TURING 图灵程序设计丛书 移动开发系列

Apress®

- 解析iOS设计模式的开山之作
- 优化Objective-C编程实践的必修宝典
- 由此迈入移动开发高手行列

Pro Objective-C Design Patterns for iOS

Objective-C编程之道

[美] Carlo Chung 著
刘威 译

人民邮电出版社
POSTS & TELECOM PRESS

“每学习一门新的编程语言，我都会去买一本介绍其设计模式的书。这些书从来没有让我失望过。从长远来看，我获得的回报十倍于我的付出，本书也不例外。作者首先介绍每一种设计模式的理论和方法，然后给出代码示例。简单地讲，本书结构清晰，易于理解，物超所值。”

——亚马逊读者评论

“这是一本启人深思的书。在学习如何将设计模式应用于复杂的iOS应用的同时，我开始静下来思考怎样优化既有代码。向每一位中高级iOS开发人员推荐本书。”

——亚马逊读者评论

Pro Objective-C Design Patterns for iOS

Objective-C编程之道

iOS设计模式解析

苹果公司的App Store拥有超过20万个应用（每秒都在增加）供用户选择，正深刻地改变着软件行业。每天都有更多的iOS开发者想投入到这一潮流，希望凭藉下一个杀手级应用发家致富。本书的目标正是带领读者完成从新手到高手的转变，关注底层的设计模式而非一味只顾着写代码，从而开发出更加高效、实用和专业的应用。

iOS应用程序的基础Cocoa Touch框架内容丰富、结构优美，通过将各种设计模式应用到其基础结构中，为第三方开发者提供了很好的可扩展性和灵活性。因此，要充分利用这一框架，应当深刻理解并恰当应用设计模式。本书受到GoF的经典著作《设计模式》的启发，旨在引导大家掌握如何在iOS平台上以Objective-C语言实现Cocoa Touch开发所要用到的传统设计模式。

在编写代码的过程中，你可能在一定程度上运用了一些设计模式，只是并没有意识到或充分利用它们。基于此，本书深入解析了这些设计模式。特定模式方法的实现将向iOS应用开发人员展示其非凡价值。你将掌握单例、抽象工厂、责任链和观察者等经典模式，还会发现一些不太知名但非常有用的模式，比如备忘录、组合、命令和中介者等。

学完本书，你将学会：

- 各种设计模式的基本概念；
- 根据不同场景，将设计模式应用于代码中；
- 用设计模式来改进应用程序；
- 提高软件开发的效率。

Apress®

图灵社区：www.ituring.com.cn
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

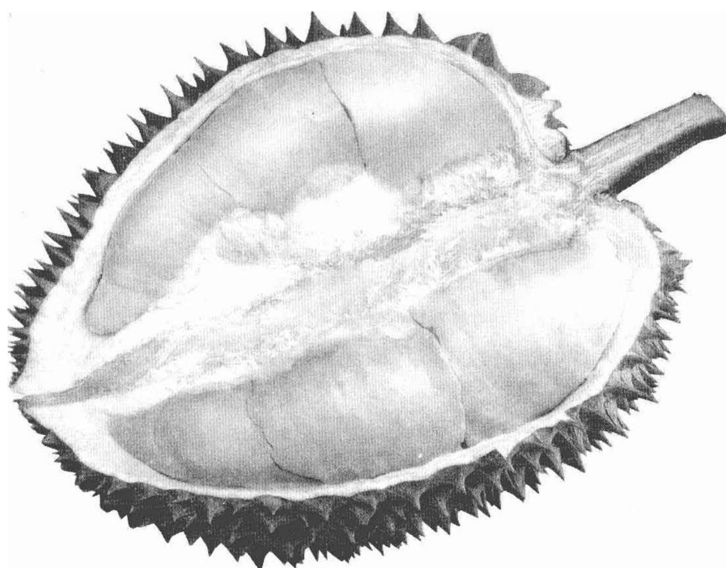
分类建议 计算机/程序设计/移动开发

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-26586-9
定价：59.00元

TURING 图灵程序设计丛书 移动开发系列

ITSRCS.COM
登录获取更多编程好书...



Pro Objective-C Design Patterns for iOS

Objective-C编程之道

iOS设计模式解析

[美] Carlo Chung 著
刘威 译

人民邮电出版社
北京

图书在版编目（C I P）数据

Objective-C编程之道 : iOS设计模式解析 / (美)
钟冠贤著 ; 刘威译. -- 北京 : 人民邮电出版社,
2011.11
（图灵程序设计丛书）
书名原文: Pro Objective-C Design Patterns for
iOS
ISBN 978-7-115-26586-9

Ⅰ. ①O… Ⅱ. ①钟… ②刘… Ⅲ. ①C语言—程序设
计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2011)第209817号

内 容 提 要

本书是基于 iOS 的软件开发指南。书中应用 GoF 的经典设计模式，介绍了如何在代码中应用创建型模式、结构型模式和行为模式，如何设计模式以巩固应用程序，并通过设计模式实例介绍 MVC 在 Cocoa Touch 框架中的工作方式。

本书适用于那些已经具备 Objective-C 基础、想利用设计模式来提高软件开发效率的中高级 iOS 开发人员。

图灵程序设计丛书

Objective-C编程之道：iOS设计模式解析

◆ 著　[美] Carlo Chung
译　刘 威
责任编辑　傅志红
执行编辑　罗词亮

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：19.25
字数：455千字　2011年11月第1版
印数：1－4 000册　2011年11月北京第1次印刷

著作权合同登记号　图字：01-2011-2955 号

ISBN 978-7-115-26586-9

定价：59.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版 权 声 明

Original English language edition, entitled *Pro Objective-C Design Patterns for iOS* by Carlo Chung, published by Apress, 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705 USA.

Copyright © 2011 by Carlo Chung. Simplified Chinese-language edition copyright © 2011 by Posts & Telecom Press. All rights reserved.

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

版权所有，侵权必究。

致　谢

“你知道吗，这本书真的很难写。”这是我和Apress出版社的一位策划编辑第一次通电话时他对我的提醒。出于他对我的信任，我在2010年夏天开始了这次写作历程。

感谢协调编辑Corbin Collins，他让整个团队保持专注。感谢项目编辑Douglas Pundick，感谢他在编辑审查阶段给我的所有提升本书的建议。感谢策划编辑Michelle Lowman的支持和耐心。感谢技术编审James Bucanek直到印刷之前最后一刻还提出了一些批评意见。感谢文字编辑Mary Ann Fugate和Mary Behr，是她们完美的文字编辑工作让这本书精彩纷呈。感谢所有了不起的Apress人——没有他们的帮助就不可能有这本书。

我要感谢Michael Fredrickson和Sreenivasa Busam，他们以其专业的视角对本书作了审阅。最后，非常感谢Mike Hambleton，感谢他审阅书稿时所体现的技术写作专业性以及细致和耐心的态度。

前　言

拥有超过20万个应用(且每秒都在增加)可供用户随意下载，苹果公司的应用商店(App Store)影响着各行各业。苹果公司广告语“There's an app for that”（总有一款应用可以做这个）的效应绝对不容忽视。不管你信不信，在这4个月里，我是坐在沙发上用iPad完成了本书的大部分内容。

每天都有更多的iOS开发者想跟随潮流，用下一个杀手级应用发家致富。截至本书写作时，全世界已有超过5万名iOS开发者，而且这一数字还在迅速增长。如果你真的有意从事iOS开发，并想通过好的软件设计原则让开发工作更加高效，那么你应该读这本书。

身为iOS开发者，我了解开发应用程序的痛苦与收获。学习新的编程语言绝非易事。最终，我们学会了并开始开发应用程序。在考虑Cocoa Touch框架时，即使是经验丰富的开发者也很容易被它优美的设计与结构所打动。这种优美来自设计者的深思熟虑，通过把各种为人熟知（或不为人知）的设计模式应用到框架的各种基础结构之中，他们为你我这样的第三方开发者提供了很好的可扩展性与灵活性。框架的大部分在一遍又一遍地重用相同的模式，后来添加到框架的新元素可以很容易地被其他应用程序开发者理解，不用再经过一次艰难的学习。

理解Cocoa Touch框架中使用的模式只是第一步。如果不花时间进行项目的设计，有了杀手级应用的想法马上就开始编码，那么随着后来为它添加更多的功能，它就会变成一个巨大的“樟脑球”。这样，最糟糕的情况是，它变得无法管理，而你（或团队中其他开发者）则根本无法理解你的代码。最终你将花更多的时间去修改代码缺陷，而不能专注于新的改进。

要充分利用Cocoa Touch框架，应该对设计模式有深刻理解，并在实现中进行恰当应用。本书受到Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides（常被称作GoF）的经典著作*Design Patterns:Elements of Reusable Object-Oriented Software*[①] 的启发。本书的每一章，模式的详细讲解中都引用了该书的定义。

今天，我们理解了在很多软件项目中使用设计模式的重要性。Cocoa Touch框架的大部分是用Objective-C写的，本书写作时，市面上还没有讲解如何用Objective-C语言实现设计模式的书。本书旨在成为该主题的第一本权威指南，引导大家了解该如何在iOS平台上以Objective-C语言实现Cocoa Touch开发所要用到的传统设计模式。

作为iOS开发者，你肯定想让开发更容易，是吧？如果不了解应用程序软件设计的最佳做法，开发过程将会困难重重，甚至最终毫无结果。而且，总的来说，想要跟上App Store或软件市场快

---

① 以下简称《设计模式》。其中文版《设计模式：可复用面向对象软件的基础》于2004年由机械工业出版社出版。
——编者注

节奏的变化，重用那些开发好并经过验证的模块非常关键。

用一点儿耐心来理解本书的内容，尽量在实际项目中应用这些模式，读者很快就能体验到设计模式的好处。

我为本书建了一个网站，www.objective-c-design-patterns.com。上面有与本书相关的一些其他信息。也欢迎大家登录网站，分享在项目中使用设计模式的成功事例以及遇到的困难。本书的源代码可从www.apress.com[1]下载。

祝编码愉快！

Carlo Chung

钟冠贤

2011年3月

① 读者也可以在图灵社区（ituring.com.cn）的本书页面下载。——编者注

目　录

第一部分　设计模式初体验

第1章　你好，设计模式……2

1.1 这是一本什么书……2
1.2 开始前的准备……2
1.3 预备知识……3
1.4 似曾相识的设计……3
1.5 设计模式的起源——模型、视图和控制器……4
1.5.1 在模型对象中封装数据和基本行为……4
1.5.2 使用视图对象向用户展示信息……5
1.5.3 用控制器对象联系起模型和视图……5
1.5.4 作为复合设计模式的MVC……5
1.6 影响设计的几个问题……6
1.6.1 针对接口编程，而不是针对实现编程……7
1.6.2 @protocol与抽象基类……7
1.6.3 对象组合与类继承……8
1.7 本书用到的对象和类……9
1.7.1 类图……9
1.7.2 对象图……12
1.8 本书如何安排模式的讲解……13
1.9 总结……13

第2章　案例分析：设计一个应用程序……14

2.1 想法的概念化……14
2.2 界面外观的设计……15
2.3 架构设计……17
2.3.1 视图管理……18
2.3.2 如何表现涂鸦……20
2.3.3 如何表现保存的涂鸦图……24
2.3.4 用户操作……27
2.4 所用设计模式的回顾……32
2.5 总结……33

第二部分　对象创建

第3章　原型……36

3.1 何为原型模式……36
3.2 何时使用原型模式……37
3.3 浅复制与深复制……38
3.4 使用Cocoa Touch框架中的对象复制……39
3.5 为Mark聚合体实现复制方法……39
3.6 将复制的Mark用作“图样模板”……47
3.7 总结……49

第4章　工厂方法……50

4.1 何为工厂方法模式……50
4.2 何时使用工厂方法……51
4.3 为何这是创建对象的安全方法……51
4.4 在TouchPainter中生成不同画布……51
4.5 在Cocoa Touch框架中应用工厂方法……57
4.6 总结……58

第5章　抽象工厂……59

5.1 把抽象工厂应用到TouchPainter应用程序……60
5.2 在Cocoa Touch框架中使用抽象工厂……66
5.3 总结……68

第 6 章 生成器……69
6.1 何为生成器模式……69
6.2 何时使用生成器模式……70
6.3 构建追逐游戏中的角色……71
6.4 总结……79
第 7 章 单例……80
7.1 何为单例模式……80
7.2 何时使用单例模式……81
7.3 在 Objective-C 中实现单例模式……81
7.4 子类化 Singleton……85
7.5 线程安全……85
7.6 在 Cocoa Touch 框架中使用单例模式……86
7.6.1 使用 UIApplication 类……86
7.6.2 使用 UIAccelerometer 类……86
7.6.3 使用 NSFileManager 类……86
7.7 总结……87

第三部分 接口适配

第 8 章 适配器……90
8.1 何为适配器模式……90
8.2 何时使用适配器模式……92
8.3 委托……92
8.4 用 Objective-C 协议实现适配器模式……93
8.5 用 Objective-C 的块在 iOS 4 中实现适配器模式……99
8.5.1 块引用的声明……99
8.5.2 块的创建……100
8.5.3 把块用作适配器……100
8.6 总结……104
第 9 章 桥接……105
9.1 何为桥接模式……105
9.2 何时使用桥接模式……106
9.3 创建 iOS 版虚拟仿真器……106
9.4 总结……112
第 10 章 外观……113
10.1 何为外观模式……113
10.2 何时使用外观模式……114
10.3 为子系统的一组接口提供简化的接口……114
10.4 在 TouchPainter 应用程序中使用外观模式……117
10.5 总结……119

第四部分 对象去耦

第 11 章 中介者……122
11.1 何为中介者模式……122
11.2 何时使用中介者模式……124
11.3 管理 TouchPainter 应用程序中的视图迁移……124
11.3.1 修改迁移逻辑的困难……126
11.3.2 集中管理 UI 交通……127
11.3.3 在 Interface Builder 中使用 CoordinatingController……132
11.4 总结……135
第 12 章 观察者……136
12.1 何为观察者模式……136
12.2 何时使用观察者模式……138
12.3 在模型-视图-控制器中使用观察者模式……138
12.4 在 Cocoa Touch 框架中使用观察者模式……138
12.4.1 通知……139
12.4.2 键-值观察……139
12.5 在 TouchPainter 中更新 CanvasView 上的线条……140
12.6 总结……149

第五部分 抽象集合

第 13 章 组合……152
13.1 何为组合模式……152
13.2 何时使用组合模式……154
13.3 理解 TouchPainter 中 Mark 的使用……154
13.4 在 Cocoa Touch 框架中使用组合模式……163
13.5 总结……164

第 14 章　迭代器……165
14.1　何为迭代器模式……165
14.2　何时使用迭代器模式……167
14.3　在 Cocoa Touch 框架中使用迭代器模式……167
14.3.1　NSEnumerator……167
14.3.2　基于块的枚举……168
14.3.3　快速枚举……169
14.3.4　内部枚举……170
14.4　遍历 Scribble 的顶点……170
14.5　总结……178

第六部分　行为扩展

第 15 章　访问者……180
15.1　何为访问者模式……180
15.2　何时使用访问者模式……182
15.3　用访问者绘制 TouchPainter 中的 Mark……182
15.4　访问者的其他用途……189
15.5　能不能用范畴代替访问者模式……189
15.6　总结……189
第 16 章　装饰……190
16.1　何为装饰模式……190
16.2　何时使用装饰模式……191
16.3　改变对象的“外表”和“内容”……192
16.4　为 UIImage 创建图像滤镜……192
16.4.1　通过真正的子类实现装饰……193
16.4.2　通过范畴实现装饰……201
16.5　总结……206
第 17 章　责任链……207
17.1　何为责任链模式……207
17.2　何时使用责任链模式……208
17.3　在 RPG 游戏中使用责任链模式……209
17.4　总结……214

第七部分　算法封装

第 18 章　模板方法……216
18.1　何为模板方法模式……216
18.2　何时使用模板方法……217
18.3　利用模板方法制作三明治……217
18.4　保证模板方法正常工作……224
18.5　向模板方法增加额外的步骤……225
18.6　在 Cocoa Touch 框架中使用模板方法……228
18.6.1　UIView 类中的定制绘图……228
18.6.2　Cocoa Touch 框架中的其他模板方法实现……228
18.7　总结……229
第 19 章　策略……230
19.1　何为策略模式……230
19.2　何时使用策略模式……231
19.3　在 UITextField 中应用验证策略……231
19.4　总结……239
第 20 章　命令……240
20.1　何为命令模式……240
20.2　何时使用命令模式……241
20.3　在 Cocoa Touch 框架中使用命令模式……241
20.3.1　NSInvocation 对象……242
20.3.2　NSUndoManager……242
20.4　在 TouchPainter 中实现撤销与恢复……243
20.4.1　使用 NSUndoManager 实现绘图与撤销绘图……244
20.4.2　自制绘图与撤销绘图的基础设施……248
20.4.3　允许用户触发撤销与恢复……255
20.5　命令还能做什么……256
20.6　总结……257

第八部分　性能与对象访问

第 21 章　享元……260
21.1　何为享元模式……260

21.2 何时使用享元模式 ……262
21.3 创建百花池 ……262
21.4 总结 ……269

第 22 章 代理 ……270

22.1 何为代理模式 ……270
22.2 何时使用代理模式 ……271
22.3 用虚拟代理懒加载图像 ……272
22.4 在 Cocoa Touch 框架中使用代理模式 ……277
22.5 总结 ……279

第九部分 对象状态

第 23 章 备忘录 ……282

23.1 何为备忘录模式 ……282
23.2 何时使用备忘录模式 ……283
23.3 在 TouchPainter 中使用备忘录模式 ……284
23.3.1 涂鸦图的保存 ……284
23.3.2 涂鸦图的恢复 ……285
23.3.3 ScribbleMemento 的设计与实现 ……286
23.4 Cocoa Touch 框架中的备忘录模式 ……295
23.5 总结 ……297

Part 1 第一部分

设计模式初体验

本部分内容

- 第 1 章　你好，设计模式
- 第 2 章　案例分析：设计一个应用程序

第 1 章

你好，设计模式

几乎每一本讲计算机编程的书开篇都是以“你好，世界！”（Hello, World!）一章来介绍主题。由于这是一本关于设计模式的书，我们就从“你好，设计模式”开始吧。

既然你已经拿起了这本书，对于面向对象编程中设计模式的概念，你也许已经熟悉。设计模式是有用的抽象化工具，用于解决工程和建筑等其他领域的设计问题。出于同样的目的，软件开发领域借用了这一概念。设计模式是一个对象或类的设计模板，用于解决特定领域经常发生的问题。

本章介绍设计模式的简史，以及设计模式与Cocoa Touch技术之间的关系。不仅讨论了影响设计的若干问题，还介绍了本书用到的对象图示法，以及本书如何安排设计模式。

1.1　这是一本什么书

本书写给想通过使用设计模式以使软件开发更高效、更有趣的专业人员和有抱负的iOS开发人员。本书的目的在于，展示如何将设计模式在iOS的应用开发中付诸实践。我会集中讨论各种设计模式对于Cocoa Touch框架及其相关技术的适用性。

尽管本书介绍的有些原则和概念可能也适用于Cocoa Touch的老大哥——Mac OS X的Cocoa，但是不保证它们完全适用于完整版的Cocoa。另外，本书还可以用做Objective-C设计模式的快速参考手册。

你将会学到：

- 各种设计模式的基本概念；
- 在各种设计场景，如何将设计模式应用到代码中；
- 设计模式如何增强应用程序。

本书的网站是www.objective-c-design-patterns.com或www.objectivecdesignpatterns.com。欢迎分享你在项目中使用设计模式的成功事例以及遇到的困难。

书中的源代码可从www.apress.com下载。

1.2　开始前的准备

同其他讲iOS应用程序开发的书一样，你需要Xcode和iOS SDK来运行本书中的示例代码。本

书的示例项目的编译和测试使用的是Xcode 3.2.5和iOS SDK 4.2，应该也可以使用更高版本。有许多免费或付费的非Mac OS X平台的工具可以开发iOS应用程序，但是不保证本书中的示例程序在这些平台也能正常运行。

本书也涉及了iOS应用程序设计的一些内容。你可能需要下载或购买一些软件工具来帮助构建线框图和UI布局，用以练习或设计实际的应用程序。本书中的线框图是用OmniGraffle（www.omnigroup.com/products/omnigraffle/）制作的。该网站还提供了各种免费模板（stencil）和线框模板的下载。

本书中大量使用了类和对象图，但是可用的类和对象的建模工具软件却很少。本书写作的时候，多数建模软件基于标准图示法，而这些标准图示法只适用于C++、Java和C#等其他面向对象编程语言。Objective-C有一些特殊功能，比如范畴（category）和扩展（匿名范畴），难以用标准图示法来表达。因此，为了表达这些特殊的功能，我新造了若干图示法。在本章稍后的1.7节中将予以介绍。可以用你的绘图软件根据这些新造的图示法来绘制Objective-C的类和对象。希望适用于Objective-C的标准化对象建模图示法尽早面世。

1.3　预备知识

本书属于专业丛书，所以并不是“iOS开发基础”或“Objective-C 24小时入门”。读者应具有iOS SDK（Cocoa Touch框架）的基础知识。此外，读者也应对Objective-C编程语言有足够的了解。否则，将无法理解本书中的很多观点和高级技巧。

尽管本书主要面向中高级开发人员，但读者并不需要对设计模式有深入的了解就能理解设计模式的概念。即便你已经在工作中接触过一些设计模式，你仍可从本书获益。介绍模式的所有章节都借助打比方，让读者对设计模式能有透彻的理解。而且，不易理解之处会有提示和说明。

准备好了吗？我们开始吧！

1.4　似曾相识的设计

身为开发人员，你可能有过这样的感受：“我以前解决过这个问题，但不记得具体是在哪里、怎样解决的。”经常会有这样的事儿，要是你重复做着特定类型的项目更是如此。比如，数据库的应用程序都有存储和检索数据的数据库访问功能。你要是记录下问题的细节和解决方法，就可以复用这些方法，而不是次次从零开始。

有关设计中最常见的似曾相识的情况以及解决方法，最早作出权威性论述和分类的是《设计模式》（Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides著，Addison- Wesley Professional出版社，1994年）。该书将设计面向对象软件的经验总结为可以有效使用的设计模式。有趣的是，Cocoa（Touch）框架的前身NEXTSTEP的设计中采用了很多漂亮、可复用的面向对象软件设计模式，而这对激发GoF（该书作者为人熟知的绰号）的灵感起到了重要作用。

根据《设计模式》一书，设计模式是对定制来解决特定场景下一般设计问题的类和相互通信的对象的描述。

简而言之，设计模式是为特定场景下的问题而定制的解决方案。特定场景指问题所在的重复出现的场景。问题指特定环境下你想达成的目标。同样的问题在不同的环境下会有不同的限制和挑战。定制的解决方案是指在特定环境下克服了问题的限制条件而达成目标的一种设计。

设计模式是经时间证明为有效的，对特定面向对象设计问题主要方面的一种抽象，体现了面向对象设计的重要思想。有些设计原则影响着设计模式。这些原则是构建可复用、可维护的面向对象应用程序的经验法则，比如，“优先使用对象组合而不是类继承”和“针对接口编程而不是针对实现编程”。

例如，通过给程序的变动部分定义接口而对其封装和隔离，这些部分的变动就独立于程序的其他部分，因为它们不依赖于任何细节。以后就可以变更或扩展这些可变的部分而不影响程序的其他部分。程序将因此能够更灵活而可靠地进行变更，因为我们消除了部分与部分之间的依赖关系并减少了耦合。这些益处使得设计模式对于可复用软件的编写非常重要。

程序（包括其中的对象和类），如果在设计中使用了设计模式，将来就更易于复用与扩展，更易于变更。而且，基于设计模式的程序会更加简洁而高效，因为达成同样目的所需的代码行会更少。

1.5　设计模式的起源——模型、视图和控制器

模型-视图-控制器（MVC）设计模式及其变体至少在Smalltalk诞生初期就已经出现了。这个设计模式是Cocoa Touch中很多机制和技术的基础。

在MVC设计模式中，对象在应用程序中被分为三组，分别扮演模型、视图和控制器。MVC模式也定义了对象之间跨越其角色的抽象边界的通信方式。MVC对Cocoa Touch应用程序设计起了重要作用。应用程序设计的一个主要步骤是决定对象或类应该属于这三组中的哪一组。如果应用程序的MVC划分得清晰，使用Cocoa Touch框架中的任何技术都会相对容易。

这个模式本身不是独立的模式，而是由几个其他基本模式组成的复合模式。这些基本模式将在本书介绍模式的章节中详细介绍。

相比于非MVC的应用程序，MVC的应用程序中的对象更加易于扩展和复用，因为其接口通常会定义得更好。而且，许多Cocoa Touch技术和架构是建立在MVC之上的，要求应用程序和对象服从于给MVC的角色设定的规则。

后面几节将阐述MVC中的各个角色在架构中如何发挥其作用。

1.5.1　在模型对象中封装数据和基本行为

模型对象维护应用程序的数据，并定义操作数据的特定逻辑。模型对象可以复用，因为它表示的知识适用于特定的问题领域。例如，模型对象可以表示复杂的数据结构，对应于用户在屏幕上所画的图形，或者仅仅表示待办事项应用程序中的一条待办事项。

只要加载的是包含有应用程序永久信息的数据，就应将其放入模型对象。理想状况下，模型对象同用于对其进行显示和编辑的用户界面之间不应有任何直接的关联。

1.5.2 使用视图对象向用户展示信息

视图对象可以响应用户操作，并懂得如何将自己展现在屏幕上。视图对象通常从应用程序的模型对象获取数据用以展示。它可以跟一个模型对象的部分、整体或者多个模型对象合作。通常，用户可以通过它修改数据。

虽然视图对象和模型对象之间关系密切，但是在MVC应用程序中它们之间没有耦合。除非因性能原因（比如视图需要对数据进行缓存），不应将视图用于存储它所展示的数据。

因为视图对象可以与许多不同的模型对象合作，所以它们往往可在不同应用程序之间复用并保持一致。UIKit框架提供了各种类型的视图类，可复用于我们的应用程序。

1.5.3 用控制器对象联系起模型和视图

控制器对象就像视图对象和模型对象的中间人。作为中间人或协调人，它建立起沟通渠道，使视图得以知晓模型的变更而给予响应。

除了协调作用之外，控制器对象还可以为应用程序执行其他操作，比如为应用程序管理其他对象的生命周期，进行设置和协调任务。

举例来说，用户通过操作视图对象（比如在文本框中输入）得到的值，可以传给控制器对象。控制器对象也可以让视图对象根据此用户操作改变其外观或行为，比如禁用某个文本输入框。

依照所需的设计，控制器对象可设计为可复用的或不可复用的（具体的[①]）。

1.5.4 作为复合设计模式的 MVC

MVC本身并不是最基本的设计模式，它包含了若干更加基本的设计模式，这些模式将在本书的模式章节中阐述。在MVC中，基本设计模式相互配合，确定了各功能之间的协作，这是MVC应用程序的特性。

Cocoa(Touch)的MVC用到的模式有：组合(Composite)、命令(Command)、中介者(Mediator)、策略（Strategy）和观察者（Observer）。

- 组合（第13章）——视图对象之间以协作的方式构成一个视图层次体系，其中既可以有复合视图（比如表格视图），也可以有独立视图（比如文本框或按钮）。每个层次的每个视图节点都可以响应用户的操作并把自己绘制到屏幕上。
- 命令（第20章）——这是一种“目标–动作”机制，视图对象可以推迟其他对象（比如控制器）的执行，让其他对象等到发生了某些事件后再执行。这一机制构成了命令模式。
- 中介者（第11章）——控制器对象起着中间人的作用，而这个中间人则采用了中介者模式，它构成了在模型和视图对象之间传递数据的双向通道。应用程序的控制器对象将模型的变更传达给视图对象。

① 原文为concrete。concrete class指既不继承其他类也不被其他类继承，没有虚函数的类。——译者注

- 策略（第19章）——控制器可以是视图对象的一个“策略”。视图对象将自身隔离，以期维持其作为数据展示器的唯一职责，而将一切应用程序特有的界面行为的决定委派给它的“策略”对象（即控制器）。
- 观察者（第12章）——模型对象向它所关注的控制器等对象发出内部状态变化的通知。

图1-1展示了虚构场景下这些模式是如何协同工作的。

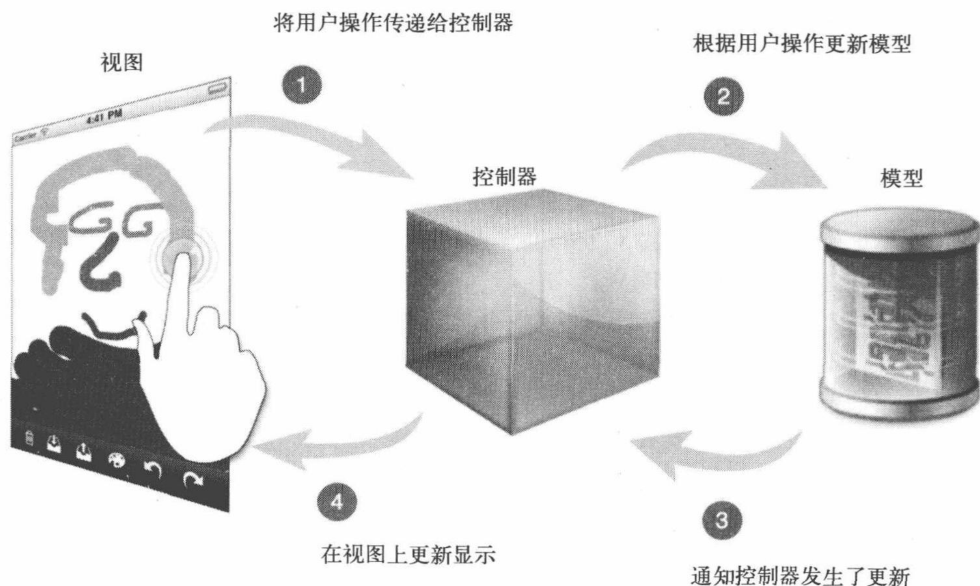


图1-1 展示模型、视图和控制器作为一组不同的实体如何交互的示意图

在图1-1中，

(1) 用户在画布视图上用手指触摸或拖动，产生一个触摸事件。被触摸的实际视图（图层）就在视图组合中的某个层次上。画布（视图）将触摸消息传达给视图控制器。

(2) 控制器对象接收到触摸事件及其相关信息，然后应用策略来变更模型的状态，必要时请求视图对象根据此事件更新其行为或外观。

(3) 每当变更发生并已反映到模型对象，模型对象就会通知所有已注册的观察者对象，如控制器。

(4) 控制器就像一个协调人，它将变更了的数据从模型传递给视图，以便视图可以相应地更新其外观。

1.6 影响设计的几个问题

设计模式肯定会从许多方面影响系统设计。但是有一些设计原则也会影响设计。有些原则针对一般的软件设计，而有些原则是针对Objective-C和Cocoa Touch的。我将在以下各节进行讨论。

1.6.1 针对接口编程，而不是针对实现编程

很多软件开发人员理解类、对象、继承、多态和接口这些面向对象概念。可是类继承与接口继承（子类型化）的区别何在？接口定义了类型，接口继承（子类型化）让我们可以用一个对象代替另一个对象。另一方面，类继承是通过复用父类的功能或者只是简单地共享代码和表述，来定义对象的实现和类型的一种机制。类继承让我们能够从现成的类继承所需大部分功能，从而快速定义新的类。其实，类和类型关系非常密切。不过，差别在于一个对象可以具有多种类型而不同类的对象可以有相同的类型。

定义具有相同接口的类群很重要，因为多态是基于接口的。其他面向对象的编程语言，如Java，允许开发者定义“接口”（区别于类）类型，它确定了客户端同所用的具体类之间的一种“合约”。Objective-C中有一种类似的东西叫做协议（protocol）。协议也是对象之间的一种合约，但本身不能实例化为对象。实现协议或者从抽象类继承，使得对象共享相同的接口（1.6.2节将讨论有关使用协议和抽象基类的问题）。因此，子类型的所有对象，都可以对针对协议或抽象类的接口的请求作出应答。

这样做有以下两点好处：

- 只要对象符合客户端所要求的接口，客户端就不必在意所使用对象的确切类型；
- 客户端只知道定义接口的协议或者抽象类，因此客户端对对象的类一无所知。

这就得出了GoF在书中所说的可复用面向对象软件设计的原则：

针对接口编程，而不是针对实现编程。

通常的做法是，在客户端的代码中不声明特定具体类的变量，而只使用协议或抽象类定义的接口。这种概念和思想贯穿本书。

1.6.2 `@protocol` 与抽象基类

嗯，应该针对接口编程，而不是针对实现编程。但是，要针对什么样的接口呢？在Objective-C中，有一种称为协议的语言功能（语法为`@protocol`）。协议并不定义任何实现，而只声明方法（method），以确定符合协议的类的行为。因此协议是只定义了抽象行为的“接口”。实现协议的类定义这些方法的实现，以执行真正的操作。另一种定义高度抽象类型的方法是定义抽象基类（Abstract Base Class，ABC）。通过抽象基类，我们可以生成一些其他子类可以共享的默认行为。抽象基类与通常的类相似，只是预留了一些可以或应该由子类重载的行为。

变更过去定义的协议可能会破坏实现该协议的类。协议（或接口）是抽象类型与具体类型之间的一种合约。如果变更合约，所有相关的事项也需要变更。唯一例外是，只使用`@optional`指令（directive）将协议的部分方法变更为“可选的”。而抽象基类在接口变更方面要稍微灵活一些。我们可以向抽象基类随意追加新的方法，而不会破坏继承链的其他部分。而且，对于子类可能用到的占位方法（stubbed-out method）中定义的默认行为，我们可以随意地进行追加、删除或分解。

客户端如果要使用由协议所定义类型的对象，比方说有个协议叫`Mark`，则需要使用以下语

法来引用它：

```
id <Mark> thisMark;
```

如果Mark被声明为抽象基类，那么语法应该跟其他类一样，如下所示：

```
Mark *thisMark;
```

在接受以Mark协议的对象作为参数的方法中，语法应该是这样：

```
- (void) anOperationWithMark:(id <Mark>) aMark;
```

如果Mark是抽象基类，那么语法是这样：

```
- (void) anOperationWithMark:(Mark *) aMark;
```

哪个看上去更好呢？很明显，Mark协议的引用显得有点儿笨拙。那么为什么还要使用协议呢？一个主要原因是，Objective-C与C++等其他面向对象语言不同，它不支持多重继承。如果一个类既要是UIView的子类，同时又要是定制的抽象类型，那么这个抽象类型就只能是协议而不能是抽象基类。只对一个抽象类型进行子类化的类，可以让它们直接对抽象基类进行子类化。

但是也许你以后会更改设计，以使抽象类型也能够是UIView 等的子类。能否两全其美？通常这种情况下灵活的做法是，首先为不需要子类化其他类的类定义一个抽象基类，然后可以定义一个同名协议，让包括这个抽象基类在内的其他类去实现。在Cocoa Touch框架中你会发现类似的策略，例如，NSObject基类符合NSObject协议。

1.6.3 对象组合与类继承

类继承或子类化让我们可以使用其他的类来定义类的实现。子类化常常被称为白箱复用(white-box reuse)，因为父类的内部描述与细节通常对子类可见。

对象组合可以替代类继承。对象组合要求被组合的对象具有定义良好的接口，并且通过从其他对象得到的引用在运行时动态定义。所以可以将对象组合到其他对象中，以构建更加复杂的功能。由于对象的内部细节对其他对象不可见，它们看上去为“黑箱”，这种类型的复用称为黑箱复用 (black-box reuse)。

使用类继承和对象组合的白箱和黑箱复用各有其优缺点。以下是对类继承的一些优缺点的总结。

优点是：

- 类继承简单直接，因为关系在编译时静态定义；
- 被复用的实现易于修改。

缺点是：

- 因为类继承在编译时定义，所以无法在运行时变更从父类继承来的实现；
- 子类的部分描述常常定义在父类中；
- 子类直接面对父类实现的细节，因此破坏了封装；
- 父类实现的任何变更都会强制子类也进行变更，因为它们的实现联系在了一起；
- 因为在新的问题场景下继承来的实现已过时或不适用，所以必须重写父类或继承来的实现。

由于实现的依存关系，对子类进行复用可能会有问题。有一个解决办法是，只从协议或抽象基类继承（子类型化），因为它们只有很少的实现，而协议则没有实现。

对象组合让我们同时使用多个对象，而每个对象都假定其他对象的接口正常运行。因此，为了在系统中正常运行，它们的接口都需要经过精心的设计。然而，对象组合也有其优缺点应予考虑。

优点是：

- 不会破坏封装，因为只通过接口来访问对象；
- 大大减少实现的依存关系，因为对象的实现是通过接口来定义的；
- 可以在运行时将任意对象替换为其他同类型的对象；
- 有助于保持类的封装以专注于单一任务；
- 类及其层次结构能保持简洁，不至于过度膨胀而无法管理。

缺点是：

- 设计中涉及较多对象；
- 系统的行为将依赖于不同对象间的关系，而不是定义于单个类中；
- 理想情况下，不需要创建新的组件就能实现复用；十分罕见的情况是，通过对象组合的方式，仅仅对已有的组件进行组合就能得到所需的全部功能；实际上，现成的组件总是不太够用。

尽管有以上缺点，对象组合仍然对系统设计有诸多好处。我们可以通过在某些部分使用类继承来克服这些缺点，使得利用已有组件创建新的组件较为容易。

优先使用对象组合而不是类继承，并不是说完全不使用类继承。需要根据具体情况对如何复用类和对象作出清晰的判断。如果系统设计得合理，类继承与对象组合可以相互配合。设计类时，通常倾向于考虑对象组合。然后寻找出冗余行为，进行设计细化。如果找到冗余行为，也许意味着此处应该使用类继承。在本书对设计模式的讨论中会谈到对象组合。

1.7 本书用到的对象和类

本书使用各种图形、图表来解释模式的一些重要思想。有时会使用屏幕截图或视觉表现来展示诸如组合树之类对象的结构。但是我需要更加正式而清楚的方法，来表达类与对象间的关系和相互作用。贯穿于本书中模式的最常用的图示法是类和对象图。我借用并修改了OMT（Object Modeling Technique，对象建模技术）的图示法，以满足我的需要。本节将介绍用于类和对象图的图示法。

1.7.1 类图

类图用来说明类、类之间的静态关系和类的结构。在Objective-C中，应用程序可以定义协议、（抽象）类以及范畴[①]。

① 原文为category，也译为“分类”，本书采用的是苹果公司中文网站上的译法。——译者注

1. 协议、抽象类、具体类和范畴

通常，用圆角矩形框表示类实体，在上部用粗体标出名字，下部是操作的名字。如果你看的是本书的电子版，你会看到协议的标题栏的背景为粉色，而其他类实体的标题栏背景为浅蓝色。抽象的名字用斜体表示。因此，协议和抽象类用粗斜体表示。协议名用尖括号括起来。实例变量放在框的底部。各种类实体的例子如图1-2所示。

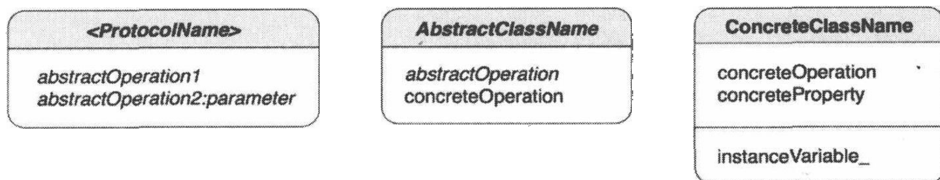


图1-2　左边为具有抽象操作的协议；中间为具有抽象操作和具体操作的抽象类；右边为具体类，带有具体操作、具体属性和实例变量

表示范畴有点儿麻烦，因为在本书写作时原来的OMT不支持范畴。范畴是对类的扩展，但又不是那个类的子类。所以要是用箭头来表示这种关系就会引起混乱。我想出了如图1-3所示的扩展类框图的图示法。

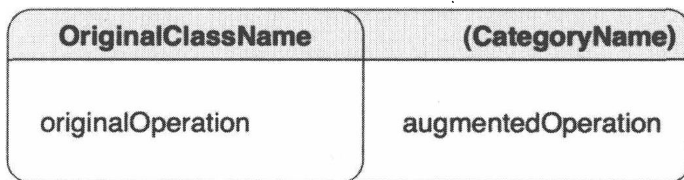


图1-3　左边为原来的类，右边为它的范畴扩展

原来的类框图在左边，有一个类似的矩形框附加到其上。范畴名用括号括起来。跟其他类相似，增加的操作放在框图的下部。这一图示法可能不是最灵活的，尤其是当设计中有100个范畴时，但用于本书的图示是足够了。

在类图或对象图中会有一些设计中的其他角色。这些角色可以是抽象实体（如客户端），或者设计范围之内或之外的其他类。灰色的圆角矩形框表示交互中的隐式角色，但对所讨论的问题来说并不重要。相反，参与者类会用黑实线的圆角矩形框来表示。如果这是电子书，参与者类的框图的背景色是浅蓝色的，隐式的类的背景色是透明或白色的（见图1-4）。

图1-4　左边为隐式类，右边为参与者类

2. 实例化

如果要表示一个类创建了另一个类，我会用带有箭头的虚线来表示这种关系。这被称为“创建”关系。箭头指向被实例化的类，如图1-5所示。

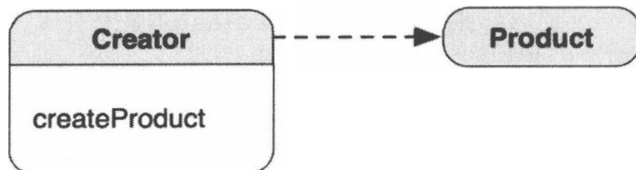


图1-5　一个类实例化另一个类

3. 继承

类继承的OMT图示法用空心三角形将子类连接到其父类。图1-6显示了这种关系，ConcreteClass是子类，代表继承的箭头指向其父类AbstractClass。对于接口继承（子类型化或符合接口），我用类似的箭头来表示这种关系，只是箭头后面的是虚线。图1-6也显示了这种关系。

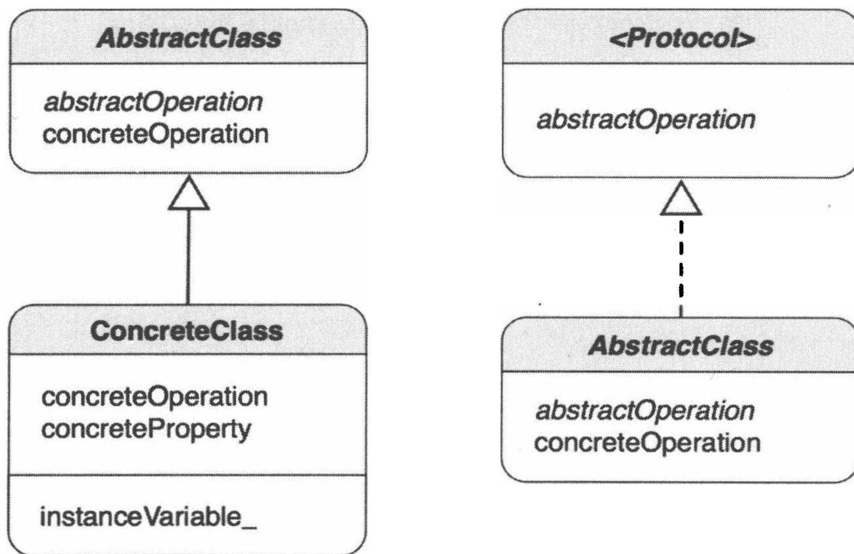


图1-6　左边，具体类继承抽象类，体现了类继承的关系。右边，一个类子类型化（符合）一个协议，体现了接口继承关系

4. 相识

我使用从一个类指向另一个类的箭头来表示相识（acquaintance）关系。这种关系与另一种叫做聚合的关系对于对象组合原则至关重要（聚合将在后面讨论）。图1-7显示了这种关系，ConcreteClass拥有对AnotherClass对象的引用，但不“拥有”AnotherClass对象的实体，而且引用也可以被其他对象分享。简单地说，ConcreteClass认识AnotherClass。

5. 聚合

跟相识关系一样，我使用箭头来表示对另一个对象的引用，只是在箭头的根部有一个菱形。但是这种引用关系有些不同。图1-8显示了ConcreteClass与AnotherClass有聚合（aggregation）关系。AnotherClass是ConcreteClass的一部分，ConcreteClass和AnotherClass构成聚

合体。而聚合体由ConcreteClass来表示。AnotherClass不是聚合体。图中还显示了引用的另一种属性。我使用双头箭头来表示“多于一个”。因此，ConcreteClass包含有AnotherClass的多个实例，即instanceVariable_。

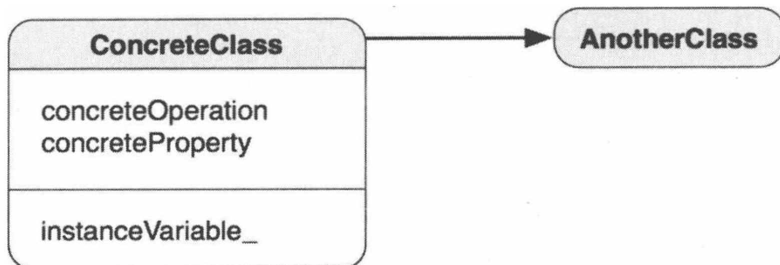


图1-7　ConcreteClass与AnotherClass形成相识关系

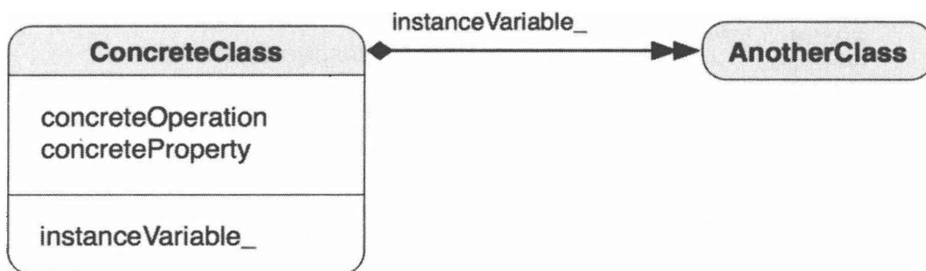


图1-8　ConcreteClass以对AnotherClass的多个引用形成聚合关系

6. 伪代码

有时候用伪代码简要记述某些操作的实现，可以更清楚地说明模式。伪代码注记的正文放在带卷角的矩形框中，如图1-9所示。

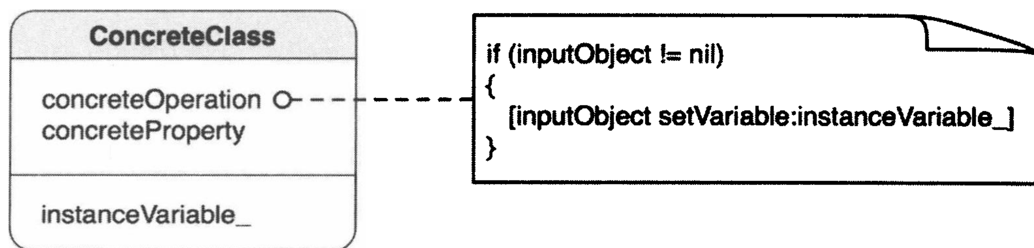


图1-9　伪代码注记

1.7.2　对象图

对象图只用来表示对象间的关系。它表示了设计模式中各个对象之间如何相互联系。对象名使用“aSomeClass”的格式，这里SomeClass是对象的类。表示对象的图形与类图中用到的很相

似。对象被放在一个圆角矩形框中，矩形框有两部分，将对象名和它的对象引用分开。标题栏的背景也是浅蓝色。根部为圆形的实心箭头指向被引用的其他对象。图1-10是对象图的一个例子。

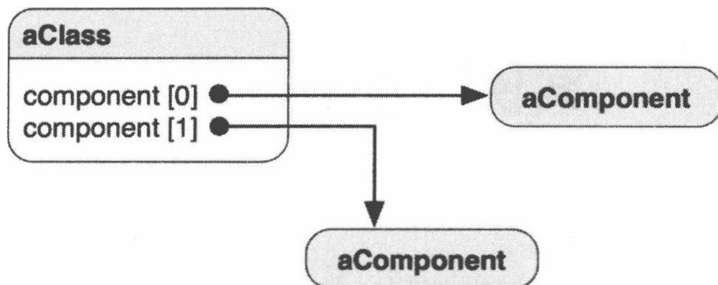


图1-10 用带有圆形根部的箭头表示源自aClass对象的引用的对象图

1.8 本书如何安排模式的讲解

本书涉及21种设计模式，根据其实际主题或应用领域分为以下8个功能部分：对象创建、接口适应、对象去耦合、抽象集合、行为扩展、算法封装、性能与对象访问，以及对象状态。

1.9 总结

本章介绍了设计模式的背景、历史和益处，以及影响应用程序架构设计的一些问题。我真的希望本章为掌握以下各章的真枪实弹作好了热身准备。

第2章 案例分析：设计一个应用程序

好的应用需要好的创意，就像App Store里下一个最赚钱的付费应用一样。对于本书的示例程序，我们的创意是一个绘图应用。我们把它命名为TouchPainter。它简单而全面，足以展示许多设计模式和最佳做法。

我们将经过几个设计阶段，每个阶段都会提出与设计相关的需求、用例和问题。在此期间，将探讨各种可以解决设计问题以满足需求的设计模式。

设计过程中有3个重要的里程碑：

- 想法的概念化；
- 界面外观的设计；
- 架构设计。

从想法的概念化开始，我们将汇集有关TouchPainter应用的一些基本需求和用例，比如用户应该怎样使用此应用，以及用户使用此应用时的体验。

有了开发应用的好点子之后，接下来应该先考虑应用的界面外观。界面外观的设计过程，让开发人员得以探讨哪些UI要素可以合乎逻辑地组合在一起。这个过程不但让开发人员首先对哪些好看哪些难看有个整体认识，也可能会消除不必要的UI要素，从而简化并增强用户体验。这是个反复的过程，所以设计要易于修改。很多开发人员用铅笔在纸上画出哪些不同视图可以放在一起。如果草稿看起来令人满意，开发人员或UI设计师可以开始用软件把线框和更具真实效果的微件（widget）组合在一起，以细化设计。若是结果并不如愿，则要返回到纸面设计，或者在屏幕上修改微件。在2.2节中，在把界面外观的线框与需求相结合的时候，我们将探讨这一过程。

界面外观完成以后，就该确定一些影响应用程序架构的技术问题了。比如可以问这样的问题："用户怎样可以打开涂鸦图？"

准备好了吗？我们开始吧。

2.1 想法的概念化

任何类型的软件开发都要有一些需求，这个应用也不例外。但并不需要在一开始就确定全部细节。我们要做的是从最基本的开始。那么，第一个需求是什么呢？

- 可以用手指涂鸦的画板。

如果只能绘制黑白图形，我觉得任何用户对这种绘图体验都不会满意。要是有设定不同颜色和线条粗细的选项肯定很不错。这样又有了一个需求：

- 用户可以改变线条颜色和粗细。

但只允许用户进行涂鸦并不够，还应该允许用户保存涂鸦图。所以得到了这个需求：

- 允许用户保存涂鸦图。

要是用户不能打开保存的涂鸦图并作修改，保存将毫无意义。所以又有了这个需求：

- 允许用户打开保存的涂鸦图。

要是用户不喜欢自己的作品，想删除重来怎么办？

- 允许用户删除当前涂鸦图。

要是允许用户撤销与恢复涂鸦肯定很不错。所以又得到了一个需求：

- 允许用户撤销和恢复涂鸦。

这个清单可以一直列下去，但眼下已经有了可以启动设计的基本需求。但在进入设计阶段以前，应该确定把握了应用程序的界面外观，以保证我们很清楚应用程序是什么样子。来总结一下这个了不起的绘图应用的第一批需求吧。

- 可以用手指涂鸦的画板。
- 用户可以改变线条颜色和粗细。
- 允许用户保存涂鸦图。
- 允许用户打开保存的涂鸦图。
- 允许用户删除当前涂鸦图。
- 允许用户撤销和恢复涂鸦。

2.2 界面外观的设计

还记得上一次被要求（或自己决定）在最后一刻修改应用程序的用户界面（或用户体验）吗？那种感觉可一点也不好。更好的做法，至少在iOS的开发中，是尽早设计接近最终产品的完整界面外观与用户体验。我称之为界面外观驱动的（Look-and-Feel-driven）设计。大家已经听说过数据驱动的设计、事件驱动的设计和测试驱动的设计。但是这些都只是针对技术细节。然而，界面外观驱动的设计让我们能在一开始就专注于用户体验。这样不只是能够为发布前的最后关头省下时间，而且也能确保利益相关者（stakeholder）与开发人员在开发过程中保持一致。即使在开发过程中，做好的应用程序界面外观和UI设计，可以给忙于编码的开发人员很好的视觉提示，告诉他们是在开发什么东西。这样可以提高生产效率，因为可以在初期就发现那些难以定位的bug和设计缺陷。

来设计绘图应用的界面外观吧。如何开始呢？可以用笔和纸，或者如果你喜欢的话，也可以用电脑上的图形模板。图2-1就是我们的应用程序的第一个界面设计。

图中的第一稿显示，用户可以用手指在屏幕上画线（或任何其他图形）。因此这应该实现了

第一个需求：可以用手指涂鸦的画板。这部分应该没问题了。

在视图底部有一个工具条，上面有6个按钮，从左至右依次为：删除、保存、打开（保存的涂鸦图）、设置（线色和线宽）、撤销和恢复（屏幕上所绘的图）。

线框图看起来就像一个典型的允许用户用手指绘图的iPhone应用。用户也可以改变其他与绘图相关的设置。目前我们对这个外观很满意。接着我们来看针对其他需求的下一个线框图（图2-2）。

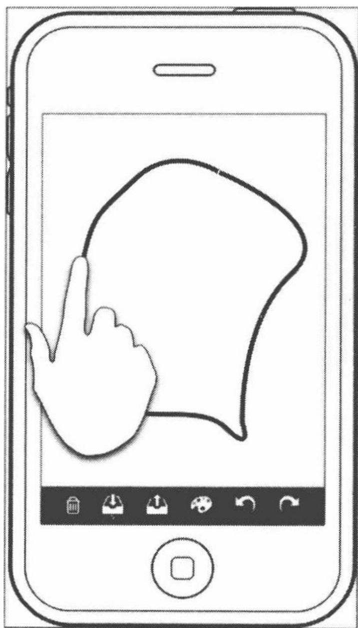


图2-1　针对第一个需求的主画布视图的线框图

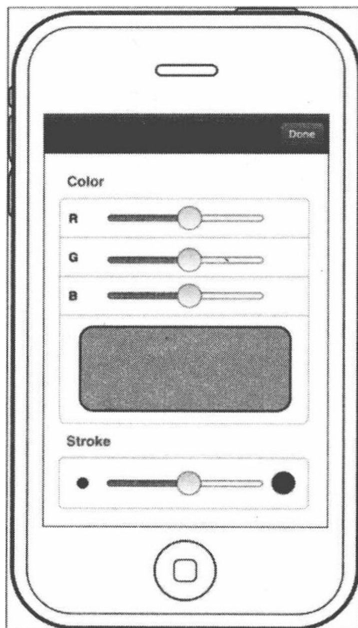


图2-2　实现第二个需求的调色板视图的线框图

这个线框图中，用户可以通过改变各个颜色分量与线宽的滑动条来调节线色与线宽。画面中间的灰色矩形区域将根据所选的RGB值显示当前的线色。调节画面底部的滑动条可以设定线宽。这个线框图实现了第二个需求：用户可以改变线条颜色和粗细。按Done按钮可以回到图2-1所示的主画布视图。

至此，可以肯定已有的线框图实现了6个需求中的5个。只剩下了打开保存的涂鸦图的第4个需求。对于这个需求，首先要问，用户如何才能知道要打开什么涂鸦图呢？肯定需要某种浏览器，让用户可以浏览全部的涂鸦图，然后从中选择想要的涂鸦图。可以把它画成一个缩略图视图。

缩略图视图的粗略线框图如图2-3所示。

当用户单击主画布视图的打开涂鸦图按钮时，会打开如图2-3所示的缩略图视图。用户可以通过在画面上上下滑动来滚动涂鸦缩略图的列表。用户也可以单击缩略图，在画布视图中打开它，以继续绘图。或者，通过单击Done按钮来返回到主画布视图。

如果需要的话，可以以后再进一步完善设计，目前我们对界面外观的线框图很满意。开始进行下一步——架构设计吧。

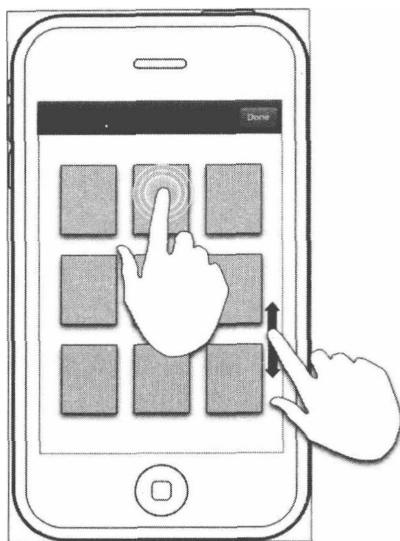


图2-3 实现第4个需求的缩略图视图的线框图

2.3 架构设计

大家都知道，本书是关于Objective-C和iOS开发的设计模式的书，因此我猜想各位已经迫不及待地要问这样一个问题了：这个应用程序里要使用什么模式呢？嗯，我相信很多人有这样的习惯，先找到模式，然后花很多时间把模式放到程序中，让程序显得比较“专业”，尽管这些模式可能并不能正确地解决问题。

在考虑解决方案之前，必须先有问题。2.2节中，我们已经通过基本的用例为应用程序提出了一些基本的需求。用例描述通过应用程序，“谁”可以做“什么”。我们可以详细描述这些需求（或用例）中的一些设计问题，然后来找到可能的解决办法。

我们要列出并考察通过细化原始需求而得出的一些问题。每个问题都有与原问题相关的细化而特定的特征或子问题。我们得出4个主要问题及其细化特征如下：

- 视图管理
  - 从一个视图到另一个视图的迁移
  - 使用中介者来协调视图迁移
- 如何表现涂鸦
  - 在屏幕上可以画“什么”
  - 用组合结构来表示痕迹（mark）
  - 绘制涂鸦图
- 如何表现保存的涂鸦图
  - 获取涂鸦图的状态

    - 恢复涂鸦图的状态
- 用户操作
    - 浏览涂鸦缩略图的列表
    - 涂鸦图的撤销和恢复
    - 变更线色与线宽
    - 删除屏幕上的当前涂鸦图

2.3.1 视图管理

第1章中讨论过模型–视图–控制器模式。模型表示视图所展示的数据。控制器在视图与模型之间起协调作用。因此每个控制器“拥有”一个视图和一个模型。在iOS开发中，这种控制器称作视图控制器（view controller）。根据2.2节的线框图，我们清楚TouchPainter需要什么。共有3个视图，每个视图应该由相应的控制器来维护。所以基于最初的UI设计，有3个控制器：

- CanvasViewController
- PaletteViewController
- ThumbnailViewController

CanvasViewController包含了图2-1所示的主画布视图，用户可以用手指在该视图中涂鸦。PaletteViewController管理一组用户控件元素，让用户可以调节线色与线宽，如图2-2所示。新的设定会传递给CanvasViewController的模型。ThumbnailViewController以缩略图的形式展示先前保存的全部涂鸦图，用户可以浏览并单击以打开涂鸦图，如图2-3所示。有关涂鸦图的全部必需信息会传递给CanvasViewController，以将涂鸦图显示在画布视图。

各个视图控制器之间有交互。它们彼此紧密依存。情况会变得一团糟，尤其是在以后需要往应用程序中加入更多视图控制器的时候。

1. 从一个视图到另一个视图的迁移

当用户单击CanvasViewController的调色板按钮时，视图会替换为PaletteViewController的视图。同样，单击CanvasViewController的打开缩略图视图的按钮，会打开ThumbnailViewController的视图。当用户单击导航条上的Done按钮以结束当前操作时，会回到CanvasViewController的视图。图2-4显示了控制器之间可能的交互。

看着图2-4，读者可能会这么想：“这没什么问题。我一直这么做，它也总能正常工作。”但其实上，对多数应用程序而言这并不是好的设计。常见的iOS应用程序中，像图2-4那样的视图迁移并不显得很复杂，尽管这些控制器之间有一定的依存关系。如果视图与其控制器之间彼此依存，应用程序就不好扩展（scale）。而且，如果修改视图变换的方式，它们的代码变更都将不可避免。其实，依存关系不仅限于控制器，也包括了按钮。按钮间接地与某些视图控制器发生关联。如果应用程序变大变复杂，依存关系将无法控制，随之而来的是大量难以理解的视图迁移逻辑。需要一种机制来减少不同视图控制器与按钮间的交互，以降低整体结构的耦合，提高其可复用性与可扩展性。像视图控制器那样的协调控制器会有助于此，但其作用不是协调视图与模型；它要协调

不同的视图控制器，以完成正确的视图迁移。

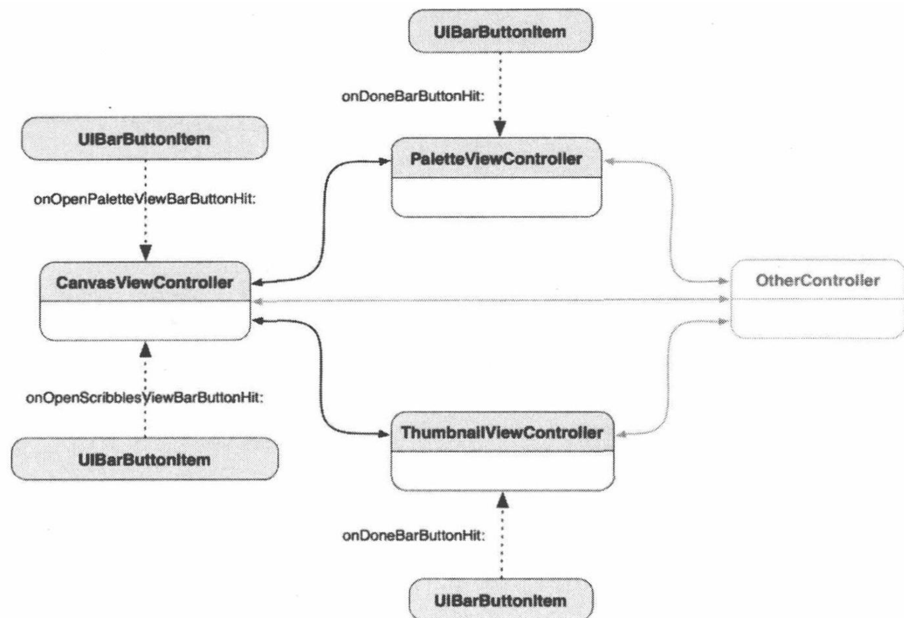


图2-4 典型的iOS应用程序设计中，CanvasViewController、PaletteViewController与ThumbnailViewController之间的依存关系图

2. 使用中介者来协调视图迁移

中介者模式（第11章）是指用一个对象来封装一组对象之间的交互逻辑。中介者通过避免对象间显式的相互引用来增进不同对象间的松耦合（loose coupling）。因此对象间的交互可以集中在一处控制，对象间的依存关系会减少。新交互模式的结构如图2-5所示。

向架构中引入了新成员CoordinatingController之后，以后对架构的变更将会容易很多。各种视图控制器与按钮发出视图变换的请求，而CoordinatingController则封装了协调这些请求的逻辑。交互不仅限于视图迁移，也可以是信息传递和对操作的调用。要是以后要对这些交互作改动，只要改CoordinatingController中的一个地方即可，不会涉及其他任何视图控制器。在这种结构下，CoordinatingController认识交互图中的每个对象。单击按钮会触发对CoordinatingController的调用，请求视图迁移。根据在按钮中保持的信息（比如标签），CoordinatingController知道按钮想要打开什么视图。如果想复用其中任何一个视图控制器，也不必一上来就考虑如何把它们联系在一起。

现在，CanvasViewController、PaletteViewController、ThumbnailViewController以及它们的按钮的行为就像在管弦乐队中演奏的不同乐手一样。演奏一首乐曲，乐手之间并不相互依赖，只是按着指挥的指令在什么时候做什么。你能想象出没有指挥的管弦乐队是什么样子吗？

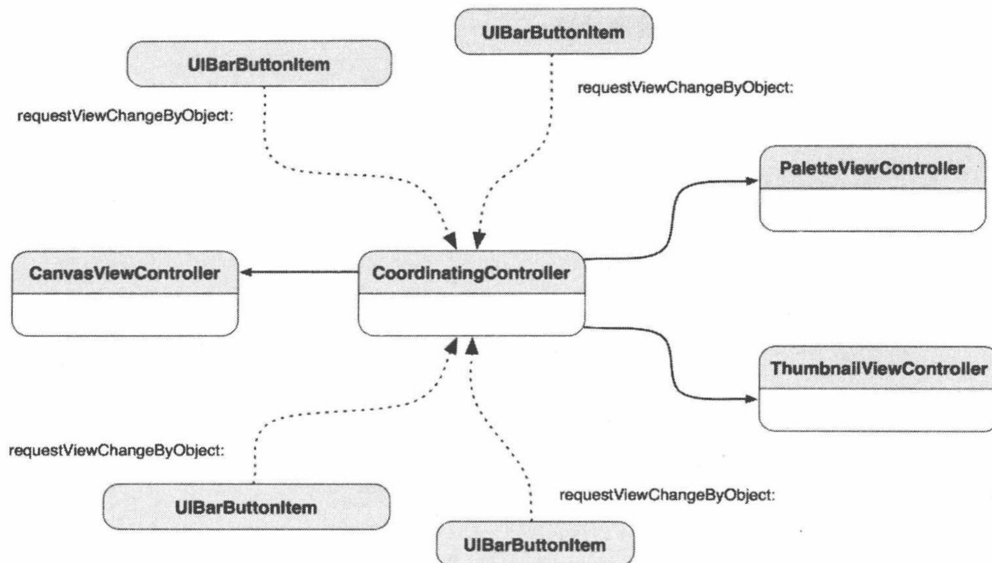


图2-5　引入了中介者CoordinatingController之后，视图控制器及其按钮之间的依存关系变少了

2.3.2　如何表现涂鸦

当用户触摸画布视图时，位置等触摸信息会被收集。需要某种数据结构来组织屏幕上的触摸，以便将所有触摸聚集为一个实体来进行管理。尤其是在以后需要解析这些数据的时候，数据结构必须非常有条理并可预测。

1. 在屏幕上可以画“什么”

在考虑用于保持触摸数据的可能的数据结构之前，先来看看应该如何定义线条。

如果用户先触摸屏幕然后移动手指，就生成一个线条。然而，如果手指并未移动而在起始位置处就结束了触摸，就应该视为一个点。如图2-6所示。

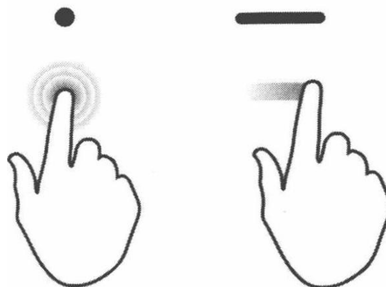


图2-6　在屏幕上点触生成的点与拖动生成的线

凭直觉会认为点是单一位置，而线条在轨迹上有多个位置。线条本身是包含了一串触摸位置的实体。这里的问题是如何将两种类型的实体当做单一类型来管理。是否可以认为线条包含多个点？如果线条使用数组来保持所有的点，那么是否可以用线条结构来既表示点又表示线条？可是那样的话，对于点的情况，会由于使用了仅有一点的数组而浪费内存。最好的情况是使用一种设计模式达到两全其美。

在研究如何能够使用一个数据结构来表示线条和点之前，先来考虑一下如何在屏幕上画点和

线。Cocoa Touch框架包括了一个叫做Quartz 2D的框架，它提供了在UIView上画二维图形的API，包括线和各种多边形。

要在UIView上绘图，需要得到运行库提供的绘图上下文（drawing context）。例如，如果想画一个点，就要把绘图上下文和表示点大小的CGRect——包含宽、高以及位置信息，传给Quartz 2D的函数CGContextFillEllipseInRect()。如果画线，首先需要通过调用函数CGContextMoveToPoint()让上下文移到第一点。把第一点赋给上下文之后，要通过对CGContextAddLineToPoint()的连续调用，添加组成线的其余点。添加完最后一点后，将调用CGContextStrokePath()来完成画线。这样，这些点就连接为一条线，画在UIView之上。图2-7是对此过程的直观描述。

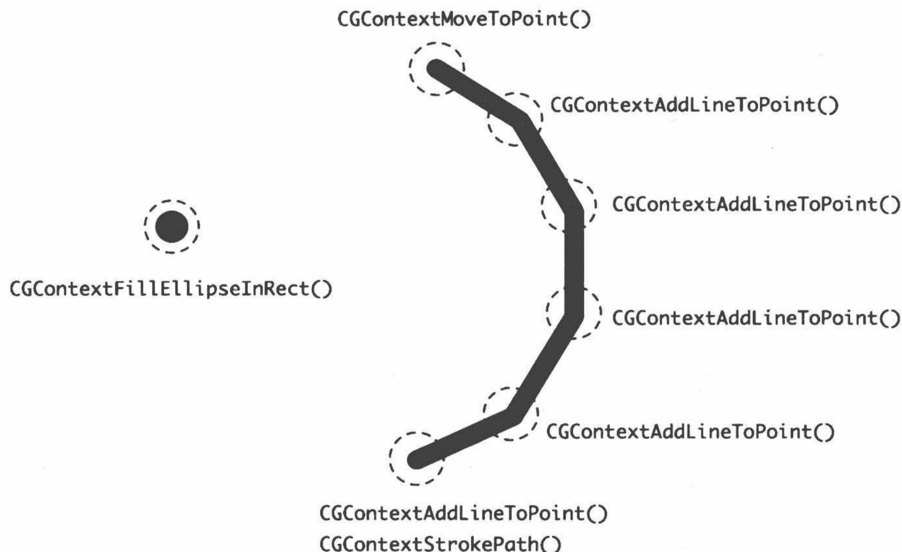


图2-7 使用Quartz 2D画点和线条的直观描述

2. 用组合结构来表示痕迹

当然，可以使用所知的任何数据结构来存储线条和点等。但是，如果全部都用（比如说）多维数组来保存，使用和解析时就需要进行很多类型检查，而且，要使结构可靠而一致，需要进行大量调试工作。如果要用一种结构既可以保存独立的点，又可以保存把点（顶点）作为子节点的线条，面向对象的做法是使用树。树的表现形式，能够把线条与点这样的复杂对象关系组织与局限于一处。解决这种结构问题的一种设计模式叫做组合模式（第13章）。

通过组合模式，可以把线条与点组合到树形结构中，以便统一处理每一节点。线条（stroke）、点（dot）和顶点（vertex）的直观结构如图2-8所示。

点是叶节点，是独立的实体。线条是组合体，包含了其他点作为顶点；同时，也可以包含其他线条组合体。它们是很不一样的实体。想要它们每个都表示相同的类型，那么就需要将其广义化（generalize）为共同的接口。因此，不管每个组件实际是什么具体类型，通用类型还是相同的。通过这种方案，在使用它们的时候就可以对其统一对待。

如果组件只是一个点，那么它会表现为一个实心圆，在屏幕上代表一个点。但是，如果是应该作为一组实体连接起来的一串点（顶点），就会被绘制成连接起来的线（线条）。这就是两者的区别。

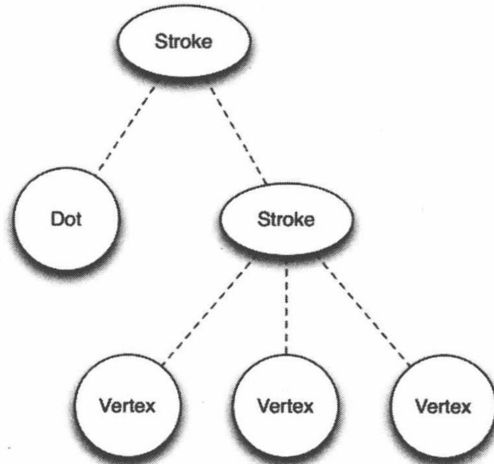


图2-8　点与线条的组合结构

不论线条或点，其实都是介质上的某种"痕迹"（mark），这里的介质是屏幕，因此我们添加`Mark`作为`Vertex`、`Dot`与`Stroke`的父类型。它们之间的关系如图2-9中的类图所示。

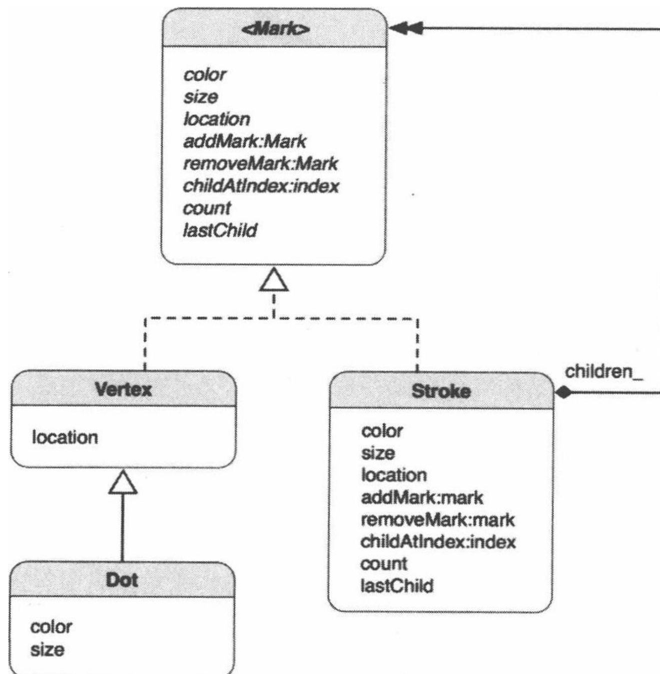


图2-9　组合结构的`Mark`模型的类图，`Dot`与`Stroke`为具体类

父类型Mark是一个协议。协议是Objective-C的一个特性，定义了行为的合约而没有实现。具体类实现Mark协议声明的接口。Vertex、Dot和Stroke都是Mark的具体类。Vertex与Dot都需要位置信息，而Dot还需要颜色和大小的附加属性，所以Dot作为子类继承Vertex。

Mark为所有具体类定义了属性和方法。所以，当客户端基于接口来操作具体类的时候，可以统一对待每个具体类。Mark对象有这样的方法，可以把其他Mark对象加为自己的子节点，形成组合体。Stroke实现了把其他Mark对象加为子节点的方法。Mark协议中还定义了与子节点管理有关的一些其他操作，比如移除子节点、按照序号（index）返回特定子节点，以及返回子节点数和子节点列表中的最后一个子节点。

3. 绘制涂鸦图

现在我们建好了一个数据结构，它可以让应用程序以一种合理的方式管理由用户触摸生成的点。在屏幕上显示的时候，触摸的位置信息就非常重要。可是在2.2节中讨论的组合结构并没有在屏幕上描画自身的算法。

大家知道，在UIView上绘制定制图形的唯一方式是重载其drawRect:实例方法。这个方法会在请求视图更新时被调用。例如，可以向UIView发一个setNeedsDisplay消息，然后它就会调用定义了定制绘图代码的drawRect:方法。然后我们就能以当前图形上下文（graphics context）作为CGContextRef，在此方法调用中绘制想要的图形。下面说明如何使用这一机制来绘制组合结构。

可以向Mark协议添加一个绘图操作，比如drawWithContext:(CGContextRef)context，以使每一节点能够按其特定目的来绘制自身。可以把从drawRect:方法中得到图形上下文传递给drawWithContext:方法，以便Mark对象用它来绘图。Dot的drawWithContext:方法可以像代码清单2-1这样来实现。

代码清单2-1 Dot中的drawWithContext:实现

```
- (void) drawWithContext:(CGContextRef)context
{
  CGFloat x = self.location.x;
  CGFloat y = self.location.y;
  CGFloat frameSize = self.size;
  CGRect frame = CGRectMake(x - frameSize / 2.0,
                            y - frameSize / 2.0,
                            frameSize,
                            frameSize);

  CGContextSetFillColorWithColor (context,[self.color CGColor]);
  CGContextFillEllipseInRect(context, frame);
}
```

在当前上下文中，能够按照位置、颜色和大小绘制一个椭圆（点）。

至于顶点，它只提供了线条中的一个特定位置。因此，Vertex对象将只在上下文中使用自身的位置（坐标）往线上添加一点（好吧，按照Quartz 2D的说法，是往点上添加一条线），如代码清单2-2所示。

代码清单2-2 Vertext中的drawWithContext:实现

```
- (void) drawWithContext:(CGContextRef)context
{
  CGFloat x = self.location.x;
  CGFloat y = self.location.y;

  CGContextAddLineToPoint(context, x, y);
}
```

对于Stroke对象，它需要把上下文移至第一点，向每个子节点传递同样的drawWithContext:消息和图形上下文，并设定其线色。然后以Quartz 2D函数CGContextSetStrokeColorWithColor与CGContextStrokePath结束整条线的绘制操作，如代码清单2-3所示。

代码清单2-3 Stroke中的drawWithContext:实现

```
- (void) drawWithContext:(CGContextRef)context
{
  CGContextMoveToPoint(context, self.location.x, self.location.y);

  for (id <Mark> mark in children_)
  {
   [mark drawWithContext:context];
  }

  CGContextSetLineWidth(context, self.size);
  CGContextSetLineCap(context, kCGLineCapRound);
  CGContextSetStrokeColorWithColor(context,[self.color CGColor]);
  CGContextStrokePath(context);
}
```

设计Mark协议的主要挑战是以最小的操作集提供可扩充的功能。为添加新功能而对Mark协议及其子类做手术，不但有创伤而且易于出错。最终，对类的理解、扩展与复用变得更加困难。因此，关键是要致力于简单而一致的接口的一组充分的基本要素。

扩展如Mark这样的组合结构的行为，另一种方法是使用称为访问者模式的设计模式。访问者模式（第15章）允许将可应用于组合结构的外部行为定义为“访问者”。访问者“访问”复杂结构中的每一节点，根据被访问的节点实际类型，执行特定的操作。

2.3.3 如何表现保存的涂鸦图

内存中一个涂鸦图的表示形式是一个组合对象，它包含点与线条的递归结构。但是如何才能把这一表示形式保存到文件系统呢？两个地方的表示形式应该兼容，也就是说，一种表示形式能够毫无问题地转换为另一种。

如果不使用结构化而简洁的机制来保存对象，代码就会变得一团糟，尤其是在硬编码（hard-coded）于一处的时候。可以把一般的对象保存过程分解为多个步骤，如下所示：

(1) 把对象结构序列化成结构化的文件块blob[①]；

① blob，binary large object的缩写，指存储二进制数据的单一实体。——译者注

(2) 构建要保存blob的文件系统中的路径；

(3) 像普通文件一样保存blob。

至于从文件系统加载同一blob并复原为原先的对象结构，需要执行同样步骤，但是以相反的顺序：

(1) 重建文件系统中保存blob的路径；

(2) 像普通文件一样从文件系统加载blob；

(3) 反序列化并恢复原先的对象结构。

如果把这些步骤都放进一个巨大的函数，让它处理所有琐事，将很难管理和复用。而且，有时关心的不只是整个对象结构的保存，也有部分（比如，只是最后的修改）的保存。肯定需要某种封装化的操作去处理涂鸦图状态保存中遇到的各种类的特殊情况。

先不管路径的构建之类，只考虑获取内存中复杂对象的状态并转换成封装好的表示形式，以便将其保存到文件系统并在以后恢复。

可以使用称为“备忘录”（Memento）的设计模式来解决这类问题。“备忘录”允许对象按其想要的任何（或者任意复杂的）方式将自己的状态保存为一个对象，根据此模式这个对象称为备忘录对象。然后某个其他对象，比如看管人（caretaker）对象，将备忘录对象保管在某处，通常是文件系统或内存中。看管人对象不知道有关备忘录对象任何细节的格式。一段时间之后，接受到请求时，看管人对象将备忘录对象传回给原来的对象，让它根据在备忘录对象中保存的信息恢复其状态。图2-10显示了它们的交互顺序。

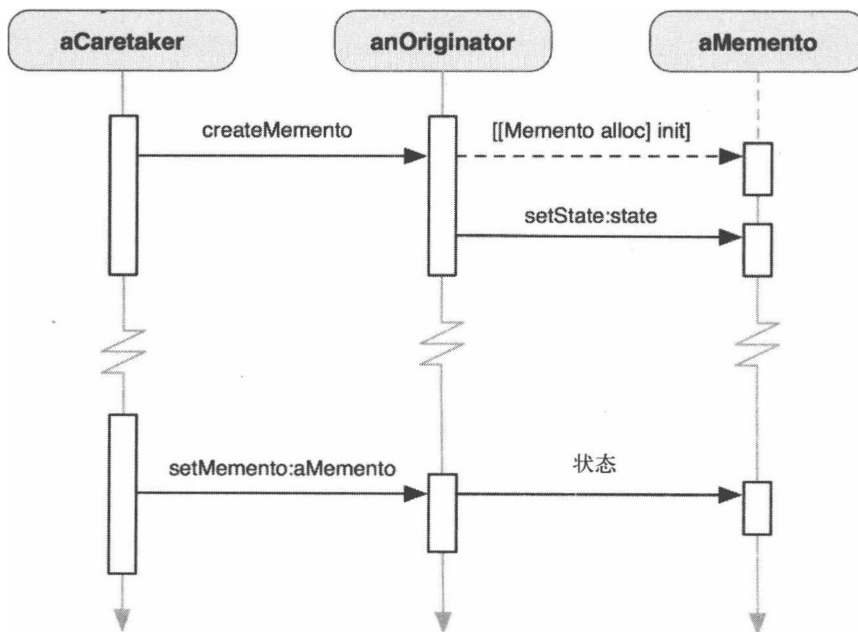


图2-10 anOriginator对象接受aCaretaker的指令，获取自身的状态为aMemento的时序图——aCaretaker管理aMemento并在后来将其传回给anOriginator，以使anOriginator恢复其状态

1. 获取涂鸦图的状态

根据我们的问题，需要将原始的对象结构与其被保存的状态分离。不过在深入细节之前，要想好需要归档并在以后反归档哪些东西。在图2-3中，我们草拟了一个用户界面，它允许用户浏览已保存涂鸦图的缩略图。除了要保存Mark对象之外，存储过程中也需要保存相应的画布视图的截屏图。

而且，可以决定归档更多要素，如画布和其他用于绘图的属性。随着不断推出这个程序的新版本，情况可能会发生变化。出于灵活性的考虑，我们决定创建另一个类来管理这些附加信息。我们把这个新类叫做Scribble。Scribble对象封装了Mark组合对象的一个实例，作为其内部状态。Scribble的作用就好像第一章介绍的模型-视图-控制器范例中的模型。在这个问题中，需要Scribble对象参与Mark组合体状态的保存与恢复过程，而不是直接使用Mark。Mark很难直接使用，因为它只提供了对组合结构的基本操作。

Scribble是系统的模型，此外，还需要看管人保存Scribble对象的状态。我们添加一个起此作用的类，并把它叫做ScribbleManager。保存涂鸦图的实际过程可能非常复杂，并且涉及很多其他对象，因此本节的其余部分将只讨论Scribble对象内部状态的保存。

如图2-11所示，CanvasViewController向ScribbleManager发起保存涂鸦图的调用。然后ScribbleManager请求传进来的Scribble对象创建“备忘录”。接着，Scribble对象创建一个ScribbleMemento的实例并用其保存其内部的Mark引用。此时，ScribbleManager既可以将返回的备忘录对象放在内存中，也可以将其保存到文件系统。在这个“保存涂鸦图”的过程中，要让ScribbleManager把备忘录对象保存到文件系统，所以ScribbleMemento对象需要把自身编码成一种数据形态，即NSData的实例。ScribbleManager对此数据一无所知，只是用自己才知道的路径将其保存于文件系统。整个过程就像传递装有备忘录对象的黑箱，箱子只能由发起对象打开和关闭。整个过程开始于从CanvasViewController到ScribbleManager的一个消息调用——saveScribble:scribble。

