Cocoa 基本原理指南的介绍

本部分包含如下内容：

对于刚刚加入这个阵营的开发者来说，Cocoa 像是一个巨大而未知的新世界。Cocoa 开发环境的各种特性、工具、概念、术语、编程接口、甚至是编程语言对他们来说可能都比较生疏。*Cocoa 基本原理指南*提供了领略 Cocoa 技术景致的方向，介绍 Cocoa 的特性、基本概念、专用术语、结构、以及潜在的设计模式，使开发者更加容易上手。

*Cocoa 基本原理指南*的组织结构使读者可以逐步获取有关 Cocoa 开发的一般性知识。它从最基础的信息开始—即 Cocoa 有什么组件和能力，以考察 Cocoa 的主要架构作为结束。每一章都建立在前一章内容的基础上，每个部分都给出某个主题的重要细节，但又仅仅在较高的级别上进行描述。文中还提供很多指引，方便读者了解其它更为详尽的文档。

在 Cocoa 的开发文档中，*Cocoa 基本原理指南*是介绍 Cocoa 概念的初级文档，是诸如 *Cocoa 描画指南*和 *Cocoa 的视编程指南*这类文档的先期读物。阅读 *Cocoa 基本原理指南*需要的前提条件很少，但读者必须是熟练的 C 程序员，且应该熟悉 Mac OS X 的能力和技术。您可以通过阅读 *Mac OS X 技术概览*一书来获得这些知识。

本文的组织

*Cocoa 基本原理指南*有如下几个章节：

1. "什么是Cocoa？" 从功能和大体架构的角度介绍什么是Cocoa，描述它的各种特性、框架、和开发环境。
2. "Cocoa对象" 解释Objective-C的基本用法和优点，以及Cocoa对象的常见行为、接口、和生命周期。
3. "为Cocoa程序添加行为" 描述如何使用Cocoa框架来编写程序，解释如何创建一个子类。
4. "Cocoa的设计模式" 描述Cocoa采纳的设计模式，特别是模型-视-控制器对象模型。
5. "和对象进行通讯" 讨论Cocoa对象之间的通讯机制和编程接口，包括委托、通告、和绑定技术。
6. "核心应用程序架构" 考察应用程序对象之间的关系，Cocoa通过这些对象来进行描画和事件处理。
7. "其它Cocoa架构" 总结Cocoa支持应用程序开发和扩展应用程序能力的主要架构。

相关信息

您可以在技术书店里找到几个介绍 Cocoa 的优秀读物，用以补充 *Cocoa 基本原理指南*一书中的知识。此外，在开始成为 Cocoa 开发者之前，您还应该阅读一些苹果公司出版的其它资料：

- *Objective-C 编程语言* 描述 Objective-C 编程语言和运行环境。
- *Cocoa 应用程序教程* 向您演示如何用 Xcode 开发环境、Cocoa 框架、以及用 Objective-C 创建一个简单的 Cocoa 应用程序。
- *模型对象实现指南* 讨论子类设计和实现的基本问题。

Cocoa 的环境

Cocoa 应用程序正逐渐成为 Mac OS X 的应用程序标准。iPhoto、Safari、和 Mail 都是 Cocoa 应用程序。这些应用程序由于聪明的设计、丰富的功能、和激动人心的用户界面而受到了相当程度的好评。但是，对于一般用户来说并不明显（和典型的开发周期相比）的是：这些程序从设计阶段到最终部署的过程是多么的快速。作为应用程序开发环境，是什么使 Cocoa 成为比 Carbon 切实可行、甚至是强制性的替代呢？

本部分包含如下内容：

介绍 Cocoa

和所有的应用程序环境一样，Cocoa 包括两个方面：即运行环境方面和开发方面。在运行环境方面，Cocoa 应用程序呈现 Aqua 用户界面，且和操作系统的其它可视部分紧密集成，这些部分包括 Finder、Dock、和基于所有环境的其它应用程序。Cocoa 无缝地成为了用户体验的一部分，在运行环境方面表现优秀。

但是，程序员更感兴趣的是开发方面。Cocoa 是一个面向对象的软件组件—类—的集成套件，它使开发者可以快速创建强壮和全功能的 Mac OS X 应用程序。这些类是可复用和可支配的软件积木，开发者可以直接使用，或者根据具体需求对其进行扩展。从用户界面对象到 Bonjour 网络，几乎每个想象得到的开发需求都存在对应的 Cocoa 类；对于没有预想到的需求，您可以轻松地从现有类派生出子类来实现。

在各种面向对象的开发环境中，Cocoa有着最为著名的血统。从 1989 年作为NeXTSTEP推出到现在，人们一直对它进行精化和测试（参见 "一点历史"部分）。它优雅而强大的设计完美地适合所有类型的快速软件开发：不仅适合开发应用程序，也适合开发命令行工具、插件、和不同类型的程序包。Cocoa为您的应用程序“免费”提供很多行为和外观，使您可以将更多的时间用于有特色的功能上（有关Cocoa提供的功能的详细信息，请参见 "Cocoa应用程序的特性"部分）。
在开发 Cocoa 软件的时候，您可以使用多种编程语言。基本的语言是 Objective-C。Objective-C 拥有自己的 Cocoa 运行环境，是 ANSI C 的超集，它在 ANSI C 的语法和语义特性上（从 Smalltalk 派生而来）进行

扩展，使之支持面向对象的编程。新增的规则简单而又易于学习和使用。由于 Objective-C 是基于 ANSI C 的，您可以自由地将 C 代码直接和 Objective-C 代码混合在一起。而且，您的代码可以调用非 Cocoa 的编程接口中定义的所有函数，比如 Carbon 和 BSD。您甚至可以将 C++代码混合到 Cocoa 代码中，并将它们连接在同一个执行文件中。最后，Cocoa 支持 Java。Cocoa 为此定义了一个平行的 Java 类库，并且实现了一个将 Java 接口映射到 Objective-C 实现的桥机制。Cocoa 的 Java 支持使您可以将本地的 Java 对象和 Cocoa 对象混合在一起使用（在某些限制下）。

