax.swing.JFrame 则是把组件放在另外用 getContentPane 方法产生的容器（container）里（图 22-3）。

actionPerformed 方法是实现 ActionListener 接口。按下 clearButton 时，会先删除绘制记录才进行再绘制。

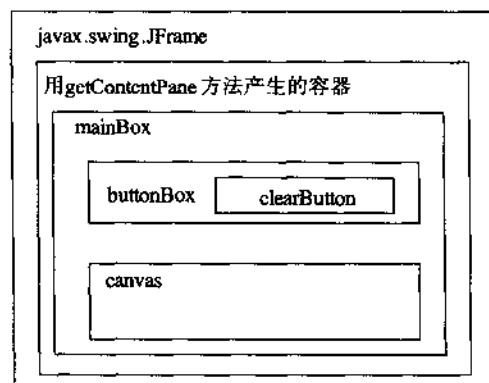


图 22-3 组件的版面编排

## Command(命令)——将命令写成类

mouseMoved 方法和 mouseDragged 方法实现 MouseMotionListener 接口。这里，在移动鼠标时 (mouseDragged) 产生“在这里画点”的命令。产生的命令再以

```
history.append(cmd);
新增到执行记录内之后，继续以
cmd.execute();
立即执行。
```

开头 window 的一组方法是实现 WindowListener 接口，这里只实现结束处理 (exit)。在 main 方法则是产生并执行 Main 类的对象实例。图 22-4 即为流程图。

**程序 22-6 Main 类 (Main.java)**

```
import command.*;
import drawer.*;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Main extends JFrame implements ActionListener,
MouseMotionListener, WindowListener {
    // 绘制记录
    private MacroCommand history = new MacroCommand();
    // 绘制区域
    private DrawCanvas canvas = new DrawCanvas(400, 400, history);
    // 删除键
    private JButton clearButton = new JButton("clear");

    // 构造函数
    public Main(String title) {
        super(title);

        this.addWindowListener(this);
        canvas.addMouseMotionListener(this);
        clearButton.addActionListener(this);

        Box buttonBox = new Box(BoxLayout.X_AXIS);
        buttonBox.add(clearButton);
```

## 3.3.3 画布类的实现

```
Box mainBox = new Box(BoxLayout.Y_AXIS);
mainBox.add(buttonBox);
mainBox.add(canvas);
getContentPane().add(mainBox);

pack();
setVisible(true);
}

// ActionListener用
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == clearButton) {
        history.clear();
        canvas.repaint();
    }
}

// MouseMotionListener用
public void mouseMoved(MouseEvent e) {
}
public void mouseDragged(MouseEvent e) {
    Command cmd = new DrawCommand(canvas, e.getPoint());
    history.append(cmd);
    cmd.execute();
}

// WindowListener用
public void windowClosing(WindowEvent e) {
    System.exit(0);
}
public void windowActivated(WindowEvent e) {}
public void windowClosed(WindowEvent e) {}
public void windowDeactivated(WindowEvent e) {}
public void windowDeiconified(WindowEvent e) {}
public void windowIconified(WindowEvent e) {}
public void windowOpened(WindowEvent e) {}}
```

## Command（命令）——将命令写成类

```

public static void main(String[] args) {
    new Main("Command Pattern Sample");
}
}

```

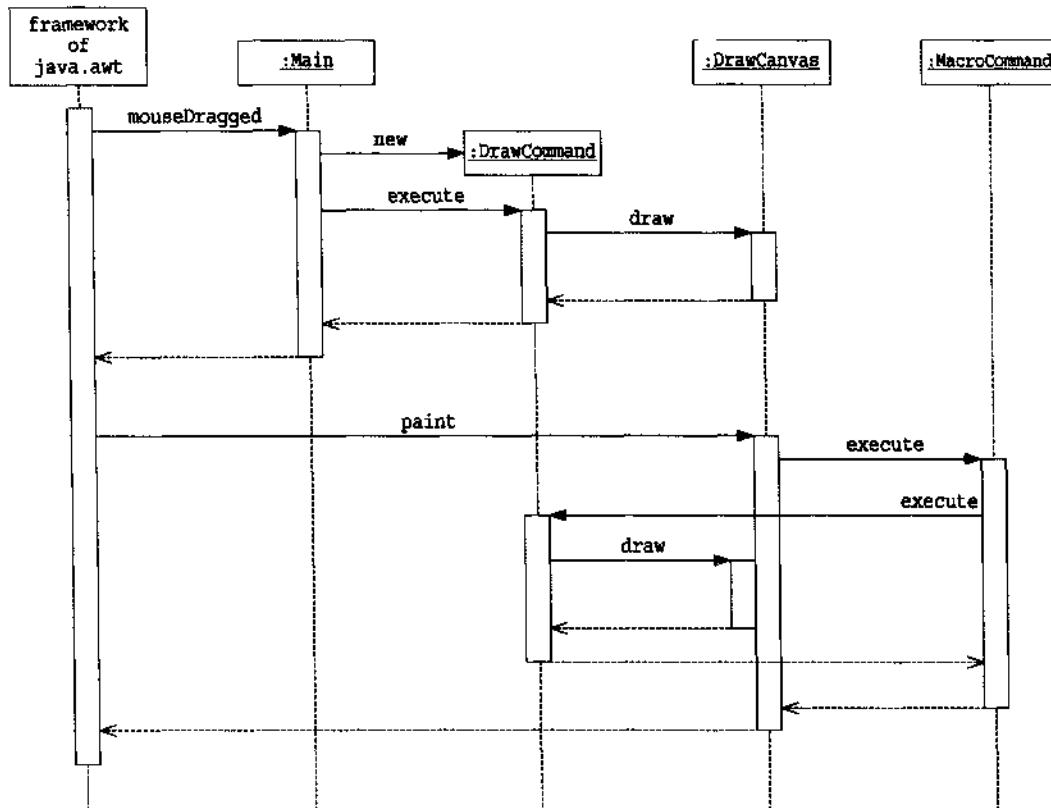


图 22-4 程序示例的流程图

## Command Pattern 的所有参与者

Command Pattern 的所有参与者可整理如下。

### ◆ Command（命令）参与者

定义命令的接口（API）。在本章的程序示例中，扮演这个参与者的是 Command 接口。

### ◆ ConcreteCommand (具体的命令) 参与者

实际实现 Command 参与者的接口 (API)。在本章的程序示例中，扮演这个角色的是 MacroCommand 类和 DrawCommand 类。

### ◆ Receiver (接收者) 参与者

Command 参与者执行命令时的行为对象。在本章的程序示例中，接收 DrawCommand 的命令的是 DrawCanvas 类。

### ◆ Client (客户) 参与者

在产生 ConcreteCommand 参与者时，分配 Receiver 的参与者。在本章的程序示例中，扮演这个角色的是 Main 类。Main 类是配合鼠标移动而产生 DrawCommand 的对象实例，不过同时也传递 DrawCanvas 的对象实例给构造函数，作为 Receiver 参与者。

### ◆ Invoker (启动者) 参与者

开始执行命令的参与者，调用 Command 所定义的接口 (API)。在本章的程序示例中，扮演这个角色的是 Main 类和 DrawCanvas 类。这两个类是调用 Command 接口的 execute 方法。Main 类同时扮演 Client 参与者和 Invoker 参与者。

Command Pattern 的类图、流程图分别如图 22-5 和图 22-6 所示。

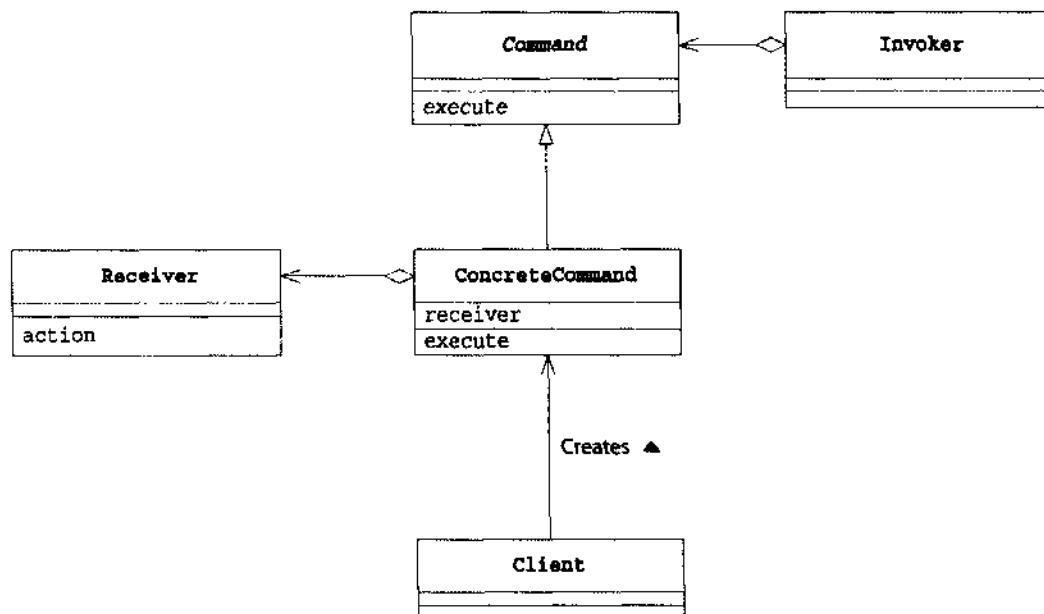


图 22-5 Command Pattern 的类图

## Command (命令) —— 将命令写成类

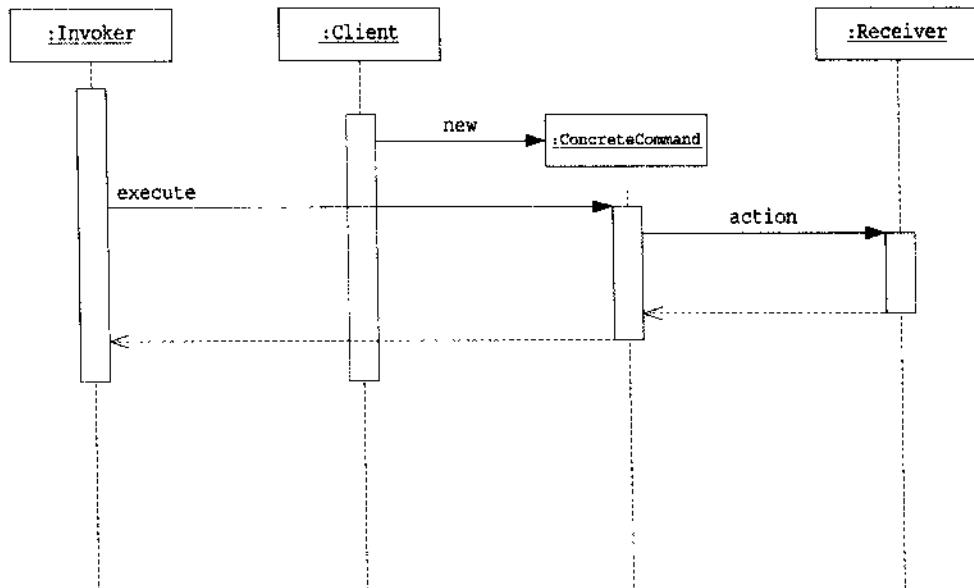


图 22-6 Command Pattern 的流程图

## 扩展自我视野的提示

### 命令应该有的信息是什么

“命令”应该有哪些信息会因目的而异。前面的 `DrawCommand` 类只有画点位置的信息，而没有点的大小或颜色、形状等的信息。

假设 `DrawCommand` 有一个“该事件发生点的时间戳 (time stamp)”的信息。如此一来，再绘制时就不再是简单的画点，或许还可以重现出现用户鼠标移动的快慢情形。

不过话说回来，`DrawCommand` 类还有一个表示绘制对象的字段 (`drawable`)。本章程式示例的 `DrawCanvas` 的对象实例只有 1 个，所有绘制当然都是针对它，所以这个 `drawable` 字段反而没有太大意义。但是如果一个程序有多个绘制对象（即 `Receiver` 参与者）时，这个字段就很有用。因为 `ConcreteCommand` 参与者本身“知道”`Receiver` 参与者，所以不管是谁在管理、或是拥有 `ConcreteCommand` 参与者，都可以做 `execute` 的操作。

### 存储记录

程序示例用 `MacroCommand` 的对象实例 (`history`) 来表示绘制的记录，因为这个

对象实例拥有目前为止的所有绘制相关信息。这意思是说，只要确实将这个对象实例存储成文件，就能存储绘制记录。

## 适配器

本章程式示例的 Main 类（程序 22-6）共实现 3 个接口，但实际使用到的接口的方法却只是其中的一小部分。

举例来说，MouseMotionListener 中下面的方法里，只使用到 mouseDragged 方法：

```
public void mouseMoved(MouseEvent e)
public void mouseDragged(MouseEvent e)
```

再以 WindowListener 为例，

```
public void windowClosing(WindowEvent e)
public void windowActivated(WindowEvent e)
public void windowClosed(WindowEvent e)
public void windowDeactivated(WindowEvent e)
public void windowDeiconified(WindowEvent e)
public void windowIconified(WindowEvent e)
public void windowOpened(WindowEvent e)
```

上面这 7 个方法中也只用到 windowClosing 方法。

为了简化整个程序，java.awt.event 包里有一个称为适配器的类群组。例如，对应 MouseMotionListener 接口的是 MouseMotionAdapter 类，对应 WindowListener 接口的是 WindowAdapter 类（表 22-2）。这种适配器就是 Adapter Pattern（第 2 章）的一种。

表 22-2 接口与适配器

| 接口                     | 适配器                   |
|------------------------|-----------------------|
| MouseMotionListener 接口 | MouseMotionListener 类 |
| WindowListener 接口      | WindowAdapter 类       |

继续再举 MouseMotionAdapter 类这个例子来说明。这个类会实现 MouseMotionListener 接口，所有该接口要求的方法都会实现。不过，这个实现都是空空如也（什么都不做的方法）。所以只要产生 MouseMotionAdapter 类的子类，实现必要的方法就能达到目的。

尤其是如果能配合 Java 中一个称为内部类（inner class）的结构来使用适配器，就能写出更好的程序。以下针对 MouseMotionListener 作为接口时（程序 22-7）和 MouseMotionAdapter 作为内部类时（程序 22-8）的情形做一比较。其它内容省略。

## Command(命令)——将命令写成类

## 程序 22-7 MouseMotionListener 的情形(须有一个空的 mouseMoved 方法)

```

public class Main extends JFrame
implements ActionListener, MouseMotionListener, WindowListener {
    ...
    public Main(String title) {
        ...
        canvas.addMouseMotionListener(this);
        ...
    }
    ...
    // MouseMotionListener 用
    public void mouseMoved(MouseEvent e) {
    }
    public void mouseDragged(MouseEvent e) {
        Command cmd = new DrawCommand(canvas, e.getPoint());
        history.append(cmd);
        cmd.execute();
    }
    ...
}

```

## 程序 22-8 MouseMotionAdapter 的情形(不需要空的 mouseMoved 方法)

```

public class Main extends JFrame
implements ActionListener, WindowListener {
    ...
    public Main(String title) {
        ...
        canvas.addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent e) {
                Command cmd = new DrawCommand(canvas, e.getPoint());
                history.append(cmd);
                cmd.execute();
            }
        });
        ...
    }
}

```

...

或许有些读者还不太熟悉内部类的语法，一时之间看不出究竟，不过仔细一看就会发现：

- new MouseMotionAdapter()这个地方刚好跟产生对象实例的表达式很像
- 后面的{ ... }跟类定义中的部分（定义方法）也很像

原来这样就能产生 MouseMotionAdapter 类的子类（但无名称），产生该对象实例。只要实现了应该覆盖的方法（mouseDragged），接下来就不必再伤脑筋该填什么。

附带说明一下，编译时内部类的文件名格式是“父类名\$号码.class”，如：

Main\$1.class

在后面的练习题 22-3，会让各位有机会练习利用 MouseMotionAdapter 和 WindowAdapter 改写程序示例。

## 相关 Pattern

### ◆ Composite Pattern (第 11 章)

在做宏命令（macro command）时，有时候会使用到 Composite Pattern。

### ◆ Memento Pattern (第 18 章)

在存储 Command 参与者的记录时，有时候会使用 Memento Pattern。

### ◆ Prototype Pattern (第 6 章)

欲复制已发生的事件（产生的命令）时，有时候会使用 Prototype Pattern。

## 重点回顾

本章所学习的是可使用对象表现“命令”，利用它来做记录或再执行的 Command Pattern。即使是平常我们不会注意到的“小兵”，只要用对象来表现的话，也可能化腐朽为神奇，来个“小兵立大功”喔。

## 练习题

解答请见附录 A

### 问题 1

请在本章的程序示例中新增一个“设定绘制颜色”的功能。有点像是换只新蜡笔的感觉，设定了新颜色之后，一移动鼠标就会画出新颜色的色点。

## Command（命令）——将命令写成类

另外建立一个 ColorCommand 类，这个类是表示设定绘制颜色的命令。

### 问题 2

请在本章的程序示例中新增一个“删除最后画的点”的复原功能。

### 问题 3

请修改本章程序示例中的 Main 类（程序 22-6），改用 MouseMotionAdapter 类跟 WindowAdapter 类（适配器）取代 MouseMotionListener 接口和 WindowListener 接口。

# 第 23 章

Interpreter（解释器）——以类来表达语法规则

## Interpreter Pattern

各位已从本书的各章节学习到多种 Design Pattern。Design Pattern 的目的之一就是提高类的可再用性。可再用性是指已经产生的类不需要多做修改（尽量不改）就能多次使用的意思。

本章要学习的是

### Interpreter Pattern

Interpreter Pattern 是用简单的“迷你语言”来表现程序要解决的问题，以迷你语言写成“迷你程序”而表现具体的问题。迷你程序本身无法独自启动，必须先用 Java 语言另外写一个负责“解释（interpreter）”的程序。解释程序能分析迷你语言，并解释、执行迷你程序。这个解释程序也称为解释器。当应解决的问题发生变化时，要修改迷你程序来对应处理，而不是修改用 Java 语言写成的程序。

下面以图示的方式说明当问题发生变化时，要修改哪个阶段的程序。如果程序设计使用的是 Java 语言，通常应如图 23-1 修改程序。虽然我们希望尽量避免修改程序，但多少都要修改 Java 语言的程序。

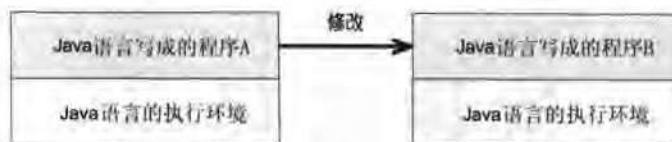


图 23-1 当问题发生变化时，通常是修改 Java 语言写成的程序

不过若使用 Interpreter Pattern，就不是修改 Java 语言的程序，而是直接修改以迷你语言写成的迷你程序（图 23-2）。

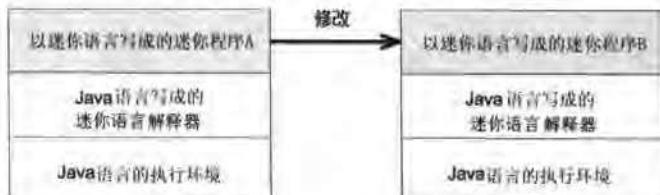


图 23-2 在 Interpreter Pattern 则是修改迷你语言写成的程序

## 迷你语言

### 迷你语言的命令

在开始说明 Interpreter Pattern 的程序示例之前，笔者要先说明本章所用到的“迷

## Interpreter（解释器）——以类来表达语法规则

你语言”。以用遥控器操纵玩具汽车的程序语言为例，玩具汽车的基本操作只有下面3种：

- 前进1米（go）
- 右转（right）
- 左转（left）

这3个动作就是对汽车发出的命令（command）。go是前进1米后停止的命令。right是立即右转的命令，left则是立即左转的命令。当然实际开车时不可能原地左转或右转，不过为了不让程序内容太过复杂，这里的左右转是像转盘那样改变方向，而不是改变位置。