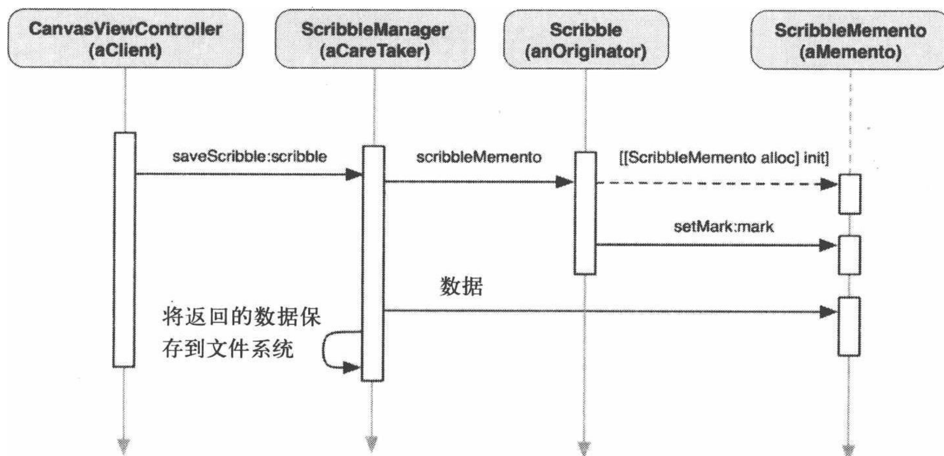


图2-11　说明Scribble的状态如何保存为ScribbleMemento对象的时序图

选用备忘录模式作为对象归档方案，至少有以下两点好处。

- ScribbleMemento对象的内部结构细节没有暴露出来。如果以后要修改Scribble对象对其状态的保存内容和保存方式，不用修改应用程序中其他的类。
- ScribbleManager不知道如何访问ScribbleMemento对象的内部表示（及其编码的NSData对象）。相反，它只是在其内部定义的操作之间传递这个对象，以将其保存到内存或文件系统。任何对备忘录保存方式的修改都不会影响其他类。

2. 恢复涂鸦图的状态

我们知道了如何将Scribble对象的状态保存为备忘录对象，那么，怎样通过同样的机制恢复Scribble对象呢？加载部分很像倒过来的保存过程。客户端（此处不必是原来的CanvasViewController）告诉ScribbleManager的实例加载哪个特定的Scribble，比如可以通过使用索引来指定。然后ScribbleManager对象重建用于保存原ScribbleMemento对象的路径，并从文件系统将其加载为一个NSData实例。ScribbleMemento类将数据解码并重新生成ScribbleMemento实例。ScribbleManager将备忘录传递给Scribble类。Scribble类使用备忘录通过其中存储的Mark引用来恢复原先的Scribble实例。先前被归档的Scribble对象恢复后，被返回给客户端。这样周期就结束了，如图2-12所示。

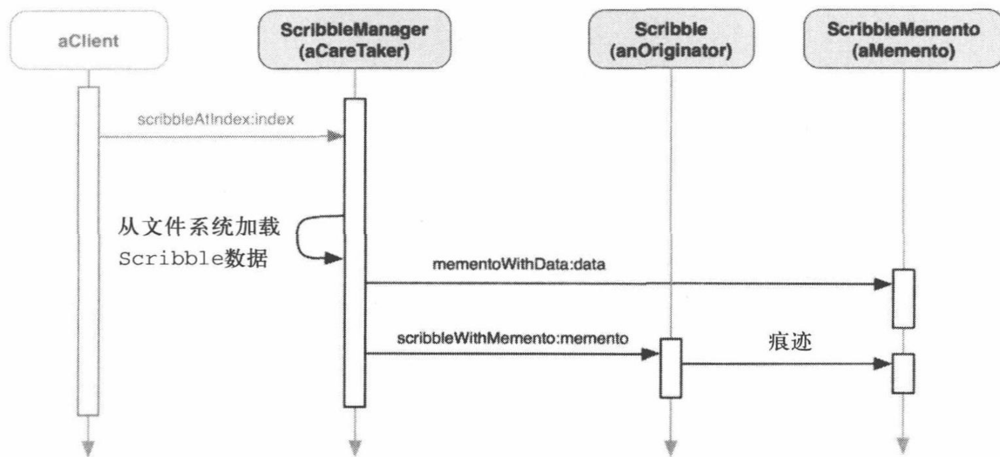


图2-12 说明如何通过ScribbleMemento对象恢复Scribble对象的时序图

此时，对于加载过程可能会有疑问：客户端如何知晓首先要加载哪个涂鸦图呢？因此下一步将创建一个视图，通过它用户可以浏览已保存的全部涂鸦图的缩略图。在用户单击之后，其中一个涂鸦图将通过CanvasView的控制器CanvasViewController，在CanvasView中打开，使用缩略图的对应索引号告诉ScribbleManager要打开哪个涂鸦图。

2.3.4 用户操作

前面几节讨论的是基础性或者说架构性的问题。还有些与应用程序的用户体验或用户期望

（user expectation）[①]相关的其他问题——比如如何浏览涂鸦缩略图的列表、涂鸦图的撤销与恢复（undo/redo）、改变线色与线宽以及删除屏幕上的当前涂鸦图。以下各小节将讨论这些问题。

1. 浏览涂鸦缩略图的列表

现在我们知道了如何保存涂鸦图及其缩略图，那么，怎样才能浏览并从中选择一个打开呢？我们来回顾一下图2-3中的缩略图视图的UI线框图；缩略图视图将全部已保存的涂鸦图显示为一个个的缩略图，以便浏览。当很多缩略图争先恐后地在视图上显示时，会让我们想到它们可能阻碍应用程序的主线程。那样就会降低用户操作的响应性。

应该采用更好的方法，让每个缩略图在后台线程加载实际的图像，而不是让主线程去一个接一个地处理大量的图像加载操作。这样，大批的缩略图各自忙于自身的处理的同时，响应性得以改善。但有个问题，缩略图自身的线程排队等待其他线程结束的时候，会怎么样呢？用户何时能看到缩略图呢？当缩略图在等待加载实际图像时，需要显示某种占位图像（placeholder image）。一旦实际图像加载完毕，就会取代占位图像进行显示。这形成了一致而可预期的用户体验。图2-13显示了操作中的缩略图视图，部分填充了已加载的涂鸦图图像。

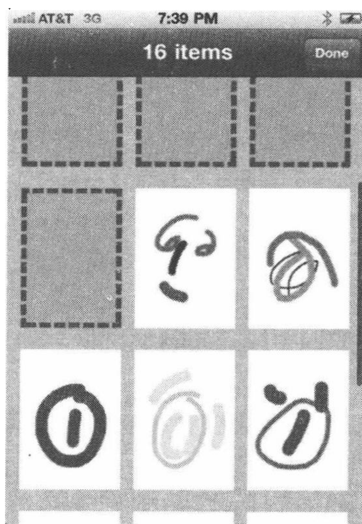


图2-13 填充了占位图像与加载完毕的涂鸦缩略图的缩略图视图

怎样才能设计一个能够做到这一点的类呢？其实，有一种叫做代理的设计模式（第22章）。代理是实际的资源或对象的占位或替代品。代理模式的一个特点是通过虚拟代理（virtual proxy）在接到请求时实现重型（heavy-weighted）资源的懒加载（lazy-load）。

需要创建一个缩略图代理类，在后台进行实际图像的某种懒加载时，它返回默认的占位图像。iOS中应用程序的一个典型例子是邮件。电子邮件的每个附件被显示为矩形的占位图像。它将在后台加载实际的附件，直到用户单击为止。程序为每个附件显示一个占位图像，以使用户不用等

① 交互设计中的概念，指用户对产品一致性的期望。——译者注

整封邮件下载完毕就能够立即阅读内容。图2-14显示了设计的概念。

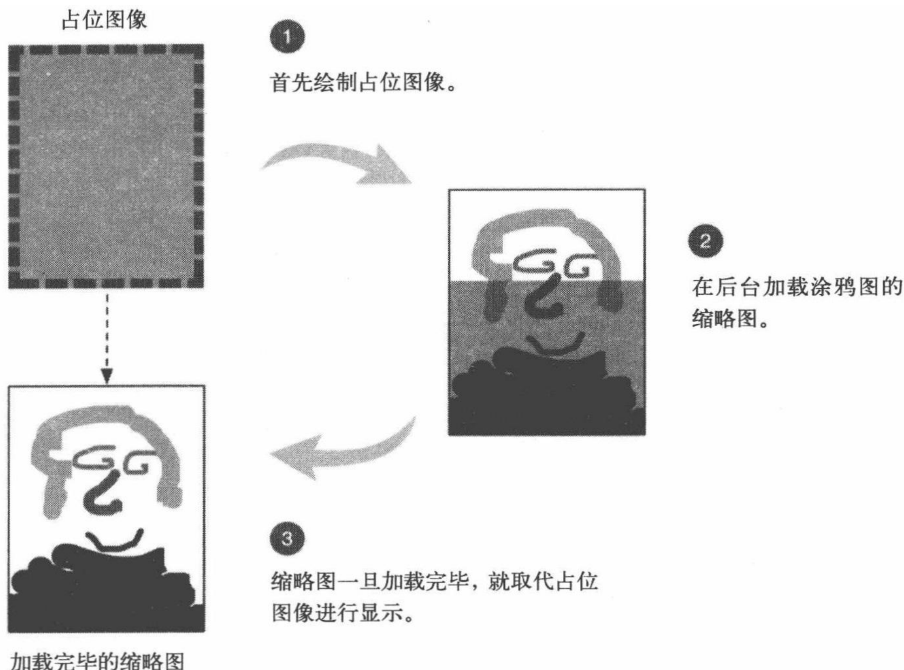


图2-14 涂鸦图的缩略图代理先显示占位图像，然后在后台加载实际缩略图。当加载过程结束后，代理用实际图像取代占位图像进行显示

可以创建一个名为ScribbleThumbnailProxy的代理类。因为它需要知道如何在视图中绘图，所以ScribbleThumbnailProxy应该是UIView的子类。ScribbleThumbnailProxy对象代表缩略图图像。每个缩略图在首次加载到缩略图视图时进行以下操作。

(1) 因为没有可用的实际图像，它在自己的drawRect:rect方法中绘制默认的占位图像。

(2) 生成一个新的线程，以文件系统中给定的位置加载实际的缩略图图像。

(3) 一旦实际图像加载完成，线程会请求自身进行重画，drawRect:rect将再次被调用。

(4) 此时，实际图像已经可用，所以它将绘制实际图像。

用户可以浏览整个缩略图视图，以选择打开哪一个涂鸦图。通过单击其中的一个，缩略图所代表的实际涂鸦图将会再次被画布视图打开，如图2-15所示。

2. 涂鸦图的撤销和恢复

需求中有一条是允许用户撤销或恢复屏幕上所画的图，每一线条或点可以被撤销或（如果必要的话）被恢复。为此，我们已经在图2-1中最初的线框图中放置了两个按钮。向左拐的按钮表示撤销，向右拐的按钮表示恢复，如图2-16所示。

要实现撤销功能，需要以某种方式记录所画的图以便从视图中将其移除，一次一个操作。同时，已被从视图中移除的图也可以在屏幕上重画。因此，即使能够把绘图机制放在实例方法或函

数中，做起来也会非常困难。这些保持绘图命令历史记录的函数和方法几乎无法有效复用。需要将绘图命令放入对象中，以便保存到历史记录，如图2-17所示。

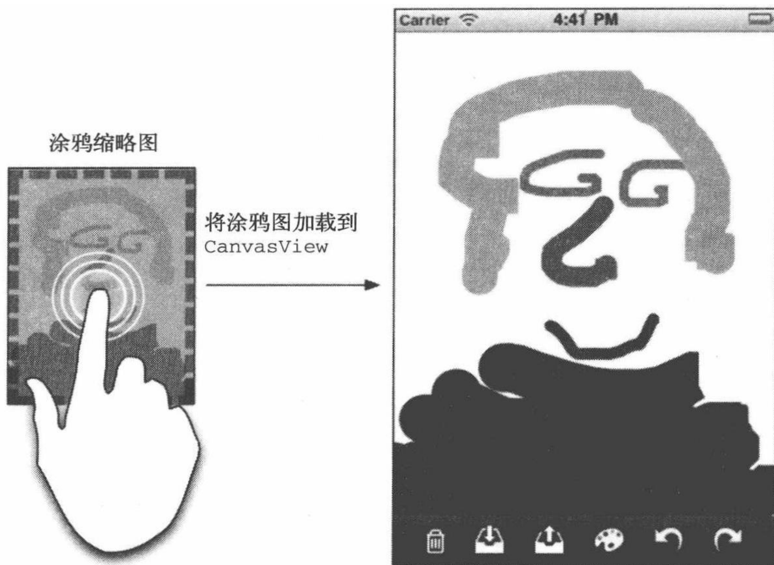


图2-15　涂鸦缩略图可被用作按钮，以触发将实际涂鸦图向画布视图的加载

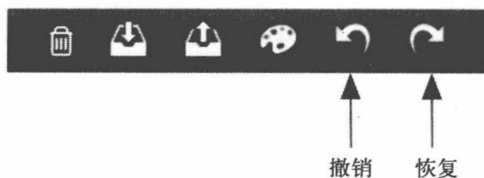


图2-16　位于画布视图工具条上的撤销与恢复按钮

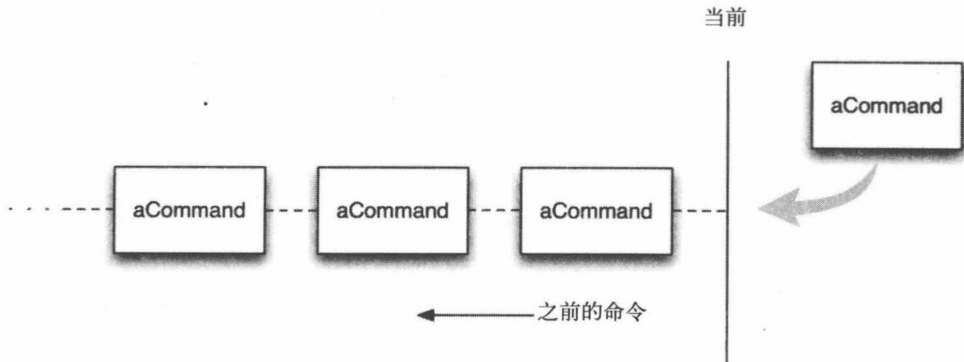


图2-17　每个命令对象被添加到命令历史记录的列表中。通过遍历这个列表，可以撤销或恢复特定的命令

每个命令有“执行”操作与“撤销”以取消操作的方法。通过遍历这个列表，可以让任意一个命令撤销或恢复其操作。可以使用索引（index）来记录当前命令。往过去调用过的命令的方向移动索引，能够逐个撤销每一个命令。同样，往现在的命令的方向移动索引，可以恢复每个命令。协助解决此类设计问题的一种设计模式称为命令模式（第20章）。

在Cocoa Touch框架中有几个用于实现命令模式的框架类，比如，`NSInvocation`的对象以目标-选择器实体的形式封装了一个命令。`NSUndoManager`实例在其撤销与恢复栈上维护一个`NSInvocation`对象的列表。一旦`NSInvocation`对象被压入`NSUndoManager`的撤销栈，就可被弹出并调用其注册的撤销操作。从撤销栈中弹出的`NSInvocation`对象会被压入恢复栈。然后可以继续撤销/恢复的循环。实现命令模式的细节将在第20章中讲解。

3. 变更线色与线宽

命令模式（第20章）不只用于实现撤销/恢复基础设施，也可用于实现任何延期命令（deferred-command）机制。在传统台式机应用程序的意义上，主菜单的每个菜单项封装了一个命令对象。当用户激活菜单项时，内嵌的命令会被调用。在加载应用程序时命令首先被分配给每个菜单项，但是命令的调用被推迟到后来菜单被激活时。在菜单设计中使用命令模式的一个好处是，菜单项中使用的命令可以在其他地方复用，比如同一命令的快捷键或者某些其他用户界面元素。

根据对`PaletteViewController`的需求，用户可以变更在`CanvasViewController`中创建下一线条或点的线宽与线色。出于这一理由与目的，我们需要一些命令来变更线宽与线色。`PaletteViewController`有3个滑动条，用户可以通过分别设置每个颜色分量——红、绿、蓝，来调节线条的颜色。我们没有创建操作单个颜色分量的单独命令，而是创建一个处理整体颜色变更的命令。

比方说，我们为此创建一个`SetStrokeColorCommand`类。根据图2-2中的线框图，有3个滑动条，各负责调节一个特定的颜色分量。当用`SetStrokeColorCommand`对象将其连接起来以后，该对象将在它的执行操作中完成工作，更新下个线条的颜色与位于滑动条正下方的调色板视图。调色板根据RGB滑动条的值反映出当前的线色。`SetStrokeColorCommand`对象成为它们的联络员，它把滑动RGB滑动条引起的任何颜色变更通知给其他观察组件。由于某种原因，我们需要有多个组件能监视颜色变化；可以让`SetStrokeColorCommand`维护多个观察者的列表。当颜色发生变化时，`SetStrokeColorCommand`可以通知其观察者，以便执行适当的操作。观察-通知或者发行-订阅模型被称为观察者模式（第12章）。让命令对象通知观察者是一个为解决设计问题而应用复合设计模式的例子。同样，可以创建类似的类`SetStrokeSizeCommand`，以更新在画布视图中创建的下一线条的宽度。

图2-18说明了`SetStrokeColorCommand`对象和`SetStrokeSizeCommand`对象如何与各种`UISlider`控件，以及`PaletteViewController`和`CanvasViewController`中的小调色板视图交互。

万一需要变更UI元素，以表现一种不同的变更线色与线宽的方法，仍然可以复用`SetStrokeColorCommand`与`SetStrokeSizeCommand`对象的内容。比如，可以用圆形的颜色选择轮代替独立的RGB滑动条，使用同一个`SetStrokeColorCommand`对象改变线色。

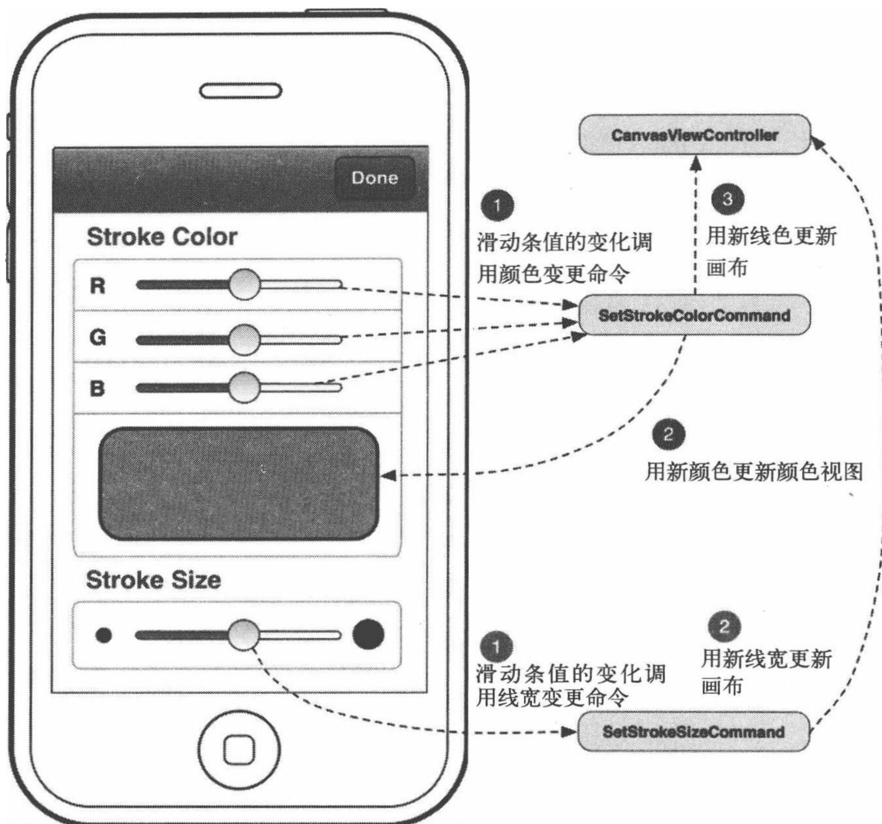


图2-18　交互流程图显示了PaletteViewController上的控件如何与各种命令对象及其目标（即PaletteViewController和CanvasViewController）交互

4. 删除屏幕上的当前涂鸦图

可以用同样的思想创建另一个命令类，并使用其对象来删除屏幕上的当前涂鸦图。有个工具条按钮绑定到了删除命令。单击了此按钮后，该命令将被执行，而且内嵌的删除当前涂鸦图数据的指令将被执行。当画布视图被此命令刷新时，原先的涂鸦图将消失，用户可以开始画新的涂鸦图。

2.4　所用设计模式的回顾

这一章内容十分丰富。学了这么多，你应该自我表扬一下！通过TouchPainter应用程序的设计过程，我们探讨了各种设计模式。有些作为单独的解决方案使用，而另一些则合并为一组复合模式。很多时候，我们会在项目中不自觉地使用它们。在此，我列出了到本章为止用到或提到的设计模式的清单。来看看你能认出多少，忘了的话可以翻回去复习：

- 中介者

- ❑ 组合
- ❑ 访问者
- ❑ 代理
- ❑ 备忘录
- ❑ 命令
- ❑ 观察者

2.5 总结

本章做了些热身，现在读者应该会迫不及待地想学习下一章，深入设计模式，以将其应用于下一个最畅销或者获奖的iOS项目中。

本书的其余部分会探索21个设计模式的细节。我们会精心制作本章使用的示例程序以及没有讲到的其他设计模式的新示例程序。还将讨论Cocoa Touch框架如何采纳了某些设计模式，以便我们免费复用而不是去重新发明轮子。

Part 2 第二部分

对象创建

本部分内容

- 第 3 章　原型
- 第 4 章　工厂方法
- 第 5 章　抽象工厂
- 第 6 章　生成器
- 第 7 章　单例

第3章

原　　型

3

在印刷术还没有普及的时代，人们常常用木头印章（后来又改用橡胶印章）往纸上印一些常用的图形和文字。多年以后人们意识到，组合各种常用印章是往纸上大量复制同一信息的一种非常容易的方法。要不是使用相同的印章来往纸上印相同的图形和文字，信息与知识的传播会远为昂贵而耗时。

在许多面向对象的应用程序中，有些对象的创建代价过大或过于复杂。要是可以重建相同的对象并作轻微的改动，事情会容易很多。典型的例子是复制组合结构（比如，树型结构）。从零开始构建一个树型组合体非常困难。我们可以通过轻微的改动重用已有的对象，以适应程序中的特定状况。

如果可以把某些对象变成橡胶印章，让其生成自身的复制品，就能节省创建它们所花的大量精力。与创建各种跟父类差异很少的独立类相比，这种做法可复用性极高并且更易于维护。

应用于“复制”操作的模式称为原型（Prototype）模式。复制（cloning）指用同一模具生产一系列的产品。模具所基于的物品称为原型。原型决定了最终产品应该是什么样子。尽管产品是用同一模具复制的，但是某些属性，如颜色与尺寸，可以稍有不同。尽管有小的差异，它们还是属于同一类。

本章先从总体上讨论对象复制的一些问题，之后讨论在Objective-C中如何实现复制复杂对象的原型模式。

3.1　何为原型模式

原型模式是一种非常简单的设计模式。客户端知道抽象`Prototype`类。在运行时，抽象`Prototype`子类的任何对象都可以按客户端的意愿被复制。因此，无需手工创建就可以制造同一类型的多个实例。说明它们之间静态关系的类图如图3-1所示。

`Prototype`声明了复制自身的接口。作为`Prototype`的子类，`ConcretePrototype`实现了`Concrete`复制自身的`clone`操作。这里的客户端通过请求原型复制其自身，创建一个新的对象。

原型模式：使用原型实例指定创建对象的种类，并通过复制这个原型创建新的对象。*

*最初的定义出现于《设计模式》（Addison-Wesley，1994）。

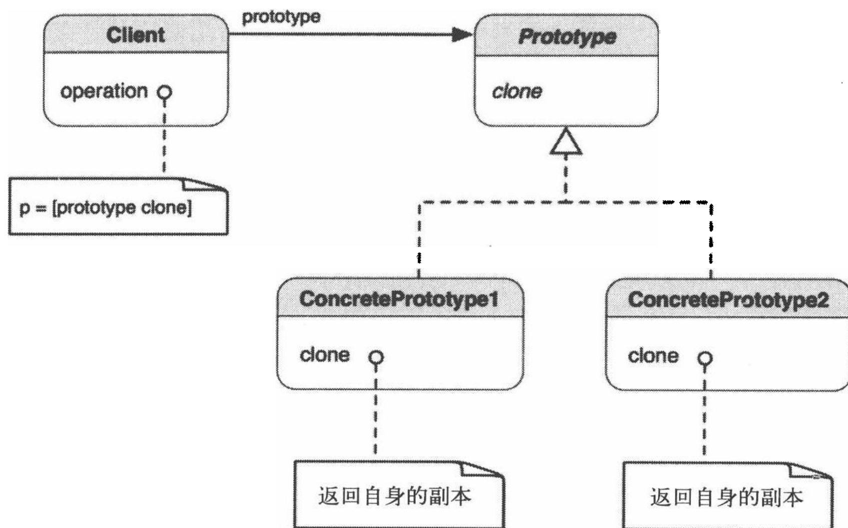


图3-1 原型模式的类图

3.2 何时使用原型模式

在以下情形，会考虑使用原型模式。

- 需要创建的对象应独立于其类型与创建方式。
- 要实例化的类是在运行时决定的。
- 不想要与产品层次相对应的工厂层次。
- 不同类的实例间的差异仅是状态的若干组合。因此复制相应数量的原型比手工实例化更加方便。
- 类不容易创建，比如每个组件可把其他组件作为子节点的组合对象。复制已有的组合对象并对副本进行修改会更加容易。

说明：有个关于使用原型模式的常见误解是，原型对象应该是典型（archetypal）对象，从不实际使用。这一误解专注于实现此模式的一种特定方式。从功能的角度来看，不管什么对象，只要复制自身比手工实例化要好，都可以是原型对象。在以下两种特别常见的情形，我们会考虑使用此模式：

(1) 有很多相关的类，其行为略有不同，而且主要差异在于内部属性，如名称、图像等；

(2) 需要使用组合（树型）对象作为其他东西的基础，例如，使用组合对象作为组件来构建另一个组合对象。

现实世界中还有许多状况应该应用这一模式。使用设计模式更像艺术行为而非科学行为。打破常规，勇于创新，更聪明地工作吧！

此模式的最低限度是生成对象的*真实副本*，以用作同一环境下其他相关事物的基础（原型）。我们接下来就讨论与对象复制相关的问题。

3.3 浅复制与深复制

如果对象有个指针型成员变量指向内存中的某个资源，那么如何复制这个对象呢？你会只是复制指针的值传给副本的新对象吗？指针只是存储内存中资源地址的占位符。在复制操作中，如果只是将指针复制给新对象（副本），那么底层的资源实际上仍然由两个实例在共享，如图3-2所示。

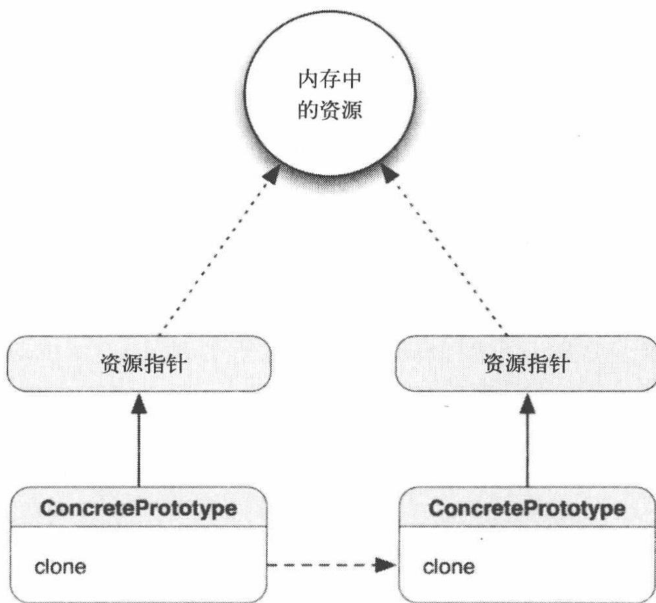


图3-2 复制ConcretePrototype的场景。只复制了资源指针，实际的资源并没有复制

在ConcretePrototype的clone操作中，把资源指针的值复制到了新的副本。尽管ConcretePrototype的实例生成了一个同类型的实例作为其副本，但两个实例的指针仍指向内存中的同一资源。因此只复制了指针值而不是实际资源，这称为浅复制。

那么什么是深复制呢？深复制是指不仅复制指针值，还复制指针所指向的资源。同一clone操作的深复制版本如图3-3所示。

clone操作不只是简单复制资源指针，还要生成内存中实际资源的真正副本。因此副本对象的指针指向了内存中不同位置的同一资源（内容）的副本。

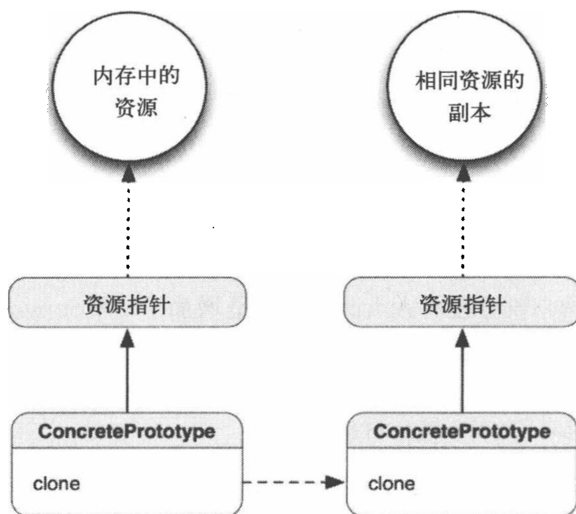


图3-3 与图3-2类似的场景。但复制过程中对内存中的实际资源进行了复制

3.4 使用 Cocoa Touch 框架中的对象复制

Cocoa Touch框架为`NSObject`的派生类提供了实现深复制的协议。`NSObject`的子类需要实现`NSCopying`协议及其方法——`(id)copyWithZone:(NSZone *)zone`。`NSObject`有一个实例方法叫做`(id)copy`。默认的`copy`方法调用`[self copyWithZone:nil]`。对于采纳了`NSCopying`协议的子类，需要实现这个方法，否则将引发异常。iOS中，这个方法保持新的副本对象，然后将其返回。此方法的调用者要负责释放返回的对象。

多数情况下，深复制的实现看起来并不很复杂。其思路是复制必需的成员变量与资源，传给此类的新实例，然后返回这个新实例。技巧在于保证确实复制了内存中的资源，而不只是指针值。在需要处理像第2章的TouchPainter应用程序中那样的组合结构时，这更为明显。

3.5 为 `Mark` 聚合体实现复制方法

在最初的TouchPainter应用程序中，有个非常重要的数据结构是包含涂鸦图所有点与线条实例的`Mark`聚合对象。至少有两种情况需要复制整个结构：

- 对其做深复制然后归档；
- 重用同一个涂鸦图作为“图样模板”（橡胶印章），重复用户所画的同一图样。

对于第一种情况，我们需要保证副本被用于归档或其他处理时不会被修改（详见第23章），而不能只是保持对原始聚合体的引用。

如果允许将同一涂鸦图重用作其他线条的基础，那么需要让当前涂鸦图复制其自身以用于“图样模板”功能。

在TouchPainter应用程序中我们将采用一种组合结构（第13章），它包含了用户的线条。作为应用程序不可或缺的部分，这个组合结构在第2章中进行了说明。结构的父类型叫做Mark，被定义为Objective-C协议。Mark有3个具体类（组件），它们在运行时定义组合对象。Dot与Vertex定义独立的组件，而Stroke定义组合组件，这些组合组件也可以包含其他Mark实例作为子节点。Dot是Vertex的子类。这里的问题是在不知道怎样解析整个树的情况下，生成整个Mark聚合结构的精确深复制。Stroke对象可以包含Dot或其子类的对象以及其他Stroke对象。需要修改最初的Mark及其实现类，添加copy方法以完成递归的深复制。图3-4为带有copy方法的Mark组合类的类图。

Mark及其具体类的接口部分没有太大改动，只是增加了一个copy方法。有一个假想的客户端用currentMark去执行savePattern，保存到称为pattern的副本中，以备以后使用。如何使用这些副本取决于应用程序，但经验法则是当对象过于复杂难以由客户端重建时，让对象来生成其自身，比如像重建已有的Mark组合对象那样。

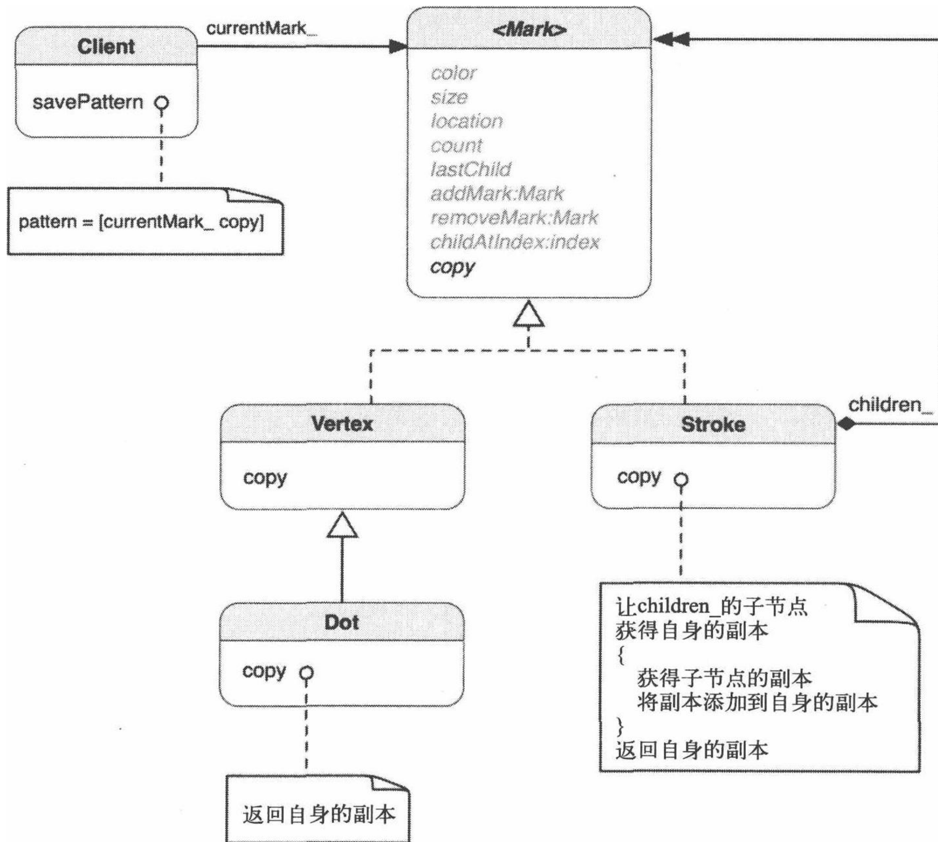


图3-4　实现原型模式的Mark组合类的类图

现在来看看代码。

代码清单3-1显示了对Mark协议的改动。

代码清单3-1 Mark.h

```
@protocol Mark <NSObject>

@property (nonatomic, retain) UIColor * color;
@property (nonatomic, assign) CGFloat size;
@property (nonatomic, assign) CGPoint location;
@property (nonatomic, readonly) NSUInteger count; // 子节点的个数
@property (nonatomic, readonly) id <Mark> lastChild;

- (id) copy;
- (void) addMark:(id <Mark>) mark;
- (void) removeMark:(id <Mark>) mark;
- (id <Mark>) childMarkAtIndex:(NSUInteger) index;

@end
```

Mark协议添加了一个新的copy方法，它返回实现类的一个实例。

有几个实现类会实现copy方法。在运行时，对象会是Mark聚合体的一部分的类有Vertex、Stroke和Vertex的子类Dot。相对于Stroke对象，Dot对象表示画在屏幕上的单个点。Stroke对象既可以是所有Mark类型的祖父节点，也可以只是普通父节点，包含Vertex对象以组成完整线条。Vertex只维护屏幕上的位置（坐标），以便显示算法将其绘制为线条（参见第13章与第15章）。

图3-5是描述这些关系的直观图。

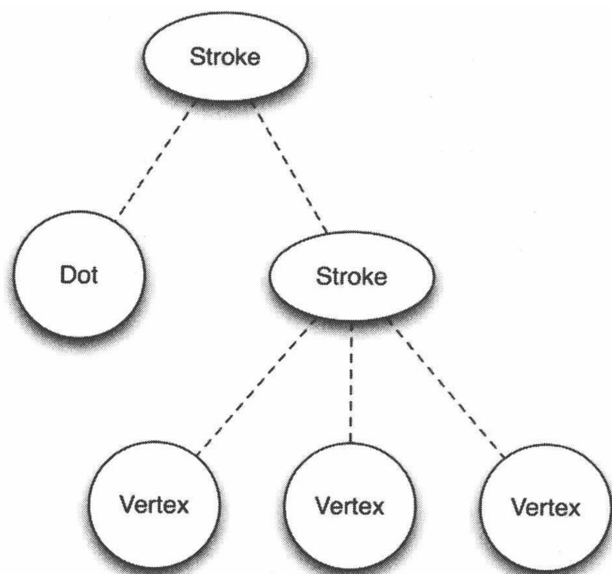


图3-5 作为组合结构的Dot、Stroke、Vertex对象之间的关系

接下来实现Mark协议的实现类的copy方法。首先是代码清单3-2中的Vertex。

代码清单3-2 Vertex.h

```
#import "Mark.h"

@interface Vertex : NSObject <Mark, NSCopying>
{
  @protected
  CGPoint location_;
}

@property (nonatomic, retain) UIColor *color;
@property (nonatomic, assign) CGFloat size;
@property (nonatomic, assign) CGPoint location;
@property (nonatomic, readonly) NSUInteger count;
@property (nonatomic, readonly) id <Mark> lastChild;

- (id) initWithLocation:(CGPoint) location;
- (void) addMark:(id <Mark>) mark;
- (void) removeMark:(id <Mark>) mark;
- (id <Mark>) childMarkAtIndex:(NSUInteger) index;

- (id) copyWithZone:(NSZone *)zone;

@end
```

等一下，为何这里不用copy方法而用copyWithZone:方法？为何在Mark协议中有copy方法而没有copyWithZone:方法？Mark协议采用了NSObject协议，而Mark的具体类采用了Mark协议并且子类化NSObject类。NSObject协议没有声明copy方法，但是NSObject声明了。NSObject型的接收器收到copy消息时，NSObject会依次向其采用了NSCopying协议的子类转发消息。子类要实现所需的在NSCopying中定义的copyWithZone:zone方法，以返回自身的副本。如果子类没有实现此方法，会抛出异常NSInvalidArgumentException的实例。这就是为什么要让Vertex类采用NSCopying协议并为复制处理实现其copyWithZone:zone方法。然而，NSObject协议没有声明copy方法，所以在Mark协议中声明它，以避免编译警告。Vertex是NSObject的子类，所以它实现了copyWithZone:方法，见代码清单3-3。

代码清单3-3 Vertex.m

```
#import "Vertex.h"

@implementation Vertex
@synthesize location=location_;
@dynamic color, size;

- (id) initWithLocation:(CGPoint) aLocation
{
  if (self = [super init])
  {
    [self setLocation:aLocation];
  }

  return self;
```

```
}

// 默认属性什么也不做
- (void) setColor:(UIColor *)color {}
- (UIColor *) color { return nil; }
- (void) setSize:(CGFloat)size {}
- (CGFloat) size { return 0.0; }

// Mark操作什么也不做
- (void) addMark:(id <Mark>) mark {}
- (void) removeMark:(id <Mark>) mark {}
- (id <Mark>) childMarkAtIndex:(NSUInteger) index { return nil; }
- (id <Mark>) lastChild { return nil; }
- (NSUInteger) count { return 0; }
- (NSEnumerator *) enumerator { return nil; }

#pragma mark -
#pragma mark NSCopying method

// 此方法需要实现，以支持备忘录
- (id)copyWithZone:(NSZone *)zone
{
  Vertex *vertexCopy = [[[self class] allocWithZone:zone]initWithLocation:location_];

  return vertexCopy;
}

@end
```

在重载的copyWithZone:方法中，使用[[self class] allocWithZone:zone]生成新的Vertex实例，并用当前位置进行初始化。使用[self class]是因为我们希望其子类也能够复用这个复制方法。要是直接用[Vertex alloc]，其子类就只会返回Vertex而不是它的实际类型的副本。Vertex只实现了location属性，因此新副本用当前位置进行了初始化之后，就返回vertexCopy。

Vertex对象用于组成线条，并不包含颜色、大小等其他信息。但是当用户在屏幕上创建了一个点，需要另一种数据结构包含颜色与大小，以表示这个点。除了颜色和大小，位置对独立的点也至关重要。因此我们创建了Vertex的子类Dot，见代码清单3-4。

代码清单3-4 Dot.h

```
#import "Vertex.h"

@interface Dot : Vertex
{
  @private
  UIColor *color_;
  CGFloat size_;
}

@property (nonatomic, retain) UIColor *color;
@property (nonatomic, assign) CGFloat size;
```

```
- (id) copyWithZone:(NSZone *)zone;

@end
```

与Vertex类似，它实现了copyWithZone:方法，与协议一起支持复制。这个方法的实现见代码清单3-5。

代码清单3-5 Dot.m

```
#import "Dot.h"

@implementation Dot
@synthesize size=size_, color=color_;

- (void) dealloc
{
  [color_ release];
  [super dealloc];
}

#pragma mark -
#pragma mark NSCopying delegate method

- (id)copyWithZone:(NSZone *)zone
{
  Dot *dotCopy = [[[self class] allocWithZone:zone] initWithLocation:location_];

  // 复制color
  [dotCopy setColor:[UIColor colorWithCGColor:[color_ CGColor]]];

  // 复制size
  [dotCopy setSize:size_];

  return dotCopy;
}

@end
```

第一条初始化语句几乎和Vertex相同。Dot支持另外两个属性类型，因此除location之外，还需要用自己的私有变量color_与size_来设置color与size属性。location与size可以直接赋值给副本，无需多想。但是这里的color需要注意。需要用color_变量中已有的CGColor值生成一个UIColor。最后方法返回dotCopy。

接下来看看Stroke类，见代码清单3-6。

代码清单3-6 Stroke.h

```
#import "Mark.h"

@interface Stroke : NSObject <Mark, NSCopying>
{
  @private
```

```
  UIColor *color_;
  CGFloat size_;
  NSMutableArray *children_;
}

@property (nonatomic, retain) UIColor *color;
@property (nonatomic, assign) CGFloat size;
@property (nonatomic, assign) CGPoint location;
@property (nonatomic, readonly) NSUInteger count;
@property (nonatomic, readonly) id <Mark> lastChild;

- (void) addMark:(id <Mark>) mark;
- (void) removeMark:(id <Mark>) mark;
- (id <Mark>) childMarkAtIndex:(NSUInteger) index;
- (id) copyWithZone:(NSZone *)zone;

@end
```

然后再看看它的实现，见代码清单3-7。

代码清单3-7 Stroke.m

```
#import "Stroke.h"

@implementation Stroke

@synthesize color=color_, size=size_;
@dynamic location;

- (id) init
{
  if (self = [super init])
  {
    children_ = [[NSMutableArray alloc] initWithCapacity:5];
  }

  return self;
}

- (void) setLocation:(CGPoint)aPoint
{
  // 不做任何位置设定
}

- (CGPoint) location
{
  // 返回第1个子节点的位置
  if ([children_ count] > 0)
  {
    return [[children_ objectAtIndex:0] location];
  }

  // 否则，返回原点
  return CGPointZero;
```

```
}

- (void) addMark:(id <Mark>) mark
{
[children_ addObject:mark];
}

- (void) removeMark:(id <Mark>) mark
{
  // 如果mark在这一层，将其移除并返回
  // 否则，让每个子节点去找它
  if ([children_ containsObject:mark])
  {
     [children_ removeObject:mark];
  }
  else
  {
     [children_ makeObjectsPerformSelector:@selector(removeMark:)
                                withObject:mark];
  }
}

- (id <Mark>) childMarkAtIndex:(NSUInteger) index
{
  if (index >= [children_ count]) return nil;

   return [children_ objectAtIndex:index];
}

// 返回最后子节点的便利方法
- (id <Mark>) lastChild
{
  return [children_ lastObject];
}

// 返回子节点数
- (NSUInteger) count
{
  return [children_ count];
}

- (void) dealloc
{
   [color_ release];
   [children_ release];
   [super dealloc];
}

#pragma mark -
#pragma mark NSCopying method

- (id)copyWithZone:(NSZone *)zone
{
  Stroke *strokeCopy = [[[self class] allocWithZone:zone] init];
```

```
    // 复制color
    [strokeCopy setColor:[UIColor colorWithCGColor:[color_ CGColor]]];

    // 复制size
    [strokeCopy setSize:size_];

    // 复制children
    for (id <Mark> child in children_)
    {
      id <Mark> childCopy = [child copy];
      [strokeCopy addMark:childCopy];
      [childCopy releaseCopy];
    }

    return strokeCopy;
}

@end
```

Stroke类跟Vertex和Dot一样实现了copyWithZone:方法。但是对于Stroke，不只需要复制Mark中定义的属性，还需要复制其子节点（前面代码中的for循环）。children_集合变量中的每个子节点，需要首先生成自身的副本，然后Stroke对象会将其添加到复制的strokeCopy中，成为新的子节点。完成了所有复制与设定后，会返回strokeCopy作为原Stroke对象的副本。

你也许已注意到，Mark协议中声明的location属性现在在Stroke的@implementation部分被定义为@dynamic。线条不保持自己在视图上的位置，只返回第一个子节点的位置（如果有子节点的话）。Location属性有两个重载的存取方法（accessor method）可用于处理这一特殊情况。setLocation:什么也不做，而location返回线条的第一个子节点的屏幕坐标。与@synthesize不同，我们使用@dynamic来告知编译器不要生成这些存取方法，因为我们创建了自己的版本。组合结构的细节将在第13章中进行讨论。

大家知道Stroke对象可以包含任何Mark引用作为子节点。Vertex是个特别的类型，其目的在于为绘图算法提供位置信息。Vertex是Dot的超类。Dot已经有了copyWithZone:方法的实现，见代码清单3-3。

现在搞定了复制的把戏，可以开始讨论如何使用原型模式来实现“图样模板”功能了。

3.6 将复制的Mark用作“图样模板”

现在Mark的每个子类都可复制，且能返回其真正副本。对象复制的一个重要应用是保存其状态。或者说，在对象被复制的那一刻，对象的样子的快照。在什么时候它需要被复制呢？撤销与恢复时。执行操作之前，对象（例如文档）的状态需要以某种方式保存下来。然后这个状态被压入撤销栈。在需要撤销操作时，对象（文档）的上一个状态被恢复到应用程序。大多数情况下，对象复制复杂而容易出错。若由客户端来处理此过程，每当目标类（比如文档）有改动，客户端的代码可能就需要修改。说到对象复制，我们想到的也许只是某些只有几个成员变量与接口的简

单对象。然而，在需要处理组合对象时，情况可能相当棘手。复制那样的组合对象可能非常复杂。组合对象可能非常庞大而难以处理。可以想象一下遍历整个结构并为每一节点生成带有相应属性的新实例。更糟的是，我们并不知道结构中每个节点的确切类型，这使得从外部生成这些对象更加复杂。在Mark家族中实现原型不只是此类情况下的救急措施，对本节要讨论的“图样模板”功能而言，它更是必不可少。

我们想复用特定的线条，把它当做“图样模板”应用到CanvasView，作为绘制不同图形的基础。这里的想法是复制特定的Mark（Stoke或Dot）并粘贴到CanvasView，这样用户就能反复重用同一图样。图3-6为基于此想法的一个样图。

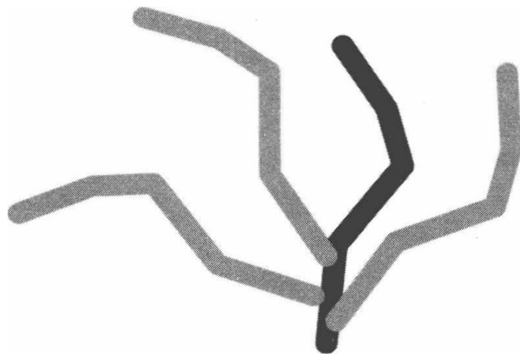


图3-6　由原始线条与复制线条组成的图案。黑色线条表示原始线条，灰色线条是黑色线条的副本

黑色线条是先前保存的原始线条，用作图样模板。灰色的是其副本。黑色线条与灰色线条形状相同。复制的线条可以有不同的属性，如颜色、位置，甚至可以变换形态。用户可以选择屏幕上任何线条，把它设为图样模板，以备后来复用。每当用户应用图样，应用程序从原始线条生成一个副本，并将其放在CanvasView上。然后用户可以修改线条副本的属性，如颜色、大小、位置。由原始线条及其副本组成的最终结构可被保存为另一个图样模板。这将有无限的可能性。

我们不会太深入这一功能，但会用代码段讨论其构思。我把示例项目中这个部分作为练习留给读者去实现。

假定用户选择了一个叫做selectedMark的Mark实例作为图样模板。我们对其进行复制并保存到称为templateArray的数据结构中，如以下代码段所示。

```
id <Mark> patternTemplate = [selectedMark copy];

// 把patternTemplate保存到一个数据结构
// 以备使用
[templateArray addObject:patternTemplate];
```

此时，我们不清楚用户究竟选择了什么，只知道这是Mark聚合体的一个实例，可能只有一个Dot，也可能有多个Stroke，每个包含多个Vertex。随着一个简单而奇妙的copy消息发给selectedMark，我们得到了它完美的副本。当用户要把先前保存的一个图样模板应用到

CanvasView的时候，我们需要使用用户提供的patternIndex从templateArray中将其取出，并添加到当前Mark组合体。于是，新的Mark变成了当前Mark的组成部分，如下面的代码段所示。

```
id <Mark> patternClone = [templateArray objectAtIndex:patternIndex];
[currentMark addMark:patternClone];
[canvasView setMark:currentMark];
[canvasView setNeedsDisplay];
```

当用更新了的Mark实例设定canvasView之后，我们向它发送setNeedsDisplay消息，把Mark绘制到屏幕上。

显然，不能原封不动地把线条的副本添加到屏幕上，至少要修改它的位置，否则如果原来的线条还在那里，它就会将其盖住。但这个代码段已经相当清楚地展示了如何使用在Mark中实现的原型模式，来让我们的TouchPainter应用程序更加迷人。

3.7 总结

本章探讨了如何使用原型模式来实现TouchPainter应用程序中Mark组合结构的复制。对于包含了树型结构的Mark聚合体，当需要生成真正副本的时候，必须要实现深复制。第23章中讨论的备忘录模式的例子，保持了整个Mark聚合体的一个副本，然后将其保存到备忘录对象。

原型模式是用于对象创建的非常简单的模式。下一章会介绍另一个对象创建模式，它并不使用copy方法创建同一类型的对象，而是使用一个决定生成何种对象的方法。

第4章

工厂方法

4

几乎在每个用面向对象语言写的应用程序里都能看到工厂方法。工厂方法模式是抽象工厂模式（第5章）的组成部分。各种具体工厂重载其抽象工厂父类中定义的工厂方法，并用这个重载的工厂方法创建自己的产品（对象）。

对象工厂与生产有形产品的真实工厂类似，例如，制鞋厂生产鞋，手机工厂生产手机。比方说，你让工厂给你生产些产品，你给它们发送一个“生产产品”的消息。制鞋厂和手机工厂都按照相同的“生产产品”的协议，启动其生产线。过程结束后，每个厂家都返回所生产的特定类型的产品。我们把“生产”这个有魔力的词称作工厂方法，因为它是命令生产者（工厂）得到想要的产品的方法。

生产者自身不必是抽象工厂，它可以是任何类。要点在于不是直接创建对象，而是使用类或对象的工厂方法创建具体产品，并以抽象类型返回。这样做好在哪里？我们将在下面详细讨论。

4.1 何为工厂方法模式

工厂方法也称为虚构造器（virtual constructor）。它适用于这种情况：一个类无法预期需要生成哪个类的对象，想让其子类来指定所生成的对象。

工厂方法模式的静态类结构如图4-1所示。

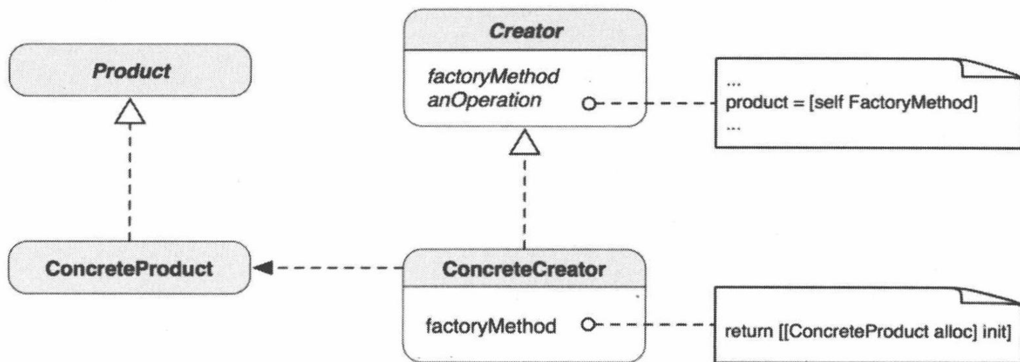


图4-1　工厂方法模式的类图

抽象的Product（产品）定义了工厂方法创建的对象的接口。ConcreteProduct实现了Product接口。Creator定义了返回Product对象的工厂方法。它也可以为工厂方法定义一个默认实现，返回默认ConcreteProduct对象。Creator的其他操作可以调用此工厂方法创建Product对象。ConcreateCreator是Creator的子类。它重载了工厂方法，以返回ConcreteProduct的实例。

> **工厂方法模式**：定义创建对象的接口，让子类决定实例化哪一个类。工厂方法使得一个类的实例化延迟到其子类。*
>
> *最初的定义出现于《设计模式》(Addison-Wesley，1994)。

工厂方法的最初定义好像专注于让子类决定创建什么对象。有一种变体，抽象类使用工厂方法创建其私有子类或任何其他类的对象。本章稍后将讨论这种变体。

4.2 何时使用工厂方法

在以下情形，你自然会想到使用工厂方法模式：

- 编译时无法准确预期要创建的对象的类；
- 类想让其子类决定在运行时创建什么；
- 类有若干辅助类为其子类，而你想将返回哪个子类这一信息局部化。

使用这一模式的最低限度是，工厂方法能给予类在变更返回哪一种对象这一点上更多的灵活性。使用这一架构的一个常见例子是Cocoa Touch框架（或一般的Cocoa）中的NSNumber。尽管可以使用常见的alloc init两步法创建NSNumber实例，但这没什么用，除非使用预先定义的类工厂方法来创建有意义的实例。例如，[NSNumber numberWithBool:YES]消息会得到NSNumber的子类NSCFBoolean的一个实例，这个实例包含传给类工厂方法的布尔值。工厂方法模式对框架设计者特别有用。我们将在4.5节进一步讨论Cocoa Touch框架中的工厂方法。

4.3 为何这是创建对象的安全方法

与直接创建新的具体对象相比，使用工厂方法创建对象可算作一种最佳做法。工厂方法模式让客户程序可以要求由工厂方法创建的对象拥有一组共同的行为。所以往类层次结构中引入新的具体产品并不需要修改客户端代码，因为返回的任何具体对象的接口都跟客户端一直在用的从前的接口相同。

4.4 在 TouchPainter 中生成不同画布

第2章讨论的TouchPainter应用程序，需要一个画布视图让用户用手指绘图。这个画布视图与绘图算法一起渲染（render）任何由触摸生成的涂鸦图（参见第13章与第15章）。画布类称为CanvasView，是UIView的子类。我们打算把这个类用作最上层的抽象类，用几个子类来扩展它，

为用户提供更多画布类型。会有两种画布类型可供选择，ClothCanvasView与PaperCanvasView。ClothCanvasView将有布质风格的背景，而PaperCanvasView将有再生纸风格的背景。两者都会向画布增加几个特定的行为。然而，我们不打算讨论其细节，而把重点放在如何应用这一模式在运行时生成画布上。

我们知道要往原来的CanvasView添加两种画布类型，但是不打算直接对其中任何一种类型进行实例化。否则，只要添加画布类型，甚至只是修改其初始化接口，就需要修改客户端代码以反映这些修改。相反，对于CanvasView与CanvasViewGenerator，我们会让客户端使用最上层的类，以消除与返回的特定CanvasView的耦合。

特定CanvasView的实例由在CanvasViewGenerator抽象类中定义的工厂方法canvas-ViewWithFrame:aFrame来创建。有两个CanvasViewGenerator的子类，各自通过重载父类的canvasViewWithFrame方法负责创建特定CanvasView的实例。PaperCanvasViewGenerator将创建PaperCanvasView的实例，而ClothCanvasViewGenerator将创建ClothCanvasView的实例。它们关系如图4-2所示。

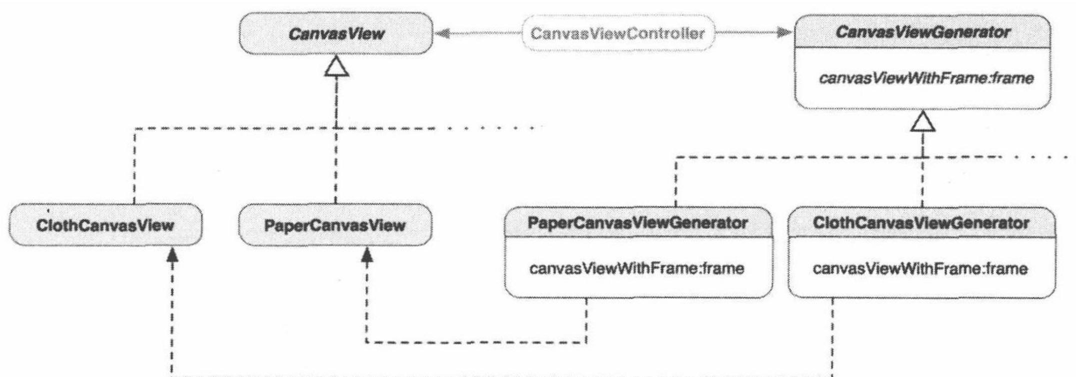


图4-2 表示ClothCanvasView、PaperCanvasView及其相应生成器的平行类层次的类图

CanvasView的控制器CanvasViewController，将是CanvasView与CanvasView-Generator的客户端。CanvasViewController将使用CanvasViewGenerator的实例，按照用户的喜好得到正确的CanvasView实例的引用。可以用不同类型的CanvasViewGenerator为CanvasViewController生成不同的画布。

最上层的CanvasView定义了任意CanvasView类型的默认行为。其子类用不同图像在屏幕上展现各种纹理并展现其他可能的特定行为。PaperCanvasView与ClothCanvasView的代码段见代码清单4-1。

代码清单4-1 PaperCanvasView.h

```
#import <UIKit/UIKit.h>
#import "CanvasView.h"

@interface PaperCanvasView : CanvasView
```

```
{
  // 一些私有变量
}

// 其他一些特定行为
@end
```

代码清单4-1显示了PaperCanvasView的类声明，省略了一些行为与私有成员变量。假定有个特定行为是它有自己的图像视图，该视图显示了纸的纹理图像，如代码清单4-2。

代码清单 4-2 PaperCanvasView.m

```
#import "PaperCanvasView.h"

@implementation PaperCanvasView

- (id)initWithFrame:(CGRect)frame
{
  if ((self = [super initWithFrame:frame]))
  {
    // 在上面添加一个纸图像视图作为画布背景
    UIImage *backgroundImage = [UIImage imageNamed:@"paper"];
    UIImageView *backgroundView = [[[UIImageView alloc]
                                     initWithImage:backgroundImage]
                                     autorelease];
     [self addSubview:backgroundView];
  }

  return self;
}

// 其他行为的实现

@end
```

@"paper"是纸纹理的图像的名字，它被赋给UIImageView的实例backgroundView。然后backgroundView作为子视图添加到主内容视图，以显示纸的图像。同样地，我们在代码清单4-3中声明了ClothCanvasView。

代码清单4-3 ClothCanvasView.h

```
#import <UIKit/UIKit.h>
#import "CanvasView.h"

@interface ClothCanvasView : CanvasView
{
  // 一些私有变量
}

// 其他一些特定行为

@end
```

与PaperCanvasView类似，限于篇幅，这里也省略了ClothCanvasView的一些可能的特定

行为。ClothCanvasView也显示不同的图像纹理作为其独特行为之一，见代码清单4-4。它的纹理图像叫做@"cloth"，构造方式几乎与代码清单4-2中的PaperCanvasView相同。

代码清单4-4 ClothCanvasView.m

```
#import "ClothCanvasView.h"

@implementation ClothCanvasView

- (id)initWithFrame:(CGRect)frame
{
  if ((self = [super initWithFrame:frame]))
  {
    // 在上面添加一个布图像视图作为画布背景
    UIImage *backgroundImage = [UIImage imageNamed:@"cloth"];
    UIImageView *backgroundView = [[[UIImageView alloc]
                                     initWithImage:backgroundImage]
                                    autorelease];
     [self addSubview:backgroundView];
  }

  return self;
}

// 其他行为的实现

@end
```

PaperCanvasView与ClothCanvasView的实现非常易懂。它们继承了CanvasView类（可能也包括其他一些类）中定义的默认行为。现在我们定义好了产品，还需要为它们各自定义一个生成器。代码清单4-5和代码清单4-6中的代码段展示了CanvasViewGenerator抽象类的实现。

代码清单4-5 CanvasViewGenerator.h

```
#import "CanvasView.h"

@interface CanvasViewGenerator : NSObject
{

}

- (CanvasView *) canvasViewWithFrame:(CGRect) aFrame;

@end
```

代码清单4-6 CanvasViewGenerator.m

```
#import "CanvasViewGenerator.h"

@implementation CanvasViewGenerator

- (CanvasView *) canvasViewWithFrame:(CGRect) aFrame
{
```

```
    return [[[CanvasView alloc] initWithFrame:aFrame] autorelease];
  }

  @end
```

CanvasViewGenerator有一个方法canvasViewWithFrame:(CGRect) aFrame。这个方法的默认实现只是创建并返回无图案的CanvasView。这个生成器的子类需要重载这个方法，返回CanvasView的实际具体类型（如PaperCanvasViewGenerator）见代码清单4-7与代码清单4-8中的代码段。

代码清单4-7 PaperCanvasViewGenerator.h

```
#import "CanvasViewGenerator.h"
#import "PaperCanvasView.h"

@interface PaperCanvasViewGenerator : CanvasViewGenerator
{

}

- (CanvasView *) canvasViewWithFrame:(CGRect) aFrame;

@end
```

代码清单4-8 PaperCanvasViewGenerator.m

```
#import "PaperCanvasViewGenerator.h"

@implementation PaperCanvasViewGenerator

- (CanvasView *) canvasViewWithFrame:(CGRect) aFrame
{
  return [[[PaperCanvasView alloc] initWithFrame:aFrame] autorelease];
}

@end
```

PaperCanvasViewGenerator重载canvasViewWithFrame:方法，返回PaperCanvasView的实例。

返回ClothCanvasView实例的生成器定义在代码清单4-9和代码清单4-10中。

代码清单4-9 ClothCanvasViewGenerator.h

```
#import "CanvasViewGenerator.h"
#import "ClothCanvasView.h"

@interface ClothCanvasViewGenerator : CanvasViewGenerator
{

}

- (CanvasView *) canvasViewWithFrame:(CGRect) aFrame;

@end
```

代码清单4-10 ClothCanvasViewGenerator.m

```
#import "ClothCanvasViewGenerator.h"

@implementation ClothCanvasViewGenerator

- (CanvasView *) canvasViewWithFrame:(CGRect) aFrame
{
  return [[[ClothCanvasView alloc] initWithFrame:aFrame] autorelease];
}

@end
```

使用画布

CanvasViewController现在使用原来的CanvasView。为了在运行时加以改变，需要在CanvasViewController中添加一个方法，通过CanvasViewGenerator取得CanvasView的实例，见代码清单4-11。

代码清单4-11 CanvasViewController.h

```
#import "CanvasView.h"
#import "CanvasViewGenerator.h"

@interface CanvasViewController : UIViewController
{
  @private
  CanvasView *canvasView; // 画布视图
}

@property (nonatomic, retain) CanvasView *canvasView;

- (void) loadCanvasViewWithGenerator:(CanvasViewGenerator *)generator;

@end
```

CanvasViewController有个叫loadCanvasViewWithGenerator:(CanvasView Generator*) generator的新方法，该方法接受CanvasViewGenerator的实例作为参数，并让它返回CanvasView的实例，返回的实例将在控制器中使用。这个方法的实现见代码清单4-12。

代码清单4-12 CanvasViewController.m

```
#import "CanvasViewController.h"

@implementation CanvasViewController

@synthesize canvasView=canvasView_;

// 实现viewDidLoad，进行视图加载后的追加设置，
// 通常视图是从nib加载。
- (void)viewDidLoad
```

```
{
  [super viewDidLoad];

  // 使用CanvasViewGenerator的工厂方法取得默认画布视图
  CanvasViewGenerator *defaultGenerator = [[[CanvasViewGenerator alloc] init]
                                            autorelease];
  [self loadCanvasViewWithGenerator:defaultGenerator];
}

// 限于篇幅，略去了无关的方法

#pragma mark -
#pragma mark Loading a CanvasView from a CanvasViewGenerator

- (void) loadCanvasViewWithGenerator:(CanvasViewGenerator*)generator
{
  [canvasView_ removeFromSuperview];
  CGRect aFrame = CGRectMake(0, 0, 320, 436);
  CanvasView *aCanvasView = [generator canvasViewWithFrame:aFrame];
  [self setCanvasView:aCanvasView];
  [[self view] addSubview:canvasView_];
}

@end
```

添加新视图之前，首先让canvasView_（保持当前CanvasView实例的成员变量）把自己从它的超视图（superview）中删除。然后，指定边框尺寸参数向generator发送canvasViewWith-Frame:aFrame消息，取得CanvasView的实例。根据生成器的类型，将返回合适的CanvasView实例。使用存取器方法setCanvasView:把刚返回的aCanvasView赋给成员变量canvasView_。为什么呢？因为返回的aCanvasView被自动释放，我们要将其保持在控制器中。设定成员变量应该使用存取器方法而不是手动发出retain消息。canvasView属性（property）定义了retain特性（attribute），所以把aCanvasView设置给canvasView属性就行了。然后把更新后的canvasView_（现在是aCanvasView）作为新的子视图添加回控制器的主视图。

因此，以后用户选择特定的画布类型时，应用程序会把具体生成器的实例传给loadCanvas-ViewWithGenerator:方法，原来的画布将被新画布所替换。

4.5 在 Cocoa Touch 框架中应用工厂方法

工厂方法在Cocoa Touch框架中几乎随处可见。大家已经知道常见的两步对象创建法[[SomeClass alloc] init]。有时，我们已注意到有一些“便利”方法返回类的实例。例如，NSNumber有很多numberWith*方法；其中有两个是numberWithBool:和numberWithChar:。它们是类方法，也就是说我们向NSNumber发送[[NSNumber numberWithBool:bool]与[[NSNumber numberWithChar:char]，以获得与传入参数同类型的各种NSNumber实例。与如何创建NSNumber的具体子类型的实例有关的所有细节，都由NSNumber的类工厂方法负责。[[NSNumber numberWithBool:bool]的情况是，方法接受值bool，并把NSNumber的内部子

类的一个实例初始化，让它能够反映传入的值bool。

我们曾提到有个工厂方法模式的变体，抽象类用它生成具体子类。NSNumber中的这些numberWith*方法是这个变体的一个例子。它们不是用来被NSNumber的私有子类重载的，而是NSNumber创建合适对象的便利方式。前面提到的TouchPainter应用程序中，CanvasView-Generator需要由某些其他类来创建。就NSNumber而言，没有可在别处生成的其他“数字生成器”，而是在类级别打包提供了方法以达到类似的效果。它们称作类工厂方法。

在TouchPainter例子中实现的工厂方法模式可以用这种变体进行简化，打包提供一组返回不同具体CanvasView类型的类工厂方法。唯一问题在于，相对于由用户选择并返回的生成器，客户端(CanvasViewController)现在需要（通过明确的类工厂方法）明确知道自己想要什么。

4.6 总结

工厂方法是面向对象软件设计中应用非常普遍的设计模式。工厂方法从代码中消除了对应用程序特有类的耦合。代码只需处理Product抽象接口。所以同一代码得以复用，在应用程序中与用户定义的任何ConcreteProcuct类一起工作。我们使用工厂方法模式协助实现了TouchPainter应用程序，使其支持多种CanvasView以供用户选择。

在下一章，将接触到与工厂方法模式密切相关另一种对象创建模式。它们非常相似，总是令人混淆。

第5章

抽象工厂

5

很多人喜欢吃比萨饼。虽然每家比萨饼店都可以制作自己的比萨，但常见比萨饼的结构几乎已经标准化。标准比萨饼有浇头（topping）、奶酪、酱、面饼。比萨饼的风味有很多种。比如，纽约风味和芝加哥风味比萨的配料就有以下几点不同：薄皮面饼和厚皮面饼，番茄大葱酱和李子番茄酱，帕马森干酪和莫萨里拉奶酪。假设你去了一家比萨饼店，里面有两位厨师，分别来自纽约与芝加哥，都擅长各自风味的比萨饼。这一回你点了一份纽约风味意大利辣肠比萨饼。在后面的厨房，来自纽约的厨师已经开始为你下的单子准备配料了，薄皮面饼、番茄大葱酱、帕马森干酪和一些意大利辣肠。下一回你又去这一家比萨饼店，点了同一种比萨饼但要的是芝加哥风味的。然后来自芝加哥的厨师会准备配料，厚皮面饼、李子番茄酱、莫萨里拉奶酪和意大利辣肠。尽管你点了同一种比萨饼（意大利辣肠），配料的特点却因风味而略有不同。

尽管它们看上去是不同的比萨饼，但至少它们都拥有比萨饼该有的基本特征。所以从高层视角来看，“比萨饼”就像是一种食物，或者你可以直接称它为抽象食物类型。抽象比萨饼类型有些基本的要求，如浇头、面饼、奶酪、酱，不管实际上这些材料到底是什么。比萨饼厨师就像生产某类产品的工厂，但实际产品的样式与属性可以不同。所有比萨饼厨师都知道同样的“通用”或“抽象”的烤比萨饼的知识，尽管比萨饼的最终风味取决于实际制作的厨师。来至纽约和芝加哥的厨师就是“实际”或“具体”比萨饼厨师。他们以自己特定风味生产比萨饼。我们消费者不在乎比萨饼是“怎样”做出来的，只要比萨饼好吃。

在软件设计中，如果客户端想手工创建一个类的对象，那么客户端需要首先知道这个类的细节。更糟的是，一组相关的对象可以在运行时按不同的标准创建得不一样，此时客户端就需要知道全部细节才能创建它们。可以通过抽象工厂方法来解决这个问题。

抽象工厂提供一个固定的接口，用于创建一系列有关联或相依存的对象，而不必指定其具体类或其创建的细节。客户端与从工厂得到的具体对象之间没有耦合。图5-1显示了一系列工厂及其产品间的可能关系。

如图5-1所示，Client只知道`AbstractFactory`和`AbstractProduct`。每个工厂类中，结构与实际操作的细节按黑箱对待。甚至产品也不知道谁将负责创建它们。只有具体工厂知道为客户端创建什么、如何创建。这个模式有趣的一点是，很多时候它都用工厂方法模式（第5章）来实现。工厂方法把实际的创建过程推迟到重载它的子类中。在类图中，方法`createProductA`和`createProductB`是工厂方法。最初的抽象方法什么也不创建。这种抽象非常通用，广泛用于任

何需要抽象创建过程的场合。抽象工厂模式常与原型模式（第3章）、单例模式（第7章）和享元模式（第21章）等其他设计模式一起使用。

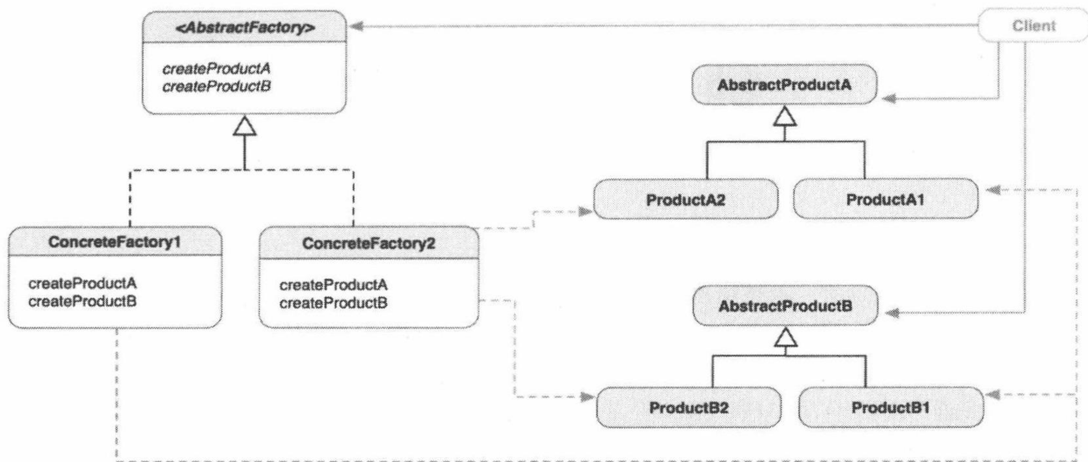


图5-1　表示一系列工厂及其相关产品之间关系的类图

本章将用抽象工厂方法模式来扩展第2章中的TouchPainter应用程序，同时也将讨论Cocoa Touch框架中常见的此种模式。

> **抽象工厂**：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。*
> * 最初的定义出现于《设计模式》(Addison-Wesley，1994)。

抽象工厂与工厂方法

抽象工厂与工厂方法模式在许多方面都非常相似。很多人常常搞不清应该在什么时候用哪一个。两个模式都用于相同的目的：创建对象而不让客户端知晓返回了什么确切的具体对象。下表为抽象工厂模式与工厂方法模式的对比。

抽象工厂	工厂方法
通过对象组合创建抽象产品	通过类继承创建抽象产品
创建多系列产品	创建一种产品
必须修改父类的接口才能支持新的产品	子类化创建者并重载工厂方法以创建新产品

5.1　把抽象工厂应用到TouchPainter应用程序

想象一个场景：对于在第2章中开发的TouchPainter应用程序，跟我们合作的那些公司想要冠上他们的名称和标志。在这一节，我们将为两个公司贴牌：Sierra Corporation和Acme Corporation。他们的品牌是Sierra和Acme。

显然，最初设计这个应用程序的时候，我们并未考虑品牌。目前加品牌最好的办法是在主视图上为特定品牌进行一些有针对性的设计。加品牌过程中，也有可能涉及其他UI元素，如让用户返回主应用程序的主画面按钮（main button），以及主画布上的工具条。似乎有多处变动需要加到架构之中。

软件设计的黄金法则：变动需要抽象。

如果有多个类共有相同的行为，但实际实现不同，则可能需要某种抽象类型作为其父类被继承。抽象类型定义所有相关具体类将共有的共同行为。例如，我们知道普通的比萨饼是什么样子，在点餐的时候能预料到会端上来什么。我们说“出去吃比萨饼吧!”这里，“比萨饼”是一个抽象类型，定义了比萨饼应该具有的共同特征。但是，从不同的店我们可以得到同一比萨饼（比方说，意大利辣肠比萨饼）的略有不同的风味。因为有太多不同类型的比萨饼，我们简单地将其叫做“比萨饼”，以称呼这种特定类型的食品。

回到加品牌的设计，我们在产品和品牌上有几种变化，因此需要对全体进行某种抽象。需要两个品牌工厂，各自会使用工厂方法生产3种不同的UI元素。每个品牌有定制的`UIView`、`UIButton`和`UIToolbar`元素。图5-2中的类图说明了它们的关系。

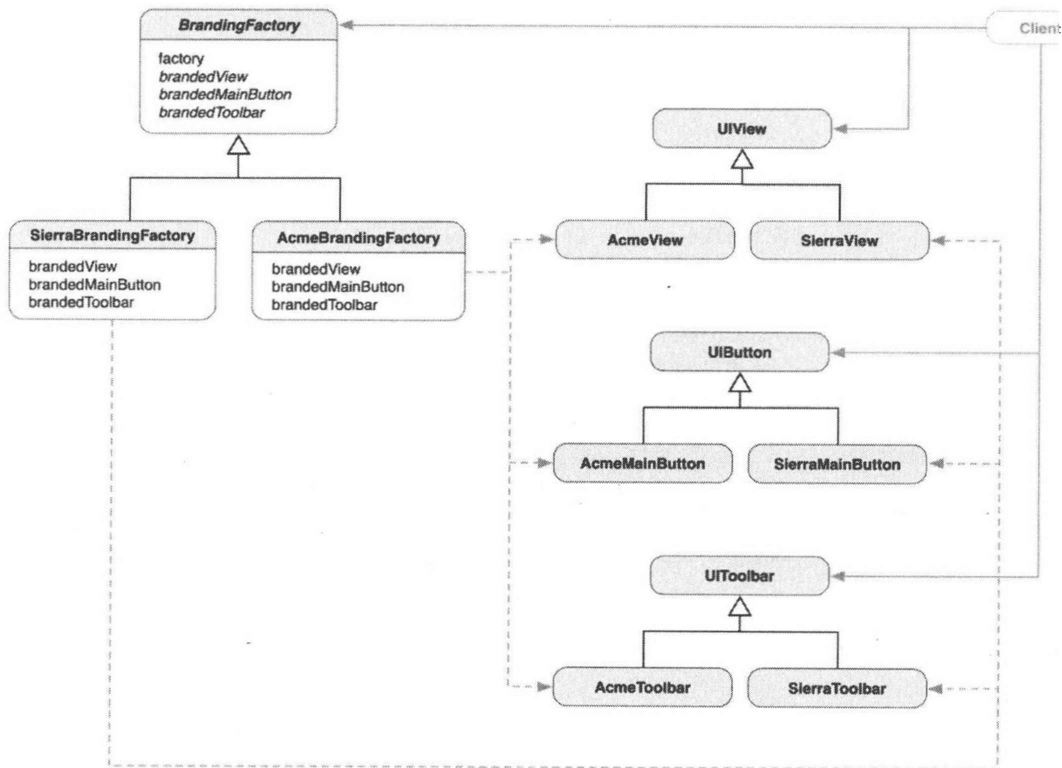


图5-2　TouchPainter应用程序的类图。这个应用程序用抽象工厂模式支持不同品牌的多个UI元素

图5-2中的类结构与图5-1中的相似。每个产品有自身的产品体系，并由各个具体工厂中的工厂方法支持。产品由两个不同的品牌工厂——SierraBrandingFactory和AcmeBrandingFactory来“生产”。它们各自重载抽象BrandingFactory类中定义的工厂方法brandedView、brandedMainButton和brandedToolbar，并根据工厂设计所针对的品牌返回具体产品。

超类的类方法factory是返回具体BrandingFactory的正确版本的工厂方法。其子类不应重载这个方法（尽管子类能够这么做）。factory方法根据当前的编译配置返回一个具体品牌工厂的实例。在BrandingFactory中的这些工厂方法（brandedView、brandedMainButton和brandedToolbar）的默认实现返回抽象产品UIView、UIButton和UIToolbar的实例，不带任何品牌细节。每个品牌所要求的一切细节将由实际的具体品牌工厂在其重载的工厂方法中生产。

在这种方案下，客户端只需要知道4个实体：BrandingFactory、UIView、UIButton和UIToolbar。这样，将来如果需要在其他领域扩展应用程序的品牌活动，可以通过添加新的产品和品牌工厂得以实现，而不影响客户端代码。

代码清单5-1说明了如何实现品牌工厂和相应产品。

代码清单5-1 BrandingFactory.h

```
@interface BrandingFactory : NSObject
{

}

+ (BrandingFactory *) factory;

- (UIView *) brandedView;
- (UIButton *) brandedMainButton;
- (UIToolbar *) brandedToolbar;

@end
```

我们朴实的抽象BrandingFactory有一个类方法+ (BrandingFactory*) factory，它返回具体BrandingFactory子类的实例。Branding Factory 还定义了3个生产实际产品实例的工厂方法：brandedView返回UIView的实例，brandedMainButton返回UIButton的实例，brandedToolbar返回UIToolbar的实例。具体的BrandingFactory将返回实际的产品。稍后将会谈到这一点。来看看代码清单5-2中BrandingFactory的实现。

代码清单5-2 BrandingFactory.m

```
#import "BrandingFactory.h"
#import "AcmeBrandingFactory.h"
#import "SierraBrandingFactory.h"

@implementation BrandingFactory

+ (BrandingFactory *) factory
{
#if defined (USE_ACME)
```

```
  return [[[AcmeBrandingFactory alloc] init] autorelease];
#elif defined (USE_SIERRA)
  return [[[SierraBrandingFactory alloc] init] autorelease];
#else
  return nil;
#endif
}

- (UIView *) brandedView
{
  return nil;
}

- (UIButton *) brandedMainButton
{
return nil;
}

- (UIToolbar *) brandedToolbar
{
return nil;
}

@end
```


代码清单5-2中的`factory`类方法中，我们使用预处理器定义告诉编译器让这个方法返回哪个具体工厂。如果定义了宏`USE_ACME`，那么运行时会使用语句`return [[[AcmeBrandingFactory alloc] init] autorelease]`。如果定义了宏`USE_SIERRA`，那么会使用语句`return [[[SierraBrandingFactory alloc] init] autorelease]`。除此以外的其他情况，此方法会返回`nil`。

这些工厂方法的最初实现默认返回`nil`。我们将看到如何通过在具体品牌工厂中重载这些方法，实现具体产品的创建。首先要写的是Acme品牌的`BrandingFactory`，见代码清单5-3。

代码清单5-3 AcmeBrandingFactory.h

```
#import "BrandingFactory.h"

@interface AcmeBrandingFactory : BrandingFactory
{

}

- (UIView *) brandedView;
- (UIButton *) brandedMainButton;
- (UIToolbar *) brandedToolbar;

@end
```

`AcmeBrandingFactory`继承基类`BrandingFactory`，并重载其工厂方法为此品牌生产正确的UI元素。原始的`factory`类方法没有被重载，因为它原本用于父类创建具体`BrandingFactory`

的实例。但是我们无法防止类方法被子类重载。代码清单5-4给出了实现AcmeBrandingFactory的方法。

代码清单5-4　AcmeBrandingFactory.m

```
#import "AcmeBrandingFactory.h"
#import "AcmeView.h"
#import "AcmeMainButton.h"
#import "AcmeToolbar.h"

@implementation AcmeBrandingFactory

- (UIView *) brandedView
{
  // 返回Acme的定制视图
  return [[[AcmeView alloc] init] autorelease];
}

- (UIButton *) brandedMainButton
{
  // 返回Acme的定制主按钮
  return [[[AcmeMainButton alloc] init] autorelease];
}

- (UIToolbar *) brandedToolbar
{
  // 返回Acme的定制工具条
  return [[[AcmeToolbar alloc] init] autorelease];
}

@end
```

每个工厂方法的实现返回反映实际品牌的Acme*产品的实例。代码清单5-5显示了另一个品牌Sierra的实现。

代码清单5-5　SierraBrandingFactory.h

```
#import "BrandingFactory.h"

@interface SierraBrandingFactory : BrandingFactory
{

}

- (UIView *) brandedView;
- (UIButton *) brandedMainButton;
- (UIToolbar *) brandedToolbar;

@end
```

具体实现见代码清单5-6。

代码清单5-6　SierraBrandingFactory.m

```
#import "SierraBrandingFactory.h"
```

```
#import "SierraView.h"
#import "SierraMainButton.h"
#import "SierraToolbar.h"

@implementation SierraBrandingFactory

- (UIView *) brandedView
{
  // 返回Sierra的定制视图
  return [[[SierraView alloc] init] autorelease];
}

- (UIButton *) brandedMainButton
{
  // 返回Sierra的定制主按钮
  return [[[SierraMainButton alloc] init] autorelease];
}

- (UIToolbar *) brandedToolbar
{
  // 返回Sierra的定制工具条
  return [[[SierraToolbar alloc] init] autorelease];
}

@end
```

除了工厂方法返回的产品名称不同之外，SierraBrandingFactory的实现与AcmeBrandingFactory几乎相同。返回的UI元素仅用于Sierra品牌。我们可以子类化BrandingFactory来创建新的品牌，并以与Acme和Sierra相同的方式重载新旧工厂方法创建任何品牌。

> **说明**：当现有的抽象工厂需要支持新产品时，需要向父类添加相应的新工厂方法。这意味着也要修改其子类以支持新产品的新工厂方法。

工厂一旦返回了某些产品，客户端就可以像在代码清单5-7中视图控制器的loadView方法里那样使用这些产品。

代码清单5-7　视图控制器的loadView方法

```
// 实现loadView，以编程方式创建视图体系，不使用nib。
- (void)loadView
{
  // 使用从BrandingFactory获得的带品牌UI元素构建视图
  BrandingFactory * factory = [BrandingFactory factory];
  //……
  UIView *view = [factory brandedView];
  //…… 把view放在视图的合适位置
  //……
  UIButton *button = [factory brandedMainButton];
  //…… 把button放在视图合适位置
  //……
  UIToolbar *toolbar = [factory brandedToolbar];
```

```
    //…… 把toolbar放在视图合适位置
}
```