重要信息：Cocoa-Java 是熟悉 Java 语言的开发者的学习环境，我们并不推荐将它用于产品开发。Objective-C API 会不断进化，而 Cocoa-Java API 并不并行维护。

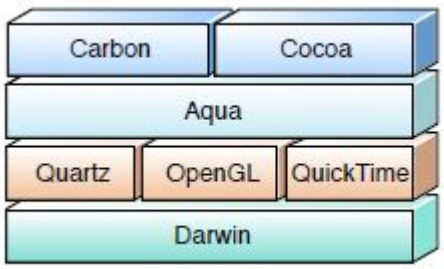
您甚至可以用PyObjC，即Python/Objective-C桥来进行Cocoa编程。基于PyObjC，您可以用Python来书写Cocoa程序。Python是一种解释性的、注重交互的、及面向对象的编程语言。PyObjC使Python对象可以向Objective-C对象传递消息，就象传递给Python对象一样；同时还使Objective-C对象可以向Python对象传递消息。更多信息请参见 "用Python开发基于PyObjC的Cocoa应用程序"文档，它位于苹果开发者联盟（Apple Developer Connection）网站上。

核心的Cocoa类库封装在两个框架中，即Foundation和Application Kit框架。和所有框架一样，这两个框架不仅包含动态共享库（有时是几个兼容版本的库），还包含头文件、API文档、和相关的资源。Application Kit和Foundation框架的分割反映了Cocoa编程接口分为图形用户界面部分和非图形接口。这两个框架对于最终产品为应用程序的Cocoa工程来说都是必要的。还有几个较小的、使用Cocoa编程接口的框架和Mac OS X一起发行，比如Screen Saver（屏幕保护）和Address Book（地址簿）框架。随着时间的推移，还会有更多框架加入到操作系统中。更多信息请参见 "Cocoa框架"部分。

Cocoa 在 Mac OS X 中的位置

图 1-1 显示了一个简化了的 Mac OS X 系统架构框图：

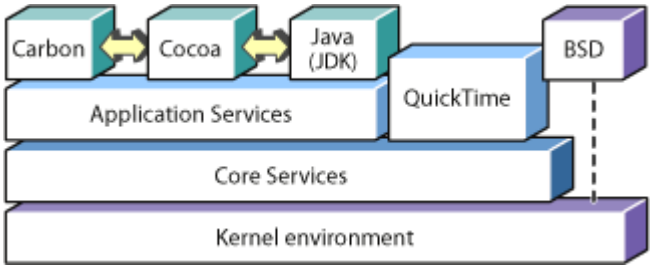
图 1-1 Mac OS X 架构—简化视图



这个框图只是为了一个简单的目的：为那些不熟悉苹果平台的开发者明确指出 Mac OS X 的主要组件及其依赖性。为了简洁，图中省略了一些重要细节，并使其它部分变得模糊。这些细节构成了框图的重要部分，显示 Cocoa 和 Mac OS X 其它部分的关系。

图 1-2 在架构级别上更为精确地反映了 Cocoa 的位置。这个框图将 Mac OS X 显示为一系列的软件层，从系统的基础 Darwin 到各种应用程序环境。位于中间的层代表包含在 Core Services（核心服务）和 Application Services（应用程序服务）这两个主要的雨伞框架下的系统软件。在这个框图中，一个层通常依赖于其下面的其它层。

图 1-2 Cocoa 在 Mac OS X 架构中的位置

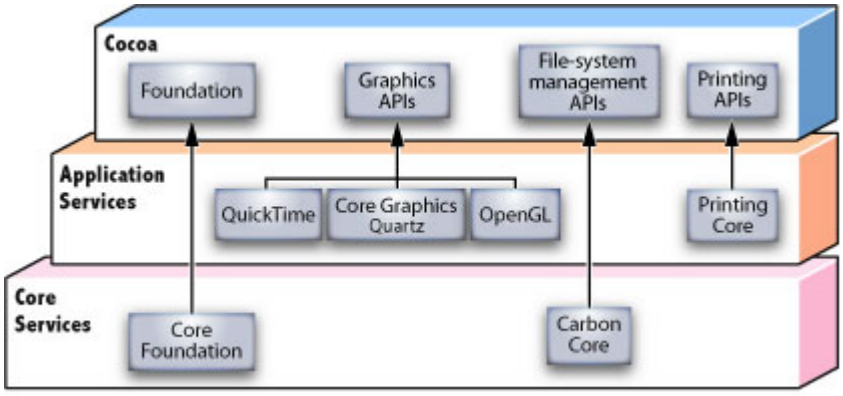


在某些方面，这个框图类似于先前的框图。举例来说，主要负责 Aqua 用户界面渲染的系统组件 Quartz（在 Core Graphics 框架中实现)是 Application Services 层的一部分。架构栈的基础部分是 Darwin，包括 Cocoa 在内的 Mac OS X 各个部分最终都依赖于 Darwin。

但是，如果您进一步查看雨伞框架中的一个（或一组）Cocoa 子类或特定的子框架，就会发现 Cocoa 或者对 Mac OS X 其它部分有特定的依赖性，或者通过自身的接口向外部提供基础的技术。图 1-3 部分显示了上述的依赖性和外部接口。

请注意：虽然 Cocoa 依赖于特定的框架，但它并不是仅仅“坐”在这些框架的上面。在某些情况下，Cocoa 和其它框架（比如 Carbon）是对等的，甚至可以实现一些其它对等框架不能实现的任务。Cocoa 不仅仅是基础技术上面的一个面向对象的封装层。

图 1-3 进一步考察 Cocoa 架构—一些主要的依赖关系



苹果公司对 Cocoa 进行了认真的设计，使 Cocoa 编程接口成为应用程序通常需要的基础技术访问通道。但是如果您需要的某些能力不能在 Cocoa 的接口中找到，或者需要对应用程序进行更为精细的控制，那么也可以直接使用底层的框架（Core Graphics 就是一个重要的例子，通过调用该框架或 OpenGL 的函数，您的代码可以画出比 Cocoa 描画方法能做到的、更加复杂而具有细微差别的图像）。幸运的是，使用这些低级别的框架并不是问题，因为绝大多数依赖框架的编程接口是用 ANSI C 写的，Objective-C 是其超集。