如果只有这么机械性的动作也很无聊，因此多加了一个反复命令。

- 重复（repeat）

组合应用以上命令而操纵汽车的程序语言就是本章的迷你语言。

## 迷你程序示例

请看下面几个用迷你语言写成的迷你程序示例。第一个是开车前进（前进后停止）的迷你程序。

```
program go end
```

我们在程序前后用 program 和 end 这两个单词，让程序的开头和结尾比较醒目（后面还会再提到迷你语言的“语法”部分）。这个迷你程序的执行示例即如图 23-4 所示（内含此 GUI 的程序就是练习题 23-1）。

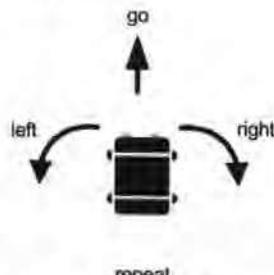


图 23-3 操纵汽车的迷你语言

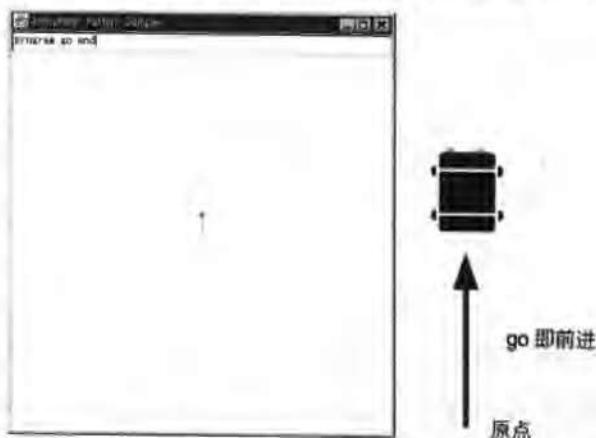


图 23-4 执行 program go end

接着是汽车前进、再右转的迷你程序。program 和 end 之间可以插入无限个命令。

```
program go right right go end
```

接着是利用右转绕一个正方形后再回到原点的迷你程序。执行示例则如图 23-5 所示。

```
program go right go right go right go right end ... (A)
```

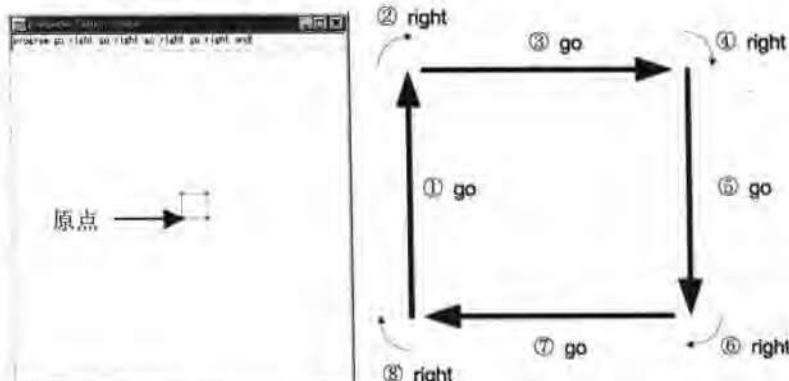


图 23-5 执行 program go right go right go right go right end

程序 (A) 的最后面 (end 之前) 会放一个 right 是为了让汽车方向回到开始状态。各位可以看出程序 (A) 共反复了 4 次 go right。利用 repeat ... end 的语法则可将此部分改写如 (B) (迷你语言的语法须规定可如此处理)。其执行示例即如图 23-6 所示。

```
program repeat 4 go right end end ... (B)
```

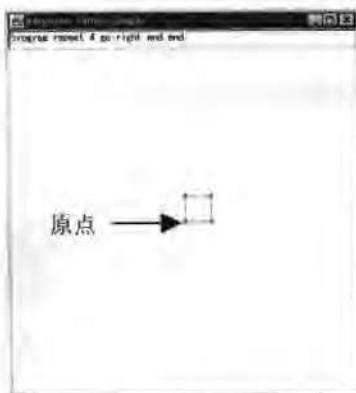


图 23-6 执行 program repeat 4 go right end end (同图 23-5)

程序 (B) 的最后面有 2 个 end, 第 1 个 (左) end 表示 repeat 的结尾, 最后一个 (右) end 表示 program 的结尾。也就是说:

```
program
repeat
```

程序开始

反复开始

## Interpreter（解释器）——以类来表达语法规则

|       |      |
|-------|------|
| 4     | 反复次数 |
| go    | 前进   |
| right | 右转   |
| end   | 反复结束 |
| end   | 程序结束 |

各位的脑海中是不是已经有汽车在转来转去了？出个问题考考你，下面的迷你程序如何操纵汽车？

```
program repeat 4 repeat 3 go right go left end right end end
```

这部汽车会画出如图 23-7 的多方角菱形。这里的 repeat 是双重 repeat 所以有点复杂，不过只要分解如下应该就不难理解。

|         |          |
|---------|----------|
| program | 程序开始     |
| repeat  | 反复开始（外部） |
| 4       | 反复次数     |
| repeat  | 反复开始（里面） |
| 3       | 反复次数     |
| go      | 前进       |
| right   | 右转       |
| go      | 前进       |
| left    | 左        |
| end     | 反复结束（里面） |
| right   | 右转       |
| end     | 反复结束（外部） |
| end     | 程序结束     |

里面的反复“go right go left”是“前进后右转、再前进后左转”的命令，共反复 3 次。这样会出现一个  $90^\circ$  转角，然后继续往右前进。整个程序的  $90^\circ$  转角的向右前进后，再做 1 次 right”的总共反复 4 次。这样画出一个多方角的菱形。

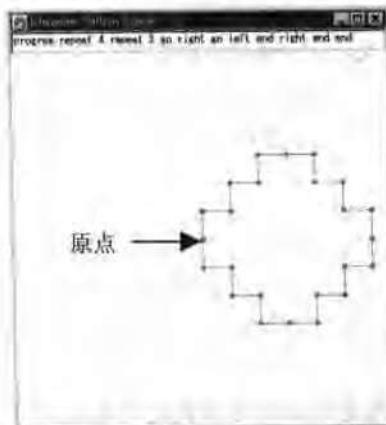


图 23-7 执行 program repeat 4 repeat 3 go right go left end right end end

## 迷你语言的语法

继续要谈到迷你语言的语法部分（图 23-8）。这种描述方式是“BNF”的一种变形。BNF 是 Backus-Naur Form、或 Backus Normal Form（贝克氏格式）的缩写，常用来描述程序语言的语法。

```

<program> ::= program <command list>
<command list> ::= <command>* end
<command> ::= <repeat command> | <primitive command>
<repeat command> ::= repeat <number> <command list>
<primitive command> ::= go | right | left

```

图 23-8 程序示例的解释器要解释的迷你语言的“语法”

依次说明如下。

<program> ::= program <command list>

这个部分是定义<program>（程序），“<program>是指 program 这个单词后面加命令行<command list>的部分”。::=的左方是被定义的对象，右方则是定义的内容。

<command list> ::= <command>\* end

这里是定义命令行<command list>，“<command list>是指命令<command>反复 0 个以上之后，再加上 end 单词”。\*表示前面的内容反复 0 次以上。

<command> ::= <repeat command> | <primitive command>

接着是定义命令<command>，“<command>是指反复命令<repeat command>或基本命令<primitive command>之一”。| 表示“或”的意思。

<repeat command> ::= repeat <number> <command list>

继续定义的是反复命令<repeat command>，“<repeat command>是指 repeat 这个单词后面先是反复次数<number>，然后是命令行<command list>”。命令行<command list>在前面已经定义过了。<command list>的定义中用到<command>，<command>的定义里用到<repeat command>，<repeat command>的定义里则用到<command list>。像这样的定义称为“递归式定义”。后面利用 Java 语言写出这个迷你语言的解释器，此时也反映了递归式定义的结构，故请牢记。

<primitive command> ::= go | right | left

定义基本命令<primitive command>，“<primitive command>是指 go、right 或 left”。

最后一个<number>，若把所有定义的详细内容都写出来的话，又会更复杂，故省略。各位只要视<number>为一个自然数（如 3、4、12345 等）即可。

注意：严格说来，这里使用到的是扩充后的 BNF。原始的 BNF 并非表示 0 次以上的反复\*，反复也是用递归式定义来表示。

## 终点表达式与非终点表达式

稍微说明一下相关的专业术语。

在前面的语法中，不会继续再展开的表达式称为“终点表达式（terminal expression）”，如<primitive command>。巴士和火车的最后一站也称为终点车站，两者意思很相似。这里是指语法规则的终点。

而像<program>、<command>等还会再展开称为“非终点表达式（nonterminal expression）”。

## 程序示例

迷你语言的介绍告一段落之后，继续说明程序示例。本章的程序示例是对前面的迷你语言进行语法解析。

前一节在说明迷你程序的内容时，我们是把迷你程序拆解成几个部分分别解释。把字符串所组成的迷你程序分解成多段，再去分析各个部分相当于哪种结构，这就是语法解析。

举例来说，

```
program repeat 4 go right end end
```

当系统遇到上面的迷你程序时，便在内存上产生如图 23-9 的结构（树状剖析）的处理即为语法解析。

程序示例会一直处理到产生这个树状剖析为止。实际上“执行”的部分如练习题 23-1。

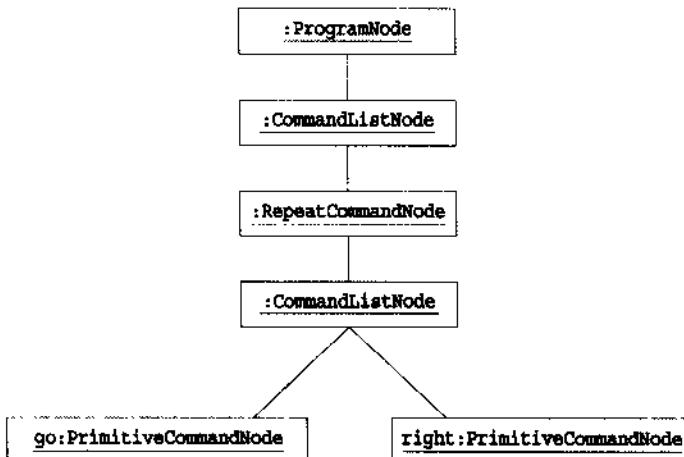


图 23-9 迷你程序 program repeat 4 go right end end 的树状剖析。

表 23-1 类一览表

| 名称                   | 说明                      |
|----------------------|-------------------------|
| Node                 | 树状剖析中“节点”的类             |
| ProgramNode          | 对应<program>的类           |
| CommandListNode      | 对应<command list>的类      |
| CommandNode          | 对应<command>的类           |
| RepeatCommandNode    | 对应<repeat command>的类    |
| PrimitiveCommandNode | 对应<primitive command>的类 |
| Context              | 表示语法解析之前后关系的类           |
| ParseException       | 语法解析中的例外类               |
| Main                 | 测试用的类                   |

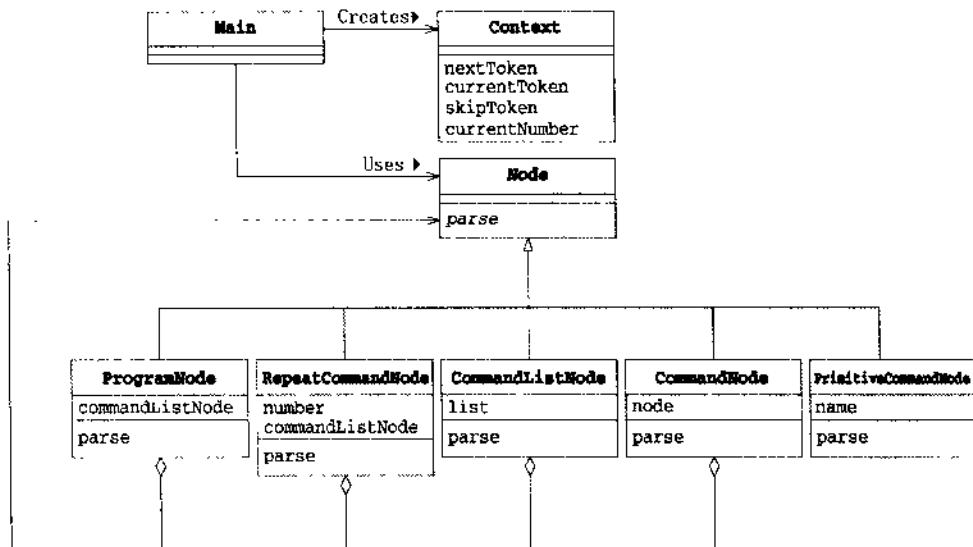


图 23-10 程序示例的类图

## Node 类

`Node` 类（程序 23-1）是组成树状剖析各部分（节点）的最上层类。在 `Node` 类中只声明抽象方法 `parse` 而已，这个 `parse` 是“进行语法解析处理”的方法。`Node` 类只声明 `parse` 方法，具体上该如何分析则交给 `Node` 类的子类（*subclass responsibility*）。`parse` 方法的参数所接收到的 `Context` 是表示进行语法解析的“状况”的类，后面会另有详细说明。`parse` 方法的声明里有一个 `throws` 语句。这个 `throws` 语句是表示如果语法解析进行中发生程序错误时，则抛出 `ParseException` 这个异常。

## Interpreter（解释器）——以类来表达语法规则

光看 Node 类的部分很难知道具体有哪些内容，所以让我们继续看下去。

程序 23-1 Node 类 (node.java)

```
public abstract class Node {
    public abstract void parse(Context context) throws
ParseException;
}
```

## ProgramNode 类

接下来我们根据前面图 23-8 的迷你语言的语法 (BNF)，一个个来看类定义。第一个就是表示程序<program>的 ProgramNode 类（程序 23-2）。这个类有 Node 类型的 CommandListNode 的字段，此字段是存储本身后面的<command list>所对应的结构（节点）。

在 ProgramNode 的 parse 方法进行了哪些处理？从 BNF 来看，<program>的最前面应该是 program 这个单词。后面的语句则略过 program 这个单词。

```
context.skipToken("program");
```

语法解析时的处理单位是记号 (token)。在迷你语言中，“记号”跟“英文单词”同义，但如果是一般的程序设计语言，+、= 等也都是记号的一种。再讲得清楚一点，字句解析 (lex) 是根据字符产生记号，语法解析 (parse) 则根据记号产生树状剖析。

这个 skipToken 方法会忽略 program 记号；而且如果没有 program 记号时，还会抛出 ParseException 异常。

BNF 的后面紧接着<command list>。就是在这里产生对应<command list>的 CommandListNode 的对象实例，调用该对象实例的 parse 方法。请注意一件事：<command list>的内容为何并没有写在 ProgramNode 类的方法里。ProgramNode 类里顶多只有下面这个 BNF 看得见的范围而已。

```
<program> ::= program <command list>
```

toString 方法则是叙述此节点的字符串表现。在 Java 环境下，当对象实例跟字符串连接起来时，就会自动调用 toString 方法，因此下面这个表达式，

```
"[program " + commandListNode + "]"
```

跟以下的表达式是等价的。

```
"[program " + commandListNode.toString() + "]"
```

请各位注意到这个 toString 方法的程序代码也写成符合 BNF 的内容。

程序 23-2 ProgramNode 类 (ProgramNode.java)

```
// <program> ::= program <command list>
public class ProgramNode extends Node {
```

## Java 语言中的应用

```

private Node commandListNode;
public void parse(Context context) throws ParseException {
    context.skipToken("program");
    commandListNode = new CommandListNode();
    commandListNode.parse(context);
}
public String toString() {
    return "[program " + commandListNode + "]";
}
}

```

---

**CommandListNode 类**

其次是 CommandListNode 类（程序 23-3）。BNF 如下：

`<command list> ::= <command>* end`

`<command>` 反复 0 次以上，最后面则是 `end`。CommandListNode 类有一个 `java.util.Vector` 类型的字段 `list`，用来存储反复 0 次以上的`<command>`。这个字段所存储的是对应`<command>`的 `CommandNode` 类的对象实例。

那么，`parse` 方法如何呢？首先，如果我们现在所看的记号（= `context.currentToken()`）的值为 `null`，则表示已无其他记号（即迷你程序已经读到最后）。此时，`parse` 方法会附加“已无 `end` (Missing 'end')”的信息，然后抛出 `ParseException` 异常。

而若所看的记号为 `end`，则表示已经到达`<command list>`的结尾。此时，先忽略 `end` 再运行 `break while` 语句。

如果现在所看的记号不是 `end`，则表示为`<command>`，故产生 `CommandNode` 的对象实例后再 `parse`。并且将该对象实例 `add` 到 `CommandListNode` 的 `list` 字段。

毋须笔者多做说明，各位聪明的读者也都看得出来这里也是只处理 BNF 里有语句到的范围而已。设计程序时要尽量忠于 BNF 的思想，顶多只是像把 BNF 改成 Java 的感觉而已。如此一来便能降低程序错误的发生率。如果一时把持不住，受到“改成○○应该可以提高处理速度吧？”的诱惑，难免会想改用能读取到结构更详细的处理，不过这样很可能会间接造成意料之外的 bug。Interpreter Pattern 本来就是采取迷你语言这种间接式的处理方法，如果想运用小技巧提高效率，恐怕只会得不偿失。

程序 23-3 CommandListNode 类 (CommandListNode.java)

```

import java.util.Vector;

// <command list> ::= <command>* end

```

## Interpreter（解释器）——以类来表达语法规则

```

public class CommandListNode extends Node {
    private Vector list = new Vector();
    public void parse(Context context) throws ParseException {
        while (true) {
            if (context.currentToken() == null) {
                throw new ParseException("Missing 'end'");
            } else if (context.currentToken().equals("end")) {
                context.skipToken("end");
                break;
            } else {
                Node commandNode = new CommandNode();
                commandNode.parse(context);
                list.add(commandNode);
            }
        }
    }
    public String toString() {
        return "" + list;
    }
}

```

---

**CommandNode 类**

如果各位前面读过 ProgramNode 类和 CommandListNode 类的说明马上融会贯通，那么 CommandNode 类（程序 23-4）也不会有什么问题才对。

BNF 如下：

<command> ::= <repeat command> | <primitive command>

Node 类型的字段 node 是用来存储对应<repeat command>的 RepeatCommandNode 类的对象实例、或对应<primitive command>的 PrimitiveCommandNode 类的对象实例。

**程序 23-4 CommandNode 类（CommandNode.java）**

```

// <command> ::= <repeat command> | <primitive command>
public class CommandNode extends Node {
    private Node node;
    public void parse(Context context) throws ParseException {
        if (context.currentToken().equals("repeat")) {
            node = new RepeatCommandNode();
            node.parse(context);
        } else {

```

## 第 23 章 带有非终结符的表达式

```

        node = new PrimitiveCommandNode();
        node.parse(context);
    }
}

public String toString() {
    return node.toString();
}
}

```

---

## RepeatCommandNode 类

RepeatCommandNode 类（程序 23-5）语法对应<repeat command>。

<repeat command> ::= repeat <number> <command list>

<number>的部分被存储在 int 类型的 number 字段，而<command list>的部分存储在 Node 类型的 commandListNode 字段。

嗯，各位好像注意到有 parse 方法的递归性，好眼力。我们来看看它的方法调用吧。

- 在 RepeatCommandNode 的 parse 方法中，  
产生 CommandListNode 的对象实例，再调用 parse 方法；
- 在 CommandListNode 的 parse 方法中，  
产生 CommandNode 的对象实例，再调用 parse 方法；
- 在 CommandNode 的 parse 方法中，  
产生 RepeatCommandNode 的对象实例，再调用 parse 方法；
- 在 RepeatCommandNode 的 parse 方法中，...