视图控制器的loadView方法中，控制器可以编程构建其视图。在代码清单5-7中，请求BrandingFactory的类方法根据当前编译配置返回适当的具体BrandingFactory。一旦得到了实际BrandingFactory的引用，就可以调用其工厂方法来返回UI元素，进而通过编程构建任何带品牌的视图元素。

这里使用类工厂方法创建具体BrandingFactory的好处是，无需设计在运行时决定使用什么工厂的复杂机制。通过简单的预处理定义，这完全在编译时决定，过程中不涉及其他类。否则，可能牵扯到对应用程序作更复杂的修改。BrandingFactory以类簇（class cluster）的形式实现，其中，一组相关的子类组合在一起，由其超类创建。这种工厂构建过程常见于Cocoa Touch框架的基础（Foundation）类库，我们将在5.2节讨论Cocoa Touch框架。

5.2 在 Cocoa Touch 框架中使用抽象工厂

抽象工厂模式常见于Cocoa Touch框架。有很多基础类采用了这一模式。特别常见的一个就是天天在用的NSNumber。创建NSNumber实例的方式完全符合抽象工厂模式。

创建Cocoa Touch对象有两种方式：使用先alloc再init的方法（两步创建过程），或者使用类中的+className...方法。在Cocoa Touch的基础框架中，NSNumber类有很多类方法用于创建各种类型的NSNumber对象，像下面这样：

```
NSNumber * boolNumber = [NSNumber numberWithBool:YES];
NSNumber * charNumber = [NSNumber numberWithChar:'a'];
NSNumber * intNumber = [NSNumber numberWithInt:1];
NSNumber * floatNumber = [NSNumber numberWithFloat:1.0];
NSNumber * doubleNumber = [NSNumber numberWithDouble:1.0];
```

每个返回的对象属于代表最初输入值的不同私有子类。这些创建NSNumber实际实例的类方法与先前例子中描述的BrandingFactory的factory方法类似。可以像下面这样用NSLog列出它们的类描述：

```
NSLog(@"%@", [[boolNumber class] description]);
NSLog(@"%@", [[charNumber class] description]);
NSLog(@"%@", [[intNumber class] description]);
NSLog(@"%@", [[floatNumber class] description]);
NSLog(@"%@", [[doubleNumber class] description]);
```

将看到如下调试器控制台（Debugger Console）输出：

```
NSCFBoolean
NSCFNumber
NSCFNumber
NSCFNumber
NSCFNumber
```

除了boolNumber的实际类型是NSCFBoolean以外，大多数实际类为NSCFNumber类型。尽管这些+className类工厂方法返回NSNumber具体子类的实例，但是返回的实例确实支持

NSNumber的公有接口。

虽然它们属于NSNumber的不同具体子类，但是其行为由抽象超类NSNumber定义，而且是公有的。若执行以下代码段，就会明白我的意思。

```
NSLog(@"%d", [boolNumber intValue]);
NSLog(@"%@", [charNumber boolValue] ? @"YES" : @"NO");
```

将看到如下调试器控制台输出：

```
1
YES
```

boolNumber在内部保持布尔值YES，但仍实现了公有iniValue方法，返回反映其内部布尔值的适当整数值。charNumber也是如此，它重载boolValue方法，返回反映其内部字符值“a”的适当布尔值。

接受不同类型的参数并返回NSNumber实例的类方法是类工厂方法（工厂方法模式，第4章）。NSNumber的类工厂方法生产各种“数工厂”。numberWithBool:创建NSCFBoolean工厂的实例，而numberWithInt:创建NSCFNumber的实例。NSNumber中的类工厂方法定义了决定实例化何种私有具体子类（比如，NSCFBoolean或NSCFNumber）的默认行为。这一版本的工厂方法是传统工厂方法模式的一个变体，虽然它达成返回抽象产品的目的，此处的抽象产品为作为工厂的具体NSNumber子类。NSNumber是抽象工厂实现的一个例子。基础框架中抽象工厂的此种特点被称为“类簇”（Class Cluster）。

类簇是基础框架中一种常见的设计模式，基于抽象工厂模式的思想。它将若干相关的私有具体工厂子类集合到一个公有的抽象超类之下。例如，“数”包含了各种数值类型的完整集合，如字符、整数、浮点数和双精度数。这些数值类型是“数”的子集。所以NSNumber自然成为这些数子类型的超类型（super-type）。NSNumber有一系列公有API，定义了各种类型的数所共有的行为。客户端在使用时无需知道NSNumber实例的具体类型。

类簇是抽象工厂的一种形式。比如，NSNumber本身是一个高度抽象的工厂，而NSCFBoolean和NSCFNumber是具体工厂子类。子类是具体工厂，因为它们重载了NSNumber中声明的公有工厂方法以生产产品。例如，intValue和boolValue根据实际NSNumber对象的内部值返回一个值，虽然值的数据类型可能不同。从这些工厂方法返回的实际值就是抽象工厂模式的最初定义中的所说“产品”。

创建抽象产品的工厂方法与创建抽象工厂的工厂方法之间有个不同点。显然，像intValue和boolValue这样的工厂方法，应在具体工厂（NSCFNumber与NSCFBoolean）中重载以返回实际值（产品）。其他像numberWithBool:和numberWithInt:这样的工厂方法并不是为了返回产品，而是为了返回能返回产品的工厂，因此它们不应在具体工厂子类中重载。

要想定义自己的NSNumber，可以子类化它并重载已定义的类工厂方法，如numberWithBool:和numberWithChar:，以返回自己的子类而不是内建的NSCFNumber和NSCFBoolean（其实，苹果公司根本没打算让我们使用其私有类）。当然，新的NSNumber子类也需要实现工厂方法，如intValue、boolValue等。

也许你已经意识到BrandingFactory实现为一种类簇的形式。要使用什么子类，客户端无需确切知晓，而是由BrandingFactory的类工厂方法来决定。具体品牌子类的细节不被客户端的视图所见，而子类实现工厂方法，返回在BrandingFactory的共有接口中定义的定制视图、按钮和工具条。

其他实现为类簇的基础类有NSData、NSArray、NSDictionary和NSString。

再次对比抽象工厂和工厂

再问一次，区别何在？我们一直在讨论的就是前者，一个被其多个具体工厂类型共有的抽象工厂类型。如果抛开“抽象”一词，“工厂”通常是指“具体”工厂，而且，它也没有工厂方法的意思（第4章）。

因此，如果听到有人说“这里需要一个工厂”，并不一定意指抽象工厂，可能只是返回抽象产品（见享元模式，第21章）的具体工厂。前面的例子中，NSNumber是个抽象工厂，而NSCFBoolean与NSCFNumber是工厂（具体的）。

有时，一开始在设计中使用具体工厂，而后将其重构（refactor）为使用多个具体工厂的抽象工厂。

5.3 总结

抽象工厂模式是一种极为常见的设计模式。它是最基本的，因为它可以涉及许多类型的对象创建。一系列相关类的好的模式，应该作为一种抽象，不为客户端所见。抽象工厂可以顺畅地提供这种抽象，而不暴露创建过程中任何不必要的细节或所创建对象的确切类型。下一章要讨论创建抽象对象的另一种方法——生成器模式。

第6章 生成器

6

选择建造自己的房子的人会把工程外包给承包商。单一承包商不能建造整个房子，他将其分解为几个部分，然后转包给几个实际的建筑商（builder），他们懂得如何将零部件组装起来。房子由风格、颜色和尺寸各不相同的部件组成。客户告诉承包商房子里都要有什么，然后承包商协调各房屋建筑商，决定需要做什么。应该如何建造，建筑商就如何施工。建房子是个复杂过程，单凭一双手就想建房子，即便可能也非常困难。如果承包商（指导者）与懂得如何建造的建筑商相互协调，这一过程将简单得多且更易管理。

有时，构建某些对象有多种不同方式。如果这些逻辑包含在构建这些对象的类的单一方法中，构建的逻辑会非常荒唐（例如，针对各种构建需求的一大片嵌套if-else或者switch-case语句）。如果能够把构建过程分解为客户–指导者–生成器（client-director-builder）的关系，那么过程将更容易管理与复用。针对此类关系的设计模式称为生成器。

本章将讨论生成器模式的概念。后面几节，也会讨论如何使用这一模式来生成RPG游戏中带有复杂特征的角色。

6.1 何为生成器模式

除了客户与其所要的产品，生成器模式还包含两个重要角色：Director（指导者）和Builder（生成器）。Builder知道究竟如何在缺少某些特定信息的情况下建造产品（什么）。Director知道Builder应该建造什么，以参数向其提供缺少的信息来建造特定产品。什么与如何有点儿难懂。尽管Director知道Builder应该建造什么，这并不意味着Director知道具体Builder究竟是什么。它们的静态关系如图6-1中的类图所示。

Builder是一个抽象接口，声明了一个buildPart方法，该builder方法由ConcreteBuilder实现，以构造实际产品（Product）。ConcreteBuilder有个getResult方法，向客户端返回构造完毕的Product。Director定义了一个construct方法，命令Builder的实例去buildPart。Director和Builder形成一种聚合关系。这意味着Builder是一个组成部分，与Director结合，以使整个模式运转，但同时，Director并不负责Builder的生存期。这种“整体–部分”的关系用图6-2中的时序图会说明得更清楚。

aClient生成ConcreteBuilder的实例（aConcreteBuilder）和以aConcreteBuilder

为初始化参数的Director的实例（aDirector），用于今后协同工作。当aClient发送construct消息给aDirector时，该方法发送要建造什么消息（比如buildPartA、buildPartB和buildPartC）给aConcreteBuilder。aDirector的construct方法返回后，aClient直接向aConcreteBuilder发送getResult消息，取回建造完毕的产品。所以aDirector所知的“什么”，就是每个Builder能够建造什么部件。

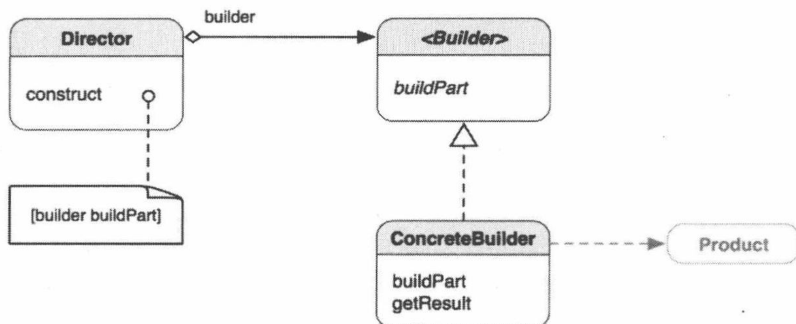


图6-1　生成器模式的类图

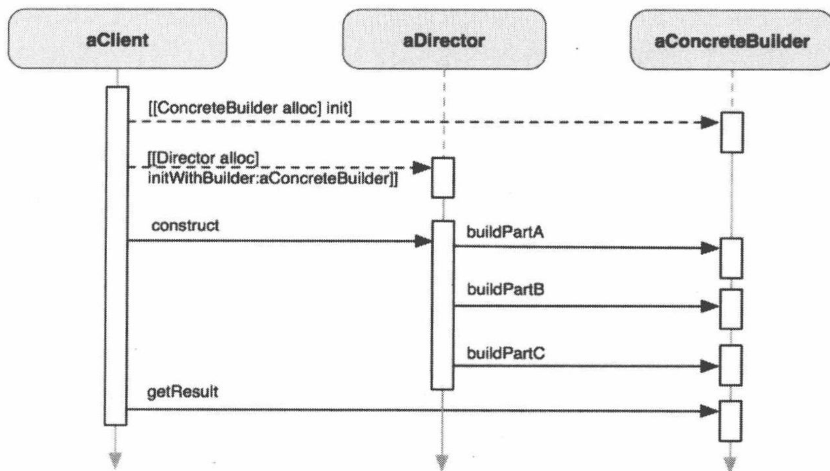


图6-2　运行时，aClient、aDirector和aConcreteBuilder之间交互的时序图

生成器模式：将一个复杂对象的构建与它的表现分离，使得同样的构建过程可以创建不同的表现。*

* 最初的定义出现于《设计模式》(Addison-Wesley，1994)。

6.2　何时使用生成器模式

在以下情形，自然会想到使用这一模式。

- 需要创建涉及各种部件的复杂对象。创建对象的算法应该独立于部件的装配方式。常见例子是构建组合对象。
- 构建过程需要以不同的方式（例如，部件或表现的不同组合）构建对象。

生成器与抽象工厂的对比

前一章讨论了抽象工厂。也许读者已意识到，抽象工厂与生成器模式在抽象对象创建方面有许多相似之处。然而，两者却大不相同。一方面，生成器关注的是分步创建复杂对象，很多时候同一类型的对象可以以不同的方式创建。另一方面，抽象工厂的重点在于创建简单或复杂产品的套件。生成器在多步创建过程的最后一步返回产品，而抽象工厂则立即返回产品。下表总结了生成器模式与抽象工厂模式的主要差异。

生 成 器	抽象工厂
构建复杂对象	构建简单或复杂对象
以多个步骤构建对象	以单一步骤构建对象
以多种方式构建对象	以单一方式构建对象
在构建过程的最后一步返回产品	立刻返回产品
专注一个特定产品	强调一套产品

图6-2中，aClient要从aBuilder得到产品，需要知道aDirector和aBuilder。你可能想知道，如果让aDirector从它的construct方法返回产品，或getResult以某种方式在aDirector中实现，是否aClient可以不必知道aBuilder。那样的话，aDirector就变成了工厂，它的construct方法就变成了返回抽象产品的工厂方法。而且，aDirector将与所支持的产品固定在一起，这降低了模式的可复用性。整体思想是分离“什么”与“如何”，使得aDirector能把同一个“什么”（规格）应用到不同的aBuilder，而它懂得“如何”按照给定的规格建造自己的特定产品，反之亦然。

在下面几节，我们将通过具体例子来说明如何用生成器模式来解决相关的设计问题。这个例子是构建带有各类角色的追逐游戏。涉及各类对象、财产或角色的游戏，其构建过程会相当复杂。可以通过生成器模式，把构建角色的算法（如何构建角色）与构建什么角色分离开来。

6.3 构建追逐游戏中的角色

我们将以假想的追逐游戏为例，演示如何实现生成器模式。假定有两种类型的角色——敌人和游戏者。敌人将追逐游戏者。玩家决定游戏者角色往哪儿走。路上可能有障碍物。两种角色有一些共同的基本特征（trait），如力量、耐力、智力、敏捷和攻击力。每一特征都影响角色的防御（Protection）与攻击（Power）能力。防御因子反映了角色防御攻击的能力，而攻击因子反映了攻击对手的能力。特征与防御或攻击因子成正比或反比关系。表6-1是显示其关系的矩阵。

表6-1　角色特征矩阵（↑表示两种角色特征间成正比例，↓表示成反比例）

	防　御	攻　击
力量（Strength）	↑	↑
耐力（Stamina）	↑	↑
智力（Intelligence）	↑	↓
敏捷（Agility）	↑	↓
攻击力（Aggressiveness）	↓	↑

力量和耐力与防御和攻击成正比例。拥有较高力量与耐力因子的角色，也多一些自我防御与反击的机会。智力和敏捷与防御因子成正比例，而与攻击成反比例。根据我们的设计，如果角色较机智，那么他有较强的防御能力，而不是反击能力。而攻击性，与智力和敏捷相反。如果角色有攻击性，那么他将有更多机会攻击，而不是在攻击中防御自己。当然，角色特征的设计纯属虚构，你也可以做出截然不同的设计。那么我们将按照矩阵中所示的关系，使用生成器模式，通过角色特征的各种组合来构建角色。

思　考　题

如果不用生成器模式，应该如何设计类，才能让它根据表6-1中矩阵对特征进行组合，来创建角色呢？

我们定义一个叫ChasingGame的类，它有两个方法，用以创建两种类型的角色——游戏者与敌人。CharacterBuilder基于前述矩阵所示各种特征的关系构建角色。每一特征因子（值）影响被构建的角色的特性。显示其静态关系的类图如图6-3所示。

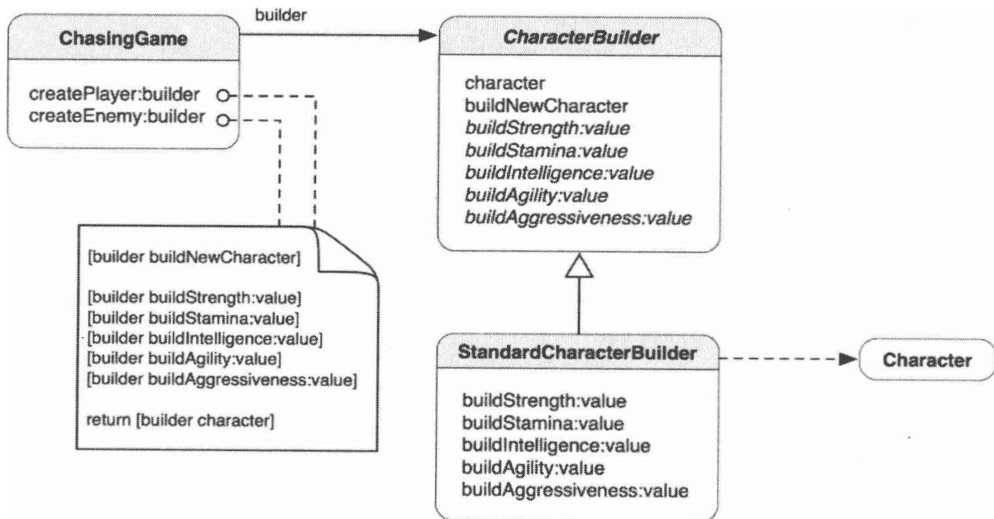


图6-3　类图中，CharacterBuilder为抽象生成器，StandardCharacterBuilder为具体生成器，ChasingGame为指导者

ChasingGame定义了createPlayer:builder和createEnemy:builder，通过CharacterBuilder的实例创建游戏者和敌人角色。每个方法有一套不同的特征因子，用来定义角色的特性。StandardCharacterBuilder是具体的CharacterBuilder，它根据不同特征因子实际构建角色。构建过程结束后，StandardCharacterBuilder将返回Character的实例。Character的定义见代码清单6-1。

代码清单6-1 Character.h

```
@interface Character : NSObject
{
  @private
  float protection_;
  float power_;
  float strength_;
  float stamina_;
  float intelligence_;
  float agility_;
  float aggressiveness_;
}

@property (nonatomic, assign) float protection;
@property (nonatomic, assign) float power;
@property (nonatomic, assign) float strength;
@property (nonatomic, assign) float stamina;
@property (nonatomic, assign) float intelligence;
@property (nonatomic, assign) float agility;
@property (nonatomic, assign) float aggressiveness;

@end
```

Character定义所有类型角色（游戏者或敌人）共有的一套特征，包括防御、攻击、力量、耐力、智力、敏捷和攻击力，同矩阵中内容一样。

Character的实现仅仅是定义了一个init方法和几个属性的同步，见代码清单6-2。

代码清单6-2 Character.m

```
#import "Character.h"

@implementation Character

@synthesize protection=protection_;
@synthesize power=power_;
@synthesize strength=strength_;
@synthesize stamina=stamina_;
@synthesize intelligence=intelligence_;
@synthesize agility=agility_;
@synthesize aggressiveness=aggressiveness_;

- (id) init
{
  if (self = [super init])
```

6

```
    {
      protection_ = 1.0;
      power_ = 1.0;
      strength_ = 1.0;
      stamina_ = 1.0;
      intelligence_ = 1.0;
      agility_ = 1.0;
      aggressiveness_ = 1.0;
    }

    return self;
  }

@end
```

Character的实例不知道如何把自己构建成有意义的角色，所以才需要CharacterBuilder基于先前定义的特征关系，构建有意义的角色。图6-2中有个抽象CharacterBuilder，它定义了任何角色生成器都该有的接口。它的类声明如代码清单6-3所示。

代码清单6-3　CharacterBuilder.h

```
#import "Character.h"

@interface CharacterBuilder : NSObject
{
  @protected
  Character *character_;
}

@property (nonatomic, readonly) Character *character;

- (CharacterBuilder *) buildNewCharacter;
- (CharacterBuilder *) buildStrength:(float) value;
- (CharacterBuilder *) buildStamina:(float) value;
- (CharacterBuilder *) buildIntelligence:(float) value;
- (CharacterBuilder *) buildAgility:(float) value;
- (CharacterBuilder *) buildAggressiveness:(float) value;

@end
```

CharacterBuilder的实例有个对目标Character的引用，该目标Charater构建完成后将被返回给客户端。有几个构建角色的方法，构建的角色具有特定的力量、耐力、智力、敏捷与攻击力值。这些值影响防御和攻击因子。抽象的CharacterBuilder定义了默认行为，它把这些值设定给目标Character，见代码清单6-4。

代码清单6-4　CharacterBuilder.m

```
#import "CharacterBuilder.h"

@implementation CharacterBuilder

@synthesize character=character_;
```

```
- (CharacterBuilder *) buildNewCharacter
{
  // 创建新角色之前，自动释放先前的角色
  [character_ autorelease];
  character_ = [[Character alloc] init];

  return self;
}

- (CharacterBuilder *) buildStrength:(float) value
{
  character_.strength = value;
  return self;
}

- (CharacterBuilder *) buildStamina:(float) value
{
  character_.stamina = value;
  return self;
}

- (CharacterBuilder *) buildIntelligence:(float) value
{
  character_.intelligence = value;
  return self;
}

- (CharacterBuilder *) buildAgility:(float) value
{
  character_.agility = value;
  return self;
}

- (CharacterBuilder *) buildAggressiveness:(float) value
{
  character_.aggressiveness = value;
  return self;
}

- (void) dealloc
{
  [character_ autorelease];
  [super dealloc];
}

@end
```

CharacterBuilder的buildNewCharacter方法生成要构建的Character新实例。每次调用此方法，它就会在生成新角色之前autorelease任何从前的角色。使用autorelease的方式，比用release立刻释放要安全，因为可能还有别的客户端在用着从前的角色，并不知道它已经在生成器中被释放。其余的方法对构建角色没有什么大的用处，除非定义CharacterBuilder的具体类来做这些事情。StandardCharacterBuilder是CharacterBuilder的子类，定义了生成具有各种

相关特征的真正角色的逻辑。它的类声明与CharacterBuilder差别不大，见代码清单6-5。

代码清单6-5　StandardCharacterBuilder.h

```
#import "CharacterBuilder.h"

@interface StandardCharacterBuilder : CharacterBuilder
{

}

// 从抽象CharacterBuilder重载的方法
- (CharacterBuilder *) buildStrength:(float) value;
- (CharacterBuilder *) buildStamina:(float) value;
- (CharacterBuilder *) buildIntelligence:(float) value;
- (CharacterBuilder *) buildAgility:(float) value;
- (CharacterBuilder *) buildAggressiveness:(float) value;

@end
```

为了清晰起见，我们再次声明了重载的方法。StandardCharacterBuilder没有重载buildNewCharacter方法，因为基类中的默认行为已经够用了。代码清单6-6是StandardCharacterBuilder的实现，来看看构建真正角色的逻辑是如何实现的。

代码清单6-6　StandardCharacterBuilder.m

```
#import "StandardCharacterBuilder.h"

@implementation StandardCharacterBuilder

- (CharacterBuilder *) buildStrength:(float) value
{
  // 更新角色的防御值
  character_.protection *= value;

  // 更新角色的攻击值
  character_.power *= value;

  // 最后设定力量值并返回此生成器
  return [super buildStrength:value];
}

- (CharacterBuilder *) buildStamina:(float) value
{
  // 更新角色的防御值
  character_.protection *= value;

  // 更新角色的攻击值
  character_.power *= value;

  // 最后设定耐力值并返回此生成器
  return [super buildStamina:value];
}
```

```
- (CharacterBuilder *) buildIntelligence:(float) value
{
  // 更新角色的防御值
  character_.protection *= value;

  // 更新角色的攻击值
  character_.power /= value;

  // 最后设定智力值并返回此生成器
  return [super buildIntelligence:value];
}

- (CharacterBuilder *) buildAgility:(float) value
{
  // 更新角色的防御值
  character_.protection *= value;

  // 更新角色的攻击值
  character_.power /= value;

  // 最后设定敏捷值并返回此生成器
  return [super buildAgility:value];
}

- (CharacterBuilder *) buildAggressiveness:(float) value
{
  // 更新角色的防御值
  character_.protection /= value;

  // 更新角色的攻击值
  character_.power *= value;

  // 最后设定攻击力值并返回此生成器
  return [super buildAggressiveness:value];
}

@end
```


前面的每个方法基本上是按照矩阵中的比例关系定义，设定正在构建的角色的防御和攻击值。比如，智力与防御成正比，因此通过`character_.protection *= value`来获得新的防御值，其中`value`是智力的输入值。这里使用Objective-C的“点”语法，尽管这可能与C结构体和C++对象之间发生混淆（在代码中混合使用C结构体和Objective-C++的时候需要注意），但因为角色的状态足够简单明了，方便起见，这样做也未尝不可。类似地，攻击力与防御成反比，因此通过`character_.protection /= value`来获得新的防御值。最后，向`super`发送消息用新值来更新目标角色，然后将自身返回。

接下来看看`ChasingGame`是如何使用`StandardCharacterBuilder`来构建各种角色的。它的类声明见代码清单6-7。

代码清单6-7 ChasingGame.h

```
#import "StandardCharacterBuilder.h"
```

```
@interface ChasingGame : NSObject
{

}

- (Character *) createPlayer:(CharacterBuilder *) builder;
- (Character *) createEnemy:(CharacterBuilder *) builder;

@end
```

ChasingGame有两个方法，createPlayer:和createEnemy:。每个方法使用CharacterBuilder的实例，构建带有一套预定义特征因子的特定类型的角色。其实现见代码清单6-8。

代码清单6-8 ChasingGame.m

```
#import "ChasingGame.h"

@implementation ChasingGame

- (Character *) createPlayer:(CharacterBuilder *) builder
{
  [builder buildNewCharacter];
  [builder buildStrength:50.0];
  [builder buildStamina:25.0];
  [builder buildIntelligence:75.0];
  [builder buildAgility:65.0];
  [builder buildAggressiveness:35.0];

  return [builder character];
}

- (Character *) createEnemy:(CharacterBuilder *) builder
{
  [builder buildNewCharacter];
  [builder buildStrength:80.0];
  [builder buildStamina:65.0];
  [builder buildIntelligence:35.0];
  [builder buildAgility:25.0];
  [builder buildAggressiveness:95.0];

  return [builder character];
}

@end
```

ChasingGame的实例在createPlayer:方法中构建了一个游戏者角色，其特征因子分别为力量50.0、耐力25.0、智力75.0、敏捷65.0、攻击力35.0。在其createEnemy:方法中，跟创建游戏者同样的方式创建了一个敌人，只是使用了不同的特征因子。显然，与游戏者相比，敌人较为强壮，且更具攻击性，而游戏者则更有智慧但力量较弱。

因为各个build*方法返回当前生成器的实例，所以可以把整个构建过程组合到单一语句中，像代码清单6-9这样。

代码清单6-9 构建角色的另一种语法风格

```
[[[[[[builder buildNewCharacter]
              buildStrength:50.0]
              buildStamina:25.0]
              buildIntelligence:75.0]
              buildAgility:65.0]
              buildAggressiveness:35.0];
```

哪个更好呢？这只是个人喜好问题而已。

现在我们对构建各种角色已经讲得比较清楚了。代码清单6-10显示了在客户端代码中是怎么做的。

代码清单6-10 客户端代码

```
CharacterBuilder *characterBuilder = [[[StandardCharacterBuilder alloc] init]
                                                                autorelease];
ChasingGame *game = [[[ChasingGame alloc] init] autorelease];

Character *player = [game createPlayer:characterBuilder];
Character *enemy = [game createEnemy:characterBuilder];

// 用player和enemy做些别的事情
```

客户端生成StandardCharacterBuilder和ChasingGame的实例。然后向ChasingGame发送createPlayer:和createEnemy:消息。characterBuilder打造好了两个角色之后，我们就可以开始游戏了。

6.4 总结

生成器模式能帮助构建涉及部件与表现的各种组合的对象。没有这一模式，知道构建对象所需细节的Director可能最终会变成一个庞大的“神”类，带有无数用于构建同一个类的各种表现的内嵌算法。涉及具有各种特征的角色的游戏，应该好好使用这一模式。不是定义单独的Director去构建游戏者与敌人，而是把角色构建算法放在一个具体CharacterBuilder中，设计会好得多。

下一章将讨论一种只生成并返回类的单一实例的模式。

第7章 单　例

数学与逻辑学中，singleton定义为“有且仅有一个元素的集合”[①]。因此不管袋子有多大，每次从里面拿出弹子的时候，拿到的都是同一个。在什么情况下会需要单元素集合（单例）呢？想想系统中那些只能共享而不能复制的资源。例如，GPS设备是iPhone中实时提供设备坐标的唯一硬件。CoreLocation框架中的`CLLocationManager`类，定义了对这个GPS设备所提供服务的单一访问点。也许有人会想，要是能复制一个`CLLocationManager`，那我的应用程序不就能获得一套额外的GPS服务吗？听起来很神奇——用一个硬件GPS的成本创建了两份软件GPS。但是实际上，同一时刻还是只有一个GPS，因为设备中只有一个GPS在同天空中的卫星进行实际的连接。所以要是你觉得写了个杀手级应用程序，可以同时操作两套单独的GPS连接，想要跟朋友们炫耀，重新考虑一下吧！

面向对象应用程序中的单例类（singleton class）总是返回自己的同一个实例。它提供了对类的对象所提供的资源的全局访问点。与这类设计相关的设计模式称为单例模式。

本章将探讨一些可能的实现方式，以及单例模式在Objective-C及iOS的Cocoa Touch框架中的使用。

7.1 何为单例模式

单例模式几乎是设计模式的最简单形式了。这一模式的意图是使得类的一个对象成为系统中的唯一实例。要实现这一点，可以从客户端对其进行实例化开始。因此需要用一种只允许生成对象类的唯一实例的机制，“阻止”所有想要生成对象的访问。我们可以用工厂方法（第4章）来限制实例化过程。这个方法应该是个静态方法（类方法），因为让类的实例去生成另一个唯一实例毫无意义。图7-1显示了简单单例模式的类结构。

`static uniqueInstance`是`Singleton`的唯一实例，`static sharedInstance`将把它返回给客户端。通常，`sharedInstance`会检查`uniqueInstance`是否已经被实例化。如果没有，它会生成一个实例然后返回`uniqueInstance`。

> **单例模式**：保证一个类仅有一个实例，并提供一个访问它的全局访问点。*
> *最初的定义出现于《设计模式》（Addison-Wesley，1994）。

---

① 数学中，singleton译为“单元素集合”，指仅由一个元素组成的集合。——译者注

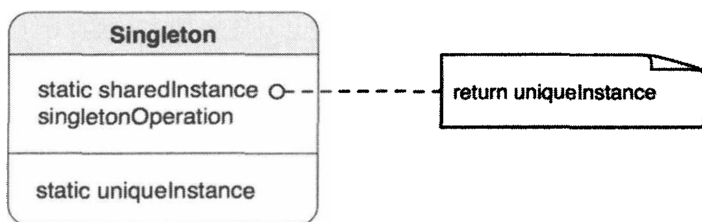


图7-1 单例模式的静态结构

7.2 何时使用单例模式

在以下情形，应该考虑使用单例模式：

- 类只能有一个实例，而且必须从一个为人熟知的访问点对其进行访问，比如工厂方法（第4章）；
- 这个唯一的实例只能通过子类化进行扩展，而且扩展的对象不会破坏客户端代码。

单例模式提供了一个为人熟知的访问点，供客户类为共享资源生成唯一实例，并通过它对共享资源进行访问。虽然静态的全局对象引用或类方法也可以提供全局访问点，但是全局对象无法防止类被实例化一次以上，而且类方法也缺少消除耦合的灵活性。

静态全局变量保持着对类的实例的唯一引用，那些访问这个全局变量的类或方法，实际上是在和使用这个变量的其他类或方法共享着同一份副本。这听起来好像是我们在本章中想要的。如果在整个应用程序中都只使用同一个全局变量，那么似乎万事大吉，好像实际上并不需要单例模式。可是，要是团队中的某位老兄或者哪个顾问也定义了相同类型的静态变量，那会怎么样呢？那样在同一个应用程序中就会有两个相同的全局对象类型——因此全局变量并不真正解决问题。

类方法提供了共享的服务，不用创建其对象就可以访问。资源的唯一实例可在类方法中维护。然而，如果类需要被子类化以提供更好的服务，这一方式就不够灵活。

单例类提供创建与访问类的唯一对象的访问点，并保证它唯一、一致而且为人熟知。这一模式提供了灵活性，使其任何子类可以重载实例方法并且完全控制自身的对象创建，而不必修改客户端的代码。更好的是，父类中的实例实现可以处理动态对象创建。类的实际类型可以在运行时决定，以保证创建正确的对象。这一技术将在本章稍后讨论。

单例模式有个变通版本，其中的一个工厂方法总是返回同一实例，但可以分配并初始化额外的实例。这个非“严格”版本的模式将在7.6.3节中讨论。

7.3 在 Objective-C 中实现单例模式

要把一个单例类设计得恰当，有些事情必须要考虑清楚。需要问的第一个问题是，如何保证类只能创建一个实例。其他面向对象语言（如C++和Java），可以把类的构造函数声明为`private`型，这样应用程序中的用户代码就无法创建类的对象。那么Objective-C如何呢？

每个Objective-C方法都是公有的，而且语言本身是动态类型的，因此所有类都可以互相发送对方的消息（C++和Java中的方法调用），而没有太多的编译时检查（只是在消息没有声明时才有编译警告）。而且Cocoa（包括Cocoa Touch）框架使用引用计数的内存管理方式来维护对象在内存中的生存期。所有这些特性使得在Objective-C中实现单例模式颇具挑战性。

在《设计模式》一书中的原始示例中，单例模式的C++例子如代码清单7-1所示。

代码清单7-1　出现在《设计模式》中的原始C++单例类

```
class Singleton
{
public:
  static Singleton *Instance();

protected:
  Singleton();

private:
  static Singleton *_instance;
};

Singleton *Singleton::_instance = 0;

Singleton *Singleton::Instance ()
{
  if (_instance == 0)
  {
    instance = new Singleton;
  }

  return _instance;
}
```

如书中描述的一样，C++的实现简单易懂。在静态的`Instance()`方法中，检查静态的`_instance`实例变量，看它是否为0（NULL）。如果是，会生成新的`Singleton`对象，然后其实例将被返回。有些人可能会以为Objective-C的版本不应该与其兄弟版本有多大差别，应该就像代码清单7-2与代码清单7-3这样。

代码清单7-2　Singleton.h中的`Singleton`的类声明

```
@interface Singleton : NSObject
{

}

+ (Singleton *) sharedInstance;

@end
```

代码清单7-3　Singleton.m中`sharedInstance`方法的实现

```
@implementation Singleton
```

```
static Singleton *sharedSingleton_ = nil;

+ (Singleton *) sharedInstance
{
  if (sharedSingleton_ == nil)
  {
    sharedSingleton_ = [[Singleton alloc] init];
  }
  return sharedSingleton_;
}

@end
```

如果真是这样的话，那么本章就是既好写又好读的一章，读者就已经学会了用Objective-C实现的这个模式。但实际上，需要克服一些障碍，才能让实现足够可靠，可以用在真正的应用程序中。如果需要实现单例模式的“严格”版本，要想在实际中使用，需要面对以下两个主要的障碍。

- 发起调用的对象（calling object）不能以其他分配方式实例化单例对象。否则，就有可能创建单例类的多个实例。
- 对单例对象实例化的限制应该与引用计数内存模型共存。

代码清单7-4显示的实现应该与我们所考虑的比较接近。代码相当长，所以将分成几个部分来讨论。

代码清单7-4 Objective-C的更合适的单例实现

```
#import "Singleton.h"

@implementation Singleton

static Singleton * sharedSingleton = nil;

+ (Singleton*) sharedInstance
{
  if (sharedSingleton_ == nil)
  {
    sharedSingleton_ = [[super allocWithZone:NULL] init];
  }

  return sharedSingleton_;
}
```

在sharedInstance方法中，跟第一个例子一样，首先检查类的唯一实例是否已创建，如果没有，就创建新的实例并将其返回。但是这回，它不是使用alloc这样的方法，而是调用[[super allocWithZone:NULL] init]来生成新的实例。为什么是super而不是self呢？这是因为已经在self中重载了基本的对象分配方法，所以需要“借用”其父类（即NSObject）的功能，来帮助处理底层内存分配的杂务。

```
+ (id) allocWithZone:(NSZone *)zone
{
  return [[self sharedInstance] retain];
}
```

```
- (id) copyWithZone:(NSZone*)zone
{
  return self;
}
```

有几个方法与内存管理有关，在单例类中需要注意。在allocWithZone:(NSZone *)zone方法中，只是返回从sharedInstance方法返回的类实例。在Cocoa Touch框架中，调用类的allocWithZone:(NSZone *)zone方法，会分配实例的内存，引用计数会置为1，然后会返回实例。我们已经见过alloc方法用于许多场合。其实，alloc是用设为NULL的zone来调用allocWithZone:，在默认区域（zone）为新实例分配内存。对象创建与内存管理的细节不在本书的讨论范围，更多的细节请参考文档。

类似地，需要重载copyWithZone:(NSZone*)zone方法，以保证不会返回实例的副本，而是通过返回self，返回同一个实例。

```
- (id) retain
{
  return self;
}

- (NSUInteger) retainCount
{
  return NSUIntegerMax; // 表示不能释放的对象
}

- (void) release
{
  // 什么也不做
}

- (id) autorelease
{
  return self;
}

@end
```

其他retain、release和autorelease等方法被重载，以确保它们（在引用计数内存模型中）什么也不做，只是返回self。retainCount返回NSUIntegerMax（4 294 967 295），以保证实例在应用程序的生存期中一直存在。

为什么保持单例

读者也许注意到了，在allocWithZone:中，保持了从sharedInstance方法返回的单例对象，但是retain已被重载而且在实现中基本被忽略。这样做，让我们有了把Singleton类变得不太“严格”的一种选择（即允许分配与初始化额外的实例，但sharedInstance工厂方法总是返回同一个实例，否则Singleton对象可被破坏）。子类可以再次重载retain、release和autorelease方法，实现合适的内存管理。

单例模式的变通版本将在7.6.3节中讨论。

关于Objective-C的单例应该是什么样子，已经讲了很多。然而在使用之前还有些其他事情需要认真考虑。如果要子类化最初的Singleton，该怎么做呢？现在来回答这个问题。

7.4 子类化 Singleton

alloc调用被转发给super，意味着NSObject会处理对象分配。如果不作修改地子类化Singleton，返回的实例总是Singleton。因为Singleton重载了所有实例化相关的方法，所以对其子类化相当需要技巧。但我们很幸运，我们可以使用一些基础（Foundation）函数，根据类的类型实例化任何对象。其中一个函数是id NSAllocateObject (Class aClass, NSUInteger extraBytes, NSZone *zone)。如果要实例化一个叫做Singleton的类的对象，可以这样做：

```
Singleton *singleton = [NSAllocateObject ([Singleton class], 0, NULL) init];
```

第一个参数是Singleton类的类型。第二个参数是用于索引的实例变量的额外字节数，它总是0。第三个参数用于指定内存中分配的区域，它一般为NULL，表示默认区域。可以通过指定类的类型，用这个函数实例化任何对象。这与子类化Singleton有什么关系呢？我们来回忆一下，最初的sharedInstance方法是这样的：

```
+ (Singleton*) sharedInstance
{
  if (sharedSingleton_ == nil)
  {
    sharedSingleton_ = [[super allocWithZone:NULL] init];
  }

  return sharedSingleton_;
}
```

如果用NSAllocateObject的技巧来创建实例，它会变成这样：

```
+ (Singleton *) sharedInstance
{
  if (sharedSingleton_ == nil)
  {
    sharedSingleton_ = [NSAllocateObject([self class], 0, NULL) init];
  }

  return sharedSingleton_;
}
```

因此，不管我们是在实例化Singleton还是其子类，这个版本都会处理得很好。

7.5 线程安全

如果单例对象要由多个线程访问，那么使它线程安全（thread-safe）至关重要。例子中的Singleton类只能胜任一般用途。要让它线程安全，需要在sharedSingleton_静态实例的nil检查周围加入一些@synchronized()程序块或者NSLock实例。如果有其他属性需要保护，也可以把它们声明为atomic型。

7.6　在 Cocoa Touch 框架中使用单例模式

在查阅Cocoa Touch开发文档时，会发现框架中随处可见大量的单例类。本节将讨论其中几个——UIApplication、UIAccelerometer和NSFileManager。

7.6.1　使用 UIApplication 类

框架中极为常用的一个单例类是UIApplication类。它提供了一个控制并协调iOS应用程序的集中点。

每个应用程序有且仅有一个UIApplication的实例。它由UIApplicationMain函数在应用程序启动时创建为单例对象。之后，对同一UIApplication实例可以通过其sharedApplication类方法进行访问。

UIApplication对象为应用程序处理许多内务管理任务（housekeeping task），包括传入的用户事件的最初路由，以及为UIControl分发动作消息给合适的目标对象。它还维护应用程序中打开的所有UIWindow对象的列表。应用程序对象总是被分配一个UIApplicationDelegate对象。应用程序将把重要的运行时事件通知给它，比如iOS应用程序中的应用程序启动、内存不足警告、应用程序终止和后台进程执行。这让委托（delegate）有机会作出适当的响应。

7.6.2　使用 UIAccelerometer 类

Cocoa Touch框架中另一个常用的单例是UIAccelerometer。UIAccelerometer类让应用程序可以进行注册，以接收来自iOS设备内置的加速度计的加速度相关数据。应用程序会收到三维空间中沿主轴的线性加速度变化，可以使用这一数据检测设备的当前方向和当前方向的瞬间变化。

UIAccelerometer是单例，所以不能直接生成它的对象。而是应该调用其shared-Accelerometer单例类方法以访问它的唯一实例。然后设定它的updateInterval属性，并用自己的delegate对象设定delegate属性，以接收来自单例实例的加速度数据。

7.6.3　使用 NSFileManager 类

在Mac OS X v10.5和iOS 2.0之前，NSFileManager曾经是单例模式的“严格”实现。调用它的init方法是空操作（什么也不做），并且其唯一实例可以通过defaultManager类方法进行创建和访问。然而其单例实现不是线程安全的。现在推荐生成新的NSFileManager实例以保证线程安全。这一方式被认为是更灵活的单例实现，其中工厂方法总是返回同一实例，但是也可以分配并初始化另外的实例。

如果需要实现“严格”的单例，需要类似于前面几节中描述的例子的实现。否则，不要重载allocWithZone:和其后的其他方法。

7.7 总结

几乎在任何类型的应用程序中，单例模式都极为常用，并不只限于iOS应用程序开发。

只要应用程序需要用集中式的类来协调其服务，这个类就应生成单一的实例，而不是多个实例。

本章标志着关于对象创建这一部分的结束。在下一个部分要讨论的一些设计模式，旨在适配或结合带有不同接口的对象。

Part 3 第三部分

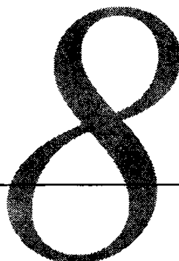
接口适配

本部分内容

- 第8章　适配器
- 第9章　桥接
- 第10章　外观

第8章

适　配　器



早在19世纪，尼古拉·特斯拉（Nikola Tesla）发明了交流电力系统。他也许想象不到，我们要把美国用的电气设备插到欧洲墙上的插座，却没有适配器时的烦恼。设想我在欧洲一个旅馆的卫生间里，手拿一个电动剃须刀，看着墙上的插座，却忘了带适配器。“现在该怎么办？”“应该在这儿买个新剃须刀吗？”“可是带回美国后还得给它弄个适配器。”

在面向对象软件设计中，有时候我们想把有用而经过精心测试的类，用于应用程序的其他新领域。但是，新功能需要新接口，而新接口与要复用的现有类不一致的情况非常普遍。此时你也许会问自己：“现在该怎么办？”“应该改写这些类去适应新的接口吗？”“可是当我又要加新功能时，我还要再改写吗？”人是聪明的，发明了适配器，可以把美国用插头插到欧洲的插座，或者反过来。在交流电发明多年以后，电源适配器仍有巨大的需求。没人愿意每出一次国就买个新电动剃须刀。同样，我们也不想为新的接口而重写可靠的类。

已有的类与新的接口之间不兼容的问题相当普遍，人们已为它找到了一个解决方案。这个解决方案广为使用，最终被编入设计模式，称为适配器。

8.1　何为适配器模式

适配器模式，可以这么说，用于连接两种不同种类的对象，使其毫无问题地协同工作。有时它也称为“包装器”（wrapper）。其思想相当简单。适配器实现客户端所要的某种接口的行为。同时，它又连接到另一个具有（完全）不同接口与行为的对象。一边是客户端懂得如何使用的目标接口，另一边是客户端一无所知的被适配者，适配器站在两者之间。适配器的主要作用是把被适配者的行为传递给管道另一端的客户端。

基本上有两种实现适配器的方式。第一种是通过继承来适配两个接口，这称为类适配器。最早在《设计模式》一书中，类适配器是通过多重继承实现的。书中主要使用的语言是C++，它并没有Java的接口或Objective-C的协议这样的语言功能，一切都是类。没有实际实现的类，叫做抽象类。它是一种被用作“抽象”类型的东西，与Objective-C不同。Objective-C有协议，作为纯粹抽象的一种形式。在Objective-C中，类可以实现协议，同时又继承超类，达到C++的多重继承的效果。讨论语言功能的书很多，这里就不去讨论了。要在Objective-C中实现类适配器，首先需要有定义了客户端要使用的一套行为的协议，然后要用具体的适配器类来实现这个协议。适配器类

同时也要继承被适配者。它们之间的关系如图8-1所示。

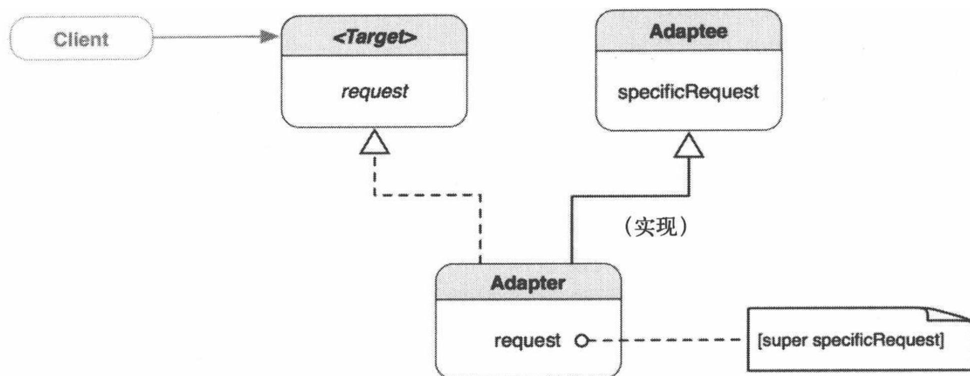


图8-1 类适配器的类图

Adapter是一个Target[①]类型，同时也是一个Adaptee[②]类型。Adapter重载Target的request[③]方法。但是Adapter没有重载Adaptee的specificRequest方法，而是在Adapter的request方法的实现中，调用超类的specificRequest方法。request方法在运行时向超类发送[super specificRequest]消息。super就是Adaptee，它在Adapter的request方法的作用域内，按自己的方式执行specificRequest方法。只有当Target是协议而不是类时，类适配器才能够用Objective-C来实现。

实现适配器模式的第二种方式称为对象适配器。与类适配器不同，对象适配器不继承被适配者，而是组合了一个对它的引用。实现为对象适配器时，它们之间新的关系如图8-2所示。

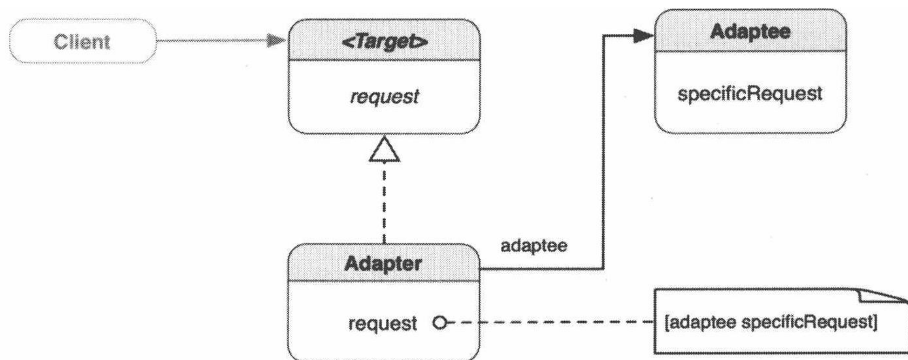


图8-2 对象适配器的类图

Target和Adapter之间的关系跟图8-1中的类适配器相同，而Adapter和Adaptee之间的关

① Target指目标接口。——译者注

② Adaptee意为“被适配者”。——译者注

③ request意为“请求”。——译者注

系从“属于”变成了“包含”。这种关系下，Adapter需要保持一个对Adaptee的引用。在request方法中，Adapter发送[adaptee specificRequest]消息给引用adaptee，以间接访问它的行为，然后实现客户端请求的其余部分。由于Adapter与Adaptee之间是一种“包含”的关系，用Adapter去适配Adaptee的子类也没什么问题。

适配器模式：将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。*

* 最初的定义出现于《设计模式》(Addison-Wesley, 1994)。

类适配器与对象适配器的对比

类适配器与对象适配器是实现适配器模式的不同方式，但是达成的目的相同。设计中采用适配器模式时应该选择哪一种呢？请看下表，以对其特征有个快速了解。

类适配器与对象适配器的特征对比

类适配器	对象适配器
只针对单一的具体Adaptee类，把Adaptee适配到Target	可以适配多个Adaptee及其子类
易于重载Adaptee的行为，因为是通过直接的子类化进行的适配	难以重载Adaptee的行为，需要借助于子类的对象而不是Adaptee本身
只有一个Adapter对象，无需额外的指针间接访问Adaptee	需要额外的指针以间接访问Adaptee并适配其行为

显然，委托（Delegate）模式属于对象适配器。

8.2 何时使用适配器模式

在以下情形，自然会想到使用这一模式。

- 已有类的接口与需求不匹配。
- 想要一个可复用的类，该类能够同可能带有不兼容接口的其他类协作。
- 需要适配一个类的几个不同子类，可是让每一个子类去子类化一个类适配器又不现实。那么可以使用对象适配器（也叫委托）来适配其父类的接口。

8.3 委托

前面几章讨论过Cocoa Touch框架中的委托（delegation）。iOS开发人员已经在框架SDK中随处可见delegate。根据苹果公司的文档，delegate是Cocoa Touch框架所采用的委托模式的一种形式。那么，什么是委托模式呢？

委托模式曾经对GoF在书中收录适配器模式起到启发作用。那么两者有何联系呢？再来想想适配器模式做了什么：把类的接口变换为客户端要求的另一种接口。这里的客户端它们是什么

呢？它们是Cocoa Touch框架中的类。那么此处什么是Target呢？是一个委托协议。实现协议的具体类会是个适配器。那么什么是与框架不匹配而需要适配的类呢？应用程序中的其他类。现在你明白了为何委托模式其实是适配器模式。

我之所以说委托模式主要是适配器模式，是因为委托机制也可以实现某些其他设计模式的意图，比如装饰模式（第16章）。Cocoa Touch框架中委托模式的实现有时会跟其他设计模式混在一起。比如，有些实现了委托模式的框架类，也是模板方法模式（第18章）的一部分。模板方法包含一套预定义的参数化的算法，留出某些特定行为要求子类来提供，但是在这种情况下特定行为是委托。模板方法的执行中，必要时会向委托（适配器）发送消息以请求某些特定行为。然后可以通过适配器（委托）的接口，从应用程序中的被适配者获取任何特定信息。委托模式用于多个设计模式的混合体并与其相互关联的情况很常见。

8.4 用 Objective-C 协议实现适配器模式

我们在Cocoa Touch框架中见过的许多框架类，是用协议中定义的某种形式的委托来实现的。我们可以把自己的委托实现为适配器。

在第2章中TouchPainter应用程序设计的讨论中，我们知道有一个视图，它允许用户改变在CanvasView中绘图的线色与线宽。收到请求时，PaletteViewController的实例显示这个视图。视图中有3个滑动条，用于改变线色的颜色分量。这一操作涉及几个组件——CanvasView-Controller、PaletteViewController及其滑动条。你也许以为可能几行代码就能搞定。但是缺点是一切都紧紧耦合在一起，难以复用。需要一个更好的方案，让我们能够把颜色变更的部分复用到应用程序的其他地方。例如，即使把选取颜色的接口变为色轮，取代表示各个颜色分量的滑动条，仍然可以复用相同的对象，完成同样的事情。类似地，假如要实现TouchPainter的iPad版本，也可以把相同的变更颜色的对象放到弹出菜单中，而iPhone不支持弹出菜单。

如果要复用某个东西，从面向对象编程的角度来说，就要把那个东西放到对象中。如果要把颜色变更动作放到对象中，一种很自然的方式是把它放到命令对象中（见命令模式，第20章）。命令对象可被放入队列并复用，可以对封装的操作进行撤销和恢复。在第20章，我们会对Cocoa使用NSInvocation实现命令模式作讲解。NSInvocation对象足够通用，可以把任何Objective-C对象作为目标，把任何消息作为选择器，封装和调用目标-动作（target-action）单元。但是NSInvocation的一个限制是，无法将多个操作放入其对象中进行调用。多个操作可以简单如验证用户所输入的密码，如果验证未通过，就显示警告框。这些操作没有必要塞进某个控制器，尤其是当它们非常通用，可以在多处被复用时。出于这一原因，有时还是需要自己的命令对象。我们既可以子类化NSInvocation，也可以从头创建自己的版本。第20章里对后者做了详细讲解。

线色变更机制的设计与实现

SetStrokeColorCommand实现了命令模式。它所做的是把线色值设给CanvasView-Controller，这样在CanvasView上画的下一条线就会使用设定的颜色。你可以回到第2章以及

其他各章，看看有关CanvasViewController和CanvasView的详细讨论。当SetStrokeColor-Command的实例被执行时，它需要RGB值以构建颜色值，作为其操作的一部分。SetStrokeColor-Command对象不在乎由谁提供这些值。这些值应该从任何采用了委托协议SetStrokeColor-CommandDelegate的适配器来获取。它们之间的静态关系如图8-3所示。

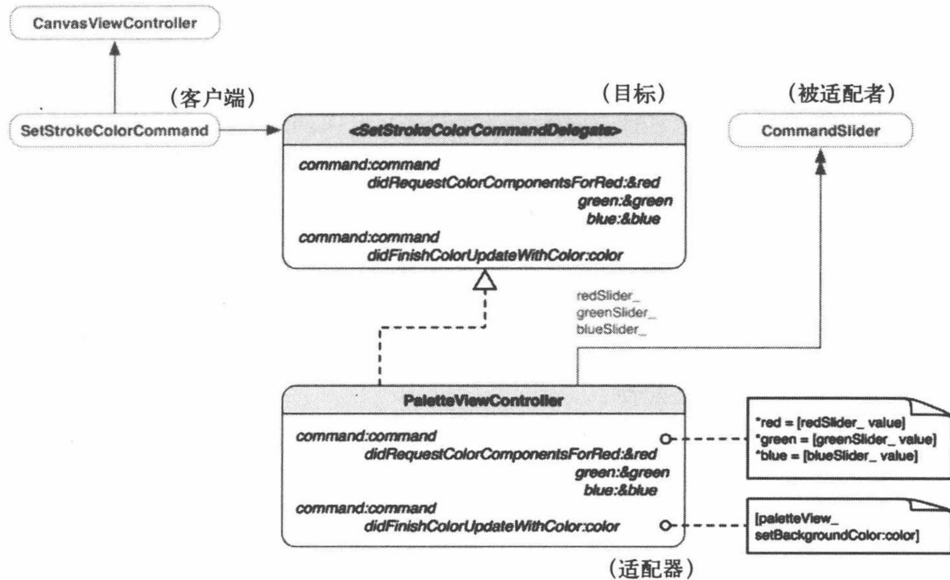


图8-3　SetStrokeColorCommand是客户端，SetStrokeColorCommandDelegate是它的Target[①]。PaletteViewController通过转换从CommandSlider实例得到的值，起到SetStrokeColorCommandDelegate的适配器的作用

这里SetStrokeColorCommand是客户端，因为是它能够向满足需要的任何东西请求用于完成任务的数据。SetStrokeColorCommand是第20章中定义的Command的子类。Command的定义如代码清单8-1所示。

代码清单8-1　Command.h

```
@interface Command : NSObject
{
  @protected
  // 其他私有成员变量……
}

// 其他属性……

- (void) execute;

@end
```

① 指目标接口。——译者注

我们不打算讨论Command类中声明的所有方法和属性，而是在实现SetStrokeColor-Command类时重点讨论execute方法。SetStrokeColorCommand需要一个目标接口，告诉适配器要提供什么。

1. 定义目标接口SetStrokeColorCommandDelegate

跟在Cocoa Touch框架中看到的其他委托一样，SetStrokeColorCommandDelegate也定义为协议。它起到客户端与适配器之间的合约的作用，在这里是SetStrokeColorCommand与PaletteViewController之间的合约。SetStrokeColorCommandDelegate的协议声明如代码清单8-2所示。

代码清单8-2 SetStrokeColorCommand.h中定义的SetStrokeColorCommandDelegate协议，它是适配器模式中的目标接口

```
#import <Foundation/Foundation.h>
#import "Command.h"

@class SetStrokeColorCommand;

@protocol SetStrokeColorCommandDelegate

- (void) command:(SetStrokeColorCommand *) command
                didRequestColorComponentsForRed:(CGFloat *) red
                                          green:(CGFloat *) green
                                          blue: (CGFloat *) blue;

- (void) command:(SetStrokeColorCommand *) command
                didFinishColorUpdateWithColor:(UIColor *) color;

@end
```

command:didRequestColorComponentsForRed:green:blue:方法通过传递红绿蓝引用参数返回独立的RGB值。另一方面，当颜色更新过程结束时，command:didFinishColor-UpdateWithColor:将被调用。命令对象会把自己和更新后的颜色对象传给适配器，这样它就可以利用这个时机用新颜色作任何其他处理。

2. 实现客户端SetStrokeColorCommand

轮到客户端SetStrokeColorCommand了。它用delegate_保持SetStrokeColorCommand-Delegate的引用，见代码清单8-3中的它的类声明。

代码清单8-3 在SetStrokeColorCommand.h中SetStrokeColorCommand的类声明

```
@interface SetStrokeColorCommand : Command
{
  @private
  id <SetStrokeColorCommandDelegate> delegate_;
}

@property (nonatomic, assign) id <SetStrokeColorCommandDelegate> delegate;

- (void) execute;

@end
```

由于这是个命令对象，你可能想知道委托是否应该是个接收器，像第20章中的命令模式中定义的那样，接收来自命令对象的任何动作消息。事实上，真正的命令接收器并不在此处定义为属性本身，而应在execute方法中获得，如代码清单8-4所示。

代码清单8-4　在SetStrokeColorCommand.m中SetStrokeColorCommand的实现

```
#import "SetStrokeColorCommand.h"
#import "CoordinatingController.h"
#import "CanvasViewController.h"

@implementation SetStrokeColorCommand

@synthesize delegate=delegate_;

- (void) execute
{
  CGFloat redValue = 0.0;
  CGFloat greenValue = 0.0;
  CGFloat blueValue = 0.0;

  // 从委托取得RGB值
  [delegate_ command:self didRequestColorComponentsForRed:&redValue
                                                    green:&greenValue
                                                     blue:&blueValue];

  // 根据RGB值创建一个颜色对象
  UIColor *color = [UIColor colorWithRed:redValue
                                   green:greenValue
                                    blue:blueValue
                                   alpha:1.0];

  // 把它赋值给当前canvasViewController
  CoordinatingController *coordinator = [CoordinatingController sharedInstance];
  CanvasViewController *controller = [coordinator canvasViewController];
  [controller setStrokeColor:color];

  // 转发更新成功消息
  [delegate_ command:self didFinishColorUpdateWithColor:color];
}

- (void) dealloc
{
[super dealloc];
}

@end
```

delegate_发出一个command:didRequestColorComponentsForRed:green:blue:消息，以取得独立的RGB值。然后根据得到的值创建一个颜色对象。接着从Coordinating-Controller取得CanvasViewController的唯一引用。CoordinatingController的单例实例起中介者作用，协调应用程序中的各个视图控制器（见中介者模式，第11章）。我们把上一步

中合并好的颜色赋给CanvasViewController，作为其新的线色。之后，delegate_被用另一个委托方法使用新的颜色对象再次触发。所选的delegate_对象会接管，并使用颜色对象作些其他的处理。

现在我们知道了客户端和目标接口是什么。来看看应该为客户端适配什么对象及这些对象的值。

3. 创建被适配者CommandSlider

我们已经多次提到，有3个滑动条用于调整颜色分量。滑块的每个细小的移动，会用新的值触发对接收器的更新。这个场景让人想到命令模式（第20章）中定义的调用器（invoker）。调用器保持命令对象的引用，并且在被调用（比如用户按了按钮）时执行它。所以滑动条也需要保持对命令对象的引用，以便对其进行某些处理（比如，更新颜色）。然而，UISlider不会考虑到要跟命令对象一起使用，所以必须通过子类化UISlider创建自己的版本。我们把它叫做CommandSlider，见代码清单8-5。

代码清单8-5 CommandSlider.h

```
#import <Foundation/Foundation.h>
#import "Command.h"

@interface CommandSlider : UISlider
{
  @protected
  Command *command_;
}

@property (nonatomic, retain) IBOutlet Command *command;

@end
```

接受并执行Command对象的定制滑动条并不足以解决我们的问题。需要把它关联到SetStrokeColorCommand对象。可以看到，command属性是IBOutlet型，就是说可以用Interface Builder来完成关联。第11章对关联UI元素和定制对象有详细讲解。

我们让滑动条监听Value Changed（值已变更）事件。当滑动条的滑块变动时，运行库会使用应该被调用的“目标–动作”触发这一事件。滑动条然后会执行保存的命令对象。我们将在下节接触这部分的细节。

4. 使用适配器PaletteViewController

最后，轮到了这一设计中的适配器。要适配来自RGB滑动条的数值，PaletteViewController是个自然的选择，因为控制器拥有并控制着滑动条。为了给客户端（即SetStrokeColorCommand对象）作任何适配，PaletteViewController类需要遵守SetStrokeColorCommandDelegate协议。来看看代码清单8-6中被采用的委托方法。

代码清单8-6 由PaletteViewController实现的SetStrokeColorCommandDelegate方法

```
#pragma mark -
```

```
#pragma mark SetStrokeColorCommandDelegate methods

- (void) command:(SetStrokeColorCommand *) command
               didRequestColorComponentsForRed:(CGFloat *) red
                                         green:(CGFloat *) green
                                          blue: (CGFloat *) blue
{
  *red = [redSlider_ value];
  *green = [greenSlider_ value];
  *blue = [blueSlider_ value];
}

- (void) command:(SetStrokeColorCommand *) command
               didFinishColorUpdateWithColor:(UIColor *) color
{
  [paletteView_ setBackgroundColor:color];
}
```

当客户端（SetStrokeColorCommand对象）请求新的RGB值，并调用其委托或适配器的command:didRequestColorComponentsForRed:green:blue:方法时，PaletteViewController作出响应，把相应滑动条的值赋给每个颜色分量。同样，当SetStrokeColorCommand对象完成线色的更新之后，会对其适配器触发另一个方法，用新的颜色对象执行任何附加操作。PaletteViewController响应这一消息，用新的颜色更新paletteView_的背景色，这里paletteView_是颜色滑动条下面的小调色板视图。

现在，我们对于如何把东西组合在一起已有了比较清楚的认识。可是在代码中如何使用CommandSlider实例呢？答案是不直接使用它，而是借助Interface Builder，在一个.xib文件中把颜色滑动条挂接到PaletteViewController。

当颜色滑动条中任何一个的值有变化时，通过调用PaletteViewController中定义的onCommandSliderValueChanged:方法，这个事件会被捕捉到，见代码清单8-7。

代码清单8-7　PaletteViewController.m中的用于所有CommandSlider实例的onCommandSliderValueChanged:事件处理器

```
#pragma mark -
#pragma mark Slider event handler

- (IBAction) onCommandSliderValueChanged:(CommandSlider *)slider
{
  [ [slider command] execute];
}
```

PaletteViewController中这一小小的事件处理方法，起到让所有类型的CommandSlider对象用自己内嵌的Command对象以同样方式运行的作用。这个方法返回的每个滑动条会调用其Command对象的execute方法。在这里，SetStrokeColorCommand的execute方法接下来会从适配器（PaletteViewController）获取RGB值。一旦委托返回了RGB值，SetStrokeColorCommand就会继续更新CanvasViewController的线色，见代码清单8-4。

看！一切都连结在同一流水线中。对象各司其责并高度可复用。要是以后决定修改调整线色

的方式，仍然可以复用同一个SetStrokeColorCommand，达到同样目的，而不必再次改写这一部分。onCommandSliderValueChanged:事件处理器也可用于带有不同命令对象的其他滑动条。明白它们是如何联系在一起而又不彼此依赖的吗？

8.5 用 Objective-C 的块在 iOS 4 中实现适配器模式

我们已经看到，适配器模式可以用作为目标接口的协议来实现，协议定义客户端要求的，也是适配器可以保证的一些标准行为。协议是Objective-C中语言级别的功能，通过它可以定义用作适配器模式的实例的接口。术语“接口”在Java等其他面向对象语言中是“协议”的同义词。用于适配器模式的接口，本质上是一系列方法声明，客户端懂得如何与其进行交流。但是通过协议，适配器与客户端都知道定义为协议的目标接口，以使得一切都得到明确定义并正确运行。称为块（block）的语言功能是Objective-C中的一个强大的功能，它让我们可以不使用协议就能实现适配器模式。从iOS 4的出现开始可以使用块。

块与函数类似，但是它在其他函数里面，跟其他代码写在一起。块的出现已有一阵子了。它是Ruby、Python、Smalltalk及Lisp等脚本或编程语言的一部分。有时块被称为closure（闭包）或lambda。它被称为“闭包”是因为包围了作用域里的变量。本节不讨论块的语言细节，但会简要介绍一些基本语法以及如何用它来实现适配器模式。

8.5.1 块引用的声明

块变量对可在程序中传递的代码“块”保持一个引用。它跟C语言中的函数指针类似，如下所示：

```
int (^ObjectiveCBlock)(int);
int (*CFunctionPointer)(float);
```

第一行是Objective-C块变量，接受一个整数型参数，返回一个整数。语法跟第二行中的C语言版本非常类似。C函数指针变量接受浮点数型变量，返回一个整数。语法上的主要区别是，C函数指针使用*字符，而Objective-C块使用^字符。

要是在多处使用同一个块，最好用typedef创建一个类型，而不要在每次定义时都敲出同一个难懂的签名（signature）[①]：

```
typedef int (^ObjectiveCBlock)(int);
```

这样以后就可以像下面这样定义：

```
ObjectiveCBlock firstBlock = ...;
ObjectiveCBlock secondBlock = ...;
```

使用了更简单的类型名，而不是完整的块签名后，现在好读多了。下面讨论一下如何定义块字面量（block literal）[②]。

① signature，在这里指函数声明去掉了名字后剩下的那些东西，包括参数个数、类型、返回值。——译者注

② literal指代码中表示固定值的符号。——译者注

8.5.2　块的创建

前一节定义了块变量ObjectiveCBlock[①]，并把它声明为类型。要在里面定义实际的块字面量（块作为函数的实际内容），可以像下面这样做：

```
ObjectiveCBlock aBlock = ^(int anInt) {
        return anInt++;
}
```

然后它可在应用程序的其他部分当做函数使用。

```
int result = aBlock(5);
```

令人惊奇的是，块实际上是Objective-C对象。块被看做栈中的匿名对象，离开了作用域就会被销毁。对块可以像其他普通Objective-C对象一样进行copy、retain、release和autorelease。但是retain对于块好像没有实际作用。要是想“retain”块，需要把它移到堆中，做个“副本”。跟普通Objective-C对象一样，需要用release/autorelease抵消retain/copy。对于刚才定义的块，可以这样进行copy和release：

```
id aBlockCopy = [ ^(int anInt) { return anInt++; } copy];

// 用aBlockCopy做点儿事儿……

[ aBlockCopy release];
```

或者这样：

```
id aBlockCopy = [aBlock copy];

// 用aBlockCopy做点儿事儿……

[aBlockCopy release];
```

或者用这个更简单的方式，尤其是想从别的方法或函数中返回块的时候：

```
id aBlockCopy = [[aBlock copy] release];

// 用aBlockCopy做点儿事儿……
```

每种写法都达到同样的效果。明白了吧。

C函数指针与Objective-C块的一个主要区别是块字面量可以与消息调用定义在一起：

```
[anObject doSomethingWithBlock: ^(int anInt) {
        return anInt++;
}];
```

但是用C函数指针却无法做到这一点。现在你一定想知道如何以块为构件来实现适配器模式。

8.5.3　把块用作适配器

---

① 其实ObjectiveCBlock只是类型，不是变量，下面代码中的aBlock才是变量。——译者注

我们记得在TouchPainter应用程序中`PaletteViewController`的设定视图上，有3个滑动条让用户改变线色。每个滑动条分别对应于iOS中颜色对象的RGB值的红、绿、蓝的值。比方说，有人想往当前颜色中添加更多红色，可以移动红色对应的滑动条，对应其他颜色分量的滑动条也是如此。

在上一个部分，我们用协议定义了适配器要采用的目标接口。这回，我们用块来实现同样用途的适配器。新适配器的类图如图8-4所示。

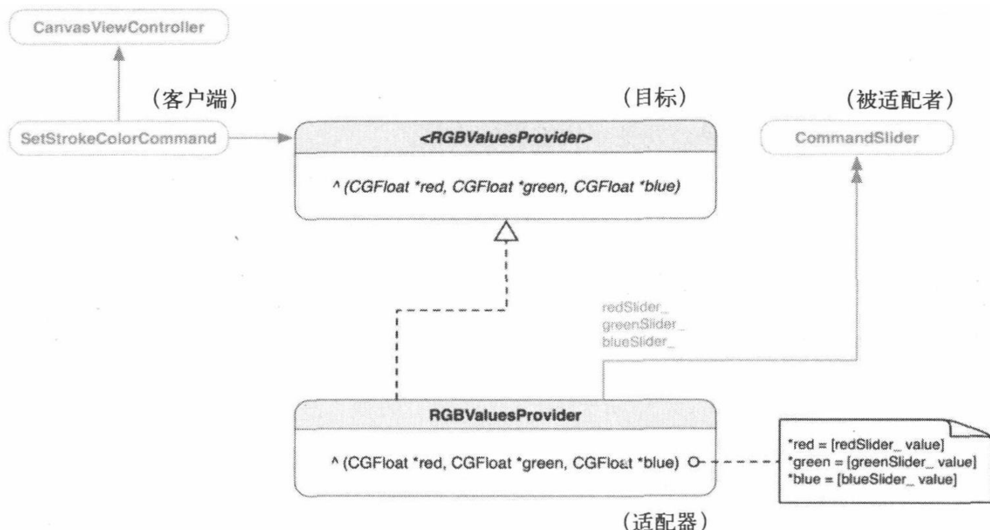


图8-4 使用对象组合适配`UISlider`实现`RGBValuesProvider`块作为`StrokeColorCommand`的对象适配器的类图

从图中可看到，定义了一个叫`RGBValuesProvider`的块，签名为`CGFloat^(CGFloat *red, CGFloat *green, CGFloat *blue)`。签名中的每个参数都是指向`CGFloat`值的指针。在向它赋予实际的块字面量之前，它只是个类型而已。块字面量应该符合块签名，并提供某些处理的实现。在图8-4中，有个块字面量`RGBValuesProvider`，它包含跟前面相同的`redSlider_`、`greenSlider_`和`blueSlider_`。当块在操作中被调用时，它会从滑动条取得值并赋给块签名中定义的指针，把值传回给调用者。来看看代码是什么样子（代码清单8-8）。

代码清单8-8 SetStrokeColorCommand.h

```
#import <Foundation/Foundation.h>
#import "Command.h"

typedef void (^RGBValuesProvider)(CGFloat *red, CGFloat *green, CGFloat *blue);

@interface StrokeColorCommand : Command
{
  @private
  RGBValuesProvider RGBValuesProvider_;
}
```

```
@property (nonatomic, copy) RGBValuesProvider RGBValuesProvider;

- (void) execute;

@end
```

在SetStrokeColorCommand的头文件中，用typedef定义RGBValuesProvider块，这样在其他地方就可以只用名字来称呼这个块，就好像它是个Objective-C对象类型一样。SetStrokeColorCommand有个刚定义的块类型的私有成员变量，而且这个变量与一个设置了copy特性的属性值相关联。当块赋值给SetStrokeColorCommand对象时需要进行复制，因为通常块字面量定义并用于方法的作用域中。当方法离开了作用域，块也就不存在了。有关块的内存管理的详细讨论，请参考苹果公司关于块编程的文档。

SetStrokeColorCommand是个Command对象，因此它有个继承来的execute方法的定义。在代码清单8-9中可以看到它的execute方法做了什么。

代码清单8-9 StrokeColorCommand.m

```
#import "StrokeColorCommand.h"
#import "CoordinatingController.h"
#import "CanvasViewController.h"

@implementation StrokeColorCommand

@synthesize RGBValuesProvider=RGBValuesProvider_;


- (void) execute
{
  CGFloat redValue = 0.0;
  CGFloat greenValue = 0.0;
  CGFloat blueValue = 0.0;

  // 从块取得RGB值
  if (RGBValuesProvider_ != nil)
  {
    RGBValuesProvider_(&redValue, &greenValue, &blueValue);
  }

  // 根据RGB值创建一个颜色对象
  UIColor *color = [UIColor colorWithRed:redValue
                                   green:greenValue
                                    blue:blueValue
                                   alpha:1.0];

  // 把它赋值给当前canvasViewController
  CoordinatingController *coordinator = [CoordinatingController sharedInstance];
  CanvasViewController *controller = [coordinator canvasViewController];
  [controller setStrokeColor:color];
}
```

```
- (void) dealloc
{
  [RGBValuesProvider_ release];
  [super dealloc];
}

@end
```

当SetStrokeColorCommand的实例被执行时，它通过调用块RGBValuesProvider_(&redValue, &greenValue, &blueValue)取得颜色分量的值。根据返回值创建新的UIColor对象，然后SetStrokeColorCommand把颜色对象设置给CanvasViewController的实例。

那么在代码中将如何使用SetStrokeColorCommand呢？让我们从代码清单8-10中PaletteViewController的代码中找到答案。

代码清单8-10 PaletteViewController.m中的viewDidLoad方法

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  // 用StrokeColorCommand初始化RGB滑动条
  StrokeColorCommand *colorCommand;

  // 得到用于改变颜色的命令的引用……

  // 把各个颜色分量的提供者设置给颜色命令
  [colorCommand setRGBValuesProvider: ^(CGFloat *red, CGFloat *green, CGFloat *blue)
   {
     *red = [redSlider_ value];
     *green = [greenSlider_ value];
     *blue = [blueSlider_ value];
   }];
}
```

我们已经分别为每个颜色滑动条定义了一个成员变量——redSlider_、greenSlider_和blueSlider_。它们都在Interface Builder中进行了连接与设定。最后，定义一个块字面量，把从这些滑动条得到的值作为颜色分量值返回，然后把块字面量赋给被这些滑动条引用的colorCommand对象。

好像我们并没有为滑动条定义任何事件处理方法。但是，实际上，我们打算复用上节中定义的那个，因为它们都是同一CommandSlider类型并且用法相同。因此只需要简单地在Interface Builder中把这些颜色滑动条挂接到PaletteViewController中的同一个onCommandSlider-ValueChanged:方法。当然，你也可以添加另一个块，让适配器对新的颜色对象做额外的处理，就像前面讲的委托方式那样。

搞定！跟使用协议的较为传统的方式相比，你觉得使用块作为适配器怎么样？使用块，几乎可以在任何地方进行块的定义，然后让接收器（SetStrokeColorCommand）在以后使用它。它也不影响所涉及类的任何继承，因为适配器是定义于viewDidLoad方法之中。跟涉及更为正式

的协议与类相比，这种方式使得代码和结构更加简洁。适配器更加直接而随意，却又保持了适配器模式的原汁原味。

8.6　总结

这一章相对较长。现在我们懂得了如何使用Objective-C协议来实现适配器模式。尽管委托模式本身可以达到多种目的，不只用于适配器模式，但是它首先对适配器模式起了重要的启发作用。本章也探讨了如何利用Objective-C的块功能，用块作为适配器来实现适配器模式。对iOS开发来说块是个新功能，但块是一项功能强大的语言特性。我们应该利用这一特性，拓展将来的可能性。

适配器模式是一种极为常用的设计模式，可以时常于代码中看到它。用于解释模式的例子只是TouchPainter应用程序中的几个而已。实际开发中，像把适配器和命令对象一起使用的例子那样，各种模式同时用于某个解决方案的情况并不罕见。去下载一份代码看看在程序里还能再找到多少适配器。

下一章会介绍另一种设计模式，它帮助消除抽象接口与实现的耦合，从而接口可以独立地进行变更。

第9章 桥 接

比方说有一家电视机制造商，他们生产的每台电视机都带一个遥控器，用户可以用遥控器进行切换频道之类的操作。这里，遥控器是控制电视机的接口。如果每个电视机型号需要一个专用的遥控器，那么单是遥控器就会导致设计激增。不过，每个遥控器都有些功能是各种型号电视机所共有的，比如切换频道、调节音量和电源开关。而且每台电视机都应该能够通过基本命令接口，响应遥控器发来的这些命令。我们可以把遥控器逻辑同实际的电视机型号分离开来。这样电视机型号的改变就不会对遥控器的设计有任何影响。遥控器的同一个设计可以被复用和扩展，而不会影响其他电视机型号。

在面向对象的软件设计中也有类似的情形。例如，比方说想要设计在不同操作系统上显示同一类型窗口的接口。多数时候，基本的窗口由线和矩形构成。操作系统A画线和矩形的方式不同于操作系统B。如果对每个类型的窗口都进行具体的实现，激增的类层次结构会大得惊人。解决这个问题的一个办法是，从针对不同操作系统的每个实现中，分离出不同窗口类型的抽象。帮助解决这一设计问题的设计模式称为桥接（Bridge）模式。

9.1 何为桥接模式

桥接模式的目的是把抽象层次结构从其实现中分离出来，使其能够独立变更。抽象层定义了供客户端使用的上层的抽象接口。实现层次结构定义了供抽象层次使用的底层接口。实现类的引用被封装于抽象层的实例中时，桥接就形成了，如图9-1所示。

`Abstraction`是定义了供客户端使用的上层抽象接口的父接口。它有一个对`Implementor`实例的引用，`Implementor`定义了实现类的接口。这个接口不必跟`Abstraction`的接口一致，其实两个接口可以相当不同。`Implementor`的接口提供基本操作，而`Abstraction`的上层操作基于这些基本操作。当客户端向`Abstraction`的实例发送`operation`消息时，这个方法向`imp`发送`operationImp`消息。底下的实际`ConcreteImplementator`（A或B）将作出响应并接受任务。

因此想要往系统中添加新的`ConcreteImplementator`时，所要做的只是为`Implementor`创建一个新的实现类，响应`operationImp`消息并在其中执行任何具体的操作。不过，这对`Abstraction`方面不会有任何影响。同样，如果想修改`Abstraction`的接口或者创建更细化的`Abstraction`类，也能做到不影响桥接的另一头。

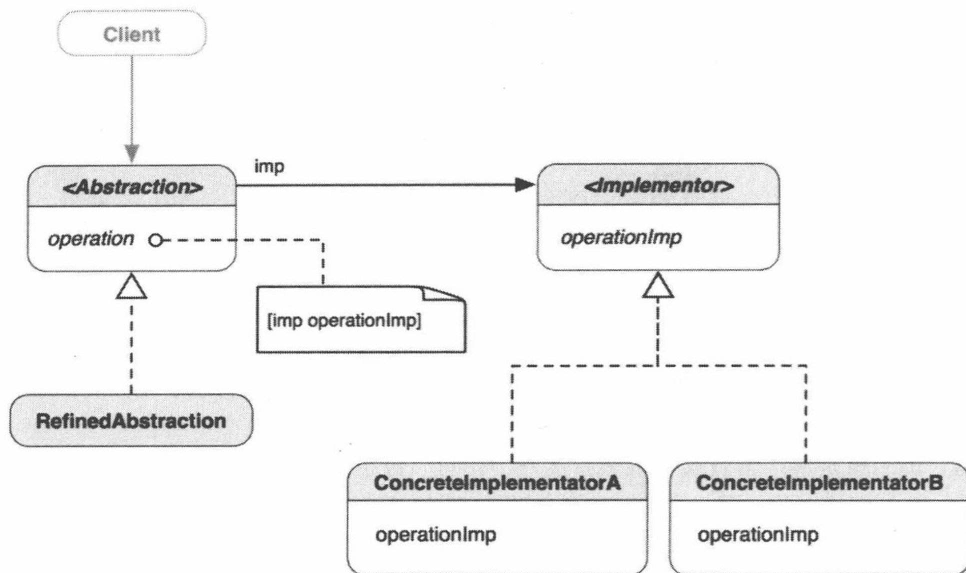


图9-1 桥接模式的类图

桥接模式：将抽象部分与它的实现部分分离，使它们都可以独立地变化。*

* 最初的定义出现于《设计模式》(Addison-Wesley，1994)。

9.2 何时使用桥接模式

在以下情形，自然会想到使用这一模式：

- 不想在抽象与其实现之间形成固定的绑定关系（这样就能在运行时切换实现）；
- 抽象及其实现都应可以通过子类化独立进行扩展；
- 对抽象的实现进行修改不应影响客户端代码；
- 如果每个实现需要额外的子类以细化抽象，则说明有必要把它们分成两个部分；
- 想在带有不同抽象接口的多个对象之间共享一个实现。

下面几节将通过开发可以运行20世纪80年代和90年代的各种掌上游戏机控制台的模拟器，阐述如何使用桥接模式来解决某些相关的设计问题。

9.3 创建 iOS 版虚拟仿真器

在Game Boy①和Game Gear②非常流行的年代，常常能见到很多孩子手中握着一台游戏机打着

① Game Boy是日本任天堂公司在1989年发售的一种电池驱动携带式电子游戏机。它是截至2004年销量最高的游戏机种。——译者注

② Game Gear是SEGA（世嘉）在1990年推出的第一部彩色掌上型游戏机。——译者注

喜欢的游戏。这些孩子已经长大，Game Boy和Game Gear成了他们儿时的回忆。很多人开发了各种仿真器，可以在完全不同的硬件体系结构的台式机和iOS设备上运行原来的Game Boy和Game Gear游戏，以唤起久违的童年记忆。

我们将讨论如何应用桥接模式构建仿真器，让它支持Game Boy和Game Gear等（可能也包括其他）多种便携游戏平台，但是不会涉及具体细节。

每个平台有两个主要组件——操作系统和接受用户输入的操作面板。Game Boy和Game Gear两者都是在操作面板的左手边有上下左右按键，在右手边有按键A和B作为动作按键，在中间有开始键。但是，Game Boy在开始键右边有个选择键，而Game Gear却没有这个键。尽管在按键布局和按键数上有少许差异，但两个平台操作面板的结构和功能几乎相同。

设计仿真器时，除了实际仿真器（运行该平台的游戏的操作系统），也要考虑作为用户输入设备的虚拟控制器。这个虚拟控制器很可能会用iOS的某些UI元素进行模拟。如果为每个具体仿真器创建专用的控制器，那么会有很多冗余，并且可能导致虚拟控制器子类的激增。而且，不同类型的虚拟输入方式可能需要不同的子类。比如，虚拟控制器可以不用方向按键，而是从加速度计读取各个方向的加速度变化来模拟上下左右指令。问题是需要把虚拟控制器和仿真器分离，这样它们可以独立地变化，而不影响对方的代码。换句话说，一组仿真器会有它们自己的类层次，同时一组虚拟控制器也会单独有自己的类层次。两个层次结构有不同的接口，但是通过对象组合关系，在两个层次结构的上层抽象类之间形成“桥接”，从而联系起来。这种设计的静态结构如图9-2所示。

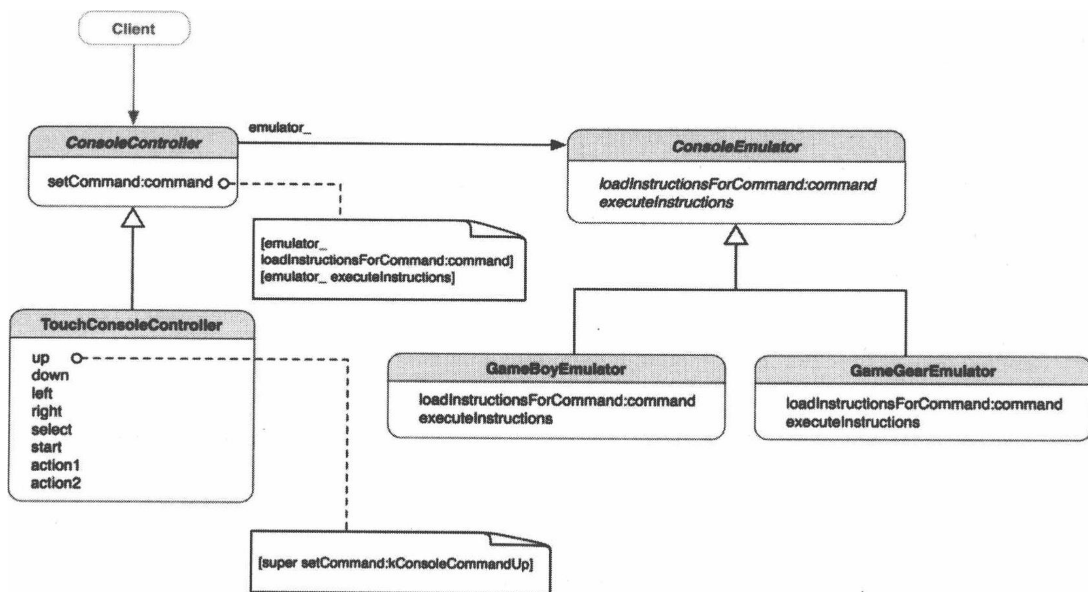


图9-2 iOS版虚拟仿真器的类图

`ConsoleController`和`ConsoleEmulator`分别是虚拟控制器和仿真器的抽象类。两个类有

不同的接口。在ConsoleController中封装一个对ConsoleEmulator的引用，是联系两者的唯一方式。因此ConsoleController的实例可以在一个抽象层次上使用ConsoleEmulator的实例。这就形成了两个不同的类ConsoleController与ConsoleEmulator之间的桥接。ConsoleEmulator为其子类定义了接口，用于处理针对特定控制台OS的低层指令。ConsoleController有个相对低层的方法，向桥接的另一端发送基本命令类型。ConsoleController的setCommand:command方法接受一个预先定义好的命令类型的参数，并通过loadInstructionsForCommand:command消息把它传递给内嵌的ConsoleEmulator引用。最后，它向这个引用发送一个executeInstructions消息，在仿真器中执行任何已加载的指令。两个不同类层次结构的连接的直观表示如图9-3所示。

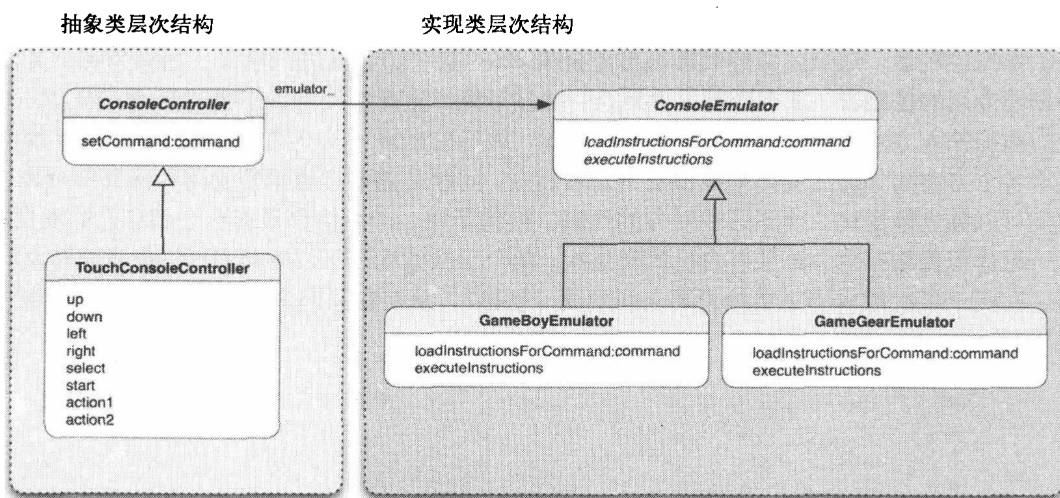


图9-3　实现类层次结构的变化不会影响抽象类的层次结构，反之亦然

ConsoleController类层次结构代表对ConsoleEmulator类层次结构的任何“实现”的一种“抽象”。抽象类层次结构提供了一层抽象，形成一个对任何兼容ConsoleEmulator的虚拟控制器层。具体的ConsoleController只能通过在父类中定义的低层setCommand:方法，与桥接的另一端的仿真器进行交流。在这种配置中这个方法不应被子类重载，因为这是一个让父类与细化的控制器之间进行通讯的接口。如果在仿真器一侧发生变更，对左侧的任何控制器都将毫无影响，反之亦然。

听起来很有趣，是吧？看看代码吧。首先，需要定义任何仿真器都应支持的若干命令，如代码清单9-1所示。

代码清单9-1　ConsoleCommands.h