请注意：架构概述部分的目的并不是列举出 Cocoa 有哪些接口或者它对 Mac OS X 其它部分有哪些依赖性。相反，概述部分只是考虑最有趣的部分，目的是给您一个有关框架架构的基本思想。

Cocoa 依赖的、或者通过类和方法为之提供访问通道的主要基础框架有 Core Foundation，Carbon，Core Graphics (Quartz)，Launch Services，和 Print Core (打印子系统)。详细信息如下：

- **Core Foundation**。Foundation 框架的很多类都基于 Core Foundation 中对应的封装类型。它们之间的这种紧密关系使"免费桥接"技术—即在兼容的 Core Foundation 和 Foundation 类型之间实现类型转换—成为可能。某些 Core Foundation 的实现又基于 Darwin 层的 BSD 部分。
- **Carbon**。Cocoa 使用了 Carbon 提供的某些服务，因为有些 Carbon 框架在 Core Services 和 Application Services 层中定位为系统级别的服务。作为例子，Carbon Core 就是这些框架中特别重要的一个，Cocoa 使用了它提供的 File Manager（文件管理器）组件来进行不同文件系统表示之间的转换。
- **Core Graphics**。Cocoa 描画和图像处理类（相当自然且紧密地）基于 Core Graphics 框架，它实现了 Quartz 和窗口服务器组件。
- **Launch Services**。NSWorkspace 类负责向外提供 Launch Services 的潜在能力。Cocoa 还使用 Launch Services 提供的应用程序注册功能来获取与应用程序及文档相关联的图标。
- **Print Core**。Cocoa 的打印类是打印子系统的一个面向对象的接口。

此外，Cocoa 还使用 Carbon 环境的 Text Encoding Converter（文本编码转换器）服务来处理一些字符串编码转换。还有一些 Cocoa 方法向外提供 I/O Kit 框架、QuickDraw (QD)框架、 Apple Event (AE)框架、和 ATS 框架的部分功能，分别用于进行电源管理、QuickDraw 描画、 Apple Event 处理、以及提供字体支持。

进一步阅读： *Mac OS X 技术概览*一书给出有关框架、服务、技术、和 Mac OS X 其它组件的概览。*苹果人机界面指南*一书则专注于说明 Aqua 人机界面的外观和行为。

Cocoa 应用程序的特性

创建一个 Cocoa 应用程序，而又不必编写哪怕一行代码的情况是可能的。在 Xcode 中建立一个新的 Cocoa 工程，然后进行连编就可以了。当然，这个应用程序不做很多工作，至少不做很多有趣的工作。但是，这个极度简单的应用程序在鼠标双击时仍然可以启动，可以在 Dock 上显示图标，可以显示其主菜单和窗口（标题为"Window"），可以根据命令将自身隐藏，可以和其它运行着的应用程序互动，还可以处理退出命令。您可以对这个窗口进行移动、调整尺寸、最小化、和关闭，甚至可以打印包含在窗口中的空白部分。

想像一下如果加入一点代码，您可以做些什么。

在编程方面，Cocoa 为开发者提供很多免费或代价很低的支持。当然，要成为一个高效率的 Cocoa 开发者意味着要熟悉新的概念、设计模式、编程接口、和开发工具，而这方面的努力并不是无足轻重的。但是熟能生巧，编程在很大程度上变成一种将 Cocoa 提供的编程组件和负责定义特殊逻辑的定制对象及代码装配在一起、再将这些装配物组合在一起的练习。

接下来的部分是一个简短的列表，说明 Cocoa 如何为您的应用程序增加价值，而只需要您加入少量的工作（有时候甚至不需要）：

- **基本应用程序框架**—Cocoa 为事件驱动的行为和应用程序、窗口、工作空间（workspace）的管理提供了基础设施。在大多数情况下，您不必直接处理事件或发送任何描画命令给渲染库。
- **用户界面对象**—Cocoa 为应用程序的用户界面提供了丰富而又现成的对象。这些对象的大部分都在 Interface Builder（创建用户界面的开发工具）的选盘上，您只要简单地将对象从选盘拖拽到界面

上，配置好属性，并将它连接到其它对象上就可以了（当然，您也可以通过编程的方式对其进行实例化、配置、以及建立对象之间的连接）。下面是一些 Cocoa 用户界面对象的实例：

windows	text fields	radio buttons	drawers
sheets	tab views	table views	browsers
pop-up lists	sliders	image views	color wells
combo boxes	scroll views	text views	steppers

- 此外，Cocoa 还有一些支持用户界面的技术，包括提高可访问性、执行正当性检查、以及连接用户界面对象和定制对象需要的技术。
- **描画和图像处理**—Cocoa 带有一个可以锁定图形焦点并将视图（或视图的一部分）标识为“变脏”的框架，从而支持高效的定制视图描画。Cocoa 中还有一些描画贝齐尔（Bezier）路径、执行远交变换、合成图像、以及创建不同图像表示的编程工具类。
- **系统交互**—Cocoa 使您的应用程序可以和文件系统、工作空间、以及其它应用程序进行交互（或使用它们提供的服务）。
- **数据交换**—Cocoa 通过拷贝-粘贴、拖拽模型、以及 Services 菜单简化了应用程序内部和应用程序之间的数据交换。
- **性能**—为了增强应用程序的性能，Cocoa 提供了多线程、空闲时间处理、资源的迟缓加载、内存管理、和运行环操作方面的编程支持。
- **基于文档的应用程序**—Cocoa 为应用程序提供一种可以包含无限数量的文档架构。每个文档都包含在它自己的窗口中（比如一个字处理程序）。事实上，如果您选择“Document-based application（基于文档的应用程序）”工程类型，那么这类应用程序需要的很多组件就自动被创建了。
- **脚本处理**—通过应用程序脚本能力信息和一组支持脚本的 Cocoa 类，您就可以使自己的应用程序具有脚本能力。也就是说，您的应用程序可以响应由 AppleScript 脚本发出的命令。应用程序也可以通过执行脚本或使用单独的 Apple Event 来向其它应用程序发送命令，或者接受其它应用程序的命令。结果是每个具有脚本能力的应用程序都可以为用户或其它应用程序提供服务。
- **国际化**—Cocoa 使用一种已经精化多年的方法来支持国际化和本地化。这种方法基于偏好语言的用户列表，将本地化的资源放到应用程序的程序包（bundle）中。Cocoa 还提供产生和访问本地化字符串的工具和编程接口。而且，Cocoa 中的文本操作缺省情况下是基于 Unicode 的，因此有利于程序的国际化。
- **Undo 管理**—您可以注册一个用户动作来和 undo 管理器协同工作，当用户选择合适的菜单项时，它们会处理 undo（或 redo）动作。Undo 管理器通过独立的栈来维护 undo 和 redo 操作。
- **文本**—Cocoa 提供了一个复杂的文本系统，使您可以进行从简单到较为复杂的文本处理，简单文本处理的一个例子是在文本视图上显示可编辑的文本，复杂的处理则比如字距和连字的控制、拼写检查、和嵌入图像。