这个“不断调用 parse 方法的旅途”到底要走到什么地方？旅途的终点当然就是终点表达式。因为有 CommandNode 的 parse 方法中的 if 语句，总有一天会结束 RepeatCommandNode，进入产生 PrimitiveCommandNode 的部分。而在 PrimitiveCommandNode 的 parse 方法中就不再调用其他的 parse 方法。这点在下一节会另外介绍。

要是有人还不太习惯递归式定义的处理作风，可能会觉得怎么多出一个无穷循环，不要想太多，那只是个错觉。无论是 BNF 也好、Java 也罢，它们总有一天会停在终点表达式的位置。如果永远都没有走到终点表达式的位置，那就表示定义写错了。

**程序 23-5 RepeatCommandNode 类 (RepeatCommandNode.java)**

```

// <repeat command> ::= repeat <number> <command list>
public class RepeatCommandNode extends Node {
    private int number;

```

## Interpreter（解释器）——以类来表达语法规则

```

private Node commandListNode;
public void parse(Context context) throws ParseException {
    context.skipToken("repeat");
    number = context.currentNumber();
    context.nextToken();
    commandListNode = new CommandListNode();
    commandListNode.parse(context);
}
public String toString() {
    return "[repeat " + number + " " + commandListNode + "]";
}
}

```

---

**PrimitiveCommandNode 类**

对应 PrimitiveCommandNode 类（程序 23-6）的 BNF 为：

<primitive command> ::= go | right | left

没错，确实没有从这里的 parse 方法去调用其他的 parse 方法。

**程序 23-6 PrimitiveCommandNode 类（PrimitiveCommandNode.java）**

```

// <primitive command> ::= go | right | left
public class PrimitiveCommandNode extends Node {
    private String name;
    public void parse(Context context) throws ParseException {
        name = context.currentToken();
        context.skipToken(name);
        if (!name.equals("go") && !name.equals("right") && !name.
equals("left")) {
            throw new ParseException(name + " is undefined");
        }
    }
    public String toString() {
        return name;
    }
}

```

---

**Context 类**

以上是 Node 类跟相关子类的解说部分。最后一部分是 Context 类。Context 类（程

序 23-7) 是提供语法解析时的必要方法。

表 23-2 Context 类所提供的方法

| 名称            | 说明                         |
|---------------|----------------------------|
| nextToken     | 取得下一个记号（进到下一个记号）           |
| currentToken  | 取得现在的记号（不进到下一个记号）          |
| skipToken     | 检查现在的记号后，再取得下一个记号（进到下一个记号） |
| currentNumber | 取得现在的记号的值（不进到下一个记号）        |

这里是使用 `java.util.StringTokenizer`，这个类相当方便好用。它会帮忙把接收到的字符串分割成记号。分割的间隔字符是空格 (' ')、制表位 ('\t')、换行 ('\n')、回车 ('\r') 或翻页 ('\f')（也可使用其他区隔文字，请参见 Java 的 API 参考文档）。

表 23-3 Context 类所利用的 `java.util.StringTokenizer` 的方法

| 名称           | 说明               |
|--------------|------------------|
| nextToken    | 取得下一个记号（进到下一个记号） |
| hasMoreToken | 检查是否有下一个记号       |

### 程序 23-7 Context 类 (Context.java)

```
import java.util.*;

public class Context {
    private StringTokenizer tokenizer;
    private String currentToken;
    public Context(String text) {
        tokenizer = new StringTokenizer(text);
        nextToken();
    }
    public String nextToken() {
        if (tokenizer.hasMoreTokens()) {
            currentToken = tokenizer.nextToken();
        } else {
            currentToken = null;
        }
        return currentToken;
    }
    public String currentToken() {
```

## Interpreter（解释器）——以类来表达语法规则

```

        return currentToken;
    }

    public void skipToken(String token) throws ParseException {
        if (!token.equals(currentToken)) {
            throw new ParseException("Warning: " + token + " "
                + "is expected, but " + currentToken + " is found.");
        }
        nextToken();
    }

    public int currentNumber() throws ParseException {
        int number = 0;
        try {
            number = Integer.parseInt(currentToken);
        } catch (NumberFormatException e) {
            throw new ParseException("Warning: " + e);
        }
        return number;
    }
}

```

---

## ParseException 类

`ParseException` 类（程序 23-8）是语法解析中的异常。没有什么特别的地方。

**程序 23-8** `ParseException` 类 (`ParseException.java`)

```

public class ParseException extends Exception {
    public ParseException(String msg) {
        super(msg);
    }
}

```

---

## Main 类

`Main` 类（程序 23-9）是用来启动我们前面所提到的迷你语言的解释器。`Main` 会读入“`program.txt`”文件，把每 1 行都当做迷你程序进行语法解析，再将结果输出成字符串。

输出结果中的开头为 `text =` 的部分是接收到的迷你程序，开头为 `node =` 的部分是

## Java语言特性的应用

语法解析后的输出内容。从图 23-11 的运行结果，就可以知道在原来的简单字符串 program ... end 中间有插入了 [ ] 包起来的内容。也就是说解释器确实解读了该迷你程序。

**注意：**在 CommandListNode 的对象实例转成字符串的部分（例如 [go, right] 的部分）加上中括号[ ]和逗号的是 java.util.Vector 的 toString 方法。

程序 23-9 Main 类 (Main.java)

```
import java.util.*;
import java.io.*;

public class Main {
    public static void main(String[] args) {
        try {
            BufferedReader reader = new BufferedReader(new FileReader
("program.txt"));
            String text;
            while ((text = reader.readLine()) != null) {
                System.out.println("text = " + text + "\n");
                Node node = new ProgramNode();
                node.parse(new Context(text));
                System.out.println("node = " + node);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

程序 23-10 迷你程序的示例 (program.txt)

```
program end
program go end
program go right go right go right go right end
program repeat 4 go right end end
program repeat 4 repeat 3 go right go left end right end end
```

## Interpreter (解释器) ——以类来表达语法规则

```

text = "program end"           ← 迷你城市的內容
node = [program []]           ← 语法解析的结果
text = "program go end"
node = [program [go]]
text = "program go right go right go right go right end"
node = [program [go, right, go, right, go, right, go, right]]
text = "program repeat 4 go right end end"
node = [program [[repeat 4 [go, right]]]]
text = "program repeat 4 repeat 3 go right go left end right end
end"
node = [program [[repeat 4 [[repeat 3 [go, right, go, left]], right]]]]

```

图 23-11 执行结果

## Interpreter Pattern 的所有参与者

Interpreter Pattern 的所有参与者可整理如下。

### ◆ *AbstractExpression* (抽象表达式) 参与者

规定树状剖析的节点的共享接口 (API)。在本章的程序示例中，扮演这个角色的是 Node 类。图 23-12 是以 interpret 这个名称来表现接口 (API)，程序示例是使用 parse 方法。

### ◆ *TerminalExpression* (终结符表达式) 参与者

对应 BNF 的终结符表达式。在本章的程序示例中，扮演这个角色的是 PrimitiveCommandNode 类。

### ◆ *NonterminalExpression* (非终结符表达式) 参与者

对应 BNF 的非终结符表达式。在本章的程序示例中，扮演这个角色的是 ProgramNode, CommandNode, RepeatCommandNode, CommandListNode 的各个类。

### ◆ *Context* (上下文) 参与者

提供解释器进行语法解析时所需信息的参与者。在本章的程序示例中，扮演这个角色的是 Context 类。

### ◆ *Client* (客户) 参与者

调用 TerminalExpression 参与者和 NonterminalExpression 参与者，以建立树状剖

析的参与者。在本章的程序示例中，扮演这个角色的是 Main 类。

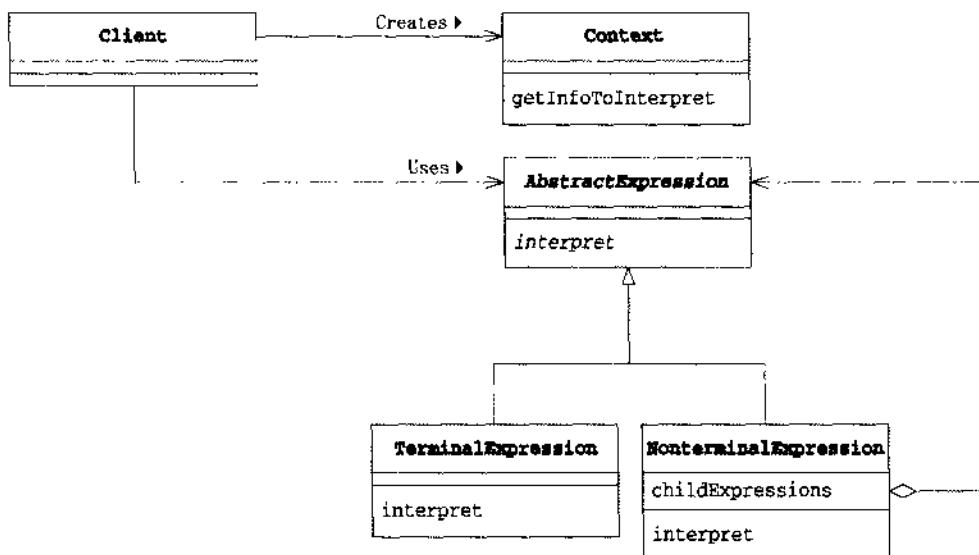


图 23-12 Interpreter Pattern 的类图

## 扩展自我视野的提示

### 还有其他不同的迷你语言吗？

本章所举的示例是无线遥控操纵汽车的迷你语言，这只是 Interpreter Pattern 的一个示例。还有没有其他的迷你语言？想知道吗？下面是几个常见的示例。

#### ◆ 正则表达式

GoF 一书举出正则表达式 (regular expression) 作为迷你语言的示例。语法解释如下，并发展出树状剖析：

`raining & (dogs | cats) *`

这是表示在 raining 后面反复 0 次以上的 dogs 或 cats。

#### ◆ 搜索用的表达式

在《应用 UML 的 Java 设计模式—可复用的程序设计集》中，有一个表现单词组合的 Little Language Pattern。例如，解释如下的内容，并发展出树状剖析：

`garlic and not onions`

这是表示含有 garlic、但没有 onions。

### ◆ 批处理程序语言

有几个基本的命令，依次或反复执行这些命令的程序语言（批处理程序语言）应该也可以用 Interpreter Pattern 来处理。本章的无线遥控也许也可以说是一种批处理程序语言。

## 略过/读入

在写解释器的部分时最常出现的 bug 就是多读、或少读 1 个记号的问题。各位在设计对应非终点表达式的方法时，要随时记得“到这个方法时，记号要读取到哪里；离开这个方法时，记号又该读取到哪里”。

## 相关 Pattern

### ◆ Composite Pattern（第 11 章）

NonterminalExpression 参与者通常都会有递归式结构，所以大多会使用 Composite Pattern。

### ◆ Flyweight Pattern（第 20 章）

有时 TerminalExpression 参与者也会利用 Flyweight Pattern 形成共享。

### ◆ Visitor Pattern（第 13 章）

完成树状剖析之后，穿梭在树状剖析的各节点间进行处理时，可能会利用到 Visitor Pattern。

## 重点回顾和最后的叮咛

本章所学习到的是利用迷你语言解决问题的 Interpreter Pattern。也请各位研究一下利用 BNF 递归式定义程序语言的方法、以及建立树状剖析的方法。

看到这里，各位这趟 GoF 的 23 个 Design Patterns 巡回之旅好像也已经踏入终点站了。不知各位有何感触？Pattern 的世界当中有简单的 Pattern、容易理解的 Pattern，相对也有复杂的 Pattern 和不易理解的 Pattern。但笔者相信各位已经对各个 Pattern 有进一步的体会，而且也培养出自己能从“Design Patterns（设计模式）”的角度看程序的能力，例如抽象类和接口的功能、继承和委让的用法、让类和方法看得见或看不见、可交换性、在不更改原始码的原则下做零件复用的方法等等。

本书到此也要画下句号，同时笔者心中默默期待着各位读者未来所设计的程序能更上一层楼。感谢各位读者耐心看完本书，希望下次还有机会推出自己的新作跟各位

见面。

Enjoy Patterns!

## 练习题

解答请见附录 A

### 问题 1

本章的程序示例仅列出语法解析的部分。请修改程序，让它也能“执行”语法解析接收到的程序内容。`go`、`right` 和 `left` 这些基本命令（`<primitive command>`）要如何“执行”，请各位多多发挥想象的空间。

**提示：**这是本书最后一个练习题，为了总结所有 Patterns，后附解答特地利用多种不同的方式。如：

- 利用 GUI 绘制基本命令的结果。
- 利用 Facade Pattern（第 15 章）让解释器更容易使用。
- 产生基本命令的类（Factory Method Pattern（第 4 章））。
- 解释器部分放在另一个包。

在图 23-4~图 23-7 解说迷你语言时的部分即为执行示例。

# 附录 A

## 练习问题的解答

# 第1章

## 问题 1-1 的解答

如下。不需要修改 Main 类的 while 循环。

程序 A1-1 BookShelf 类 (BookShelf.java)

```
import java.util.Vector;

public class BookShelf implements Aggregate {
    private Vector books;
    public BookShelf(int initialsizer) {
        this.books = new Vector(initialsize);
    }
    public Book getBookAt(int index) {
        return (Book)books.get(index);
    }
    public void appendBook(Book book) {
        books.add(book);
    }
    public int getLength() {
        return books.size();
    }
    public Iterator iterator() {
        return new BookShelfIterator(this);
    }
}
```

程序 A1-2 Main 类 (Main.java)

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        BookShelf bookShelf = new BookShelf(4);
        bookShelf.appendBook(new Book("Around the World in 80
Days"));
        bookShelf.appendBook(new Book("Bible"));
```

## 练习问题的解答

```

bookShelf.appendBook(new Book("Cinderella"));
bookShelf.appendBook(new Book("Daddy-Long-Legs"));
bookShelf.appendBook(new Book("East of Eden"));
bookShelf.appendBook(new Book("Frankenstein"));
bookShelf.appendBook(new Book("Gulliver's Travels"));
bookShelf.appendBook(new Book("Hamlet"));
Iterator it = bookShelf.iterator();
while (it.hasNext()) {
    Book book = (Book)it.next();
    System.out.println(" " + book.getName());
}
}

```

---

Around the World in 80 Days  
 Bible  
 Cinderella  
 Daddy-Long-Legs  
 East of Eden  
 Frankenstein  
 Gulliver's Travels  
 Hamlet

图 A1-1 执行结果

## 第 2 章

### 问题 2-1 的解答

因为想强调“只使用 Print 接口的方法”。在程序示例中，PrintBanner 类跟 Print 接口所提供的方法虽然相同，但在某些情形下，PrintBanner 类的方法可能会比较多。程序员的用意很明显，先将其指定到 Print 类型变量后再使用，其实是要利用 Print 接口的方法，而不是 PrintBanner 类的方法。

**补充：**即使指定到 Print 类型变量，只要该对象实例实际上还是 PrintBanner 类的对象实例，如下强迫转型（cast）也可以调用 PrintBanner 原来的方法（如果有的话）。

```
((PrintBanner)p).methodWhichExistsOnlyInPrintBanner();
```

如果指定到 p 的不是 PrintBanner 类的对象实例，则因调用的是不存在的方法，所以会产生运行时的错误 (java.lang.ClassCastException)。

## 问题 2-2 的解答

解答如下。在此是适用不同类的 Adapter Pattern。

**程序 A2-1 FileProperties 类 (FileProperties.java)**

```
import java.io.*;
import java.util.*;

public class FileProperties extends Properties implements FileIO {
    public void readFromFile(String filename) throws IOException {
        load(new FileInputStream(filename));
    }
    public void writeToFile(String filename) throws IOException {
        store(new FileOutputStream(filename), "written by
FileProperties");
    }
    public void setValue(String key, String value) {
       .setProperty(key, value);
    }
    public String getValue(String key) {
        return getProperty(key, "");
    }
}
```

## 第 3 章

### 问题 3-1 的解答

要求在 `java.io.InputStream` 的子类进行实现的方法是 `java.io.InputStream` 的 `read()` 方法(无参数)。`read()`方法被 `java.io.InputStream` 的模板方法 `read(byte[] b, int off, int len)` 反复调用出来。

换句话说，“读取 1 byte”这个处理的具体内容交给子类，在 `java.io.InputStream` 这端则产生“将指定 byte 读入到数组的指定位置”的处理的模板。

## 问题 3-2 的解答

表示 display 方法不能在子类重载 (override)。

如果要用文字叙述补充说明的话，就是这个类的设计者对子类设计者要求：“如果你要建立子类，扩充功能时不要重载 display 方法”的意思。

GoF一书里的写法是：模板方法 (Template Method) 不应重载。如欲禁止重载方法，就要像程序示例这样进行 final 声明。

## 问题 3-3 的解答

在 AbstractDisplay 类里 protected 声明 open、print、close 方法。如此一来，就可以从有继承关系的子类调用这些方法，不过从其他包的类则无法调用（但可从同一包的类调用）。

## 问题 3-4 的解答

因为 Template Method Pattern 的 AbstractClass 参与者必须实现处理的骨架。在抽象类可以具体实现一部分的方法（如 AbstractDisplay 类的 display 方法）。不过，接口下的所有方法必须是抽象方法。因此 Template Method Pattern 无法以接口取代抽象类。

# 第 4 章

## 问题 4-1 的解答

表示无法从 idcard 包以外的地方使用 new 产生 IDCard 的对象实例。因此，如欲产生 IDCard 的对象实例，一定要利用 IDCARDFactory。

例如，不能从 Main 类如下产生 IDCard 的对象实例

```
IDCard idcard = new IDCard("结城浩");
```

这样的程序代码会在编译时出现错误。

补充：在 Java 环境下，必须在同一包内的类才能使用没有任何存取控制（如 public、protected、private 等）的构造函数和方法。

## 问题 4-2 的解答

如程序 A4-1，程序 A4-2 所示。不需要修改 framework.Product 类（程序 4-1）、framework.Factory 类（程序 4-2）或 Main 类（程序 4-5）。请注意，即使要修改 IDCard

## Java语言中的应用

类和 IDCardFactory 类，也不必动到框架端的原始程序代码。

这里的编号从 100 开始，但没有什么特别用意。

把 IDCardFactory 类的 createProduct 方法设成 synchronized 是为了预防它在多线程结构下运行时，会把同一编号给两个（或以上）不同的对象实例。

## 程序 A4-1 有流水编号的 IDCard 类 (IDCard.java)

```
package idcard;
import framework.*;

public class IDCard extends Product {
    private String owner;
    private int serial;
    IDCard(String owner, int serial) {
        System.out.println("建立" + owner + "(" + serial + ")" + "的卡。");
        this.owner = owner;
        this.serial = serial;
    }
    public void use() {
        System.out.println("使用" + owner + "(" + serial + ")" + "的卡。");
    }
    public String getOwner() {
        return owner;
    }
    public int getSerial() {
        return serial;
    }
}
```

## 程序 A4-2 有流水编号的 IDCardFactory 类 (IDCardFactory.java)

```
package idcard;
import framework.*;
import java.util.*;

public class IDCardFactory extends Factory {
    private Hashtable database = new Hashtable();
    private int serial = 100;
```

## 练习问题的解答

```

protected synchronized Product createProduct(String owner) {
    return new IDCard(owner, serial++);
}

protected void registerProduct(Product product) {
    IDCard card = (IDCard)product;
    database.put(card.getOwner(), new Integer(card.getSerial()));
}

public Hashtable getDatabase() {
    return database;

}
}

```

建立结城浩(100)的卡。  
 建立户村(101)的卡。  
 建立佐藤花子(102)的卡。  
 使用结城浩(100)的卡。  
 使用户村(101)的卡。  
 使用佐藤花子(102)的卡。

图 A4-1 执行结果

**问题 4-3 的解答**

因为在 Java 无法产生 abstract 的构造函数。Java 里不会继承构造函数，所以 abstract 的构造函数没有意义。

不要直接用构造函数对产品命名，而是必须另外声明一个对产品命名的方法。

**第 5 章****问题 5-1 的解答**

如程序 A5-1 所示。

虽然这个问题跟 Singleton Pattern 没什么关系，不过还是请注意到这里把 getNextTicketNumber 设成 synchronized 方法。这是为了即使有从多个串行调用 getNextTicketNumber 方法时也能正常运行的特殊处理。如果没有 synchronized 的话，可能会对多个串行返回同值。

## Java语言中的应用

## 程序 A5-1 变成 Singleton Pattern 后的 TicketMaker 类 (TicketMaker.java)

```
public class TicketMaker {
    private int ticket = 1000;
    private static TicketMaker singleton = new TicketMaker();
    private TicketMaker() {
    }
    public static TicketMaker getInstance() {
        return singleton;
    }
    public synchronized int getNextTicketNumber() {
        return ticket++;
    }
}
```

## 程序 A5-2 利用 TicketMaker 类的类 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Start.");
        for (int i = 0; i < 10; i++) {
            System.out.println(i + ":" +
                TicketMaker.getInstance().getNextTicketNumber());
        }
        System.out.println("End.");
    }
}
```

Start.  
0:1000  
1:1001  
2:1002  
3:1003  
4:1004  
5:1005  
6:1006  
7:1007  
8:1008  
9:1009  
End.

图 A5-1 执行结果

## 问题 5-2 的解答

让 Triple 类（程序 A5-3）的对象实例有自己的编号（id）。先做一个数组 triple 并产生 3 个对象实例，当作 static 变量。getInstance 方法则以参数所接收到的 id 为标注，从数组 triple 只返回 1 个对象实例。

预先建立 toString 方法，这样将 Triple 的对象实例输出成字符串时就能知道其 id。

**程序 A5-3** Triple 类 (Triple.java)

```
public class Triple {
    private static Triple[] triple = new Triple[] {
        new Triple(0),
        new Triple(1),
        new Triple(2),
    };
    private int id;
    private Triple(int id) {
        System.out.println("The instance " + id + " is created.");
        this.id = id;
    }
    public static Triple getInstance(int id) {
        return triple[id];
    }
    public String toString() {
        return "[Triple id=" + id + "]";
    }
}
```

**程序 A5-4** 利用 Triple 类的类 (TripleMain.java)

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Start.");
        for (int i = 0; i < 9; i++) {
            Triple triple = Triple.getInstance(i % 3);
            System.out.println(i + ":" + triple);
        }
        System.out.println("End.");
    }
}
```

```

    }
}

```

```

Start.
The instance 0 is created.
The instance 1 is created.
The instance 2 is created.
0:[Triple id=0]
1:[Triple id=1]
2:[Triple id=2]
3:[Triple id=0]
4:[Triple id=1]
5:[Triple id=2]
6:[Triple id=0]
7:[Triple id=1]
8:[Triple id=2]
End.