```
typedef enum
{
  kConsoleCommandUp,
  kConsoleCommandDown,
```

```
  kConsoleCommandLeft,
  kConsoleCommandRight,
  kConsoleCommandSelect,
  kConsoleCommandStart,
  kConsoleCommandAction1,
  kConsoleCommandAction2
} ConsoleCommand;
```

上、下、左、右、选择、开始、动作1和动作2作为通用命令，定义成一组enum。要是将来想扩展命令列表，以支持更复杂的模拟器，都不会破坏任何一边的设计。请看代码清单9-2中抽象ConsoleEmulator的定义。

代码清单9-2 ConsoleEmulator.h

```
#import "ConsoleCommands.h"

@interface ConsoleEmulator : NSObject
{

}

- (void) loadInstructionsForCommand:(ConsoleCommand) command;
- (void) executeInstructions;

// 其他行为与属性

@end
```

抽象ConsoleEmulator有两个先前简要讨论过的基本方法loadInstructionsForCommand:和executeInstructions。

loadInstructionsForCommand:应该把任何具体的操作系统指令，根据在代码清单9-1中定义的命令类型加载到内部数据结构。executeInstructions会执行任何加载到这个数据结构中的指令。操作系统指令以及它的执行方式是平台相关的，因此具体仿真器应该重载这些方法，如代码清单9-3与代码清单9-4所示。

代码清单9-3 GameBoyEmulator.h

```
#import "ConsoleEmulator.h"

@interface GameBoyEmulator : ConsoleEmulator
{

}

// 从抽象类重载的行为
- (void) loadInstructionsForCommand:(ConsoleCommand) command;
- (void) executeInstructions;

// 其他行为与属性

@end
```

GameBoyEmulator和GameGearEmulator都是ConsoleEmulator的子类。它们重载抽象方

法，以提供自己平台的特定行为。

代码清单9-4 GameGearEmulator.h

```
#import "ConsoleEmulator.h"

@interface GameGearEmulator : ConsoleEmulator
{

}

// 从抽象类重载的行为
- (void) loadInstructionsForCommand:(ConsoleCommand) command;
- (void) executeInstructions;

// 其他行为与属性

@end
```

现在我们完成了仿真器层次结构的定义。在设计的虚拟控制器这一侧，ConsoleController是整个虚拟控制器类层次的起点。它以emulator_保持着ConsoleEmulator实例的一个内部引用。它也定义了一个setCommand:command方法，供其子类用预先定义的命令类型输入命令。马上会介绍其细节。它的类定义如代码清单9-5所示。

代码清单9-5 ConsoleController.h

```
#import "ConsoleEmulator.h"
#import "ConsoleCommands.h"

@interface ConsoleController : NSObject
{
  @private
  ConsoleEmulator *emulator_;
}

@property (nonatomic, retain) ConsoleEmulator *emulator;

- (void) setCommand:(ConsoleCommand) command;

// 其他行为与属性

@end
```

setCommand:只是简单地向emulator_引用发送loadInstructionsForCommand: command和executeInstructions消息，以完成指令执行过程，如代码清单9-6所示。

代码清单9-6 ConsoleController.m

```
#import "ConsoleController.h"

@implementation ConsoleController

@synthesize emulator=emulator_;
```

```
- (void) setCommand:(ConsoleCommand) command
{
  [emulator_ loadInstructionsForCommand:command];
  [emulator_ executeInstructions];
}

@end
```

到此，虚拟控制器与仿真器的基本桥接就完成了。现在要开始创建第一个虚拟控制器TouchConsoleController，以形成多点触摸屏与隐藏在视图之后具体仿真器之间的接口。它有几个基本方法的声明，反映了在代码清单9-1中预先定义的command类型。它的类声明如代码清单9-7所示。

代码清单9-7 TouchConsoleController.h

```
#import "ConsoleController.h"

@interface TouchConsoleController : ConsoleController
{

}

- (void) up;
- (void) down;
- (void) left;
- (void) right;
- (void) select;
- (void) start;
- (void) action1;
- (void) action2;

@end
```

TouchConsoleController的方法无需解释。每个方法只是用适当的ConsoleCommand值向super发送一个[super setCommand:ConsoleCommand]消息，如代码清单9-8所示。

代码清单9-8 TouchConsoleController.m

```
#import "TouchConsoleController.h"
#import "ConsoleEmulator.h"

@implementation TouchConsoleController

- (void) up
{
  [super setCommand:ConsoleCommandUp];
}

- (void) down
{
  [super setCommand:kConsoleCommandDown];
}
```

```
- (void) left
{
  [super setCommand:kConsoleCommandLeft];
}

- (void) right
{
  [super setCommand:kConsoleCommandRight];
}

- (void) select
{
  [super setCommand:kConsoleCommandSelect];
}

- (void) start
{
  [super setCommand:kConsoleCommandStart];
}

- (void) action1
{
  [super setCommand:kConsoleCommandAction1];
}

- (void) action2
{
  [super setCommand:kConsoleCommandAction2];
}

@end
```

我们想让所有方法使用在父类中定义的同一个setCommand:实现，通过向super而不是self发送消息，以避免混淆。这里self和super实际上是一回事，因为子类并没有重载setCommand:方法进行自己的桥接。不过我们仍然使用super以强调这一架构。然后在桥接的另一端，会根据传过来的ConsoleCommand值加载合适的操作系统指令，并在具体仿真器中予以执行。

通过桥接模式，可以看到对象组合的力量。我们为ConsoleEmulator实现的桥接不可能通过直接继承来实现。这也是优先使用对象组合而不是继承的一个原因。

9.4　总结

本章讨论了如何使用桥接模式来实现在iOS上运行的仿真器应用程序。没有深入研究实现真正仿真器的额外细节，只是重点探讨了桥接模式能帮助我们解决的几个设计问题。所以当陷入困境，苦于“如何从实现中把抽象分离出来而又要让它们联系在一起”之时，凭直觉就会想到桥接模式。那时可以再次浏览示例代码，看看怎样能把它用于自己的代码。

桥接模式是把一个接口适配到不同接口的一种方式。下一章，将介绍另一种设计模式，它不仅能够将不同的接口组合起来，而且可以把它们简化成单一入口，就像建筑物的外观（正门）一样。

第 10 章 外 观

比方说你今天不想开车，于是打电话叫了出租车。只要出租车能把你送到目的地，你不太在意车的牌子和型号。你会对出租车司机说的第一句话就是"送我去X"，X就是你想去的地方。然后出租车司机就开始"执行"一系列的"命令"（松刹车、换挡、踩油门等）。出租车司机抽象走了驾驶汽车的底层复杂操作的细节。他通过提供驾驶服务（简化了的接口），把你与原本复杂的车辆操作接口分离开来。出租车司机与你之间的接口只是一个简单的"送我去X"命令。并且这个命令也不依赖车辆具体的品牌或型号。

很多旧的面向对象应用程序中，可能有许多类分散于带有各种功能的系统之中。要把这些类用于某个功能，需要知道全部细节才能在一组算法中使用它们。如果从逻辑上将其中一些类组合成一个简化的接口，可以让这些类更易于使用。为子系统中一组不同接口提供统一接口的一种方式称为外观模式。

本章将讨论什么是外观模式以及如何用Objective-C来实现这一模式。

10.1 何为外观模式

外观模式为子系统中一组不同的接口提供统一的接口。外观定义了上层接口，通过降低复杂度和隐藏子系统间的通信及依存关系，让子系统更易于使用。

比方说子系统中有一组不同的类，其中一些彼此依赖，如图10-1所示。这让客户端难以使用子系统中的类，因为客户端需要知道每一个类。有时如果客户端只是需要其默认行为而不做定制，这会是不必要的麻烦。外观起到整个子系统的入口的作用。有些客户端只需要子系统的某些基本行为，而对子系统的类不做太多定制，外观为这样的客户端提供简化的接口。外观就像前面例子里的出租车司机。只有需要从某些子系统的类定制更多行为的客户端，才会关注外观背后的细节。出租车的场景可以提供带有路上的"默认"行为的驾驶服务。但是如果你想要"定制的"出行（比如，做几次途中停留或者在公路上高速行驶等），那么还是自己开车比较好。

> **外观模式：**为系统中的一组接口提供一个统一的接口。外观定义一个高层接口，让子系统更易于使用。*
>
> *最初的定义出现于《设计模式》（Addison-Wesley，1994）。

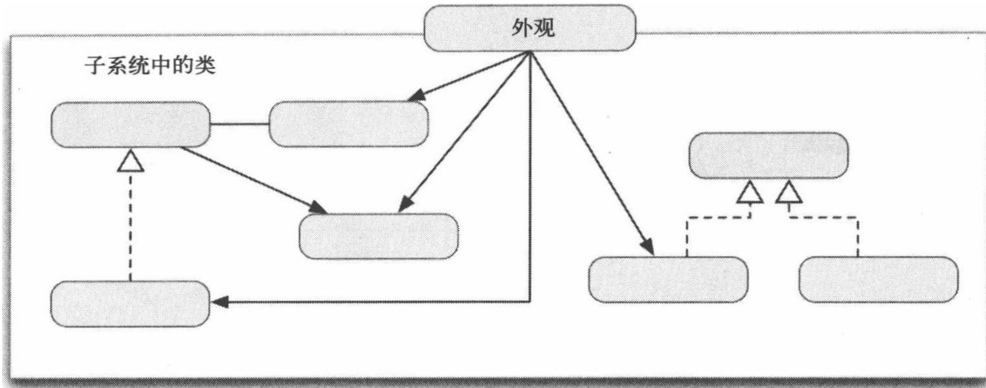


图10-1　一个外观实现的示例结构图

10.2　何时使用外观模式

在以下两种常见的情形下，会考虑使用这一模式。

- 子系统正逐渐变得复杂。应用模式的过程中演化出许多类。可以使用外观为这些子系统类提供一个较简单的接口。
- 可以使用外观对子系统进行分层。每个子系统级别有一个外观作为入口点。让它们通过其外观进行通信，可以简化它们的依赖关系。

在下面几节，将用出租车司机的例子实现外观模式，以阐述其基本概念。而且，随后将使用这一模式简化TouchPainter应用程序的涂鸦保存过程。

10.3　为子系统的一组接口提供简化的接口

回到本章引言部分出租车的例子。有一位乘客和一辆出租车，出租车为驾驶出租车的一组复杂接口提供了一个简化了的接口。如果放到类图中，它们会是图10-2这个样子。

可以看到整个出租车服务作为一个封闭系统，包括一名出租车司机、一辆车和一台计价器。同系统交互的唯一途径是通过CabDriver中定义的接口driveToLocation:x。一旦乘客向出租车司机发出driveToLocation:x消息，CabDriver就会收到这个消息。司机需要操作两个子系统——Taximeter和Car。CabDriver先会启动（start）Taximeter，让它开始计价，然后司机对汽车会松刹车（releaseBrakes）、换挡（changeGears）、踩油门（pressAccelerator），把车开走。直到到达了地点x，CabDriver会松油门（releaseAccelerator）、踩刹车（pressBrakes）、停止（stop）Taximeter，结束行程。一切都发生于发给CabDriver的一个简单的driveToLocation:x命令之中。无论这两个子系统有多么复杂，它们隐藏于乘客的视线之外。因此CabDriver是在为出租车子系统中的其他复杂接口提供一个简化的接口。CabDriver像“外观”一样，处于乘客与出租车子系统之间。请看代码清单10-1，看看在代码中它们是什么样子。

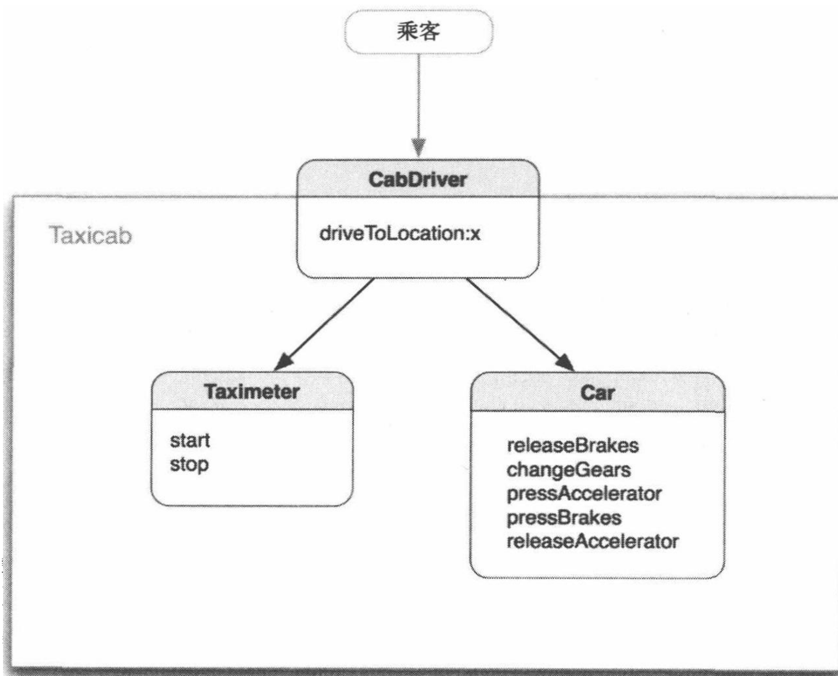


图10-2 出租车例子的类图。CabDriver类是Taxicab（出租车）子系统的外观

代码清单10-1 Car.h

```
@interface Car : NSObject
{

}

// ……

- (void) releaseBrakes;
- (void) changeGears;
- (void) pressAccelerator;
- (void) pressBrakes;
- (void) releaseAccelerator;

// ……

@end
```

代码清单10-1中，Car定义了几个操作内部对象用的方法，如releaseBrakes（松刹车）、changeGears（换挡）、pressAccelerator（踩油门）、pressBrakes（踩刹车）和releaseAccelerator（松油门）。客户端要想使用Car的内部对象，必须了解如何使用这些方法进行正确操作。除了Car，也来看看代码清单10-2中Taximeter的代码。

代码清单10-2　Taximeter.h

```
@interface Taximeter : NSObject
{

}

- (void) start;
- (void) stop;

@end
```

虽然Taximeter本身是个复杂系统，但它有两个让客户端操作其对象的方法。start和stop方法只是让Taximeter开始或停止。我们不会深入Taximeter的细节。目前，出租车服务系统里面有两个复杂的子系统。需要一个CabDriver（出租车司机）作为“外观”以简化接口。其代码段见代码清单10-3。

代码清单10-3　CabDriver.h

```
#import "Car.h"
#import "Taximeter.h"

@interface CabDriver : NSObject
{

}

- (void) driveToLocation:(CGPoint) x;

@end
```

CabDriver的外观方法决定了客户端可以用多简单的方式使用整个出租车服务系统。如前面提到的那样，客户只需要调用driverToLocation:x方法（这里的x是客户的目的地），然后其余的操作就会在消息调用中发生。客户端不需要了解底层所发生的一切。请在代码清单10-4中看看这个方法实际在做什么。

代码清单10-4　CabDriver.m

```
#import "CabDriver.h"

@implementation CabDriver

- (void) driveToLocation:(CGPoint) x
{
  // ……

  // 启动计价器
  Taximeter *meter = [[Taximeter alloc] init];
   [meter start];

  // 操作车辆，直到抵达位置x
```

```
    Car *car = [[Car alloc] init];
    [car releaseBrakes];
    [car changeGears];
    [car pressAccelerator];

    // ……

    // 当到达了位置x，就停下车和计价器
    [car releaseAccelerator];
    [car pressBrakes];
    [meter stop];

    // ……
}

@end
```

在driveToLocation:方法中，首先启动一个Taximeter（计价器）对象，让它从那一刻开始计价。然后转到Car（汽车）对象，开始对它进行操作。向Car发送releaseBrakes消息以松开刹车，接着changeGears（换挡），pressAccelerator（踩油门）把车开走。当到达了目的地x，命令Car（汽车）releaseAccelerator（松油门），pressBrakes（刹车），最后让Taximeter（计价器）对象停止计价。这样服务就结束了。

看起来不复杂，是吧？使用CabDriver作为外观，可以简化整个服务系统。准备好去看看如何把外观放在真正的应用程序TouchPainter之中了吗？请看下一节。

10.4 在 TouchPainter 应用程序中使用外观模式

第2章介绍了TouchPainter应用程序，在它原来的设计中，有一个需求是把画在CanvasView中的图形保存为一种Scribble形式，并将这个数据结构放进文件系统。涂鸦图的文档可以在以后进行读取并重建为真正的Scribble对象，恢复其被保存时的状态。这个操作涉及不同对象，它们组合在一起完成这一过程。根据原有的设计，保存过程的一部分涉及一种数据结构，它包含屏幕上线条的当前状态和一个屏幕截图（屏幕截图以后会用作缩略图以供浏览）。过程的另一部分是使用NSFileManager的实例，进行把数据结构保存到文件系统的实际操作。Scribble对象把这个数据结构创建为备忘录（见备忘录模式，第23章）。备忘录对象应该独立于对应的缩略图图像来保存。整个过程涉及许多步骤、操作和各种对象。若不简化接口，将很容易失控，尤其是当以后需要在应用程序的其他地方对同样的操作进行复用的时候。因此需要一个ScribbleManager通过几个简化的接口来处理底层的所有复杂操作。图10-3是表示它们关系的类图。

在ScribbleManager中有好几个共有接口，但我们只讨论saveScribble:scribble thumbnail:image方法。你可以从本书的网站下载源代码，看看ScribbleManager的实际实现。ScribbleManager的总体思想是对保存和恢复Scribble对象相关的所有操作予以简化。Scribble是模型（从模型-视图-控制器的角度来说），它包含一个内部数据结构，用于保存用户画在CanvasView上的所有当前线条。CanvasViewController的实例中有一个Scribble的实例（见观察者模式，第12章），作为模型-视图-控制器结构的一部分。

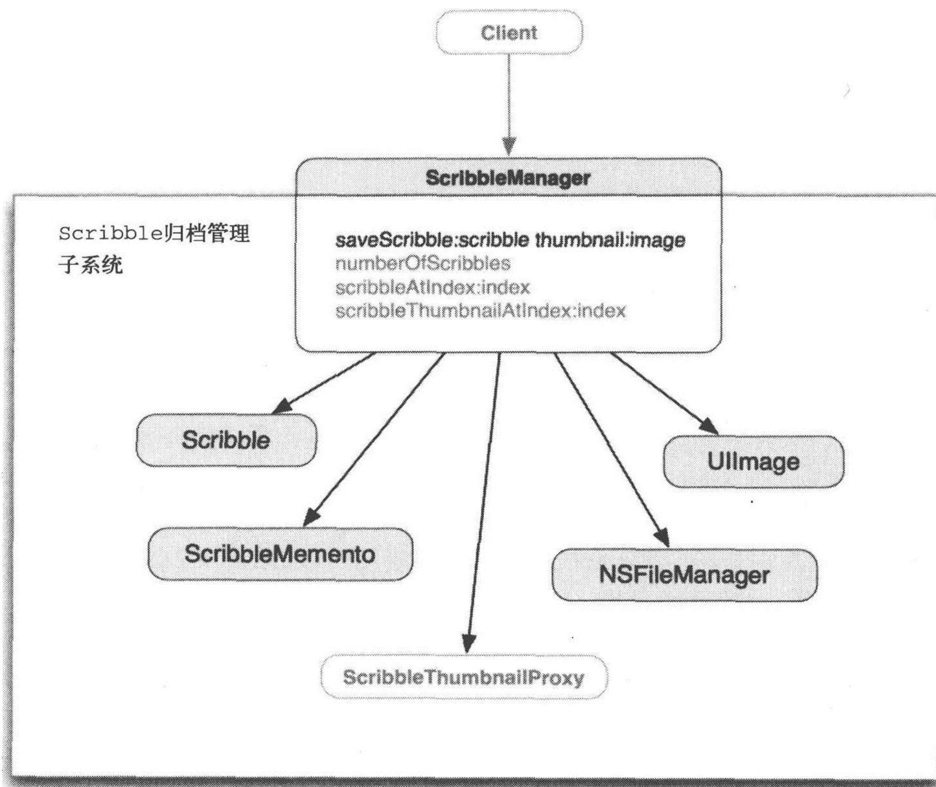


图10-3　ScribbleManager与子系统中其他类的类图

当发起了保存当前Scribble（CanvasView上面的东西）的请求时，当前Scribble对象的引用和涂鸦的屏幕截图，作为消息的一部分被传给ScribbleManager对象的saveScribble: scribble thumbnail:image方法。ScribbleManager的代码段如代码清单10-5和代码清单10-6所示。

代码清单10-5　ScribbleManager.h

```
#import <Foundation/Foundation.h>
#import "Scribble.h"
#import "ScribbleThumbnail.h"
#import "ScribbleThumbnailProxy.h"

@interface ScribbleManager : NSObject
{

}

- (void) saveScribble:(Scribble*)scribble thumbnail:(UIImage*)image;
- (NSInteger) numberOfScribbles;
- (Scribble*) scribbleAtIndex:(NSInteger)index;
```

```
- (ScribbleThumbnail*) scribbleThumbnailAtIndex:(NSInteger)index;

@end
```

代码清单10-6 ScribbleManager.m中saveScribble:方法的定义

```
- (void) saveScribble:(Scribble*)scribble thumbnail:(UIImage*)image
{
  // 为新的涂鸦数据及其缩略图取得新的索引值
  NSInteger newIndex = [self numberOfScribbles] + 1;

  // 把索引值用作名字的一部分
  NSString *scribbleDataName = [NSString stringWithFormat:@"data_%d", newIndex];
  NSString *scribbleThumbnailName = [NSString stringWithFormat:@"thumbnail_%d.png",
                                                                         newIndex];

  // 从涂鸦获得备忘录，然后保存到文件系统
  ScribbleMemento *scribbleMemento = [scribble scribbleMemento];
  NSData *mementoData = [scribbleMemento data];
  NSString*mementoPath = [[self scribbleDataPath]
                             stringByAppendingPathComponent:scribbleDataName];
  [mementoData writeToFile:mementoPath atomically:YES];

  // 把缩略图直接保存到文件系统
  NSData *imageData = [NSData dataWithData:UIImagePNGRepresentation(image)];
  NSString*imagePath=[[self scribbleThumbnailPath]
                             stringByAppendingPathComponent:scribbleThumbnailName];
  [imageData writeToFile:imagePath atomically:YES];
}
```

saveScribble:scribble thumbnail:image方法掌管着向文件系统保存涂鸦图与屏幕截图的图像的操作。它首先生成文件系统中涂鸦图与屏幕截图图像的路径。然后它向scribble发送一个scribbleMemento消息，得到用以保存数据的ScribbleMemento实例（参见备忘录模式，第23章）。ScribbleMemento的引用包含scribble的当前状态。只有Scribble对象才能够读取ScribbleMemento对象里面存储的数据。然后请求ScribbleMemento对象根据其内部结构生成一个NSData的实例。在为新返回的数据构建好完整的路径之后，ScribbleManager会向ScribbleMemento的对象发送writeToFile:mementoPath消息，把数据对象保存到文件系统。

10.5 总结

当程序逐渐变大变复杂时，会有越来越多小型的类从设计和应用模式中演化出来。如果没有一种简化的方式来使用这些类，客户端代码最终将变得越来越大、越来越难以理解，而且，维护起来会繁琐无趣。外观有助于提供一种更为简洁的方式来使用子系统中的这些类。处理这些子系统类的默认行为的，可能只是定义在外观中的一个简单的方法，而不必直接去使用这些类。

本书的这一部分介绍了几个设计模式，它们主要针对用更简单的接口或用不同的接口去适配各种接口。下一个部分中的设计模式用于分离协同工作的多个对象，这些对象具有共同或不同接口。

Part 4 第四部分

对象去耦

本部分内容

- 第 11 章　中介者
- 第 12 章　观察者

第11章

中　介　者

要是飞行员都争抢着起飞或降落而机场没有交通管制，会出什么事？要是没有集中的空中交通管制，天上或地面的飞行员就需要知道彼此的意图与位置，这既危险又复杂。

集中交通管制保证每架飞机在雷达监视之下，随时在屏幕上更新其飞行信息。因此空中交通管制员能够准确了解什么飞机现在位于管制的空域之中。每个飞行员需要向控制中心请求许可才能进行飞行、着陆与起飞。虽然飞行员仍然需要对交通中的其他飞机是什么有相当清楚的了解，但是这已不像完全没有集中空中管制时那么至关重要。空中交通管制中心起着重要的协调作用，确保不会因秩序混乱或沟通不良导致飞机在半空相撞。

在面向对象软件中，我们在设计中见过许多这样的场景。典型的例子是应用程序中的UI元素。比如有个对话框带有静态文本（label）、列表框（list box）、文本框（text field），以及几个别的输入框。当列表框中的一项被选中时，静态文本会被更新为从列表框选定的值。或者当用户在文本框输入了新的值时，需要把这个新的值加到列表框的列表中。当更多的UI元素参与到这一错综复杂的关系之中时，情况可能变得难以控制。元素之间需要彼此了解并相互操作。最终，会发展到难以复用与维护的地步。需要有个交通管制员来管理所有的UI交通[①]。组织各种UI元素在同一个语境下进行交互的集中化的角色，称为中介者（mediator）。从这一概念精心设计而来的设计模式叫做中介者模式。

在本章，将讨论中介者模式是怎么回事，并使用TouchPainter示例来说明如何使用这一模式管理视图迁移。

11.1 何为中介者模式

面向对象的设计鼓励把行为分散到不同对象中。这种分散可能导致对象之间的相互关联。在最糟糕的情况下，所有对象都彼此了解并相互操作。

虽然把行为分散到不同对象增强了可复用性，但是增加的相互关联又减少了获得的益处。增加的关联使得对象很难或不能在不依赖其他对象的情况下工作。应用程序的整体行为可能难以进行任何重大修改，因为行为分布于许多对象。于是结果可能是创建越来越多的子类，以支持应用

① 指信息交流。——译者注

程序的任何新行为。

中介者模式用于定义一个集中的场所，对象间的交互可以在一个中介者对象中处理。其他对象不必彼此交互，因此减少了它们之间的依存关系。

> **中介者模式**：用一个对象来封装一系列对象的交互方式。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。*
>
> * 最初的定义出现于《设计模式》(Addison-Wesley，1994)。

图11-1是参与这一模式的假想类的类图，表明了它们之间的关系。

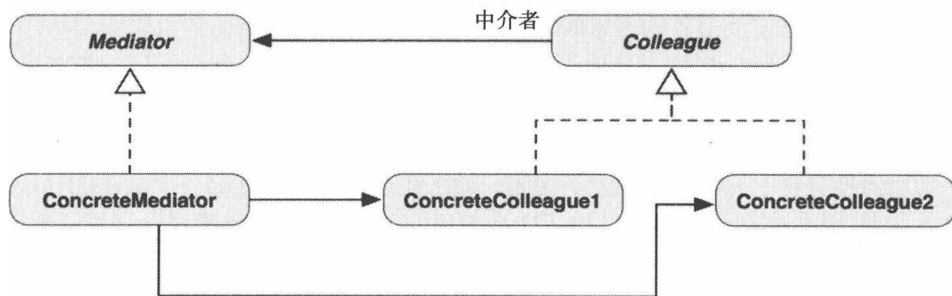


图11-1 中介者模式的类图

抽象的Mediator定义了用于同Colleague交互的一般行为。典型的同事（colleague）是以明确定义的方式进行相互通信的对象，并且彼此紧密依存。ConcreteMediator为Concrete Colleague定义了更加具体的行为，因此可以子类化Mediator，把各种Colleague交互算法应用到相同或不同的Colleague类型。如果应用程序只需要一个中介者，有时抽象的Mediator可以省略。

Colleague的实例有一个Mediator实例的引用，同时Mediator的实例知道参与这个组织的每个对象。运行时中介者模式的一种可能的对象结构如图11-2所示。

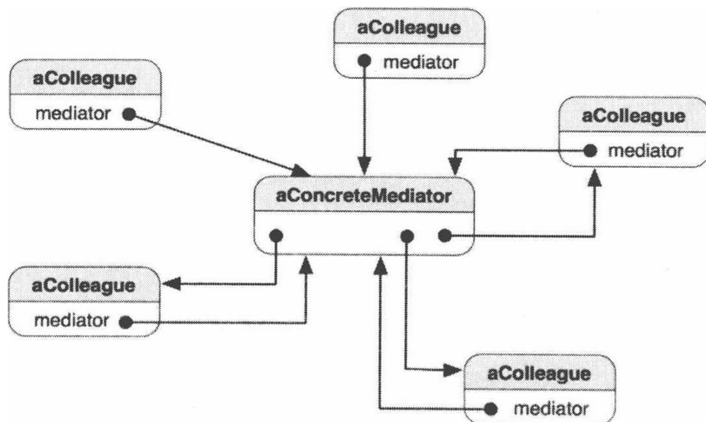


图11-2 对象图显示了运行时中介者模式的对象结构

11.2　何时使用中介者模式

在以下情形，自然会考虑使用这一模式：

- 对象间的交互虽定义明确然而非常复杂，导致一组对象彼此相互依赖而且难以理解；
- 因为对象引用了许多其他对象并与其通讯，导致对象难以复用；
- 想要定制一个分布在多个类中的逻辑或行为，又不想生成太多子类。

11.3　管理 TouchPainter 应用程序中的视图迁移

中介者模式不只适用于把各种对象间错综复杂的关系集中化，也适合组织两个不同视图间视图迁移。通过把一个视图加到另一个视图之上来管理视图迁移的iOS应用程序相当常见。这样第一个视图需要知道第二个视图并保持对它的引用，然后是第三个，依次类推。

有多种方式能进行视图迁移，我们将讨论其中三种。最常见的一种方式是从另一个视图控制器把视图添加到当前视图控制器，作为子视图。如果在添加之前不把前一个删除的话，整个栈上的子视图将难于管理。最终整个栈会堆满了许多不用的子视图。通常我不建议这种视图迁移方式，尤其是应用程序中有许多视图的时候。这一方式如图11-3所示。

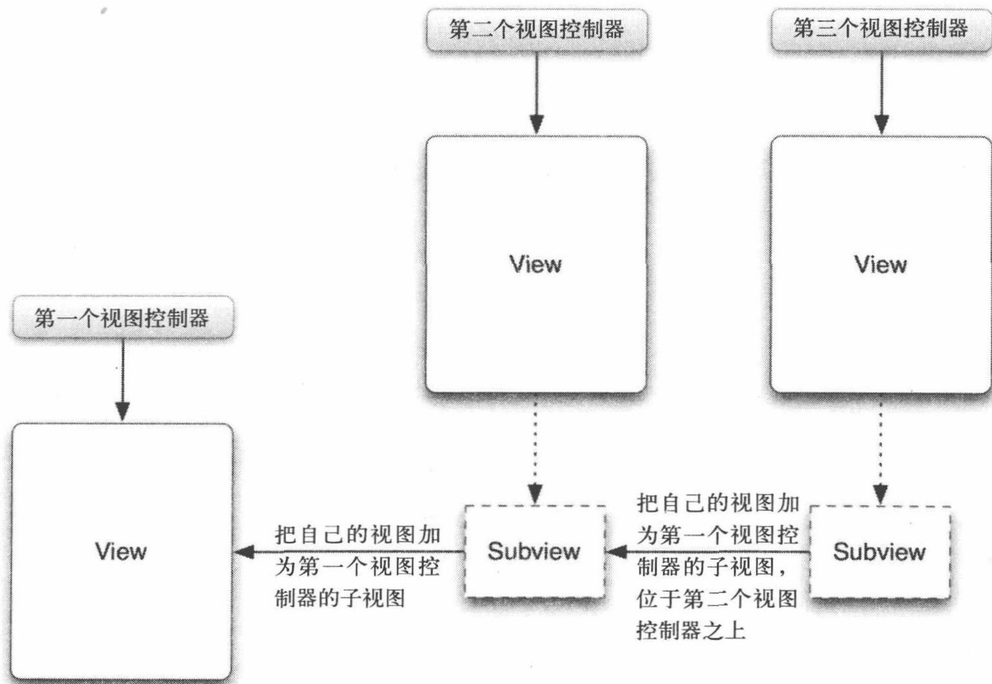


图11-3　通过向第一个视图控制器的视图添加不同视图控制器的子视图，管理应用程序中的不同视图

另一种视图迁移是通过使用带有视图顶部导航条的UINavigationController的实例。这个方法比前一种更为漂亮。UINavigationController的实例被用一个根控制器初始化，根控制器把自己的视图提供给UINavigationController作为第一个视图。当需要迁移到另一个视图的时候，视图的所有者把它的控制器压入UINavigationController。然后UINavigationController会处理视图迁移的全部细节，包括新视图进入屏幕的迁移效果与方向。UINavigationController管理内部的视图控制器栈。当前位于栈顶的视图控制器会显示在屏幕上。当视图“返回”到前一个视图时，将从栈上“弹出”其控制器，然后下面的一个将被显示。这一思想如图11-4所示。

这一视图迁移方法也许仅适用于某些应用程序而不适用于另一些，因为它显得相当“标准”，而且框架只提供了非常有限的视图迁移效果。并且有时应用程序需要更多的屏幕区域，没有空间用于显示导航条。

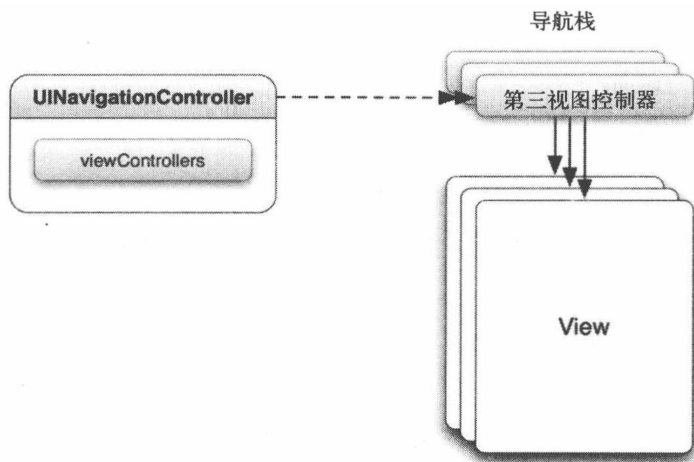


图11-4 通过将不同视图控制器压入UINavigationController的视图控制器栈来管理不同视图

最后一种，但也非常重要的方法是一次一个地显示模式-视图-控制器。视图迁移通过直接使用视图控制器来管理，而不涉及其他视图控制器的个别视图。在我看来，如果不需要花哨的效果又不想让导航条占去屏幕的空间，推荐使用这种方法做视图迁移。

其流程非常简洁，一次显示一个视图控制器然后从一个视图控制器迁移到另一个。这一方法的思想如图11-5所示。

现在已经讨论了Cocoa Touch框架中的几个视图迁移方法，从不太容易管理的，到流程简洁的。无论你喜欢哪种，或多或少视图控制器之间都需要相互了解、关联与交互，以完成视图迁移过程。如果将来需要修改视图迁移方法，或者只是简单地复用视图控制器，代码修改的工作量会非常大，难以维护并且容易出错。

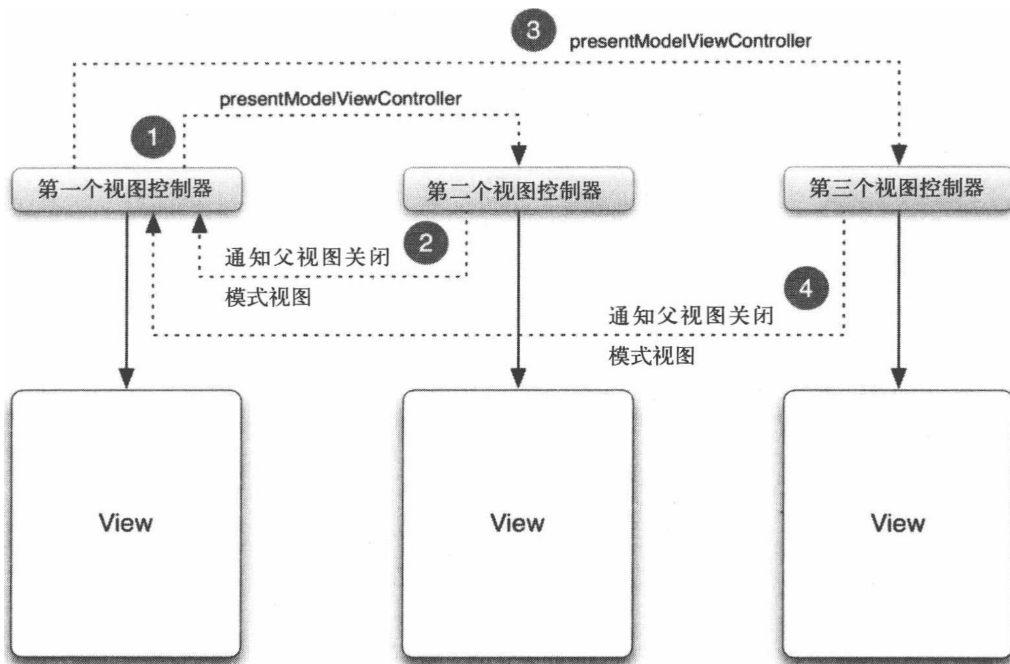


图11-5　通过让第一个视图控制器使用另一个视图控制器来显示模式视图，实现视图迁移

11.3.1　修改迁移逻辑的困难

很多时候我们需要向已有的UI流程添加新的视图，或者修改视图或视图控制器。在前面的例子中，其他视图控制器与其前面的视图控制器紧紧耦合在一起。我们来问一个视图管理的基本问题：如果需要向第一个视图的前面扩展视图流程，怎么办？因为所有其他视图控制器需要知道谁是第一个视图控制器，它们全部都需要修改代码，以反映这个支配流程的新视图控制器。如果应用程序只有几个视图控制器，情况并不太糟糕。如果应用程序有10个、20个或更多视图，全部进行修改将非常可怕。另一个基本问题：如果要改变主流程的迁移机制，怎么办？视图变换的风格取决于流程中如何管理视图。如果`UINavigationController`管理流程，那么流程将遵守`UINavigationController`设定的规则，带有导航条和有限的视图迁移效果。如果视图流程是通过把其他视图添加为第一个视图的子视图的方式来实现的，那么跟`UINavigationController`相比，会多几种来自`UIView`的迁移效果。

无论走哪条路，无论以后是否要修改，很可能每次流程变更都需要修改所有视图控制器。参与迁移的全部视图紧紧耦合在一起。

TouchPainter应用程序中，使用传统方法在视图控制器之间进行视图变换的假想场景如图11-6所示。

每个视图控制器有自己的按钮，当用户单击时触发流程的改变。按钮的控制器会立刻处理任

何视图变换。例如，UIBarButtonItem的实例向CanvasViewController的实例发送一个事件，事件带有一个标签，表明它是要激活PaletteViewController的视图，当收到事件时，CanvasViewController会把PaletteViewController的实例放入视图迁移的场景中。迁移可以是显示模式视图或者通过UINavigationController的实例来进行协调。当用户完成了PaletteViewController的视图的使用时，用户会单击Done按钮（UIBarButtonItem）以“返回”。在显示模式视图的情况下，PaletteViewController的实例要关闭自己，就需要取得CanvasViewController实例的引用（CanvasViewController是模式表示的父视图控制器）。如果用户要打开ThumbnailViewController的视图，情况也几乎相同。

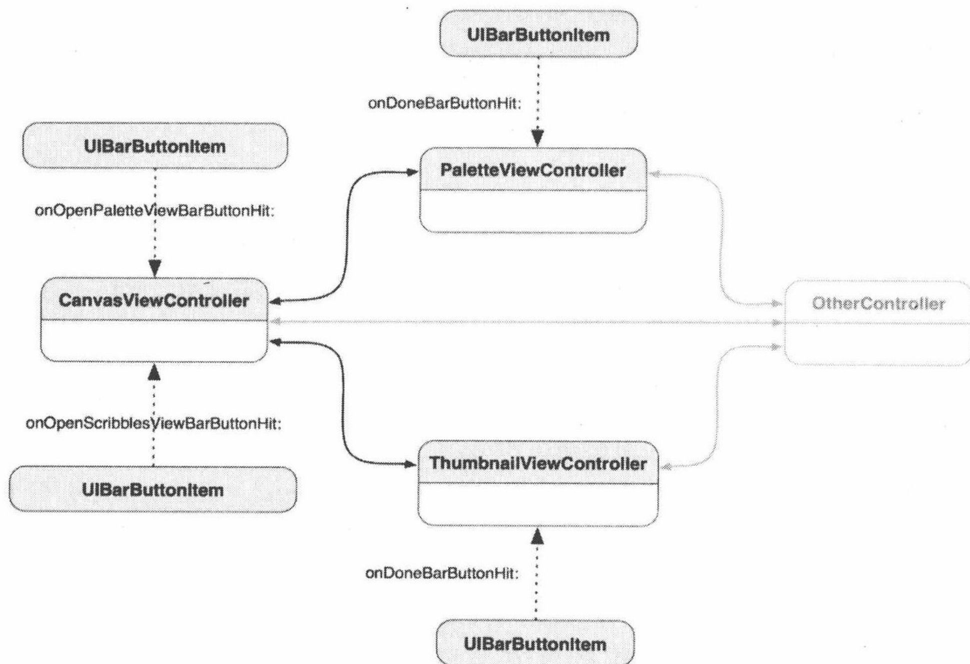


图11-6　不同视图控制器之间一种可能的视图变换关系

想象一下我们要向流程中添加更多的视图控制器。这会让事情更复杂，更难于维护。比较好的办法是把流程逻辑放到一个集中的对象之中。完整的UI流程可以在一个地方被监控而且更易于理解。

11.3.2 集中管理 UI 交通

如果有某种集中的角色来组织全部UI交通，只要代码可复用并可维护，日子就会好过得多。这一角色应该像一名以前的交通警察，在十字路口协调来自不同方向的交通。改变十字路口的交通模式只需要把新的策略给交通警察，而不是路上的车。交通警察的方式提供了管理不同视图的一种松耦合方案。

我们需要某种应用程序协调员作为中介者来协调UI流程。其角色只是协调UI交通，仅此而已。中介者应该知道参与流程的所有视图控制器。它像电影中的导演。导演应该知道所有参与影片的角色以及任何场景变换。电影如果有任何最后一刻的修改，导演应该是最先知道的人，并且应该为修改组织所有相关的资源。所以场景变换（视图迁移）由中介者决定。

我们将引入新的控制器，为视图迁移充当交通警察。我们把它称为CoordinatingController。CoordinatingController的实例维护UI要素与视图控制器之间UI流程的逻辑。没有用户交互的参与，不会发生视图迁移。用户会单击按钮，触发视图迁移的请求，然后CoordinatingController实现请求。CoordinatingController将根据按钮的标签处理视图迁移。一组有效的标签定义成枚举类型，可用于各种视图的激活，如代码清单11-1所示。

代码清单11-1　为用于识别工具条按钮的各种标签而定义的枚举类型

```
typedef enum
{
  kButtonTagDone,
  kButtonTagOpenPaletteView,
  kButtonTagOpenThumbnailView
} ButtonTag;
```

比如，当工具条按钮的标签设为kButtonTagOpenPaletteView（整数值1）的时候，它可以在CoordinatingController中激活调色板视图，依次类推。稍后将讨论其细节。

在TouchPainter例子中，CanvasViewController的视图是用户启动应用程序后的默认视图。控制器有个能打开调色板视图（PaletteViewController的视图）的工具条按钮。工具条按钮是UIBarButtonItem的实例，其标签设置为kButtonTagOpenPaletteView（整数值1）。当用户单击工具条按钮的时候，它直接发送一个动作消息requestViewChangeByObject:给CoordinatingController实例。然后它首先检查是什么对象发出的请求。如果对象是UIBarButtonItem的实例，那么它就检查其标签，看它能激活什么视图。然后CoordinatingController会激活适当的视图控制器并且进行向它的迁移。如果按钮的标签是kButtonTagOpenThumbnailView的话，也会同样地打开ThumbnailViewController的视图。至于PaletteViewController上面的Done按钮，它的标签设置为kButtonTagDone（整数值0）。当DoneBarButton被单击时，它会发送同样的消息到CoordinatingController。CoordinatingController会进行同样的标签检查并意识到它应该关闭当前视图，返回到主视图——CanvasViewController的视图。使用新的CoordinatingController后它们新的关系如图11-7所示。

现在，对于如何把东西连接起来，我们有了非常好的构思。下面，我们要予以实现。开始之前，首先要给它们画一个类图，以确认知道自己在做什么。为TouchPainter应用程序所作的中介者实现的类图，如图11-8所示。

这个类图与原汁原味的中介者模式稍有不同。首先，没有使用抽象中介者而是使用具体的CoordinatingController作为唯一的中介者，供其他参与的对象使用。其次，我们没有使用高层的抽象Colleague类型为其他具体Colleague执行任何操作。我们只需要UIBarButtonItem类型以定义我们的Colleague，因为视图上的工具条按钮是触发视图变换请求的关键元素。定制

的UIBarButtonItem类的实例，会保持一个对CoordinatingController实例的引用作为目标（target）。当单击事件发生时，它会以如下方式向CoordinatingController发送消息：

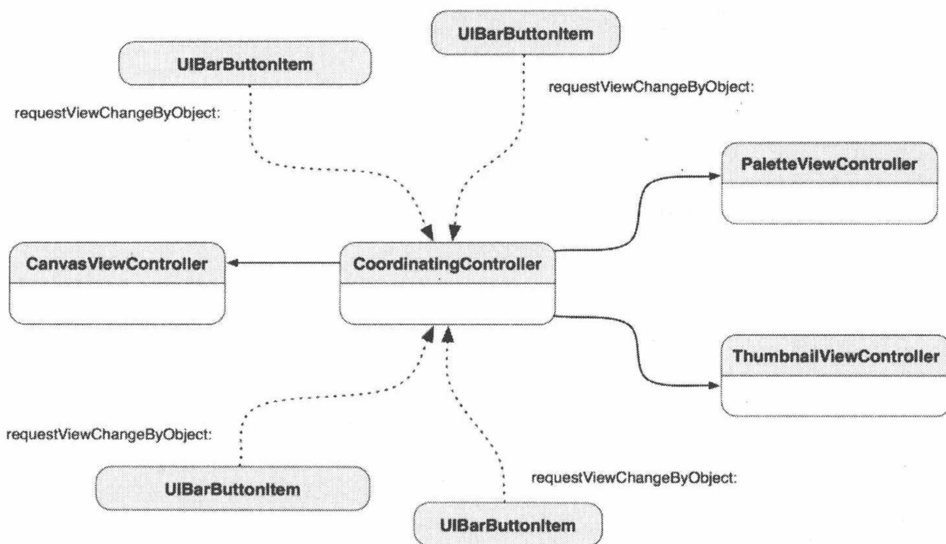


图11-7 CoordinatingController作为中介者，集中了视图迁移逻辑

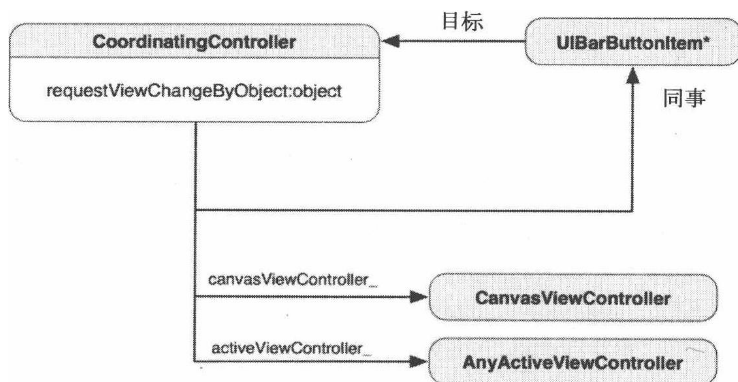


图11-8 CoordinatingController与其他接受协调的视图控制器的类图

```
[target requestViewChangeByObject:self]
```

按钮本身只需要知道这些，然后由CoordinatingController接管，在幕后施展视图变换的本领。

接下来我们来实现CoordinatingController，见代码清单11-2。

代码清单11-2 CoordinatingController.h

```
#import "CanvasViewController.h"
```

11

```
#import "PaletteViewController.h"
#import "ThumbnailViewController.h"


@interface CoordinatingController : NSObject
{
@private
  CanvasViewController *canvasViewController_;
  UIViewController *activeViewController_;
}

@property (nonatomic, readonly) UIViewController *activeViewController;
@property (nonatomic, readonly) CanvasViewController *canvasViewController;

+ (CoordinatingController *) sharedInstance;
- (IBAction) requestViewChangeByObject:(id)object;

@end
```

CoordinatingController只需要保持对主视图控制器CanvasViewController的引用，以便以后通过canvasViewController属性将它返回给其他发出请求的对象。被Coordinating-Controller使用的其他视图控制器，如果它在屏幕上被显示为激活状态，会通过activeView-Controller属性被返回。所有属性均为readonly（只读）型，因为不想在运行时被其他对象修改。CoordinatingController的关键是(IBAction) requestViewChange-ByObject:(id)object方法。你可能想问为什么把它的返回类型设为IBAction。这是因为这样就可以在Interface Builder中使用CoordinatingController来连接其他UI元素。我们会在后面几节讨论这些细节。

在应用程序的生命周期中只应该有一个CoordinatingController实例，因为每个视图控制器也只允许一个实例。如果允许有多个CoordinatingController的实例，事情会一团糟，前后矛盾，因此把它实现为单例（见单例模式，第7章）。CoordinatingController有一个sharedInstance类方法，返回它的单例。代码清单11-3显示了如何实现Coordinating-Controller，简洁起见省去了单例的部分。

代码清单11-3 CoordinatingController.m

```
#import "CoordinatingController.h"

@implementation CoordinatingController

@synthesize activeViewController=activeViewController_;
@synthesize canvasViewController=canvasViewController_;

// 简洁起见，省去了单例的实现部分

#pragma mark -
#pragma mark A method for view transitions

- (IBAction) requestViewChangeByObject:(id)object
{
  if ([object isKindOfClass:[UIBarButtonItem class]])
```

```
  {
    switch ([(UIBarButtonItem *)object tag])
    {
      case kButtonTagOpenPaletteView:
      {
        // 加载PaletteViewController
        PaletteViewController *controller = [[[PaletteViewController alloc]
                                               init] autorelease];

        // 迁移到PaletteViewController
        [canvasViewController_ presentModalViewController:controller
                                                 animated:YES];

        // 把activeViewController设置为PaletteViewController
        activeViewController_ = controller;
      }
        break;

      case kButtonTagOpenThumbnailView:
      {
        // 加载ThumbnailViewController
        ThumbnailViewController *controller = [[[ThumbnailViewController alloc]
                                                 init] autorelease];

        // 迁移到ThumbnailViewController
        [canvasViewController_ presentModalViewController:controller
                                                   animated:YES];

        // 把activeViewController设置为ThumbnailViewController
        activeViewController_ = controller;
      }
        break;

        default:
          // 对于其他类型的标签，只是回到主视图canvasViewController
        {
          // 除了CanvasViewController，每个视图控制器都共有一个Done按钮
          // 当单击Done按钮时
          // CoordinatingController应该按照设计把用户带回到第一个页面
          // 其他UIBarButtonItem对象也会走这里
          [canvasViewController_ dismissModalViewControllerAnimated:YES];

          // 把activeViewController设回canvasViewController
          activeViewController_ = canvasViewController_;
        }
          break;
    }
  }
  // 在其他情况，回到主视图canvasViewController
  else
  {
    [canvasViewController_ dismissModalViewControllerAnimated:YES];

    // 把activeViewController设置回canvasViewController
    activeViewController_ = canvasViewController_;
  }
}

@end
```

我们重点讨论requestViewChangeByObject:(id)object方法。方法的大部分是起实际作用的一个中型if-else语句块（至少在我看来是中型的）。在策略模式这一章（第19章）会讲到，用巨型的switch-case或if-else语句块来决定使用什么算法，可能意味着需要分解为各种策略。简洁起见，这里只使用一个if-else语句块来示范实现的这个部分。当然，如果在实际应用程序中if-else语句块变成巨无霸，就应该考虑采用策略模式。

> **说明：** 中介者模式以中介者内部的复杂性代替交互的复杂性。因为中介者封装与合并了colleague（同事）的各种协作逻辑，自身可能变得比它们任何一个都复杂。这会让中介者本身成为无所不知的庞然大物，并且难以维护。

方法中的if-else语句块首先检查对象是否为UIBarButtonItem的实例。如果是，switch-case语句块就会根据代码清单11-1中定义的ButtonTag枚举类型确定其标签。如果标签为kButtonTagOpenPaletteView，就会命令canvasViewController_用paletteViewController_实例显示一个模式视图。switch-case的其余部分以同样的逻辑，如前面几节讨论的那样显示与关闭视图控制器。

11.3.3　在 Interface Builder 中使用 CoordinatingController

我们可以在代码中手动实例化一组前面所讨论的UIBarButtonItem实例。但是怎么都不如使用Interface Builder来得方便，尤其是可以在一个地方管理所有UI元素。

接下来的几个图，以CanvasViewController和UIBarButtonItem为例，逐步展示如何在Interface Builder中使用UI元素。这个UIBarButtonItem会触发视图迁移，迁移到PaletteViewController的视图。

从项目打开CanvasViewController.xib文件之后，把Object（NSObject）的实例拖动到主文档窗口，如图11-9所示。

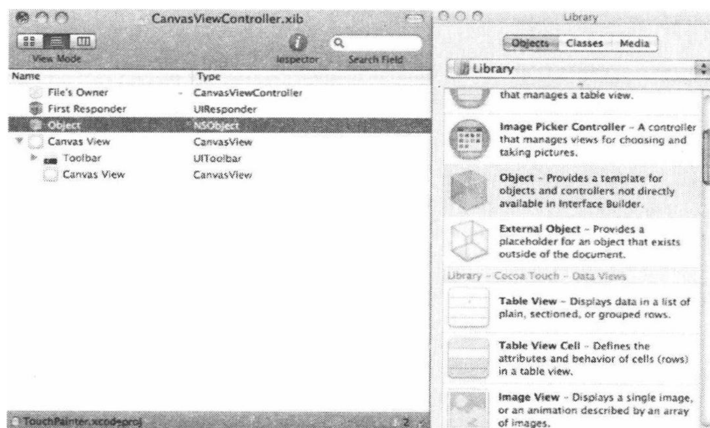


图11-9　把外部Object的引用拖动到主xib文档窗口

然后到Identity Inspector窗口（如果没有打开，就按⌘4，或从菜单选择Tools→Identity Inspector打开），再把Object的class identity（类身份）设为`CoordinatingController`，如图11-10所示。

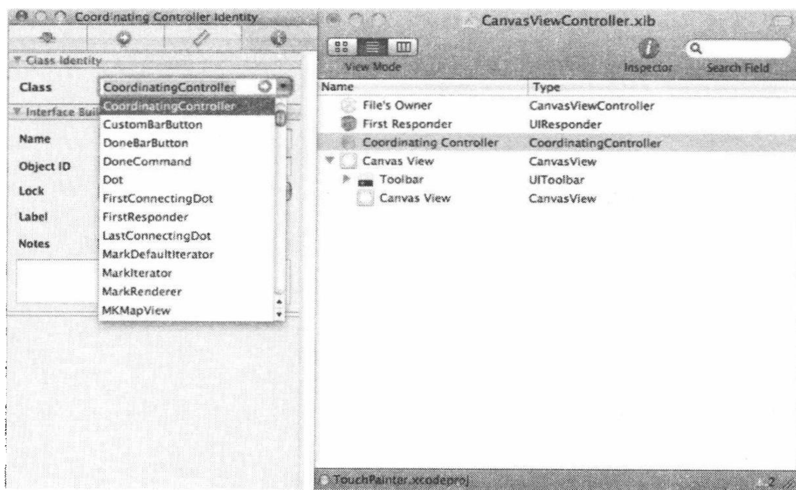


图11-10 把外部对象设置为`CoordinatingController`类

设好了`CoordinatingController`之后，就可以继续修改视图上的一个工具条按钮的某些属性。选择调色板按钮，把它的标签改为1（即`kButtonTagOpenPaletteView`），如图11-11所示。

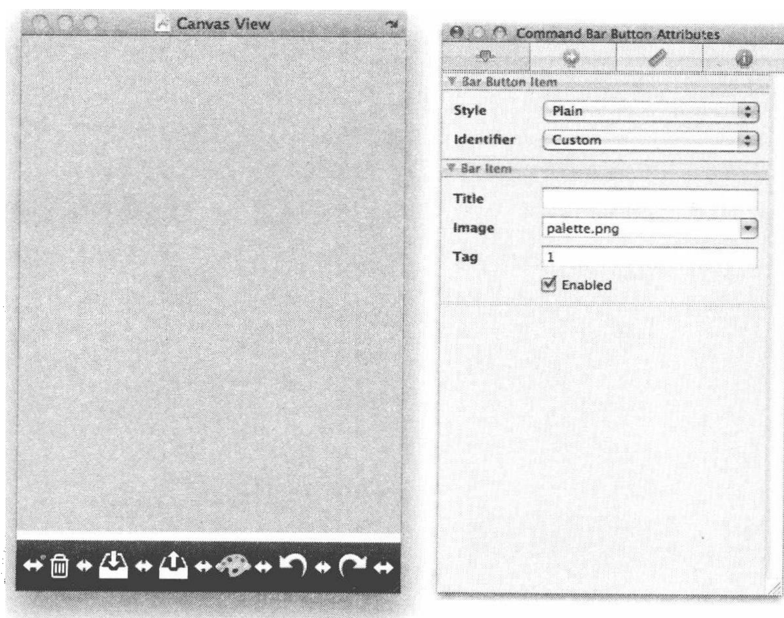


图11-11 把调色板按钮的标签设为1（`kButtonTagOpenPaletteView`）

现在，按住Control和Option键，从调色板按钮拖动到主文档窗口中CoordinatingController对象的引用。然后从小下拉菜单选择requestViewChangeByObject:方法的选项，如图11-12和图11-13所示。

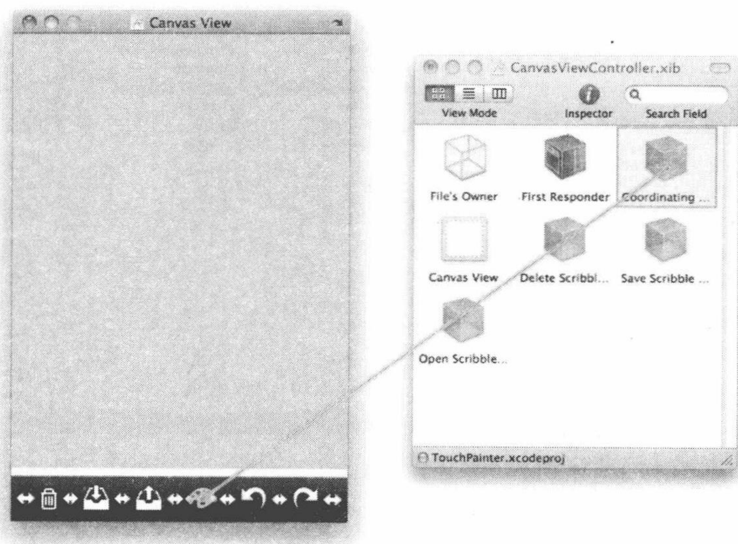


图11-12　按住Control+Option，同时拖动调色板按钮到CoordinatingController项目

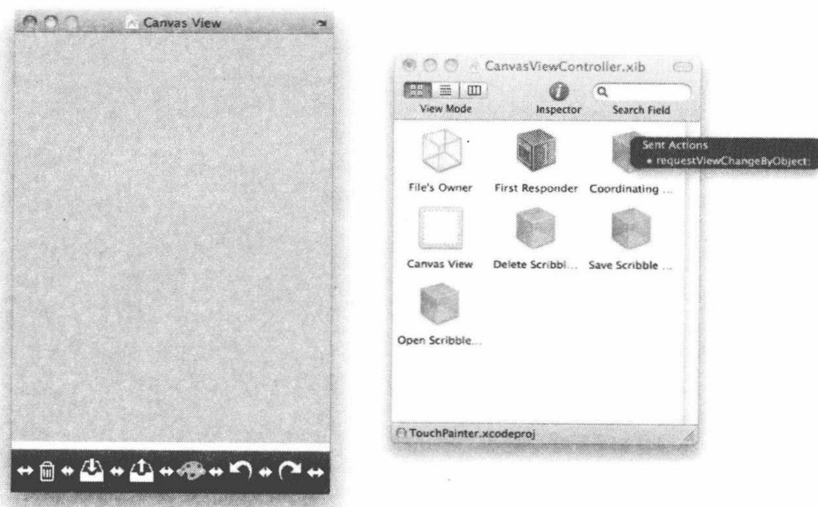


图11-13　选择requestViewChangeByObject:方法作为单击调色板工具条按钮时的动作选择器（action selector）

PaletteViewController、ThumbnailViewController及其工具条按钮可通过同样的过程连接起来——搞定了！

现在，与视图迁移逻辑有关的所有事情都到位了。甚至都不需要在任何视图控制器中写一行代码。整个逻辑由CoordinatingController的一个方法在管理。由于CoordinatingController被实现为单例（见第17章），实际上在应用程序的整个生命周期里通过.xib文件在运行时生成的实例，与通过CoordinatingController的单例alloc和init方法生成的实例，是同一个实例。

11.4 总结

本章探讨了与中介者模式有关的很多内容，也探讨了如何使用Cocoa Touch框架用Objective-C来实现这一模式。

虽然对于处理应用程序的行为分散于不同对象并且对象互相依存的情况，中介者模式非常有用，但是应当注意避免让中介者类过于庞大而难以维护。如果已经这样了，可以考虑使用另一种设计模式把它分解。要创造性地混用和组合各种设计模式解决同一个问题。每个设计模式就像一个乐高（Lego）积木块。整个应用程序可能要使用彼此配合的各种“积木块”来建造。

在下一章，将会讨论另一种设计模式，它使用一种“发布–订阅”机制来消除对象耦合。

第12章

观察者

12

在第11章的例子中已经讲过了，航空交通需要集中的空中交通管制。有很多操作员坐在控制塔里面盯着自己的雷达屏幕，以确保不会发生空中相撞。同时，那些开着价值数百万美元的“大鸟”的飞行员，需要知道在他们周围发生着什么，也就是空中的交通状况。飞行员可以把他们的无线电调到特定的频道，收听（观察）周围的交通状况。如果交通管制员向收听这个频道的飞行员广播了某些预警或警告，飞行员可以用某些行动表示已收到了消息。因此在这一模式中，任何人都知道他们在观察哪个交通管制，反过来却不然（至少空中交通管制员不知道所有的观察者，只知道雷达屏幕上的光点）。任何人（包括实际的飞行员）可以在任何时候收听广播或换台，而不会打扰其他人。

我们把这一思想引入面向对象软件设计中来，用以消除具有不同行为的对象之间的耦合（或者用其他不同的行为来扩展现有的行为）。通过这一模式，不同对象可以协同工作，同时它们也可以被复用于其他地方。我们把它称为观察者模式（Observer pattern）。

在本章，将讨论这一模式的概念以及它在Cocoa Touch框架下如何被改写。观察者模式也是MVC（模型-视图-控制器）模式的一部分。我们将使用第2章中的TouchPainter应用程序的例子，来讨论如何应用这一模式，当存储在CanvasViewController中的模型里面的线条数据发生变化时，让CanvasView作出反映。

12.1 何为观察者模式

观察者模式也叫做发布-订阅模式。如它的别名暗示的那样，它很像杂志的订阅。当从杂志发行商订阅杂志的时候，读者把名字和邮寄地址提供给发行商，这样新的一期就能送到读者手上。发行商保证把正确的杂志送到正确的地址。一般来说，读者不会收到他没有订阅的杂志。这正是观察者模式的工作方式。观察者通过通知器（发行商）把自己注册到（订阅）特定的通知（杂志）。当有通知的时候，观察者只从通知器得到它订阅的通知。它们的静态关系如图12-1所示。

观察者模式是一种发布-订阅模型。Observer从Subject订阅通知。ConcreteObserver实现抽象Observer并重载其update方法。一旦Subject的实例需要通知Observer任何新的变更，Subject会发送update消息来通知存储在内部列表中所有注册的Observer。在ConcreteObserver的update方法的实际实现中，Subject的内部状态可被取得并在以后进行处理。

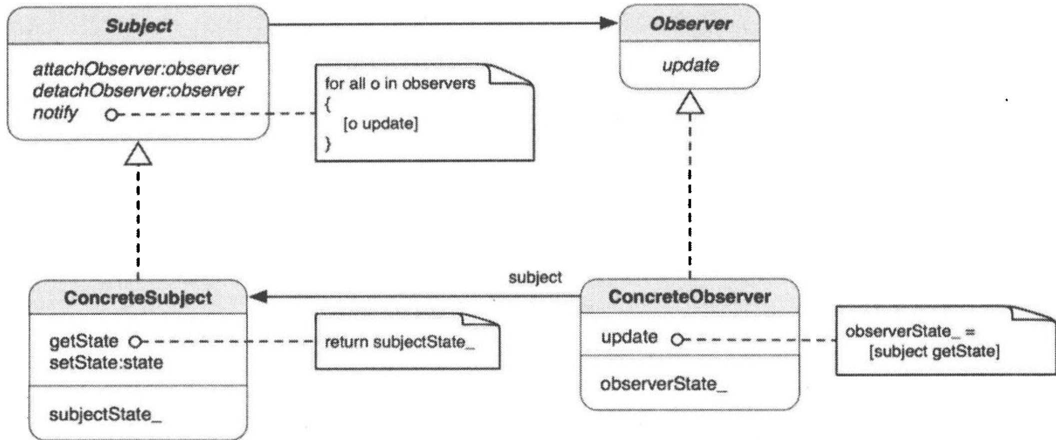


图12-1 观察者模式的类图

> **观察者模式**：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。*
>
> * 最初的定义出现于《设计模式》(Addison-Wesley，1994)。

发布–订阅机制的整体思想相当简单而且很好理解。`Subject`提供注册与取消注册的方法，任何实现了`Observer`协议而且想要处理`update`消息的对象，都可以进行注册或取消注册。当`Subject`的实例发生变更时，它会向自己发送`notify`消息。`notify`方法里有个算法，定义了如何向已注册的观察者广播`update`消息。常见的运行时通知–更新时序见图12-2中的时序图。

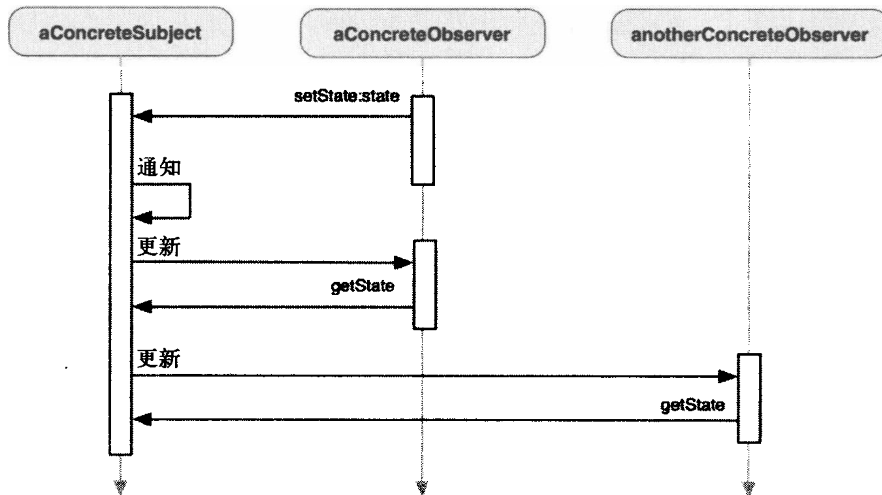


图12-2 显示`aConcreteSubject`与其他具体观察者之间交互的时序图。具体观察者观察来自`aConcreteSubject`对象的更新

aConcreteObserver首先修改aConcreteSubject的状态。因为内部状态发生了变化，aConcreteSubject向自己发送一个notify消息，以便向一组aConcreteObserver广播update消息。aConcreteObserver和anotherConcreteObserver收到消息，然后向aConcreteSubject发送getState消息，以取得其内部状态进行进一步处理。

使用观察者模式的一个最明显的好处是，可以用*N*个Observer来扩展Subject的行为，这些Observer具有处理存储在Subject中的信息的特定实现。它也是一种用于消除不同对象间的耦合的一种设计模式。Cocoa Touch框架向开发者提供了一些类，让开发者不必写自己的类就可以使用这一模式。

12.2 何时使用观察者模式

在以下情形，自然会考虑使用这一模式。

- 有两种抽象类型相互依赖。将它们封装在各自的对象中，就可以对它们单独进行改变和复用。
- 对一个对象的改变需要同时改变其他对象，而不知道具体有多少对象有待改变。
- 一个对象必须通知其他对象，而它又不需知道其他对象是什么。

12.3 在模型-视图-控制器中使用观察者模式

模型-视图-控制器模式（MVC）是一个由各种类型的设计模式组成的复合结构，这在前面几章已讨论过。观察者是其中的设计模式之一。视图会与控制器联系在一起，等待会影响应用程序表现的事件发生。例如，当用户单击视图上的“排序”按钮时，事件会传递给控制器，让模型在后台对其数据进行排序。当模型成功执行了对数据的操作后，它会通知所有相关的控制器，让它们用新的数据更新其视图。

通过在MVC模式中使用观察者模式，每个组件都能够被独立复用与扩展，而对关系中的其他组件没有太多干扰。所得到的高度可复用性与可扩展性，是把其全部逻辑放入一个类中所无法获得的。因此，向控制器添加额外的视图时，不用修改已有的设计和代码。同样，不同的控制器可以使用同一个模型，而不用对使用它的其他控制器作修改。尤其是模型，多个对象（本地的或远程的）能够修改其内部数据。因此模型会向观察中的控制器广播特定的变更。接下来，这些控制器会命令其视图用来自模型的新信息来更新其显示。后面几节将进一步讨论这一更新机制。

12.4 在Cocoa Touch框架中使用观察者模式

Cocoa Touch框架用两种技术改写了观察者模式——通知和键值观察（Key-Value Observing）。尽管是两种不同的Cocoa技术，两者都实现了观察者模式。我们将讨论它们的特征以及两者的差别。

12.4.1 通知

Cocoa Touch框架使用NSNotificationCenter和NSNotification对象实现了一对多的发布-订阅模型。它们允许主题与观察者以一种松耦合的方式通信。两者在通信时对另一方无需多少了解。

主题要通知其他对象时，需要创建一个可通过全局的名字来识别的通知对象，然后把它投递到通知中心。通知中心查明特定通知的观察者，然后通过消息把通知发送给它们。对象订阅了特定类型的通知时，需要通过选择器提供一个方法的名字。这个方法必须符合一种单一参数的签名。方法的参数是通知对象，它包含通知名称、被观察的对象以及带有任何补充信息的字典。当有通知到来时，这个方法会被调用。

模型对象在内部数据改变之后，能够把通知投递到通知中心，使消息能够广播给其他正在观察的对象，然后这些对象可作出适当的响应。模型可以像下面这样构造一个通知然后投递到通知中心：

```
NSNotification *notification = [NSNotification notificationWithName:@"data changes"
                                                              object:self];
NSNotificationCenter * notificationCenter = [NSNotificationCenter defaultCenter];
[notificationCenter postNotification:notification];
```

通知的实例可以用NSNotification类的类工厂方法（见工厂方法模式，第4章），通过指定通知名和作为传给观察者的参数的任何对象来创建。在前面的例子中，通知名是@“data changes”。确切的名字随实现而不同。如果主题要传递自身作为对象参数，可在创建过程中指定self来实现。

一旦创建了通知，就用它作为[notificationCenter postNotification:notification]消息调用的参数，投递到通知中心。通过向NSNotificationCenter类发送defaultCenter消息，可以得到NSNotificationCenter实例的引用。每个进程只有一个默认的通知中心，所以默认的NSNotificationCenter是个单例对象（见单例模式，第7章）。defaultCenter是返回应用程序中NSNotificationCenter的唯一默认实例的工厂方法。

任何要订阅这个通知的对象，首先需要为自己进行注册。如下面的的代码段所示：

```
[notificationCenter addObserver:self
                                 selector:@selector(update:)
                                     name:@"data changes"
                                   object:subject];
```

notificationCenter是用与主题投递通知的步骤里相同的方法得到的。要注册观察者，进行观察的对象需要在addObserver消息调用中把self注册为观察者。它也需要指定选择器，用以识别在通知中心通知这个作观察的对象时被调用的方法。对收到通知时被调用的方法，作观察的对象也可以选择设定所关心的通知的名字，以及任何其他对象作为参数。通知中心用提供的信息来确定应该向作观察的对象分发何种通知。就我们的例子来说，为了接收同一个通知，至少它需要指定同样的通知名。

12.4.2 键-值观察

Cocoa（包括Cocoa Touch）提供了一种称为键-值观察的机制，对象可以通过它得到其他对

象特定属性的变更通知。这种机制在模型–视图–控制器模式的场景中尤其重要，因为它让视图对象可以经由控制器层观察模型对象的变更。

这一机制基于NSKeyValueObserving非正式协议，Cocoa通过这个协议为所有遵守协议的对象提供了一种自动化的属性观察能力。要实现自动观察，参与键–值观察（Key-Value Observing，KVO）的对象需要符合键–值编码（KVC）的要求，并且需要符合KVC的存取方法（accessor method）。KVC基于有关非正式协议，通过存取对象属性实现自动观察（键–值编码的详细规范，请阅读iOS开发者网站上的"Key-Value Coding Programming Guide"（键–值编码编程指南）。也可以使用NSKeyValueObserving的方法和相关范畴来实现手动的观察者通知。对于手动实现，可以禁止默认的自动通知，也可以两者都保留。

通知和键-值观察都是Cocoa对观察者模式的改写。尽管两者都依赖同样的发布者–订阅者关系，但是它们是为不同的解决方案而设计的。表12-1总结了两者的主要差别。

表12-1　通知与键–值观察之间的主要差别

通　知	键-值观察
一个中心对象为所有观察者提供变更通知	被观察的对象直接向观察者发送通知
主要从广义上关注程序事件	绑定于特定对象属性的值

使用键–值观察实现真正的观察者模式，尤其是在模型-视图-控制器的场合，并不是什么难事。但是，这一主题足够自成一章，这里不作详细介绍。如果想对此有更多的了解，可以访问iOS开发者网站，阅读"Key-Value Observing Programming Guide"（键–值观察编程指南）、"Key-Value Coding Programming Guide"（键–值编码编程指南）和"Cocoa Bindings Programming Topics"（Cocoa绑定编程主题）。

我们来通过TouchPainter应用程序，看看如何使用KVO把变更通知从模型（经由控制器）传递到视图。

12.5　在 TouchPainter 中更新 CanvasView 上的线条

TouchPainter应用程序中，让用户可以涂鸦的画布部分是应用程序的心脏和灵魂。在以前的几章简要介绍了所涉及的组件。从整体上看，它基本上是个模型–视图控制器结构：Scribble是模型，CanvasView是视图，CanvasViewController是控制器。CanvasView通过响应者链把触摸传递给CanvasViewController（见责任链模式，第17章）。然后CanvasViewController处理触摸信息并指示Scribble用新的触摸信息更新其内部结构。一旦Scribble更新了其内部状态（Mark组合体；更多细节参见第13章），就会通知所有注册了更新通知的观察者。在这里，CanvasViewController是唯一想从Scribble接受更新的观察者。最后CanvasViewController会通知CanvasView更新其显示。它们的关系如图12-3所示。

我们将通过键-值观察机制（也叫KVO），使用NSKeyValueObserving协议实现这一方案。很幸运，NSObject已经为我们进行了实现，所以不必从头做起。图12-4中的类图表示一种可能的结构。

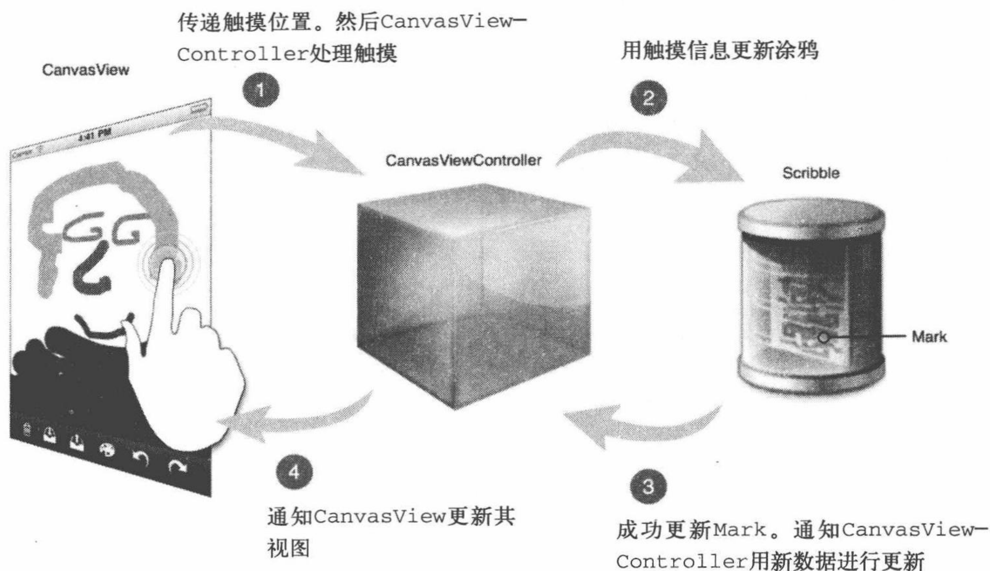


图12-3 表示CanvasView、CanvasViewController和Scribble如何互相交互的动作流程图

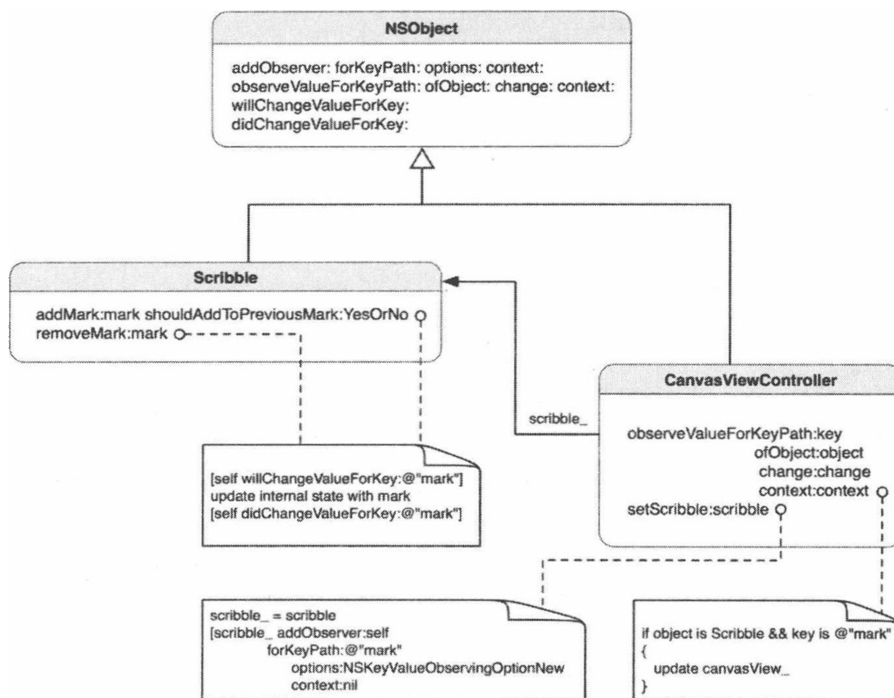


图12-4 表示Scribble和CanvasViewController之间通过键–值观察机制形成的静态关系的类图。键–值观察机制是由NSObject实现的

类图显得有点吓人，是吧？简洁起见，图中省去了无关的方法。因为我们在用KVO来实现这一模式，而且隐藏了很多细节，所以图12-4中的类图与图12-1中的不太一致。但是图中显示的NSObject的两个关键方法会很眼熟。addObserver:forKeyPath:options:context:可以把对象作为观察者添加到接受者。观察者需要重载observeValueForKeyPath:ofObject:change:context:，以接收来自主题变更的回调并处理任何返回的值。在前面几节曾经提到，每当与键匹配的属性通过其存取方法（即set<key>方法）被更新，KVO更新机制通常自动发出通知。如果想手动触发KVO的更新广播，那么就需要使用willChangeValueForKey:和didChangeValueForKey:把变更包起来。

每次把不同的Scribble实例赋给CanvasViewController实例的时候，CanvasViewController会使用addObserver:forKeyPath:options:context:方法把自己作为观察者添加到Scribble的实例。如果Scribble实例的内部状态发生变化（在其Mark组合对象中添加或删除组件），需要通过对willChangeValueForKey:@"mark"和didChangeValueForKey:@"mark"的一对调用，手动触发全局更新。NSObject中的KVO实现会传递observeValueForKeyPath:ofObject:change:context:消息给观察者CanvasViewController。在回调方法中，会用从消息得到的新Mark实例，更新其canvasView_，并令其重画。Scribble的类声明如代码清单12-1所示。

代码清单12-1　Scribble.h中Scribble的类声明