- **打印**—和文本系统相类似，打印架构使您可以打印文档和其它应用程序内容，并进行各种控制调整。在最简单的级别上，您缺省可以打印各种视图的内容；在较为复杂的级别上，您可以定义打印的内容和格式，控制一个打印作业如何进行，以及在打印面板上添加必要的视图。
- **偏好设置**—用户缺省设置系统基于一个系统范围内的数据库，您可以将全局或应用程序特有的偏好设置存储在这个数据库中。
- **连网**—Cocoa 包含一个分布式对象（Distributed Objects）架构，它使一个 Cocoa 进程可以和相同或不同的计算机上的其它进程进行通讯。这个架构还提供将 Bonjour 能力集成到应用程序的编程接口。
- **多媒体**—Cocoa 提供了 QuickTime 视频和基本音频能力的支持。

开发环境

说 Cocoa 有它自己的开发环境并不十分精确。一个原因是程序员可以使用苹果主要的开发工具—Xcode 和 Interface Builder—来开发 Mac OS X 其它应用程序环境的程序，比如 Carbon；其次，开发 Cocoa 应用程序时完全不使用 Xcode 和 Interface Builder 也是可能的，比如，您可以使用 Metrowerks 的 CodeWarrior 来管理、编译、和调试 Cocoa 工程；而且，如果您实在不愿意改变，也可以使用像 Emacs 这样的文本编辑器来编写代码，用 make file 和命令行进行应用程序的连编，然后在命令行上通过 gdb 调试器来调试程序。

但是，Xcode 和 Interface Builder 是开发 Cooca 软件优先使用的工具。它们的起源正好和 Cocoa 是一致的，所以在工具和框架之间存在高度的兼容性。Xcode 和 Interface Builder 一起，使设计、管理、连编、和调试 Cocoa 软件工程变得非常容易。还有一个叫 AppleScript Studio 的工具，可以扩展应用程序的能力，您可以通过这个工具来创建具有脚本能力的 Cocoa 程序和通过 AppleScript 控制其它应用程序的程序。

本部分包含如下内容：

Xcode

Xcode 是苹果公司在 Mac OS X 下的集成开发环境（IDE）引擎。它负责处理从最开始到工程的最终部署这一过程中的大多数细节。您可以通过这个工具来完成如下工作：

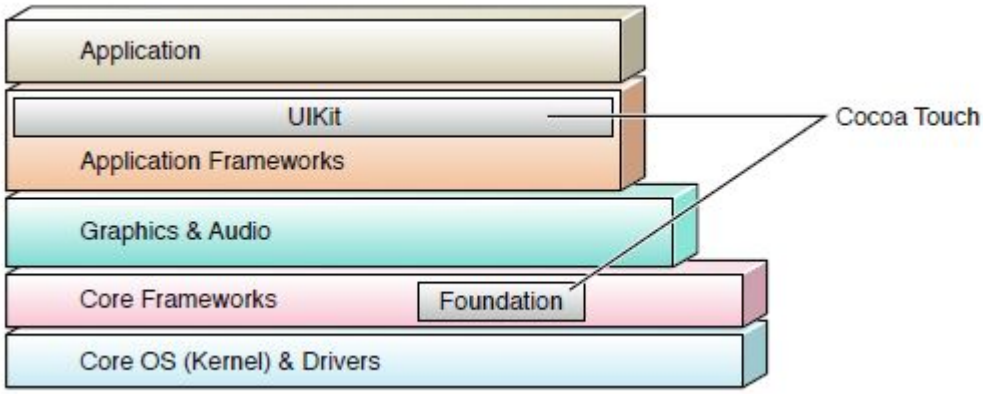
- 创建和管理工程，包括指定目标的需求、依赖性、和连编的风格。
- 在具有语法染色、自动缩进等功能的编辑器上编写源代码。
- 在工程的不同组件中进行漫游和检索，包括头文件和文档。

- 连编工程
- 在图形化的源代码级调试器上调试工程

Xcode 可以连编由 C、C++、Objective-C、Objective-C++、和 Java 编写的源代码组成的工程，可以生成 Mac OS X 支持的所有类型的执行代码，包括命令行工具、框架、插件、内核扩展、程序包、和应用程序。Xcode 允许您对连编和调试工具、可执行文件的打包方式（包括信息属性列表和本地化程序包）、连编过程（包括拷贝文件、脚本文件处理、和其它连编阶段）、以及用户界面（包括分立和多视图的代码编辑器）进行几乎无限的定制；还支持几个源代码管理系统（其中有 CVS 和 Perforce），使您可以将文件追加到代码库、提交修改、获取更新版本、以及进行版本的比较。