```

图 A5-2 执行结果

**说明：给还不熟悉类字段的读者**

这里是以 `Triple` 的对象实例去初始化 `Triple` 类的 `triple` 字段，但并不是无穷循环。有人也许会觉得明明就是在产生自己时还要用到自己，不过这只是错觉。因为 `triple` 字段是类字段而非对象实例字段，所以它不会变成无穷循环。只有在一开始的第一次才有初始化 `triple` 字段，并不是每次建立 `Triple` 类的对象实例时都会产生 `triple` 字段。如果 `triple` 字段没有 `static` 的话，就会变成无穷循环，造成执行时的错误（`stack overflow`）。

**问题 5-3 的解答**

因为从多个串行几乎同时调用 `Singleton.getInstance` 方法时，可能会产生多个对象实例。

**程序 A5-5 从多个串行调用 `Singleton.getInstance` (`Main.java`)**

```

public class Main extends Thread {
    public static void main(String[] args) {
        System.out.println("Start.");
        new Main("A").start();
        new Main("B").start();
    }
}

```

## 练习问题的解答

```
new Main("C").start();
System.out.println("End.");
}

public void run() {
    Singleton obj = Singleton.getInstance();
    System.out.println(getName() + ": obj = " + obj);
}

public Main(String name) {
    super(name);
}

}
```

程序 A5-5 的执行结果会因执行时的机器状态而异，所以如果要产生多个对象实例则应写成如程序 A5-6。其执行结果可能如图 A5-3 所示。

**程序 A5-6 为了能产生多个对象实例而故意减慢速度 (Singleton.java)**

```
public class Singleton {
    private static Singleton singleton = null;
    private Singleton() {
        System.out.println("已产生对象实例。");
        slowdown();
    }
    public static Singleton getInstance() {
        if (singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
    private void slowdown() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
    }
}
```

```

Start.
End.
已产生对象实例。          -多次产生对象实例
已产生对象实例。
已产生对象实例。
B: obj = Singleton@2a9835  ← A, B, C 的对象实例内容都不一样
C: obj = Singleton@6ec612
A: obj = Singleton@dd1f7

```

图 A5-3 执行结果示例

会变成这样是因为下面的条件判断不够严谨（非串行存储）的关系：

```

if (singleton == null) {
    singleton = new Singleton();
}

```

这是利用：

```
singleton == null
```

判断其是否为第一个对象实例之后，再执行：

```
singleton = new Singleton();
```

这样可能会在执行指定之前，其他串行就已经抢先评估第一个条件表达式。

只要如程序 A5-7 利用 synchronized，即可修改成滴水不漏的 Singleton Pattern（这个解决方式参考《Java 格言》）。

#### 程序 A5-7 改成 Singleton Pattern 的结果 (Singleton.java)

```

public class Singleton {
    private static Singleton singleton = null;
    private Singleton() {
        System.out.println("已产生对象实例。");
        slowdown();
    }
    public static synchronized Singleton getInstance() {
        if (singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
    private void slowdown() {

```

```

try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
}
}

```

---

## 第 6 章

### 问题 6-1 的解答

方法不只一种，例如：

- 将 Product 接口改为 Product 类，而在 Product 类当中实现 createClone 方法（Template Method Pattern）。
- 定义一个 ConcreteProduct 类作为 UnderlinePen 类和 MessageBox 类共同的父类。ConcreteProduct 类实现 Product 接口，而在其中又实现 createClone 方法。两者都是利用继承 createClone 方法的方式而共有。

### 问题 6-2 的解答

没有。java.lang.Object 并未实现 java.lang.Cloneable 接口。

如果 Object 类有实现 Cloneable 接口的话，那么无论是在哪个类的对象实例调用 clone 方法都不会抛出异常 CloneNotSupportedException。

## 第 7 章

### 问题 7-1 的解答

有下列 3 个部分需要修改。但程序 7-2 的 Director 类和程序 7-5 的 Main 类不需修改。

- Builder 类（程序 7-1）的修改部分

```

public abstract class Builder
{
    public interface Builder

```

## Java语言中的应用

- TextBuilder 类（程序 7-3）的修改部分

```
public class TextBuilder extends Builder
{
    ↓
```

```
public class TextBuilder implements Builder
```

- HTMLBuilder 类（程序 7-4）的修改部分

```
public class HTMLBuilder extends Builder
{
    ↓
```

```
public class HTMLBuilder implements Builder
```

## 问题 7-2 的解答

加入在 Builder 类的阶段就检查调用顺序的程序代码。接着再将子类应实现的方法修改如下（Template Method Pattern）。

```
makeTitle → buildTitle
makeString → buildString
makeItems → buildItems
getResult → buildResult
```

因为只有子类使用到上面的 buildXXXX 方法，所以不是 public 而是 protected。  
采用这种做法就不需要修改 Director 类。

---

### 程序 A7-1    Builder 类 (Builder.java)

---

```
public abstract class Builder {
    private boolean initialized = false;
    public void makeTitle(String title) {
        if (!initialized) {
            buildTitle(title);
            initialized = true;
        }
    }
    public void makeString(String str) {
        if (initialized) {
            buildString(str);
        }
    }
    public void makeItems(String[] items) {
        if (initialized) {
            buildItems(items);
        }
    }
}
```

## 练习问题的解答

```

    }
    public Object getResult() {
        if (initialized) {
            return buildResult();
        } else {
            return null;
        }
    }
    protected abstract void buildTitle(String title);
    protected abstract void buildString(String str);
    protected abstract void buildItems(String[] items);
    protected abstract Object buildResult();
}

```

---

## 程序 A7-2 HTMLBuilder 类 (HTMLBuilder.java)

```

import java.io.*;

public class HTMLBuilder extends Builder {
    private String filename; // 产生的文件名
    private PrintWriter writer; // 写入到文件的 PrintWriter
    public void buildTitle(String title) { // HTML 文件的标题
        filename = title + ".html"; // 根据标题决定文件名
        try {
            writer = new PrintWriter(new FileWriter(filename));
            // 建立 PrintWriter
        } catch (IOException e) {
            e.printStackTrace();
        }
        writer.println("<html><head><title>" + title + "</title>");
        // 输出标题
        writer.println("<h1>" + title + "</h1>");
    }
    public void buildString(String str) { // HTML 文件的字符串
        writer.println("<p>" + str + "</p>"); // 以<p>标签输出
    }
    public void buildItems(String[] items) { // HTML 文件的项目
        writer.println("<ul>"); // 以<ul>和<li>输出
    }
}

```

## Java語言中的應用

```

        for (int i = 0; i < items.length; i++) {
            writer.println("<li>" + items[i] + "</li>");
        }
        writer.println("</ul>");
    }

    public Object buildResult() { // 完成的文件
        writer.println("</body></html>"); // 關閉標籤

        writer.close(); // 關閉文件
        return filename; // 返回文件名
    }
}

```

---

程序 A7-3 TextBuilder 类 (TextBuilder.java)

```

public class TextBuilder extends Builder {
    private StringBuffer buffer = new StringBuffer();
    // 开始在此字段建立文件

    public void makeTitle(String title) { // 一般文字格式的标题
        buffer.append("=====\n"); //花边
        buffer.append("!" + title + "!\n"); // 有!的标题
        buffer.append("\n"); // 空行
    }

    public void makeString(String str) { // 一般文字格式的字符串
        buffer.append('■' + str + "\n"); // 有■的字符串
        buffer.append("\n"); // 空行
    }

    public void makeItems(String[] items) { // 一般文字格式的项目
        for (int i = 0; i < items.length; i++) {
            buffer.append("*" + items[i] + "\n"); // 有*的项目
        }
        buffer.append("\n"); // 空行
    }

    public Object getResult() { // 完成的文件
        buffer.append("=====\n"); //花边
        return buffer.toString(); //把 StringBuffer 转换成 String
    }
}

```

---

## 练习问题的解答

还可更进一步加上“若已调用 getResult，则不可再调用任何方法”的限制。

以上的解答仅为参考示例，当然还有其他不同做法，请继续动动脑筋。

### 问题 7-3 的解答

试建立一个利用 JFC (Java Foundation Classes) 的图形用户界面 (GUI) 作为 ConcreteBuilder 参与者。在程序 A7-4 的 FrameBuilder 类，makeString 的部分是窗口上的卷标 (JLabel)，makeItems 的部分则是窗口上的按键 (JButton)。Builder 类和 Director 类则同程序示例。

程序执行后，便出现如图 A7-1 的窗口。点击按键，标准输出的地方就会有按键上的“早安。”等字样。

程序 A7-4 FrameBuilder 类 (FrameBuilder.java)

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class FrameBuilder extends Builder implements ActionListener
{
    private JFrame frame = new JFrame();
    private Box box = new Box(BoxLayout.Y_AXIS);
    public void makeTitle(String title) {
        frame.setTitle(title);
    }
    public void makeString(String str) {
        box.add(new JLabel(str));
    }
    public void makeItems(String[] items) {
        Box innerbox = new Box(BoxLayout.Y_AXIS);
        for (int i = 0; i < items.length; i++) {
            JButton button = new JButton(items[i]);
            button.addActionListener(this);
            innerbox.add(button);
        }
        box.add(innerbox);
    }
    public Object getResult() {
        frame.getContentPane().add(box);
        frame.setVisible(true);
        return null;
    }
}
```

```
frame.pack();
frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});
return frame;
}
public void actionPerformed(ActionEvent e) {
    System.out.println(e.getActionCommand());
}
}
```

---

### 程序 A7-5 Main 类 (Main.java)

---

```
import javax.swing.*;

public class Main {
    public static void main(String[] args) {
        Director director = new Director(new FrameBuilder());
        JFrame frame = (JFrame)director.construct();
        frame.setVisible(true);
    }
}
```

---

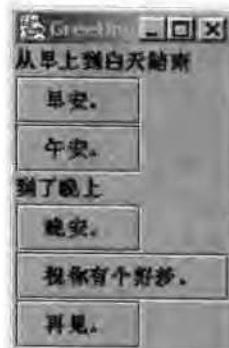


图 A7-1 执行结果

## 问题 7-4 的解答

就算要用 String，只要如程序 A7-6 把 append 修改成 += 就行了。不过像这样经常需要修改或新增字符串的程序，利用 StringBuffer 的执行速度会比 String 更快。因为利用 String 时，每次修改或新增时就会产生一个 String 类的新的对象实例。

程序 A7-6 TextBuilder 类 (TextBuilder.java)

```
public class TextBuilder extends Builder {
    private String buffer = ""; // 开始在此字段建立文件
    public void makeTitle(String title) { // 一般文字格式的标题
        buffer += "=====\n"; // 花边
        buffer += "!" + title + "!\n"; // 有!的标题
        buffer += "\n"; // 空行
    }
    public void makeString(String str) { // 一般文字格式的字符串
        buffer += '■' + str + "\n"; // 有■的字符串
        buffer += "\n"; // 空行
    }
    public void makeItems(String[] items) { // 一般文字格式的项目
        for (int i = 0; i < items.length; i++) {
            buffer += " * " + items[i] + "\n"; // 有*的项目
        }
        buffer += "\n"; // 空行
    }
    public Object getResult() { // 完成的文件
        buffer += "=====\n"; // 花边
        return buffer;
    }
}
```

## 第 8 章

### 问题 8-1 的解答

改成 private 的好处是 Tray 的子类（即具体的零件）的程序代码不会跟 tray 字段的实现有依赖关系。

而改成 private 的坏处是必须要建立一个适当存取用的方法供使用。如果子类不

能直接看得到 tray 字段（即 tray 设为 private），则必须另有一个取得 tray 字段值的方法。

## 问题 8-2 的解答

如下所示。只有 Factory 类和 Main 类需要修改。

程序 A8-1 Factory 类 (Factory.java)

```
package factory;

public abstract class Factory {
    public static Factory getFactory(String classname) {
        Factory factory = null;
        try {
            factory = (Factory)Class.forName(classname).newInstance();
        } catch (ClassNotFoundException e) {
            System.err.println("找不到类 " + classname + "。");
        } catch (Exception e) {
            e.printStackTrace();
        }
        return factory;
    }

    public abstract Link createLink(String caption, String url);
    public abstract Tray createTray(String caption);
    public abstract Page createPage(String title, String author);
    public Page createYahooPage() {
        Link link = createLink("Yahoo!", "http://www.yahoo.com/");
        Page page = createPage("Yahoo!", "Yahoo!");
        page.add(link);
        return page;
    }
}
```

程序 A8-2 Main 类 (Main.java)

```
import factory.*;
```

## 练习问题的解答

```

public class Main {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java Main class.name.of.Concrete
Factory");
            System.out.println("Example 1: java Main listfactory.List
Factory");
            System.out.println("Example 2: java Main tablefactory.Table
Factory");
            System.exit(0);
        }
        Factory factory = Factory.getFactory(args[0]);
        Page page = factory.createYahooPage();
        page.output();
    }
}

```



图 A8-1 利用 listfactory 写成的 Yahoo!链接



图 A8-2 利用 tablefactory 写成的 Yahoo!链接

**问题 8-3 的解答**

因为在 Java 不能继承构造函数。

即使父类有 Link (String caption, String url) 构造函数，只要 ListLink 端没有定义 ListLink (String caption, String url) 构造函数，还是不能调用：

```
new ListLink("Yahoo!", "http://www.yahoo.com/")
```

这样会造成编译时的错误。

## 问题 8-4 的解答

因为 Page 不能 add 到 Tray (HTML 语法不符)。如果把 Page 类设为 Tray 类的子类，则 Page 也是 Item 的子类，故亦可 add。

但是必须在 Page 类声明 makeHTML。若如下建立一个含有 makeHTML 方法的 Java 的接口 HTMLable，修改为由 Item 类和 Page 类 implements HTMLable，则可让整个程序更加严谨。

```
public interface HTMLable {
    public abstract String makeHTML();
}
```

## 第 9 章

### 问题 9-1 的解答

这是新增到“功能的类层次”。

RandomCountDisplay 类（程序 A9-1）也可以做成 Display 的子类，但这里设为 CountDisplay 的子类。

java.util.Random 表示随机数产生器的类，nextInt(n)方法随机产生大于 0、小于 n 的整数作为返回值。

在程序 A9-2 中 Main 类调用 RandomCountDisplay 的 randomDisplay 方法。一执行此程序，画面上就会出现 0~9 个“Hello, Japan.”的字符串。

#### 程序 A9-1 RandomCountDisplay 类 (RandomCountDisplay.java)

```
import java.util.Random;

public class RandomCountDisplay extends CountDisplay {
    private Random random = new Random();
    public RandomCountDisplay(DisplayImpl impl) {
        super(impl);
    }
    public void randomDisplay(int times) {
        multiDisplay(random.nextInt(times));
    }
}
```

## 程序 A9-2 Main 类 (Main.java)

```
public class Main {  
    public static void main(String[] args) {  
        RandomCountDisplay d = new RandomCountDisplay(new String  
DisplayImpl("Hello, Taiwan."));  
        d.randomDisplay(10);  
    }  
}
```

```
+-----+  
|Hello, Taiwan.!  
|Hello, Taiwan.!  
|Hello, Taiwan.!  
|Hello, Taiwan.!  
+-----+
```

图 A9-1 执行示例 1 (重复 4 次; 可能每次执行都不一样)

```
+-----+  
|Hello, Taiwan.!  
+-----+
```

图 A9-2 执行示例 2 (重复 7 次)

```
+-----+  
+-----+
```

图 A9-3 执行示例 3 (重复 0 次)

类图如图 A9-4 所示。

## Java 实现中的应用

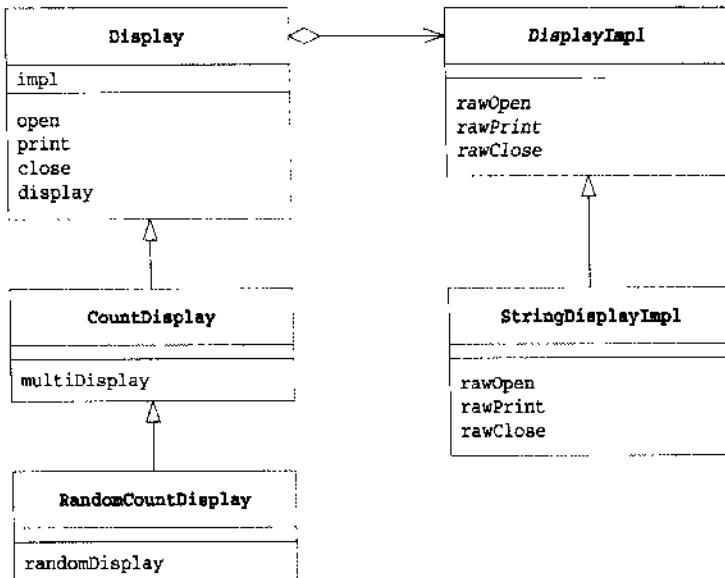


图 A9-4 新增了 RandomCountDisplay 后的类图

## 问题 9-2 的解答

这是新增到“实现的类层次”。建立一个 `FileDisplayImpl` 类作为 `DisplayImpl` 类的子类。

程序 A9-3 的 `FileDisplayImpl` 类只是直接打印，显得太单调，这里稍微做点美工。

程序 A9-4 的 Main 类则利用 `CountDisplay` 类和 `FileDisplayImpl` 类输出 3 次 star.txt 文件（程序 A9-5）。

利用前一个问题中的 `RandomCountDisplay` 和 `FileDisplayImpl`，就能将文件打印随机次数。

## 程序 A9-3 FileDisplayImpl 类 (FileDisplayImpl.java)