```
#import "Mark.h"

@interface Scribble : NSObject
{
  @private
  id <Mark> parentMark_;
}

// 管理Mark的方法
- (void) addMark:(id <Mark>)aMark shouldAddToPreviousMark:(BOOL)shouldAddToPreviousMark;
- (void) removeMark:(id <Mark>)aMark;

@end
```

Scribble保持一个Mark实例作为其内部状态，使它不暴露给其他对象。客户端可以向Scribble发送addMark:(id <Mark>)aMark shouldAddToPreviousMark:(BOOL)shouldAddToPreviousMark消息，把任何Mark实例插入到其内部组合体。BOOL型参数shouldAddToPreviousMark告诉方法是否把输入的Mark对象作为一个部分附加到原有的聚合体中（例如，Stroke中的Vertex）。removeMark:(id <Mark>)aMark可以把aMark从父Mark中删除。

发给Scribble实例的addMark:或removeMark:消息会触发对其观察者的更新广播。代码清单12-2显示了它们是如何实现的。

代码清单12-2　Scribble.m中Scribble的实现

```
#import "Scribble.h"
```

```
#import "Stroke.h"

// Scribble的私有范畴
// 它有一个只供Scribble对象访问的mark属性
@interface Scribble ()

@property (nonatomic, retain) id <Mark> mark;

@end


@implementation Scribble

@synthesize mark=parentMark_;

- (id) init
{
  if (self = [super init])
  {
    // 父节点应该是个组合对象 (即Stroke)
    parentMark_ = [[Stroke alloc] init];
  }

  return self;
}

#pragma mark -
#pragma mark Methods for Mark management

-(void)addMark:(id<Mark>)aMark shouldAddToPreviousMark:(BOOL)shouldAddToPreviousMark
{
  // 手工调用KVO
  [self willChangeValueForKey:@"mark"];

  // 如果标志设为YES
  // 就把这个aMark加到前一个Mark作为聚合体的一部分
  // 根据我们的设计, 它应该是根节点的最后一个子节点
  if (shouldAddToPreviousMark)
  {
    [[parentMark_ lastChild] addMark:aMark];
  }
  // 否则把它附加到父节点
  else
  {
    [parentMark_ addMark:aMark];
  }

  // 手工调用KVO
  [self didChangeValueForKey:@"mark"];
}

- (void) removeMark:(id <Mark>)aMark
{
  // 如果aMark是父节点则什么也不做
  if (aMark == parentMark_) return;
```

```
  // 手工调用KVO
  [self willChangeValueForKey:@"mark"];

  [parentMark_ removeMark:aMark];

  // 手工调用KVO
  [self didChangeValueForKey:@"mark"];
}

- (void) dealloc
{
  [parentMark_ release];
  [super dealloc];
}

@end
```

@"mark"是Scribble的KVO中唯一的键。通常，如果想修改其内部的mark，可以使用其存取方法（如setMark:）。那样的话，修改完成后不用做任何额外的事情就能广播更新，因为这由NSObject中定义的默认KVO机制来处理。我们肯定需要手动更新，因为Scribble可以对其mark作局部的修改，而不只是用set方法对其作完全的替换。实际上，自动与手动的更新我们都想要。所以我们会保留由NSObject提供的原始机制，并在方法中实际修改mark的地方增加几行代码。在addMark:方法中，有一个if语句块判断传进来的aMark是要添加到父节点还是最后一个Mark。两种情况都会对内部的mark作修改，所以我们把整个语句块用这两个语句包起来：[self willChangeValueForKey:@"mark"]和[self didChangeValueForKey:@"mark"]。第一个语句告诉NSObject保持mark原来的值，而第二个语句保持新的值。其中一个值或者全部两个值将被传给观察者，取决于观察者想要从更新得到什么。

观察者该如何处理更新呢？一旦Scribble发出了一整批更新消息，这就不再跟它有什么关系了。来看看CanvasViewController是如何处理update消息来更新它的canvasView_的，请看代码清单12-3和代码清单12-4中的代码段。

代码清单12-3 CanvasViewController.h中CanvasViewController的类定义

```
#import <UIKit/UIKit.h>
#import "Scribble.h"
#import "CanvasView.h"
#import "CanvasViewGenerator.h"

@interface CanvasViewController : UIViewController
{
  @private
  Scribble *scribble_;
  CanvasView *canvasView_;

  CGPoint startPoint_;
  UIColor *strokeColor_;
  CGFloat strokeSize_;
}
```

```
@property (nonatomic, retain) CanvasView *canvasView;
@property (nonatomic, retain) Scribble *scribble;
@property (nonatomic, retain) UIColor *strokeColor;
@property (nonatomic, assign) CGFloat strokeSize;
@end
```

我们差不多原样重用了其他章里出现的CanvasViewController。Scribble起模型-视图-控制器中模型的作用，帮助存储用户的线条与点的信息。CanvasViewController对象中的Scribble实例成员正是起这个作用。

CanvasViewController也用成员变量canvasView_管理一个CanvasView的实例。它使用Mark的实例在屏幕上进行实际的绘图。

CanvasViewController的实现代码相当长，所以我们把它分成多个部分来讨论。先来看设置部分，见代码清单12-4。

代码清单12-4 CanvasViewController.m中定义的CanvasViewController的实现

```
#import "CanvasViewController.h"
#import "Dot.h"
#import "Stroke.h"

@implementation CanvasViewController

@synthesize canvasView=canvasView_;
@synthesize scribble=scribble_;
@synthesize strokeColor=strokeColor_;
@synthesize strokeSize=strokeSize_;

// 把所有东西挂接到Scribble实例
- (void) setScribble:(Scribble *)aScribble
{
  if (scribble_ != aScribble)
  {
    [scribble_ autorelease];
    scribble_ = [aScribble retain];

    // 把自己作为观察者加到scribble
    // 观察其内部状态——mark的任何变化
    [scribble_ addObserver:self
                forKeyPath:@"mark"
                   options:NSKeyValueObservingOptionInitial |
                           NSKeyValueObservingOptionNew
                   context:nil];
  }
}

// 实现viewDidLoad，进行视图加载后的追加设置，
// 通常视图是从nib加载。
- (void)viewDidLoad
{
  [super viewDidLoad];

  // …
```

```
    // 设置默认的画布视图
    // 但为了简洁起见略去了这一部分
    // ……

    // 初始化Scribble模型
    Scribble *scribble = [[[Scribble alloc] init] autorelease];
    [self setScribble:scribble];

    // 其他设置……
}

- (void)dealloc
{
    [canvasView_ release];
    [scribble_ release];
    [super dealloc];
}
```

因为CanvasViewController的scribble属性是自动合成的，所以通常不需要为它提供任何定制的存取方法。但是事情是这样的：CanvasViewController依靠scribble_发来的更新通知，以进一步指示其canvasView_如何画或重画scribble_中的mark。它需要用下面的消息调用，将自己加为其私有成员变量scribble_的观察者：

```
[scribble_ addObserver:self
            forKeyPath:@"mark"
               options:NSKeyValueObservingOptionInitial |
                       NSKeyValueObservingOptionNew
               context:nil];
```

NSKeyValueObservingOptionInitial选项让scribble_通知CanvasViewController，在这个消息调用之后立刻提供其mark属性的初始值。这个选项很重要，因为当Scribble对象在其init*方法中进行初始化，第一次设置mark属性时，CanvasViewController也需要接收通知。NSKeyValueObservingOptionNew选项指示scribble_，每当其mark属性被设定了新值时通知CanvasViewController。context参数指定可选的对象作为通知的参数。好了，我们回到消息调用本身。问题是，应该把它放在哪儿呢？如果只放在viewDidLoad方法中，那么当客户端把别的Scribble实例赋给控制器的时候，观察连接就会断开。所以它们之间建立连接最好的地方是在scribble_的set存取方法中。它就像一道关口，防止观察连接被破坏的任何可能。同时，在把别的Scribble引用赋给CanvasViewController之后，存取方法会发一个消息给canvasView_，让它用新的Scribble中的mark重画。至于canvasView_是如何在屏幕上描画整个Mark组合体的有关细节，我们将在第15章中讨论。

所以，设置CanvasViewController与其scribble_之间的观察连接的部分，并没有放在viewDidLoad方法中，而是每当控制器被加载时，就在那里创建第一个Scribble实例，并且使用存取方法进行赋值。

CanvasViewController与它的scribble_现在挂接好了，来看看用canvasView_上面的触摸来更新scribble_时的几个动作。我们在CanvasViewController中添加了几个触摸事件处理器，见代码清单12-5。

代码清单12-5 CanvasViewController中的触摸事件处理器

```
#pragma mark -
#pragma mark Touch Event Handlers

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
  startPoint_ = [[touches anyObject] locationInView:canvasView_];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
  CGPoint lastPoint = [[touches anyObject] previousLocationInView:canvasView_];

  // 如果这是手指的拖动，就向涂鸦添加一个线条
  if (CGPointEqualToPoint(lastPoint, startPoint_))
  {
    id <Mark> newStroke = [[[Stroke alloc] init] autorelease];
    [newStroke setColor:strokeColor_];
    [newStroke setSize:strokeSize_];
    [scribble_ addMark:newStroke shouldAddToPreviousMark:NO];
  }

  // 把当前触摸作为顶点添加到临时线条
  CGPoint thisPoint = [[touches anyObject] locationInView:canvasView_];
  Vertex *vertex = [[[Vertex alloc]
                    initWithLocation:thisPoint]
                    autorelease];

  [scribble_ addMark:vertex shouldAddToPreviousMark:YES];
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
  CGPoint lastPoint = [[touches anyObject] previousLocationInView:canvasView_];
  CGPoint thisPoint = [[touches anyObject] locationInView:canvasView_];

  // 如果触摸从未移动（抬起之前一直在同一处）
  // 就向现有Stroke组合体添加一个点
  // 否则就把它作为最后一个顶点添加到临时线条
  if (CGPointEqualToPoint(lastPoint, thisPoint))
  {
    Dot *singleDot = [[[Dot alloc]
                      initWithLocation:thisPoint]
                      autorelease];
    [singleDot setColor:strokeColor_];
    [singleDot setSize:strokeSize_];

    [scribble_ addMark:singleDot shouldAddToPreviousMark:NO];
  }

  // 在此重置起点
  startPoint_ = CGPointZero;

  // 如果这是线条的最后一点
```

```
    // 就用不着画它
    // 因为用户看不出什么区别
  }

  - (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
  {
    // 其实不必在此重置起点
    // 但要以防万一
    startPoint_ = CGPointZero;
  }
```

关于这个应用程序绘图部分的需求的详细讨论，请参阅第2章、第13章和第15章。

我们有一大段代码用来处理触摸事件。目的是要生成两种不同的绘图场景。如果手指在初次接触的地方抬起，就是一个点；如果手指拖动了，就是一条线条（线）。处理这些条件的算法大部分在touchesMoved:和touchesEnded:方法里。在touchesMoved:方法中，如果是拖动的第2点就创建一个新的Stroke实例，并用消息调用[scribble_ addMark:vertex shouldAddToPreviousMark:NO]把它赋给scribble_。拖动中后来的触摸消息将使用对scribble_的同样类型的消息调用，用Vertex的实例添加到以前添加的Mark（一种Stroke形式）中。但是这回shouldAddToPreviousMark参数设成了YES。

如果是手指的单个触摸消息，那么touchesEnded:方法会生成一个新的Dot实例singleDot，把shouldAddToPreviousMark设为NO，让scribble_把它加到其内部结构之中。

在每次addMark:消息调用中，scribble_会通知其观察者（由NSObject维护）。我们知道CanvasViewController刚才已通过addObserver:消息挂接到scribble_。但它还需定义一个回调方法，才能在有更新通知从scribble_发来时做些有意义的事情。KVO处理的一部分要求观察者重载至少一个观察方法。在这个例子中，需要CanvasViewController监听observeValueForKeyPath:消息，如代码清单12-6。

代码清单12-6　CanvasViewController中实现的updateWithScribble:方法，处理当数据变更时从Scribble发来的任何更新消息

```
#pragma mark -
#pragma mark Scribble observer method

- (void) observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context
{
  if ([object isKindOfClass:[Scribble class]] &&
      [keyPath isEqualToString:@"mark"])
  {
    id <Mark> mark = [change objectForKey:NSKeyValueChangeNewKey];
    [canvasView_ setMark:mark];
    [canvasView_ setNeedsDisplay];
  }
}
```

它首先再次确认更新消息是来自Scribble的实例并且是它的@"mark"状态的更新。如果是

这样的话，它将通过向变更字典发送[change objectForKey:NSKeyValueChangeNewKey]消息，从它取得新的Mark。字典里与NSKeyValueChangeNewKey关联的值就是取自scribble_的mark属性的新值。然后，它命令canvasView_使用新的Mark绘制自己。

我们讨论了一些示例代码，示范了为TouchPainter应用程序实现观察者模式的一种可能的方式。现在，你也许要问，为什么要让CanvasViewController用触摸信息更新Scribble，同时等待Scribble发来的通知，然后让CanvasView进行更新呢？好像有些事情的几个附加的步骤可以做得更简单。你要是这么想，就应该返回去再看看Scribble的代码。在Scribble中，有两个方法可以修改其内部Mark组合体引用。一个是向父Mark节点添加新的Mark，另一个是从父节点删除Mark。我们知道CanvasViewController会用addMark:方法把任何Dot和Stroke加到Scribble中。但从Scribble中删除Mark呢？谁会使用这个方法呢？CanvasViewController里没有从scribble_中删除Mark的代码。这个操作很可能会被另一个实体调用——比如，一个命令对象（见命令模式，第20章）。撤销或恢复操作可以在CanvasViewController不知情的情况下修改scribble_。那样的话，CanvasViewController就需要知道它的scribble_在背地里发生了什么。另一种也能在CanvasViewController不知情的情况下修改scribble_的情况是，当模型（模型-视图-控制器中的模型）需要像数据库那样共享给其他对象时，数据库可以由不同进程中的对象，或者甚至通过网络读取和修改。一个对象所作的修改，如果对其他应该保持同步的对象很重要，但是又没能保持同步的话，后果可能非常严重。

如果使用前面几节中讨论的NSNotification和NSNotificationCenter代替KVO来实现同样功能的话，会是下面这个样子。

- 需要为主题和观察者（scribble_和canvasViewController）定义一个共同的标识符。
- 当scribble_的内部状态发生改变时，它会把带有指定标识符的通知，使用任何必要的对象作为参数（用NSNotification的实例）投递到NSNotificationCenter。
- 接下来，所有订阅了标有这个标识符的通知的注册观察者，会从NSNotificationCenter收到这一消息。
- 然后观察者会在作为回调函数提供给NSNotification的选择器中处理这一通知。

除了使用框架提供的这两种方式之外，有些人也许想从零开始构建自己的观察者基础设施。

12.6 总结

本章讨论了观察者模式的背景信息以及这一模式的用途。本章也探讨了如何在TouchPainter应用程序中为其模型-视图-控制器架构实现这一模式，在用户用手指绘图时把线条的变更反映到屏幕上。

我们也可以不必从头开始实现整个方案，而是利用Cocoa Touch框架中使用键-值观察（KVO）以及NSNotification和NSNotificationCenter对象实现好的观察者模式。

在下一个部分，我们将讨论几个用于形成抽象集合结构的设计模式，以及与它们的行为直接相关的其他几个模式。

Part 5

第五部分

抽象集合

本部分内容

- 第 13 章　组合
- 第 14 章　迭代器

第13章 组　合

可以把组合体想象为一个实体，它包含着同一类型的其他实体。整个结构就像由父节点实体和子节点实体连接而成的树。它就像同一个祖先的族谱树一样。族谱树中每个节点（孩子）都有相同的姓。别人称呼我家的时候，管我们叫“Chungs”[①]。这样包括了我家的所有人。因此，要是有人问，“嗨，Chungs，能给我5美元吗？”，那么家里每个人都会掏出5美元。请求可能是递归的，比如问，“Chungs，请问你们家有几口人？”假设我们家曾祖父会收到这个消息，操作将从树根往下传。树中的每个有孩子的家庭（组合体）会对孩子的总数与孩子们返回的数求和，然后把这个和返回。家里没结婚的人（叶节点）呢？他们会返回零作为结果。曾祖父收到了从树中的其他家人返回的所有数之后，他把这些数加起来，把总和作为答案返回给请求者。不必分别请求Chung家的族谱树中每个成员（或家庭）去做某些任务，可以向Chungs（树）发送消息对其整体进行操作。

在面向对象软件设计中我们借用类似的思想。组合结构可以非常复杂，而且其内部表示不应暴露给客户端。我们需要通过统一的接口把整个复杂结构作为一个整体来使用，所以客户端不必知道某个节点是什么就能够使用它。

本章将讨论组合（Composite）模式的概念。我们也会使用第2章的TouchPainter应用程序来讨论如何实现这一模式。在本章稍后，将讨论Cocoa Touch框架如何在`UIView`架构中改写这一模式。

13.1　何为组合模式

组合模式让我们可以把相同基类型（base type）的对象组合到树状结构中，其中的父节点包含同类型的子节点。换句话说，这种树状结构形成“部分-整体”的层次结构。什么是“部分-整体”的层次结构呢？它是既包含对象的组合（容器）又包含作为叶节点（基元）的单个对象的一种层次结构。每个组合体包含的其他节点，可以是叶节点或者是其他组合体。这种关系在这个层次结构中递归重复。因为每个组合或叶节点有相同的基类型，同样的操作可应用于它们中的每一个，而不必在客户端作类型检查。客户端对组合与叶节点进行操作时可以忽视它们之间的差别。图13-1是运行时组合对象结构的一个例子。

---

① 作者Carlo Chung姓Chung。英语中姓氏的复数形式表示整个家庭，即家庭里所有人。所以Chungs表示作者全家。——译者注

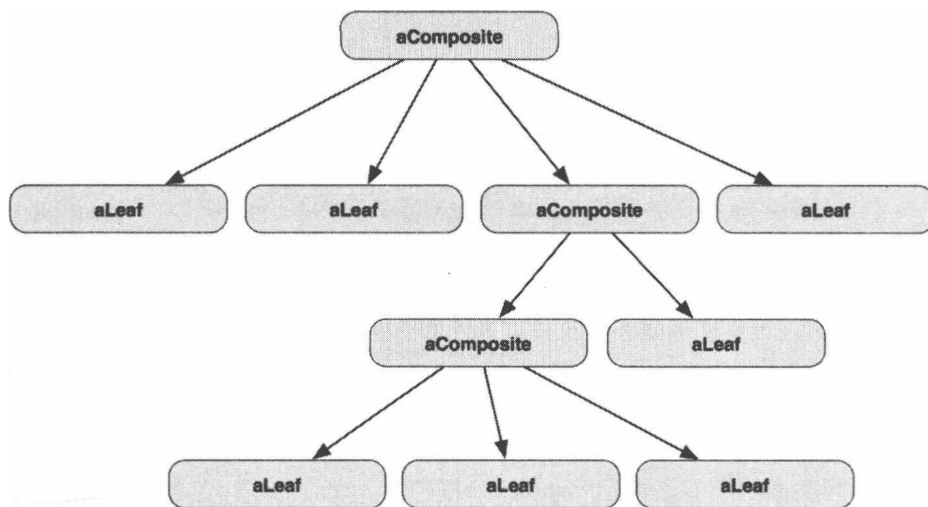


图13-1 一个典型的组合对象结构

组合模式的静态结构如图13-2中的类图所示。

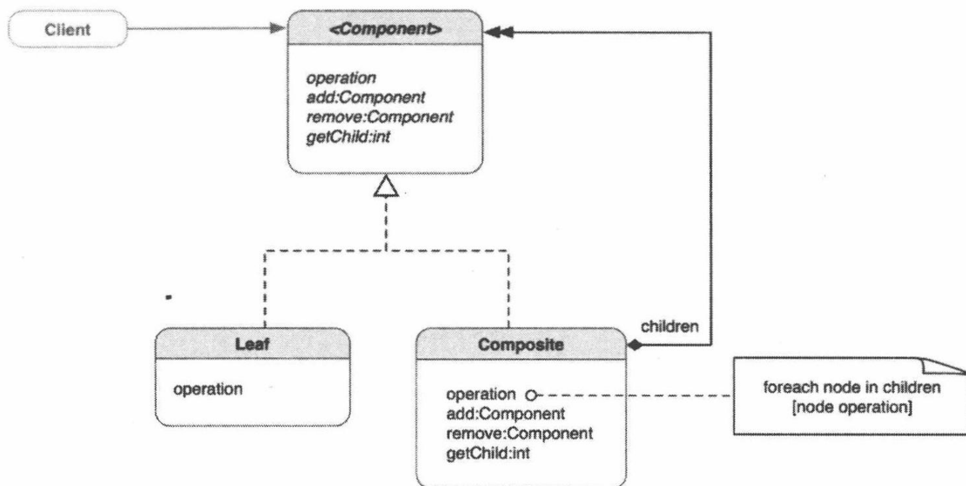


图13-2 说明组合模式的概念性结构的类图

基接口（base interface）是定义了Leaf类和Composite类的共同操作的Component。

有些操作只对Composite类有意义，比如add:Component、remove:Component和getChild:int。出于明显的原因，Leaf类不实现这些方法（其实它做了实现，但只是个空的实现）。为什么不只把这些方法放在Composite类中呢？因为我们不想让客户端在运行时知道它们在处理哪种类型的节点，也不想把组合结构的内部细节暴露给客户端。这就是为什么虽然操作只对Composite类有意义，我们还是把它们声明在基接口，使得各类节点有相同的接口。这样可以让客户端对它们统一处理。

Leaf中的子节点管理

我们知道组合模式的目的是让客户端可以统一处理Leaf和Composite对象，就好像两者都是单个对象。这意味着Leaf类和Composite类必须有共同的接口（方法和属性）。这样客户端就不必知道对象是Leaf还是Composite。

Composite需要方法来管理子节点，比如add:component和remove:component。因为Leaf和Composite有共同的接口，这些方法必须也是接口的一部分。当Leaf也需要实现这些方法时就带来了混乱。有人认为操作Composite对象的子节点的客户端必须知道它所处理的是什么。所以向Leaf对象发送组合体操作消息没有意义，也不起作用，在Leaf类中定义这些操作并没有合理的理由。

每个节点代表一个叶节点或组合体节点。Leaf节点与Composite节点的主要区别在于，Leaf节点不包含同类型的子节点，而Composite则包含。Composite包含同一基类型的子节点。由于Leaf类与Composite类有同样的接口，任何对Component类型的操作也能安全地应用到Leaf和Composite。客户端不必理会那一连窜用来判定被处理对象的确切类型的if-else或switch-case语句。

我们回到层次结构的最顶层。这些节点的祖先节点被当做Component类的实例的引用来对待。如果内部结构有任何变更，客户端代码都不用作修改。

在最终的父节点，整个聚合体结构可以被当做它们的共同的基类型Component来使用。客户端不必知道任何遍历策略就可以使用它，因为遍历或枚举策略或者由组合结构提供，或者通过外部或内部的迭代器来访问（见迭代器模式，第14章）。

组合模式：将对象组合成树形结构以表示“部分-整体”的层次结构。组合使得用户对单个对象和组合对象的使用具有一致性。*

* 最初的定义出现于《设计模式》（Addison-Wesley，1994）。

13.2　何时使用组合模式

在以下情形，自然会想到使用这一模式：

- 想获得对象抽象的树形表示（部分-整体层次结构）；
- 想让客户端统一处理组合结构中的所有对象。

在第2章，对使用抽象的树结构维护用户创建（画）的线条作了简要的讨论。在下面几节，将讨论如何在TouchPainter应用程序中通过实现这一模式来实现这个设计。

13.3　理解 TouchPainter 中 Mark 的使用

让我们回到第2章中定义的Mark组合结构。Mark协议是Dot、Vertex和Stroke类型的基类型。Dot的实例可以画在视图上，而Stroke的子节点Vertex对象只用来帮助在同一线条中把线

连接起来。

懂得如何构建Mark组合结构树的客户端会把Vertex实例添加到Stroke实例，把Stroke实例添加到涂鸦的祖父节点。至于添加可绘制的点，客户端需要把Dot的单个实例添加到祖父节点。它们的关系如图13-3所示。

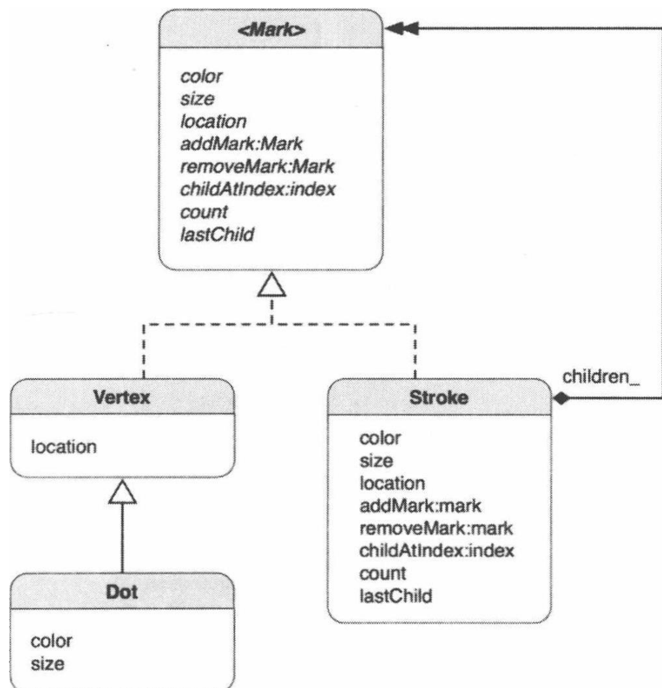


图13-3 组合结构的类图显示了Dot、Vertex和Stroke的类关系。其中Dot为叶节点类，Stroke为容器类

> **说明**：有人有这样的问题：客户端需要知道何时添加 Dot 与何时添加 Vertex，这不是违反了组合模式的封装吗？如果客户端需要知道是否在处理 Stroke、Vertex 或 Dot，这不是与这一模式的目的相左吗？
>
> 对这两个问题的回答都是否定的。这只是因为有两种客户端，一种是组合结构的生成器，另一种填充或操作这个结构。生成器需要知道把什么放入结构中。使用结构的其他客户端则不需要关心其中节点的确切类型。

Vertex只实现了location属性。Dot子类化Vertex并增加color与size属性，因为Vertex不需要color和size而Dot需要。Mark树的直观的运行时结构如图13-4所示。

在运行时aStroke可以包含Dot（aDot）或Vertex（aVertex）对象。因此Stroke对象既可以是各种Mark的父节点，也可以是由Vertex对象构成的真正的线条组合，作为一个整体绘制在屏幕上。

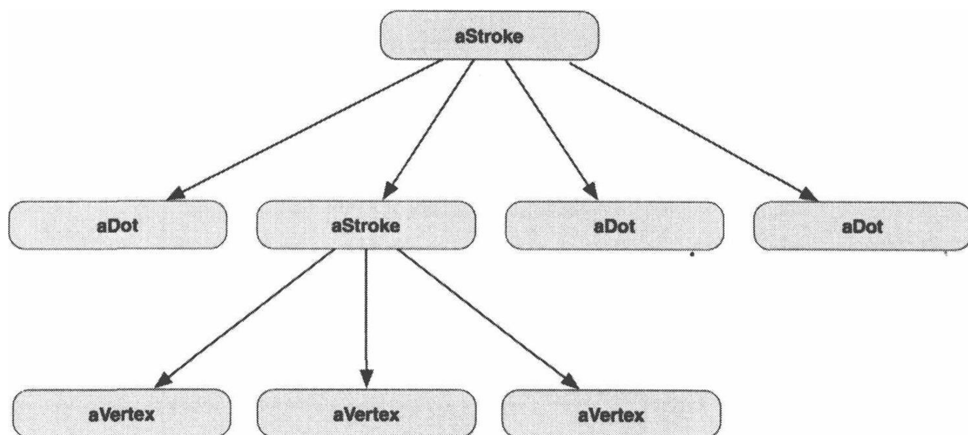


图13-4　组合的对象结构，包含Stroke、Dot和Vertex对象

我们来为Mark协议编些代码，如代码清单13-1所示。

代码清单13-1　Mark.h

```
@protocol Mark <NSObject>

@property (nonatomic, retain) UIColor *color;
@property (nonatomic, assign) CGFloat size;
@property (nonatomic, assign) CGPoint location;
@property (nonatomic, readonly) NSUInteger count;
@property (nonatomic, readonly) id <Mark> lastChild;

- (void) addMark:(id <Mark>) mark;
- (void) removeMark:(id <Mark>) mark;
- (id <Mark>) childMarkAtIndex:(NSUInteger) index;

@end
```

Mark有3个属性：color、size和location。它们应该是任何可以绘制到屏幕上的Mark类型所共有的。在原来的类图中，在Mark类型中声明了两个方法，addMark:mark和removeMark:mark。它们好像只对Stroke类有意义，因为Dot实例根本没有任何组合。把组合操作声明在最上层的抽象接口中的理由是，不想让客户端在添加或删除子节点时做任何运行时检查来判断Mark实例是不是Stroke。目的是让每个节点被客户端用起来都一样。比如，如果需要写一个方法接受Mark型的参数并向它添加或从它删除其他Mark对象，这种统一性显然非常重要。这个方法不用知道所关心的Mark是Stroke还是Vertex（或者是Dot）。

现在来看看Vertex类，见代码清单13-2。

代码清单13-2　Vertex.h

```
#import "Mark.h"

@interface Vertex : NSObject <Mark>
{
```

```
  @protected
  CGPoint location_;
}

@property (nonatomic, retain) UIColor *color;
@property (nonatomic, assign) CGFloat size;
@property (nonatomic, assign) CGPoint location;
@property (nonatomic, readonly) NSUInteger count;
@property (nonatomic, readonly) id <Mark> lastChild;

- (id) initWithLocation:(CGPoint) location;
- (void) addMark:(id <Mark>) mark;
- (void) removeMark:(id <Mark>) mark;
- (id <Mark>) childMarkAtIndex:(NSUInteger) index;

@end
```

重新声明子类重载的属性与方法是个好习惯。虽然Vertex看起来像要实现Mark协议里声明的所有属性，但是只有location属性是真的。location_成员变量声明为@protected，因为它以后会被至少一个子类用到。其余的属性什么也没做（假的），如代码清单13-3所示。

代码清单13-3 Vertex.m

```
#import "Vertex.h"


@implementation Vertex
@synthesize location=location_;
@dynamic color, size;

- (id) initWithLocation:(CGPoint) aLocation
{
  if (self = [super init])
  {
    [self setLocation:aLocation];
  }

  return self;
}

// 默认属性什么也不做
- (void) setColor:(UIColor *)color {}
- (UIColor *) color { return nil; }
- (void) setSize:(CGFloat)size {}
- (CGFloat) size { return 0.0; }

// Mark操作什么也不做
- (void) addMark:(id <Mark>) mark {}
- (void) removeMark:(id <Mark>) mark {}
- (id <Mark>) childMarkAtIndex:(NSUInteger) index { return nil; }
- (id <Mark>) lastChild { return nil; }
- (NSUInteger) count { return 0; }

@end
```

Vertex只用@synthesize合成了location属性，用@dynamic声明了color和size。@dynamic告诉编译器我们会为color和size提供自己的存取方法。虽然去掉@dynamic指令编译器可能不会报错，但是保留它是个好习惯，因为在以后回头再看代码时，就比较容易看懂。initWithLocation:方法只是把aLocation的值赋给Vertex的location属性并返回自身。在Mark中声明的其他组合方法是假的。

Dot子类化Vertex并提供了对Mark协议的更多实现，如代码清单13-4所示。

代码清单13-4　Dot.h

```
#import "Vertex.h"

@interface Dot : Vertex
{
  @private
  UIColor *color_;
  CGFloat size_;
}

@property (nonatomic, retain) UIColor *color;
@property (nonatomic, assign) CGFloat size;

@end
```

Dot只需要用@synthesize声明size和color属性，因为Vertex已经为Dot做好了许多事情。Dot的实现如代码清单13-5所示。

代码清单13-5　Dot.m

```
#import "Dot.h"

@implementation Dot
@synthesize size=size_, color=color_;

- (void) dealloc
{
  [color_ release];
  [super dealloc];
}

@end
```

Vertex和Dot弄好了。我们接着写Stroke这个组合类，如代码清单13-6所示。

代码清单13-6　Stroke.h

```
#import "Mark.h"

@interface Stroke : NSObject <Mark>
{
  @private
  UIColor *color_;
  CGFloat size_;
```

```
  NSMutableArray *children_;
}

@property (nonatomic, retain) UIColor *color;
@property (nonatomic, assign) CGFloat size;
@property (nonatomic, assign) CGPoint location;
@property (nonatomic, readonly) NSUInteger count;
@property (nonatomic, readonly) id <Mark> lastChild;

- (void) addMark:(id <Mark>) mark;
- (void) removeMark:(id <Mark>) mark;
- (id <Mark>) childMarkAtIndex:(NSUInteger) index;

@end
```

因为Stroke是个组合类，跟Vertex与Dot不同，它确实需要实现Mark协议中声明的所有方法与属性，如代码清单13-7所示。

代码清单13-7 Stroke.m

```
#import "Stroke.h"

@implementation Stroke

@synthesize color=color_, size=size_;
@dynamic location;

- (id) init
{
  if (self = [super init])
  {
    children_ = [[NSMutableArray alloc] initWithCapacity:5];
  }

  return self;
}

- (void) setLocation:(CGPoint)aPoint
{
  // 不设定任何位置
}

- (CGPoint) location
{
  // 返回第一个子节点的位置
  if ([children_ count] > 0)
  {
    return [[children_ objectAtIndex:0] location];
  }

  // 否则，返回原点
  return CGPointZero;
}

- (void) addMark:(id <Mark>) mark
```

```
{
  [children_ addObject:mark];
}

- (void) removeMark:(id <Mark>) mark
{
  // 如果mark在这一层，将其移除并返回
  // 否则，让每个子节点去找它
  if ([children_ containsObject:mark])
  {
    [children_ removeObject:mark];
  }
  else
  {
    [children_ makeObjectsPerformSelector:@selector(removeMark:)
                              withObject:mark];
  }
}
- (id <Mark>) childMarkAtIndex:(NSUInteger) index
{
  if (index >= [children_ count]) return nil;

  return [children_ objectAtIndex:index];
}

// 返回最后子节点的便利方法
- (id <Mark>) lastChild
{
  return [children_ lastObject];
}

// 返回子节点数
- (NSUInteger) count
{
  return [children_ count];
}

- (void) dealloc
{
  [color_ release];
  [children_ release];
  [super dealloc];
}
@end
```

Stroke用自己的children_作为NSMutableArray的实例，来保存Mark子节点。在其init方法中，它随便用容量5虽然再大一点也没关系来初始化一个NSMutableArray的实例。

位置属性没有用@synthesize来声明，因为Stroke对象没有直接提供location值，而是从第一个子节点中取得。如果Stroke对象没有子节点，location存取方法就返回表示屏幕原点的CGPointZero。

addMark: (id <Mark>) mark方法使用addObject:mark把mark参数传给children_。removeMark: (id <Mark>) mark有点复杂。它先在children_层上搜索mark参数。如果没

有找到，就通过makeObjectsPerformSelector:@selector(removeMark:) withObject:mark消息，把搜索操作传递给每个子节点，因此搜索会一直递归地进行下去，直到到达树的最后一个子节点。每个Mark对象都实现了removeMark方法，因此在发消息之前不用考虑某个子节点是Stroke、Vertex还是别的什么。

childMarkAtIndex:、lastChild和count方法只返回children_（NSMutableArray）的类似方法。

客户端构造Mark组合体结构的方式如代码清单13-8所示。

代码清单13-8 用Dot、Vertex和Stroke对象创建Mark组合体结构的客户端代码

```
// ...

Dot *newDot = [[[Dot alloc] init] autorelease];
Stroke *parentStroke = [[[Stroke alloc] init] autorelease];

[parentStroke addMark:newDot];

// ...

Vertex *newVertex = [[[Vertex alloc] init] autorelease];
Stroke *newStroke = [[[Stroke alloc] init] autorelease];

[newStroke addMark:newVertex];

[parentStroke addMark:newStroke];

// ...
```

单个Dot对象可被添加到parentStroke作为叶节点。parentStroke也可以接受组合体Stroke对象，组合体Stroke对象为了让绘图算法绘制相连的线，管理着自己的Vertex子节点。

在TouchPainter应用程序中，有一个CanvasViewController，处理一些基本的有关触摸的绘图操作。控制器中的触摸事件操作构造一个主Mark组合结构体，包含所有单个的、可绘制的点和线条，线条包含其他不可绘制的顶点以形成相连接的线。CanvasViewController的视图叫做CanvasView，会在屏幕上显示（绘制）整个Mark结构。

图13-5显示了一个画点和线条的实际绘图场景。

可以向Mark协议添加一个绘图操作，如drawWithContext:(CGContextRef)context，让每个节点可以根据其特定目的绘制自己。Dot中的绘图方法如代码清单13-9所示。

代码清单13-9 Dot中的drawWithContext:的实现

```
- (void) drawWithContext:(CGContextRef)context
{
  CGFloat x = self.location.x;
  CGFloat y = self.location.y;
  CGFloat frameSize = self.size;
  CGRect frame = CGRectMake(x, y, frameSize, frameSize);

  CGContextSetFillColorWithColor (context,[self color CGColor]);
  CGContextFillEllipseInRect(context, frame);
}
```

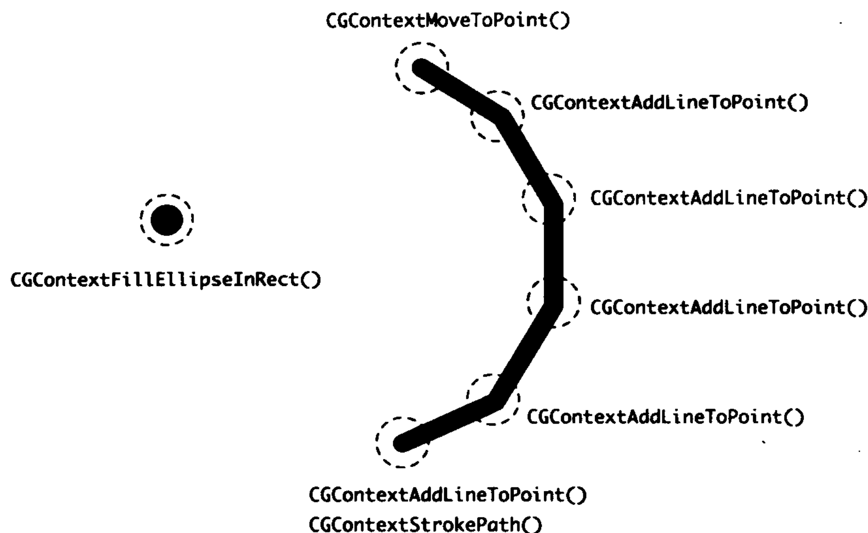


图13-5　使用Quartz 2D画点与线条的实际顺序

使用当前的上下文（context），可以指定位置、颜色和大小绘制一个椭圆（一个点）。

但是，顶点只提供了线条中特定的位置。因此Vertex对象只会在上下文中用其位置（坐标）往线上添加一个点（实际上这个Quartz 2D函数是向点添加一条线），如代码清单13-10所示。

代码清单13-10　Vertex中的drawWithContext:的实现

```
- (void) drawWithContext:(CGContextRef)context
{
  CGFloat x = self.location.x;
  CGFloat y = self.location.y;

  CGContextAddLineToPoint(context, x, y);
}
```

至于Stroke对象，它需要把上下文移动到其子节点的第一点，然后使用Quartz 2D函数CGContextSetStrokeColorWithColor和CGContextStrokePath完成整条线的绘图制作，如代码清单13-11所示。

代码清单13-11　Stroke中的drawWithContext:的实现

```
- (void) drawWithContext:(CGContextRef)context
{
  CGContextMoveToPoint(context, self.location.x, self.location.y);

  for (id <Mark> mark in children_)
  {
    [mark drawWithContext:context];
  }

  CGContextSetStrokeColorWithColor(context,[self.color CGColor]);
```

```
    CGContextStrokePath(context);
}
```

设计Mark协议时的主要挑战是以最小的操作集提供可扩充的功能。为添加新功能而对Mark协议及其子类做手术，不但会带来创伤而且易于出错。最终，对类的理解、扩展与复用变得更加困难。因此，关键是要致力于简单而一致的接口的一组充分的基本要素。

在有关访问者的一章（第15章），我们将探讨组合模式与访问者模式如何协同工作，针对树形结构相关的许多问题建立功能强大而灵活的解决方案。

13.4 在 Cocoa Touch 框架中使用组合模式

在Cocoa Touch框架中，UIView被组织成一个组合结构。每个UIView的实例可以包含UIView的其他实例，形成统一的树形结构。让客户端对单个UIView对象和UIView的组合统一对待。

窗口中的UIView在内部形成视图的树形结构。层次结构的根部是一个UIWindow对象和它的内容视图。添加进来的其他UIView成为它的子视图。它们的每一个可以包含其他视图而变成自己的子视图的超视图。UIView对象只能有一个超视图，可以有零到多个子视图。图13-6说明了这种关系。

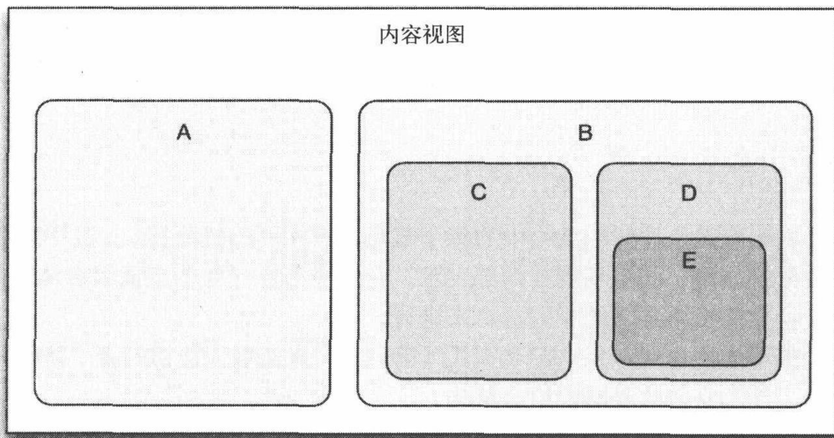


图13-6 UIView的直观结构

它们的关系可以从另一个角度来看，如图13-7中的层次结构图所示。

视图组合结构参与绘图事件处理。当请求超视图为显示进行渲染时，消息会先在超视图被处理，然后传给其子视图。消息会传播到遍及整个树的其他子视图。因为它们都是相同的类型——UIView，它们可以被统一处理，而且UIView层次结构的一个分支也可以同样当做一个视图来处理。

统一的视图也作为一个响应者链，用于事件处理和动作消息。绘图消息像事件处理消息那样，顺着结构从超视图向子视图传递。Cocoa Touch框架中的响应者链是对责任链模式的实现（见第17章）。

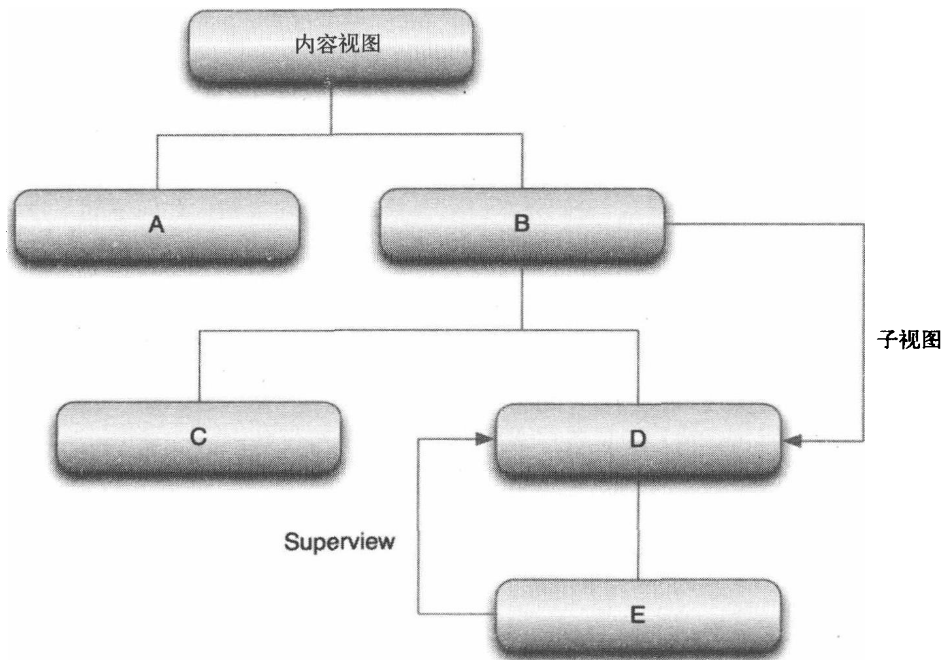


图13-7　UIView的层次结构

13.5　总结

组合模式的主要意图是让树形结构中的每个节点具有相同的抽象接口。这样整个结构可作为一个统一的抽象结构使用，而不暴露其内部表示。对每个节点（叶节点或组合体）的任何操作，可以通过协议或抽象基类中定义的相同接口来进行。

对这个结构新增的操作可以用访问者模式（第17章）来实现，让访问者“访问”每一节点进行进一步的处理，而不必修改现有的组合结构。

组合结构的内部表示不应暴露给客户端，因此组合模式总是跟迭代器模式一起使用，以遍历组合对象中的每一个项目。迭代器模式和相关的设计问题将在下一章中讨论。

第14章 迭 代 器

每次从自动售货机买汽水的时候，顾客投几个硬币进去，选择想要的汽水，然后它就被送到出货托盘。

显然，自动售货机里不止装了一瓶汽水。顾客只是不知道这些瓶子是怎么放在这个大铁柜子里的，又是怎么出货的，除非拆开机器看看里面是怎么回事。不然的话，就只好每次买汽水时等着机器来发售下一瓶汽水。该从货仓中的一堆汽水里发售哪一瓶，机器里运行的分配器软件一清二楚。

售货机中至少有两个主要部件在完成这个工作：

- 容纳一堆汽水（汽水的集合）的内部货架；
- 从一堆汽水（集合）中取出下一瓶的分配器。

内部货架不知道发售的事情，它唯一的作用就是“容纳”瓶子。它依赖分配器来进行所有实际的分配与出货操作。除非售货机的箱体是透明的，否则我们不会知道过程中发生了什么。

内部货架就像MVC对象结构中的模型，或者只是个数据结构。其唯一职责就是维护一个数据集合（一堆瓶装汽水），仅此而已。有多种方法可以枚举数据结构中的数据（发放内部货架中的瓶子）。要是把多种发放瓶子的方式都塞给内部托盘，系统可能会非常复杂而难以维护。因此最好使用分离的机械装置来发放（枚举）瓶子，尤其是当制造商想在其他型号的售货机上复用同一种分配装置的时候。

在面向对象软件中，针对抽象集合迭代行为的设计模式叫做迭代器（Iterator）。本章将讨论这一模式的思想，并通过Cocoa Touch基础框架实现各种类型的迭代器。

14.1 何为迭代器模式

迭代器提供了一种顺序访问聚合对象（集合）中元素的方法，而无需暴露结构的底层表示和细节。遍历集合中元素的职能从集合本身转移到迭代器对象。迭代器定义了一个用于访问集合元素并记录当前元素的接口。不同的迭代器可以执行不同的遍历策略。

在图14-1的类图中可以看出集合与迭代器的基本关系。

`List`定义了修改集合以及返回集合中元素个数的方法。`ListIterator`保持一个对`List`对象的引用，以便迭代器遍历结构中的元素并将其返回。`ListIterator`定义了让客户端从迭代过程中访问下一项的方法。迭代器有个内部的`index_`变量，记录集合中的当前位置。聚合体与迭

代器之间更为详细的关系如图14-2中的类图所示。

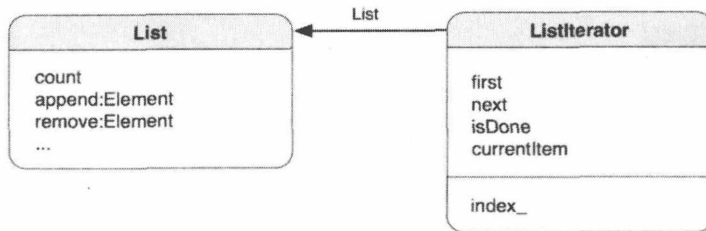


图14-1　说明List与ListIterator之间关系的类图

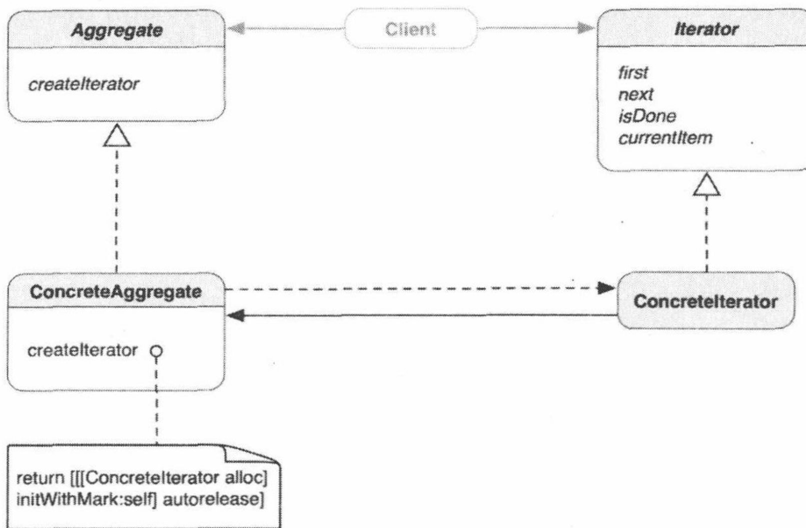


图14-2　更进一步表示抽象列表与迭代器之间关系的类图

抽象的Aggregate定义了createIterator方法，它返回一个Iterator对象。Concrete-Aggregate对象子类化Aggregate，重载其createIterator方法并返回ConcreteIterator的实例。Iterator抽象类定义了所有Iterator应具有的基本迭代行为。客户端会使用定义好的抽象接口来遍历任何Aggregate类型对象中的元素。

> **迭代器**：提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。*
>
> *最初的定义出现于《设计模式》（Addison-Wesley，1994）。

外部迭代器与内部迭代器

基本上有两种迭代器：外部迭代器和内部迭代器。外部迭代器让客户端直接操作迭代过程，所以客户端需要知道外部迭代器才能使用。另一种情况是，集合对象（被迭代的目标对象）在

其内部维护并操作一个外部迭代器。提供内部迭代器的典型的集合对象为客户端定义一个接口，或者从底层的集合一次访问一个元素，或者向每个元素发送消息。外部迭代器与内部迭代器的区别总结在表14-1中。

表14-1 外部迭代器与内部迭代器区别的总结

外部迭代器	内部迭代器
客户端需要知道外部迭代器才能使用，但是它为客户端提供了更多的控制	客户端不需要知道任何外部迭代器，而是可以通过集合对象的特殊接口，或者一次访问一个元素，或者向集合中的每个元素发送消息
客户端创建并维护外部迭代器	集合对象本身创建并维护它的外部迭代器
客户端可以使用不同外部迭代器实现多种类型的遍历	集合对象可以在不修改客户端代码的情况下，选择不同的外部迭代器

14.2 何时使用迭代器模式

在以下情形，自然会想到使用迭代器模式：

- ❑ 需要访问组合对象的内容，而又不暴露其内部表示；
- ❑ 需要通过多种方式遍历组合对象；
- ❑ 需要提供一个统一的接口，用来遍历各种类型的组合对象。

在下面的几节，我们将通过几个迭代器实现这一模式，这些迭代器遍历第2章中介绍的TouchPainter示例应用程序里的涂鸦图。

14.3 在 Cocoa Touch 框架中使用迭代器模式

苹果公司用自己的命名规则“枚举器/枚举”改写了迭代器模式，用于相关基础类的各种方法。从现在开始，我将使用“枚举”一词，它就是苹果版本的“迭代”。在这个模式中它们是一个意思。基础框架中的`NSEnumerator`类实现了迭代器模式。抽象`NSEnumerator`类的私有具体子类返回枚举器对象，能够顺序遍历各种集合——数组、集（set）、字典（值与键），把集合中的对象返回给客户端。

`NSDirectoryEnumerator`是个关系较远的类。这个类的实例递归枚举文件系统中一个目录的内容。

`NSArray`、`NSSet`和`NSDictionary`这样的集合类，定义了返回与集合的类型相应的`NSEnumerator`子类实例的方法。所有的枚举器都以同样的方式工作。可以在一个循环中向枚举器发送`nextObject`消息，从枚举器取得对象，直到它返回`nil`表示遍历结束。

14.3.1 NSEnumerator

从iOS 2.0开始，可以使用`NSEnumerator`来枚举`NSArray`、`NSDictionary`和`NSSet`对象中的元素。`NSEnumerator`本身是个抽象类。它依靠几个工厂方法（见工厂方法模式，第4章），如

objectEnumerator或keyEnumerator，来创建并返回相应的具体枚举器对象。客户端用返回的枚举器对象遍历集合中的元素，如下面的代码段所示。

```
NSArray *anArray = ... ;
NSEnumerator *itemEnumerator = [anArray objectEnumerator];

NSString *item;
while (item = [itemEnumerator nextObject])
{
  // 对item作些处理
}
```

假设anArray存储着一些NSString对象。在while循环中用NSString的方法对每个item进行处理。

当数组的内容被取完之后，消息调用[itemEnumerator nextObject]会返回nil，然后枚举过程就结束了。

从iOS 4开始，有了另一种枚举Cocoa Touch框架中集合对象的方法，它叫做基于块的枚举。

14.3.2 基于块的枚举

在iOS 4中为Cocoa Touch框架中的集合对象引入了基于块的枚举（Block-Based Enumeration）。块是Objective-C的一项语言功能（本书写作时，苹果公司还在争取把块作为对C语言的扩展而标准化）。块是一种类型化的函数，就是说块是函数也是类型。定义好的块是一个可在方法调用之间传递的变量，就跟对象中的其他变量一样。同时，块变量在方法中可作为函数使用。当把块作为参数传递给方法时，块可以像C程序中的函数指针那样被用作回调函数。因此块正适合于实现内部迭代器（枚举器）。客户端不再需要手动生成迭代器，只需要提供一个符合目标集合对象所要求的签名的块。然后块将在每个遍历步骤中被调用。在每次块被目标集合对象调用时，定义块的算法可以对返回的元素进行处理。

块是Objective-C语言中很酷的一项功能。它让我们可以把回调算法的定义内嵌在消息调用之中。如果不使用块，在Cocoa Touch框架中实现“回调”的传统方式是使用委托（见适配器模式，第8章）。需要为要响应客户端回调的所有对象（适配器）单独定义一个协议（目标）。要是应用程序的这个部分复杂到需要另外的适配器机制的程度，那也未尝不可。有时块可以提供一种比枚举器更漂亮的解决方案。

在iOS 4中，苹果公司在NSArray、NSDictionary和NSSet对象中引入了新方法，用于基于块的枚举。其中一个方法叫enumerateObjectsUsingBlock:(void (^)(id obj, NSUInteger idx, BOOL *stop))block。我们可以把自己的算法定义在内嵌到消息调用之中的块里，或者在别的什么地方预先定义一个块，然后作为参数传给消息调用。下面的代码段通过一个NSArray对象演示了它是如何在代码中实现的。

```
NSArray *anArray=[NSArray arrayWithObjects:@"This", @"is", @"a", @"test", nil];
NSString *string=@"test";
```

```
[anArray enumerateObjectsUsingBlock:^(id obj, NSUInteger index, BOOL *stop)
{
  if([obj localizedCaseInsensitiveCompare:string] == NSOrderedSame)
  {
    // 对返回的obj做点别的事情
    *stop=YES;
  }
}
];
```


要是anArray对象中有个单词是@"test",那么就把指针*stop设置为YES,以通知anArray对象提前停止枚举。块除了id obj和BOOL *stop参数，还有一个NSUInteger index参数。index参数让块中的算法知道当前元素的位置，这对这样的并发枚举非常有用。要是没有这个参数，访问索引的唯一方式就是使用indexOfObject:方法，这样影响效率。

NSSet对象中基于块的枚举与NSArray对象中的非常类似,只是在块签名中没有index参数。NSSet对象是一种模拟“集合”（set）的数据结构，集合中的元素没有表示元素在结构中位置的索引。

使用NSArray、NSDictionary和NSSet的内部迭代器的一个重要好处是，处理其内容的算法可在其他地方由其他开发人员来定义。与传统的for循环中定义的算法不同，定义清晰的块可被复用。当块逐渐变大时，可把它们放到单独的实现文件中，不跟其他代码挤在一起。虽然块是一种为复杂的事物添加内联算法的方便途径，无需定义单独的委托协议，但是当块过大而难以维护时，应该考虑使用策略模式（第19章）。

14.3.3 快速枚举

Objective-C 2.0提供了一种枚举，称为快速枚举。它是苹果公司推荐的枚举方法。它允许把对集合对象的枚举直接用作for循环的一部分，无需使用其他枚举器对象，而且比传统的基于索引的for循环效率更高。快速枚举的语法如下。

```
NSArray * anArray = ... ;

for (NSString * item in anArray)
{
  // 对item作些处理
}
```

现在枚举循环使用指针运算（pointer arithmetic），让它比使用NSEnumerator的标准方法效率更高。

要利用快速枚举，集合类需要实现NSFastEnumeration协议，以向运行库提供关于集合的必要信息。基础框架中的所有集合类与NSEnumerator类都支持快速枚举。因此不必使用while循环从NSEnumerator枚举每个元素，直到nextObject返回nil，我们可以使用其快速枚举的版本，如下面的代码段所示。

```
NSArray * anArray = ... ;
NSEnumerator * itemEnumerator = [anArray objectEnumerator];
```

```
for (NSString * item in itemEnumerator)
{
  // 对item作些处理
}
```

虽然既可以使用集合对象的快速枚举，也可以使用枚举器的快速枚举，但如果只需要默认遍历(通常只按升序)，直接对集合对象进行快速枚举更为合理。NSEnumerator使用其nextObject方法实现NSFastEnumeration协议。从性能上说，它比直接在while循环中手动调用这个方法好不了多少。尽管跟传统的使用nextObject的while循环相比，快速枚举中的for循环显得更为整洁。

实现NSFastEnumeration不在本书的范围，所以不在此讨论它。

14.3.4 内部枚举

NSArray有个实例方法叫(void)makeObjectsPerformSelector:(SEL)aSelector，它允许客户端向数组中每个元素发送一个消息，让每个元素执行指定的aSelector（假定元素支持它）。可以用前面提到的任何一种枚举方法让每个元素执行相同的选择器，达到相同的目的。这个方法在内部枚举集合并向每个元素发送performSelector:消息。这种方式的缺点是如果集合中任何元素不响应选择器，就会抛出异常。因此它主要适用于不需太多运行时检查的简单操作。

14.4 遍历 Scribble 的顶点

在第2章中讨论的TouchPainter应用程序中，有一个组合数据结构，容纳在屏幕上画线条用的所有触摸点。这一结构有个抽象类型叫Mark。Stroke对它进行了实现。第13章中对Mark及组合模式进行了详细讨论。Mark定义了包含单个点和由顶点组成的线条的“部分-整体”组合结构（树）的行为。这个“部分-整体”结构没有定义与遍历有关的任何逻辑。因为它是树，所以它可按不同顺序遍历。如果把任何遍历行为硬编码到树中，以后Mark的接口与实现以及客户代码将面临大量的改动。同把遍历策略放到树结构中相比，更好的办法是，向Mark添加一个工厂方法（见工厂方法，第4章)，让协议的实现者创建并返回适当的枚举器，对树做特定顺序的遍历。我们将通过子类化NSEnumerator为Stroke实现一个迭代器（枚举器）。我们把它叫做MarkEnumerator，它对Mark组合树做后序遍历。NSEnumerator有两个抽象方法：allObjects和nextObject。allObjects返回组合体中未被遍历的Mark实例的数组，而nextObject返回名册中的下一个元素。说明这一思想的类图如图14-3所示。

开始讨论MarkEnumerator之前，先来看一下Mark，这样会对MarkEnumerator如何遍历其子节点有更好的理解。我们需要向Mark添加一个叫enumerator的工厂方法，其实现者可通过这个方法返回NSEnumerator的子类（即MarkEnumerator）的实例，如代码清单14-1所示。

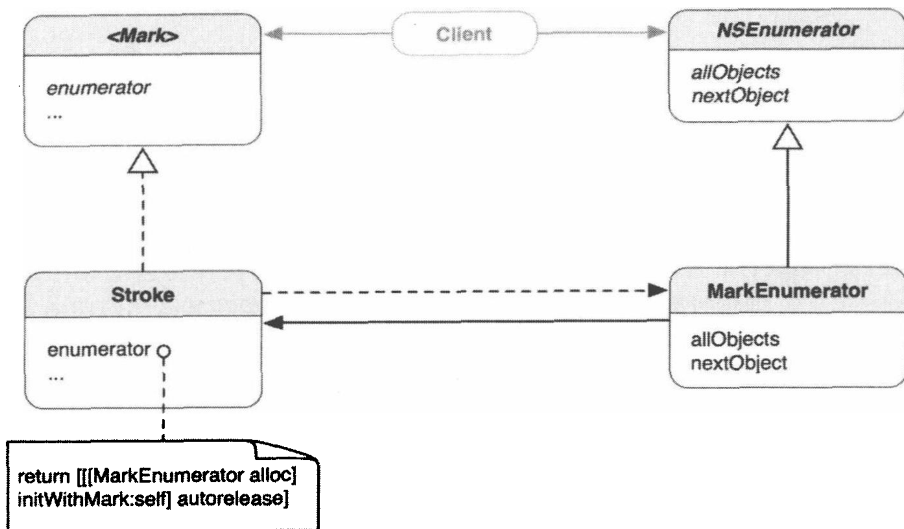


图14-3　表示Client、Mark、Stroke、NSEnumerator和MarkEnumerator之间关系的类图

代码清单14-1　Mark.h

```
@protocol Mark <NSObject>

@property (nonatomic, retain) UIColor *color;
@property (nonatomic, assign) CGFloat size;
@property (nonatomic, assign) CGPoint location;
@property (nonatomic, readonly) NSUInteger count;
@property (nonatomic, readonly) id <Mark> lastChild;

- (void) addMark:(id <Mark>) mark;
- (void) removeMark:(id <Mark>) mark;
- (id <Mark>) childMarkAtIndex:(NSUInteger) index;

- (NSEnumerator *) enumerator;

@end
```

MarkEnumerator会对Mark树结构执行后序遍历，一次枚举一个元素。因此我们打算用栈来实现这个功能。可是，基础框架中没有现成的栈类。我们需要自己动手做一个。但在做之前，先来完成MarkEnumerator的类定义。虽然Objective-C中在子类再次声明重载的方法不是必需的，但这么做是个好习惯。带有重载NSEnumerator方法的MarkEnumerator的类定义如代码清单14-2所示。

代码清单14-2　MarkEnumerator.h

```
#import "NSMutableArray+Stack.h"
#import "Mark.h"

@interface MarkEnumerator : NSEnumerator
```

```
{
  @private
  NSMutableArray *stack_;
}

- (NSArray *)allObjects;
- (id)nextObject;

@end
```

根据我们的类设计，MarkEnumerator对象应该由Mark的实现类创建并初始化。同时，MarkEnumerator在创建时需要知道要处理什么Mark，因此需要在其匿名范畴中定义一个私有的initWithMark:方法，如代码清单14-3所示。

代码清单14-3　MarkEnumerator+Private.h中声明私有方法的匿名范畴

```
@interface MarkEnumerator ()

- (id) initWithMark:(id <Mark>)mark;
- (void) traverseAndBuildStackWithMark:(id <Mark>)mark;

@end
```

把私有方法放到匿名范畴中的理由是，我们想把它们的实现也放到@implementation主代码块。在这个私有范畴中，我们还定义了另一个用于遍历Mark组合对象的私有方法。虽然在两个地方既定义了共有方法又定义了私有方法，但是它们的实现都在同一个.m文件中，如代码清单14-4所示。

代码清单14-4　MarkEnumerator.m

```
#import "MarkEnumerator.h"
#import "MarkEnumerator+Internal.h"

@implementation MarkEnumerator

- (NSArray *)allObjects
{
  // 返回还未访问的Mark节点的数组
  // 也就是栈中的剩余元素
  return [[stack_ reverseObjectEnumerator] allObjects];
}

- (id)nextObject
{
  return [stack_ pop];
}
- (void) dealloc
{
   [stack_ release];
   [super dealloc];
}

#pragma mark -
#pragma mark Private Methods
```

```
- (id) initWithMark:(id <Mark>)aMark
{
  if (self = [super init])
  {
    stack_ = [[NSMutableArray alloc] initWithCapacity:[aMark count]];

    // 后序遍历整个Mark聚合体
    // 然后把单个Mark加到私有栈中
     [self traverseAndBuildStackWithMark:aMark];
  }

  return self;
}

- (void) traverseAndBuildStackWithMark:(id <Mark>)mark
{
  // 把后序遍历压入栈中
  if (mark == nil) return;

  [stack_ push:mark];

  NSUInteger index = [mark count];
  id <Mark> childMark;
  while (childMark = [mark childMarkAtIndex:--index])
  {
    [self traverseAndBuildStackWithMark:childMark];
  }
}

@end
```

在其私有initWithMark:方法中，它使用传进来的Mark引用，然后调用自己的traverseAndBuildStackWithMark:(id <Mark>)mark方法来遍历aMark。这个方法对自身递归调用并把mark及其子节点压入栈中（在栈中构建后序遍历）。注意，子节点是被反向压入栈中的（从右至左）。当访问栈中的元素时，子节点会以原来的顺序被取出（从左至右）。现在Mark聚合体中的所有元素都放入了栈中，已做好了准备，等待客户端向MarkEnumerator发送来nextObject消息取得集合中的下一个Mark。

你也许注意到，nextObject方法只有一条语句——return [stack_ pop];。遍历Mark聚合体之后，最先压入栈中的元素将最后弹出。所以第一子节点将是nextObject方法返回的第一个元素。父节点会在所有子节点之后返回。allObjects应该返回NSArray的实例，包含未被访问的元素的集合。因为栈在集合中向前弹出，并且弹出的元素会从栈中删除，以升序方向返回栈中剩余元素就刚好合适。

我们借助栈的帮助遍历了Mark树，但是基础框架中并没有NSStack这样的类可供我们使用。因此我们需要利用基础类中最接近的一个——NSMutableArray，自己做一个栈，如代码清单14-5所示。

代码清单14-5 NSMutableArray+Stack.h

```
@interface NSMutableArray (Stack)

- (void) push:(id)object;
```

```
- (id) pop;

@end
```

我们向NSMutableArray增加了两个方法作为范畴，像真正的栈一样压入和弹出对象。它的push方法把对象添加在最后面，而pop方法总是返回并删除最后一个元素，如代码清单14-6所示。

代码清单14-6 NSMutableArray+Stack.m

```
#import "NSMutableArray+Stack.h"

@implementation NSMutableArray (Stack)

- (void) push:(id)object
{
  [self addObject:object];
}

- (id) pop
{
  if ([self count] == 0) return nil;

  id object = [[[self lastObject] retain] autorelease];
  [self removeLastObject];

  return object;
}

@end
```

现在我们回到之前离开的Mark家族。Stroke是家族中唯一一个对象含有子节点的成员，因此它实现了enumerator方法，而家族中其他成员没有。简洁起见，省去了其他成员。Stroke的enumerator方法只是创建一个MarkEnumerator实例并用self为参数进行初始化，然后返回这个实例，如代码清单14-7所示。

代码清单14-7 Stroke.m

```
#import "Stroke.h"
#import "MarkEnumerator+Internal.h"

@implementation Stroke

@synthesize color=color_, size=size_;

- (id) init
{
  if (self = [super init])
  {
    children_ = [[NSMutableArray alloc] initWithCapacity:5];
  }

  return self;
}

- (void) setLocation:(CGPoint)aPoint
{
```

```
// 不设定任何位置
}

- (CGPoint) location
{
  // 返回第一个子节点的位置
  if ([children_ count] > 0)
  {
    return [[children_ objectAtIndex:0] location];
  }

  // 否则，返回原点
  return CGPointZero;
}

- (void) addMark:(id <Mark>) mark
{
  [children_ addObject:mark];
}

- (void) removeMark:(id <Mark>) mark
{
  // 如果mark在这一层，将其移除并返回
  // 否则，让每个子节点去找它
  if ([children_ containsObject:mark])
  {
    [children_ removeObject:mark];
  }
  else
  {
    [children_ makeObjectsPerformSelector:@selector(removeMark:)
                               withObject:mark];
  }
}
- (id <Mark>) childMarkAtIndex:(NSUInteger) index
{
  if (index >= [children_ count]) return nil;

  return [children_ objectAtIndex:index];
}

// 返回最后子节点的便利方法
- (id <Mark>) lastChild
{
  return [children_ lastObject];
}

// 返回子节点数
- (NSUInteger) count
{
  return [children_ count];
}

- (void) dealloc
{
```

```
    [color_ release];
    [children_ release];
    [super dealloc];
}

#pragma mark -
#pragma mark enumerator method

- (NSEnumerator *) enumerator
{
    return [[[MarkEnumerator alloc] initWithMark:self] autorelease];
}

@end
```

因为enumerator是工厂方法，所以它可以返回不同的MarkEnumerator子类的对象而无需修改客户代码。如果想要工厂方法支持不同的遍历方式，可以增加一个参数指定遍历类型，以便在运行时选择不同的MarkEnumerator。

> **说明**：在遍历时修改聚合体对象可能有危险。如果向聚合体添加或从聚合体删除了元素，可能导致对一个元素访问两次或完全漏掉一个元素。简单的办法是对聚合体进行一个深复制，再对副本进行遍历，但是如果创建与存储聚合体的另一个副本可能影响性能，代价就比较大。
>
> 有很多方法可以实现不受元素插入与删除影响的迭代器。大部分依靠向聚合体注册迭代器。一种实现方法是在插入与删除操作时，聚合体或者调整由它生成的迭代器的内部状态，或者在内部维护信息，以保证正确的遍历。

遍历 Scribble 的顶点（内部迭代器）

到目前为止所讨论的迭代器（枚举器）是个外部迭代器，就是迭代器模式的原始描述中的那种。它需要客户端请求集合对象返回其迭代器，然后通过迭代器做循环，访问其每个元素。客户端在循环中每次发一个nextObject消息给迭代器，取出一个元素，直到集合取光为止。

可以不让客户端直接使用任何迭代器（枚举器），实现一个内部迭代器作为另一种方式。内部或被动迭代器（通常就是集合对象本身）控制迭代。可以让客户端向内部迭代器提供某种回调机制，让它准备好之后返回集合中的下一个元素，来实现内部迭代器。

从iOS SDK 4.0开始，可以在应用程序的开发中使用Objective-C的块。块是一种函数类型。定义好后，便可在程序中任何地方复用。块在很多方面比C语言的函数指针功能更强大。在Objective-C用块来实现内部迭代器是自然而然的选择。

图14-4中的类图显示了Mark家族的内部迭代器的一种可能的实现方式。

我们向Mark添加了一个新方法：enumerateMarksUsingBlock:markEnumerationBlock。客户端会提供一个定义好的块作为参数，其签名为^(id <Mark> item, BOOL *stop)。块定义了处理从内部迭代器返回的每个Mark元素的算法。如果算法想要在当前位置停止枚举，它可以把*stop变量设为YES。正如前面附表中讨论的那样，是组合对象本身而不是客户端在维护内

部迭代器。enumerateMarksUsingBlock:方法使用通过MarkEnumerator的实例进行的循环来监控枚举过程。从枚举器得到的每个元素会被传给指定的块，以使其中的算法能够处理元素。

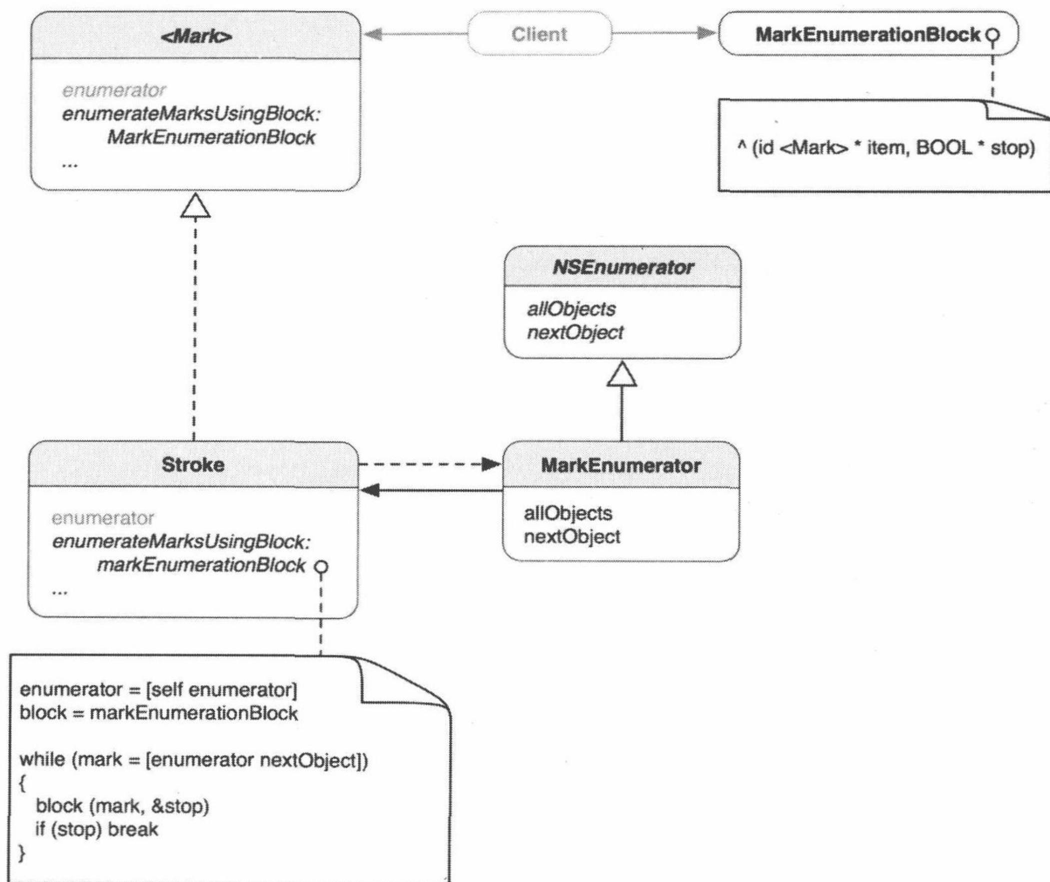


图14-4 MarkEnumerator和Stroke的修改版本的类图。通过引入MarkEnumerationBlock实现了内部迭代器

如果想让客户端只能使用内部迭代器，那么可以将返回MarkEnumerator实例的enumerator工厂方法放到匿名范畴中，就像代码清单14-3中MarkEnumerator类那样。这完全取决于设计。

代码清单14-8和代码清单14-9中的代码段显示了对Mark协议以及Stroke类的修改。

代码清单14-8 Mark.h

```
@protocol Mark <NSObject>

@property (nonatomic, retain) UIColor *color;
@property (nonatomic, assign) CGFloat size;
@property (nonatomic, assign) CGPoint location;
@property (nonatomic, readonly) NSUInteger count;
```

```
@property (nonatomic, readonly) id <Mark> lastChild;

- (void) addMark:(id <Mark>) mark;
- (void) removeMark:(id <Mark>) mark;
- (id <Mark>) childMarkAtIndex:(NSUInteger) index;

- (NSEnumerator *) enumerator;

// 用于实现内部迭代器
- (void) enumerateMarksUsingBlock:(void (^)(id <Mark> item, BOOL *stop)) block;

@end
```

Stroke中对enumerateMarksUsingBlock:(void (^)(id <Mark> item, BOOL *stop)) block的实现如代码清单14-9所示。

代码清单14-9　Stroke.m中enumerateMarksUsingBlock:方法的实现

```
- (void) enumerateMarksUsingBlock:(void (^)(id <Mark> item, BOOL *stop)) block
{
  BOOL stop = NO;