图 1-4 显示一个 Xcode 工程的实例。

图 1-4 Xcode 中的 TextEdit 实例



Xcode特别适合用于Cocoa开发。在创建工程时，Xcode可以通过对应于Cocoa工程类型的工程模板，为您建立一个初始的开发环境。Cocoa工程类型有：应用程序（Objective-C或Java）、基于文档的应用程序（Objective-C或Java）、工具、程序包、和框架。Xcode使用GNU C编译器（gcc）来编译Cocoa软件，使用GNU源代码级调试器（gdb）来调试软件。在Cocoa开发中使用gcc和gdb从它还是NeXTSTEP（参见"一点历史"部分）的时代就开始了。对Cocoa二进制代码的编译和调试经过多年的精化、扩展、和调优。Xcode 也有一个类浏览的功能，可以查看所有导入的 Cocoa 框架类和您自己的定制类，还有它们的继承关系；从类浏览器中，您可以请求查看任何类的文档。Xcode 还包含一些设计工具，其中有一个工具可以用于设计 Core Data 程序中使用的数据实体的属性和关系。

Xcode 和另外一个主要的开发工具 Interface Builder 良好地集成在一起。在 Interface Builder 中，您可以定义一个类（包括超类，插座变量，和动作），并为工程中的每个类生成源代码文件的框架。在 Xcode 中，您可以为定制类添加插座变量和动作，然后让 Interface Builder 将这些实体导入到 nib 文件中。

请注意：简单地说，插座变量（outlet）是一个对象和另一个对象的归档连接（表示为对象中的一个实例变量）；动作则是当按键或滑块这类对象被操作时，在被称为目标（target）的对象（通常是一个定制对象）中被调用的方法。Interface Builder也会把目标对象和其它对象（称为控件）之间的连接进行归档。更多有关插座变量、目标、和动作的信息请参见 "插座变量"和 "目标-动作机制"部分。
进一步阅读：*Xcode 快速指南*可以使您概览Xcode，并为您提供其它开发工具文档的连接。

Interface Builder

Cocoa 工程的第二个主要开发工具是 Interface Builder。顾名思义，Interface Builder 是用于创建用户界面的图形工具。Interface Builder 在 Cocoa 还是 NeXTSTEP 的时候就已经存在了，而且从那时侯起，它就作为同类软件中的佼佼者而获得广泛的认可。很自然，它和 Cocoa 的结合是很紧密的。而且您也可以用它来为 Carbon 应用程序创建用户界面。

Interface Builder 以三个主要的设计元素为中心：

- **Nib 文件**。nib 文件实际上是以档案的形式对用户界面中出现的对象进行文件包装（一个封装的目录）。这种档案本质上是一种对象图，包含每个对象的信息，包括对象的尺寸及在其屏幕（如果是个窗口）或窗口内的位置信息。Cocoa 应用程序中的 Nib 文件还包含定制类的代理引用和对象间连接信息，包括使用 Cocoa 绑定技术建立起来的连接。当您在 Interface Builder 中创建并保存一个用户界面时，重建该界面需要的所有信息都会被存在 nib 文件中。nib 文件还可以包含界面中使用的图像和声音文件。

  Interface Builder 将 nib 文件存储在 Cocoa 工程中的一个本地化目录下。在连编工程时，nib 文件就会被拷贝到新建的程序包中对应的本地化目录下（nib 文件因此可以提供一种很好的用户界面本地化方法）。 Cocoa 应用程序缺省（也就是说 Xcode 会自动创建）有一个主 nib 文件，在运行时自动被装载和显示。主 nib 文件包含应用程序的主菜单，并可能有一个或多个窗口。您的应用程序可以根据需要装载辅助的 nib 文件，比如文档或预置窗口所在的文件。

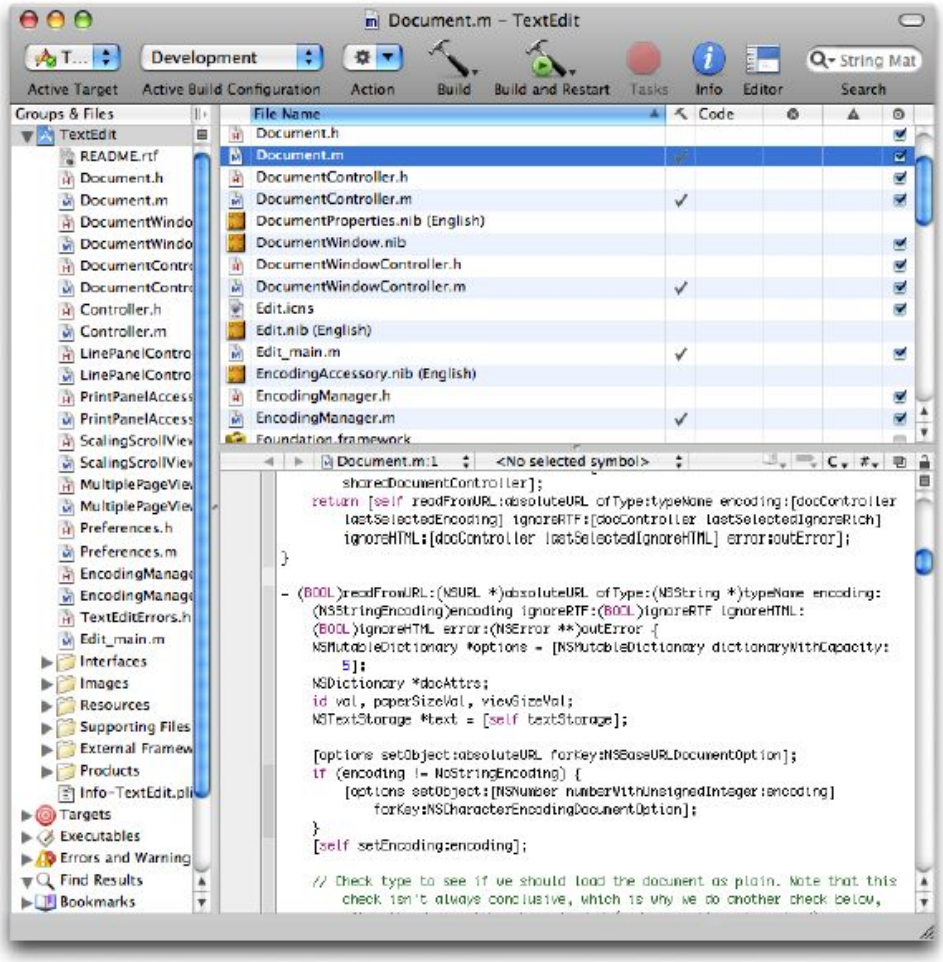
  Interface Builder 将 nib 文件的内容表示在 nib 文件窗口中。您也可以在 nib 文件窗口中定义定制类及考察对象之间的连接。