```

import java.io.*;

public class FileDisplayImpl extends DisplayImpl {
    private String filename;
    private BufferedReader reader;
    private final int MAX_READAHEAD_LIMIT = 4096;
    // 可反复打印的上限 (缓冲器容量)
    public FileDisplayImpl(String filename) {
        this.filename = filename;
    }
}
  
```

## 练习问题的解答

```

    }
    public void rawOpen() {
        try {
            reader = new BufferedReader(new FileReader(filename));
            reader.mark(MAX_READAHEAD_LIMIT);
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println("===== " + filename + " =====");
    }
    public void rawPrint() {
        try {
            String line;
            reader.reset(); // 返回到 mark 的位置
            while ((line = reader.readLine()) != null) {
                System.out.println("> " + line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public void rawClose() {
        System.out.println("===== "); // 花边
        try {
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

---

## 程序 A9-4 Main 类 (Main.java)

```

public class Main {
    public static void main(String[] args) {
        CountDisplay d = new CountDisplay(new FileDisplayImpl
("star.txt"));
        d.multiDisplay(3);
    }
}

```

```

    }
}

```

**程序 A9-5 打印用文本文件 (star.txt)**

```

Twinkle, twinkle, little star,
How I wonder what you are.

```

```

===== star.txt =====
> Twinkle, twinkle, little star,
> How I wonder what you are.
> Twinkle, twinkle, little star,
> How I wonder what you are.
> Twinkle, twinkle, little star,
> How I wonder what you are.
=====
```

图 A9-5 执行结果

类图如图 A9-6 (p.371) 所示。

### 问题 9-3 的解答

这个问题是要在一个类里同时新增功能和实现。不过，要先把欲新增的内容中的功能和实现两部分分成不同的类，一个放到功能的类层次、另一个则放到实现的类层次，这样就能符合 Bridge Pattern 的要求。而且这样一来，从其他类（CountDisplay 或 RandomCountDisplay）也可以使用新增的实现，新增的功能也能在其他类（StringDisplayImpl 或 FileDisplayImpl）上被调用。

类似这样分离的类称为“Cheshire cat（微笑猫）”(J. Carolan 命名)。微笑猫是刘易斯·凯罗的童话故事“爱丽思梦游仙境”里的那只奇妙的猫，在故事中牠会脸上浮现一抹深不可测的微笑而消失在黑暗里。命名的意义则是把“微笑的猫”分成“微笑”（功能）和“猫”（实现）。

要能符合问题要求的类可分成下面 2 个类。

- IncreaseDisplay 类（程序 A9-6）——表示输出内容逐渐增加次数的“功能”的类
- CharDisplayImpl 类（程序 A9-7）——表示输出成字符的“实现”的类

## 练习问题的解答

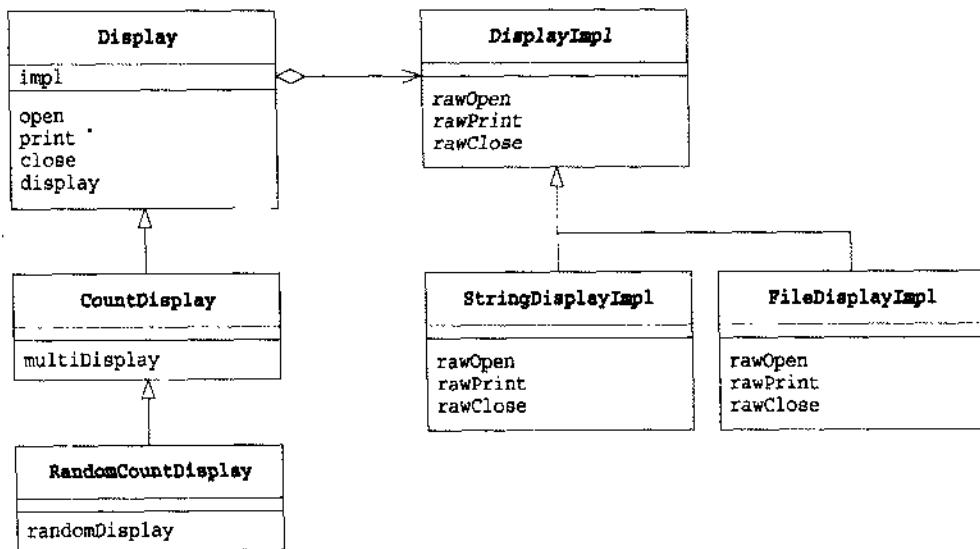


图 A9-8 新增了 FileDisplayImpl 之后的类图

## 程序 A9-6 IncreaseDisplay 类 (IncreaseDisplay.java)

```

public class IncreaseDisplay extends CountDisplay {
    private int step; // 增加次数
    public IncreaseDisplay(DisplayImpl impl, int step) {
        super(impl);
        this.step = step;
    }
    public void increaseDisplay(int level) {
        int count = 0;
        for (int i = 0; i < level; i++) {
            multiDisplay(count);
            count += step;
        }
    }
}
  
```

## 程序 A9-7 CharDisplayImpl 类 (CharDisplayImpl.java)

```

public class CharDisplayImpl extends DisplayImpl {
    private char head;
    private char body;
    private char foot;
  
```

## 例 A9-8 CharDisplay 的实现

```
public CharDisplayImpl(char head, char body, char foot) {
    this.head = head;
    this.body = body;
    this.foot = foot;
}
public void rawOpen() {
    System.out.print(head);
}
public void rawPrint() {
    System.out.print(body);
}
public void rawClose() {
    System.out.println(foot);
}
}
```

---

### 程序 A9-8 Main 类 (Main.java)

---

```
public class Main {
    public static void main(String[] args) {
        IncreaseDisplay d1 = new IncreaseDisplay(new CharDisplayImpl('<', '*', '>'), 1);
        IncreaseDisplay d2 = new IncreaseDisplay(new CharDisplayImpl('!', '#', '-'), 2);
        d1.increaseDisplay(4);
        d2.increaseDisplay(6);
    }
}
```

---

```
<>
<*>
<**>
<***>
!-
|#-
|##-
|###-
|####-
|#####
|#####-
```

图 A9-7 执行结果

## 练习问题的解答

类图如图 A9-8 所示。这里的类图包含所有原本程序示例中的类和练习题中的类。左边是功能的类层次，右边是实现的类层次，各位应该看得出来连接两者的委让桥梁吧？

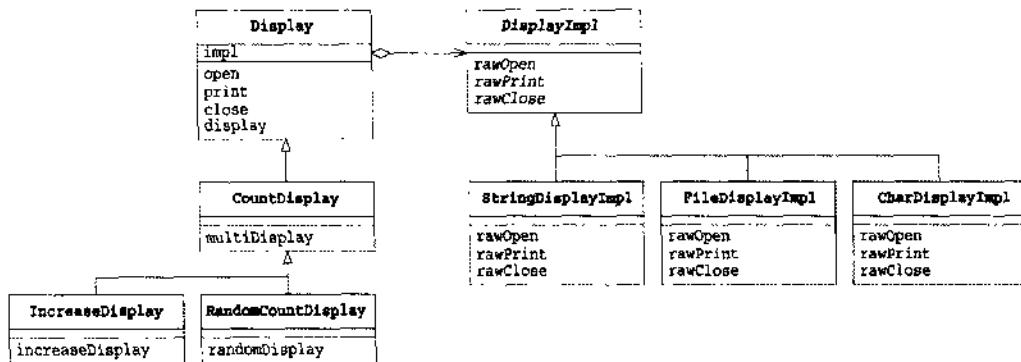


图 A9-8 新增了 IncreaseDisplay 和 CharDisplayImpl 之后的类图

## 第 10 章

### 问题 10-1 的解答

如程序 A10-1 所示。因为是随便决定下一个手势，所以 `study` 是空的。

#### 程序 A10-1 RandomStrategy 类 (`RandomStrategy.java`)

```

import java.util.Random;

public class RandomStrategy implements Strategy {
    private Random random;
    public RandomStrategy(int seed) {
        random = new Random(seed);
    }
    public void study(boolean win) {
    }
    public Hand nextHand() {
        return Hand.getHand(random.nextInt(3));
    }
}
  
```

## 面向对象的编程

## 程序 A10-2 Main 类 (Main.java)

```
public class Main {  
    public static void main(String[] args) {  
        if (args.length != 2) {  
            System.out.println("Usage: java Main randomseed1  
randomseed2");  
            System.out.println("Example: java Main 314 15");  
            System.exit(0);  
        }  
        int seed1 = Integer.parseInt(args[0]);  
        int seed2 = Integer.parseInt(args[1]);  
        Player player1 = new Player("Taro", new ProbStrategy(seed1));  
        Player player2 = new Player("Hana", new RandomStrategy(seed2));  
        for (int i = 0; i < 10000; i++) {  
            Hand nextHand1 = player1.nextHand();  
            Hand nextHand2 = player2.nextHand();  
            if (nextHand1.isStrongerThan(nextHand2)) {  
                System.out.println("Winner:" + player1);  
                player1.win();  
                player2.lose();  
            } else if (nextHand2.isStrongerThan(nextHand1)) {  
                System.out.println("Winner:" + player2);  
                player1.lose();  
                player2.win();  
            } else {  
                System.out.println("Even...");  
                player1.even();  
                player2.even();  
            }  
        }  
        System.out.println("Total result:");  
        System.out.println(" " + player1);  
        System.out.println(" " + player2);  
    }  
}
```

## 问题 10-2 的解答

因为 Hand 类的对象实例本来就只有 3 个(剪刀、石头、布)。如果有 2 个 handvalue 字段的值相等的对象实例，事实上两者就是同一对象实例。

## 问题 10-3 的解答

因为在 Java 当中，会自动去初始化那些没有清楚初始化的字段。boolean 类型字段在 false 时初始化。数值和参照类型则分别在 0、null 之时初始化。

注意：虽然字段被初始化，但局部变量（区域变量）并没有初始化。

## 问题 10-4 的解答

QuickSorter（程序 A10-3）是使用了快速排序算法的类。

程序 A10-3 QuickSorter 类 (QuickSorter.java)

```
public class QuickSorter implements Sorter {
    Comparable[] data;
    public void sort(Comparable[] data) {
        this.data = data;
        qsort(0, data.length - 1);
    }
    private void qsort(int pre, int post) {
        int saved_pre = pre;
        int saved_post = post;
        Comparable mid = data[(pre + post) / 2];
        do {
            while (data[pre].compareTo(mid) < 0) {
                pre++;
            }
            while (mid.compareTo(data[post]) < 0) {
                post--;
            }
            if (pre <= post) {
                Comparable tmp = data[pre];
                data[pre] = data[post];
                data[post] = tmp;
                pre++;
            }
        }
    }
}
```

```

        post--;
    }
} while (pre <= post);
if (saved_pre < post) {
    qsort(saved_pre, post);
}
if (pre < saved_post) {
    qsort(pre, saved_post);
}
}
}
}

```

---

**程序 A10-4 Main 类 (Main.java)**

```

public class Main {
    public static void main(String[] args) {
        String[] data = {
            "Dumpty", "Bowman", "Carroll", "Elfland", "Alice",
        };
        SortAndPrint sap = new SortAndPrint(data, new QuickSorter
());
        sap.execute();
    }
}

```

---

## 第 11 章

### 问题 11-1 的解答

例如 HTML 的项目清单（ul 元素、ol 元素、dl 元素）或表格（table）等都可以表现成 Composite Pattern。

### 问题 11-2 的解答

方法很多种，这里只举出把 parent 字段（有此对象实例的 Directory 的对象实例）保留在 Entry 类的做法。在根目录（最上层目录）时，则 parent 字段为 null。从接收到的对象实例往上回溯到 parent 字段，即可建立出全路径。要修改的是 Entry 类和

## 练习问题的解答

Directory 类。在 Directory 类则用 add 方法修改 parent 字段。

**程序 A11-1 修改后的 Entry 类 (Entry.java)**

```
public abstract class Entry {
    protected Entry parent;
    public abstract String getName();
    public abstract int getSize();
    public Entry add(Entry entry) throws FileTreatmentException {
        throw new FileTreatmentException();
    }
    public void printList() {
        printList("");
    }
    protected abstract void printList(String prefix);
    public String toString() {
        return getName() + " (" + getSize() + ")";
    }
    public String getFullName() {
        StringBuffer fullname = new StringBuffer();
        Entry entry = this;
        do {
            fullname.insert(0, "/" + entry.getName());
            entry = entry.parent;
        } while (entry != null);
        return fullname.toString();
    }
}
```

**程序 A11-2 修改后的 Directory 类 (Directory.java)**

```
import java.util.Iterator;
import java.util.Vector;

public class Directory extends Entry {
    private String name;
    private Vector directory = new Vector();
    public Directory(String name) {
        this.name = name;
    }
}
```

## Java面向对象设计

```

public String getName() {
    return name;
}
public int getSize() {
    int size = 0;
    Iterator it = directory.iterator();
    while (it.hasNext()) {
        Entry entry = (Entry)it.next();
        size += entry.getSize();
    }
    return size;
}
public Entry add(Entry entry) {
    directory.add(entry);
    entry.parent = this;
    return this;
}
protected void printList(String prefix) {
    System.out.println(prefix + "/" + this);
    Iterator it = directory.iterator();
    while (it.hasNext()) {
        Entry entry = (Entry)it.next();
        entry.printList(prefix + "/" + name);
    }
}
}

```

## 程序 A11-3 Main 类 (Main.java)

```

public class Main {
    public static void main(String[] args) {
        try {
            Directory rootdir = new Directory("root");

            Directory usrdir = new Directory("usr");
            rootdir.add(usrdir);

            Directory yuki = new Directory("yuki");
            usrdir.add(yuki);
        }
    }
}

```

## 练习问题的解答

```

        File file = new File("Composite.java", 100);
        yuki.add(file);
        rootdir.printList();

        System.out.println("");
        System.out.println("file = " + file.getFullName());
        System.out.println("yuki = " + yuki.getFullName());
    } catch (FileTreatmentException e) {
        e.printStackTrace();
    }
}
}

```

```

/root (100)
/root/usr (100)
/root/usr/yuki (100)
/root/usr/yuki/Composite.java (100)

file = /root/usr/yuki/Composite.java
yuki = /root/usr/yuki

```

图 A11-1 执行结果

## 第 12 章

### 问题 12-1 的解答

解答如下。

#### 程序 A12-1 UpDownBorder 类 (UpDownBorder.java)

```

public class UpDownBorder extends Border {
    private char borderChar;           // 装饰字符
    public UpDownBorder(Display display, char ch) {
        // 以构造函数指定 Display 和装饰字符
        super(display);
        this.borderChar = ch;
    }
}

```

## 12.2 多行显示类 (MultiStringDisplay)

```

public int getColumns() {           // 字数同内容的字数
    return display.getColumns();
}
public int getRows() {   // 行数要再加上内容上下的装饰字符的行数
    return 1 + display.getRows() + 1;
}
public String getRowText(int row) { // 指定该行的内容
    if (row == 0 || row == getRows() - 1) {
        return makeLine(borderChar, getColumns());
    } else {
        return display.getRowText(row - 1);
    }
}
private String makeLine(char ch, int count) {
// 以字符 ch 建立重复 count 次的连续字符串
    StringBuffer buf = new StringBuffer();
    for (int i = 0; i < count; i++) {
        buf.append(ch);
    }
    return buf.toString();
}
}

```

---

FullBorder 类 (程序 12-5) 里也用到 makeLine 方法。把 makeLine 方法放到 Border 类 (程序 12-3) 里并设成 protected，也是一种做法。

## 问题 12-2 的解答

解答如下。updateColumn 方法是在字符串的最后面加入一个空白，让宽幅保持固定。

### 程序 A12-2 MultiStringDisplay 类 (MultiStringDisplay.java)

```

import java.util.Vector;

public class MultiStringDisplay extends Display {
    private Vector body = new Vector(); // 打印的字符串
    private int columns = 0;           // 最大字数
    public void add(String msg) {      // 新增字符串
        body.add(msg);
    }
}

```

## 练习问题的解答

```

        updateColumn(msg);
    }

    public int getColumns() { // 字数
        return columns;
    }

    public int getRows() { // 行数
        return body.size();
    }

    public String getRowText(int row) { // 该行的内容
        return (String)body.get(row);
    }

    private void updateColumn(String msg) { // 更新字数
        if (msg.getBytes().length > columns) {
            columns = msg.getBytes().length;
        }
        for (int row = 0; row < body.size(); row++) {
            int fills = columns - ((String)body.get(row)).getBytes()
                ().length;
            if (fills > 0) {
                body.set(row, body.get(row) + spaces(fills));
            }
        }
    }

    private String spaces(int count) { // 产生空白
        StringBuffer buf = new StringBuffer();
        for (int i = 0; i < count; i++) {
            buf.append(' ');
        }
        return buf.toString();
    }
}

```

**第 13 章****问题 13-1 的解答**

如程序 A13-1 所示。不需要修改 File.java(程序 13-4)或 Directory.java(程序 13-5)。

## 逐步分析综合的应用

## 程序 A13-1 FileFindVisitor 类 (FileFindVisitor.java)

```

import java.util.Iterator;
import java.util.Vector;

public class FileFindVisitor extends Visitor {
    private String filetype;
    private Vector found = new Vector();
    public FileFindVisitor(String filetype) {
        // 设定时在扩展名前加上., 如".txt"
        this.filetype = filetype;
    }
    public Iterator getFoundFiles() { // 取得找到的文件
        return found.iterator();
    }
    public void visit(File file) { // 访问文件时即调用
        if (file.getName().endsWith(filetype)) {
            found.add(file);
        }
    }
    public void visit(Directory directory) { // 访问目录时即调用
        Iterator it = directory.iterator();
        while (it.hasNext()) {
            Entry entry = (Entry)it.next();
            entry.accept(this);
        }
    }
}

```

## 问题 13-2 的解答

如程序 A13-2, 程序 A13-3 所示。执行结果则同改写前的 Directory 类 (图 13-2)。

## 程序 A13-2 Directory 类 (Directory.java)

```

import java.util.Iterator;
import java.util.Vector;

public class Directory extends Entry {
    private String name; // 目录名称
    private Vector dir = new Vector(); // 目录进入点的集合

```

## 练习问题的解答

```

public Directory(String name) {           // 构造函数
    this.name = name;
}
public String getName() {                // 取得名称
    return name;
}
public int getSize() {                  // 取得目录容量
    SizeVisitor v = new SizeVisitor();
    accept(v);
    return v.getSize();
}
public Entry add(Entry entry) {          // 新增进入点
    dir.add(entry);
    return this;
}
public Iterator iterator() {
    return dir.iterator();
}
public void accept(Visitor v) {
    v.visit(this);
}
}

```

---

## 程序 A13-3 SizeVisitor 类 (SizeVisitor.java)

```

import java.util.Iterator;

public class SizeVisitor extends Visitor {
    private int size = 0;
    public int getSize() {
        return size;
    }
    public void visit(File file) {
        size += file.getSize();
    }
    public void visit(Directory directory) {
        Iterator it = directory.iterator();
        while (it.hasNext()) {
            Entry entry = (Entry)it.next();

```

```
        entry.accept(this);
    }
}
}
```

---

## 问题 13-3 的解答

AcceptorVector 类(程序 A13-4)是 java.util.Vector 的子类, 被定义要实现 Acceptor。add 方法是继承自 Vector, 所以不须另行定义。

程序 A13-4 AcceptorVector 类 (AcceptorVector.java)

---

```
import java.util.Vector;
import java.util.Iterator;

class AcceptorVector extends Vector implements Acceptor {
    public void accept(Visitor v) {
        Iterator it = iterator();
        while (it.hasNext()) {
            Acceptor a = (Acceptor) it.next();
            a.accept(v);
        }
    }
}
```