  NSEnumerator *enumerator = [self enumerator];

  for (id <Mark> mark in enumerator)
  {
    block (mark, &stop);
    if (stop)
      break;
  }
}
```

14.5　总结

迭代器模式与访问者模式（第15章）有些类似，尤其是把遍历算法放到访问者模式中或者在遍历聚合体时让内部迭代器对元素执行操作的时候。

其他相关的模式有组合（第13章）、工厂方法（第4章）和备忘录（第23章）。组合模式常常依靠迭代器来遍历其递归结构。多态的迭代器依靠工厂方法来实例化适当的迭代器具体子类。有时，备忘录跟迭代器模式一起使用。迭代器可以使用备忘录来截取迭代的状态。迭代器在内部保存备忘录，在以后从它恢复其内部状态。

本章讨论了主要和抽象集合及其遍历有关的几个设计模式。接下来将要讨论的几个设计模式，使用对现有设计影响最小的方式，扩展抽象集合或其他类型对象的行为。

Part 6 第六部分

行为扩展

本部分内容

- 第 15 章　访问者
- 第 16 章　装饰
- 第 17 章　责任链

第 15 章

访　问　者

打个比方，你家里的管道坏了，可是你自己不会修。虽然你是房子的主人，但这并不意味着你对它里面的一切都了如指掌。因此，解决这个问题最有效率的办法就是，尽快找行家来修理。

在软件设计中，如果架构师为了扩展类的功能而往一个类里塞进了太多方法，类就会变得极为复杂。更好的做法是创建外部的类来扩展它，而对原始代码不作太多改动。

访问者（Visitor）模式可以用日常生活中的例子简单描述一下。在管道问题的例子中，你不会去学习修管子（即向类中添加更多方法），所以你叫来管道工（“访问者”）。他来了就会按门铃，你开门让他进来（“接受”），然后他就进来修管子（“访问”）。

本章会介绍访问者模式的概念，并解释管道工的例子跟它有什么关系。我们会给第2章中的TouchPainter应用程序设计并实现一个访问者，用来绘制用户所画的线条。

15.1　何为访问者模式

访问者模式涉及两个关键角色（或者说组件）：访问者和它访问的元素。元素可以是任何对象，但通常是“部分-整体”结构中的节点（见组合模式，第13章）。部分-整体结构包含组合体与叶节点，或者任何其他复杂的对象结构。元素本身不仅限于这些种类的结构。访问者知道复杂结构中每个元素，可以访问每个元素的节点，并根据元素的特征、属性或操作执行任何操作。图15-1中的类图显示了它们之间的静态关系。

Visitor协议声明了两个很像的visit*方法，用于访问和处理各种Element类型的对象。ConcreteVisitor（1或2）实现这一协议及其抽象方法。visit*的操作定义了针对特定Element类型的适当操作。Client创建ConcreteVisit（1或2）的对象，并将其传给一个Element对象结构。Element对象结构中有一个方法接受一般化的Visitor类型。继承Element的类中，所有acceptVisitor方法中的操作几乎一样，只有一条语句，让Visitor对象访问发起调用的具体Element对象。实际使用的visit*消息，定义在每个具体Element类中，这是具体Element类之间的唯一不同之处。每当把acceptVistor:消息传给Element结构，这个消息就会被转发给每个节点。在运行时确定Element对象的实际类型，再根据实际类型决定该调用哪个visit*方法。

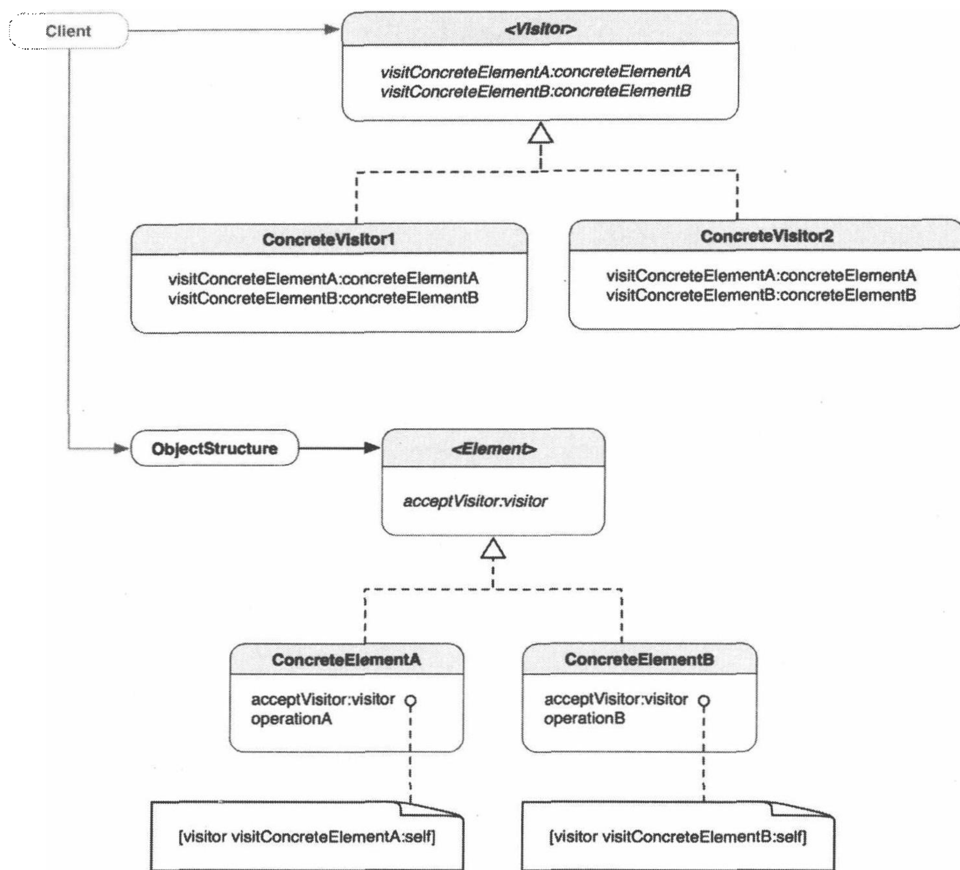


图15-1 显示访问者模式的静态结构的类图

访问者模式：表示一个作用于某对象结构中的各元素的操作。它让我们可以在不改变各元素的类的前提下定义作用于这些元素的新操作。*

* 最初的定义出现于《设计模式》(Addison-Wesley，1994)。

图15-2中的类图显示了适用于前面的“访问者–元素”关系的房屋承包商场景。

访问者模式的这个承包商版本，反映了从房屋承包商的例子角度的一种实现。Plumber（管道工）和Electrician（电工）是访问者。House是个复杂结构，包含有Fixable（可修理的）抽象物品，Contractor（承包商）可对其进行访问并修理。Plumber可以用其专有的visitPlumbing:操作，访问House的Plumbing（管路）结构。同样地，电工可用他的visitElectrical:操作，以同样的理由对同一个House的Electrical（电路）组件进行访问。普通的Contractor好像既会修Plumbing又会修Electrical，但实际上，它把这些工作转包给能实际完成工作的人，跟现实世界里是一样的。你不用管这些细节，只管打开门让他进来，等修好了付钱就行了。承包商（访问者）可以执行特定的技术性工作，而业主不必学习新的技能（修改现有代码）。

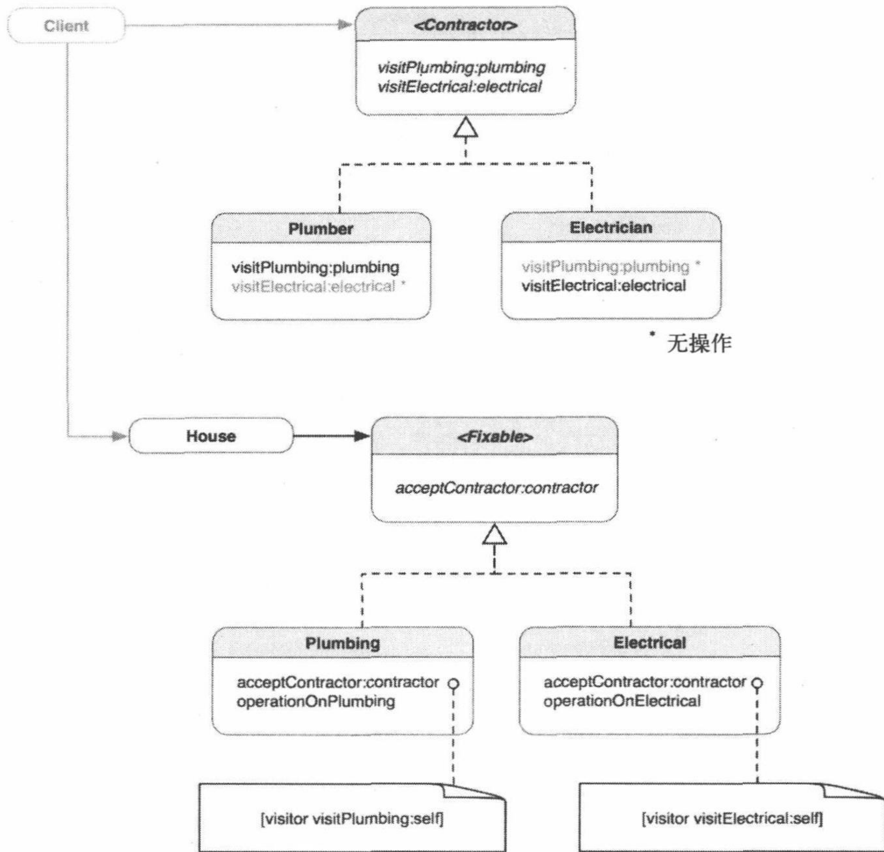


图15-2　承包商例子的类图，其中承包商是访问者

15.2　何时使用访问者模式

在以下场景，可以使用访问者模式。

- 一个复杂的对象结构包含很多其他对象，它们有不同的接口（比如组合体），但是想对这些对象实施一些依赖于其具体类型的操作。
- 需要对一个组合结构中的对象进行很多不相关的操作，但是不想让这些操作“污染”这些对象的类。可以将相关的操作集中起来，定义在一个访问者类中，并在需要在访问者中定义的操作时使用它。
- 定义复杂结构的类很少作修改，但经常需要向其添加新的操作。

15.3　用访问者绘制 TouchPainter 中的 `Mark`

在第2章讨论了TouchPainter应用程序，它的一个主要功能是用户可以通过触摸在屏幕上画任

何图形。在第13章，我们定义了一个组合数据结构，容纳用户生成的线条和点。组合结构的主要目的是，维护这种抽象，让客户端可对其操作，又无需暴露其内部表示和复杂性。为各种叶节点和组合节点定义的上层抽象接口，声明了几个基本操作，任何一个叶节点或组合节点都可以执行这些操作。如果需要向组合体添加操作，则需要修改所有节点类的接口。如果经常这样，会造成巨大影响，对大型项目来说更是如此。设计类的时候，可以预先设想组合体都可能有哪些功能扩展，并把它们应用到类的设计中。但是对于这样一个开放性的问题，这并没有真正解决问题。

更好的办法是向组合体使用访问者模式。访问者可以是相关操作的一个集合，这些操作可以通过组合体对象，根据每个节点的类型来执行。这就像管道工的例子那样，管道工是个访问者，他访问房屋，解决管道的问题。组合体中的每个节点“接受”访问者对节点的“访问”，以解决问题或执行操作。

我们可以应用同样的思想来扩展`Mark`组合体的行为。在第13章中的代码里，我们为每个节点添加了一个绘图操作，使用当前图形上下文在`CanvasView`上执行一些基本的绘图作业。也可以把绘图算法重构为单独的访问者，我们把它叫做`MarkRenderer`。`MarkRenderer`对象可以访问`Mark`组合体的每个节点，并执行任何Quartz 2D操作，以在屏幕上绘制点、顶点和线条。图15-3显示了`MarkRenderer`如何遍历`Mark`组合体来绘制线和点。

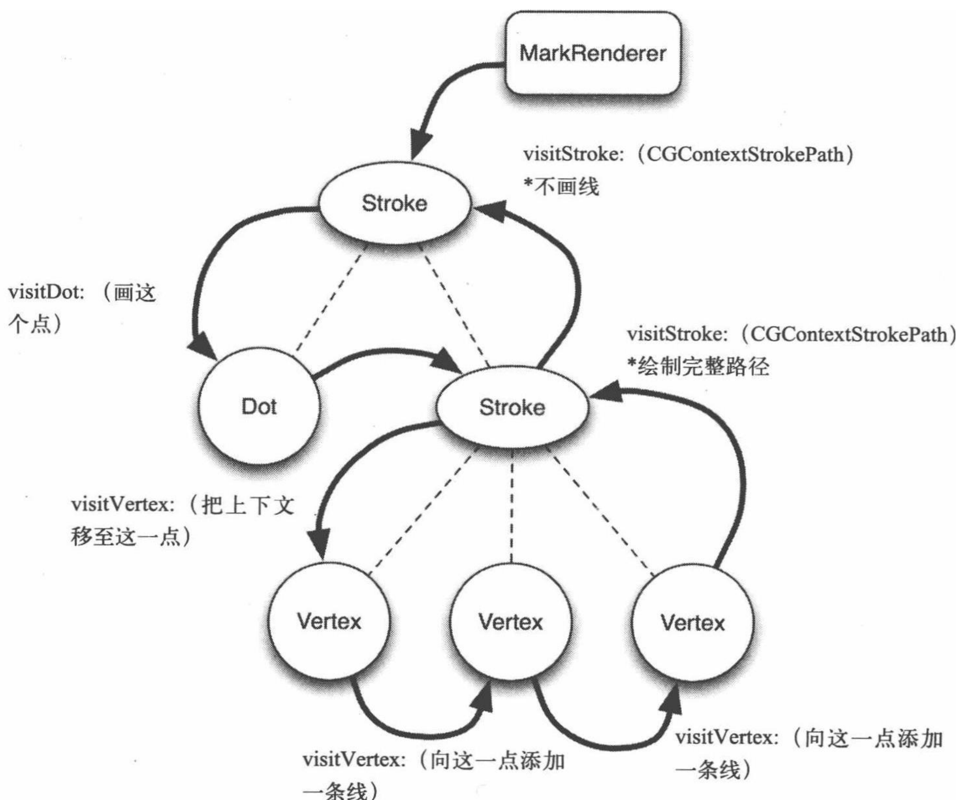


图15-3 使用访问者`MarkRenderer`的实际绘图过程的流程图

在MarkRenderer对象访问Mark组合体结构时，它根据所访问的每个节点的类型，调用相应的Quartz 2D操作。它们的静态关系如图15-4中的类图所示。

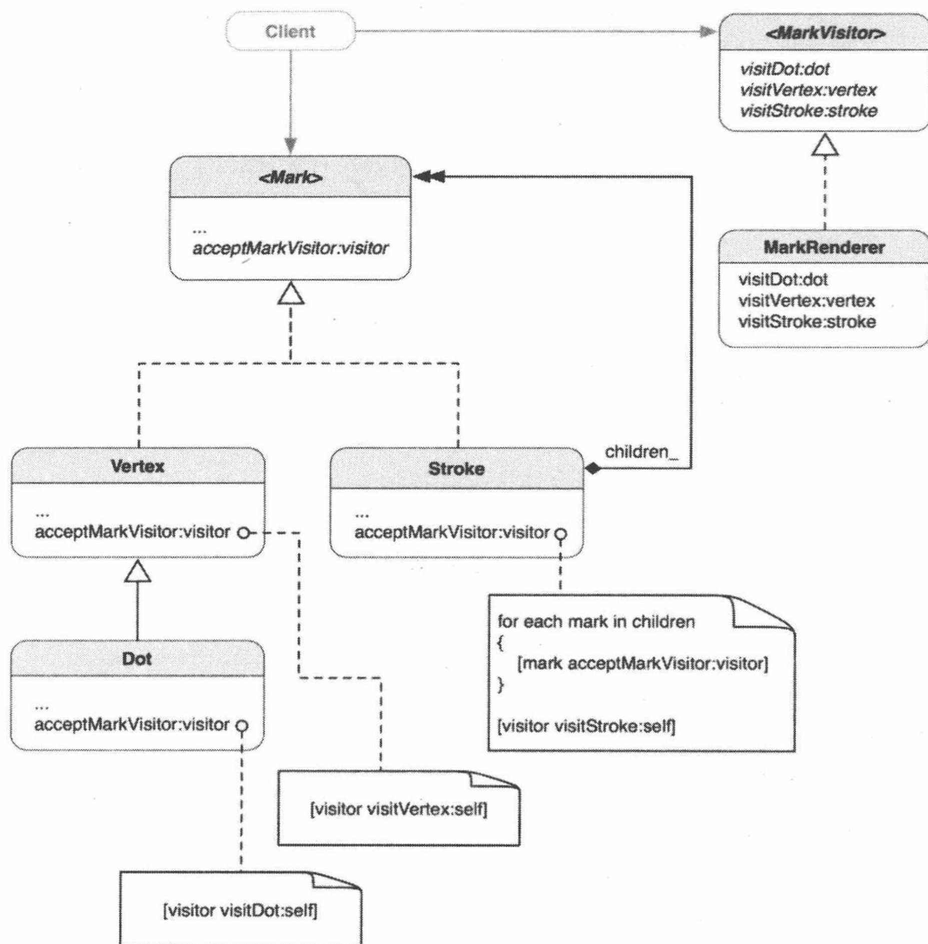


图15-4　说明MarkRenderer以及它和Mark组合体类之间关系的类图

MarkVisitor协议定义了对Mark组合体的各种访问操作，MarkRenderer实现这个协议。每个visit*方法只能用于访问一种特定类型的节点，在运行时由节点本身提供一个节点参数。

在Mark与其子类这一侧，每个节点都添加了一个新方法——acceptMarkVisitor:。当客户端通过acceptMarkVisitor:方法，把一个MarkVisitor对象传给Mark组合体时，这个MarkVisitor对象会沿着整个结构体传递。客户端不知道访问者对象如何在流水线中传递。但是每当访问者对象访问某个节点时，节点的acceptMarkVisitor:方法会向访问者发送相应的visit*消息。根据这个visit*方法，会相应地执行适当的操作。

来看看如何用代码来实现这一切。先看看MarkVisitor协议吧，如代码清单15-1所示。

代码清单15-1 MarkVisitor.h中MarkVisitor协议的声明

```
@protocol Mark;
@class Dot, Vertex, Stroke;

@protocol MarkVisitor <NSObject>

- (void) visitMark:(id <Mark>)mark;
- (void) visitDot:(Dot *)dot;
- (void) visitVertex:(Vertex *)vertex;
- (void) visitStroke:(Stroke *)stroke;

@end
```

这里把MarkVisitor的顶层抽象类型定义为协议，一个理由是，不能确定将来的MarkVisitor的实现类是否需要子类化其他的类来提供服务。

现在，MarkRenderer是NSObject的子类，如代码清单15-2所示。

代码清单15-2 MarkRenderer.h中MarkRenderer的类声明

```
#import "MarkVisitor.h"
#import "Dot.h"
#import "Vertex.h"
#import "Stroke.h"

@interface MarkRenderer : NSObject <MarkVisitor>
{
  @private
  BOOL shouldMoveContextToDot_;

  @protected
  CGContextRef context_;
}

- (id) initWithCGContext:(CGContextRef)context;

- (void) visitMark:(id <Mark>)mark;
- (void) visitDot:(Dot *)dot;
- (void) visitVertex:(Vertex *)vertex;
- (void) visitStroke:(Stroke *)stroke;

@end
```

MarkRenderer有几个私有和保护型的成员变量，稍后会谈到它们。MarkRenderer实现了MarkVisitor及其visit*操作。它也声明了一个初始化方法，接受一个CGContextRef型参数，以后用它绘制Mark组合体中的所有节点。现在来看看MarkRenderer的实现，请看代码清单15-3。我会讲解代码中的几个主要部分。

代码清单15-3 MarkRenderer.m中MarkRenderer的实现

```
#import "MarkRenderer.h"

@implementation MarkRenderer

- (id) initWithCGContext:(CGContextRef)context
```

```
{
  if (self = [super init])
  {
    context_ = context;
    shouldMoveContextToDot_ = YES;
  }

  return self;
}

- (void) visitMark:(id <Mark>)mark
{
  // 默认行为
}
```

visitMark:方法应该为任何未知Mark类型定义默认行为，或被用作“万能”访问者方法，用于未来的新节点类（参见本章稍后的“说明”，那里有对相关问题的讨论）。MarkRenderer没有为Mark实现任何默认行为，那个方法是空的。再来看看下一个方法，请看代码清单15-4。

代码清单15-4　从节点获取位置、大小和颜色信息

```
- (void) visitDot:(Dot *)dot
{
  CGFloat x = [dot location].x;
  CGFloat y = [dot location].y;
  CGFloat frameSize = [dot size];
  CGRect frame = CGRectMake(x - frameSize / 2.0,
                            y - frameSize / 2.0,
                            frameSize,
                            frameSize);

  CGContextSetFillColorWithColor (context_,[[dot color] CGColor]);
  CGContextFillEllipseInRect(context_, frame);
}
```

当访问Dot节点时，从节点得到位置、大小和颜色信息。然后在屏幕的特定位置，使用context_绘制一个实心圆。

绘制Vertex时，情况就不同了。在Quartz 2D中画一条线需要3个基本的步骤：把上下文移至一点，向点加一条线，然后绘制整个路径（path）。前两个步骤应该在访问顶点时决定，我们使用布尔值shouldMoveContextToDot来判定所访问的是否是Stroke组合体中的Vertex节点，如代码清单15-5所示。

代码清单15-5　绘制Vertex

```
- (void) visitVertex:(Vertex *)vertex
{
  CGFloat x = [vertex location].x;
  CGFloat y = [vertex location].y;

  if (shouldMoveContextToDot_)
  {
    CGContextMoveToPoint(context_, x, y);
```

```
    shouldMoveContextToDot_ = NO;
  }
  else
  {
    CGContextAddLineToPoint(context_, x, y);
  }
}
```

当访问者结束了对线条中所有顶点的访问后，最后一个步骤是绘制路径以完成线条，如代码清单15-6所示。

代码清单15-6 完成Stroke的绘制

```
- (void) visitStroke:(Stroke *)stroke
{
  CGContextSetStrokeColorWithColor (context_,[[stroke color] CGColor]);
  CGContextSetLineWidth(context_, [stroke size]);
  CGContextSetLineCap(context_, kCGLineCapRound);
  CGContextStrokePath(context_);
  shouldMoveContextToDot_ = YES;
}

@end
```

然后visitStroke:方法完成线条绘制的过程。Stroke对象决定着整个线条的属性，比如颜色和线宽。我们还需要把线的端点表现为圆形。最后的函数调用CGContextStrokePath()结束了线条绘制过程。

> **说明：** 访问者模式有个需要注意的缺点，那就是，访问者与目标类耦合在一起。因此，如果访问者需要支持新的类（比如向 Mark 家族增加新的节点类型），访问者的父类和子类都需要修改，才能反映新的功能。不过，要是不经常往目标类家族中添加新类，也没什么大问题。
>
> 由于将来对访问者的修改不可预见，为每个访问者准备一个“万能”的访问方法，来支持未来的目标类，是个好主意。我们已经在 MarkVisitor 协议中定义了一个 visitMark:方法（见代码清单 15-1）。这个方法可以应付未来的任何新 Mark 类型。然而，这只是个权宜之计，要是经常需要增加新节点，就应该下定决心，修改访问者的接口，以支持新的节点类型。

至此，完成了针对Mark组合体家族的访问者的实现。但是如果Mark家族不接受访问者的访问，这个设计就不能工作。

如前面所提到的那样，每个节点都需要acceptMarkVisitor:方法，接受任何MarkVisitor的访问。每个节点类型都实现了相同的接口，但是让MarkVisitor以不同的方式访问它。我们来看看Dot中acceptMarkVisitor:方法的实现，请看代码清单15-7。

代码清单15-7 Dot中的acceptMarkVisitor:方法的实现

```
- (void) acceptMarkVisitor:(id <MarkVisitor>)visitor
{
  [visitor visitDot:self];
}
```

这个方法让visitor访问一个Dot（即self）。现在再来看看Vertex中的这个方法，请看代码清单15-8。

代码清单15-8 Vertex中的acceptMarkVisitor:方法的实现

```
- (void) acceptMarkVisitor:(id <MarkVisitor>)visitor
{
  [visitor visitVertex:self];
}
```

visit*消息跟Dot中的几乎一样，但是这回它让访问者访问一个Vertex，即self。

Stroke的情况有些不同，因为它是个组合类，而且它的对象需要管理子节点。同一个方法的实现如代码清单15-9所示。

代码清单15-9 Stroke中的acceptMarkVisitor:方法的实现

```
- (void) acceptMarkVisitor:(id <MarkVisitor>)visitor
{
  for (id <Mark> dot in children_)
  {
     [dot acceptMarkVisitor:visitor];
  }

  [visitor visitStroke:self];
}
```

它遍历所有子节点，并向它们发送acceptMarkVisitor:消息。如前面的代码清单15-7和代码清单15-8所示，运行时节点的类型将决定访问者会使用哪个visit*方法。然后它最后让visitor用visitStroke:消息来访问它自己。从树的遍历的角度来看，这叫做*后序遍历*。这意味着树中的子节点会先于其父节点被访问。为什么要先访问子节点呢？因为需要保证所有的Vertex都被处理了之后，再在Stroke中执行最终的步骤。当然，读者也可以为组合体定义自己的遍历策略，这完全取决于所处理的问题。应该只有组合体自己才知道自己的内部结构和遍历策略。使用迭代器（见迭代器模式，第14章），客户端可以枚举组合结构中的每个节点，而不必知道它的任何内部表示。除后序遍历之外，还有前序遍历和中序遍历，但是在组合对象中后序遍历更为常用。这是因为，对组合对象的操作，往往是整合自各个子节点的处理结果，然后从组合体节点返回一个处理结果。

现在我们做好了一切准备，就等着“管道工”的来访了。我们把其余的处理都放到CanvasView之中吧，请看代码清单15-10。

代码清单15-10 CanvasView的drawRect:方法

```
- (void)drawRect:(CGRect)rect
{
  // 绘图代码
  CGContextRef context = UIGraphicsGetCurrentContext();

  // 创建renderer访问者
  MarkRenderer *markRenderer = [[[MarkRenderer alloc] initWithCGContext:context]
                                                                autorelease];
```

```
    // 把renderer沿着mark组合结构传递
    [mark_ acceptMarkVisitor:markRenderer];
}
```

CanvasView在前面的章节提到过，它负责展现用户绘制的线条和点。任何定制的绘图算法都放在UIView的drawRect:方法里。当视图需要重画或刷新其内容时，它会调用这个方法，执行其中的任何定制的绘图代码。照例，在绘图之前，需要获得当前图形上下文。然后使用这个上下文实例化一个MarkRenderer对象。CanvasView对象有个私有成员变量mark_，它代表运行时用户所画线条的整个抽象组合结构。通过把MarkRenderer作为访问者，用acceptMarkVisitor:消息传给mark_，CanvasView把mark_绘制到屏幕上。然后，随着这个MarkRenderer对象对每个节点的访问，绘图过程会沿着结构传播开来。不管Mark对象有多复杂，只需要3行代码就能把它全部绘制到屏幕上。多么神奇啊！

15.4 访问者的其他用途

这个例子中，用绘制节点对象的访问者扩展了Mark家族类，这样就可以把它们显示到屏幕上了。还可以再增加一个访问者，比如，访问Mark组合体的每个节点，对它实施仿射变换（旋转、缩放、切变、平移）。其他可能的操作包括向Mark组合对象应用各种样式访问者，改变线条的样式，或者创建一个调试结构用的访问者。请注意，如果把这些操作放在Mark接口，那么就需要同时修改每个节点类。因此，一旦对组合结构实现了访问者模式，通常就再也不需要修改组合体类的接口了。

15.5 能不能用范畴代替访问者模式

当然可以使用范畴来扩展任何Mark类型的行为。我们在第13章讲过，可以在每个节点定义一个drawWithContext:方法，来把节点画在屏幕上。但是不必把这个方法定义在每个节点，可以把它定义在独立的范畴中。对应每个节点类型的范畴，实现同样的drawWithContext:方法。这种情况下，需要3个不同的范畴，对应于Mark组合体及叶节点。而使用访问者的话，只需要一个访问者，它可以把针对所有节点的相关算法合并到一处。同样，如需再次扩展这些节点，就需要修改已有的范畴，或者为这些节点创建新的范畴。从扩展现有的Mark接口及其节点类的工作量来说，使用范畴与直接修改相比差别不大。很多情况下，它比使用访问者好不了多少。

15.6 总结

访问者模式是扩展组合结构功能的一种强有力的方式。如果组合结构具有精心设计的基本操作，而且结构将来也不会变更，就可以使用访问者模式，用各种不同用途的访问者，以同样的方式访问这个组合结构。访问者模式用尽可能少的修改，可以把组合结构与其他访问者类中的相关算法分离。

在下一章，将讨论另一种模式，它通过从外部进行“装饰”，同样可以扩展对象的行为。

第16章 装　饰

通常，我们拍照的时候，不会去想将来如何装饰它。拍照只是因为想捕捉那个瞬间。比方说，后来冲印了照片，决定把它放入一个特殊玻璃制成的精美相框中，诸如此类。但是要是改变主意，以后也可以把同一张照片放在别的相框里。虽然换了相框，照片还是那张照片，没受到影响，因为只是往照片添加了东西，而并没有改变它。

在面向对象软件中，我们借用了类似的思想，向对象添加“东西”（行为），而不破坏其原有风格，因此增强了的对象是同一个类的加强版（带相框的照片）。任何“增强”（相框）均可动态添加和删除。我们把这个设计模式叫做“装饰”，装饰对象可以附加到另一个装饰对象，也可以附加到原始对象，对其功能进行扩展，同时保留原始行为不受影响。

本章首先讨论这一模式的概念，以及何时使用这一模式。然后将讨论如何利用这一模式来为UIImage对象设计一组图像滤镜类。

16.1 何为装饰模式

标准的装饰模式包括一个抽象Component父类，它为其他具体组件（component）声明一些操作。抽象的Component类可被细化为另一个叫做Decorator的抽象类。Decorator包含了另一个Component的引用。ConcreteDecorator为其他Component或Decorator定义了几个扩展行为，并且会在自己的操作中执行内嵌的Component操作。它们的关系如图16-1所示。

Component定义了一些抽象操作，其具体类将进行重载以实现自己特定的操作。Decorator是一个抽象类，它通过将一个Component（或Decorator）内嵌到Decorator对象，定义了扩展这个Component的实例的“装饰性”行为。默认的operation方法只是向内嵌的component发送一个消息。ConcreteDecoratorA和ConcreteDecoratorB重载父类的operation，通过super把自己增加的行为扩展给component的operation。如果只需要向component添加一种职责，那么就可以省掉抽象的Decorator类，让ConcreteDecorator直接把请求转发给component。如果有以此方式连接的对象，那么就好像形成了一种操作链，把一种行为加到另一种之上，如图16-2中的对象图所示。

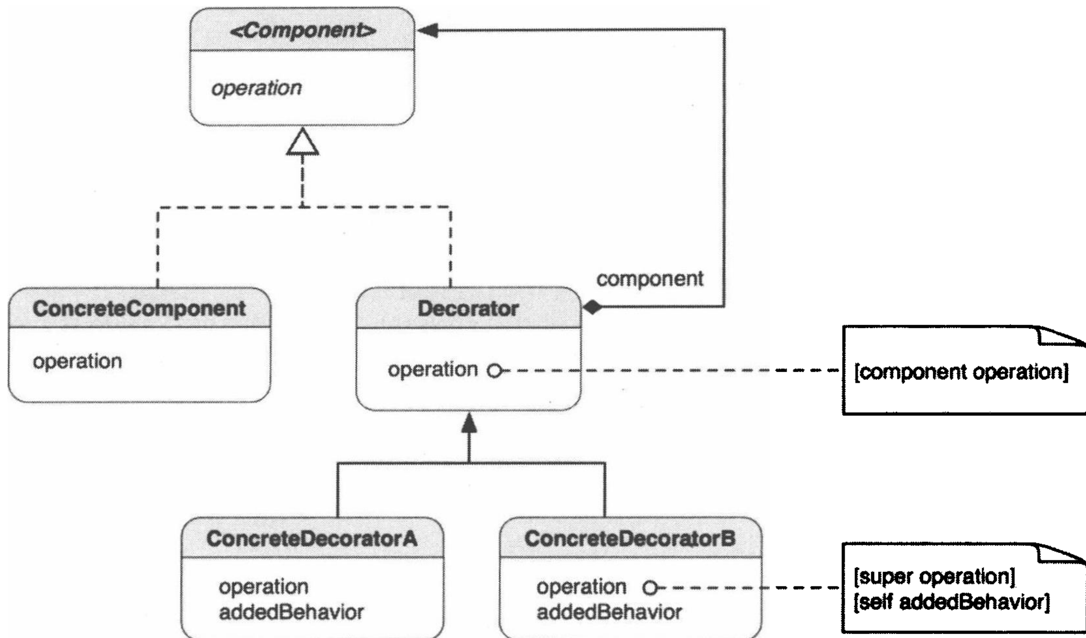


图16-1 装饰模式的类图

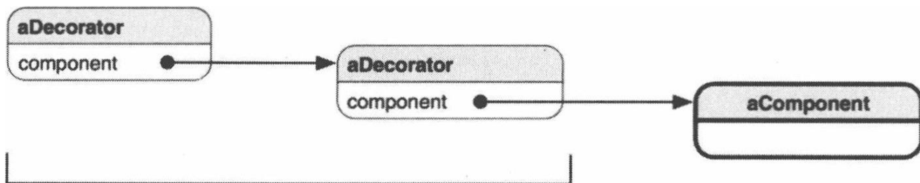


图16-2 装饰模式的一种实现，扩展了装饰性的功能

装饰模式：动态地给一个对象添加一些额外的职责。就扩展功能来说，装饰模式相比生成子类更为灵活。*

* 最初的定义出现于《设计模式》（Addison-Wesley，1994）。

16.2 何时使用装饰模式

在以下3种常见的情形下，应该考虑使用这一模式。

- 想要在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。
- 想要扩展一个类的行为，却做不到。类定义可能被隐藏，无法进行子类化；或者，对类的每个行为的扩展，为支持每种功能组合，将产生大量的子类。

- 对类的职责的扩展是可选的。

16.3　改变对象的“外表”和“内容”

在前面我们已经介绍了，各种装饰器通过内置于每个装饰器节点内部的`component`，在运行时连接起来，如图16-2所示。图中也表明了每个装饰器是从外部改变内嵌的`component`，或者说只是改变对象的外表。关键在于，每个节点并不知道谁在改变它。

然而，一旦每个节点从内部意识到其他不同节点，连接就会向不同方向，也就是向内部伸展。这种模式叫做策略模式（见第19章）。每个节点需要在内部容纳一组不同的API，以挂接另一个策略节点。这一概念的直观表现如图16-3所示。

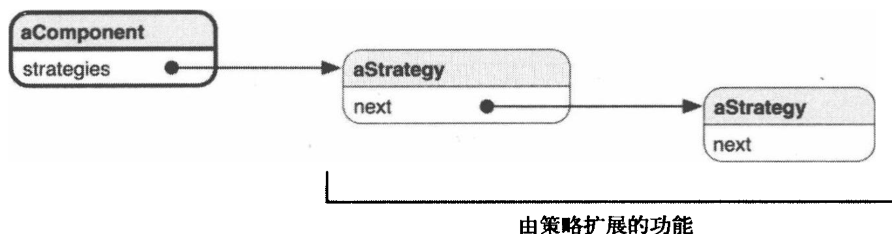


图16-3　用策略模式改变对象的“内容”

变更对象的“外表”（装饰）与“内容”（策略）的本质区别，在表16-1中作了总结。

表16-1　装饰与策略之间的差异的总结

“外表”变更（装饰）	“内容”变更（策略）
从外部变更	从内部变更
每个节点不知道变更	每个节点知道一组预定义的变更方式

16.4　为 `UIImage` 创建图像滤镜

图像滤镜处理是修改图像的颜色与几何特征等属性的一种处理。既可以使用滤镜改变图像的色相（hue）。也可以使用高斯滤镜进行模糊，让图像看起来跟跑焦了一样。甚至可以使用一种二维变形变换，让它看起来不是平面。已经有很多不同的滤镜，可以用来向图像施加“特殊效果”。很多照片编辑软件包，如Photoshop和GIMP，带有各种各样的滤镜。图像滤镜处理跟装饰模式有什么关系呢？

装饰模式是向对象添加新的行为与职责的一种方式，它不改变任何现有行为与接口。比如说有个图像对象，它只包含让客户端管理其属性的接口，仅此而已。我们想向它添加点花哨的东西，比如变换滤镜，但不想修改图像对象的现有接口。我们能做的是，再定义一个跟这个图像对象一样的类，但它包含对另一个图像对象的引用，增加这个图像对象的行为。新的类有一个用绘图上下文绘制自身的方法。在这个绘图方法中，它向内嵌的图像引用应用变换算法，绘制整个图像，

然后返回结果图像。可以把这个过程想象为在图片上放一层玻璃。图片不用去管玻璃，而且我们看它的时候，还管它叫图片。玻璃本身可以有颜色，在表面有波纹，或者其他别的可以让图片看上去不同的东西。以后如果想向图像应用另一层滤镜，可以再定义一个滤镜类，就像变换用的滤镜那样，通过这个类，可以应用同样的机制来向图像增加自己的操作。变换滤镜之外的其他滤镜，可以获取结果图像，并继续处理。但有一点，在装饰滤镜的流水线上传递的图像，不必是原来的那个，但必须是同样的类型。因此从变换滤镜返回的图像是一幅变换后的图像。然后当它被传给颜色滤镜的时候，返回的图像就是一幅经过着色的变换后的图像，如此等等。

Cocoa Touch框架的UIKit中的UIImage，可被初始化为图像对象。UIImage类本身只有非常有限的操作图像的接口，不过是图像尺寸、颜色空间等几个属性而已。我们要用几个Quartz 2D框架中的图像操作工具来扩展普通的图像对象。有两种方式可以实现这个模式：真正的子类和范畴。

16.4.1 通过真正的子类实现装饰

真正的子类方式中，定义了一个与图16-1中这一模式的原始风格类似的结构。唯一区别是，末端组件的类型是UIImage的一个范畴，而不是它的子类。有个小小的结构性问题，不让我们使用与UIImage相同的“接口”。UIImage是NSObject的直接子类，仅此而已。它本身就是一种末端类。为了使用一种Component接口（如图16-1的类图中的父类接口），把UIImage和这些滤镜类连接起来，需要一种创造性的解决方案。现在，我们面临以下两个问题。

- 要让图像滤镜类跟UIImage相同，但UIImage没有可以共有的上层接口（这个模式不推荐使用子类化的方案）。
- UIImage有多个与向当前绘图上下文绘制其内容相关的方法，如drawAsPatternInRect:、drawAtPoint:、drawAtPoint:blendMode:alpha:、drawInRect:和drawInRect:blendMode:alpha:。让图像滤镜类实现同样的方法很复杂，而且由于Quartz 2D的工作方式，可能无法得到预期的结果。这一点稍后再作讨论。

怎么办呢？首先，有一点是肯定的，需要一个接口把UIImage共享给一组滤镜类，使得两种类有共同的基类型，好让这个模式工作。而且我们不想为此把UIImage用作上层类型（即对其子类化），因为这样会让滤镜用起来比较笨重。我们创建了一个叫ImageComponent的接口，它是个协议，作为顶层的基类型。但是，不是说UIImage没有继承任何接口，只是NSObject的直接子类吗？是的，这就是需要创造性解决方案的地方。我们要创建一个UIImage的范畴，实现ImageComponent。然后编译器就知道UIImage和ImageComponent是有关的，就不会报错了。UIImage甚至不清楚它有了个新的基类型。只有使用滤镜的人才需要知道这一点。

而且，我们不会改动UIImage中定义的draw*方法。那怎么向ImageComponent增加绘图行为呢？稍后就会讲到。

表示它们的静态关系的类图如图16-4所示。

ImageComponent协议用UIImage的draw*方法声明了一个抽象接口。任何具体的ImageComponent和装饰器都可以处理这些调用。UIImage实例的draw*消息会把自己的内容绘制到当前图形上下文中。每个方法也考虑了任何应用于上下文的变换和效果。所以我们就可把滤

镜处理加到任何draw*操作之前。

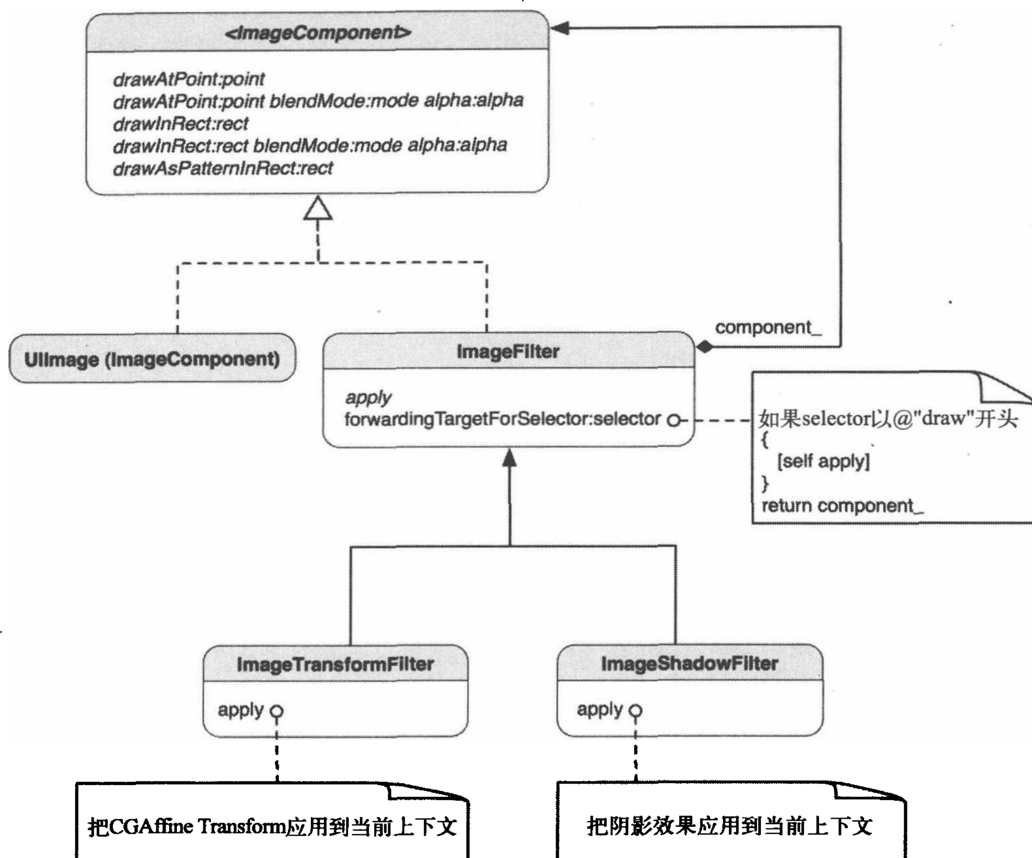


图16-4　实现装饰模式的各种图像滤镜的类图

在这里，具体组件是一个UIImage类型，但我们不想只是为了让它参与进来就继承它，因此我们为它定义了一个范畴。UIImage (ImageComponent)范畴采用了ImageComponent协议。因为协议中定义的方法在UIImage中已经都有了，所以不必在范畴中进行实现。这个范畴只是告诉编译器，UIImage还是一种ImageComponent。

ImageFilter就像图16-1中的抽象Decorator类。ImageFilter的apply方法让具体滤镜子类向component_增加额外的行为。我们没有重载所有draw*方法来添加滤镜行为，而是用了一个 (id) forwardingTargetForSelector:(SEL) aSelector方法来做这个事情。forwardingTargetForSelector:定义于NSObject，这个方法让子类返回代替的接收器来处理aSelector。ImageFilter的实例首先检测aSelector是否为draw*消息。如果是，它就向自己发一个apply消息，增加一些行为，然后返回component_以响应默认行为。apply的默认实现什么也没做。缺少的信息应该由子类提供。这一方式可以保持架构的简洁，而不是让每个具

体滤镜类实现同样的机制来增加行为。

ImageTransformFilter和ImageShadowFilter专注于通过重载apply方法提供自己的滤镜算法。它们继承抽象的ImageFilter基类，ImageFilter里有一个对ImageComponent的引用，即私有成员变量component_。各种ImageComponent对象可在运行时进行连接，如图16-5所示。

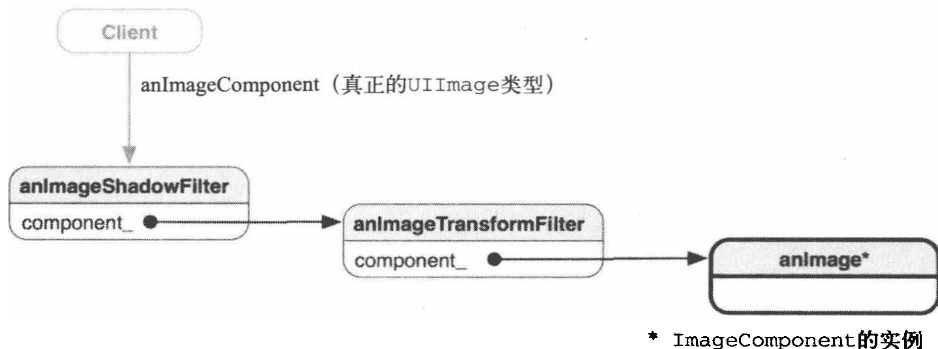


图16-5 ImageComponent的对象图。每个层次的ImageComponent都被另一个ImageComponent实例引用

链条的右端是图16-6中左边的原始图像。我们先是把它添加到anImageTransformFilter，然后又把anImageTransformFilter添加到anImageShadowFilter，客户端将会得到如图16-6中右边所示的图像。每个节点被封装在另一个ImageComponent实例的component_之中。这就像大鱼吃小鱼，小鱼吃虾米。显然，客户端这些装饰器的细节，看到的只是原来的UIImage的实例的引用(以一种ImageComponent的形式，因为UIImage通过它的范畴实现了ImageComponent)。

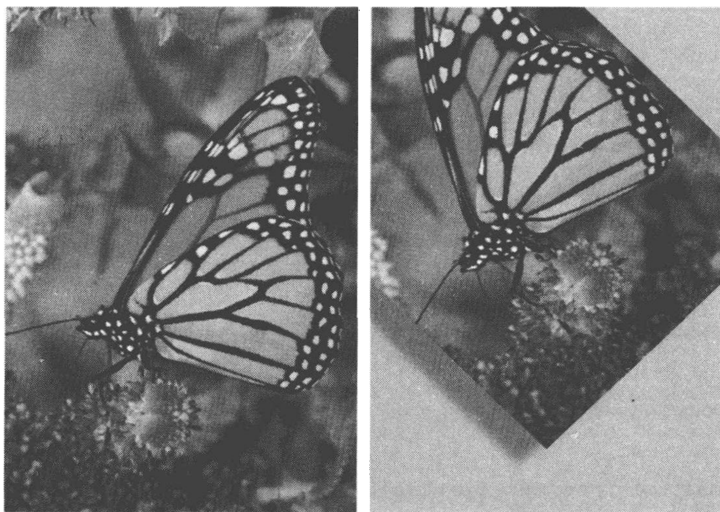


图16-6 原始图像与应用了一系列图像滤镜之后的图像

真令人激动。看看怎么用代码来完成它吧。先来看看声明为协议的抽象ImageComponent，请看代码清单16-1。

代码清单16-1　ImageComponent.h

```
@protocol ImageComponent <NSObject>

// 我们要截获这些UIImage的方法，插入附加的行为
@optional
- (void) drawAsPatternInRect:(CGRect)rect;
- (void) drawAtPoint:(CGPoint)point;
- (void) drawAtPoint:(CGPoint)point
           blendMode:(CGBlendMode)blendMode
               alpha:(CGFloat)alpha;
- (void) drawInRect:(CGRect)rect;
- (void) drawInRect:(CGRect)rect
          blendMode:(CGBlendMode)blendMode
              alpha:(CGFloat)alpha;

@end
```

全部的draw*方法都声明为@optional，因为我们想让所有ImageComponent都能支持这些操作，但实际上又不在实现类中重载它们。关键词@optional告诉编译器，在找不到这些方法的对应实现时不要报错。

代码清单16-2声明了UIImage的范畴，让我们可以把UIImage和其他装饰器一起使用。

代码清单16-2　UIImage+ImageComponent.h

```
#import "ImageComponent.h"

@interface UIImage (ImageComponent) <ImageComponent>

@end
```

它遵守ImageComponent协议，但完全没有实际的实现。现在来看看核心的装饰器类ImageFilter。它的类声明如代码清单16-3所示。

代码清单16-3　ImageFilter.h

```
#import "ImageComponent.h"
#import "UIImage+ImageComponent.h"

@interface ImageFilter : NSObject <ImageComponent>
{
  @private
  id <ImageComponent> component_;
}

@property (nonatomic, retain) id <ImageComponent> component;

- (void) apply;
- (id) initWithImageComponent:(id <ImageComponent>) component;
```

```
- (id) forwardingTargetForSelector:(SEL)aSelector;

@end
```

它用component_保持一个ImageComponent的引用，这个引用会被其他具体装饰器装饰。ImageFilter重载forwardingTargetForSelector:并声明了apply方法。它的实现如代码清单16-4所示。

代码清单16-4　ImageFilter.m

```
#import "ImageFilter.h"

@implementation ImageFilter

@synthesize component=component_;

- (id) initWithImageComponent:(id <ImageComponent>) component
{
  if (self = [super init])
  {
    // 保存ImageComponent
    [self setComponent:component];
  }

  return self;
}

- (void) apply
{
  // 应该由子类重载，应用真正的滤镜
}

- (id) forwardingTargetForSelector:(SEL)aSelector
{
  NSString *selectorName = NSStringFromSelector(aSelector);
  if ([selectorName hasPrefix:@"draw"])
  {
     [self apply];
  }

  return component_;
}

@end
```

在initWithImageComponent:方法中，没做太多的事情，只是把参数的ImageComponent引用赋值给自己。而且，它的apply方法什么也没做，直到在具体滤镜类中重载。

有趣的是，使用forwardingTargetForSelector:截获的消息调用，ImageFilter的实例却不知如何响应。这个方法让子类把替代的接收器转发给运行库，使原始消息得到转发。但我们只关心以@"draw"开头的方法，然后通过返回component_，把剩下的事情都直接转给component_。比如，当消息drawAtRect:被发到ImageFilter的实例时，它将会被forwarding-

TargetForSelector:方法截获，等待替代的接收器，因为ImageFilter并没有实现它。由于消息以"draw"开头，它会在component_处理消息之前，先向自己发一个apply消息来做些处理。

接下来，我们要实现真正的滤镜了。要创建的第一个滤镜是ImageTransformFilter，如代码清单16-5所示。

代码清单16-5　ImageTransformFilter.h

```
#import "ImageFilter.h"

@interface ImageTransformFilter : ImageFilter
{
  @private
  CGAffineTransform transform_;
}

@property (nonatomic, assign) CGAffineTransform transform;

- (id) initWithImageComponent:(id <ImageComponent>)component
                    transform:(CGAffineTransform)transform;
- (void) apply;

@end
```

ImageTransformFilter是ImageFilter的子类，它重载了apply方法。它也声明了一个CGAffineTransform型的私有成员transform_，以及能够访问它的一个关联属性。由于CGAffineTransform是一个C结构体，所以这个属性应该是assign类型的，因为它不能像其他Objective-C对象那样被保持。滤镜有自己的初始化方法。initWithImageComponent: (id <ImageComponent>) component transform: (CGAffineTransform) transform方法，接受一个ImageComponent实例和一个CGAffineTransform值作为参数，进行初始化。component将被转发给super的initWithImageComponent:方法，transform将会被赋值给那个私有成员变量，如代码清单16-6所示。

代码清单16-6　ImageTransformFilter.m

```
@implementation ImageTransformFilter

@synthesize transform=transform_;

- (id) initWithImageComponent:(id <ImageComponent>)component
                    transform:(CGAffineTransform)transform
{
  if (self = [super initWithImageComponent:component])
  {
    [self setTransform:transform];
  }

  return self;
}
```

```
- (void) apply
{
  CGContextRef context = UIGraphicsGetCurrentContext();

  // 设置变换
  CGContextConcatCTM(context, transform_);
}

@end
```

在apply方法中，我们用Quartz 2D函数UIGraphicsGetCurrentContext()取得CGContextRef引用。这里不打算讨论Quartz 2D绘图的细节。拿到了有效的当前绘图上下文之后，把transform_值传给CGContextConcatCTM()以将其加到上下文。之后在上下文中无论画什么，都会用传入的CGAffineTransform值作变换。现在仿射变换滤镜就做好了。

16

跟ImageTransformFilter一样，ImageShadowFilter也是ImageFilter的直接子类，也只重载了apply方法。在代码清单16-7中显示的方法中，我们取得当前图形上下文，然后安排一个Quartz 2D函数调用，CGContextSetShadow()，以向上下文添加阴影效果。然后剩下的处理就跟ImageTransformFilter的差不多了。之后在上下文中不管画什么，都会有阴影效果，像图16-6中右边的图像那样。

代码清单16-7　ImageShadowFilter.m

```
#import "ImageShadowFilter.h"

@implementation ImageShadowFilter

- (void) apply
{
  CGContextRef context = UIGraphicsGetCurrentContext();

  // 设置阴影效果
  CGSize offset = CGSizeMake (-25, 15);
  CGContextSetShadow(context, offset, 20.0);
}

@end
```

现在我们写好了全部滤镜，可以继续写客户端代码了。本章的示例项目中，有一个叫DecoratorViewController的类，它会运行在其viewDidLoad方法中定义的所有滤镜，如代码清单16-8所示。

代码清单16-8　DecoratorViewController.m中的viewDidLoad方法

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  // 加载原始图像
  UIImage *image = [UIImage imageNamed:@"Image.png"];
```

```
    // 创建一个变换
    CGAffineTransform rotateTransform = CGAffineTransformMakeRotation(-M_PI / 4.0);
    CGAffineTransform translateTransform = CGAffineTransformMakeTranslation(
                                                -image.size.width / 2.0,
                                                 image.size.height / 8.0);
    CGAffineTransform finalTransform = CGAffineTransformConcat(rotateTransform,
                                                    translateTransform);

    // 真正子类的方式
    id<ImageComponent>transformedImage=[[[ImageTransformFilter alloc]
                                          initWithImageComponent:image
                                          transform:finalTransform]
                                          autorelease];
    id<ImageComponent>finalImage=[[[ImageShadowFilter alloc]
                                   initWithImageComponent:transformedImage]
                                   autorelease];

    // 用滤镜处理过的图像
    // 创建新的DecoratorView
    DecoratorView *decoratorView = [[[DecoratorView alloc]
                                      initWithFrame:[self.view bounds]]
                                      autorelease];
    [decoratorView setImage:finalImage];
    [self.view addSubview:decoratorView];
}
```

我们首先创建一个对原始蝴蝶图像（图16-6中左边的图像）的引用，然后构造一个CGAffineTransform来相应地旋转和平移这个图像。图像和变换都用于将ImageTransformFilter的实例初始化为图像的第一个滤镜。然后使用整个组件来构造一个ImageShadowFilter的实例，向从ImageTransformFilter得到的图像添加阴影效果。此时，finalImage是链表的头节点，链表中包含ImageTransformFilter、ImageShadowFilter和原始图像。然后，在把DecoratorView的实例作为子视图添加到控制器之前，先把整个组件赋给它。DecoratorView所做的就是在其drawRect:rect方法中绘制图像，如代码清单16-9所示。

代码清单16-9 DecoratorView.m中的drawRect:rect方法

```
- (void)drawRect:(CGRect)rect
{
  // 绘图代码
  [image_ drawInRect:rect];
}
```

DecoratorView用image_保持一个UIImage的目标引用。drawRect:rect方法使用rect参数向image_转发drawInRect:rect消息。然后一系列的装饰操作就开始了。ImageShadowFilter将首先收到消息，在当前图像上下文设置阴影效果，然后从forwardingTargetForSelector:方法返回内嵌的component_。在下一步，同样的方法会被转发给返回的component_。此时的component_，实际上就是在前面构造链条时的ImageTransformFilter的实例。它也被同样的forwardingTargetForSelector:方法截获，并用预先定义的CGAffineTransform值对

当前上下文进行设置。然后它跟ImageShadowFilter一样，再次返回内嵌的component_。但这次，是原始的蝴蝶图像，因此当它被ImageTransformFilter返回，并在最后一步中处理消息的时候，将在应用了阴影效果和仿射变换的当前上下文中进行绘图。这就是得到图16-6中变换后并加了阴影的图像的整个操作顺序。

说明： 滤镜可以用不同顺序进行连接。
可以使用范畴来生成同样的东西，但是需要一些小技巧。16.4.2 节将讲解具体怎么做。

16.4.2 通过范畴实现装饰

在使用范畴的方式中，只需向UIImage类添加滤镜，构成范畴，它们虽然不是单独的UIImage类，却能够像单独的UIImage类那样使用。这就是Objective-C中范畴的好处。我们要加两个滤镜，一个对图像应用二维变换，另一个向图像添加阴影效果。图16-7为显示它们关系的类图。

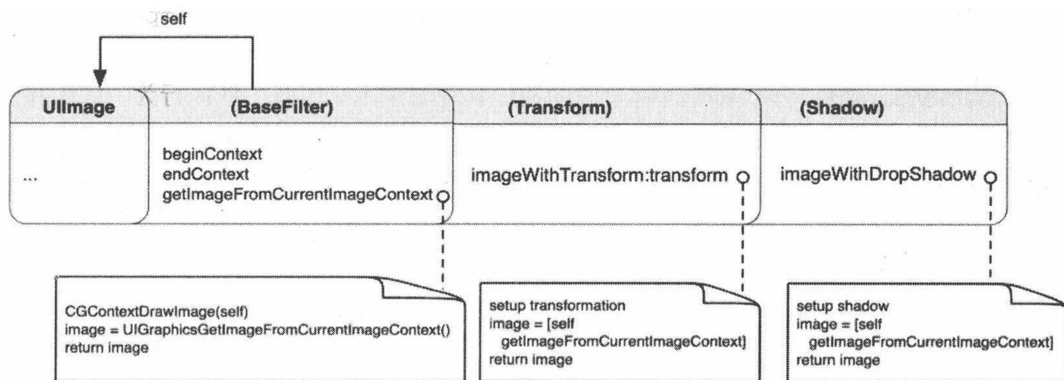


图16-7 作为UIImage私有范畴的各种图像滤镜的类图

像16.4.1节一样，我们要实现变换和阴影滤镜。这一方式中有3个范畴：UIImage (BaseFilter)、UIImage (Transform)和UIImage (Shadow)。从现在起，我把它们分别简称为BaseFilter、Transform和Shadow。BaseFilter定义了几个基本的二维绘图操作，使用当前绘图上下文绘制自己，跟16.4.1节中的抽象类ImageFilter类似。其他滤镜范畴可以使用同样的方法来绘制保持的任何图像引用。Transform和Shadow没有继承BaseFilter，但它们属于同一类，因为它们都是UIImage的范畴。BaseFilter中定义的方法也能在Transform和Shadow范畴中使用，而不必像子类化那样进行继承。Transform范畴定义了一个imageWithTransform:transform方法，接受一个转换引用（稍后将详细讲解），然后把它应用于内部的图像引用，并让它把自己画出来，然后返回变换后的图像。Shadow范畴定义了一个imageWithDropShadow方法，向内部的图像引用添加阴影效果，并返回应用效果之后的最终图像。你可能已经注意到，它们也可以像16.4.1节中讲解的真正子类化方式那样连接起来。图16-8中的对象图

说明了这一点。

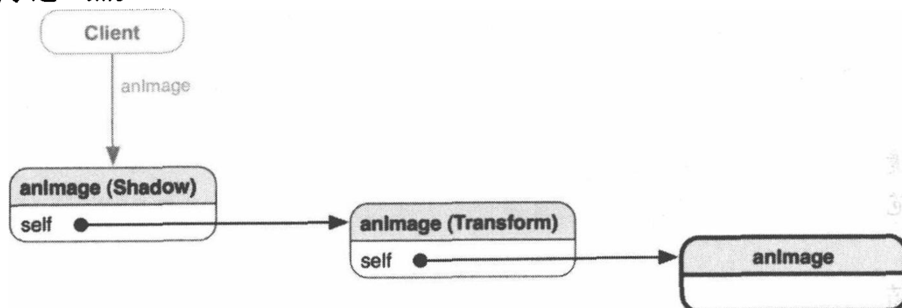


图16-8 显示运行时各种滤镜范畴如何引用其他UIImage实例的对象图

链表的右端是原始图像，就是图16-6中左边的图像。把原始图像加到Shadow滤镜，再加到Transform滤镜之后，客户端所引用的图像对象就是图16-6中右边的图像了。链表结构与真正子类的版本非常相似，只是每个范畴使用self来引用底层的图像，而不是使用component_之类的单独引用。

写点儿代码吧。先定义BaseFilter，它为其他具体滤镜定义了几个默认行为，如代码清单16-10所示。

代码清单16-10 UIImage+BaseFilter.h

```
@interface UIImage (BaseFilter)

- (CGContextRef) beginContext;
- (UIImage *) getImageFromCurrentImageContext;
- (void) endContext;

@end
```

BaseFilter有3个方法，用于在当前上下文中绘制自己，如代码清单16-11所示。

代码清单16-11 UIImage+BaseFilter.m

```
#import "UIImage+BaseFilter.h"

@implementation UIImage (BaseFilter)

- (CGContextRef) beginContext
{
  // 以目标尺寸创建一个图形上下文
  // 在iOS 4和其后的版本，使用UIGraphicsBeginImageContextWithOptions
  // 来处理缩放
  // 在iSO4以前的版本，使用以前的UIGraphicsBeginImageContext
  CGSize size = [self size];
  if (NULL != UIGraphicsBeginImageContextWithOptions)
    UIGraphicsBeginImageContextWithOptions(size, NO, 0);
  else
```

```
    UIGraphicsBeginImageContext(size);

  CGContextRef context = UIGraphicsGetCurrentContext();

  return context;
}

- (UIImage *) getImageFromCurrentImageContext
{
  [self drawAtPoint:CGPointZero];

  // 从当前上下文取得UIImage
  UIImage *imageOut = UIGraphicsGetImageFromCurrentImageContext();

  return imageOut;
}

- (void) endContext
{
  UIGraphicsEndImageContext();
}

@end
```

beginContext跟真正子类版本中的几乎一样。它为在当前上下文中绘制自己进行所有必要的准备工作。绘图用的上下文准备好了之后，这个方法就把它返回给调用者。

getImageFromCurrentImageContext用这个上下文绘制自己，并调用UIGraphicsGet-ImageFromCurrentImageContext()，从上下文中取出图像并将其返回。

然后，endContext消息通过Quartz 2D函数调用UIGraphicsEndImageContext()结束这个过程，释放与上下文相关的资源。

现在可以开始真正的滤镜范畴了。先来看Transform范畴。Transform只有一个方法，它接受一个CGAffineTransform结构体参数，并把它应用到图像。它的声明在代码清单16-12中。

代码清单16-12 UIImage+Transform.h

```
@interface UIImage (Transform)

- (UIImage *) imageWithTransform:(CGAffineTransform)transform;

@end
```

它的实现非常简单，如代码清单16-13所示。

代码清单16-13 UIImage+Transform.m

```
#import "UIImage+Transform.h"
#import "UIImage+BaseFilter.h"

@implementation UIImage (Transform)

- (UIImage *) imageWithTransform:(CGAffineTransform)transform
```

16

```
{
  CGContextRef context = [self beginContext];
  // 设置变换
  CGContextConcatCTM(context, transform);

  // 向上下文绘制原始图像
  UIImage *imageOut = [self getImageFromCurrentImageContext];

  [self endContext];

  return imageOut;
}

@end
```

它接受一个带有仿射变换矩阵信息的CGAffineTransform结构体参数。这个方法把transform和一个上下文传给Quartz 2D函数CGContextConcatCTM(context, transform)。然后transform被加到当前绘图上下文。现在它向self发送getImageFromCurrentImageContext消息，把自己绘制在屏幕上[①]。这个消息定义在BaseFilter范畴中。UIImage的实例从这个消息返回之后，它向自己发一个endContext消息，关闭当前绘图上下文，最后返回这个图像。

这样，Transform滤镜就写好了。很容易，不是吗？现在可以来定义Shadow滤镜了，这跟Transform一样简单，如代码清单16-14所示。

代码清单16-14 UIImage+Shadow.h

```
@interface UIImage (Shadow)

- (UIImage *) imageWithDropShadow;

@end
```

跟Transform滤镜一样，Shadow是UIImage的一个范畴，而且只有一个方法。方法没有参数，但它的实现比Transform滤镜多了几个步骤。在代码清单16-15中，可以看到如何向图像添加阴影效果。

代码清单16-15 UIImage+Shadow.m

```
#import "UIImage+Shadow.h"
#import "UIImage+BaseFilter.h"

@implementation UIImage (Shadow)

- (UIImage *) imageWithDropShadow
{
  CGContextRef context = [self beginContext];

  // 设置阴影
  CGSize offset = CGSizeMake (-25, 15);
```

① 实际上这里使用的是基于位图的上下文，并不会绘制到屏幕上。——译者注

```
  CGContextSetShadow(context, offset, 20.0);

  // 向上下文绘制原始图像
  UIImage * imageOut = [self getImageFromCurrentImageContext];

  [self endContext];

  return imageOut;
}

@end
```

我们首先使用Quartz 2D函数调用`CGSizeMake (-25, 15)`构造阴影的几个属性，这两个参数表示x与y方向的偏移量。然后把图形上下文传给`CGContextSetShadow(context, offset, 20.0)`。这也是个Quartz 2D函数，浮点型参数`20.0`表示模糊因子。最后，跟`Transform`范畴中的`addTransform:`方法类似，它把自己画到屏幕上[①]，从中取回`UIImage`，然后将其返回。

至此，我们就完成了对`UIImage`的实例进行图像滤镜处理所需的一切。将如何使用它们呢？我们同样把它们放到`DecoratorViewController`的`viewDidLoad`方法中，如代码清单16-16所示。

代码清单16-16　DecoratorViewController.m中的`viewDidLoad`方法

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  // 加载原始图像
  UIImage *image = [UIImage imageNamed:@"Image.png"];

  // 创建一个变换
  CGAffineTransform rotateTransform = CGAffineTransformMakeRotation(-M_PI / 4.0);
  CGAffineTransform translateTransform = CGAffineTransformMakeTranslation(
                                                    -image.size.width / 2.0,
                                                     image.size.height / 8.0);
  CGAffineTransform finalTransform = CGAffineTransformConcat(rotateTransform,
                                                        translateTransform);

  // 使用范畴的方式
  // 添加变换
  UIImage *transformedImage = [image imageWithTransform:finalTransform];

  // 添加阴影
  id <ImageComponent> finalImage = [transformedImage imageWithDropShadow];

  // 用滤镜处理过的图像
  // 创建新的DecoratorView
  DecoratorView *decoratorView = [[[DecoratorView alloc]
                                   initWithFrame:[self.view bounds]]
                                   autorelease];
```

① 这里使用的也是基于位图的上下文，不会绘制到屏幕上。——译者注

```
    [decoratorView setImage:finalImage];
    [self.view addSubview:decoratorView];
}
```

原始图像引用和变换设置等跟代码清单16-8中的真正子类版本相同。仅有的区别是，向图像应用变换的imageWithTransform:方法由图像自己来执行，并且它返回变换后的图像（原始图像未受影响）。然后变换后的图像执行imageWithDropShadow，为自己添加阴影，然后用新的图像返回自身的加了阴影效果的版本，即finalImage。finalImage被加到imageView，并显示到屏幕上，跟真正子类的方式相同。把所有滤镜和原始图像放在一起可以写成一行：

```
finalImage = [[image imageWithTransform:finalTransform] imageWithDropShadow];
```

现在，你能说出范畴和真正子类化方式的区别吗？是的，UIImage的滤镜在范畴方式中是实例方法，而在真正子类方式中是子类。在范畴方式中没有继承，因为所有滤镜仍是UIImage的一部分。真正子类方式中，我们把ImageComponent用作抽象类型，而在范畴方式中可以从头到尾使用UIImage。然而，跟真正子类方式一样，范畴方式中的滤镜也可以按不同顺序应用。

对UIImage对象实现同样的图像滤镜处理，范畴版本的设计看起来更为简单。这是因为我们使用了范畴，而没有为了扩展装饰器链而实际地子类化并封装另一个UIImage对象。下面将会讨论使用范畴来实现这个模式的几个优缺点。

Objective-C范畴与装饰模式

范畴是一个Objective-C的语言功能，通过它可以向类添加行为（方法的接口与实现），而不必进行子类化。通过范畴添加的方法对类原有的方法没有不良影响。范畴中的方法成为了类的一部分，并可由其子类继承。

正如前面的例子那样，我们可以使用范畴来实现装饰模式。然而，这并不是一种严格的实现，它实现了模式的意图，但却是一种变体。由装饰器范畴添加的行为是编译时绑定的，虽然Objective-C原本支持动态绑定（应该用方法的哪个实现）。而且装饰器范畴实际上没有封装被扩展的类的实例。

尽管使用范畴来实现这个模式跟原始风格有些偏离，但是实现少量的装饰器的时候，它比真正子类方式更为轻量、更为容易。虽然前面例子中UIImage的范畴没有严格封装另一个组件，但是在UIImage中被扩展的实例被通过self来间接引用。

16.5　总结

本章介绍了装饰模式的概念和Objective-C中不同的实现方式。真正子类方式的实现使用一种较为结构化的方式连接各种装饰器。范畴的方式更为简单和轻量，适用于现有类只需要少量装饰器的应用程序。虽然范畴不同于实际的子类化，不能严格实现模式的原始风格，但它实现了解决同样问题的意图。设计图像滤镜处理示例程序那样的应用时，装饰模式是自然而然的选择。图像滤镜的任何组合都能动态应用或删除，而不影响UIImage原有行为的完整性。

下一章将讨论一个与装饰模式类似的设计模式，但它实现不同的目的，它是责任链模式。

第 17 章

责 任 链

17

谁也不能无所不知，俗话说“人多智广”。如果此话不假，那么把很多人的智慧连接成一个链条会更好。每个人都有自己的专长，联合起来就能形成强大的实体。这很像邻里间的互相帮助，或者项目的团队成员之间的合作。智慧链条中的每个单元都可以为问题的解决作出贡献。如果一个人不知道如何解决问题，他就把问题沿着链条向下传，也许就有人能够解决问题。有的时候，即使有人知道如何解决，依然会把问题传下去。这样就能完成解决问题的特定过程。这类似于装配线，每个工人都知道如何为传送带上未完成的产品安装特定的零件。但产品需要在装配线上沿着工人的“链条”传递，直到最后一名工人完成装配，产品才能出厂。生产线上的每一名工人都有自己的专长和职责。因此责任链是这样形成的：“这个问题我不懂，也许你懂”，或者“我的活儿干完了，现在该你了”。这种“智慧”或“责任”的链条，允许进一步的升级或扩展，而不必修改已有单元的功能。只要把新的单元添加到链中，链就会有更多的功能。这就像为了向产品增加零件，在生产线的末端增加了额外的工人。向生产线增加工人的过程，不应影响原有的工人。

这种概念对于面向对象的软件设计同样适用。比如，让一组对象处理特定的请求，而对这个组添加或删除处理程序（handler）都不应影响组的完整性。

本章将讨论责任链（Chain of Responsibility）模式的概念，以及如何应用这一模式，设计角色扮演游戏（RPG）中人物的防御机制。

17.1 何为责任链模式

责任链模式的主要思想是，对象引用了同一类型的另一个对象，形成一条链。链中的每个对象实现了同样的方法，处理对链中第一个对象发起的同一个请求。如果一个对象不知道如何处理请求，它就把请求传给下一个响应器（即successor）。处理请求的对象与其successor之间的关系如图17-1所示。

`Handler`是上层抽象类，定义了一个方法——`handleRequest`，处理它知道如何处理的请求对象。`ConcreteHandler1`和`ConcreteHandler2`实现了`handleRequest`方法，来处理它们认识的请求对象。`Handler`也有一个指向另一个同类型实例的引用，即`successor`。当调用`Handler`实例的`handleRequest`消息时，如果这个实例不知道如何处理请求，它会用同样的消

息把请求转发给successor。如果successor可以处理，就行了；否则，它就把请求传给下一个successor（如果有的话）。这个过程会一直进行下去，直到请求被传到链中的最后一个Handler。图17-2中的对象图表示了Handler的实例用successor形成链的方式。

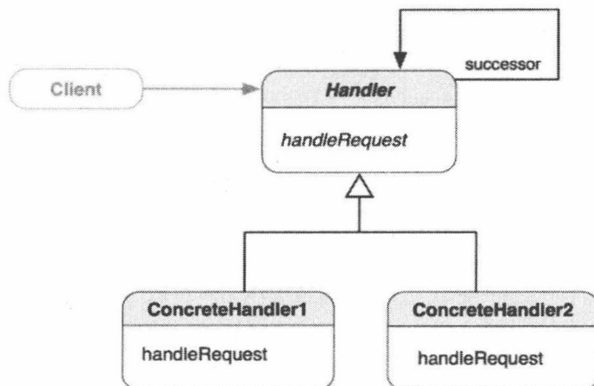


图17-1 责任链模式的类图

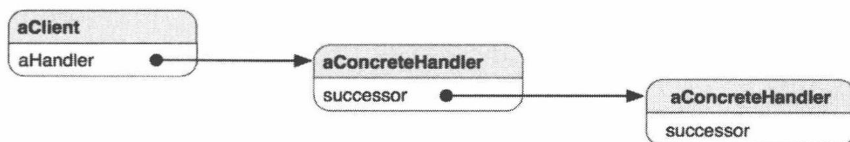


图17-2 运行时的请求处理程序链的一种典型结构

aClient有一个对Handler实例的引用，叫aHandler。aHandler是处理程序链的第一个对象，即aConcreteHandler。aConcreteHandler用它内部的successor引用跟另一个Handler实例连接起来。用这种策略处理请求的最低要求是，如果不懂如何处理请求，就传给下一个处理程序（有些实现中，无论处理程序是否能够处理请求，都要求处理程序把请求传递下去）。

> **责任链模式**：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间发生耦合。此模式将这些对象连成一条链，并沿着这条链传递请求，直到有一个对象处理它为止。*
>
> * 最初的定义出现于《设计模式》（Addison-Wesley，1994）。

17.2 何时使用责任链模式

在以下情形，自然会考虑使用这一模式：

- 有多个对象可以处理请求，而处理程序只有在运行时才能确定；
- 向一组对象发出请求，而不想显式指定处理请求的特定处理程序。

下面几节将讨论如何应用责任链模式，为RGP游戏中的人物实现各种防御道具。

17.3　在 RPG 游戏中使用责任链模式

假定我们要开发一个RPG游戏，里面的每个人物能够通过赚取点数来升级防御道具。防御道具可以是盾牌或盔甲。每种形式的防御只能应付一种特定的攻击。如果防御道具不认识一种进攻，它就把进攻的作用传给下一个会响应它的“实体”。图17-3是说明这一概念的示意图。

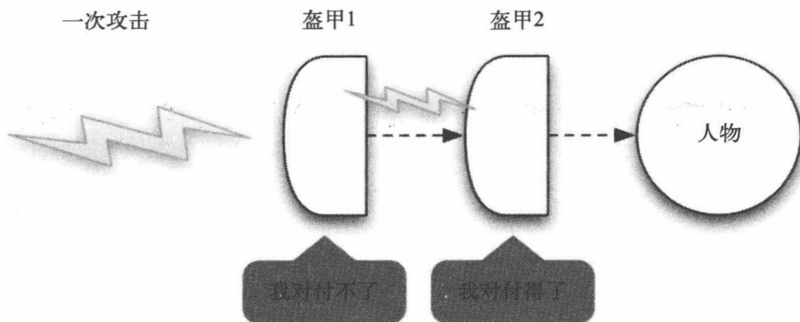


图17-3　人物的链式防御层的直观表示。虽然盔甲1不能对付攻击，盔甲2可以吸收这次攻击，因而人物安然无恙

在图17-3中，盔甲1不知道如何对付对手的攻击，所以把它传给下一个盔甲，盔甲2。盔甲2刚好知道如何对付这次攻击，化解了人物可能遭受的损伤。由于某种原因，如果没有盔甲可以对这次攻击作出响应，攻击的作用最终会传到人物。人物对攻击作出响应时，会表现为一定程度的损伤。这个场景如图17-4所示。

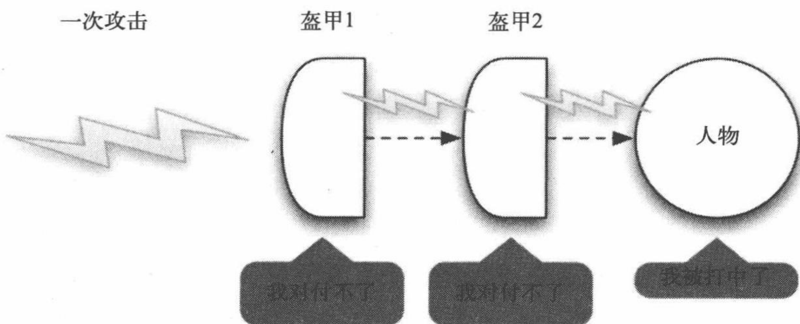


图17-4　跟图17-3相同的直观表示，只是这回两个盔甲都不能对付攻击，人物受到损伤

这种只让每个独立的防御道具对特定类型的攻击作出响应的机制，简化了人物使用各种防御道具的复杂性。每种盔甲和盾牌各自负责非常特定的功能。这就是责任链模式的作用所在。

我们将讨论如何使用此模式实现这个设计。假设要实现两种防御：金属盔甲和水晶盾牌。它们都只能按照设计对付某些攻击。金属盔甲可以防御剑的攻击，而水晶盾牌可以对付任何魔法火焰的攻击。如前面的两个图所示，人物也是响应链的一部分，因此它也应该跟其他防御道具有共同的行为，对攻击作出响应。它们的静态关系如图17-5中的类图所示。

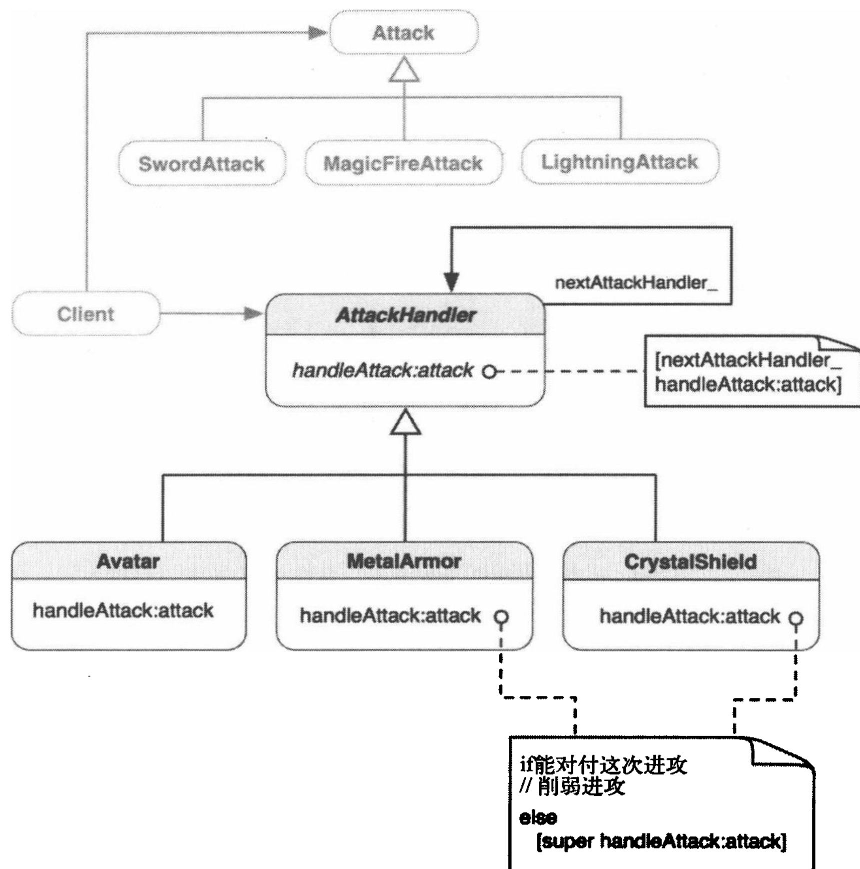


图17-5　由一个攻击处理程序链构成的一组AttackHandler的类图

Avatar、MetalArmor和CrystalShield是AttackHandler的子类。AttackHandler定义了一个方法——handleAttack:attack，该方法的默认行为是，把攻击传给另一个AttackHandler的引用，即成员变量nextAttackHandler_。子类重载这个方法，对攻击提供实际的响应。如果AttackHandler不知道如何响应一个攻击，那么就使用[super handle Attack:attack]消息，把它转发给super，这样super中的默认实现就会把攻击沿着链传下去。

类图中定义了3种类型的攻击：SwordAttack、MagicFireAttack和LightingAttack。稍后我们将讨论哪种AttackHandler能响应哪种攻击。但是首先，先来看看AttackHandler父类的代码，请看代码清单17-1。

代码清单17-1　AttackHandler.h

```
#import "Attack.h"

@interface AttackHandler : NSObject
{
```

```
  @private
  AttackHandler *nextAttackHandler_;
}

@property (nonatomic, retain) AttackHandler *nextAttackHandler;

- (void) handleAttack:(Attack *)attack;

@end
```

AttackHandler定义了一个同类型的私有变量nextAttackHandler_，它是攻击的下一个响应者。AttackHandler的子类应该重载handleAttack:方法，以响应它能够识别的一种攻击。抽象的AttackHandler为这个方法定义了默认行为，如代码清单17-2所示。

代码清单17-2 AttackHandler.m

```
#import "AttackHandler.h"

@implementation AttackHandler

@synthesize nextAttackHandler=nextAttackHandler_;

- (void) handleAttack:(Attack *)attack
{
  [nextAttackHandler_ handleAttack:attack];
}

@end
```

如果子类没有重载这个方法，默认的handleAttack:实现就会被调用。这个方法只是把攻击传给nextAttackHandler_去处理。

我们来看看Avatar的第一个防具MetalArmor。MetalArmor子类化AttackHandler并重载其handleAttack:方法，如代码清单17-3所示。

代码清单17-3 MetalArmor.h

```
#import "AttackHandler.h"

@interface MetalArmor : AttackHandler
{

}

// 重载的方法
- (void) handleAttack:(Attack *)attack;

@end
```

跟在其他章节中一样，在子类的头文件中再次声明重载的方法不是必需的，但是这样做更加清楚。MetalArmor只能识别SwordAttack的实例。如果攻击确实是SwordAttack类型，那么handleAttack:将用NSLog输出一条@"No damage from a sword attack!"消息。否则，它输出

另一条消息并使用[super handleAttack:attack]把攻击转给super，如代码清单17-4所示。

代码清单17-4　MetalArmor.m

```
#import "MetalArmor.h"
#import "SwordAttack.h"

@implementation MetalArmor

- (void) handleAttack:(Attack *)attack
{
  if ([attack isKindOfClass:[SwordAttack class]])
  {
    // 攻击没有通过这个盔甲
    NSLog(@"%@", @"No damage from a sword attack!");
  }
  else
  {
    NSLog(@"I don't know this attack: %@", [attack class]);
    [super handleAttack:attack];
  }
}

@end
```

类似地，CrystalShield也是一种AttackHandler类型。除了类名外，它的类声明几乎跟MetalArmor一样，这里就不重复了。CrystalShield的实现跟MetalArmor也很相似，它只识别一种攻击——MagicFireAttack。除此之外，它把不认识的攻击，通过同样的[super handleAttack:attack]消息传给super，让下一个AttackHandler处理它，如代码清单17-5所示。

代码清单17-5　CrystalShield.m

```
#import "CrystalShield.h"
#import "MagicFireAttack.h"

@implementation CrystalShield

- (void) handleAttack:(Attack *)attack
{
  if ([attack isKindOfClass:[MagicFireAttack class]])
  {
    // 攻击没有通过这个盾牌
    NSLog(@"%@", @"No damage from a magic fire attack!");
  }
  else
  {
    NSLog(@"I don't know this attack: %@", [attack class]);
    [super handleAttack:attack];
  }
}

@end
```

如果没有防具能够对付攻击，攻击最终将传给Avatar。Avatar也是AttackHandler的子类，而且与MetalArmor和CrystalShield有相同的响应Attack的机制。但是，攻击到达这里的时候，Avatar将没有防御而受到损伤。它重载的handleAttack:方法通过NSLog(@"Oh! I'm hit with a %@!", [attack class])，输出攻击的名称，如代码清单17-6所示。

代码清单17-6 Avatar.m

```
#import "Avatar.h"

@implementation Avatar

- (void) handleAttack:(Attack *)attack
{
  // 当攻击到达这里时，我就被击中了。
  // 实际损伤的点数取决于攻击的类型。
  NSLog(@"Oh! I'm hit with a %@!", [attack class]);
}

@end
```


现在我们定义好了所有的AttackHandler。再来看看客户端代码，看看这些AttackHandler是如何连接起来形成防御的响应链的（见代码清单17-7）。

代码清单17-7 管理人物和各种攻击的客户端代码

```
// 创建新的人物
AttackHandler *avatar = [[[Avatar alloc] init] autorelease];

// 让它穿上金属盔甲
AttackHandler *metalArmoredAvatar = [[[MetalArmor alloc] init] autorelease];
[metalArmoredAvatar setNextAttackHandler:avatar];

// 然后给金属盔甲中的人物增加一个水晶盾牌
AttackHandler *superAvatar = [[[CrystalShield alloc] init] autorelease];
[superAvatar setNextAttackHandler:metalArmor];

// ……其他行动

// 用剑攻击人物
Attack *swordAttack = [[[SwordAttack alloc] init] autorelease];
[superAvatar handleAttack:swordAttack];

// 然后用魔法火焰攻击人物
Attack *magicFireAttack = [[[MagicFireAttack alloc] init] autorelease];
[superAvatar handleAttack:magicFireAttack];

// 现在用闪电进行新的攻击……
Attack *lightningAttack = [[[LightningAttack alloc] init] autorelease];
[superAvatar handleAttack:lightningAttack];

// ……进一步的行动
```

这个攻击处理程序有点像栈（即先进后出）。因为需要让Avatar是攻击的最后一站，所以它要最先创建。然后创建MetalArmor的实例，把Avatar作为它的下一个AttackHandler。它们被当做“增强”了的Avatar，MetalArmor是它通往真正Avatar实例的第一道门。仅有MetalArmor还不够，还需要创建CrystalShield的实例，作为Avatar的另一种防御。我们使用MetalArmor形式的AttackHandler（与Avatar连接在一起），作为CrystalShield实例的下一个攻击处理程序。此时，Avatar已是一个具有两种防御的“超级人物”。

在游戏中的某个时刻，我们创建了3种类型的攻击——swordAttack、magicFireAttack和lightningAttack，让“超级人物”用它的handleAttack:方法去处理。以下是来自责任链中各种AttackHandler的输出，通过这些输出可以了解发生了什么，请看代码清单17-8。

代码清单17-8　客户端代码的输出

```
I don't know this attack: SwordAttack
No damage from a sword attack!
No damage from a magic fire attack!
I don't know this attack: LightningAttack
I don't know this attack: LightningAttack
Oh! I'm hit with a LightningAttack!
```

金属盔甲为人物挡住了剑的攻击，因为有水晶盾牌，魔法火焰攻击也没有伤到人物。但是第三次的闪电攻击，盔甲和盾牌都不知道如何应付，而是打出了消息：I don't know this attack: LightningAttack（我不认识这个攻击：闪电攻击）。最后，攻击由人物自己来处理，打出了消息Oh! I'm hit with a LightningAttack!（啊！我被闪电攻击击中了！），表示因闪电攻击而受到损伤。

这个简单的RPG游戏的例子演示了如何使用责任链模式，来简化人物处理各种攻击的编码和逻辑。如果不用这个模式，防御逻辑很可能都塞到一个类中（比如Avatar），代码会乱作一团。

17.4　总结

本章的例子中，我们把RPG游戏中人物的各种防御机制实现为责任链模式。每种防御机制只能应付一种特定的攻击。一个攻击处理程序链决定了人物可以防御何种攻击。在游戏过程中，任何攻击处理程序都能在任何时间被添加或删除，而不会影响人物的其他行为。对于此类设计，责任链模式是很自然的选择。否则，攻击处理程序的复杂组合会让人物的代码非常庞大，让处理程序的变更非常困难。

已经讨论过的这几个模式，都是在扩展对象的行为的同时，对对象进行最少的修改甚至不作修改。接下来将讨论如何通过封装和扩展对象的算法来改变对象的行为。

Part 7

第七部分

算法封装

本部分内容

- 第 18 章　模板方法
- 第 19 章　策略
- 第 20 章　命令

第 18 章 模板方法 18

烹饪中有些特定的步骤，可以一般化为通用工序，如准备食材、烹饪食材和上餐。实际食谱的每个步骤，都可以在其范围变化。对于某种特定的食品，烹饪步骤甚至可以进一步一般化。例如，不管你是做哪种三明治，都有些一般步骤。每种三明治可以在面包、肉和调味料的选用上有所不同。有些三明治可能与普通三明治有些不同，增加了一些步骤，但制作中仍然基本上依照一般步骤。

制作三明治的一组一般化步骤（食谱）是一个模板方法（template method）。说它是个模板是因为它在完成工序的步骤中仍然缺少某些特定的片段。这些缺少的片段会在需要制作各种三明治的时候填补上去。制作鲁宾三明治的时候，在“准备面包”步骤中需要准备几片黑麦面包。汉堡包在同一步骤里需要汉堡圆面包。不管是什么三明治，总是需要“面包”才能完成制作过程。我们可以把这些一般化的步骤组合成一个操作，叫做“制作三明治”。各种三明治食谱用独特的工序和配料填补这些空缺的步骤。

本章将讨论模板方法模式的概念，还会在通过“食谱”模板制作各种三明治的一个例子中，实现这一模式。

18.1 何为模板方法模式

模板方法模式是面向对象软件设计中一种非常简单的设计模式。其基本思想是在抽象类的一个方法中定义“标准”算法。在这个方法中调用的基本操作应由子类重载予以实现。这个方法被称为“模板”，因为方法定义的算法缺少一些特有的操作。图18-1显示了抽象类与具体子类之间的关系——抽象类定义模板，子类重载基本操作以提供独特操作供模板方法调用。

`AbstractClass`不完整地定义了一些方法与算法，留出一些操作未作定义。`AbstractClass`调用的`templateMethod`时，方法中未定义的空白部分，由`ConcreteClass`重载`primitiveOperation1`（或2）来填补。

> **模板方法模式**：定义一个操作中算法的骨架，而将一些步骤延迟到子类中。模板方法使子类可以重定义算法的某些特定步骤而不改变该算法的结构。*
>
> *最初的定义出现于《设计模式》（Addison-Wesley，1994）。

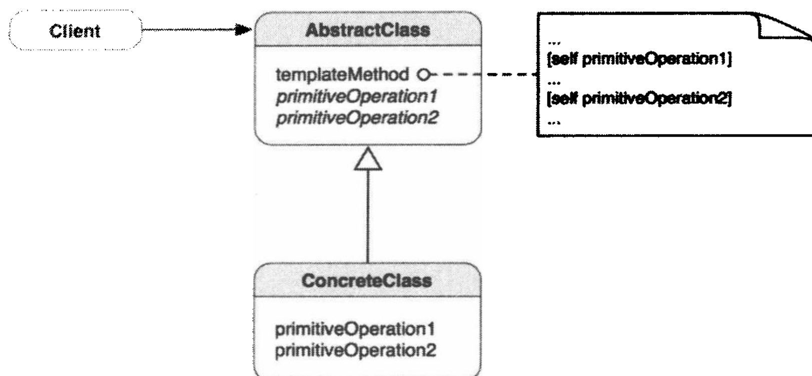


图18-1 模板方法模式的类图。ConcreteClass重载AbstractClass的primitiveOperation1和primitiveOperation2，以在Client调用AbstractClass中的templateMethod时提供独特的操作

18.2 何时使用模板方法

在以下情形，应该考虑使用模板方法。

- 需要一次性实现算法的不变部分，并将可变的行为留给子类来实现。
- 子类的共同行为应该被提取出来放到公共类中，以避免代码重复。现有代码的差别应该被分离为新的操作。然后用一个调用这些新操作的模板方法来替换这些不同的代码。
- 需要控制子类的扩展。可以定义一个在特定点调用“钩子”（hook）操作的模板方法。子类可以通过对钩子操作的实现在这些点扩展功能。

钩子操作给出了默认行为，子类可对其扩展。默认行为通常什么都不做。子类可以重载这个方法，为模板算法提供附加的操作。

模板方法模式中的控制结构流程是倒转的，因为父类的模板方法调用其子类的操作，而不是子类调用父类的操作。这与“好莱坞原则”类似：别给我们打电话，我们会打给你。

模板方法会调用5种类型的操作：

- 对具体类或客户端类的具体操作；
- 对抽象类的具体操作；
- 抽象操作；
- 工厂方法（见第4章）；
- 钩子操作（可选的抽象操作）。

18.3 利用模板方法制作三明治

来看看怎么做三明治吧，看看做三明治跟模板方法模式有什么关系。

我想，简单的三明治谁都会做。那么，至少要有哪些东西，才能叫（非素食的）三明治呢？

- 面包。
- 肉。
- 调味料。

做简单三明治的配料准备好了。基本步骤应该像下面这样：

(1) 准备面包；

(2) 把面包放在盘子上；

(3) 往面包上加肉；

(4) 加调味料；

(5) 上餐。

可以定义一个叫make的模板方法，它调用上述各个步骤来制作真正的三明治。制作真正三明治的默认算法有些特定的操作没有实现，所以模板方法只是定义了制作三明治的一般方式。当具体的三明治子类重载了三明治的行为之后，客户端仅用make消息就能制作真正的三明治了。AnySandwich和ConcreteSandwich之间的静态关系如图18-2中的类图所示。

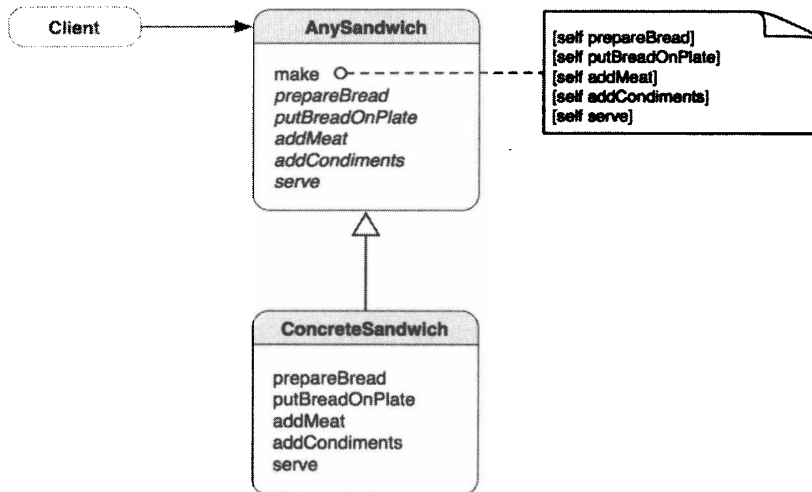


图18-2　AnySandwich和ConcreteSandwich的类图。ConcreteSandwich重载模板方法make所调用的抽象操作

如果为AnySandwich写一个Objective-C类，就会是代码清单18-1这样。

代码清单18-1　AnySandwich.h