- **选盘**。 Interface Builder 的选盘（Palette）窗口中包含多个面板，或者称为“选盘”，每个选盘中包含一个彼此有关系的用户界面对象的集合。将对象从选盘拖拽到恰当的位置上就可以创建用户界面了，这里的位置可以是屏幕、窗口、各种视图、或者主菜单（选盘对象自身不一定是可见的，但是必须以某种形式影响用户界面）。当您将一个对象从选盘上拖出时，Interface Builder 会实例化一个该对象的缺省实例。这个实例是一个真正的 Cocoa 对象，而不是运行时需要创建的实例的代理对象。如果您愿意的话，可以将您自己的定制对象放在选盘上。

- **查看器**。 Interface Builder 为用户界面上的对象提供一个查看器（称为 Info 窗口）。Info 窗口由一系列可选择的面板组成，用于设置对象的初始属性和尺寸（虽然尺寸和很多属性也可以直接进行操作）。其中的两个面板用于建立对象之间的连接，一个基于插座变量和动作，另一个基于绑定技术。还有一个面板用于将定制类代替 Application Kit 框架中的超类。其它面板较为特殊，专用于建立用户界面对象和帮助标签及 AppleScript 事件处理器之间的关联。

图 1-5 显示了一个在 Interface Builder 中打开的 nib 文件，以及支持窗口。

图 1-5 Interface Builder 中 TextEdit 的预置窗口



用 Interface Builder 创建用户界面的步骤比较直接：

1. 将窗口或面板拖拽到屏幕上（面板等同于对话框或辅助窗口）。
2. 设置窗口的初始（或固定的）位置、尺寸、和属性。
3. 将文本框、按键、表视图控件、和弹出式列表等对象拖到窗口或之前放好的视图对象上。
4. 为这些对象设置初始（或固定的）位置、尺寸、和属性。
5. 为应用程序定义定制类。

   您可以直接在 Interface Builder 中完成这个工作，或者将事先创建好的头文件装载到 Interface Builder。在进行类的定制时，可以通过 Interface Builder 指定插座变量和动作。
6. 在对象之间建立绑定和连接。这有两种形式：
   - 在应用程序的视图、控制器、和模型对象之间建立绑定关系。
   - 把插座变量连接到它们引用的对象，接着把动作连接到目标对象中合适的方法上。
7. 保存和测试用户界面。

   Interface Builder 有一个功能，可以在设计的各个阶段测试界面（定制的行为除外）。

8. 为您定义的各个定制类创建头文件和源代码文件，这些文件会出现在关联的 Xcode 工程上。

Interface Builder 还包含一个功能：在对一个摆好位置的对象进行移动或调整其尺寸时，会通过一些短暂出现的蓝线来显示当前位置是否遵循 Aqua 人机界面指南，包括推荐的尺寸、对齐、与用户界面上的其它对象或窗口边界的相对位置。

进一步阅读：有关用户界面开发工具的进一步信息，请参见*Interface Builder*部分。此外，"Nib文件"部分给出了nib文件及其在应用程序中如何被使用的更多信息。您还可以参考 "对象之间的通讯"部分，以概要了解插座变量、目标-动作机制、以及Cocoa绑定技术。

AppleScript Studio

多年来，Mac OS 一直有一个定义良好的特性，就是用户可以通过由 AppleScript 语言写成的脚本控制应用程序。很多用户发现这个特性是不可或缺的，因为它可以将涉及多个应用程序的复杂操作序列串接在一起。AppleScript 的能力在 Mac OS X 系统上更进了一步。AppleScript Studio 是一种开发技术，用于创建通过 AppleScript 脚本控制复杂用户界面的 Cocoa 程序。

AppleScript Studio 把来自 AppleScript、Xcode、Interface Builder、和 Cocoa 的各种元素结合起来，提供了一个创建 AppleScript 解决方案的开发环境。您可以通过它来制作应用程序，完成如下任务：

- 执行 AppleScript 脚本
- 控制应用程序的界面
- 控制具有脚本能力的应用程序或操作系统中支持脚本的部分

由于 AppleScript Studio 将 AppleScript 和 Xcode、Interface Builder、和 Cocoa 集成在一起，脚本编程者可以利用这些组件各自的优势和能力。他们可以从 Interface Builder 选盘中拖出一组丰富的用户界面对象，根据个人的喜好进行定制；可以得到内置的 Aqua 用户界面指南的支持；还能够连编和维护带有多个目标和连编步骤的的复杂工程。

这个开发环境使我们有可能通过脚本对 Script Editor 程序（它是创建 AppleScript 脚本的传统工具）不能提供的能力进行控制，这些能力包括：

- 创建任意大的脚本
- 在脚本中进行检索和替代
- 单步脚本调试，支持各种执行方式
- 方便访问脚本中的处理函数和属性
- 灵活的字典查看器，和应用程序的脚本用语一起使用。

进一步阅读：更多信息请参见 *AppleScript Studio编程指南*。

其它开发工具

虽然 Xcode 和 Interface Builder 是开发 Cocoa 应用程序的主要工具，但是还有许多工具可以使用。在一些应用程序开发阶段中，您可能会找到很多辅助的应用程序和命令行工具。

这个部分将回顾一些辅助性的开发工具并简短地讨论一些命令行工具，然而命令行工具的数量太多，即使对它们进行一个浓缩的总结也超出了本文的范围。您的最好选择就是访问/usr/bin 和/usr/sbin 目录下各种工具的使用手册（man 页面），只要在 Terminal 外壳下键入 man 命令，后面跟着命令名称就可以了。在/Developer/Tools 目录下还有一些苹果开发的命令行工具。