---

## 问题 13-4 的解答

效率的关系。

String 类是处理 Java 字符串的基类, 扮演相当重要的角色。因此, Java 语言的处理会在 String 类“没有扩充”的前提条件下, 执行处理速度最佳化和内存容量最佳化。

# 第 14 章

---

## 问题 14-1 的解答

通常是在组件所在的父窗口为“next”的情形下。传递给组件的事件(要求)如果不是由该组件处理时, 会继续传过去给父窗口。

## 问题 14-2 的解答

图 14-5 对话窗口的职责链即如图 A14-1 所示。箭头表示责任转送出去的位置 next。 “输入的方向键”会变成事件（要求），传送到现在的有效行为窗口（列表或复选框）。列表窗口会自行处理↑↓键，所以不再传给 next 的对话窗口。但复选框本身无法处理↑↓键，因此会再传给 next 的对话窗口。↑↓键的事件（要求）被转送过来的对话窗口则进行将有效动作位置从复选框移到清单窗口的处理。这样就会产生如问题 14-2 所述的现象。

## 问题 14-3 的解答

当其他类要要求 Support 类的对象实例帮忙“解决问题”时，请不要用 resolve 方法，要使用 support 方法。

如果 resolve 方法是 public，那么从跟 Support 类没有关系的类也能调用 resolve。这样一来，resolve 方法的使用方式可能会跟原本 Support 类预期的差了十万八千里。

而且若 resolve 方法设为 public，将来如果修改 resolve 方法名称或产品序号（signature）的话，程序里同时也要跟着修改的部分就会没有中心点，七零八落难以收拾。

**注意：**在 Java 环境下，不只是从子类可以看见 protected 的名称，从同一包的类也同样可以看见。因此，如果是像程序示例这样只在同一个包里使用的话，public 或 protected 都没有差别（程序示例的类均在未命名的包里），不过，如果将来类移动到其他特殊包时，就会发挥 protected 的效果。

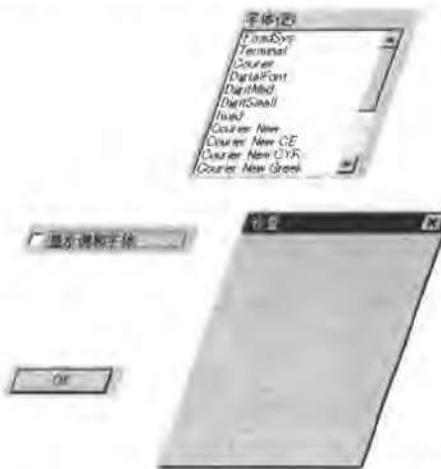


图 A14-1 对话窗口、列表窗口和复选框的关系图

## 问题 14-4 的解答

展开成循环的结果即如程序 A14-1 所示。

**程序 A14-1 改写过的 Support 类 (Support.java)**

```

public abstract class Support {
    private String name;                                // 问题解决者的名称
    private Support next;                               // 转送位置
    public Support(String name) {                      // 产生问题解决者
        this.name = name;
    }
    public Support setNext(Support next) { // 设定转送位置
        this.next = next;
        return next;
    }
    public final void support(Trouble trouble) {
        for (Support obj = this; true; obj = obj.next) {
            if (obj.resolve(trouble)) {
                obj.done(trouble);
                break;
            } else if (obj.next == null) {
                obj.fail(trouble);
                break;
            }
        }
    }
    public String toString() {                         // 打印字符串
        return "[" + name + "]";
    }
    protected abstract boolean resolve(Trouble trouble); // 解决的
// 方法
    protected void done(Trouble trouble) { // 已解决
        System.out.println(trouble + " is resolved by " + this + ".");
    }
    protected void fail(Trouble trouble) { // 尚未解决
        System.out.println(trouble + " cannot be resolved.");
    }
}

```

## 第 15 章

### 问题 15-1 的解答

如下将 Database 类(程序 15-1)和 HtmlWriter 类(程序 15-3)的定义改成无 public。这样就不能从 pagemaker 包以外的部分看到 Database 类和 HtmlWriter 类的名称(方法上的 public 可以不删除)。

● 修改前

**Database 类**

```
public class Database {
    ...
}
```

**HtmlWriter 类**

```
public class HtmlWriter {
    ...
}
```

● 修改后

```
class Database {
    ...
}
```

```
class HtmlWriter {
    ...
}
```

### 问题 15-2 的解答

解答如程序 A15-1 所示。

**程序 A15-1 修改过的 PageMaker 类 (PageMaker.java)**

```
package pagemaker;

import java.io.FileWriter;
import java.io.IOException;
import java.util.Properties;
import java.util.Enumeration;

public class PageMaker {
    private PageMaker() { // 不建立对象实例，所以声明 private
    }
    public static void makeWelcomePage(String mailaddr, String
filename) {
        try {
            Properties mailprop = Database.getProperties("maildata");
            String username = mailprop.getProperty(mailaddr);
        }
    }
}
```

## Java语言实现应用

```
    HtmlWriter writer = new HtmlWriter(new FileWriter(filename));
    writer.title("Welcome to " + username + "'s page!");
    writer.paragraph("欢迎来到" + username + "的网页。");
    writer.paragraph("等你来信喔! ");
    writer.mailto(mailaddr, username);
    writer.close();
    System.out.println(filename + " is created for " + mailaddr +
" (" + username + ")");
} catch (IOException e) {
    e.printStackTrace();
}
}

public static void makeLinkPage(String filename) {
    try {
        HtmlWriter writer = new HtmlWriter(new FileWriter
(filename));
        writer.title("Link page");
        Properties mailprop = Database.getProperties
("maildata");
        Enumeration en = mailprop.propertyNames();
        while (en.hasMoreElements()) {
            String mailaddr = (String)en.nextElement();
            String username = mailprop.getProperty(mailaddr,
"(unknown)");
            writer.mailto(mailaddr, username);
        }
        writer.close();
        System.out.println(filename + " is created.");
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
```

## 第 16 章

### 问题 16-1 的解答

将 LoginFrame 类（程序 16-6）的 userpassChanged 方法修改如下。

```
if (textPass.getText().length() > 0) {
    ...
    if (textUser.getText().length() >= 4 && textPass.getText().length()
        >= 4) {
        ...
    }
}
```

不须修改其他类。

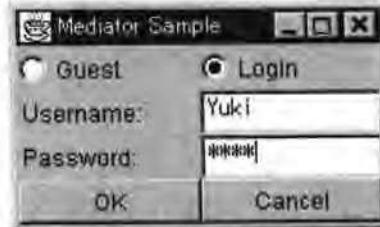
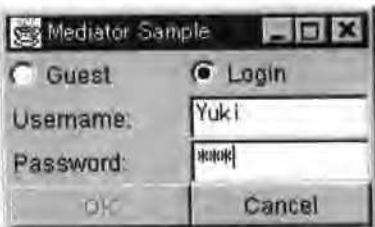


图 A16-1 若密码为 3 个字，则 OK 键还不能按下 图 A16-2 若密码为 4 个字，则 OK 键可按下

程序 A16-1 LoginFrame 类 (LoginFrame.java)

```
import java.awt.Frame;
import java.awt.Label;
import java.awt.Color;
import java.awt.CheckboxGroup;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class LoginFrame extends Frame implements ActionListener,
Mediator {
    private ColleagueCheckbox checkGuest;
    private ColleagueCheckbox checkLogin;
    private ColleagueTextField textUser;
    private ColleagueTextField textPass;
    private ColleagueButton buttonOk;
    private ColleagueButton buttonCancel;
```

## Java语言中的应用

```
// 构造函数
// 产生并配置好 Colleague 之后再输出到画面上
public LoginFrame(String title) {
    super(title);
    setBackground(Color.lightGray);
    // 使用页面编排，产生 4?的画面
    setLayout(new GridLayout(4, 2));
    // 产生 Colleague
    createColleagues();
    // 配置
    add(checkGuest);
    add(checkLogin);
    add(new Label("Username:"));
    add(textUser);
    add(new Label("Password:"));
    add(textPass);
    add(buttonOk);
    add(buttonCancel);
    // 有效 / 无效的初始设定
    colleagueChanged(checkGuest);
    // 输出到画面上
    pack();
    show();
}

// 产生 Colleague
public void createColleagues() {
    // 产生
    CheckboxGroup g = new CheckboxGroup();
    checkGuest = new ColleagueCheckbox("Guest", g, true);
    checkLogin = new ColleagueCheckbox("Login", g, false);
    textUser = new ColleagueTextField("", 10);
    textPass = new ColleagueTextField("", 10);
    textPass.setEchoChar('*');
    buttonOk = new ColleagueButton("OK");
    buttonCancel = new ColleagueButton("Cancel");
    // Mediator 的部分
    checkGuest.setMediator(this);
```

## 练习问题的解答

```
checkLogin.setMediator(this);
textUser.setMediator(this);
textPass.setMediator(this);
buttonOk.setMediator(this);
buttonCancel.setMediator(this);
// Listener 的部分
checkGuest.addItemListener(checkGuest);
checkLogin.addItemListener(checkLogin);
textUser.addTextListener(textUser);
textPass.addTextListener(textPass);
buttonOk.addActionListener(this);
buttonCancel.addActionListener(this);
}

// 根据 Colleague 的通知，判断各 Colleague 的有效 / 无效
public void colleagueChanged(Colleague c) {
    if (c == checkGuest || c == checkLogin) {
        if (checkGuest.getState()) { // Guest 模式
            textUser.setColleagueEnabled(false);
            textPass.setColleagueEnabled(false);
            buttonOk.setColleagueEnabled(true);
        } else { // Login 模式
            textUser.setColleagueEnabled(true);
            userpassChanged();
        }
    } else if (c == textUser || c == textPass) {
        userpassChanged();
    } else {
        System.out.println("colleagueChanged:unknown colleague
= " + c);
    }
}
// textUser 或 textPass 有变动
// 判断各 Colleague 的有效 / 无效
private void userpassChanged() {
    if (textUser.getText().length() > 0) {
        textPass.setColleagueEnabled(true);
    if (textUser.getText().length() >= 4 && textPass.getText()
```

## 观察者模式应用

```

        () .length() >= 4) {
            buttonOk.setColleagueEnabled(true);
        } else {
            buttonOk.setColleagueEnabled(false);
        }
    } else {
        textPass.setColleagueEnabled(false);
        buttonOk.setColleagueEnabled(false);
    }
}
}

public void actionPerformed(ActionEvent e) {
    System.out.println(" " + e);
    System.exit(0);
}
}

```

---

## 问题 16-2 的解答

无法实现。

因为接口不能有对象实例字段（对象实例变量-Instance variable），而且也无法实现具体方法（非抽象方法的方法）。

同时也无法将 Colleague 接口设为类，进行问题所指的共享作业。假设现在建立一个 Colleague 类，定义其子类为 ColleagueButton 类。如此一来，就不能把 ColleagueButton 类设为 Button 类的子类。因为 Java 规定父类只能有 1 个。

## 第 17 章

## 问题 17-1 的解答

解答如程序 A17-1 所示。

**程序 A17-1 逐一加计数值的 IncrementalNumberGenerator 类  
(IncrementalNumberGenerator.java)**

```

public class IncrementalNumberGenerator extends NumberGenerator {
    private int number;                                // 目前数值
    private int end;                                   // 结束数值(不含此值)
    private int inc;                                  // 增加若干

```

## 练习问题的解答

```

public IncrementalNumberGenerator(int start, int end, int inc)
{
    this.number = start;
    this.end = end;
    this.inc = inc;
}
public int getNumber() { // 取得数值
    return number;
}
public void execute() {
    while (number < end) {
        notifyObservers();
        number += inc;
    }
}

```

---

## 问题 17-2 的解答

利用 GUI 建立一个饼图的 ConcreteObserver 参与者（图 A17-1）。

这里虽然有 3 种 ConcreteObserver 参与者，但实际从 RandomNumberGenerator 调用出来的只有 FrameObserver（程序 A17-2）。

rameObserver 则调用 GraphText 和 GraphCanvas（委让）。

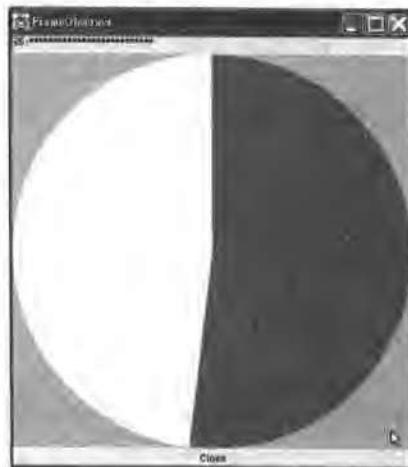


图 A17-1 执行的情形

## 程序 A17-2 利用 GUI 输出结果的 FrameObserver 类 (FrameObserver.java)

```
import java.awt.Frame;
import java.awt.TextField;
import java.awt.Canvas;
import java.awt.Color;
import java.awt.Button;
import java.awt.Graphics;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class FrameObserver extends Frame implements Observer,
ActionListener {

    private GraphText textGraph = new GraphText(60);
    private GraphCanvas canvasGraph = new GraphCanvas();
    private Button buttonClose = new Button("Close");

    public FrameObserver() {
        super("FrameObserver");
        setLayout(new BorderLayout());
        setBackground(Color.lightGray);
        textGraph.setEditable(false);
        canvasGraph.setSize(500, 500);
        add(textGraph, BorderLayout.NORTH);
        add(canvasGraph, BorderLayout.CENTER);
        add(buttonClose, BorderLayout.SOUTH);
        buttonClose.addActionListener(this);
        pack();
        show();
    }

    public void actionPerformed(ActionEvent e) {
        System.out.println(" " + e);
        System.exit(0);
    }

    public void update(NumberGenerator generator) {
        textGraph.update(generator);
        canvasGraph.update(generator);
    }
}
```

## 练习问题的解答

```

}

class GraphText extends TextField implements Observer {
    public GraphText(int columns) {
        super(columns);
    }
    public void update(NumberGenerator generator) {
        int number = generator.getNumber();
        String text = number + ":";
        for (int i = 0; i < number; i++) {
            text += '*';
        }
        setText(text);
    }
}
class GraphCanvas extends Canvas implements Observer {
    private int number;
    public void update(NumberGenerator generator) {
        number = generator.getNumber();
        repaint();
    }
    public void paint(Graphics g) {
        int width = getWidth();
        int height = getHeight();
        g.setColor(Color.white);
        g.fillArc(0, 0, width, height, 0, 360);
        g.setColor(Color.red);
        g.fillArc(0, 0, width, height, 90, - number * 360 / 50);
    }
}

```

**程序 A17-3 Main 类 (Main.java)**

```

public class Main {
    public static void main(String[] args) {
        NumberGenerator generator = new RandomNumberGenerator();
        Observer observer1 = new DigitObserver();
        Observer observer2 = new GraphObserver();
        Observer observer3 = new FrameObserver();
        generator.addObserver(observer1);

```

```

        generator.addObserver(observer2);
        generator.addObserver(observer3);
        generator.execute();
    }
}

```

## 第 18 章

### 问题 18-1 的解答

会丧失“Caretaker 参与者”和“Originator 参与者、Memento 参与者”之间的独立性。

如果 Caretaker 参与者能随意操作 Memento 参与者，则当 Originator 参与者内部有修改时，Caretaker 参与者也必须做同样的修改。

若 Caretaker 参与者只使用狭义接口(API)，则只要修改没有动到这个接口(API)，就能随意修改 Originator 参与者和 Memento 参与者。

### 问题 18-2 的解答

如果计算跟已存储的 Memento 之间的差距，或许就算空间不太大也还能存储。或者是做数据压缩，减少存储的数据量也是一种做法（数据压缩的示例在问题 18-4 的解答有稍微提到）。

### 问题 18-3 的解答

将 number 设为 private 的字段，再把取值的方法 getNumber 设成预设的存取控制（指无 public、protected、private）即可。

```

public class Memento {
    ...
    private int number;
    ...
    int getNumber() {
        return number;
    }
}

```

若为 Owner 类取得 number 的值，则使用 getNumber 方法。

## 问题 18-4 的解答

解答如程序 A18-1，程序 A18-2 所示。

### 程序 A18-1 支持串行化的 Memento 类 (Memento.java)

```
package game;
import java.io.*;
import java.util.Vector;

public class Memento implements Serializable {
    int money;                                // 手边金钱总额
    Vector fruits;                            // 水果
    public int getMoney() {           // 取得手边金钱总额 (narrow interface)
        return money;
    }
    Memento(int money) {           // 构造函数 (wide interface)
        this.money = money;
        this.fruits = new Vector();
    }
    void addFruit(String fruit) {      // 新增水果 (wide interface)
        fruits.add(fruit);
    }
}
```

### 程序 A18-2 支持串行化的 Main 类 (Main.java)

```
import game.Memento;
import game.Gamer;
import java.io.*;

public class Main {
    public static final String SAVEFILENAME = "game.dat";
    public static void main(String[] args) {
        Gamer gamer = new Gamer(100);    // 一开始手边金钱总额为 100
        Memento memento = loadMemento(); // 从文件载入
        if (memento != null) {
            System.out.println("从前回存储的结果开始。");
            gamer.restoreMemento(memento);
        } else {
```

## Java语言中的应用

```

        System.out.println("重新开始。");
        memento = gamer.createMemento();
    }
    for (int i = 0; i < 100; i++) {
        System.out.println("==== " + i); // 输出次数
        System.out.println("现况:" + gamer); // 输出主人翁的目前状态

        gamer.bet(); // 进行游戏

        System.out.println("手边金钱总额为" + gamer.getMoney() + "元。");

        // 决定如何处理 Memento
        if (gamer.getMoney() > memento.getMoney()) {
            System.out.println("（因为已经赢了不少，故先存储目前状态）");
        };
        memento = gamer.createMemento();
        saveMemento(memento); // 存储到文件
    } else if (gamer.getMoney() < memento.getMoney() / 2) {
        System.out.println("（因为已经输了很多，故恢复到前次状态）");
    };
    gamer.restoreMemento(memento);
}

// 等候
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
}
System.out.println("");
}
}

public static void saveMemento(Memento memento) {
try {
    ObjectOutputStream out =
new ObjectOutputStream(new FileOutputStream(SAVEFILENAME));
    out.writeObject(memento);
    out.close();
} catch (IOException e) {
}
}

```

## 练习问题的解答

```

        e.printStackTrace();
    }

}

public static Memento loadMemento() {
    Memento memento = null;
    try {
        ObjectInput in = new ObjectInputStream(new FileInputStream(SAVEFILENAME));
        memento = (Memento) in.readObject();
        in.close();
    } catch (FileNotFoundException e) {
        System.out.println("+" + e);
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    return memento;
}
}

```

---

将程序 A18-2 的 Main 类修改如下，即可压缩存储数据。尤其在所存储的数据量很大时，特别有效。

(1) 新增 import java.util.zip.\*;

(2) 输出部分含 DeflaterOutputStream

```

ObjectOutput out = new ObjectOutputStream(new FileOutputStream
(SAVEFILENAME));
↓
ObjectOutput out = new ObjectOutputStream(new DeflaterOutput
Stream(new
FileOutputStream(SAVEFILENAME)));

```

(3) 输入部分含 InflaterInputStream

```

ObjectInput in = new ObjectInputStream(new FileInputStream
(SAVEFILENAME));
↓
ObjectInput in = new ObjectInputStream(new InflaterInputStream(new
FileInputStream(SAVEFILENAME)));

```

## 第 19 章

### 问题 19-1 的解答

由于 Java 是单一继承，因此如果以类表示 Context 参与者时，就不能再把已经是 Frame 的子类的 SafeFrame 类设为 Context 类的子类。

应该要另外做一个 Context 类的子类，将其对象实例存储成 SafeFrame 的字段，再把处理委让给该对象实例。

### 问题 19-2 的解答

须修改 DayState 类（程序 19-4）和 NightState 类（程序 19-5）的 doClock 方法。

如果一开始先在 SafeFrame 类（程序 19-7）建立 isDay 方法和 isNight 方法，预先准备好检查目前时间是白天还是夜晚的方式，就能把具体的时间范围放到 SafeFrame 类之内。采取这样的措施之后，欲修改时间范围时可以只修改 SafeFrame 类就行。