```
@interface AnySandwich : NSObject
{
}

- (void) make;

// 制作三明治的步骤
```

```
- (void) prepareBread;
- (void) putBreadOnPlate;
- (void) addMeat;
- (void) addCondiments;
- (void) serve;

@end
```

跟在图18-2中的类图里看到的一样，调用make制作三明治时，它会调用一般步骤，如代码清单18-2所示。

代码清单18-2 AnySandwich.m

```
#import "AnySandwich.h"

@implementation AnySandwich

- (void) make
{
  [self prepareBread];
  [self putBreadOnPlate];
  [self addMeat];
  [self addCondiments];
  [self serve];
}

- (void) putBreadOnPlate
{
  // 做任何三明治都要先把面包放在盘子上
}

- (void) serve
{
  // 任何三明治做好了都要上餐
}

#pragma mark -
#pragma Details will be handled by subclasses

- (void) prepareBread
{
  // 要保证子类重载这个方法
}

- (void) addMeat
{
  // 要保证子类重载这个方法
}

- (void) addCondiments
{
  // 要保证子类重载这个方法
}

@end
```

prepareBread、addMeat和addCondiments方法需要由子类重载，以提供真正三明治的细节。需要一种能确保子类实现这些方法的机制，让真正的三明治制作过程有意义。相关问题将在18.4节中进行讨论。

现在我们要做一个真正的三明治——鲁宾三明治。既然鲁宾三明治也是一种三明治，其制作步骤应该与其他三明治相同，尽管它有自己的配料和一些特有的小操作。鲁宾三明治的Objective-C类应该像代码清单18-3这样。

代码清单18-3　ReubenSandwich.h

```
#import "AnySandwich.h"

@interface ReubenSandwich : AnySandwich
{
}

- (void) prepareBread;
- (void) addMeat;
- (void) addCondiments;

// 鲁宾三明治的特有操作
- (void) cutRyeBread;
- (void) addCornBeef;
- (void) addSauerkraut;
- (void) addThousandIslandDressing;
- (void) addSwissCheese;

@end
```

ReubenSandwich是AnySandwich的子类。制作鲁宾三明治有其特有的步骤和配料。鲁宾三明治的面包需要黑麦面包，肉需要腌牛肉，还需要德国酸菜，调味料需要千岛酱和瑞士奶酪。虽然奶酪不能算调味料，但这么做可以简化制作三明治的一般步骤，因为不是所有三明治都有奶酪。我们把鲁宾三明治的操作放入一般三明治的方法中，如代码清单18-4所示。

代码清单18-4　ReubenSandwich.m

```
#import "ReubenSandwich.h"

@implementation ReubenSandwich

- (void) prepareBread
{
  [self cutRyeBread];
}

- (void) addMeat
{
  [self addCornBeef];
}

- (void) addCondiments
{
```

```
    [self addSauerkraut];
    [self addThousandIslandDressing];
    [self addSwissCheese];
}

#pragma mark -
#pragma mark ReubenSandwich Specific Methods

- (void) cutRyeBread
{
  // 鲁宾三明治需要两片黑麦面包
}

- (void) addCornBeef
{
  // …… 加大量腌牛肉
}

- (void) addSauerkraut
{
  // …… 还有德国酸菜
}

- (void) addThousandIslandDressing
{
  // …… 别忘了千岛酱
}
- (void) addSwissCheese
{
  // …… 还有上等瑞士奶酪
}

@end
```


这个ReubenSandwich不用管putBreadOnPlate和serve。通用的AnySandwich已经定义了这些行为，其他具体三明治可以共享。ReubenSandwich所要关心是，面包、肉和调味料。

prepareBread方法调用ReubenSandwich的特有方法cutRyeBread，为三明治切黑麦面包片。如果三明治类是ReubenSandwich，那么prepareBread方法实际上准备的不是别的而是黑麦面包片。

几乎所有鲁宾三明治里都有腌牛肉。所以鲁宾三明治类有个addCornBeef方法，其实它就是AnySandwich抽象类中的addMeat步骤。当然，如果你想自己做个鲁宾三明治，你可以加不同种类的肉或者跟腌牛肉一起再放点儿别的肉。

鲁宾三明治还要关心的最后一步是addCondiments。我们知道这有很多花样，但还是按照普通做法吧。调味料呢，我们会加德国酸菜（addSauerkraut）、千岛酱（addThousandIsland-Dressing）还有瑞士奶酪（addSwissCheese）。所以addCondiments被调用时，Reuben-Sandwich会往肉上加德国酸菜、千岛酱和瑞士奶酪。

当最后一步serve完成之后，就可以享用这个鲁宾三明治了。其他种类的三明治呢？使用同

样的步骤能制作不同的三明治吗？答案是肯定的，绝对可以。做个汉堡包怎么样？请看代码清单18-5。

代码清单18-5　Hamburger.h

```
#import "AnySandwich.h"

@interface Hamburger : AnySandwich
{
}

- (void) prepareBread;
- (void) addMeat;
- (void) addCondiments;

// 汉堡包的特有方法
- (void) getBurgerBun;
- (void) addKetchup;
- (void) addMustard;
- (void) addBeefPatty;
- (void) addCheese;
- (void) addPickles;

@end
```

Hamburger也是AnySandwich的子类，它也有自己的制作细节。汉堡包的面包需要小圆面包，肉需要牛肉饼（除非你要奶酪汉堡），调味料需要番茄酱、芥末酱、腌黄瓜和奶酪。跟ReubenSandwich一样，出于同样的原因，我们把奶酪当做一种调味料。当我们把特有的汉堡包操作放入一般的三明治制作操作之中，代码就会如代码清单18-6这样。

代码清单18-6　Hamburger.m

```
#import "Hamburger.h"

@implementation Hamburger

- (void) prepareBread;
{
  [self getBurgerBun];
}

- (void) addMeat
{
  [self addBeefPatty];
}

- (void) addCondiments
{
  [self addKetchup];
  [self addMustard];
  [self addCheese];
  [self addPickles];
```

```
}

#pragma mark -
#pragma mark Hamburger Specific Methods

- (void) getBurgerBun
{
  // 汉堡包需要小圆面包
}

- (void) addKetchup
{
  // 先要放番茄酱，才能加其他材料
}

- (void) addMustard
{
  // 然后加点儿芥末酱
}

- (void) addBeefPatty
{
  // 汉堡包的主料是一片牛肉饼
}

- (void) addCheese
{
  // 假定汉堡包都有奶酪
}

- (void) addPickles
{
  // 最后加点儿腌黄瓜
}

@end
```

制作三明治的组合真是太多了。我们要做的只是创建自己的三明治类，提供prepareBread、addMeat和addCondiments的特定实现。然后调用AnySandwich（父类）实例的make方法，标准步骤就得以实施，直至三明治做好可以上餐为止。

AnySandwich父类中的“标准步骤”代表了制作三明治的算法。算法实现于make方法之中，并预留出一些操作，由子类做具体的实现。所以子类其实不必了解算法的细节，同时父类也不必了解子类所提供的具体操作细节。

模板方法与委托的比较

模板方法和委托模式（也叫适配器模式，见第8章）常见于Cocoa Touch框架。它们对框架类设计来说是非常自然的选择。为什么呢？用户应用程序可以复用（或扩展）框架类，而且框架类在设计时不会知道什么样的类会使用它们。可是对于特定的软件设计问题应该使用哪一个模式呢？以下是简要的指导方针。

18

模板方法	委托（适配器）
父类定义一个一般算法，但缺少某些特定/可选的信息或算法，它通过这些缺少的信息或算法起到一个算法“食谱”的作用	委托（适配器）与预先定义好的委托接口一起定义一个特定算法
缺少的信息由子类通过继承来提供	特定算法由任何对象通过对象组合来提供

18.4 保证模板方法正常工作

Objective-C的方法没有任何属性，可以要求子类必须进行重载。那么怎样保证一般的三明治制作方法中预留的方法被ReubenSandwich和Hamburger重载呢？

我们知道，如果三明治不执行prepareBread、addMeat和addCondiments步骤，就不是三明治了（根据我们最初对三明治作的假设）。这就是说，这些步骤如果少了任何一个，我们的程序就毫无意义。听起来有点儿像发生了异常（比如，哎呦！addMeat方法没加肉）。这有很多方法可以实现。一个简单的做法是让方法返回BOOL值，默认值设为NO。这样make方法就可以检查每个方法的返回值，确保没有方法是空着的。但有个陷阱——若因某种原因这些方法被复用，而新的调用者没有进行同样的检查来保证算法的流程正确，那么算法就完蛋了。如果必须的基本方法没有被重载，何不抛出异常呢？抛出NSException的实例，保证在开发过程中问题会被尽早捕获。但是异常并不是要在用户使用应用时来捕获。当抛出异常时，会打出一个错误消息，其中包括方法和类的名字，还有问题的原因。万一在具体的AnySandwich子类中忘了重载这些抽象方法，这样的信息在以后的调试中将非常有用。在这些方法中添加了几条抛出异常的语句之后，它们就变成了代码清单18-7中的样子。

代码清单18-7　在每个预留的一般三明治制作方法中加入一个抛出NSException实例的语句

```
- (void) prepareBread
{
  [NSException raise:NSInternalInconsistencyException
              format:@"You must override%@in a subclass",NSStringFromSelector(_cmd)];
}

- (void) addMeat
{
  [NSException raise:NSInternalInconsistencyException
              format:@"You must override %@ in a subclass",NSStringFromSelector(_cmd)];
}

- (void) addCondiments
{
  [NSException raise:NSInternalInconsistencyException
              format:@"You must override %@ in a subclass", NSStringFromSelector(_cmd)];
}
```

prepareBread、addMeat和addCondiments方法的原始版本只是空操作，现在每个都添加了抛出NSException实例的语句。当客户端调用AnySandwich的make方法并到达AnySandwich

的prepareBread时，就会抛出异常。raise是NSException的类方法，其参数为异常的名称和消息。我们使用Cocoa Touch框架中定义的NSInternalInconsistencyException，表明在所调用的代码中遇到了意外情况。此处是否定义自己的异常名称，完全由你决定。方法的消息参数部分使用字符串格式来构造一个NSString。我们将_cmd作为消息的一部分。_cmd是Objective-C对象的属性，包含被调用或被转发的选择器名称。在Objective-C 2.0中，可以使用以下语句：

```
@throw [NSException exceptionWithName:... reason:... userInfo:...];
```

语句中的@throw关键字差不多起到同样的作用。但是，使用NSException的类方法raise要简洁一些。

如果prepareBread方法所抛出异常的调用栈中没有@catch语句块，则会在程序崩溃的日志中看到如下消息（简洁起见，省去了完整的调用栈输出）：

```
[Session started at 2010-08-01 18:16:59 -0700.]
2010-08-01 18:17:00.632 TemplateMethod[1315:207] *** Terminating app due to uncaught
exception 'NSInternalInconsistencyException', reason: 'You must override prepareBread
in a subclass'
```

18.5 向模板方法增加额外的步骤

好了，现在我们知道了如何在前面的三明治制作过程中实现模板方法模式。制作某些三明治时，要是在上餐之前能做点儿小加工肯定不错——比如，把鲁宾三明治稍微烤一下。但这最后的加工通常是可选的，就是说不是所有三明治都有额外的步骤。而且这额外步骤的默认实现是个空操作。必要时子类可以扩展的父类中的方法叫做“钩子”，默认的钩子方法通常什么也不做。

回到AndySandwich类。我们可以为所有子类提供一个可选的钩子方法extraStep，必要时可在三明治上餐之前进行最后加工。新版的AnySandwich类如代码清单18-8所示。

代码清单18-8 带有额外步骤的AnySandwich.h

```
@interface AnySandwich : NSObject
{
}

- (void) make;

// 制作三明治的步骤
- (void) prepareBread;
- (void) putBreadOnPlate;
- (void) addMeat;
- (void) addCondiments;
- (void) extraStep;
- (void) serve;

@end
```

在原先的make方法中，extraStep是倒数第二个操作，就在serve之前，如代码清单18-9所示。

代码清单18-9 带有extraStep的AnySandwich.m中的make方法

```
- (void) make
{
  [self prepareBread];
  [self putBreadOnPlate];
  [self addMeat];
  [self addCondiments];
  [self extraStep];
  [self serve];
}
```

上餐之前需要一两个附加步骤的三明治，可以重载extraStep方法。在make方法中，这个方法会作为serve方法的前一步被调用。extraStep是可选的，如果这个方法是空的，一般不会影响整个三明治制作过程。而且，它也不用引发异常来保证被子类重载。

我们知道鲁宾三明治稍微烤一下会更好吃，所以要向ReubenSandwich类增加一些实现来支持这个操作（见代码清单18-10）。

代码清单18-10 ReubenSandwich在头文件中重载了extraStep并增加了grillIt

```
#import "AnySandwich.h"

@interface ReubenSandwich : AnySandwich
{
}

- (void) prepareBread;
- (void) addMeat;
- (void) addCondiments;
- (void) extraStep;

// 鲁宾三明治的特有方法
- (void) cutRyeBread;
- (void) addCornBeef;
- (void) addSauerkraut;
- (void) addThousandIslandDressing;
- (void) addSwissCheese;
- (void) grillIt;

@end
```

把grillIt作为extraStep的实现如代码清单18-11所示。

代码清单18-11 ReubenSandwich把grillIt作为它的extraStep

```
#import "ReubenSandwich.h"

@implementation ReubenSandwich

- (void) prepareBread
{
  [self cutRyeBread];
}
```

```
- (void) addMeat
{
  [self addCornBeef];
}

- (void) addCondiments
{
  [self addSauerkraut];
  [self addThousandIslandDressing];
  [self addSwissCheese];
}

- (void) extraStep
{
  [self grillIt];
}

#pragma mark -
#pragma mark ReubenSandwich Specific Methods

- (void) cutRyeBread
{
  // 鲁宾三明治需要两片黑麦面包
}

- (void) addCornBeef
{
  // ……加大量腌牛肉
}

- (void) addSauerkraut
{
  // ……还有德国酸菜
}

- (void) addThousandIslandDressing
{
  // ……别忘了千岛酱
}
- (void) addSwissCheese
{
  // ……还有上等瑞士奶酪
}

- (void) grillIt
{
  // 最后要把它烤一下
}

@end
```


如果你做的鲁宾三明治还有额外的步骤，可以随意地添加到extraStep方法当中。这个方法被调用时，其中的所有操作也都会被调用。所以添加更多可选的额外步骤时，不用修改

AnySandwich。今后，再制作鲁宾三明治时，就会烤一下。而Hamburger不需要ReubenSandwich那样的额外步骤。AnySandwich中改动的算法对其他三明治毫无影响，因为extraStep是可选的。

说明： 一般来说，默认的“钩子”方法什么也不做，而且“钩子”方法对子类来说是可选的。

18.6 在 Cocoa Touch 框架中使用模板方法

在框架设计中，模板方法模式相当常见。模板方法是代码复用的基本技术。通过它，框架的设计师可以把算法中应用程序相关的元素留给应用程序去实现。模板方法是抽出共同行为放入框架类中的手段。这一方式有助于提高可扩展性与可复用性（使用同一个“食谱”），而维持各种类（框架类与用户类）之间的松耦合。Cocoa Touch框架也采用了模板方法模式。在框架中经常能看到这些框架类，虽然不如Delegation那么常见。本节将介绍框架中几个常见的模板方法。

18.6.1 UIView 类中的定制绘图

从iOS 2.0开始，应用程序可以通过重载UIView类中的以下方法，执行定制绘图：

```
- (void)drawRect:(CGRect)rect
```

这个方法的默认实现什么也不做。UIView的子类如果真的需要绘制自己的视图，就重载这个方法。所以这个方法是个“钩子”方法。

当需要改变屏幕上的视图时，这个方法会被调用。框架会处理所有底层的苦差事，以实现这一点。由UIView处理的绘图过程的部分会调用drawRect:。如果这个方法中有代码，那么代码也会被调用。子类可以使用Quartz 2D函数UIGraphicsGetCurrentContext来取得当前图形上下文，用于在框架中提供任何2D元素。

客户程序也可以通过调用以下UIView方法，手动激活绘图过程。

```
- (void)setNeedsDisplay
```

它通知UIView的实例重画屏幕上的整个矩形区域。也可以调用另一个实例方法，指定视图中特定矩形区域进行重画：

```
- (void)setNeedsDisplayInRect:(CGRect)invalidRect
```

invalidRect是接收器会标记为无效的矩形区域，就是说只有这个区域会被重画，其他地方不会被重画。

下面我们来看看实现了模板方法模式的另一个框架类。

18.6.2 Cocoa Touch 框架中的其他模板方法实现

我们无法介绍框架中的所有模板方法应用，但值得一提的是UIViewController。它定义了一些让用户程序处理设备不同方向的方法。下面是当方向发生改变时会被调用的消息的清单。

```
shouldAutorotateToInterfaceOrientation:
rotatingHeaderView
rotatingFooterView
willAnimateRotationToInterfaceOrientation:duration:
willRotateToInterfaceOrientation:duration:
willAnimateFirstHalfOfRotationToInterfaceOrientation:duration:
didAnimateFirstHalfOfRotationToInterfaceOrientation:
willAnimateSecondHalfOfRotationFromInterfaceOrientation:duration:
didRotateFromInterfaceOrientation:
```

这些是纯"钩子"方法，需要重载以实现真正的动作。每个方法会在设备的方法发生改变的特定时刻被调用。子类可以选择某些方法予以重载，以增加屏幕选择过程中不同步骤和时间的特定行为。

18.7 总结

模板方法模式是代码复用的一项基本技术。模板方法在框架类设计中非常重要，因为它是抽出共同行为放入框架类中的手段。

模板方法用在Cocoa Touch框架的很多地方。18.6节中只讨论了其中几个。读者在iOS开发者文档中可以找到更多。

下一章将讨论另一种把对象中的算法封装成各种策略的方式。

第19章 策略

还记不记得这样的情景：把一堆算法塞到同一段代码中，然后使用if-else或switch-case条件语句来决定要使用哪个算法？这些算法可能是一堆相似的类函数或方法，用以解决相关的问题。例如，我有一个验证输入数据的例程。数据本身可以是任何数据类型（如CGFLoat、NSString、NSInteger等）。每种数据类型需要不同的验证算法。如果能把每个算法封装成一个对象，那么就能消除根据数据类型决定使用什么算法的一堆if-else或switch-case语句。

面向对象软件设计中，我们可以把相关算法分离为不同的类，成为策略。与这种做法有关的一种设计模式称为策略模式。本章将讨论策略模式的概念和关键特征。本章稍后要设计几个数据验证类，把它们设计成不同策略并予以实现，用来验证UITextField对象的输入。

19.1 何为策略模式

策略模式中的一个关键角色是策略类，它为所有支持的或相关的算法声明了一个共同接口。另外，还有使用策略接口来实现相关算法的具体策略类。场景（context）类的对象配置有一个具体策略对象的实例，场景对象使用策略接口调用由具体策略类定义的算法。它们的静态关系如图19-1中的类图所示。

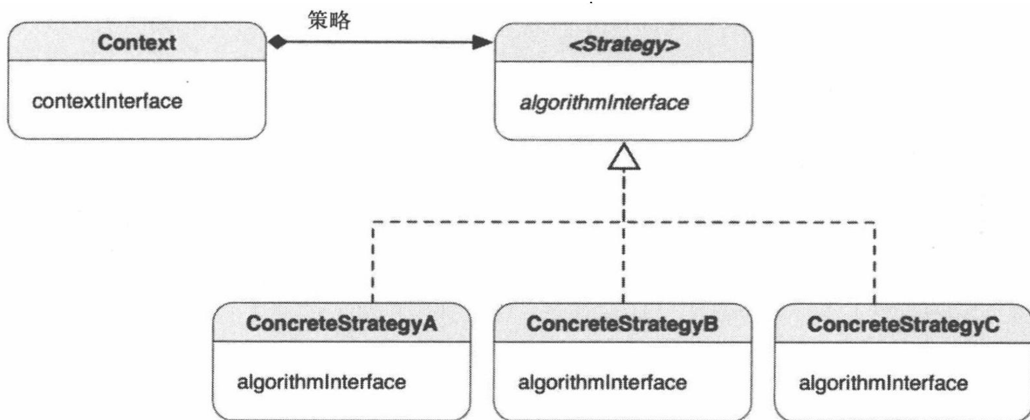


图19-1 策略模式的类结构

一组算法，或者说算法的一个层次结构，以ConcreteStrategy（A、B和C）类的形式，共享相同的algorithmInterface接口，这样Context就能使用相同的接口访问算法的各种变体。

> **策略模式**：定义一系列算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。*
>
> *最初的定义出现于《设计模式》(Addison-Wesley，1994)。

Context的实例可以在运行时用不同的ConcreteStrategy对象进行配置。这可以理解成更换Context对象的“内容”，因为变更是发生在对象的内部。装饰器（见装饰模式，第16章）改变对象的“外表”，因为修改是从外面叠加起来。有关两者区别的详细讨论，请看16.3节。

模型-视图-控制器中的策略模式

模型-视图-控制器模式中，控制器决定视图对模型数据进行显示的时机和内容。视图本身知道如何绘图，但需要控制器告诉它要显示的内容。同一个视图如果与不同的控制器合作，数据的输出格式可能一样，但数据的类型和格式可能随不同控制器的不同输出而不同。因此这种情况下的控制器是视图的策略。在前面的章节提到过，控制器与视图之间是一种基于策略模式的关系。

19.2　何时使用策略模式

在以下情形，自然会考虑使用这一模式。

- 一个类在其操作中使用多个条件语句来定义许多行为。我们可以把相关的条件分支移到它们自己的策略类中。
- 需要算法的各种变体。
- 需要避免把复杂的、与算法相关的数据结构暴露给客户端。

19.3　在 UITextField 中应用验证策略

我们用个简单的例子，在应用程序中实现策略模式。假设应用程序中需要有个UITextField以接受用户的输入，然后要在应用程序的处理中使用这个输入值。应用程序有个文字字段，只接受字母，即a~z或A~Z，还有个字段只接受数值型的值，即0~9。为了保证每个字段的输入有效，需要在用户结束文本框的编辑时作些验证。

可以把数据验证放到UITextField的委托方法textFieldDidEndEditing:之中。UITextField的实例每当失去焦点时会调用这个方法。这个方法中，可以检查数值型文本框的输入是否只有数值，字母字符型文本框的输入是否只有字母。这个委托方法提供当前文本框的一个输入引用（textField），但怎么才能知道这个文本框是数值型还是字母型的呢？

如果不用策略模式，代码会写成代码清单19-1这个样子。

代码清单19-1 在textFieldDidEndEditing委托方法中检查UITextField的内容的典型场景

```
- (void)textFieldDidEndEditing:(UITextField *)textField
{
  if (textField == numericTextField)
  {
    // 验证[textField text]，保证其值为数值型
  }
  else if (textField == alphaTextField)
  {
    // 验证[textField text]，保证其值只包含字母
  }
}
```

要是有更多不同类型的文本框，条件语句还会继续下去。如果能去掉这些条件语句，代码会更易管理，将来对代码的维护也会容易得多。

> **提示：**如果代码中有很多条件语句，就可能意味着需要把它们重构成各种策略对象。

现在的目标是把这些验证检查提到各种策略类中，这样它们就能在委托和其他方法之中重用。每个验证都从文本框取出输入值，然后根据所需的策略进行验证，最后返回一个BOOL值；如果验证失败，还会返回一个NSError实例。返回的NSError可以解释失败的原因。因为数值型和字母型输入检查有一定关联（有共同的输入输出类型），所以能抽出共同的接口。类的设计如图19-2中的类图所示。

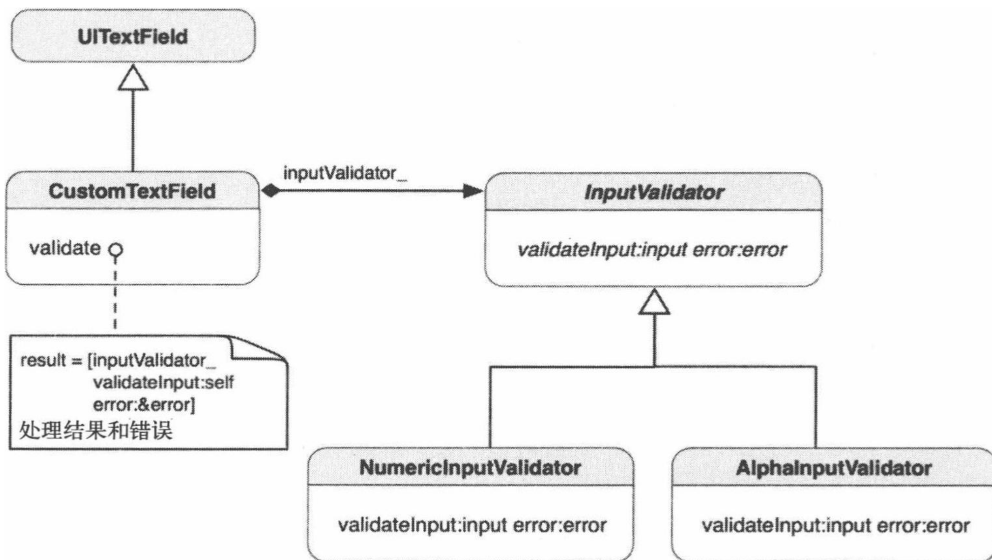


图19-2 显示CustomTextField及其相关InputValidator策略之间静态关系的类图

这里不把接口声明为协议，而是声明为抽象基类。抽象基类更适合解决这种问题，因为以后它更容易重构各种具体策略子类的某些共同行为。抽象基类InputValidator应该是代码清单

19-2中的这个样子。

代码清单19-2　InputValidator.h中抽象InputValidator的类声明

```
@interface InputValidator : NSObject
{
}

// 实际验证策略的存根方法
- (BOOL) validateInput:(UITextField *)input error:(NSError **) error;

@end
```

validateInput: error:方法把UITextField引用作为输入参数，这样就能验证文本框的内容，然后返回BOOL值，表示验证的结果。这个方法还有个参数是NSError指针的引用。当有错误发生时（即验证失败），方法会构造一个NSError实例，并赋值给这个指针，这样使用验证器的地方就能做详细的错误处理。

这个方法的默认实现只是把错误指针设为nil，并向调用者返回NO，如代码清单19-3所示。

代码清单19-3　InputValidator.m中抽象InputValidator的默认实现

```
#import "InputValidator.h"

@implementation InputValidator

// 实际验证策略的存根方法
- (BOOL) validateInput:(UITextField *)input error:(NSError **) error
{
  if (error)
  {
    *error = nil;
  }
  return NO;
}

@end
```

为什么不用NSString作为输入参数呢？这里如果只用NSString值，那么策略对象中的动作就是单向的。就是说，验证器就只能检查，然后返回结果，而不能修改原始值。使用UITextField型的输入参数，就可以两全其美。验证器可以选择修改文本框的原始值（比如删除无效的值），或者只检查而不修改。

另一个问题是，如果输入值错误，为什么不引发一个NSException异常呢？这是因为在Cocoa Touch框架中，引发自己的异常然后再用try-catch语句块来捕捉，这非常消耗资源，所以不推荐（但是用try-catch捕捉系统引发的异常又是另一回事）。返回NSError对象的开销相对较小，也是*Cocoa Touch Developer's Guide*中推荐的做法。阅读Cocoa Touch框架的文档时，会注意到有很多API都在异常发生时返回NSError的实例。常见的例子是NSFileManager的一个实例方法——(BOOL)moveItemAtPath:(NSString *)srcPath toPath:(NSString *)dstPath error:(NSError **)error。如果在MSFileManager尝试移动文件时发生错误，它会创建一

19

个NSError的实例，描述发生的问题。发起调用的方法，可以使用返回的NSError中包含的信息进行进一步的错误处理。所以这个存根方法输出NSError对象的目的，是提供有关错误情况的信息。

现在，我们定义了输入验证器的行为。然后我们要编写真正的输入验证器了。先来写数值型的，如代码清单19-4所示。

代码清单19-4　NumericInputValidator.h中NumericInputValidator的类定义

```
#import "InputValidator.h"

@interface NumericInputValidator : InputValidator
{
}

// 保证输入只包含数字（即0～9）的验证方法
- (BOOL) validateInput:(UITextField *)input error:(NSError **) error;

@end
```

NumericInputValidator子类化抽象基类InputValidator，并重载其validateInput:error:方法。这里重新声明了这个方法，以强调这个子类实现或重载了什么。这不是必需的，但这是个好习惯。

这个方法的实现如代码清单19-5所示。

代码清单19-5　NumericInputValidator.m中NumericInputValidator的实现

```
#import "NumericInputValidator.h"

@implementation NumericInputValidator

- (BOOL) validateInput:(UITextField *)input error:(NSError**) error
{
  NSError *regError = nil;
  NSRegularExpression *regex = [NSRegularExpression
                                 regularExpressionWithPattern:@"^[0-9]*$"
                                 options:NSRegularExpressionAnchorsMatchLines
                                 error:&regError];

  NSUInteger numberOfMatches = [regex
                               numberOfMatchesInString:[input text]
                               options:NSMatchingAnchored
                               range:NSMakeRange(0, [[input text] length])];

  // 如果没有匹配，就返回错误和NO
  if (numberOfMatches == 0)
  {
    if (error != nil)
    {
      NSString *description = NSLocalizedString(@"Input Validation Failed", @"");
      NSString *reason = NSLocalizedString(@"The input can contain only numerical
                                                                    values", @"");
```

```
    NSArray *objArray = [NSArray arrayWithObjects:description, reason, nil];
    NSArray *keyArray = [NSArray arrayWithObjects:NSLocalizedDescriptionKey,
                         NSLocalizedFailureReasonErrorKey, nil];

    NSDictionary *userInfo = [NSDictionary dictionaryWithObjects:objArray
                                                         forKeys:keyArray];
    *error = [NSError errorWithDomain:InputValidationErrorDomain
                                 code:1001
                             userInfo:userInfo];
   }
   return NO;
  }

  return YES;
}

@end
```

validateInput: error:方法主要做以下两件事情。

(1) 它使用配置好的NSRegularExpression对象，检查文本框中数值型值的匹配次数。我们使用的正则表达式是"^[0-9]*$"，意思是从行的开头（表示为^）到结尾（表示为$）应该有数字集（表示为[0-9]）中的0或更多个字符（表示为*）。

(2) 如果没有匹配，那么它就创建一个NSError对象，包含这样的消息：The input can contain only numerical values（只能输入数值），把它赋值给输入的NSError指针参数。最后它返回BOOL值，表示验证的结果。错误被关联到定制的错误代码1001和在InputValidator的头文件中定义的错误域：


```
static NSString * const InputValidationErrorDomain =
@"InputValidationErrorDomain";
```

NumericInputValidator的兄弟版本AlphaInputValidator，验证输入中是否只有字母，它用相似的算法验证输入的内容。AlphaInputValidator跟NumericInputValidator重载同样的方法。显然，它的验证算法要检查是否输入字符串只包含字母，如代码清单19-6中的实现文件所示。

代码清单19-6 AlphaInputValidator.m中AlphaInputValidator的实现

```
#import "AlphaInputValidator.h"

@implementation AlphaInputValidator

- (BOOL) validateInput:(UITextField *)input error:(NSError**) error
{
  NSError *regError = nil;
  NSRegularExpression *regex = [NSRegularExpression
                                regularExpressionWithPattern:@"^[a-zA-Z]*$"
                                options:NSRegularExpressionAnchorsMatchLines
                                error:&regError];

  NSUInteger numberOfMatches = [regex
```

```
                                        numberOfMatchesInString:[input text]
                                        options:NSMatchingAnchored
                                        range:NSMakeRange(0, [[input text] length])];

  // 如果没有匹配，就返回错误和NO
  if (numberOfMatches == 0)
  {
    if (error != nil)
    {
      NSString *description = NSLocalizedString(@"Input Validation Failed", @"");
      NSString *reason =NSLocalizedString(@"The input can contain only letters", @"");

      NSArray *objArray = [NSArray arrayWithObjects:description, reason, nil];
      NSArray *keyArray = [NSArray arrayWithObjects:NSLocalizedDescriptionKey,
                           NSLocalizedFailureReasonErrorKey, nil];

      NSDictionary *userInfo = [NSDictionary dictionaryWithObjects:objArray
                                                           forKeys:keyArray];
      *error = [NSError errorWithDomain:InputValidationErrorDomain
                                   code:1002
                               userInfo:userInfo];
    }

    return NO;
  }

  return YES;
}

@end
```

AlphaInputValidator也是实现了validateInput:方法的InputValidator类型。它的代码结构和算法跟它的兄弟版本NumericInputValidator相似，只是在NSRegularExpression对象中使用了不同的正则表达式，以及针对字母验证的不同错误代码和消息。检查字母的正则表达式为"^[a-zA-Z]*$"。这跟数值的验证很像，只是有效字符集包含大小写字母。可以看到，两个版本的代码有很多重复。两个算法结构相同，我们可以把这个结构重构成抽象父类中的模板方法（见第18章）。InputValidator的具体子类可以重载InputValidator中定义的基本操作，向模板算法返回特有的信息，例如，正则表达式和构建NSError对象的各种属性等。我把这个部分作为练习留给读者。

至此，我们写好了输入验证器，可以用在客户程序中了。但是，UITextField不认识它们，所以我们需要自己的UITextField版本。我们要创建UITextField的子类，其中有一个InputValidator的引用，以及一个方法——validate，如代码清单19-7所示。

代码清单19-7　CustomTextField.h中CustomTextField的类声明

```
#import "InputValidator.h"

@interface CustomTextField : UITextField
{
```

```
  @private
  InputValidator *inputValidator_;
}

@property (nonatomic, retain) IBOutlet InputValidator *inputValidator;

- (BOOL) validate;

@end
```

CustomTextField有一个属性保持着对InputValidator的引用。当调用它的validate方法时，它会使用这个InputValidator引用，开始进行实际的验证过程。来看看代码清单19-8的实现中是如何把它们结合在一起的。

代码清单19-8 CustomTextField.m中CustomTextField的实现

```
#import "CustomTextField.h"

@implementation CustomTextField

@synthesize inputValidator=inputValidator_;

- (BOOL) validate
{
  NSError *error = nil;
  BOOL validationResult = [inputValidator_ validateInput:self error:&error];

  if (!validationResult)
  {
    UIAlertView *alertView = [[UIAlertView alloc]
                                initWithTitle:[error localizedDescription]
                                      message:[error localizedFailureReason]
                                     delegate:nil
                            cancelButtonTitle:NSLocalizedString(@"OK", @"")
                            otherButtonTitles:nil];
    [alertView show];
    [alertView release];
  }

  return validationResult;
}

- (void) dealloc
{
  [inputValidator_ release];
  [super dealloc];
}

@end
```

validate方法向inputValidator_引用发送一条[inputValidator_ validateInput:self error:&error]消息。CustomTextField无需知道使用的是什么类型的InputValidator以及算法的任何细节，这就是这个模式的好处。因此在将来如果添加了新的InputValidator，

CustomTextField对象将会以同样的方式使用它。

所有准备工作都好了。我们假定客户端是一个UIViewController，它实现了UITextField-Delegate，并且有两个CustomTextField的IBOutlet，如代码清单19-9所示。

代码清单19-9　StrategyViewController.h中StrategyViewController的类声明

```
#import "NumericInputValidator.h"
#import "AlphaInputValidator.h"
#import "CustomTextField.h"

@interface StrategyViewController : UIViewController <UITextFieldDelegate>
{
  @private
  CustomTextField *numericTextField_;
  CustomTextField *alphaTextField_;
}

@property (nonatomic, retain) IBOutlet CustomTextField *numericTextField;
@property (nonatomic, retain) IBOutlet CustomTextField *alphaTextField;

@end
```

我们决定让控制器实现一个委托方法——(void)textFieldDidEndEditing:(UIText-Field *) textField，并把检查放到这个地方。每当文本框做了变更并失去了焦点，这个方法就会被调用。当用户输入结束时，CustomTextField会对其委托调用这个方法，如代码清单19-10所示。

代码清单19-10　textFieldDidEndEditing:方法中定义的客户代码，使用内部的策略对象InputValidator验证CustomTextField实例

```
@implementation StrategyViewController

@synthesize numericTextField, alphaTextField;

// ……
// 视图控制器中的其他方法
// ……

#pragma mark -
#pragma mark UITextFieldDelegate methods

- (void)textFieldDidEndEditing:(UITextField *)textField
{
  if ([textField isKindOfClass:[CustomTextField class]])
  {
    [(CustomTextField*)textField validate];
  }
}

@end
```

当一个文本框编辑结束，执行到textFieldDidEndEditing:时，它会检查textField是

否为CustomTextField类。如果是，它就向textField发一条validate消息，对文本输入框激活验证过程。可以看出，我们不再需要那些条件语句了。相反，我们使用一条简洁得多的语句，实现同样的数据验证。除了上面多了一条确保textField对象的类型是CustomField的额外检查之外，不应再有任何复杂的东西。

嘿，别急。有点儿不对头。怎样把正确的具体InputValidator实例赋给Strategy-ViewController中定义的numericTextField_和alphaTextField_呢？代码清单19-9中，两个*TextField都声明为IBOutlet型。我们可以在Interface Builder中通过IBOutlet把它们与视图控制器连接起来，就像连接其他按钮那样。类似地，在代码清单19-7中CustomTextField的类声明中，inputValidator属性也是个IBOutlet，就是说我们可以在Interface Builder中用同样的方式，把InputValidator的实例赋给*TextField。因此如果把类的某个属性声明为IBOutlet，一切都可以在Interface Builder中使用引用连接来实际构建。关于如何在Interface Builder中使用定制对象的详细讨论，请看11.3.3节。

19.4 总结

本章讨论了策略模式的概念以及如何应用这个模式让客户（场景）类使用相关算法的各种变体，为定制的UITextField实现输入验证器的例子，示范了各种验证器如何更换定制的UITextField的“内容”。策略模式与装饰模式（第16章）有些相似。装饰器从外部扩展对象的行为，而各种策略则被封装在对象之中。所以说装饰器改变对象的“外表”而策略改变对象的“内容”。

下一章将讨论与算法封装有关的另一种模式。封装的算法主要用于推迟命令对象的执行。

第20章 命令

在战场上，将军把封在信封里的定时指令交给部下，然后他们到时候打开信封并执行其中的指令。同样的指令既可以是一次性的，也可以是能被不同人在指定时间再次执行的。因为指令封在信封里，所以为了各种目的在不同地区之间传递起来会比其他方式（比如电话或其他通信线路上的口头指令）更为容易。

在面向对象设计中，我们借用了类似的思想，把指令封装在各种命令对象中。命令对象可以被传递并且在指定时刻被不同的客户端复用。从这一概念精心设计而来的设计模式叫做命令（Command）模式。

本章将讨论这一模式的概念，也将为TouchPainter应用程序开发一个撤销（undo）的基础设施，让用户可以撤销与恢复屏幕上所画的图形。Cocoa Touch对这一模式的改写将在本章稍后进行讨论。

20.1 何为命令模式

命令对象封装了如何对目标执行指令的信息，因此客户端或调用者不必了解目标的任何细节，却仍可以对它执行任何已有的操作。通过把请求封装成对象，客户端可以把它参数化并置入队列或日志中，也能够支持可撤销的操作。命令对象将一个或多个动作绑定到特定的接收器。命令模式消除了作为对象的动作和执行它的接收器之间的绑定。

在深入Objective-C中这个模式的细节之前，先来大略地看一下这个模式的结构（见图20-1）。

下面是图中所发生的一切：

- `Client`（客户端）创建`ConcreteCommand`对象并设定其receiver（接收器）；
- `Invoker`要求通用命令（实际上是`ConcreteCommand`）实施请求；
- `Command`是为`Invoker`所知的通用接口（协议）；
- `ConcreteCommand`起`Receiver`和对它的操作`action`之间的中间人的作用；
- `Receiver`可以是随着由`Command`（`ConcreteCommand`）对象实施的相应请求，而执行实际操作的任何对象。

> **命令模式**：将请求封装为一个对象，从而可用不同的请求对客户进行参数化，对请求排队或记录请求日志，以及支持可撤销的操作。*
>
> * 最初的定义出现于《设计模式》（Addison-Wesley，1994）。

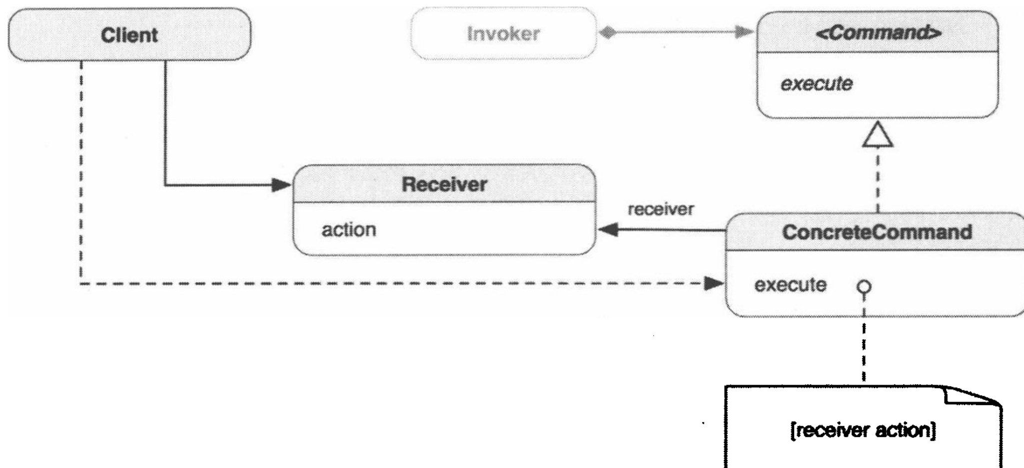


图20-1 展示命令模式结构的类图

20.2 何时使用命令模式

在下列情形，自然会考虑使用这一模式。

- 想让应用程序支持撤销与恢复。
- 想用对象参数化一个动作以执行操作，并用不同命令对象来代替回调函数。
- 想要在不同时刻对请求进行指定、排列和执行。
- 想记录修改日志，这样在系统故障时，这些修改可在后来重做一遍。
- 想让系统支持事务（transaction），事务封装了对数据的一系列修改。事务可以建模为命令对象。

你可能会问："为什么不能省掉这些麻烦，直接执行方法呢？"从技术上说，对，可以在程序中调用想要的任何方法或函数，程序仍能编译执行。问题是，每个类或方法都需要知道彼此的细节才能工作，而且要实现任何其他操作（比如撤销）将非常复杂。当程序变得越来越大时，对象的管理及复用会非常困难。

设计和实现撤销的基础设施之前，先来看看Cocoa Touch框架中的一些有用的资源，它们可用于下一节中基础设施的建立。

20.3 在 Cocoa Touch 框架中使用命令模式

与其重新发明轮子，不如复用Cocoa Touch框架中现成的东西，这是一种好习惯。命令模式是框架收录的模式之一。使用框架中现成的类，我们能专注于内容的开发。

`NSInvocation`、`NSUndoManager`和"目标–动作"机制是框架中对这个模式的典型应用。"目标-动作"机制在许多初级iOS编程书中都有介绍，因此这里只讲`NSInvocation`和`NSUndoManager`。

20

20.3.1 NSInvocation对象

NSInvocation类的实例跟图20-1中的原始ConcreteCommand类很相似。NSInvocation对象封装运行时库以向接收器转发执行消息所需的所有必要信息，如目标对象、方法选择器和方法的参数。因此，可以借助于NSInvocation实例，使用内部的选择器和其他信息，在任何时候调用接收器。同一个NSInvocation实例可重复调用接收器的同一个方法，或者通过不同的目标和方法签名进行复用。

NSInvocation是对调用的一个通用层面的抽象。苹果公司的框架设计师在设计NSInvocation类的基础设施时，当然对我们的实际接收器、调用器、选择器等一无所知。因此在创建NSInvocation对象时，需要以NSMethodSignature的形式，提供所有必要的信息。NSMethodSignature对象包含方法调用的所有参数和返回值类型。现在要使用NSInvocation代替原先的Command和ConcreteCommand类，对本章开头那个相当一般化的例子进行细化，如图20-2所示。

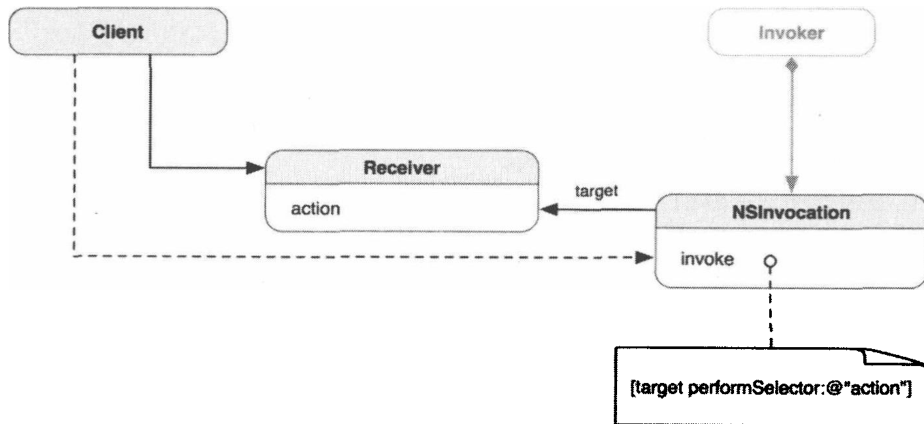


图20-2　使用NSInvocation类的命令模式的新类图

我们从第一个类图中拿掉原先的Command协议，并且用Cocoa Touch框架提供的NSInvocation类换掉ConcreteCommand类。这里的想法是，客户端用Receiver的实例创建NSInvocation对象，并把Receiver的操作用作选择器。然后将用这个NSInvocation对象对Invoker（如UIButton对象）进行设置。当Invoker需要调用动作时，它会调用Invoker中保存的NSInvocation对象的invoke方法。最后，NSInvocation对象会对其目标调用选择器，完成整个过程。

20.3.2 NSUndoManager

自从iOS 3.0面世以来，就可以使用Cocoa Touch框架中的NSUndoManager进行iOS应用开发了。NSUndoManager作为通用的撤销栈的管理类，设计得很漂亮，它强大而多功能，足以胜任几乎各种应用程序。这里我们要介绍NSUndoManager的主要功能和思想。

NSUndoManager的实例管理自己的撤销和恢复栈。撤销栈以对象形式保存所有已调用的操作。通过调用从撤销栈压入恢复栈的操作对象，NSUndoManager也能够“逆转”撤销操作（即恢复）。注册的撤销操作，可以是“逆转”客户端调用的上一个操作的动作。与自制撤销栈相比，NSUndoManager通过使用各种延迟调用转发机制，省去了处理撤销与恢复操作的麻烦。撤销管理器对象收集发生于一个运行循环（run loop）周期中的所有注册的撤销操作，让撤销操作可以逆转周期中的所有修改。如果请求NSUndoManager撤销上一次的操作，它就调用撤销栈顶部的操作对象。完成之后，它会弹出操作对象，把它压入恢复栈。类似地，当恢复完成后，它会把操作对象从恢复栈移到撤销栈，就可以再次撤销。因此NSUndoManager在两个栈之间移动操作对象，管理着整个命令历史记录。

注册撤销命令操作

一开始，撤销栈是空的。注册撤销操作，就会向撤销栈添加一个调用对象。要往撤销栈添加撤销操作，需要用执行撤销操作的对象注册它。使用NSUndoManager对象注册撤销操作有两种方式：

- 简单撤销注册；
- 基于调用的注册。

注册简单的撤销操作需要一个选择器，它标识使用接收器作为参数的撤销操作。在变更之前，状态被传给撤销管理器。在调用撤销操作时，会用先前的状态或属性重设接收器。

基于调用的撤销涉及一个NSInvocation对象。使用NSInvocation对象意味着可以使用一个有任意个数和类型的参数的方法。当状态变更需要多个参数时，这种方式非常有用。

多数情况下，应用程序中容纳或管理其他对象的客户端对象，有一个NSUndoManager实例。因为NSUndoManager不保持其调用目标接收器，所以对于被压入NSUndoManager的撤销栈的接收器，客户端对象需要保证它的引用计数至少为1。否则，在进行撤销与恢复时，如果接收器已被释放，撤销管理器可能会导致应用程序崩溃。有时，有这样一种情况，被修改的对象有自己的撤销管理器，自己执行撤销和恢复操作，例如，绘图应用有多个绘图视图，每个都管理所有线条及其颜色。当前绘图视图添加的每个线条都会向视图的撤销管理器注册一个新的撤销操作。如果对特定的视图请求撤销操作，相应的撤销管理器将会对接收器执行注册的撤销操作，删除最后一个线条。

20.4 在 TouchPainter 中实现撤销与恢复

如前面几节所述，命令模式的一个典型应用是支持应用程序的撤销与恢复操作。

我们要为TouchPainter应用程序设计并实现一个撤销与恢复的架构。在前面几节，已经讨论了使用NSInvocation把执行封装为命令对象，也探讨了把NSUndoManager的撤销架构借为我用的可能性。

本节将用两个小节介绍不同的实现方式。第一个小节将用NSUndoManager进行设计。之后会讨论如何从零开始构建自己的撤销和恢复。

20.4.1 使用 NSUndoManager 实现绘图与撤销绘图

在前面几节，我们已经了解了NSUndoManager的一些关键概念和功能。在下面的小节中，将介绍使用NSUndoManager管理可操作的NSInvocation对象，对主Scribble对象进行线条和点的绘制和撤销的过程。

1. 添加生成绘图与撤销绘图调用的方法

每次需要NSUndoManager注册撤销和恢复操作时，都需要一个新的NSInvocation对象。方便起见，我们添加一两个方法，生成NSInvocation对象的原型，我们就可以仅仅用某些Stroke和Dot对象修改几个参数，得到想要的NSInvocation对象。代码清单20-1是出于这一目的的两个方法。

代码清单20-1　生成用于绘图和撤销绘图的操作的NSInvocation对象的方法

```
- (NSInvocation *) drawScribbleInvocation
{
  NSMethodSignature*executeMethodSignature = [scribble_
                                              methodSignatureForSelector:
                                              @selector(addMark:
                                                        shouldAddToPreviousMark:)];
  NSInvocation *drawInvocation = [NSInvocation
                                  invocationWithMethodSignature:
                                  executeMethodSignature];
  [drawInvocation setTarget:scribble_];
  [drawInvocation setSelector:@selector(addMark:shouldAddToPreviousMark:)];
  BOOL attachToPreviousMark = NO;
  [drawInvocation setArgument:&attachToPreviousMark atIndex:3];

  return drawInvocation;
}

- (NSInvocation *) undrawScribbleInvocation
{
  NSMethodSignature *unexecuteMethodSignature = [scribble_
                                                 methodSignatureForSelector:
                                                 @selector(removeMark:)];
  NSInvocation *undrawInvocation = [NSInvocation
                                    invocationWithMethodSignature:
                                    unexecuteMethodSignature];
  [undrawInvocation setTarget:scribble_];
  [undrawInvocation setSelector:@selector(removeMark:)];

  return undrawInvocation;
}
```

drawScribbleInvocation方法生成一个NSInvocation对象，用于向scribble_添加Mark对象。它需要一个NSMethodSignature对象作为选择器，通过调用这个选择器实例化NSInvocation对象。在这里，我们需要Scribble对象的addMark:shouldAddToPreviousMark:方法的方法签名。创建了NSInvocation对象之后，把BOOL型的第二个参数（从0开始的第3个）

默认设为NO，因为我们只撤销和恢复整个线条和点，而不是顶点。在NSInvocation对象中收集的用户参数从序号2开始，因为第一个（序号0）是接收器，第二个是包含被调用选择器名字的_cmd。

类似地，undrawScribbleInvocation方法也以同样的方式创建一个NSInvocation对象，只是调用的选择器是Scribble对象的removeMark:。调用对象用于以后撤销整条线条或点。等讲到实际的绘图代码的时候，就能看到怎样把真正的Mark对象赋给由这些方法生成的调用对象。

2. 添加向NSUndoManager注册撤销与恢复的方法

我们介绍了如何生成用于线条与点的绘制与撤销的NSInvocation对象，但是还需要把它们注册到NSUndoManager，让这些调用能被撤销。代码清单20-2是向CanvasViewController的NSUndoManager注册撤销与恢复操作的方法的实现。

代码清单20-2　借助于NSUndoManager执行与撤销NSInvocation对象的方法

```
#pragma mark Draw Scribble Command Methods

- (void) executeInvocation:(NSInvocation *)invocation
        withUndoInvocation:(NSInvocation *)undoInvocation
{
  [invocation retainArguments];

  [[self.undoManager prepareWithInvocationTarget:self]
    unexecuteInvocation:undoInvocation
    withRedoInvocation:invocation];

  [invocation invoke];
}

- (void) unexecuteInvocation:(NSInvocation *)invocation
          withRedoInvocation:(NSInvocation *)redoInvocation
{
  [[self.undoManager prepareWithInvocationTarget:self]
    executeInvocation:redoInvocation
    withUndoInvocation:invocation];

  [invocation invoke];
}
```

executeInvocation: withUndoInvocation:和unexecuteInvocation: withRedoInvocation:两个方法看起来非常像，有点容易混淆。第一个方法接受调用对象的参数，直接执行并把另一个调用对象注册为撤销操作。后一个方法使用第一个参数的调用对象来执行撤销操作，并把第二个参数注册为恢复操作。

executeInvocation:方法中，我们向CanvasViewController的NSUndoManager发送prepareWithInvocationTarget:消息，以注册撤销操作。我们传入unexecuteInvocation:undoInvocation withRedoInvocation:invocation，作为撤销事件的完整调用消息。unexecuteInvocation:方法把executeInvocation:中的调用对象注册为恢复操作。如果仔细看，会注意到它们交叉使用一个绘图的调用对象和另一个撤销绘图的调用对象。

20

3. 为调用而修改触摸事件处理程序

现在要修改CanvasViewController中原来的触摸事件处理器，以创建调用对象，为撤销和恢复准备所有绘图动作。代码清单20-3显示了所需的修改。我们把它分成两个部分加以讨论。

代码清单20-3　对CanvasViewController中原来的触摸事件处理程序所作的修改，增加了NSUndoManager和基于NSInvocation的撤销和恢复操作

```
#pragma mark -
#pragma mark Touch Event Handlers

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
  startPoint_ = [[touches anyObject] locationInView:canvasView_];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
  CGPoint lastPoint = [[touches anyObject] previousLocationInView:canvasView_];

  // 如果这是手指的拖动，就向涂鸦添加一个线条
  if (CGPointEqualToPoint(lastPoint, startPoint_))
  {
    id <Mark> newStroke = [[[Stroke alloc] init] autorelease];
    [newStroke setColor:strokeColor_];
    [newStroke setSize:strokeSize_];

    [scribble_ addMark:newStroke shouldAddToPreviousMark:NO];

    // 取得用于绘图的NSInvocation,
    // 并为绘图命令设置新的参数
    NSInvocation *drawInvocation = [self drawScribbleInvocation];
    [drawInvocation setArgument:&newStroke atIndex:2];

    // 取得用于撤销绘图的NSInvocation,
    // 并为撤销绘图命令设置新的参数
    NSInvocation *undrawInvocation = [self undrawScribbleInvocation];
    [undrawInvocation setArgument:&newStroke atIndex:2];

    // 执行带有撤销命令的绘图命令
    [self executeInvocation:drawInvocation withUndoInvocation:undrawInvocation];
  }

  // 把当前触摸作为顶点添加到临时线条
  CGPoint thisPoint = [[touches anyObject] locationInView:canvasView_];
  Vertex *vertex = [[[Vertex alloc]
                      initWithLocation:thisPoint]
                    autorelease];

  // 由于不需要撤销每个顶点，所以保留这条语句
  [scribble_ addMark:vertex shouldAddToPreviousMark:YES];
}
```

这个代码清单中的触摸事件处理程序在12.5节中作过讨论，因此这里就不再重复其细节，但是会重点讨论触摸处理过程中的一些关键部分，以及为撤销和恢复所作的修改。

在touchesMoved:方法中，Mark对象的类型有两种情况。

- 如果上一次触摸是屏幕上的第一次触摸，那么会创建Stroke的实例并附加到scribble_，在根节点之下。
- 在方法中，其他的触摸会被当做线条的顶点，所以我们要创建Vertex的实例并附加到scribble_，在根节点的最后一个Mark子节点（即以前添加的Stroke）之下。

在Stroke对象开始的地方，我们删除了向scribble_添加新的Stroke对象的原始消息语句addMark:shouldAddToPreviousMark:。作为替代，我们使用先前定义的*scribbleInvocation方法（这里的*是通配符，不是指针记号），生成添加和删除新的Stroke对象的模板调用对象。

为drawInvocation和undrawInvocation对象设置了适当的参数之后，把它们传给executeInvocation:方法，开始执行绘图操作。这个方法调用drawInvocation，同时把undrawInvocation注册为撤销操作。

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
  CGPoint lastPoint = [[touches anyObject] previousLocationInView:canvasView_];
  CGPoint thisPoint = [[touches anyObject] locationInView:canvasView_];

  // 如果触摸从未移动（抬起之前一直在同一处）
  // 就向现有Stroke组合体添加一个点
  // 否则就把它作为最后一个顶点添加到临时线条
  if (CGPointEqualToPoint(lastPoint, thisPoint))
  {
    Dot *singleDot = [[[Dot alloc]
                       initWithLocation:thisPoint]
                      autorelease];
    [singleDot setColor:strokeColor_];
    [singleDot setSize:strokeSize_];

    [scribble_ addMark:singleDot shouldAddToPreviousMark:NO];

    // 取得用于绘图的NSInvocation,
    // 并为绘图命令设置新的参数
    NSInvocation *drawInvocation = [self drawScribbleInvocation];
    [drawInvocation setArgument:&singleDot atIndex:2];

    // 取得用于撤销绘图的NSInvocation,
    // 并为撤销绘图命令设置新的参数
    NSInvocation *undrawInvocation = [self undrawScribbleInvocation];
    [undrawInvocation setArgument:&singleDot atIndex:2];

    // 执行带有撤销命令的绘图命令
    [self executeInvocation:drawInvocation withUndoInvocation:undrawInvocation];
  }

  // 在此重置起点
  startPoint_ = CGPointZero;
```

```
    // 如果这是线条的最后一点
    // 就用不着画它
    // 因为用户看不出什么区别
  }

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
{
    // 在此重置起点
    startPoint_ = CGPointZero;
}
```

对于touchesEnded:方法中的画点处理，我们采用与绘制线条相同的步骤。我们使用同一个singleDot对象来执行executeInvocation:方法。

至此，我们借助于NSUndoManager实现了撤销与恢复的基础设施。下面几节将讨论如何从头开始构建自己的撤销和恢复。如果读者对在Objective-C中使用命令模式实现基本的撤销和恢复感兴趣，可以继续阅读下一节。否则，可以跳过下面这一节，从20.4.3节继续往下读。

20.4.2　自制绘图与撤销绘图的基础设施

祝贺你！你赶上了我们即将开始的构建自己的撤销和恢复架构。开始设计之前，先看看可以怎样收集应用程序中所有执行过的命令。执行过的命令对象可以收集起来形成一个命令历史记录的队列，如图20-3所示。

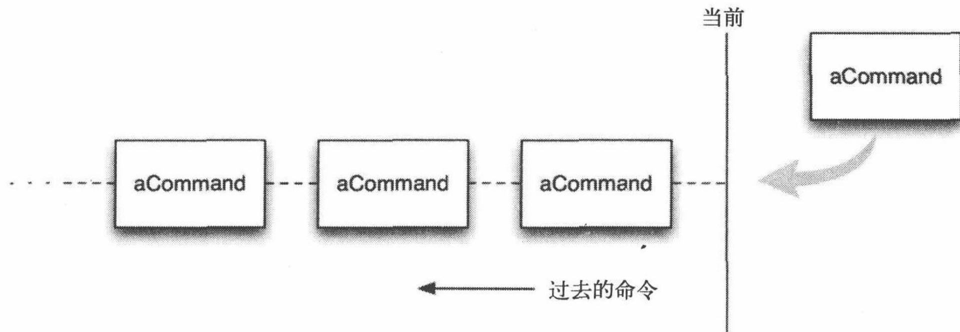


图20-3　执行过的命令的序列形成一个现行的命令历史记录

只要有新的命令对象被执行并加到队列中，队列就往一个方向增长。在传统意义上，我们可以通过遍历命令历史记录，用其中任何一个执行撤销或恢复操作。一个常见的方法是使用索引来保存队列中的当前命令对象。通过增减索引的值，我们可以移动到特定命令对象进行撤销或恢复。实际上，使用索引来移动然后进行撤销或恢复操作可能相当麻烦。有一种较为简单而不易出错的方式。我们可以使用两个栈，一个用于撤销，一个用于恢复，就像NSUndoManager那样，而不是用单一命令列表管理撤销和恢复。执行过的命令对象被压入撤销栈。栈顶的总是上一个执行过的命令。当应用程序需要撤销上一个操作的时候，它会从撤销栈弹出最后一个命令，然后执行撤销。完成撤销之后，应用程序会把这个命令压入恢复栈。所有命令对象都被撤销了之后，恢复栈

就填满了所有的恢复命令对象。对于恢复也是类似的过程，只是方向相反。图20-4是解释这种机制的一个示意图。

命令对象只是从一个栈弹出压入另一个栈，而不是使用任何复杂的索引方式在列表中移动。

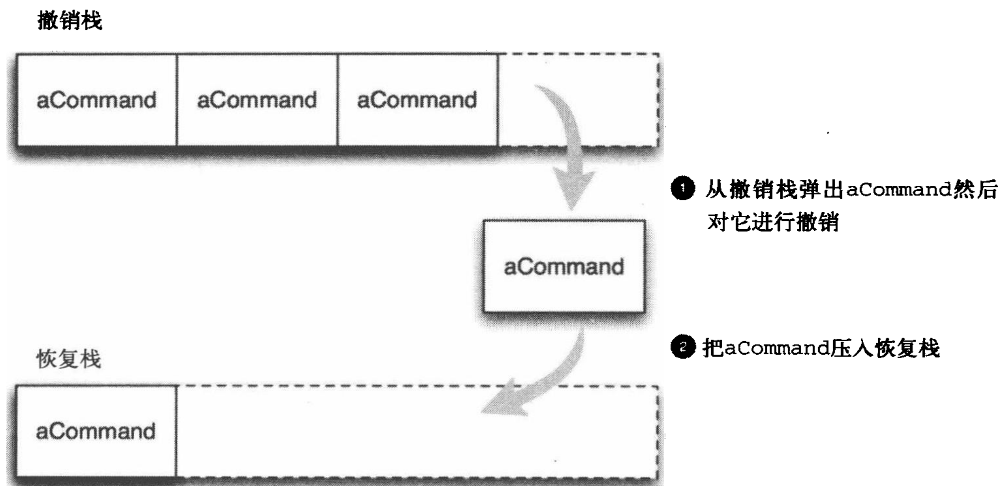


图20-4 在撤销操作中，把aCommand从撤销栈的顶部移到恢复栈

1. 命令基础设施的设计

我们要使用两个栈的方式给TouchPainter应用实现撤销和恢复操作。当用户触摸画布的时候，屏幕上就出现一个线条或者一个点。首先，需要把“绘图”动作作成一个命令对象。它绘制某个图形之后，应用程序会把它压入撤销栈，等以后需要时就能撤销它所画的图形。我们把这个绘图用的命令叫做DrawScribbleCommand，如图20-5中的类图所示。

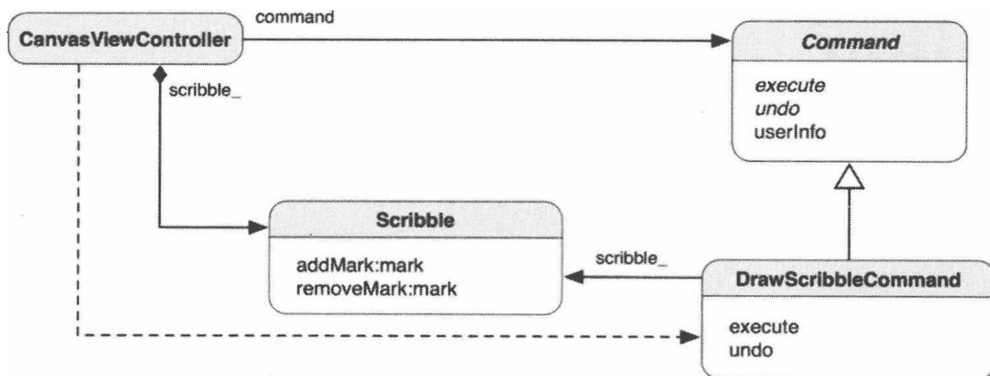


图20-5 DrawScribbleCommand和相关类的类图

Command是DrawScribbleCommand的抽象父类。它声明了抽象操作execute和undo。Command还有一个具体属性userInfo，客户端可通过它向Command对象提供附加的参数。

DrawScribbleCommand有一个Scribble对象的引用，可以向它添加或从它删除Mark对象。这里的CanvasViewController既是客户端又是调用者。整个类结构与图20-1中命令模式的原始类图非常相似。

2. 命令类的实现

那么，在Objective-C中如何实现它呢？请看代码清单20-4中的抽象Command类。

代码清单20-4　Command.h中Command的类声明

```
@interface Command : NSObject
{
  @protected
  NSDictionary *userInfo_;
}

@property (nonatomic, retain) NSDictionary *userInfo;

- (void) execute;
- (void) undo;

@end
```

除了execute和undo方法，Command类还声明了NSDictionary型的userInfo属性。Command的子类的对象可以使用userInfo中的信息，在重载的execute方法中执行任何操作。Command的实现如代码清单20-5所示。

代码清单20-5　Command.m中Command的实现

```
#import "Command.h"

@implementation Command
@synthesize userInfo=userInfo_;

- (void) execute
{
  // 应该抛出异常
}

- (void) undo
{
  // 什么也不做
  // 子类需要重载这个方法，执行实际的撤销
}

- (void) dealloc
{
  [userInfo_ release];
  [super dealloc];
}

@end
```

Command的实现中没做什么事情，因为execute和undo方法都是抽象操作。其子类应该在其中加入一些动作。现在来看看代码清单20-6中的DrawScribbleCommand。

代码清单20-6 DrawScribbleCommand.h中DrawScribbleCommand的类声明

```
#import "Command.h"
#import "Scribble.h"

@interface DrawScribbleCommand : Command
{
  @private
  Scribble *scribble_;
  id <Mark> mark_;
  BOOL shouldAddToPreviousMark_;
}

- (void) execute;
- (void) undo;

@end
```

它声明了几个私有成员变量，辅助将来对Scribble对象的操作的执行。子类中也重新声明了execute和undo方法，这样将来再来看这个类的时候就比较容易看懂。execute和undo方法的实现如代码清单20-7所示。

代码清单20-7 DrawScribbleCommand.m中DrawScribbleCommand的实现

```
#import "DrawScribbleCommand.h"

@implementation DrawScribbleCommand

- (void) execute
{
  if (!userInfo_) return;

  scribble_ = [userInfo_ objectForKey:ScribbleObjectUserInfoKey];
  mark_ = [userInfo_ objectForKey:MarkObjectUserInfoKey];
  shouldAddToPreviousMark_=[(NSNumber*)[userInfo_
                                      objectForKey:AddToPreviousMarkUserInfoKey]
                              boolValue];

  [scribble_ addMark:mark_ shouldAddToPreviousMark:shouldAddToPreviousMark_];
}

- (void) undo
{
  [scribble_ removeMark:mark_];
}

@end
```

重载的execute方法依赖userInfo属性中的信息。如果没有提供userInfo，方法就会退出。那么，究竟userInfo之中有什么对DrawScribbleCommand如此重要呢？userInfo字典含有3个关键元素，对于Scribble对象的使用至关重要。首先，是一个目标Scribble对象。没有它，其他的就都没用了。然后，是一个应该被加入到这个Scribble对象中的Mark实例。最后，是一

20

个BOOL值，表示这个Mark实例应该如何附加到Scribble对象。有关Scribble及其操作的详细讨论，见第12章。

undo方法只是告诉保存的Scribble对象删除保存的Mark引用。

3. 为命令修改CanvasViewController

现在我们知道了DrawScribbleCommand如何使用Scribble对象执行实际的绘图。我们来看看几个DrawScribbleCommand对象是如何参与CanvasViewController中定义的撤销和恢复操作的，请看代码清单20-8。

代码清单20-8 管理CanvasViewController中绘制涂鸦的命令对象的方法

```
#pragma mark -
#pragma mark Draw Scribble Command Methods

- (void) executeCommand:(Command *)command
         prepareForUndo:(BOOL)prepareForUndo
{
  if (prepareForUndo)
  {
    // 懒加载undoStack_
    if (undoStack_ == nil)
    {
      undoStack_ = [[NSMutableArray alloc] initWithCapacity:levelsOfUndo_];
    }

    // 如果撤销栈满了，就丢掉栈底的元素
    if ([undoStack_ count] == levelsOfUndo_)
    {
      [undoStack_ dropBottom];
    }

    // 把命令压入撤销栈
    [undoStack_ push:command];
  }

  [command execute];
}
```

executeCommand:prepareForUndo:方法负责执行传进来的Command对象，并把它压入undoStack_。这个方法也管理undoStack_的大小，如果栈满了，就把栈底的元素丢掉。

```
- (void) undoCommand
{
  Command *command = [undoStack_ pop];
  [command undo];

  // 把命令压入恢复栈
  if (redoStack_ == nil)
  {
    redoStack_ = [[NSMutableArray alloc] initWithCapacity:levelsOfUndo_];
  }

  [redoStack_ push:command];
}
```

当调用undoCommand方法时，它首先尝试从undoStack_的栈顶取出最后一个命令的引用，然后向它发送undo消息，如代码清单20-7中讨论过的那样，撤销它保存的Mark对象。然后，如果redoStack_还没有实例化，就进行实例化，再把刚刚从undoStack_弹出的Command对象压栈。

```
- (void) redoCommand
{
  Command *command = [redoStack_ pop];
  [command execute];

  // 把命令压回到撤销栈
   [undoStack_ push:command];
}
```

redoCommand甚至比undoCommand更简单。前半部分除了上一个命令是从redoStack_弹出之外，跟undoCommand几乎相同，另外command调用execute而不是undo。之后，它把command压回到undoStack_。因此当undoCommand方法被再次调用时，这个过程得以重复。

那么如何将DrawScribbleCommand对象同实际的绘图事件联系起来呢？我们将通过代码清单20-9中定义在CanvasViewController中的几个触摸事件处理程序进行说明。根据触摸事件绘制不同类型Mark的机制，已经在第12章和20.4.1节第3小节中作了讨论。因此我们只重点介绍与自制的基础设施有关的修改。跟上一个部分一样，我们将把代码清单分为两个部分进行讨论。

代码清单20-9　CanvasViewController中操作DrawScribbleCommand对象的触摸事件处理程序

```
#pragma mark -
#pragma mark Touch Event Handlers

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
  startPoint_ = [[touches anyObject] locationInView:canvasView_];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
  CGPoint lastPoint = [[touches anyObject] previousLocationInView:canvasView_];

  // 如果这是手指的拖动，就向涂鸦添加一条线
  if (CGPointEqualToPoint(lastPoint, startPoint_))
  {
    id <Mark> newStroke = [[[Stroke alloc] init] autorelease];
    [newStroke setColor:strokeColor_];
    [newStroke setSize:strokeSize_];

  - [scribble_ addMark:newStroke shouldAddToPreviousMark:NO];

    NSDictionary *userInfo = [NSDictionary dictionaryWithObjectsAndKeys:
                              scribble_, ScribbleObjectUserInfoKey,
                              newStroke, MarkObjectUserInfoKey,
                              [NSNumber numberWithBool:NO],
                              AddToPreviousMarkUserInfoKey, nil];
    DrawScribbleCommand *command = [[[DrawScribbleCommand alloc] init] autorelease];
```

```
    [command setUserInfo:userInfo];
    [self executeCommand:command prepareForUndo:YES];
  }

  // 把当前触摸作为顶点添加到临时线条
  CGPoint thisPoint = [[touches anyObject] locationInView:canvasView_];
  Vertex *vertex = [[[Vertex alloc]
                     initWithLocation:thisPoint]
                     autorelease];

  [scribble_ addMark:vertex shouldAddToPreviousMark:YES];
}
```

现在我们使用一个DrawScribbleCommand对象，代替addMark:shouldAddToPreviousMark:方法，来添加Stroke对象。它接受一个字典类型的userInfo数据。userInfo为3个键设定了值：ScribbleObjectUserInfoKey、MarkObjectUserInfoKey和AddToPreviousMarkUserInfoKey。这些键在Scribble中做了如下定义：

```
NSString *const ScribbleObjectUserInfoKey = @"ScribbleObjectUserInfoKey";
NSString *const MarkObjectUserInfoKey = @"MarkObjectUserInfoKey";
NSString *const AddToPreviousMarkUserInfoKey = @"AddToPreviousMarkUserInfoKey";
```

我们分别用scribble_、newStroke和作为NSNumber实例的BOOL值NO，设定这3个键的值。然后创建一个DrawScribbleCommand对象的实例，并用刚生成userInfo字典对它作初始化。我们不立即执行它，而是把它传给CanvasViewController的实例方法executeCommand:-command prepareForUndo:来执行，并为撤销作准备，如代码清单20-8中那样。这以后，这个DrawScribbleCommand对象被压入先前定义的撤销栈（即undoStack_），为撤销作好了准备。

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
  CGPoint lastPoint = [[touches anyObject] previousLocationInView:canvasView_];
  CGPoint thisPoint = [[touches anyObject] locationInView:canvasView_];

  // 如果触摸从未移动（抬起之前一直在同一处）
  // 就向现有Stroke组合体添加一个点
  // 否则就把它作为最后一个顶点添加到临时线条
  if (CGPointEqualToPoint(lastPoint, thisPoint))
  {
    Dot *singleDot = [[[Dot alloc]
                       initWithLocation:thisPoint]
                      autorelease];
    [singleDot setColor:strokeColor_];
    [singleDot setSize:strokeSize_];

    [scribble_ addMark:singleDot shouldAddToPreviousMark:NO];

    NSDictionary *userInfo = [NSDictionary dictionaryWithObjectsAndKeys:
                              scribble_, ScribbleObjectUserInfoKey,
                              singleDot, MarkObjectUserInfoKey,
                              [NSNumber numberWithBool:NO],
                              AddToPreviousMarkUserInfoKey, nil];
    DrawScribbleCommand *command = [[[DrawScribbleCommand alloc] init] autorelease];
    [command setUserInfo:userInfo];
    [self executeCommand:command prepareForUndo:YES];
}
```

touchesEnded:方法判断触摸是否在屏幕上最先按下处结束。如果是，它就创建一个新的Dot对象并附加到scribble_中的Mark根节点。我们为新的DrawScribbleCommand对象构造一个类似的userInfo，它用相同类型的元素与用户键相关联。然后像绘制由顶点构成的线条那样，把它传给executeCommand:。

```
    // 在此重置起点
    startPoint_ = CGPointZero;

    // 如果这是线条的最后一点
    // 就用不着画它
    // 因为用户看不出什么区别
}

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
{
    // 在此重置起点
    startPoint_ = CGPointZero;
}
```

代码的其余部分基本没有动。读者可能会问，“是不是忘了在touchesMoved:方法中把添加顶点的操作压栈？”不是的，不压栈是有原因的。我们把撤销基础设施设计成只能撤销和恢复完整的线条和点，而不包括顶点。如果撤销用户创建线条的每“一步”，用户体验和性能都会很糟糕。所以这个地方保留scribble_原来的addMark:消息。

20.4.3　允许用户触发撤销与恢复

至此，似乎一切就绪，我们的撤销基础设施可以工作了。但是在TouchPainter应用程序中用户如何启动撤销和恢复过程呢？在TouchPainter的主画布视图，在工具条的右手边有撤销按钮和恢复按钮，如图20-6所示。

图20-6　显示工具条上的撤销与恢复按钮的主画布视图的屏幕截图

每个按钮都加了标签。当用户单击了两个按钮中任何一个的时候，我们可以在onBarButton-Hit:方法中捕捉到并予以识别，如代码清单20-10所示。

代码清单20-10　CanvasViewController中处理撤销和恢复按钮单击的IBAction方法

```
#pragma mark -
#pragma mark Toolbar button hit method

- (IBAction) onBarButtonHit:(id)button
{
  UIBarButtonItem *barButton = button;

  if ([barButton tag] == 4)
  {
    [self undoCommand];
  }
  else if ([barButton tag] == 5)
  {
    [self redoCommand];
  }
}
```

代码本身不言自明，根据按钮的标签判断它是撤销还是恢复。

对于NSUndoManager的版本，撤销语句要改成下面这句：