性能工具

下面这些应用程序用于软件性能的测量和分析。它们位于/Developer/Applications 目录下。

- **Sampler** 用于分析程序运行时的行为和内存分配。顾名思义，Sampler 会按一定的时间周期对程序的函数调用栈进行采样，并在采样结束时向您显示调用频率最高的函数或方法。这种信息有助于定位消耗大量 CPU 时间或进行内存分配的函数或方法。
- **ObjectAlloc** 用于跟踪各种程序的内存分配与释放行为。这种历史数据可以显示重复的内存分配行为和总体的分配趋势。对于 Objective-C 代码，ObjectAlloc 在记录 alloc 调用的同时，也记录每个 copy、retain、release、autorelease 调用，而且还记录在 Core Foundation 中与这些方法相对应的函数，以及 malloc（及相关）函数进行的内存分配。
- **MallocDebug** 按分配时的调用栈顺序显示程序中当前已分配的内存块。通过这个工具一下子就可以看到您的应用程序消耗多少内存、这些内存是从哪里分配的、以及哪些函数分配了大量的内存。MallocDebug 还可以找出在程序中分配的、却没有被引用的内存，因此可以帮助您发现内存泄露并跟踪这些泄露的内存是在哪里分配的。
- **QuartzDebug** 是一个帮助您对应用程序的显示机制进行调试的工具，对大量进行描画和图像处理的应用程序特别有用。QuartzDebug 包括如下几个调试选项：
  - 自动闪烁描画模式，这种模式会在每个描画操作之后闪烁一下图形上下文。
  - 在更新屏幕区域之前先用黄色对该区域进行描画的模式。
  - 用于取得整个系统的窗口列表的静态快照，同时给出每个窗口的拥有者及该窗口消耗多少内存的选项。
- **Thread Viewer** 显示一个进程中各个线程的活动。这个工具显示每个线程的活动时间线，动作在时间线上用不同的颜色表示。点击时间线可以得到与点击位置相对应的活动回溯样本。

还有一些命令行工具可以用于性能分析，比如：

- top，对当前正在运行的进程进行采样统计并显示统计结果。
- gprof，用于产生程序的执行轮廓
- fs_usage，显示文件系统访问的统计信息

还有其它很多命令行工具可以用于性能分析。有关 Cocoa 应用程序开发可以用哪些性能分析工具以及性能分析的概念、技术、和策略的更多信息，请参见性能编程主题（Performance Programming Topics）文档。

请注意：*性能概述*对 Mac OS X 的性能工具进行讨论。

其它工具

您还可以发现下面的工具对 Cocoa 应用程序开发很有用（位于/Developer/Applications 目录下）：

- **Icon Composer 和 Icns Browser** 您可以用 Icon Composer 来导入各种格式的图像，创建应用程序图标和文档图标的图标文件；还可以用 Icns Browser 程序来创建不同尺寸、位深度、和位掩码的图标变体。
- **FileMerge** 这个工具可以可视化地“diffs”各种文本文件（比如源代码文件、头文件、和属性列表），并具有选择合并的能力。
- **Package Maker** 通过 Installer 程序为应用程序（其它类型的软件）制作安装包。
- **Property List Editor** 这是一个编辑器，用于创建和编辑 XML 及较老风格的属性列表。

Cocoa 框架

是什么因素使一个程序成为 Cocoa 程序呢？肯定不是编程语言，因为在 Cocoa 开发中您可以使用各种语言；也不是开发工具，因为您在命令行上就可以创建 Cocoa 程序（虽然那会使开发过程变得复杂，且需要消耗大量时间）。那么，所有 Cocoa 程序的共同点是什么？是什么使它们变得与众不同？答案是这些程序都是由一些对象组成，而这些对象最终都是从 NSObject 这个根类继承下来的；还有，这些程序都是基于 Objective-C 运行环境的。这个说法对于所有的 Cocoa 框架来说也是正确的。

请注意：上面的说法还需要做一点限制。首先，Cocoa 还有另一个根类，即 NSProxy。只是 NSProxy 很少用于 Cocoa 编程。其次，您可以创建您自己的根类，只是这需要很多工作（包括编写与 Objective-C 运行环境进行交互的代码），而且为此花费时间可能是不值得的。

Mac OS X 包含多个 Cocoa 框架，苹果和第三方厂商也随时会发布更多的框架。无论 Cocoa 框架有多么丰富，有两个框架总是与众不同：即 Foundation 和 Application Kit 框架，它们是核心的 Cocoa 框架。如果您没有连接并使用 Application Kit 框架中的类，就不能开发任何类型的 Cocoa 软件；同样地，如果您没有连接并使用 Foundation 框架中的类，也不能开发任何类型的 Cocoa 软件（当您连接 Cocoa 雨伞框架的时候，Xcode 会自动连接这些框架）。Foundation 和 Application Kit 框架在 Cocoa 开发中是必要的，其它框架则是辅助和可选的。

下面的部分将讨论上述两个核心的 Cocoa 框架，并简要描述一些辅助性的框架。为了使这些大框架更加容易理解，在介绍 Foundation 和 Application Kit 框架时，我们将每个层次中的数十个类分为不同的功能组。虽然这种分组方式有很强的逻辑基础，但是人们也可以按其它方式合理地进行分组。

本部分包含如下内容：

Foundation

Foundation 框架定义了一些基础类，可以用于各种类型的 Cocoa 程序。Foundation 框架和 Application Kit 框架的区分标准在于用户界面。如果一个对象既不出现在用户界面上，也不是专门用于支持用户界面，那么它就属于 Foundation 框架。您可以仅用 Foundation 框架创建一个 Cocoa 程序，而不涉及其它框架；命令行工具和 Internet 服务器就是这样的例子。

苹果公司在设计 Foundation 框架时牢记如下目标：

- 为诸如内存管理、对象改变、和通告这样的事物定义基本的对象行为和引入一致的规则。
- 通过程序包技术和 Unicode 字符串（和其它技术一起）支持国际化和本地化。
- 支持对象的持久保存。
- 支持对象的分发。
- 在一定程度上独立于操作系统，以支持移植。
- 为编程的元类型提供对象封装或等价物，比如数值、字符串、和集合，以及为访问底层系统实体和服务提供工具类，比如端口、线程、和文件系统。