### 问题 19-3 的解答

建立一个表示午餐时间状态的 NoonState 类（程序 A19-1），作为 ConcreteState 参与者。DayState 类和 NightState 类的 doClock 方法也都要修改（参见程序 A19-2，A19-3）。

程序 A19-1      NoonState 类（NoonState.java）

```
public class NoonState implements State {
    private static NoonState singleton = new NoonState();
    private NoonState() { // 构造函数为 private }
    public static State getInstance() { // 取得唯一的对象实例
        return singleton;
    }
    public void doClock(Context context, int hour) { // 设定时间
        if (hour < 9 || 17 <= hour) {
            context.changeState(NightState.getInstance());
        } else if (9 <= hour && hour < 12 || 13 <= hour && hour < 17)
        {
            context.changeState(DayState.getInstance());
        }
    }
}
```

## 练习问题的解答

```

    }

    public void doUse(Context context) { // 使用金库
        context.callSecurityCenter("异常：午餐时间使用金库！");
    }

    public void doAlarm(Context context) { // 警铃
        context.callSecurityCenter("警铃(午餐时间)");
    }

    public void doPhone(Context context) { // 一般通话
        context.recordLog("午餐时间的通话录音");
    }

    public String toString() { // 输出字符串
        return "[午餐时间]";
    }
}

```

---

## 程序 A19-2 DayState 类 (DayState.java)

```

public class DayState implements State {
    private static DayState singleton = new DayState();
    private DayState() { // 构造函数为 private
    }
    public static State getInstance() { // 取得唯一的对象实例
        return singleton;
    }
    public void doClock(Context context, int hour) { // 设定时间
        if (hour < 9 || 17 <= hour) {
            context.changeState(NightState.getInstance());
        } else if (12 <= hour && hour < 13) {
            context.changeState(NoonState.getInstance());
        }
    }
    public void doUse(Context context) { // 使用金库
        context.recordLog("使用金库(白天)");
    }
    public void doAlarm(Context context) { // 警铃
        context.callSecurityCenter("警铃(白天)");
    }
    public void doPhone(Context context) { // 一般通话
        context.callSecurityCenter("一般的通话(白天)");
    }
}

```

## Java语言特性的应用

```

    public String toString() { // 输出字符串
        return "[白天]";
    }
}

```

程序 A19-3 NightState 类 (NightState.java)

```

public class NightState implements State {
    private static NightState singleton = new NightState();
    private NightState() { // 构造函数为 private
    }
    public static State getInstance() { // 取得唯一的对象实例
        return singleton;
    }
    public void doClock(Context context, int hour) { // 设定时间
        if (9 <= hour && hour < 17) {
            context.changeState(DayState.getInstance());
        } else if (12 <= hour && hour < 13) {
            context.changeState(NoonState.getInstance());
        }
    }
    public void doUse(Context context) { // 使用金库
        context.callSecurityCenter("异常：晚间使用金库！");
    }
    public void doAlarm(Context context) { // 警铃
        context.callSecurityCenter("警铃(晚间)");
    }
    public void doPhone(Context context) { // 一般通话
        context.recordLog("晚间的通话录音");
    }
    public String toString() { // 输出字符串
        return "[晚间]";
    }
}

```

## 问题 19-4 的解答

加入表示异常的 UrgentState (紧急状况) 类 (程序 A19-4)。在 DayState 类和 NightState 类的 doAlarm 也要新增状态迁移的程序代码 (参见程序 A19-5, A19-6)。

## 练习问题的解答

这个新规格的问题在于没有在发生异常之后能恢复到原来状态的方式。

## 程序 A19-4 UrgentState 类 (UrgentState.java)

```
public class UrgentState implements State {
    private static UrgentState singleton = new UrgentState();
    private UrgentState() { // 构造函数为 private
    }
    public static State getInstance() { // 取得惟一的对象实例
        return singleton;
    }
    public void doClock(Context context, int hour) { // 设定时间
        // 设定时间的部分不做处理
    }
    public void doUse(Context context) { // 使用金库
        context.callSecurityCenter("异常：异常使用金库！");
    }
    public void doAlarm(Context context) { // 警铃
        context.callSecurityCenter("警铃(异常)");
    }
    public void doPhone(Context context) { // 一般通话
        context.callSecurityCenter("一般的通话(异常)");
    }
    public String toString() { // 输出字符串
        return "[异常]";
    }
}
```

## 程序 A19-5 DayState 类 (DayState.java)

```
public class DayState implements State {
    private static DayState singleton = new DayState();
    private DayState() { // 构造函数为 private
    }
    public static State getInstance() { // 取得惟一的对象实例
        return singleton;
    }
    public void doClock(Context context, int hour) { // 设定时间
        if (hour < 9 || 17 <= hour) {
```

## Java语言中的应用

```
        context.changeState(NightState.getInstance());
    }
}

public void doUse(Context context) // 使用金库
    context.recordLog("使用金库(白天)");
}

public void doAlarm(Context context) // 警铃
    context.callSecurityCenter("警铃(白天)");
    context.changeState(UrgentState.getInstance());
}

public void doPhone(Context context) // 一般通话
    context.callSecurityCenter("一般的通话(白天)");
}

public String toString() // 输出字符串
    return "[白天]";
}

}
```

### 程序 A19-6 NightState 类 (NightState.java)

```
public class NightState implements State {
    private static NightState singleton = new NightState();
    private NightState() { // 构造函数为 private
    }
    public static State getInstance() { // 取得唯一的对象实例
        return singleton;
    }
    public void doClock(Context context, int hour) { // 设定时间
        if (9 <= hour && hour < 17) {
            context.changeState(DayState.getInstance());
        }
    }
    public void doUse(Context context) // 使用金库
        context.callSecurityCenter("异常：晚间使用金库！");
    }
    public void doAlarm(Context context) // 警铃
        context.callSecurityCenter("警铃(晚间)");
        context.changeState(UrgentState.getInstance());
    }
}
```

```

public void doPhone(Context context) { // 一般通话
    context.recordLog("晚间的通话录音");
}
public String toString() { // 输出字符串
    return "[晚间]";
}
}

```

---

## 第 20 章

### 问题 20-1 的解答

若不共享 BigChar 时，则不使用 BigCharFactory 而 new BigChar。在程序 A20-1，为了提高程序的可读性，故建立了转包工作用的 private 的方法 init\_shared 和 init\_unshared。

**程序 A20-1 BigString 类 (BigString.java)**

```

public class BigString {
    // 大型文字的数组
    private BigChar[] bigchars;
    // 构造函数
    public BigString(String string) {
        init_shared(string);
    }
    // 构造函数
    public BigString(String string, boolean shared) {
        if (shared) {
            init_shared(string);
        } else {
            init_unshared(string);
        }
    }
    // 共享且初始化
    private void init_shared(String string) {
        bigchars = new BigChar[string.length()];
        BigCharFactory factory = BigCharFactory.getInstance();
        for (int i = 0; i < bigchars.length; i++) {
            bigchars[i] = factory.getBigChar(string.charAt(i));
        }
    }
}

```

## 如何共享字符串应用

```

        }
    }

    // 不共享的初始化
    private void init_unshared(String string) {
        bigchars = new BigChar[string.length()];
        for (int i = 0; i < bigchars.length; i++) {
            bigchars[i] = new BigChar(string.charAt(i));
        }
    }

    // 输出到画面上
    public void print() {
        for (int i = 0; i < bigchars.length; i++) {
            bigchars[i].print();
        }
    }
}

```

---

## 程序 A20-2 Main 类 (Main.java)

```

public class Main {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Usage: java Main digits");
            System.out.println("Example: java Main 1212123");
            System.exit(0);
        }
        BigString bs;
        bs = new BigString(args[0], false);      // 不共享
        bs.print();
        bs = new BigString(args[0], true);       // 共享
        bs.print();
    }
}

```

---

## 问题 20-2 的解答

程序 A20-3 的 Main 类是在数组上保留 1000 个对应"1212123"的 BigString 的对象实例，再比较两者的内存使用量。这样可以看出共享的内存使用量确实少很多。

而再执行程序的话，也能得知不共享的执行速度比较慢。这是因为如果不共享的

## 练习问题的解答

话，每次要产生 BigChar 的对象实例时都得读取文件的关系。

## 程序 A20-3 Main 类 (Main.java)

```
public class Main {
    private static BigString[] bsarray = new BigString[1000];
    public static void main(String[] args) {
        System.out.println("共享时：");
        testAllocation(true);
        System.out.println("不共享时：");
        testAllocation(false);
    }
    public static void testAllocation(boolean shared) {
        for (int i = 0; i < bsarray.length; i++) {
            bsarray[i] = new BigString("1212123", shared);
        }
        showMemory();
    }
    public static void showMemory() {
        Runtime.getRuntime().gc();
        long used = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
        System.out.println("内存使用量= " + used);
    }
}
```

共享时：

内存使用量 = 162176

不共享时：

内存使用量 = 2515032

图 A20-1 执行结果示例（其结果会因使用环境而异）

### 问题 20-3 的解答

若多个串行几乎同时调用这个方法时，可能会误判其对象实例是否已经产生而重复 new 的情形。

试以下面代码说明。

## JDK 的 synchronized 应用

程序 A20-4 无 synchronized 时（左边的数字编号系参考用行号）

```

1: public BigChar getBigChar(char charname) {
2:     BigChar bc = (BigChar)pool.get("") + charname);
3:     if (bc == null) {
4:         bc = new BigChar(charname);
5:         pool.put("") + charname, bc);
6:     }
7:     return bc;
8: }

```

假设串行 A 跟串行 B 以同样的 charname 调用 getBigChar，则会变成如图 A20-2 的结果。

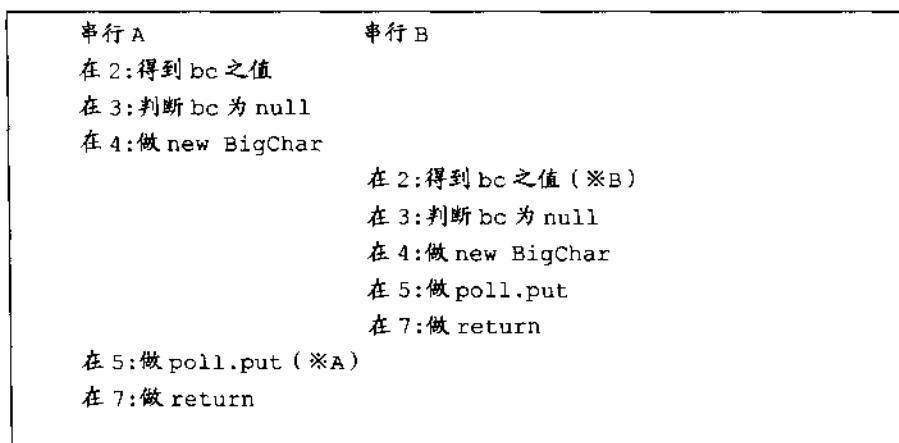


图 A20-2 如果没有 synchronized，可能会多出不必要的 new

此时，串行 A 和串行 B 两边都会 new BigChar。这是抢在（※A）之前就先执行了（※B）的时候所发生的现象。

必须设定在得到 bc 的值到 pool.put 之间不能有其他串行进来插队，才能避免发生这类现象。所以利用 synchronized 就能达到这个目的。

## 第 21 章

## 问题 21-1 的解答

解答如程序 A21-1, A21-2 所示。对象实例产生的部分为：

```
real = (Printable) Class.forName(className).newInstance();
```

## 练习问题的解答

另外，real 字段要设为 Printable 类型而非 Printer 类型。

这样处理之后，PrinterProxy 类就是独立在 Printer 类之外的零件，可以对所有实现 Printable 接口的类扮演 Proxy 参与者。

注意：如下所示，进行编译时除了 Main.java 之外，也要指定 Printer.java（因为 Printer 类没有直接使用）。

javac Main.java Printer.java

Class 类和 forName 方法的部分在 Abstract Factory (p.97) 已经有详细说明，如有需要请自行参阅。

**程序 A21-1 PrinterProxy 类 (PrinterProxy.java)**

```
public class PrinterProxy implements Printable {
    private String name;           // 名称
    private Printable real;        // “本人”
    private String className;      // “本人”的类名称
    public PrinterProxy(String name, String className) { // 构造函数
        this.name = name;
        this.className = className;
    }
    public synchronized void setPrinterName(String name) { // 命名
        if (real != null) {
            real.setPrinterName(name); // “本人”也要命名
        }
        this.name = name;
    }
    public String getPrinterName() { // 取得名称
        return name;
    }
    public void print(String string) { // 输出到画面上
        realize();
        real.print(string);
    }
    private synchronized void realize() { // 产生“本人”
        if (real == null) {
            try {
                real = (Printable) Class.forName(className).newInstance();
                real.setPrinterName(name);
            } catch (ClassNotFoundException e) {

```

## 设计模式的应试

```
        System.err.println("找不到类 " + className + ". ");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
```

## 程序 A21-2 Main 类 (Main.java)

```
public class Main {
    public static void main(String[] args) {
        Printable p = new PrinterProxy("Alice", "Printer");
        System.out.println("现在的名称是" + p.getPrinterName() + ".");
    };
    p.setPrinterName("Bob");
    System.out.println("现在的名称是" + p.getPrinterName() + ".");
    p.print("Hello, world.");
}
}
```

```
javac Main.java Printer.java
java Main
现在的名称是 Alice.
现在的名称是 Bob.
正在产生 Printer 的对象实例.....完成。
==== Bob ===
Hello, world.
```

图 A21-1 编译与执行结果

请仔细比较程序示例的执行结果（图 21-3）跟图 A21-1 的执行结果。图 A21-1 的“正在产生 Printer 的接口”这个位置并没有输出名称（Bob）。这是由于它要以 newInstance 方法产生对象实例，所以调用的是无 Printer 类参数的构造函数。

## 问题 21-2 的解答

如果不设为 synchronized 方法，当多个串行分别调用 setPrinterName 和 realize 时，PrinterProxy 类的 name 跟 Printer 类的 name 可能会有出入。

## 练习问题的解答

无 synchronized 的程序示例如程序 A21-3 所示。

程序 A21-3 无 synchronized 时（左边的编号数字与英文字母系参考用）

```

1: public void setPrinterName(String name) { // 命名
2:     if (real != null) {
3:         real.setPrinterName(name);           // “本人”也要命名
4:     }
5:     this.name = name;
6: }

...
a: private void realize() {                      // 产生“本人”
b:     if (real == null) {
c:         real = new Printer(name);
d:     }
e: }
```

| Name 的值 | real 的值 | 串行 A              | 串行 B                                                         |
|---------|---------|-------------------|--------------------------------------------------------------|
| "Alice" | null    |                   | 跑到 1:                                                        |
| "Alice" | null    |                   | 在 2: 判断 real 为 null                                          |
|         |         | -----切换为串行 B----- |                                                              |
| "Alice" | null    |                   | 跑到 a:                                                        |
|         |         |                   |                                                              |
| "Alice" | null    |                   | 在 b: 判断 real 为 null                                          |
| "Alice" | 非 null  |                   | 在 c: 将 Printer 的对象实例 ("Alice") 指定到<br>real                   |
|         |         | -----切换为串行 A----- |                                                              |
| "Bob"   | 非 null  |                   | 在 5: 将 "Bob" 指定到 name                                        |
| ※       |         |                   | 此时 PrinterProxy 类的 name 是 "Bob"，但 Printer 类的 name 却是 "Alice" |

图 A21-2 无 synchronized 时所发生的 name 相异

假设最初 PrinterProxy 的 name 字段的值为 "Alice"，real 字段的值为 null（即 Printer 类的对象实例尚未产生）。

而在串行 A 执行 setPrinterName("Bob") 的同时，串行 B 刚好也（以 print 方法）调用了 realize 方法。如果串行的切换方式是如图 A21-2，那么 PrinterProxy 类的 name 字段之值是 "Bob"，但 Printer 类的 name 字段的值却是 "Alice"。

把 setPrinterName 方法和 realize 方法设为 synchronized 方法，就不会发生类似这样的串行切换问题。因为设成 synchronized 方法之后，判断 real 字段之值跟修改该值的动作就不能个别进行。也就是说，这里利用 synchronized 方法来“捍卫” real 字段。

## 第 22 章

### 问题 22-1 的解答

新增的方法有很多种，这里采取的方法为：

- (1) 在 drawer 包新增一个表示“设定绘制颜色的命令”的 ColorCommand 类（程序 A22-1）
  - (2) 在 Drawable 接口新增一个“更改绘制颜色的方法” setColor（程序 A22-2）
  - (3) 新增后并实现 DrawCanvas 类（程序 A22-3）
  - (4) 在 Main 类新增可更改色彩“红”“绿”“蓝”的变色按键（程序 A22-4）
- command 包的类和接口都不须修改。



图 A22-1 在程序示例新增一个绘制颜色设定功能

#### 程序 A22-1 ColorCommand 类 (ColorCommand.java)

```
package drawer;

import command.Command;
import java.awt.Color;

public class ColorCommand implements Command {
    // 绘制对象
    protected Drawable drawable;
```

## 练习问题的解答

```
// 绘制颜色
private Color color;
// 构造函数
public ColorCommand(Drawable drawable, Color color) {
    this.drawable = drawable;
    this.color = color;
}
// 执行
public void execute() {
    drawable.setColor(color);
}
}
```

---

## 程序 A22-2 | Drawable 接口 (Drawable.java)

```
package drawer;

import java.awt.Color;

public interface Drawable {
    public abstract void init();
    public abstract void draw(int x, int y);
    public abstract void setColor(Color color);
}
```

---

## 程序 A22-3 | DrawCanvas 类 (DrawCanvas.java)

```
package drawer;

import command.*;
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DrawCanvas extends Canvas implements Drawable {
    // 绘制颜色
    private Color color;
```

## Java语言的运用

```

// 绘制点的半径
private int radius;
// 记录
private MacroCommand history;
// 构造函数
public DrawCanvas(int width, int height, MacroCommand history)
{
    setSize(width, height);
    setBackground(Color.white);
    this.history = history;
    init();
}
// 再度绘制整个记录
public void paint(Graphics g) {
    history.execute();
}
// 初始化
public void init() {
    color = Color.red;
    radius = 6;
}
// 绘制
public void draw(int x, int y) {
    Graphics g = getGraphics();
    g.setColor(color);
    g.fillOval(x - radius, y - radius, radius * 2, radius * 2);
}
public void setColor(Color color) {
    this.color = color;
}
}

```

---

## 程序 A22-4 Main 类 (Main.java)

```

import command.*;
import drawer.*;

import java.awt.*;
import java.awt.event.*;

```

## 练习问题的解答

```
import javax.swing.*;  
  
public class Main extends JFrame implements ActionListener,  
MouseMotionListener, WindowListener {  
    // 绘制记录  
    private MacroCommand history = new MacroCommand();  
    // 绘制区域  
    private DrawCanvas canvas = new DrawCanvas(400, 400, history);  
    // 删除键  
    private JButton clearButton = new JButton("clear");  
    // 红色键  
    private JButton redButton = new JButton("red");  
    // 绿色键  
    private JButton greenButton = new JButton("green");  
    // 蓝色键  
    private JButton blueButton = new JButton("blue");  
  
    // 构造函数  
    public Main(String title) {  
        super(title);  
  
        this.addWindowListener(this);  
        canvas.addMouseMotionListener(this);  
        clearButton.addActionListener(this);  
        redButton.addActionListener(this);  
        greenButton.addActionListener(this);  
        blueButton.addActionListener(this);  
  
        Box buttonBox = new Box(BoxLayout.X_AXIS);  
        buttonBox.add(clearButton);  
        buttonBox.add(redButton);  
        buttonBox.add(greenButton);  
        buttonBox.add(blueButton);  
        Box mainBox = new Box(BoxLayout.Y_AXIS);  
        mainBox.add(buttonBox);  
        mainBox.add(canvas);  
        getContentPane().add(mainBox);  
    }  
}
```