```
[self.undoManager undo];
```

恢复语句要改成：

```
[self.undoManager redo];
```

我们完成了在TouchPainter中实现撤销和恢复操作的例子。面向对象软件的设计之中经常能见到命令模式。

20.5　命令还能做什么

命令模式允许封装在命令对象中的可执行指令。这使得在实现撤销和恢复基础设施的时候自然会选择这个模式。但这个模式的用途不只如此。命令对象的另一个为人熟知的应用是推迟调用器的执行。调用器可以是菜单项或按钮。使用命令对象连结不同对象之间的操作相当常见，比如，单击视图控制器中的按钮，可以执行一个命令对象，对另一个视图控制器进行某些操作。命令对象隐藏了与这些操作有关的所有细节。

从本章的示例项目中，读者还可以发现另外几个Command类，它们使用TouchPainter中的其他部分，有着不同的用途。源程序项目的文件中包含很多信息，这里不能一一介绍，欢迎大家下载研究。

20.6 总结

本章介绍了命令模式，以及如何在Objective-C中实现这一模式。我们为TouchPainter应用实现了一个撤销基础设施，这样用户就可以撤销和恢复屏幕上所画的线条和点。我们也讲解了Cocoa Touch框架如何用不同的调用和撤销/恢复策略实现这个模式，让它能够应用到任何iOS应用程序之中。

消除应用程序中命令、调用器、接收器和客户端的耦合，其好处显而易见。如果特定的命令需要在实现中作修改，那么其他组件中的大部分都不受影响。而且添加新的命令类非常容易，因为不必为此修改已有的类。

有关算法封装的这个部分到此就结束了。下一个部分将讨论几个与性能和对象访问有关的设计模式。

Part 8 第八部分

性能与对象访问

本部分内容

- 第 21 章　享元
- 第 22 章　代理

第21章 享 元

21

公共交通（如公共汽车）已有一百多年的历史了。大量去往相同方向的乘客可以分担保有和经营车辆（如公共汽车）的费用。公共交通设有多个车站。乘客沿着路线在接近他们目的地的地方上下车。到达目的地的费用仅与行程有关。跟保有车辆相比，乘坐公共交通要便宜得多。这就是利用公共资源的好处。

在面向对象软件设计中，利用公共对象不仅能节省资源还能提高性能。比方说，某个任务需要一个类的一百万个实例，但我们可以把这个类的一个实例让大家共享，而把某些独特的信息放在外部，节省的资源可能相当可观（一个实例与一百万个实例的差别）。共享的对象只提供某些内在的信息，而不能用来识别对象。专门用于设计可共享对象的一种设计模式叫做享元模式（Flyweight pattern）[①]。

本章将讨论这个模式的一些概念。我们也会为一个示例程序实现这一模式，该程序显示上百个花朵图案，但只使用了6个实例。

21.1 何为享元模式

实现享元模式需要两个关键组件，通常是可共享的享元对象和保存它们的池。某种中央对象维护这个池，并从它返回适当的实例。工厂（抽象工厂或具体工厂，见抽象工厂模式，第5章）是这一角色的理想候选。它可以通过一个工厂方法，根据父类型返回各种类型的具体享元对象。各种框架中，这种工厂通常称为“管理器”（至少“管理器”好像比“工厂”听起来酷[②]）。不管它叫什么，其主要目的就是维护池中的享元对象，并适当地从中返回享元对象。

使得享元对象是“蝇量级”的最重要原因是什么呢？不是它们的大小，而是通过共享能够节省的空间总量。某些（或多数）对象的独特状态（外在状态）可以拿到外部，在别处管理，其余部分被共享。比如说，原来需要一个类的一百万个对象，但因为这个类的对象为享元，现在只要一个就够了。这就是由于可共享的享元对象让整个系统变得轻量的原因。通过仔细的设计，内存的节省可以非常可观。在iOS开发中，节省内存意味着提升整体性能。

---

① Flyweight是体育比赛中体重级别的一个术语——蝇量级。职业拳击比赛中，蝇量级指体重大于108而小于等于112磅（约为49~51公斤）的级别。——译者注

② 管理器，英文为manager，也有“经理”的意思。——译者注

图21-1中的类图表示了它们的静态关系。

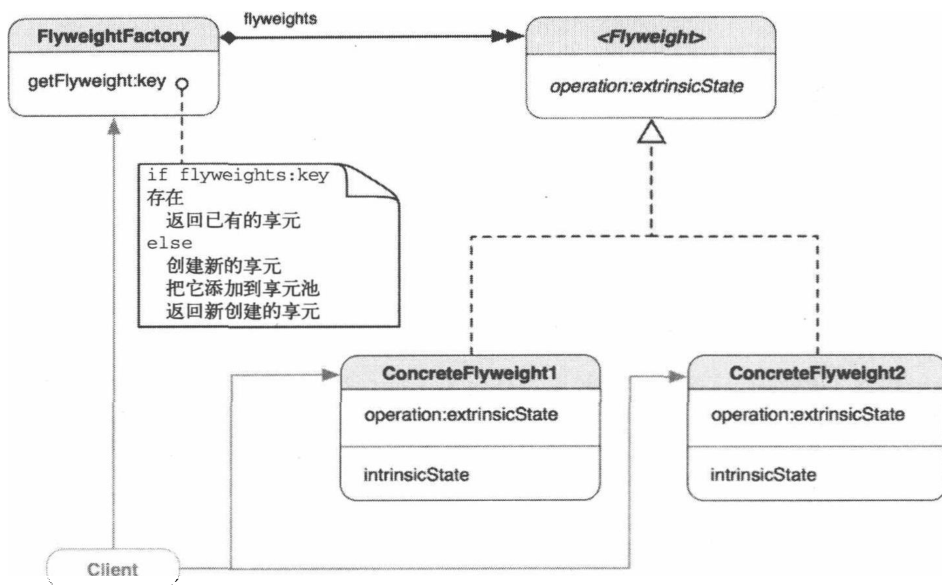


图21-1 享元模式的类图

Flyweight是两个具体享元类ConcreteFlyweight1和ConcreteFlyweight2的父接口（协议）。每个ConcreteFlyweight类维护不能用于识别对象的内在状态intrinsicState。Flyweight声明了operation:extrinsicState方法，由这两个ConcreteFlyweight类实现。intrinsicState是享元对象中可被共享的部分，而extrinsicState补充缺少的信息，让享元对象唯一。客户端向operation:消息提供extrinsicState，让享元对象使用extrinsicState中的独一无二的信息完成其任务。

图21-2显示了运行时FlyweightFactory的实例如何管理池中的享元对象。

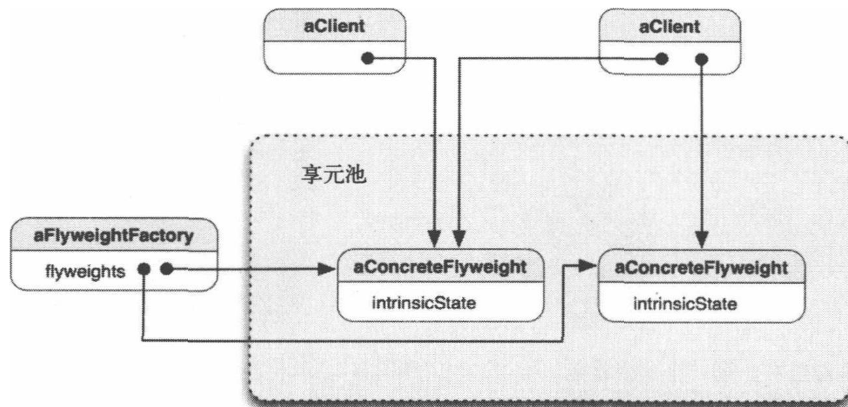


图21-2 说明运行时享元对象如何共享的对象图

享元模式：运用共享技术有效地支持大量细粒度的对象。*

* 最初的定义出现于《设计模式》(Addison-Wesley，1994)。

21.2　何时使用享元模式

当满足以下所有条件时，自然会考虑使用这个模式：

- 应用程序使用很多对象；
- 在内存中保存对象会影响内存性能；
- 对象的多数特有状态（外在状态）可以放到外部而轻量化；
- 移除了外在状态之后，可以用较少的共享对象替代原来的那组对象；
- 应用程序不依赖于对象标识，因为共享对象不能提供唯一的标识。

我们将开发一个应用程序，使用可共享的花朵池绘制几百个花朵图案，以说明这一模式的概念。

21.3　创建百花池

我们要开发一个小应用程序，在屏幕上随机显示花朵图案。我们要显示如图21-3所示的6种花朵。

画了很多朵这些花之后，屏幕就填满了花，如图21-4所示。

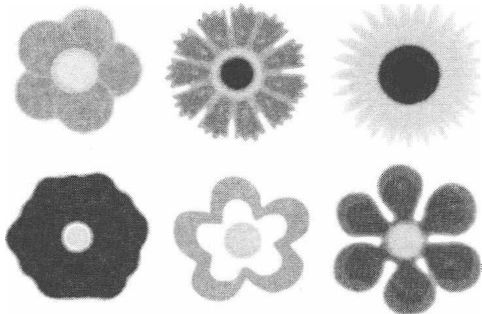


图21-3　从左至右，第一排：银莲花、大波斯菊、非洲菊；第二排：蜀葵、茉莉、百日菊

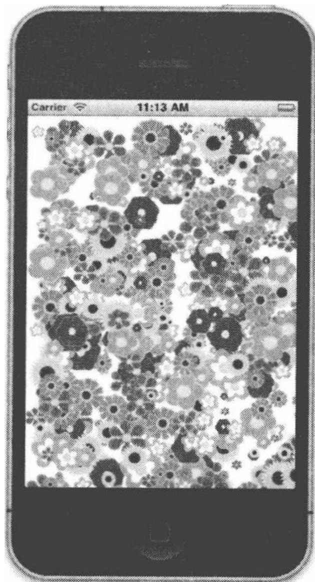


图21-4　“百花池”的实际截屏图。尽管屏幕上画了500朵花，但只使用了6个不同的花朵视图的实例

我们的目标是用6个不同的实例，画很多（几百个或更多）随机尺寸和位置的花。如果为屏幕上所画的每朵花创建一个实例，程序会占用很多内存。我们的方案是使用享元模式来限制花朵实例的数量，让它不多于可选花朵类型的总数。

对于图21-1中的类图这样的设计，需要一种享元工厂和一些享元产品。FlowerView是UIImageView的子类，用它可以绘制一幅花朵图案。这个程序所用的享元工厂称为Flower-Factory，它管理一个FlowerView实例的池。尽管池中对象的类是FlowerView，但客户端只要求FlowerFactory返回UIView的实例。与让工厂返回UIImage型的最终产品相比，这样的设计更加灵活。因为要是由于某种原因，我们也需要能够自行绘制的花朵，而不只是显示固定的图像，那么几乎全部都要修改——那就麻烦了。UIView被看做任何需要在屏幕上绘图的事物的高层抽象。FlowerFactory可以返回任何UIView子类的对象，而不会破坏系统。这就是“针对接口编程，而不是针对实现编程”的一个好处。

设计并实现可共享的花朵

我们有如图21-3中所示的6种花朵图案：银莲花、大波斯菊、非洲菊、蜀葵、茉莉和百日菊。每个图案由一个唯一的FlowerView的实例来维护，而与需要的花朵总数无关。它们的静态关系如图21-5中的类图所示。

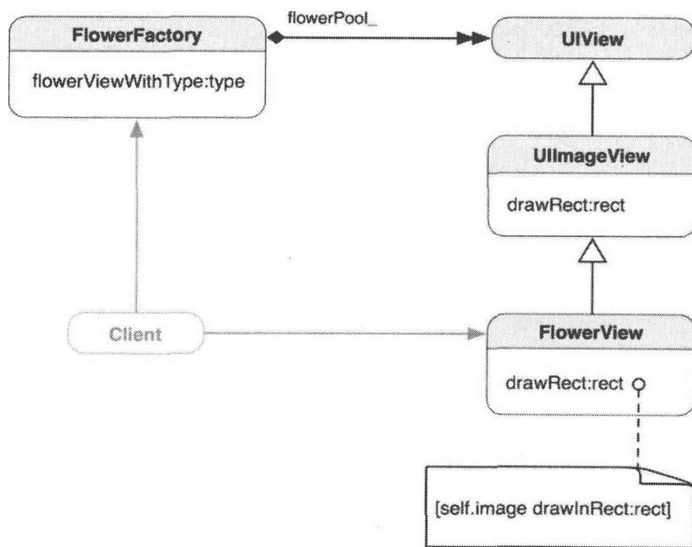


图21-5 作为享元的FlowerView的类图

FlowerFactory用flowerPool_聚合了一个花朵池的引用。flowerPool_是一个保存FlowerView的所有实例的数据结构。FlowerFactory用flowerViewWithType:方法，以UIView的形式返回FlowerView的实例。如果池中没有所请求花朵的类型，就会创建一个新的FlowerView的实例。

1. FlowerView类的实现

FlowerView的类声明如代码清单21-1所示。

代码清单21-1 FlowerView.h中FlowerView的类声明

```
@interface FlowerView : UIImageView
{

}

- (void) drawRect:(CGRect)rect;

@end
```

再简单不过了！它只重载了UIImageView的drawRect:rect方法，仅此而已。这里重新声明了这个方法，这样这个类实现了哪些方法就一目了然。这个方法的实现只是让super（UIImage）中存储的图像在一个指定的矩形区域内单独进行绘制，如代码清单21-2所示。

代码清单21-2 FlowerView.m中FlowerView的实现

```
#import "FlowerView.h"

@implementation FlowerView

- (void) drawRect:(CGRect)rect
{
  [self.image drawInRect:rect];
}

@end
```

2. FlowFactory类的实现

FlowerFactory的声明跟图21-5中所见到的一样，如代码清单21-3所示。

代码清单21-3 FlowerFactory.h中FlowerFactory的类声明

```
@interface FlowerFactory : NSObject
{
  @private
  NSMutableDictionary *flowerPool_;
}

- (UIView *) flowerViewWithType:(FlowerType)type;

@end
```

FlowerFactory有一个私有的NSMutableDictionary变量flowerPool_，用来保存整个可供返回花朵的池。它也定义了根据FlowerType参数返回特定UIView实例的工厂方法flowerViewWithType:(FlowerType)type。FlowerType是根据图21-3中花的名称定义的一组枚举值，如代码清单21-4所示。

代码清单21-4 FlowerType的定义

```
typedef enum
{
  kAnemone,
  kCosmos,
  kGerberas,
  kHollyhock,
  kJasmine,
  kZinnia,
  kTotalNumberOfFlowerTypes
} FlowerType;
```

显然，kTotalNumberOfFlowerTypes是所支持花朵类型的总数。FlowerFactory的flowerWithType:方法使用除kTotalNumberOfFlowerTypes之外的值，来决定要返回哪种花。

来看看FlowerFactory是如何管理花朵池的，请看代码清单21-5。

代码清单21-5 FlowerFactory.m中FlowerFactory的实现

```
#import "FlowerFactory.h"
#import "FlowerView.h"

@implementation FlowerFactory

- (UIView *) flowerViewWithType:(FlowerType)type
{
  // 懒加载花朵池
  if (flowerPool_ == nil)
  {
    flowerPool_ = [[NSMutableDictionary alloc]
                   initWithCapacity:kTotalNumberOfFlowerTypes];
  }

  // 尝试从花朵池中取出一朵花
  UIView *flowerView = [flowerPool_ objectForKey:[NSNumber
                                                   numberWithInt:type]];

  // 如果请求的类型不存在，
  // 那么就创建一个，并加到池里
  if (flowerView == nil)
  {
    UIImage *flowerImage;

    switch (type)
    {
      case kAnemone:
        flowerImage = [UIImage imageNamed:@"anemone.png"];
        break;
      case kCosmos:
        flowerImage = [UIImage imageNamed:@"cosmos.png"];
        break;
      case kGerberas:
        flowerImage = [UIImage imageNamed:@"gerberas.png"];
```

```
        break;
      case kHollyhock:
        flowerImage = [UIImage imageNamed:@"hollyhock.png"];
        break;
      case kJasmine:
        flowerImage = [UIImage imageNamed:@"jasmine.png"];
        break;
      case kZinnia:
        flowerImage = [UIImage imageNamed:@"zinnia.png"];
        break;
      default:
        break;
    }

    flowerView = [[[FlowerView alloc]
                   initWithImage:flowerImage] autorelease];
    [flowerPool_ setObject:flowerView
                    forKey:[NSNumber numberWithInt:type]];
  }

  return flowerView;
}

- (void) dealloc
{
  [flowerPool_ release];
  [super dealloc];
}

@end
```

花朵池(flowerPool_)没有在FlowerFactory的init方法中初始化，而是在工厂方法flowerViewWithType:中进行懒加载。池的默认容量是kTotalNumberOfFlowerTypes（也就是6)。如果请求的实例不在池中，工厂就会用适当的花朵图像创建一个新的FlowerView的实例，并以type作为键，以新的实例作为值，加到池中。对于已创建的花朵，随后的请求将返回池中的实例。从池中共享花朵的机制相当简单。

3. 如何共享FlowerView

在这个模式的原始定义中，享元对象似乎总是和某种可共享的内在状态联系在一起。尽管并不完全如此，但我们的FlowerView享元对象确实共享了作为其内在状态的内部花朵图案。可共享的享元对象也可以被用作“策略”（见策略模式，第19章)。顺便提一下，策略通常指内嵌的算法。但是，不管享元对象是否有可共享的内在状态，仍然需要定义某种外部的数据结构，以保存享元对象的外在状态（独特的信息)。我们为此定义一个C结构体ExtrinsicFlowerState，如代码清单21-6所示。

代码清单21-6　ExtrinsicFlowerState结构体的定义

```
typedef struct
{
  UIView *flowerView;
```

```
    CGRect area;
} ExtrinsicFlowerState;
```

每朵花都有各自的显示区域，所以这需要作为外在状态来处理。我们需要一个数组为客户端生成的花朵保存这种信息。图21-6是一个示意图，它解释了外在花朵状态的数组如何与FlowerFactory的享元池中的FlowerView独特实例联系起来。

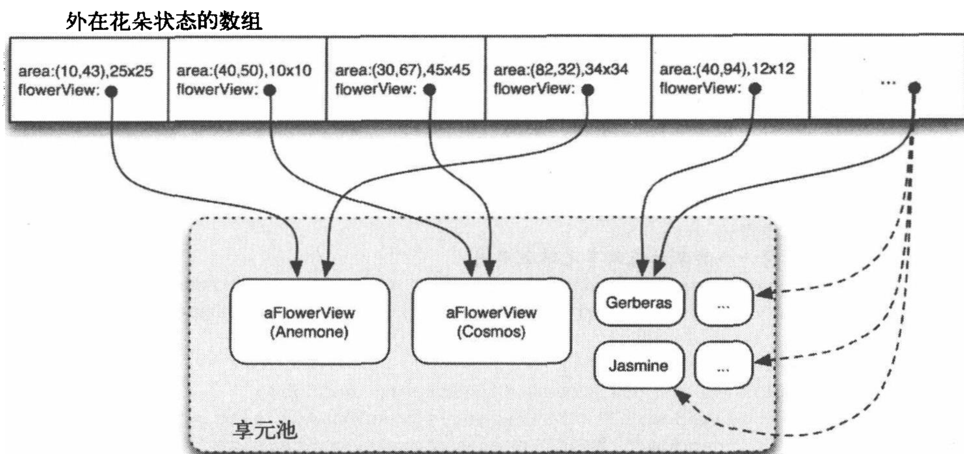


图21-6 外在花朵状态的数组中的元素，共享享元池中由FlowerManager维护的FlowerView对象的相同独特实例

数组中的每个元素保存有一个指针和花朵的显示区域，指针指向池中真正的FlowerView对象。数组的大小与池的大小无关。

能够节省多少空间

通过享元对象能够节省的空间，取决于几个因素：

- 通过共享减少的对象总数；
- 每个对象中内在状态（即，可共享的状态）的数量；
- 外在状态是计算出来的还是保存的。

然而，对享元对象外在状态的传递、查找和计算，可能产生运行时的开销，尤其在外在状态原本是作为内在状态来保存的时候。当享元的共享越来越多时，空间的节省会抵消这些开销。共享的享元越多，节省的存储就越多。节省直接跟共享的状态相关。如果对象有大量内在和外在状态，外在状态又能够计算出来而不用存储的时候，就能节省最大的空间。这样我们以两种方式节省了存储空间：共享减少了内在状态的开销，通过牺牲计算时间又节省了外在状态的存储空间。

4. 客户端代码的实现

现在所有部分都到位了，我们可以看看客户端如何使用这个享元基础设施绘制500朵花，请

看代码清单21-7。

代码清单21-7　构造花朵列表的客户端代码

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  // 构造花朵列表
  FlowerFactory *factory = [[[FlowerFactory alloc] init] autorelease];
  NSMutableArray *flowerList = [[[NSMutableArray alloc]
                                  initWithCapacity:500] autorelease];

  for (int i = 0; i < 500; ++i)
  {
    // 使用随机花朵类型,
    // 从花朵工厂取得一个共享的花朵享元对象实例
    FlowerType flowerType = arc4random() % kTotalNumberOfFlowerTypes;
    UIView *flowerView = [factory flowerViewWithType:flowerType];

    // 设置花朵的显示位置和区域
    CGRect screenBounds = [[UIScreen mainScreen] bounds];
    CGFloat x = (arc4random() % (NSInteger)screenBounds.size.width);
    CGFloat y = (arc4random() % (NSInteger)screenBounds.size.height);
    NSInteger minSize = 10;
    NSInteger maxSize = 50;
    CGFloat size = (arc4random() % (maxSize - minSize + 1)) + minSize;

    // 把花朵的属性赋给一个外在状态对象
    ExtrinsicFlowerState extrinsicState;
    extrinsicState.flowerView = flowerView;
    extrinsicState.area = CGRectMake(x, y, size, size);

    // 把外在花朵状态添加到花朵列表
    [flowerList addObject:[NSValue value:&extrinsicState
                          withObjCType:@encode(ExtrinsicFlowerState)]];
  }

  // 把花朵列表添加到这个FlyweightView实例
  [(FlyweightView *)self.view setFlowerList:flowerList];
}
```

在for循环的每次迭代中，FlowerFactory的实例根据随机选择的花朵类型FlowerType flowerType = arc4random() % kTotalNumberOfFlowerTypes返回一个UIView的实例（其实是FlowerView）。flowerType的整数值不应大于kTotalNumberOfFlowerTypes。特定的位置和大小被设置到ExtrinsicFlowerState结构体之中，这个结构体与指向FlowerView的实例的一个指针关联在一起。然后整个外在状态结构体被添加到一个NSArray对象——flowerList之中。因为ExtrinsicFlowerState不是Objective-C对象，需要用@encode把它编码成NSValue对象，然后才能安全地添加到数组。当向数组中添加完500朵花的列表之后，我们把这个列表设置给FlyweightView的实例（即self.view）。

把花朵列表添加到FlyweightView只能算完成了一半，还需要知道FlyweightView如何把列表中的花朵展现在屏幕上，请看代码清单21-8。花朵列表构造好之后，把它赋给定制视图，显示列表中的每朵花。例子中的列表叫做flowerList_。

代码清单21-8　定制视图重载了UIView的drawRect:方法，在屏幕上绘制花朵

```
- (void)drawRect:(CGRect)rect
{
  // 绘图代码

  for (NSValue *stateValue in flowerList_)
  {
    ExtrinsicFlowerState state;
    [stateValue getValue:&state];

    UIView *flowerView = state.flowerView;
    CGRect area = state.area;

    [flowerView drawRect:area];
  }
}
```

drawRect:方法通常是定义定制的绘图操作的地方。在这个方法中，我们用一个for循环绘制花朵数组中先前保存的NSValue项的整个列表。每一步迭代中，我们从NSValue实例取出ExtrinsicFlowerState结构体。从ExtrinsicFlowerState结构体取得FlowerView的指针和绘图区域之后，我们向FlowerView实例发送一个drawRect:rect消息，执行它的绘图操作（在代码清单21-2中作过讲解）。

视图上绘制了500朵花之后，就会看到图21-4中那样的画面。

21.4　总结

分享是人类的美德。分享相同的资源以执行任务，可能比使用个人的资源完成同样的事情更加高效。本章，我们设计了一个可以在屏幕上显示500多朵花的程序，而只用了6个不同花朵图案的实例。这些不同的花朵实例，把一些与众不同的可被标识的信息（即位置和大小）去掉，只剩下显示花朵图案的基本操作。在请求特定的花朵的时候，客户端需要向花朵实例提供某些与众不同的信息（外在状态），让它使用这些信息绘制一朵与众不同的花。不用享元模式的话，要在屏幕上画多少朵花，程序就需要实例化多少个UIImageView（即500多个）。经过精心的设计，享元模式可以通过共享一部分必需的对象，来节省大量的内存。

在下一章将介绍另一种设计模式，它通过把操作推迟给代理对象来提升性能并提供对象访问。

第22章 代 理

提供免费试用期是相当常见的促销手段，有时甚至30天退款保证也不如提供“免费”使用的效果好。对于某些贵重商品或者在线订阅服务，这种做法尤为有效。

有很多在线交友网站向寻找伴侣的单身用户提供会员服务。很多都很贵，比方说3个月的无限制搜索和与其他会员通信需要100美元。这很贵，而且不保证付了100美元之后在3个月内一定能找到理想的对象。于是很多交友网站提供免费的搜索、挑逗或者其他有限的功能，但不提供跟其他会员的实际通信。这种方式比让人先掏100美元然后祝自己好运更容易被接受。如果用户真的喜欢这个服务，并从其他会员那里（包括付费和非付费的）收到了很多挑逗和消息，那么用户可能更愿意掏那100美元来参与游戏。

那么读者会问：“这跟代理模式有什么关系？”代理的一个常见用处是作为一个轻量的替身对象，它允许客户端首先访问某些廉价的信息或功能。直到值得或需要使用“真货”的时候，代理才会去为客户端准备真正的、高价的资源。因此代理在一开始向用户提供试用会员资格，当用户愿意为真正的、高价的会员资格付费的时候，代理会敞开大门让用户访问更多只对付费会员开放的功能。

从这一思想细化而来的一种设计模式叫做代理模式。本章将讨论这一模式的概念及其主要功能。我们也要设计并实现一个虚拟图像代理，在真正的图像在后台加载期间显示一幅占位图像。

22.1 何为代理模式

有以下几种代理。

- 远程代理（remote proxy）为位于不同地址空间或网络上的对象提供本地代表。
- 虚拟代理（virtual proxy）根据需要创建重型对象。
- 保护代理（protection proxy）根据各种访问权限控制对原对象的访问。
- 智能引用代理（smart-reference proxy）通过对真正对象的引用进行计数来管理内存。也用于锁定真正对象，让其他对象不能对其进行修改。

> **代理模式：**为其他对象提供一种代理以控制对这个对象的访问。*
>
> *最初的定义出现于《设计模式》（Addison-Wesley，1994）。

通常，代理是一种替代或者占位，它控制对另一些对象的访问，而这些对象可能是远程对象，

创建的开销较大的对象，或者是对安全性有要求的对象。这里没有办法对这些代理作一一介绍，只重点讲解虚拟代理。

代理模式的思想是使用一个基本上跟实体对象行为相同的代理。客户端可以“透明地”使用代理，即不必知悉所面对的只是一个代理而不是实体对象。当客户端请求某些创建的开销较大的功能时，代理将把请求转发给实体对象，准备好请求的功能并返回给客户端。客户端不知道幕后发生了什么。代理和实体对象同样拥有客户端要求的行为。图22-1中的类图解释了这一思想。

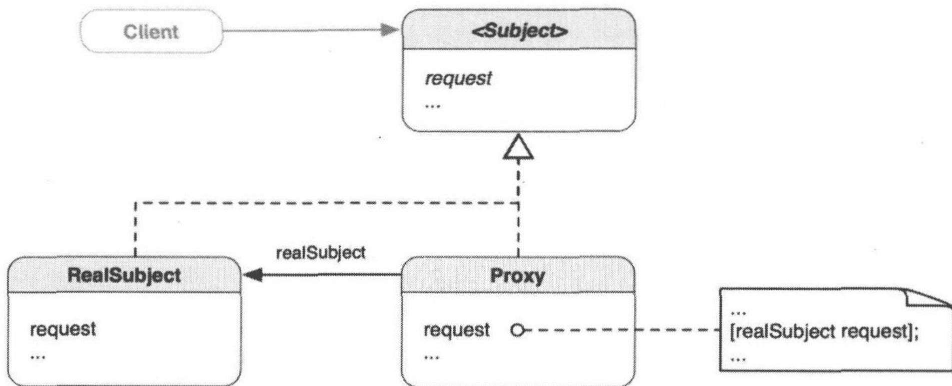


图22-1 代理模式的类图

当客户端向Proxy对象发送request消息时，Proxy对象会把这个消息转发给Proxy对象之中的RealSubject对象。RealSubject会实施实际的操作间接满足客户端的请求。

在运行时，我们可以想象这样一个场景：客户端以抽象类型引用一个对象。这个引用实际上是个Proxy对象。Proxy对象本身有一个对RealSubject实例的引用，以后如果接到请求，此实例将执行高强度的工作。这个运行时的场景如图22-2所示。

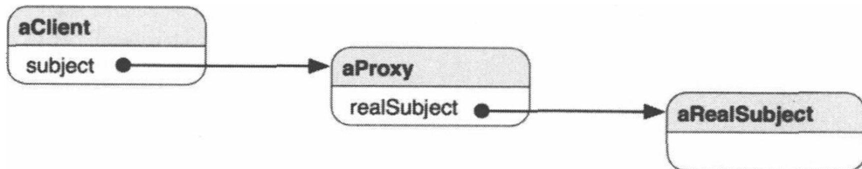


图22-2 代理模式在运行时的一种可能的对象结构

22

22.2 何时使用代理模式

在下列情形，自然会考虑使用这一模式。

- 需要一个远程代理，为位于不同地址空间或网络中的对象提供本地代表。
- 需要一个虚拟代理，来根据要求创建重型的对象。本章稍后将在示例代码中实现这种代理。
- 需要一个保护代理，来根据不同访问权限控制对原对象的访问。

- 需要一个智能引用代理，通过对实体对象的引用进行计数来管理内存。也能用于锁定实体对象，让其他对象不能修改它。

22.3 用虚拟代理懒加载图像

第2章中为TouchPainter应用定义了一些需求。其中有一个是让`ThumbnailViewController`允许用户浏览所有保存在文件系统中的涂鸦图。但是把它们全部加载又不可行，尤其是当设备的内存很有限的时候。即便可以在用户进入视图时把所有的缩略图图像都加载到内存，但如果给应用程序留下的内存很少，性能也会受到影响。

在第2章中讨论的最初设计中，涂鸦被表示成小缩略图，一行一行地排列于屏幕之上。每一行是一个定制的`UITableViewCell`，它包含多个缩略图的占位器，将会显示相应涂鸦的缩略图图像。在涂鸦的实际缩略图图像加载之前，缩略图占位器会显示一幅占位图像，这幅占位图像是与视图的可视区域中的其他占位器共享的。运行时的框架将管理`UITableViewCell`对象。不可见（在屏幕可见区域之外）的单元格将被销毁，而进入可见区域的单元格将被创建或初始化，重用屏幕外的单元格资源。

当真正的缩略图图像完成加载之后，原先的占位图像将被真实图像替换。这一过程一直继续，直到屏幕上的所有缩略图图像都完成了加载为止。图22-3的左边是刚被加载时的缩略图视图的截屏图，全部都是占位图像。图22-3的右边是加载了部分涂鸦缩略图图像的同一个视图。

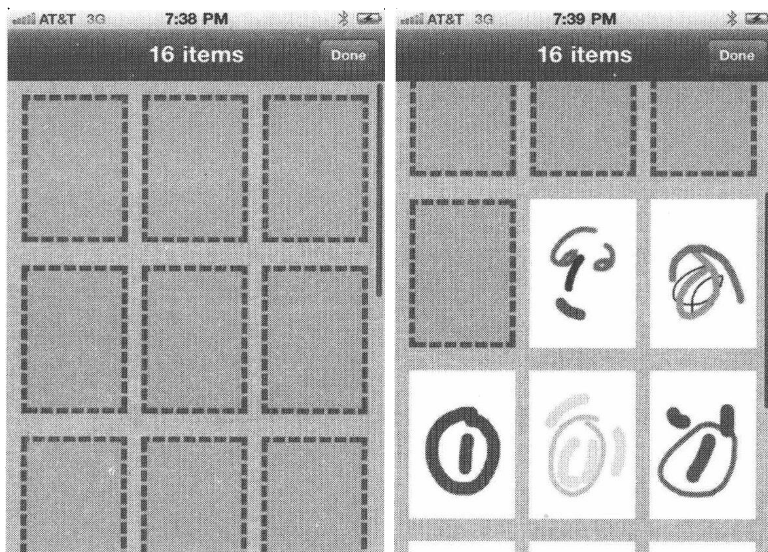


图22-3　只有占位图像的涂鸦缩略图视图与加载了部分涂鸦缩略图图像之后的同一个视图

这种用户界面用于某些苹果公司开发的iOS应用之中，在用户滚动全部缩略图的视图时加载缩略图图像。在后台加载一个一个的缩略图的同时，给予用户一种流畅的操作与响应。

涂鸦缩略图视图的设计与实现

产生这种“神奇”的缩略图视图主要依靠两个元素，一个是占位图像代理，另一个是在后台加载真正图像的机制，它们使整个过程显得很流畅。我们可以把基本的代理设计画在类图中，如图22-4所示。

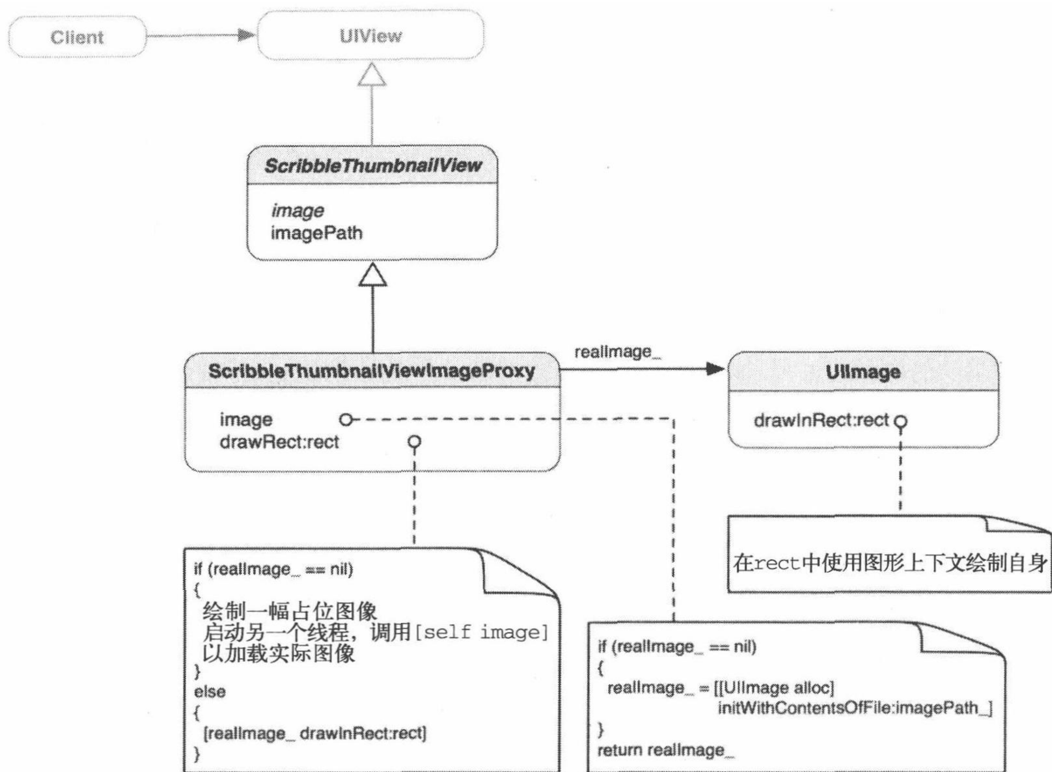


图22-4 ScribbleThumbnail代理结构的类图

ScribbleThumbnailViewImageProxy与由UIImage所表示的真实图像对象的类图基本结构，把握了原始代理模式的本质。它们经过定制，以适应TouchPainter应用的需要。ScribbleThumbnailViewImageProxy和UIImage实现了差不多一样的draw*接口，用以在UIView对象之上绘图。ScribbleThumbnailView本身是UIView的子类，所以其子类可用于对它们进行显示的UITableViewCell。整个用户界面的完整实现涉及许多其他部分。对于这个例子，我们只会介绍一些能够强调代理模式的概念的基本要素。如果想要了解各个部分的实现细节，读者可以下载一份本章的示例代码。

在ScribbleThumbnailViewImageProxy中的图像转发加载机制的思想是，如果实际图像还没有加载，就进行加载，然后在屏幕上显示。但这只是最低要求。要把它用于TouchPainter应用，还需要一些更复杂的步骤。我们将在稍后讨论示例代码的实现部分时进行介绍。

至此，我们对于视图上显示的缩略图的代理会设计成什么样子，已有了整体概念。为了对运行时的代理结构有个直观印象，请看图22-5。

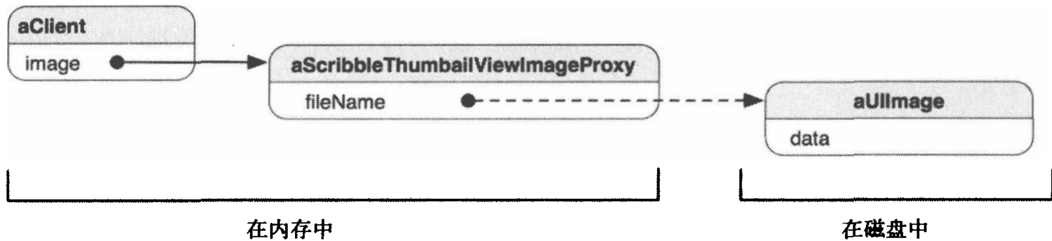


图22-5 运行时的ScribbleThumbnailViewImageProxy对象和UIImage对象

在运行时，aClient访问ScribbleThumbnailView的子类的实例，在这里是ScribbleThumbnailViewImageProxy。直到aClient发出请求时，才会创建UIImage的实例。因此ScribbleThumbnailViewImageProxy对象收到加载请求之前，真实图像一直是在磁盘中。如果没有请求实际的加载，那么内存中就只有ScribbleThumbnailViewImageProxy对象。我们可以使用类似的懒加载方式，来加载任何其他需要显示在视图中的大开销资源。

1. ScribbleThumbnailView类的实现

代码清单22-1是ScribbleThumbnailView的类声明。

代码清单22-1 ScribbleThumbnailView.h中ScribbleThumbnailView的类声明

```
@interface ScribbleThumbnailView : UIView
{
  @protected
  NSString *imagePath_;
}

@property (nonatomic, readonly) UIImage *image;
@property (nonatomic, copy) NSString *imagePath;

@end
```

总的来说，关于ScribbleThumbnailView的行为，我们唯一关心的是用来加载并返回真实图像的实际路径。作为抽象基类，ScribbleThumbnailView保存着抽象的image和imagePath属性。这两个属性对于后面将要看到的整个虚拟代理操作至关重要。在ScribbleThumbnailView的实现中，只定义了几个有关属性的指令，如代码清单22-2所示。

代码清单22-2 ScribbleThumbnailView.m中ScribbleThumbnailView的实现

```
#import "ScribbleThumbnailView.h"

@implementation ScribbleThumbnailView

@dynamic image;
@synthesize imagePath=imagePath_;

@end
```

2. ScribbleThumbnailViewImageProxy类的实现

目前一切似乎都很简单。现在我们要开始代理设计中最麻烦的部分——ScribbleThumbnailViewImageProxy的实现。它的类声明如代码清单22-3所示。

代码清单22-3 ScribbleThumbnailViewImageProxy.h中ScribbleThumbnailViewImageProxy的类声明

```
#import "ScribbleThumbnailView.h"

@interface ScribbleThumbnailViewImageProxy : ScribbleThumbnailView
{
  @private
  UIImage *realImage_;
  BOOL loadingThreadHasLaunched_;
}

@property (nonatomic, readonly) UIImage *image;

@end
```

realImage_用于保存加载后的真实图像引用。私有的BOOL型成员变量loadingThreadHasLaunched_，以后会用于转发真实图像的加载过程。ScribbleThumbnailViewImageProxy的实现相当长。先去倒杯咖啡我们再开始。

如果你准备好了，那我们就开始吧，请看代码清单22-4中它的实现。

代码清单22-4 ScribbleThumbnailViewImageProxy.m中ScribbleThumbnailViewImageProxy的实现

```
#import "ScribbleThumbnailViewImageProxy.h"

// 用于转发加载线程的私有范畴
@interface ScribbleThumbnailViewImageProxy ()

- (void) forwardImageLoadingThread;

@end

@implementation ScribbleThumbnailViewImageProxy

@dynamic imagePath;

// 如果不需要把对象显示在视图上，客户端可以直接使用这个方法来转发真实图像的加载
- (UIImage *) image
{
  if (realImage_ == nil)
  {
    realImage_ = [[UIImage alloc] initWithContentsOfFile:imagePath_];
  }

  return realImage_;
}

// 在不同的线程会建立转发的调用，从真实图像加载实际内容
```

```
// 在实际内容返回以前，drawRect:会处理后台的加载过程，并绘制一个占位图框
// 一旦实际内容加载完成，就会用实际内容进行重画
- (void)drawRect:(CGRect)rect
{
  // 如果realImageView_中没有真实图像，
  // 就绘制一幅空白图框，作为占位图像
  if (realImage_ == nil)
  {
    // 绘图代码
    CGContextRef context = UIGraphicsGetCurrentContext();

    // 使用虚线绘制占位图框
    // 虚线的画线长度为10个用户空间单位，
    // 画线之间的间隙为3个用户空间单位
    CGContextSetLineWidth(context, 10.0);
    const CGFloat dashLengths[2] = {10,3};
    CGContextSetLineDash (context, 3, dashLengths, 2);
    CGContextSetStrokeColorWithColor(context, [[UIColor darkGrayColor] CGColor]);
    CGContextSetFillColorWithColor(context, [[UIColor lightGrayColor] CGColor]);
    CGContextAddRect(context, rect);
    CGContextDrawPath(context, kCGPathFillStroke);

    // 如果还没有加载实际内容，
    // 就启动一个线程进行加载
    if (!loadingThreadHasLaunched_)
    {
      [self performSelectorInBackground:@selector(forwardImageLoadingThread)
                             withObject:nil];
      loadingThreadHasLaunched_ = YES;
    }
  }
  // 否则就向realImage_传递draw*：消息，让它绘制真实图像
  else
  {
    [realImage_ drawInRect:rect];
  }
}

- (void) dealloc
{
  [realImage_ release];
  [super dealloc];
}

#pragma mark -
#pragma mark A private method for an image forward loading thread

- (void) forwardImageLoadingThread
{
  NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

  // 转发实际内容的加载
  [self image];

  // 使用刚加载的图像进行重画
```

```
    [self performSelectorInBackground:@selector(setNeedsDisplay) withObject:nil];

    [pool release];
  }

  @end
```

恭喜你读完了！因为这个实现代码很长，不能详细介绍。但可以将整个代理操作总结如下。

- 如果realImage_没有加载，image属性方法会使用[[UIImage alloc] initWithContentsOf File: imagePath_]加载真实图像。最后返回realImage_。
- ScribbleThumbnailViewImageProxy是UIView的子类，它的drawRect:方法给出了定制的绘图算法。本例中，它或者绘制一幅占位图像，或者，如果实际图像已加载，就绘制实际图像。如果实际图像没有加载，那么创建一个新的线程，启动image属性方法中定义的加载过程。
- 线程方法forwardImageLoadingThread，定义在一个私有匿名范畴中。这个方法会执行image属性方法，来把真实图像加载到UIImage对象。然后向self发送setNeedsDisplay消息，触发视图内容的刷新。此时由于真实图像已经加载完毕，drawRect:方法会向它转发一个drawInRect:消息，以绘制真正的UIImage对象，而不再绘制占位图像。

有几点需要提一下。私有成员变量loadingThreadHasLaunched_用于判断是否已经在drawRect:方法中启动了一个forwardImageLoadingThread线程。如果是，就只绘制原来的占位图像，因为代理只需要执行一次真实图像的加载。如果代理缩略图视图位于视图的可显示区域之外，UITableView将会销毁它。如果缩略图视图又回到视图中，那么UITableView会重新构建它并再次启动代理的转发加载过程。

22.4 在 Cocoa Touch 框架中使用代理模式

Objective-C不支持多重继承。所以，如果代理对象不是Cocoa Touch框架中任何类的子类的话，可以考虑使用NSProxy作为占位或代替对象。

NSProxy是Cocoa框架中如NSObject一样的根类（root class）。NSProxy实现了NSObject协议，所以NSProxy对象实际上也是NSObject类型。NSProxy类是一个抽象基类，所以它没有自己的初始化方法。对NSProxy对象调用它不知如何响应的方法，将会抛出异常。

NSProxy的主要作用是，为其他对象（甚至还不存在的对象）的替身对象定义一个API。发给代理对象的消息会被转发给实体对象，或者，让代理加载实体对象或把代理自身变成实体对象。NSProxy的子类可以用来实现创建的开销比较大的对象的懒实例化，例如，像前几节的例子中那样来自文件系统的一幅大图像。

尽管NSProxy被看做NSObject类型，它的存在只有一个目的——当代理。forwardInvocation:和methodSignatureForSelector:这两个实例方法对于整个代理过程至关重要。NSProxy的子类除了可能会需要初始化方法和几个有用的属性之外，甚至不需要其他额外的方法。关键在于，

当一个NSProxy子类的对象不能响应实体对象可能会有的方法时，Objective-C的运行库会向代理对象发送一个methodSignatureForSelector:消息，取得被转发的消息的正确方法签名。接下来，运行库会使用返回的方法签名，构造一个NSInvocation实例并使用forwardInvocation:消息把它发送给代理对象，让它把调用转发给其他对象。如果作为NSProxy子类的代理对象可以响应这个消息，那么forwardInvocation:方法就根本不会被调用。虽然Objective-C不支持多重继承，但是我们可以使用NSProxy的消息转发机制，来转发可由其他类的对象处理的任务，达成同样的目的。

iOS应用中代理模式的一个常见的例子是邮件（Mail）应用。邮件消息中收到的附件只会显示一些基本信息，比如文件名和大小，如图22-6中的截屏图所示。

当用户点击占位图像，开始加载附件的内容时，图标会替换为一个小进度视图，表示实际的加载进度，如图22-7所示。

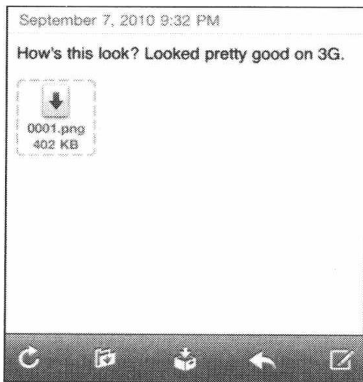


图22-6　邮件消息中尚未加载的附件

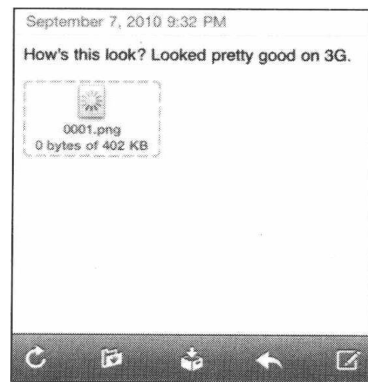


图22-7　邮件应用中正在加载的附件

当附件加载完毕，实际的图像就会显示在消息区域，如图22-8所示。

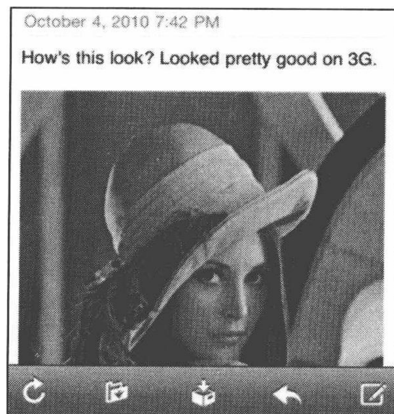


图22-8　附在邮件消息中加载完毕的图像

原先的附件占位图标是一个代理，在收到请求时通过网络加载实际的图像。如果用户不单击占位图标，附件就永远不会加载，而只会显示附件的基本信息。这不仅节省了网络资源，也节省了内存和邮件应用运行时的等待时间。

22.5 总结

在iOS应用开发中，总是要关注内存的使用量。不论应用程序运行在何种iOS设备上，出于性能的考虑，总是推荐懒加载技术。读者可以在iOS应用中使用代理模式，对开销大的数据实施懒加载，如文件系统中的大图像文件或者通过低速网络从服务器下载的大型数据。如果大开销的对象在收到请求之前不需要加载，则可通过虚拟代理向客户端提供某些轻量的信息。

下一个部分将介绍另一种设计模式，它会对对象状态的保存过程进行抽象。

Part 9 第九部分

对象状态

本部分内容

■ 第23章　备忘录

第23章 备 忘 录

备忘录定义为“作为某人或某事的提醒物或纪念品而保留下来的物品”。这真的让我想到了即时贴。我不如我的同事那么喜欢用即时贴。不管什么事情他都写到即时贴，贴在办公桌上。那真是简短的提醒，有时就是涂鸦，潦草得只有原作者（他本人）才能看得懂。在其他人（也包括我）看来，这就是一堆废纸。这些纸对原作者之外的人来说没有任何意义。每当我们需要记下一个简单小巧的提醒时，就把它写到即时贴上。后来这条信息用过之后，这个提醒就没用了，我们就会把它丢进垃圾桶，忘了它（我知道这一点也不环保）。

我们借用了类似的思想，来保存对象的状态并在后来进行恢复。状态本身被创建为一种对象形式（即时贴）。它封装了原始对象的内部状态（作者创作的涂鸦）。只有创建即时贴的原始对象才能看懂保存的状态并用它恢复原来的状态。从这一思想精心设计而来的一种设计模式叫做备忘录（Memento）模式。

23.1 何为备忘录模式

在响应某些事件时，应用程序需要保存自身的状态，比如当用户保存文档或程序退出时。例如，游戏退出之前，可能需要保存当前会话的状态，如游戏等级、敌人数量、可用武器的种类等。游戏再次打开时，玩家可以从离开的地方接着玩。很多时候，保存程序的状态真的不需要什么特别奇妙的方法。任何简单有效的方法都可以，但是同时，保存信息应该只对原始程序有意义。原始程序应该是能够解码它所保存文档中的信息的唯一实体。这就是备忘录模式应用于游戏、文字处理等程序的软件设计中的方式，这些程序需要保存当前上下文的复杂状态的快照并在以后恢复。

> **备忘录模式**：在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。*
>
> *最初的定义出现于《设计模式》（Addison-Wesley，1994）。

这个模式中有3个关键角色：原发器（originator）、备忘录（memento）和看管人（caretaker）。其思想非常简单。原发器创建一个包含其状态的备忘录，并传给看管人。看管人不知如何与备忘录交互，但会把备忘录放在安全之处保管好。它们的静态关系如图23-1中的类图所示。

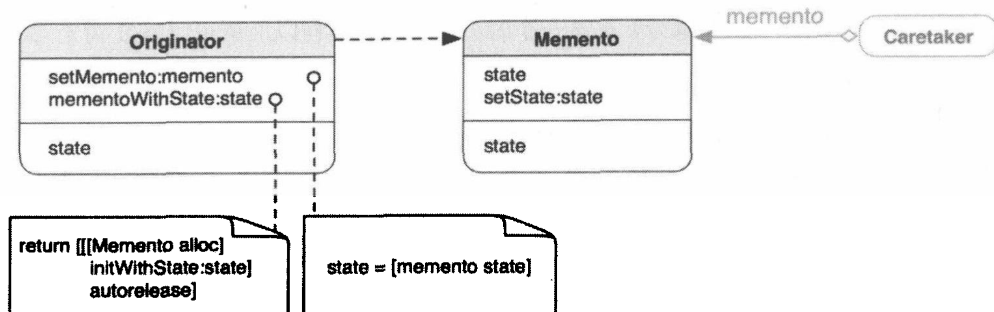


图23-1　备忘录模式结构的类图

当看管人请求Originator对象保存其状态时，Originator对象将使用其内部状态创建一个新的Memento实例。然后看管人保管Memento对象，或者把它保存到文件系统，一段时间之后再把它传回给Originator对象。Originator对象不知道这个Memento对象将如何被保存。看管人也不知道Memento对象里是什么。图23-2中的时序图解释了它们之间的交互过程。

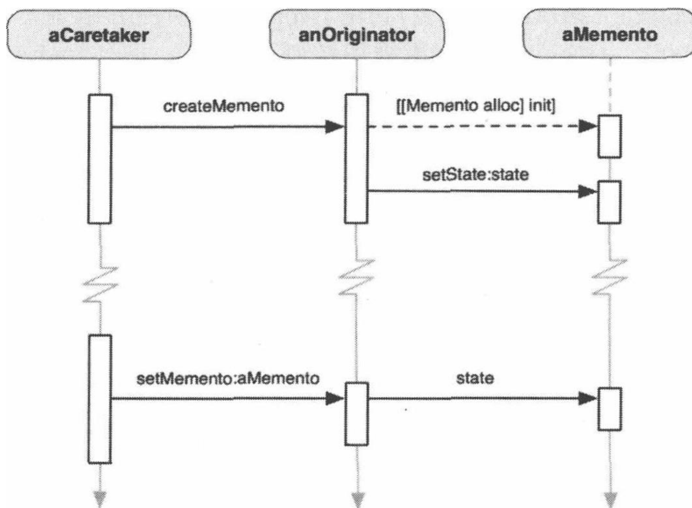


图23-2　备忘录模式的时序图

这个设计的关键是维持Memento对象的私有性，只让Originator对象访问保存在Memento对象中的内部状态（即Originator过去的内部状态）。Memento类应该有两个接口：一个宽接口，给Originator用；一个窄接口，给其他对象用。前面的图中，setState:、state和init方法应该定义为私有，不让Originator和Memento以外的对象使用。

23.2　何时使用备忘录模式

当同时满足以下两个条件时，应当考虑使用这一模式：

- 需要保存一个对象（或某部分）在某一个时刻的状态，这样以后就可以恢复到先前的状态；
- 用于获取状态的接口会暴露实现的细节，需要将其隐藏起来。

23.3 在 TouchPainter 中使用备忘录模式

希望读者还记得我们的小应用TouchPainter。它有个需要实现的功能是，把CanvasView上的当前涂鸦图保存到文件系统。这应该不难。只要用Mark组合对象作为参数，调用NSKeyedUnarchiver（将在23.4节讨论）的类方法archivedDataWithRootObject:，然后把返回的NSData对象保存到文件系统就行了。但有个问题：Mark对象如何知道应该把自己保存到文件系统中的什么地方（路径）呢？如果真的需要Mark知道这一点，就要回到第13章（组合）进行修改。每次对Mark作修改，所有它的实现类都要受到影响。而且，现在说的是保存整个Mark组合结构而不只是单个节点，此外还有其他一些问题，比如，如何保存组合体的一部分（即部分节点而不是全部）？组合体怎样才能知道应该把保存的节点放回到结构体的什么地方？下面几节将回答这些问题。

Mark组合对象只有基本操作，因此直接操作结构体并不简单明了。需要用另一个叫Scribble的类来从更高的层次管理它。第12章（观察者）讨论了如何在模型-视图-控制器范式中把Scribble用作模型，与CanvasViewController等视图控制器交互。Scribble对象包含一个反映CanvasView当前所绘图形状态的Mark实例结构。Scribble对象的作用不只是包含一个Mark实例，也把它的快照[①]保存为备忘录对象，并交由其他对象保存。快照备忘录本身可以是当时Scribble对象的完整结构，也可以是结构的一部分，作为对内部状态的变更。保存-恢复的操作只是在Scribble对象与它的备忘录对象之间发生，不涉及其他对象。Scribble对象不知道它的备忘录会被保存于何处。所以需要一个看管人对象，来保管和传递备忘录对象。能够担任这个角色的类可以是ScribbleManager，我们在第10章中跟外观模式一起讨论过它。它的作用主要是负责管理与Scribble对象的保存与恢复有关的杂务。虽然我们不会介绍ScribbleManager类的细节，但会简要讨论它在备忘录模式中所起的作用。

23.3.1 涂鸦图的保存

我们来回忆一下第2章中与涂鸦图的保存有关的几个需求。用户单击保存按钮时，CanvasViewController会对CanvasView上所画的图截屏并保存为一幅图像。然后它向ScribbleManager的实例发送一个消息，用刚刚得到的CanvasView的屏幕截图来保存当前Scribble对象。然后就是ScribbleManager的工作了。首先，它向Scribble对象发一个消息，创建包含其内部状态的快照。Scribble创建并返回ScribbleMemento之后，ScribbleManager可以选择把它保存到文件系统还是在内存中保存一段时间。

暂时只考虑把ScribbleMemento保存到文件系统的情况。ScribbleManager不能就这么原封不动地保存返回的ScribbleMemento对象。更方便的方式是让ScribbleMemento以一种

① snapshot，即在某个时刻的状态。——译者注

NSData的形式出现，这样ScribbleManager就能使用NSData对象的某些与文件相关的便利方法，把它保存到文件系统。

本章不会介绍保存缩略图之类处理的所有细节，但会集中讲解如何使用ScribbleManager对象的更为简单的方法saveScribble:来保存Scribble对象。提到的这些对象之间的交互的整体过程如图23-3中的时序图所示。

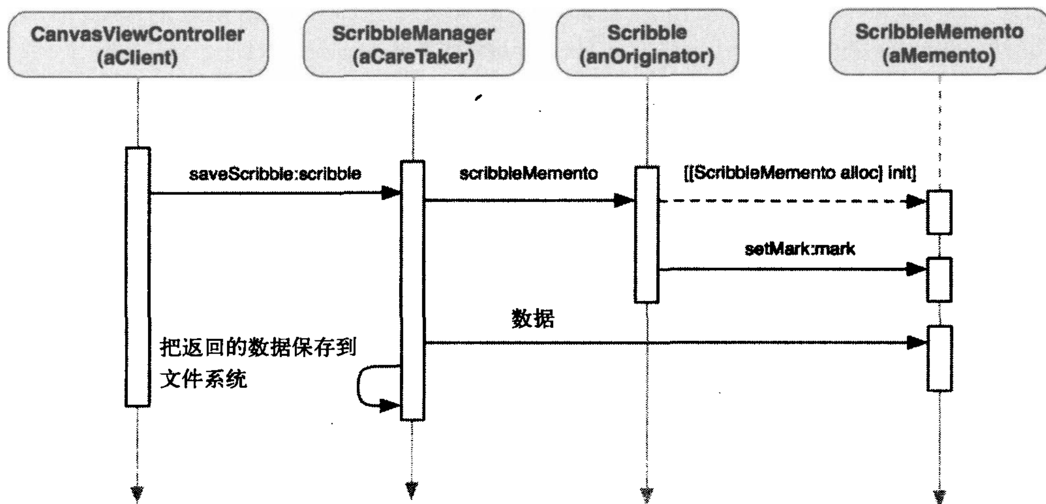


图23-3 TouchPainter应用中把Scribble对象保存为备忘录的时序图

CanvasViewController类型的客户端aClient，以要保存的Scribble对象为参数向ScribbleManager对象发送一个saveScribble:消息。然后在这个方法中，它让Scribble对象创建保存用的ScribbleMemento对象。读者可能会问，为什么ScribbleManager不直接要求Scribble传一个NSData对象给它，然后保存呢？仅仅直接使用NSData并不足以解决我们的问题，因为备忘录数据本身能够容纳有关Scribble对象内部状态的各种信息。有时只想让Scribble对象保存修改的内容。因此需要一个备忘录对象，它会告诉我们如何恢复Scribble，是只需要把保存的状态附加到Scribble中的当前状态，还是需要完全恢复。备忘录对象中保存的状态可以非常复杂（只要Scribble对象能明白就行）。而且内存中相同的ScribbleMemento对象可以重用于应用程序的其他地方，比如线条绘制的撤销与恢复，或者通过网络与其他用户共享以模拟远程绘图。尽管如此，如果只使用NSData对象进行受内存限制的状态保存，性能会受很大影响，因为归档-解档（archiving-unarchiving）处理需要额外的开销。

23.3.2 涂鸦图的恢复

刚才讨论了如何保存Scribble对象。读者可能想知道如何把它从文件系统取出来。

从前面几节我们看到，看管人最终会把先前保存的备忘录传回给原发器，以恢复到过去的某个状态。当然，不都是这样，有时可能不把备忘录传回其原发器，涂鸦图保存/加载策略就是这

样的一个典型例子。

从第2章我们知道，用户通过单击相应的缩略图来打开保存的涂鸦。调用这一过程的对象可以是一个按钮，或者是单独的命令对象（见命令模式，第20章）。作为客户端的调用者，invoker向ScribbleManager的实例发送一个消息，带有从（文件系统中的）文档加载和返回Scribble对象所需的所有信息。显然，调用者不知道要处理的涂鸦的保存位置和方式。

然后ScribbleManager会使用预定义的位置从文件系统中加载相应的NSData对象。ScribbleManager把数据传给ScribbleMemento的类方法mementoWithData:data，来生成ScribbleMemento的实例。此时，我们从这个ScribbleMemento对象创建一个全新的Scribble实例，而不是把它传回给Scribble原发器，因为我们认为原来的Scribble对象的生命周期应该在保存之前就已经结束。调用者在得到新的Scribble实例并恢复状态之后，会在应用程序中传递它，以备将来之用。使用复原的Scribble对象的一个特殊用例是，让CanvasViewController用它替换当前的Scribble对象（即打开保存的涂鸦图）。

所有提到的这些类和角色之间交互的时序图如图23-4所示。

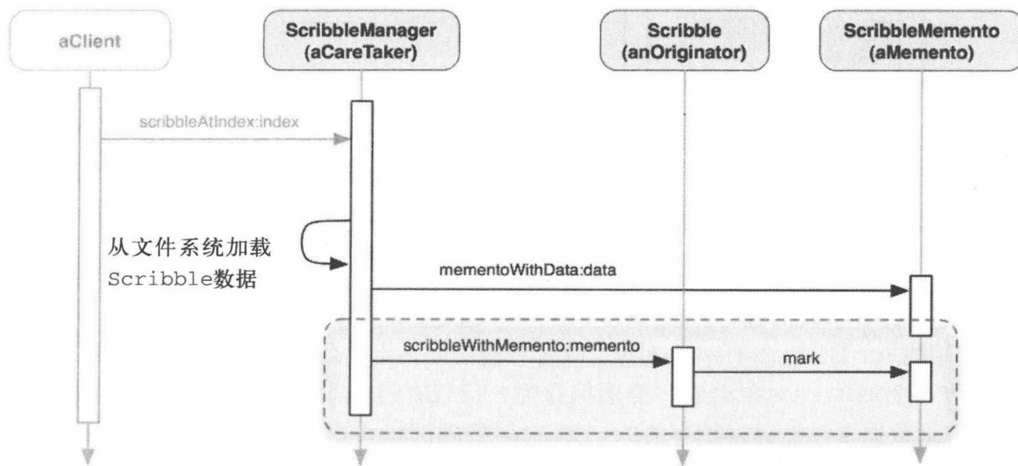


图23-4　TouchPainter应用中，以备忘录形式读取保存的Scribble对象的时序图

可以看出，图中阴影部分与图23-2中的原始备忘录模式时序图的对应部分很相似。不同的是，我们不把加载的ScribbleMemento对象传回给创建它的原始Scribble对象。也可以使用不同的操作，把来自ScribbleMemento对象的一个内部状态附加到现有的Scribble对象之上。稍后在本章的实现部分将详细讨论这个操作。

23.3.3　ScribbleMemento 的设计与实现

在第一次介绍备忘录模式的概念时，我们提到了备忘录对象应该对其原发器提供宽接口，而对看管人等其他对象提供窄接口。宽接口（跟窄接口相对）为对象的使用提供更多的选项与自由度。备忘录模式要求宽接口只应提供给原发器和备忘录。在C++等其他面向对象语言中，宽接口

一般用private（操作或构造函数）和friend进行声明。对本地方法或类使用friend指令，别的类就能像访问自己的私有资源一样访问它们。但Objective-C中一切都是公有的，所以需要另外的技巧来达到这个目的。

Objective-C中有个特有的功能叫范畴。通过范畴，开发者可以向已有的类添加额外的方法而不必进行子类化。如何使用范畴把宽接口变成私有呢？需要做的只是为ScribbleMemento创建一个叫ScribbleMemento (Private)的范畴。其他类如果得不到ScribbleMemento (Private)的接口的声明，这些接口就不会暴露出来。因此，可以把私有（或友元）操作放到这个范畴的声明中，这样他们就只对Scribble有效。ScribbleMemento的公有（窄）接口与ScribbleMemento (Private)的声明不在一起。在Objective-C 2.0中，可以使用匿名范畴，如ScribbleMemento()，来声明私有操作。这称为扩展（extension）。扩展与普通范畴之间的主要差别在于，扩展的操作应定义在类的主实现中，但普通范畴的实现可以写在别的文件中。同时，如果找不到扩展中声明的操作的实现，编译器就会发出警告。一般认为用扩展声明私有方法比用一般范畴要好，但并不是说原来的范畴功能已经废弃了，还需要用它对某些框架类进行扩展，因为我们无法修改其原始类文件。

图23-5中的类图解释了有关ScribbleMemento基本设计的思想。

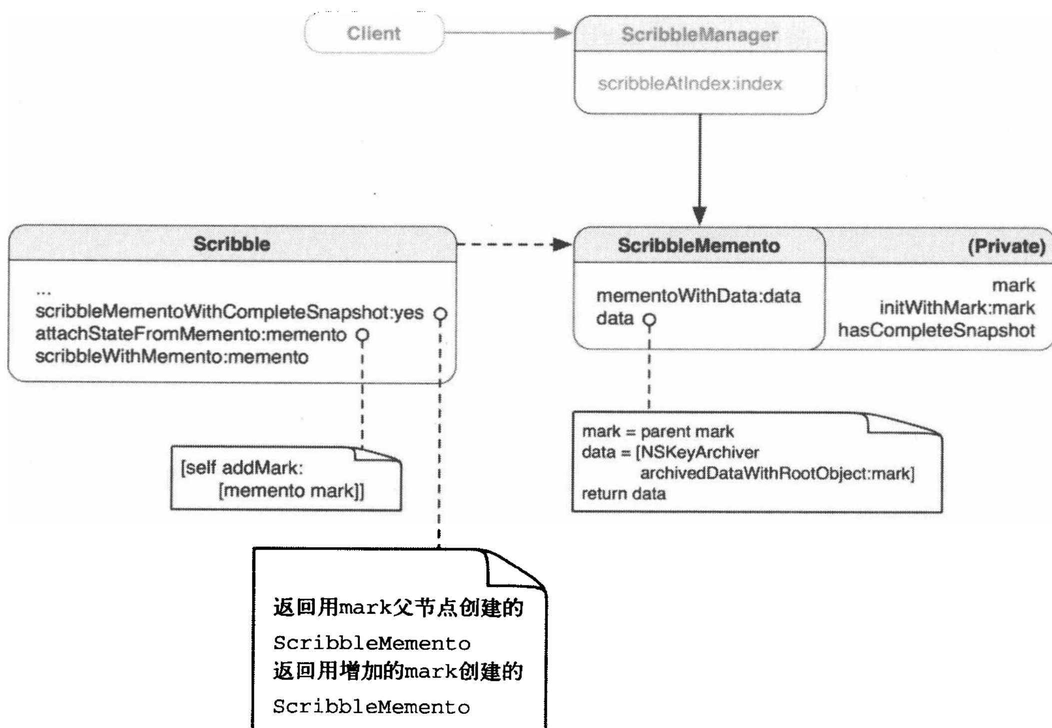


图23-5 ScribbleMemento及相关类的类图

ScribbleMemento使用mementoWithData:data和data方法提供窄接口的功能。其他类可

以使用mementoWithData:data方法，通过指定NSData对象来取得ScribbleMemento的实例。而data方法正相反：它以NSData返回归档形式的ScribbleMemento对象。其他类永远无法看到定义在私有范畴里的宽接口。

1. ScribbleMemento类的实现

来看看为以上ScribbleMemento设计而写的一些代码吧，如代码清单23-1所示。

代码清单23-1 ScribbleMemento.h中ScribbleMemento的类声明

```
#import "Mark.h"

@interface ScribbleMemento : NSObject
{
  @private
  id <Mark> mark_;
  BOOL hasCompleteSnapshot_;
}

+ (ScribbleMemento *) mementoWithData:(NSData *)data;
- (NSData *) data;

@end
```

ScribbleMemento保存一个私有Mark对象的引用，作为Scribble的内部状态。它还有个BOOL型私有成员变量hasCompleteSnapshot_，它将告诉Scribble对象，保存的Mark引用是完整的快照还是只是个片段。窄接口操作的声明也在这里。稍后将在其实现部分介绍这些操作。

它的扩展（私有范畴）声明了只供Scribble对象使用的宽接口，如代码清单23-2所示。

代码清单23-2 声明在ScribbleMemento+Friend.h中的ScribbleMemento的私有范畴

```
#import "Mark.h"
#import"ScribbleMemento.h"

@interface ScribbleMemento ()

- (id) initWithMark:(id <Mark>)aMark;

@property (nonatomic, copy) id <Mark> mark;
@property (nonatomic, assign) BOOL hasCompleteSnapshot;

@end
```

Scribble对象可以创建一个ScribbleMemento对象并使用自己的内部Mark引用将其初始化。Scribble对象也可以访问ScribbleMemento的mark和hasCompleteSnapshot属性。

其实现如代码清单23-3所示。

代码清单23-3 ScribbleMemento.m中ScribbleMemento的实现

```
#import "ScribbleMemento.h"
#import "ScribbleMemento+Friend.h"

@implementation ScribbleMemento
```

```
@synthesize mark=mark_;
@synthesize hasCompleteSnapshot=hasCompleteSnapshot_;

- (NSData *) data
{
  NSData *data = [NSKeyedArchiver archivedDataWithRootObject:mark_];
  return data;
}

+ (ScribbleMemento *) mementoWithData:(NSData *)data
{
  // 如果data不是有效的文档，就引发NSInvalidArchiveOperationException异常
  id <Mark>retoredMark = (id <Mark>)[NSKeyedUnarchiver unarchiveObjectWithData:data];
  ScribbleMemento *memento = [[[ScribbleMemento alloc]
                               initWithMark:retoredMark] autorelease];

  return memento;
}

- (void) dealloc
{
  [mark_ release];
  [super dealloc];
}

#pragma mark -
#pragma mark Private methods

- (id) initWithMark:(id <Mark>)aMark
{
  if (self = [super init])
  {
    [self setMark:aMark];
  }

  return self;
}

@end
```

实例方法data向NSKeyedArchiver类发送archivedDataWithRootObject:self消息，得到self经过编码之后的版本，然后将它返回。类方法mementoWithData:(NSData *)data用指定的NSData对象创建新的ScribbleMemento实例。data首先由NSKeyedUnarchiver解码回ScribbleMemento的实例，然后返回这个实例。

在ScribbleMemento的主实现部分，使用@synchronize合成了mark与hasCompleteSnapshot属性。这么做是因为Objective-C不允许在范畴中合成属性。同时编译器要求扩展的实现一定要放在主@implementation块之中。因为属性在实现中用@synthesize做了合成，所以这不会破坏ScribbleMemento的扩展的私有性。

除私有属性外，我们还把initWithMark:方法的定义也放在了这里的主实现中。

值得一提的是，mark属性实际上是在复制另一个Mark对象，而不是保持（retain）它。为什

么是这样呢？因为如果只是保持它，而原来的Mark对象又在程序的其他部分被修改了，那么我们获取的ScribbleMemento对象的状态就完蛋了。代码中copy和retain的区别似乎不大，但这种区别的影响可能无法估量。这就像在属性中复制NSString的实例而不是只保持它，因为字符串有可能在类之外被修改。

2. 修改Scribble类

我们已经实现了ScribbleMemento类。为了让Scribble和ScribbleMemento能用在一起，需要对Scribble类作些修改，如代码清单23-4所示。

代码清单23-4　Scribble.h中Scribble的修改后的类声明

```
#import "Mark.h"
#import "ScribbleMemento.h"

@interface Scribble : NSObject
{
  @private
  id <Mark> parentMark_;
  id <Mark> incrementalMark_;
}

// 管理Mark的方法
- (void) addMark:(id <Mark>)aMark shouldAddToPreviousMark:(BOOL)shouldAddToPreviousMark;
- (void) removeMark:(id <Mark>)aMark;

// 备忘录用的方法
- (id) initWithMemento:(ScribbleMemento *)aMemento;
+ (Scribble *) scribbleWithMemento:(ScribbleMemento *)aMemento;
- (ScribbleMemento *) scribbleMemento;
- (ScribbleMemento *) scribbleMementoWithCompleteSnapshot:(BOOL)hasCompleteSnapshot;
- (void) attachStateFromMemento:(ScribbleMemento *)memento;

@end
```

我们向Scribble添加了另一个Mark引用incrementalMark_，用于保存添加到parentMark_的完整线条或点的引用。

有几个只给ScribbleMemento对象用的方法。前面几节在模式的实现架构设计中已经用过了其中几个。我们将对Scribble实现的修订版中的这些方法作详细讨论，如代码清单23-5所示。代码相当长，所以将把它分成几段来讨论。

代码清单23-5　Scribble.m中经过修改的Scribble实现

```
#import "ScribbleMemento+Friend.h"
#import "Scribble.h"
#import "Stroke.h"

// Scribble的私有范畴
// 它有一个只供Scribble对象访问的mark属性
@interface Scribble ()
```

```
@property (nonatomic, retain) id <Mark> mark;

@end

@implementation Scribble

@synthesize mark=parentMark_;

- (id) init
{
  if (self = [super init])
  {
    // 父节点应该是个组合对象（即Stroke）
    parentMark_ = [[Stroke alloc] init];
  }
  return self;
}
```

我们导入代码清单23-2中定义的私有范畴，通过它可以在Scribble里创建ScribbleMemento对象。Scribble对象有一个对Mark对象的引用，它是Mark组合结构的根节点。这个Mark属性是Scribble对象的内部状态，而且只有自己能看得到。

Scribble 类中有关Mark对象的操作的详细讨论，请参阅第12章（观察者）。

```
#pragma mark -
#pragma mark Methods for Mark management

// 与Mark管理有关的方法的细节，请参阅第12章

-(void)addMark:(id <Mark>)aMark shouldAddToPreviousMark:(BOOL)shouldAddToPreviousMark
{
  // 手工调用KVO
  [self willChangeValueForKey:@"mark"];

  // 如果标志设为YES
  // 就把这个aMark加到前一个Mark作为聚合体的一部分
  // 根据我们的设计，它应该是根节点的最后一个子节点
  if (shouldAddToPreviousMark)
  {
    [[parentMark_ lastChild] addMark:aMark];
  }
  // 否则把它附加到父节点
  else
  {
    [parentMark_ addMark:aMark];
    incrementalMark_ = aMark;
  }

  // 手工调用KVO
  [self didChangeValueForKey:@"mark"];
}

- (void) removeMark:(id <Mark>)aMark
{
  // 如果aMark是父节点则什么也不做
```

23

```
    if (aMark == parentMark_) return;

    // 手工调用KVO
    [self willChangeValueForKey:@"mark"];

    [parentMark_ removeMark:aMark];

    // 不需要保存incrementalMark_引用，因为它刚从父节点删除
    if (aMark == incrementalMark_)
    {
      incrementalMark_ = nil;
    }

    // 手工调用KVO
    [self didChangeValueForKey:@"mark"];
  }
```

我们向addMark:shouldAddToPreviousMark:方法添加了一条语句，当aMark被直接附加到mark根节点（即aMark为线条或点）时，把aMark保存到incrementalMark_。

如果我们删除的Mark对象同时也被incrementalMark_引用，那么删除之后需要把incrementalMark设置为nil。否则，再次使用它将会导致整个程序崩溃。

```
#pragma mark -
#pragma mark Methods for memento

- (id) initWithMemento:(ScribbleMemento*)aMemento
{
  if (self = [super init])
  {
    if ([aMemento hasCompleteSnapshot])
    {
      [self setMark:[aMemento mark]];
    }
    else
    {
      // 如果备忘录中只包含一个增量的mark，那就需要创建容纳它的父节点
      parentMark_ = [[Stroke alloc] init];
      [self attachStateFromMemento:aMemento];
    }
  }

  return self;
}
```

Scribble对象可以使用一个ScribbleMemento对象来初始化，Scribble对象可以访问这个ScribbleMemento对象的私有mark属性，恢复自己的状态。

```
- (void) attachStateFromMemento:(ScribbleMemento *)memento
{
  // 把来自备忘录对象的mark附加到根节点
  [self addMark:[memento mark] shouldAddToPreviousMark:NO];
}
```

attachStateFromMemento:方法能够让Scribble对象添加来自ScribbleMemento对象

的任何Mark对象。由于我们的设计只处理先前附加到根节点的增量Mark对象，这里也把备忘录的Mark对象添加到根节点。

```
- (ScribbleMemento *) scribbleMementoWithCompleteSnapshot:(BOOL)hasCompleteSnapshot
{
  id <Mark> mementoMark = incrementalMark_;

  // 如果要求返回完整的快照，就把它设为parentMark_
  if (hasCompleteSnapshot)
  {
    mementoMark = parentMark_;
  }
  // 但如果incrementalMark_是nil，我们什么也做不了，只好退出
  else if (mementoMark == nil)
  {
    return nil;
  }

  ScribbleMemento *memento = [[[ScribbleMemento alloc]
                               initWithMark:mementoMark] autorelease];
  [memento setHasCompleteSnapshot:hasCompleteSnapshot];

  return memento;
}

- (ScribbleMemento *) scribbleMemento
{
  return [self scribbleMementoWithCompleteSnapshot:YES];
}

+ (Scribble *) scribbleWithMemento:(ScribbleMemento *)aMemento
{
  Scribble *scribble = [[[Scribble alloc] initWithMemento:aMemento]autorelease];
  return scribble;
}

- (void) dealloc
{
  [parentMark_ release];
  [super dealloc];
}

@end
```

在scribbleMementoWithCompleteSnapshot:方法中，如果hasCompleteSnapshot参数值为YES，则使用parentMark_创建ScribbleMemento实例，否则使用incrementalMark_。

scribbleMemento是个便利方法，它把BOOL型参数设为YES来调用scribbleMementoWithCompleteSnapshot:方法，创建并返回一个包含当前状态完整快照的ScribbleMemento对象。

另一个方法是scribbleWithMemento:，这个类方法用ScribbleMemento对象创建新的Scribble对象。

练　　习

大家已经注意到，我们使用incrementalMark_保存的是添加到Scribble对象的parentMark_的特定Mark对象的引用。Scribble对象可以使用incrementalMark_创建一个ScribbleMemento对象，用来保存添加到其内部状态的特定修改。怎样修改前面例子的实现，才能为Scribble对象提供另一种选择，让它也能保存从父节点删除的Mark对象呢？

3. 使用看管人把一切联系起来

一切都准备好了，就差看管人管理ScribbleMemento对象这个部分了。先前我们假定ScribbleManager可以担当看管人的角色。它有个叫saveScribble:的简单操作，让客户端能保存Scribble对象。它还有个scribbleAtIndex:方法，该方法用索引确定某个Scribble对象，ScribbleManager对象会加载它并将其返回。

来看看saveScribble:方法的简化实现代码，看看它是如何捕捉Scribble对象的内部状态并保存到文件系统的。请看代码清单23-6。

代码清单23-6　把ScribbleMemento对象保存到文件系统的saveScribble:的简化代码

```
// 从涂鸦取得备忘录
ScribbleMemento *scribbleMemento = [scribble scribbleMemento];

// 从备忘录取得NSData对象，
// 以便保存到文件系统
NSData *mementoData = [scribbleMemento data];

NSString *mementoPath;
// ……
// 构建保存备忘录用的路径
// 并在实际保存到文件系统之前执行其他任何操作
// ……
[mementoData writeToFile:mementoPath atomically:YES];
```

我们先让Scribble对象通过其scribbleMemento方法生成一个ScribbleMemento实例。但我们不能原封不动地把这个ScribbleMemento对象保存到文件系统。因此需要让它调用ScribbleMemento对象的data消息，把自己包装成NSData对象。然后使用文件系统中的一个路径保存这个NSData对象。

恢复操作跟保存非常相似，如代码清单23-7所示。同样，它也是scribbleAtIndex:方法的一个简化版本。

代码清单23-7　scribbleAtIndex:的简化代码，它从文件系统取出一个ScribbleMemento对象，然后把它恢复成一个Scribble对象

```
NSString *scribbleMementoPath;
// ……
// 使用指定的索引取出以前保存备忘录用的路径
// 以后会用这个路径加载备忘录
// ……
```

```
// 用刚刚重建的路径，使用NSFileManager把备忘录文件加载成NSData
NSFileManager *fileManager = [NSFileManager defaultManager];
NSData *scribbleMementoData = [fileManager contentsAtPath:scribbleMementoPath];

// 从这个NSData对象创建一个ScribbleMemento
// 然后使用这个备忘录，根据其中所保存的内容来恢复Scribble对象
ScribbleMemento *scribbleMemento = [ScribbleMemento
                                    mementoWithData:scribbleMementoData];
Scribble *resurrectedScribble = [Scribble scribbleWithMemento:scribbleMemento];
```

首先用指定的索引来定位文件系统中所保存的数据文件的路径。然后加载归档文件得到原来的NSData对象，并使用这个数据对象用类消息mementoWithData:创建一个ScribbleMemento对象。最后使用ScribbleMemento对象通过scribbleWithMemento:来恢复一个Scribble对象。

23.4 Cocoa Touch 框架中的备忘录模式

Cocoa Touch框架在归档、属性列表序列化和核心数据中采用了备忘录模式。属性列表序列化和核心数据不在本书的讨论范围。我们只讨论归档，并简要介绍其关键功能以及如何将其应用于前面几节中的例子。

Cocoa的归档是对对象及其属性还有同其他对象间的关系进行编码，形成一个文档，该文档既可保存于文件系统，也可在进程或网络间传送。对象与其他对象的关系被看做对象图的网络。归档过程把对象图保存为一种与架构无关的字节流，保持对象的标识以及对象之间的关系。对象的类型也同数据一起保存。从字节流解码出来的对象通常用与对象编码时相同的类进行实例化。

如果想归档一个对象，很多时候我们是在考虑保存程序的状态。在模型-视图-控制器范式中，程序的状态通常由模型对象来维护。我们把模型对象编码到文档，然后再对其解码读回来。在运行时使用NSCoder对象进行编码与解码操作。NSCoder本身是个抽象类。苹果公司建议通过NSCoder的具体类NSKeyedArchiver和NSKeyedUnarchiver，使用基于键的归档技术。被编码与解码的对象必须遵守NSCoding协议并实现以下方法：

```
- (id)initWithCoder:(NSCoder *)coder;
- (void)encodeWithCoder:(NSCoder *)coder;
```

这两个方法与归档和解档过程中对象的编码与解码有关。稍后将讨论这些方法。

在ScribbleMemento的例子中，用NSKeyedArchiver和NSKeyedUnarchiver类实现了归档和解档过程。被编码的对象是一个Mark组合对象。读者可以回到第13章，阅读对Mark及其组合对象的操作有关的讨论。要让NSKeyedArchiver和NSKeyedUnarchiver很好地工作，需要保证所有Mark类都遵守NSCoding协议并实现必需的方法。首先，需要让Mark协议采用NSCoding，这样其他类也就能遵守它，如代码清单23-8所示。

代码清单23-8 Mark实现NSCoding协议

```
@protocol Mark <NSObject, NSCopying, NSCoding>
```

```
@property (nonatomic, retain) UIColor *color;
@property (nonatomic, assign) CGFloat size;
@property (nonatomic, assign) CGPoint location;
@property (nonatomic, readonly) NSUInteger count;
@property (nonatomic, readonly) id <Mark> lastChild;

- (id) copy;
- (void) addMark:(id <Mark>) mark;
- (void) removeMark:(id <Mark>) mark;
- (id <Mark>) childMarkAtIndex:(NSUInteger) index;

// 一些在其他章节中定义的方法

@end
```

这样所有Mark的实现类只要实现了必需的方法，就遵守了NSCoding协议。来看看Vertex对这些方法的实现，如代码清单23-9所示。

代码清单23-9 Vertex中NSCoding方法的实现

```
- (id)initWithCoder:(NSCoder *)coder
{
  if (self = [super init])
  {
    location_=[(NSValue *)[coder decodeObjectForKey:@"VertexLocation"] CGPointValue];
  }

  return self;
}

- (void)encodeWithCoder:(NSCoder *)coder
{
  [coder encodeObject:[NSValue valueWithCGPoint:location_]orKey:@"VertexLocation"];
}
```

coder在运行时由归档器或解档器提供，我们需要通过coder来指示这些方法如何使用相同的键编码与解码对象。对于Vertex，当encodeWithCoder:方法在运行时被调用以保存其对象时，它发送一个encodeObject:消息，用键@"VertexLocation"对其location_实现进行编码。在initWithCoder:方法中，它使用同样的键，向解档器发送decodeObjectForKey:消息，来解码这个属性。

Dot的步骤几乎相同，只是其对象多了几个要处理的属性，如代码清单23-10所示。

代码清单23-10 Dot中NSCoding方法的实现

```
- (id)initWithCoder:(NSCoder *)coder
{
  if (self = [super initWithCoder:coder])
  {
    color_ = [[coder decodeObjectForKey:@"DotColor"] retain];
    size_ = [coder decodeFloatForKey:@"DotSize"];
  }
```

```
  return self;
}

- (void)encodeWithCoder:(NSCoder *)coder
{
  [super encodeWithCoder:coder];
  [coder encodeObject:color_ forKey:@"DotColor"];
  [coder encodeFloat:size_ forKey:@"DotSize"];
}
```

Vertex的initWithCoder:和encodeWithCoder:方法没有把同样的消息转发给super，这是因为Vertex是NSObject的直接子类，而NSObject没有实现这些方法。

Stroke的实现与Vertex和Dot非常相似，如代码清单23-11所示。

代码清单23-11 Stroke中NSCoding方法的实现

```
- (id)initWithCoder:(NSCoder *)coder
{
  if (self = [super init])
  {
    color_ = [[coder decodeObjectForKey:@"StrokeColor"] retain];
    size_ = [coder decodeFloatForKey:@"StrokeSize"];
    children_ = [[coder decodeObjectForKey:@"StrokeChildren"] retain];
  }

  return self;
}

- (void)encodeWithCoder:(NSCoder *)coder
{
  [coder encodeObject:color_ forKey:@"StrokeColor"];
  [coder encodeFloat:size_ forKey:@"StrokeSize"];
  [coder encodeObject:children_ forKey:@"StrokeChildren"];
}
```

Stroke对象的子节点也可以被递归编码。编码过程中我们只是把整个children属性扔给coder，而在initWithCoder:方法中，我们可以从一个解档器的coder把它整个取回来。

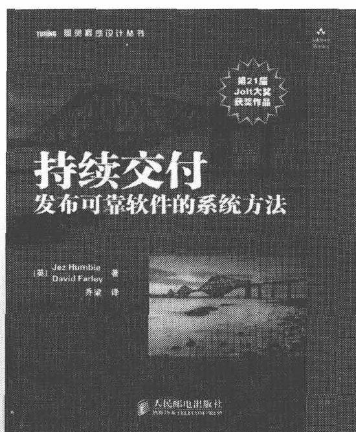
23.5 总结

本章讨论了备忘录模式的概念，并通过TouchPainter示例应用，示范了如何将这一模式应用到iOS应用的开发中。通过保存对Scribble对象内部状态的小修改的一个列表，ScribbleMemento也能够被复用于实现撤销与恢复操作。只保存了对Scribble的小修改的ScribbleMemento对象列表，可用于在网络上共享线条。这样共享相同绘图会话的远程用户，就能看到模拟的远程绘图，一次画一条线。

到这里本书中介绍设计模式的所有章节就都结束了。现在，读者应该能够在实际项目中自由应用这些设计模式了。我真心希望这些设计模式能在诸多方面对读者软件设计的实践有所帮助。

TURING 图灵教育

图灵最新重点图书

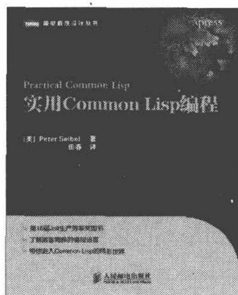


书　名：持续交付：发布可靠软件的系统方法
书　号：978-7-115-26459-6
作　者：Jez Humble　David Farley
译　者：乔梁
定　价：89.00 元

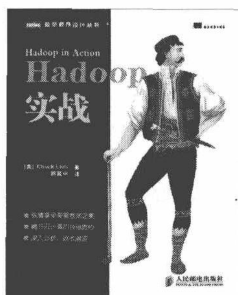
- **第 21 届 Jolt 大奖获奖作品**
- **Martin Fowler 作序推荐**
- **软件构建、部署、测试和发布的必备手册**
- **改善负责软件交付相关人员的协作**
- **项目中极为重要的度量项**
- **持续交付有价值的软件让客户满意**

本书讲述如何实现更快、更可靠、低成本的自动化软件交付，描述了如何通过增加反馈，并改进开发人员、测试人员、运维人员和项目经理之间的协作来达到这个目标。本书由三部分组成。第一部分阐述了持续交付背后的一些原则，以及支持这些原则的实践。第二部分是本书的核心，全面讲述了部署流水线。第三部分围绕部署流水线的投入产出讨论了更多细节，包括增量开发技术、高级版本控制模式，以及基础设施、环境和数据的管理和组织治理。

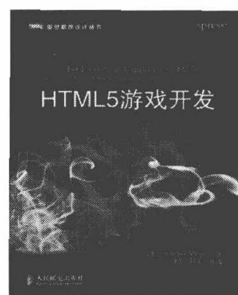
本书适合所有开发人员、测试人员、运维人员和项目经理学习参考。



实用 Common Lisp 编程
书号：978-7-115-26374-2
定价：89.00 元

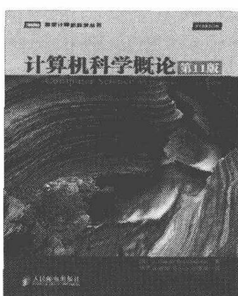


Hadoop 实战
书号：978-7-115-26448-0
定价：59.00 元



HTML5 游戏开发
书号：978-7-115-26363-6
定价：49.00 元

认知与设计：理解 UI 设计准则
书号：978-7-115-26141-0
定价：59.00 元



计算机科学概论（第 11 版）
书号：978-7-115-26196-0
定价：69.00 元

SQL 反模式
书号：978-7-115-26127-4
定价：59.00 元

[General Information]
书名=OBJECTIVE-C编程之道 IOS设计模式解析
作者=
页数=1000
SS号=12865932
出版日期=
出版社=