## Java语言中的应用

```
    pack();
    setVisible(true);
}

// ActionListener用
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == clearButton) {
        history.clear();
        canvas.init();
        canvas.repaint();
    } else if (e.getSource() == redButton) {
        Command cmd = new ColorCommand(canvas, Color.red);
        history.append(cmd);
        cmd.execute();
    } else if (e.getSource() == greenButton) {
        Command cmd = new ColorCommand(canvas, Color.green);
        history.append(cmd);
        cmd.execute();
    } else if (e.getSource() == blueButton) {
        Command cmd = new ColorCommand(canvas, Color.blue);
        history.append(cmd);
        cmd.execute();
    }
}

// MouseMotionListener用
public void mouseMoved(MouseEvent e) {
}
public void mouseDragged(MouseEvent e) {
    Command cmd = new DrawCommand(canvas, e.getPoint());
    history.append(cmd);
    cmd.execute();
}

// WindowListener用
public void windowClosing(WindowEvent e) {
    System.exit(0);
}
```

## 练习问题的解答

```

public void windowActivated(WindowEvent e) {}
public void windowClosed(WindowEvent e) {}
public void windowDeactivated(WindowEvent e) {}
public void windowDeiconified(WindowEvent e) {}
public void windowIconified(WindowEvent e) {}
public void windowOpened(WindowEvent e) {}

public static void main(String[] args) {
    new Main("Command Pattern Sample");
}

```

---

## 问题 22-2 的解答

将 Main 类修改如程序 A22-5。修改部分有以下 2 点。

- 新增复原键（undo）
- 按下复原键时，则调用 history.undo 重新绘制（repaint）  
command 包和 drawer 包都不需要修改。

使用复原功能的情形即如图 A22-2~图 A22-4 所示。

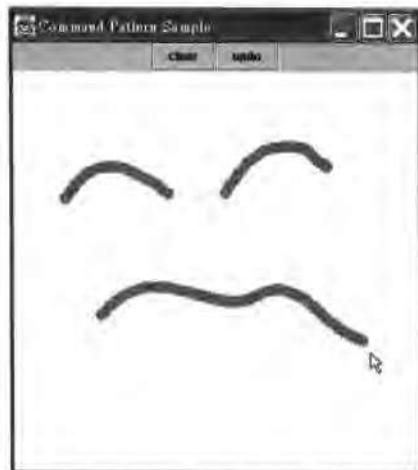


图 A22-2 觉得不好看，想复原



图 A22-3 按了几次 undo 键之后

## 设计模式 —— Java语言中的应用

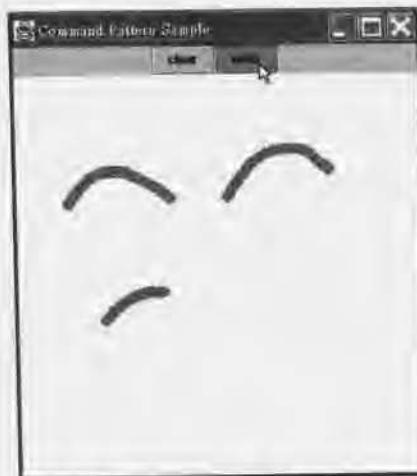


图 A22-4 再多按几次 Undo 键后

### 程序 A22-5 Main 类 (Main.java)

```
import command.*;
import drawer.*;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Main extends JFrame implements ActionListener,
MouseMotionListener, WindowListener {
    // 绘制记录
    private MacroCommand history = new MacroCommand();
    // 绘制区域
    private DrawCanvas canvas = new DrawCanvas(400, 400, history);
    // 删除键
    private JButton clearButton = new JButton("clear");
    // 复原键
    private JButton undoButton = new JButton("undo");

    // 构造函数
    public Main(String title) {
        super(title);

        this.addWindowListener(this);
```

## 练习问题的解答

```
canvas.addMouseMotionListener(this);
clearButton.addActionListener(this);
undoButton.addActionListener(this);

Box buttonBox = new Box(BoxLayout.X_AXIS);
buttonBox.add(clearButton);
buttonBox.add(undoButton);
Box mainBox = new Box(BoxLayout.Y_AXIS);
mainBox.add(buttonBox);
mainBox.add(canvas);
getContentPane().add(mainBox);

pack();
setVisible(true);
}

// ActionListener用
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == clearButton) {
        history.clear();
        canvas.repaint();
    } else if (e.getSource() == undoButton) {
        history.undo();
        canvas.repaint();
    }
}

// MouseMotionListener用
public void mouseMoved(MouseEvent e) {
}
public void mouseDragged(MouseEvent e) {
    Command cmd = new DrawCommand(canvas, e.getPoint());
    history.append(cmd);
    cmd.execute();
}

// WindowListener用
public void windowClosing(WindowEvent e) {
```

```

        System.exit(0);
    }

    public void windowActivated(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}

    public static void main(String[] args) {
        new Main("Command Pattern Sample");
    }
}

```

---

## 问题 22-3 的解答

解答如程序 A22-6 所示。

**程序 A22-6 Main 类 (Main.java)**

```

import command.*;
import drawer.*;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Main extends JFrame implements ActionListener {
    // 绘制记录
    private MacroCommand history = new MacroCommand();
    // 绘制区域
    private DrawCanvas canvas = new DrawCanvas(400, 400, history);
    // 删除键
    private JButton clearButton = new JButton("clear");

    // 构造函数
    public Main(String title) {
        super(title);

        this.addWindowListener(new WindowAdapter() {

```

## 练习问题的解答

```
public void windowClosing(WindowEvent e) {
    System.exit(0);
}

};

canvas.addMouseMotionListener(new MouseMotionAdapter() {
    public void mouseDragged(MouseEvent e) {
        Command cmd = new DrawCommand(canvas, e.getPoint());
        history.append(cmd);
        cmd.execute();
    }
});

clearButton.addActionListener(this);

Box buttonBox = new Box(BoxLayout.X_AXIS);
buttonBox.add(clearButton);
Box mainBox = new Box(BoxLayout.Y_AXIS);
mainBox.add(buttonBox);
mainBox.add(canvas);
getContentPane().add(mainBox);

pack();
setVisible(true);
}

// ActionListener用
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == clearButton) {
        history.clear();
        canvas.repaint();
    }
}

public static void main(String[] args) {
    new Main("Command Pattern Sample");
}
}
```

## 第 23 章

### 问题 23-1 的解答

这里是把跟 GUI 有关的部分放在 turtle 包，让 language 包里面没有 GUI。只要建立一个能在其他程序包里执行 Executor 和 ExecutorFactory 接口的类，不必修改 language 包就能另建一个“执行”相同程序的新程序。

表 A23-1 类与接口的一览表

| 程序包      | 名称                   | 说明                                                                |
|----------|----------------------|-------------------------------------------------------------------|
| language | InterpreterFacade    | 让解释器容易使用的类(Facade Pattern 的 Facade 参与者)                           |
| language | ExecutorFactory      | 产生基本命令的接口(Factory Method Pattern 的 Creator 参与者)                   |
| language | Content              | 同程序示例                                                             |
| language | Mode                 | 同程序示例                                                             |
| language | Executor             | 表现“执行”的接口                                                         |
| language | ProgramNode          | 同程序示例                                                             |
| language | CommandNode          | 同程序示例                                                             |
| language | RepeatCommandNode    | 同程序示例                                                             |
| language | CommandListNode      | 同程序示例                                                             |
| language | PrimitiveCommandNode | 同程序示例                                                             |
| language | ExecuteException     | 执行时的例外类                                                           |
| language | ParseException       | 文法分析时的例外类                                                         |
| turtle   | TurtleCanvas         | 实现 turtle canvas 的类(Factory Method Pattern 的 ConcreteCreator 参与者) |
| turtle   | TurtleExecutor       | (包内的类)                                                            |
| turtle   | GoExecutor           | (包内的类)                                                            |
| turtle   | DirectionExecutor    | (包内的类)                                                            |
| 未命名      | Main                 | 测试用的类                                                             |

程序 A23-1 InterpreterFacade 类(InterpreterFacade.java)

```
package language;

public class InterpreterFacade {
```

## 练习问题的解答

```
private ExecutorFactory factory;
private Context context;
private Node programNode;
public InterpreterFacade(ExecutorFactory factory) {
    this.factory = factory;
}
public boolean parse(String text) {
    boolean ok = true;
    this.context = new Context(text);
    this.context.setExecutorFactory(factory);
    this.programNode = new ProgramNode();
    try {
        programNode.parse(context);
        System.out.println("") + programNode);
    } catch (ParseException e) {
        e.printStackTrace();
        ok = false;
    }
    return ok;
}
public boolean execute() {
    boolean ok = true;
    try {
        programNode.execute();
    } catch (ExecuteException e) {
        e.printStackTrace();
        ok = false;
    }
    return ok;
}
```

## 程序 A23-2 | ExecutorFactory 接口 (ExecutorFactory.java)

```
package language;

public interface ExecutorFactory {
    public abstract Executor createExecutor(String name);
}
```

## 程序 A23-3 Context 类 (Context.java)

```
package language;

import java.util.*;

public class Context implements ExecutorFactory {
    private ExecutorFactory factory;
    private StringTokenizer tokenizer;
    private String currentToken;
    public Context(String text) {
        tokenizer = new StringTokenizer(text);
        nextToken();
    }
    public String nextToken() {
        if (tokenizer.hasMoreTokens()) {
            currentToken = tokenizer.nextToken();
        } else {
            currentToken = null;
        }
        return currentToken;
    }
    public String currentToken() {
        return currentToken;
    }
    public void skipToken(String token) throws ParseException {
        if (!token.equals(currentToken)) {
            throw new ParseException("Warning: " + token + " is
expected, but " + currentToken + " is found.");
        }
        nextToken();
    }
    public int currentNumber() throws ParseException {
        int number = 0;
        try {
            number = Integer.parseInt(currentToken);
        } catch (NumberFormatException e) {
            throw new ParseException("Warning: " + e);
        }
    }
}
```

## 练习问题的解答

```

        }
        return number;
    }

    public void setExecutorFactory(ExecutorFactory factory) {
        this.factory = factory;
    }

    public Executor createExecutor(String name) {
        return factory.createExecutor(name);
    }
}

```

---

**程序 A23-4 Node 类 (Node.java)**

```

package language;

public abstract class Node implements Executor {
    public abstract void parse(Context context) throws
ParseException;
}

```

---

**程序 A23-5 Executor 接口 (Executor.java)**

```

package language;

public interface Executor {
    public abstract void execute() throws ExecuteException;
}

```

---

**程序 A23-6 ProgramNode 类 (ProgramNode.java)**

```

package language;

// <program> ::= program <command list>
public class ProgramNode extends Node {
    private Node commandListNode;
    public void parse(Context context) throws ParseException {
        context.skipToken("program");
        commandListNode = new CommandListNode();
        commandListNode.parse(context);
    }
}

```

## Java语言基础应用

```

        }
        public void execute() throws ExecuteException {
            commandListNode.execute();
        }
        public String toString() {
            return "[program " + commandListNode + "]";
        }
    }
}

```

程序 A23-7 CommandNode 类 (CommandNode.java)

```

package language;

// <command> ::= <repeat command> | <primitive command>
public class CommandNode extends Node {
    private Node node;
    public void parse(Context context) throws ParseException {
        if (context.currentToken().equals("repeat")) {
            node = new RepeatCommandNode();
            node.parse(context);
        } else {
            node = new PrimitiveCommandNode();
            node.parse(context);
        }
    }
    public void execute() throws ExecuteException {
        node.execute();
    }
    public String toString() {
        return node.toString();
    }
}

```

程序 A23-8 RepeatCommandNode 类 (RepeatCommandNode.java)

```

package language;

// <repeat command> ::= repeat <number> <command list>
public class RepeatCommandNode extends Node {

```

## 练习问题的解答

```

private int number;
private Node commandListNode;
public void parse(Context context) throws ParseException {
    context.skipToken("repeat");
    number = context.currentNumber();
    context.nextToken();
    commandListNode = new CommandListNode();
    commandListNode.parse(context);
}
public void execute() throws ExecuteException {
    for (int i = 0; i < number; i++) {
        commandListNode.execute();
    }
}
public String toString() {
    return "[repeat " + number + " " + commandListNode + "]";
}
}

```

---

## 程序 A23-9 CommandListNode 类 (CommandListNode.java)

```

package language;

import java.util.*;

// <command list> ::= <command>* end
public class CommandListNode extends Node {
    private Vector list = new Vector();
    public void parse(Context context) throws ParseException {
        while (true) {
            if (context.currentToken() == null) {
                throw new ParseException("Missing 'end'");
            } else if (context.currentToken().equals("end")) {
                context.skipToken("end");
                break;
            } else {
                Node commandNode = new CommandNode();
                commandNode.parse(context);
                list.add(commandNode);
            }
        }
    }
}

```

```

        }
    }

    public void execute() throws ExecuteException {
        Iterator it = list.iterator();
        while (it.hasNext()) {
            ((CommandNode)it.next()).execute();
        }
    }

    public String toString() {
        return "" + list;
    }
}

```

---

程序 A23-10 PrimitiveCommandNode 类 (PrimitiveCommandNode.java)

```

package language;

// <primitive command> ::= go | right | left
public class PrimitiveCommandNode extends Node {
    private String name;
    private Executor executor;
    public void parse(Context context) throws ParseException {
        name = context.currentToken();
        context.skipToken(name);
        executor = context.createExecutor(name);
    }
    public void execute() throws ExecuteException {
        if (executor == null) {
            throw new ExecuteException(name + ": is not defined");
        } else {
            executor.execute();
        }
    }
    public String toString() {
        return name;
    }
}

```

---

## 练习问题的解答

**程序 A23-11 ExecuteException 类 (ExecuteException.java)**

```
package language;

public class ExecuteException extends Exception {
    public ExecuteException(String msg) {
        super(msg);
    }
}
```

**程序 A23-12 ParseException 类 (ParseException.java)**

```
package language;

public class ParseException extends Exception {
    public ParseException(String msg) {
        super(msg);
    }
}
```

**程序 A23-13 TurtleCanvas 类 (TurtleCanvas.java)**

```
package turtle;

import language.Executor;
import language.ExecutorFactory;
import java.awt.*;

public class TurtleCanvas extends Canvas implements ExecutorFactory
{
    final static int UNIT_LENGTH = 30;      // 移动时的长度单位
    final static int DIRECTION_UP = 0;       // 往上
    final static int DIRECTION_RIGHT = 3;   // 往右
    final static int DIRECTION_DOWN = 6;    // 往下
    final static int DIRECTION_LEFT = 9;    // 往左
    final static int RELATIVE_DIRECTION_RIGHT = 3; // 往右
    final static int RELATIVE_DIRECTION_LEFT = -3; // 往左
    final static int RADIUS = 3; // 半径
```

## Java繪圖技術應用

```
private int direction = 0;
private Point position;
public TurtleCanvas(int width, int height) {
    setSize(width, height);
    initialize();
}
void setRelativeDirection(int relativeDirection) {
    setDirection(direction + relativeDirection);
}
void setDirection(int direction) {
    if (direction < 0) {
        direction = 12 - (-direction) % 12;
    } else {
        direction = direction % 12;
    }
    this.direction = direction % 12;
}
void go(int length) {
    int newx = position.x;
    int newy = position.y;
    switch (direction) {
    case DIRECTION_UP:
        newy -= length;
        break;
    case DIRECTION_RIGHT:
        newx += length;
        break;

    case DIRECTION_DOWN:
        newy += length;
        break;
    case DIRECTION_LEFT:
        newx -= length;
        break;
    default:
        break;
    }
    Graphics g = getGraphics();
```

## 练习问题的解答

```
if (g != null) {
    g.drawLine(position.x, position.y, newx, newy);
    g.fillOval(newx - RADIUS, newy - RADIUS, RADIUS * 2 + 1,
RADIUS * 2 + 1);
}
position.x = newx;
position.y = newy;
}

public Executor createExecutor(String name) {
    if (name.equals("go")) {
        return new GoExecutor(this);
    } else if (name.equals("right")) {
        return new DirectionExecutor(this, RELATIVE_DIRECTION_
RIGHT);
    } else if (name.equals("left")) {
        return new DirectionExecutor(this, RELATIVE_DIRECTION_
LEFT);
    } else {
        return null;
    }
}

public void initialize() {
    Dimension size = getSize();
    position = new Point(size.width / 2, size.height / 2);
    direction = 0;
    setForeground(Color.red);
    setBackground(Color.white);
    Graphics g = getGraphics();
    if (g != null) {
        g.clearRect(0, 0, size.width, size.height);
    }
}

abstract class TurtleExecutor implements Executor {
    protected TurtleCanvas canvas;
    public TurtleExecutor(TurtleCanvas canvas) {
        this.canvas = canvas;
```

## 命令模式的应用

```

    }
    public abstract void execute();
}

class GoExecutor extends TurtleExecutor {
    public GoExecutor(TurtleCanvas canvas) {
        super(canvas);
    }
    public void execute() {
        canvas.go(TurtleCanvas.UNIT_LENGTH);
    }
}

class DirectionExecutor extends TurtleExecutor {
    private int relativeDirection;
    public DirectionExecutor(TurtleCanvas canvas, int relative
Direction) {
        super(canvas);
        this.relativeDirection = relativeDirection;
    }
    public void execute() {
        canvas.setRelativeDirection(relativeDirection);
    }
}

```

## 程序 A23-14 Main 类 (Main.java)

```

import language.InterpreterFacade;
import turtle.TurtleCanvas;

import java.util.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.*;

public class Main extends Frame implements ActionListener {
    private TurtleCanvas canvas = new TurtleCanvas(400, 400);

```

## 练习问题的解答

```
private InterpreterFacade facade = new InterpreterFacade(canvas);
private TextField programTextField = new TextField("program
repeat 3 go right go left end end");

// 构造函数
public Main(String title) {
    super(title);

    setLayout(new BorderLayout());
    programTextField.addActionListener(this);

    this.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
}

add(programTextField, BorderLayout.NORTH);
add(canvas, BorderLayout.CENTER);
pack();
setVisible(true);
parseAndExecute();
}

// ActionListener用
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == programTextField) {
        parseAndExecute();
    }
}

private void parseAndExecute() {
    String programText = programTextField.getText();
    System.out.println("programText = " + programText);
    facade.parse(programText);
    repaint();
}
```

## Interpreter 模式

### Interpreter 模式的应用

```
}

public void paint(Graphics g) {
    canvas.initialize();
    facade.execute();
}

public static void main(String[] args) {
    new Main("Interpreter Pattern Sample");
}
}
```

---

# 附录 B —

## GoF 的设计 Pattern 分类

GoF 一书中的设计 Pattern 分类方式如下, ( )里是该设计 Pattern 在本书的章节位置。

## 创建模式 (Creational Patterns)

|                          |                   |
|--------------------------|-------------------|
| Abstract Factory (第 8 章) | Builder (第 7 章)   |
| Factory Method (第 4 章)   | Prototype (第 6 章) |
| Singleton (第 5 章)        |                   |

## 结构模式 (Structural Patterns)

|                    |                    |
|--------------------|--------------------|
| Adapter (第 2 章)    | Bridge (第 9 章)     |
| Composite (第 11 章) | Decorator (第 12 章) |
| Façade (第 15 章)    | Flyweight (第 20 章) |
| Proxy (第 21 章)     |                    |

## 行为模式 (Behavioral Pattern)

|                                  |                         |
|----------------------------------|-------------------------|
| Chain of Responsibility (第 14 章) | Command (第 22 章)        |
| Interpreter (第 23 章)             | Iterator (第 1 章)        |
| Mediator (第 16 章)                | Memento (第 18 章)        |
| Observer (第 17 章)                | State (第 19 章)          |
| Strategy (第 10 章)                | Template Method (第 3 章) |
| Visitor (第 13 章)                 |                         |