Cocoa 应用程序定义为需要连接 Application Kit 框架，同时也总是必须连接 Foundation 框架的程序。这两个类层次都共用同一个根类，即 `NSObject` 类，很多（如果不是绝大多数的话）Application Kit 的方法和函数都将 Foundation 对象作为参数或返回值。一些 Foundation 类可能看起来像是为应用程序设计的，`NSUndoManager` 和 `NSUserDefaults` 类就是其中的两个例子，但是由于它们没有涉及到用户界面，所以被包含在 Foundation 框架中。

Foundation 的范式和策略

Foundation 为 Cocoa 编程引入了几个范式和策略，以保证程序中的对象在特定的环境下具有一致的行为和期望。包括：

- **对象的所有权和对象的清除**。Foundation 建立了一个对象所有权策略，用以代替垃圾收集机制。该策略指定对象需要释放自己创建、拷贝、或显式保留的其它对象。`NSObject`（类和协议）定义了保留和释放对象的方法。自动释放池（在 `NSAutoreleasePool` 类中定义）实现了一种迟缓释放（delayed-release）机制，使 Cocoa 程序在处理不由调用者负责的返回对象上有一个一致的规则。有关对象所有权和对象清除的更多信息，请参见 *Cocoa 的内存管理编程指南*。
- **可变类的变体**。在Foundation中，很多值和容器类的不可变类都有一个可以修改的变体，可变类总是不可变类的子类。如果您需要动态地改变一个经过封装的值，或者改变这种对象的所属关系，可以创建一个可变类的实例。由于它是从相应的不可变类继承下来的，所以您可以在接受不可变类型参数的方法中传入可变类的实例。对象可变性的更多信息请参见 "对象的可变性"部分。
- **类簇**。类簇是一个抽象类及一组私有的具体子类的组合，抽象类是这些子类的雨伞接口。根据不同的上下文（特别是创建对象所用的方法），类簇可以为您返回恰当的、经过优化的类实例。举例来说，`NSString`和`NSMutableString`就是针对不同的存储需要进行优化的各种私有子类实例的经纪类。多年来，具体类进行了几次修改，但应用程序依然可以工作。类簇的更多信息请参见 "类簇"部分。
- **通告**。通告是Cocoa的主要设计模式。它基于广播机制，该机制使一个对象（称为观察者）可以在另一个对象进行某种任务或遇到某种情况时，以用户或系统事件的方式得到通告。产生通告的对象可能并不知道通告观察者的存在或身份。有几种类型的通告：同步、异步、和分布式通告。Foundation 的通告机制由`NSNotification`、`NSNotificationCenter`、`NSNotificationQueue`、和`NSDistributedNotificationCenter`类实现的。更多关于通告的信息请参见 "通告"部分。

Foundation 类

Foundation类层次的根是NSObject类，它（和NSObject及NSCopying协议一起）定义了基本的对象属性和行为。更多有关NSObject和基本对象行为的信息请参见 "根类"部分。

Foundation框架的剩余部分由几组相互关联的类和一些独立的类组成。有一些代表基本数据类型的类，如字符串、字节数组、用于存储其它对象的集合类；一些代表系统信息的类，如日期类；还有一些代表系统实体的类，比如端口、线程、和进程。图 1-6、图 1-7、和 图 1-8 所示的类层次描述了这些类的逻辑分组及其继承关系。

图 1-6 Foundation 类层次—Objective-C（第一部分）

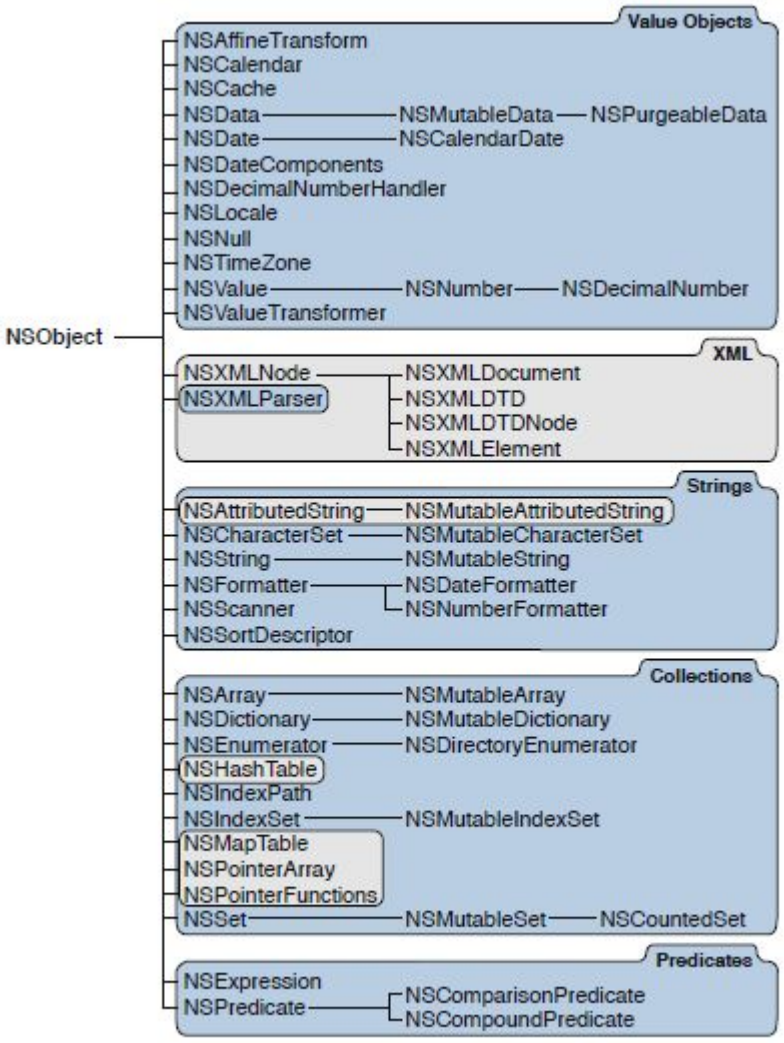


图 1-7 Foundation 类层次—Objective-C（第二部分）

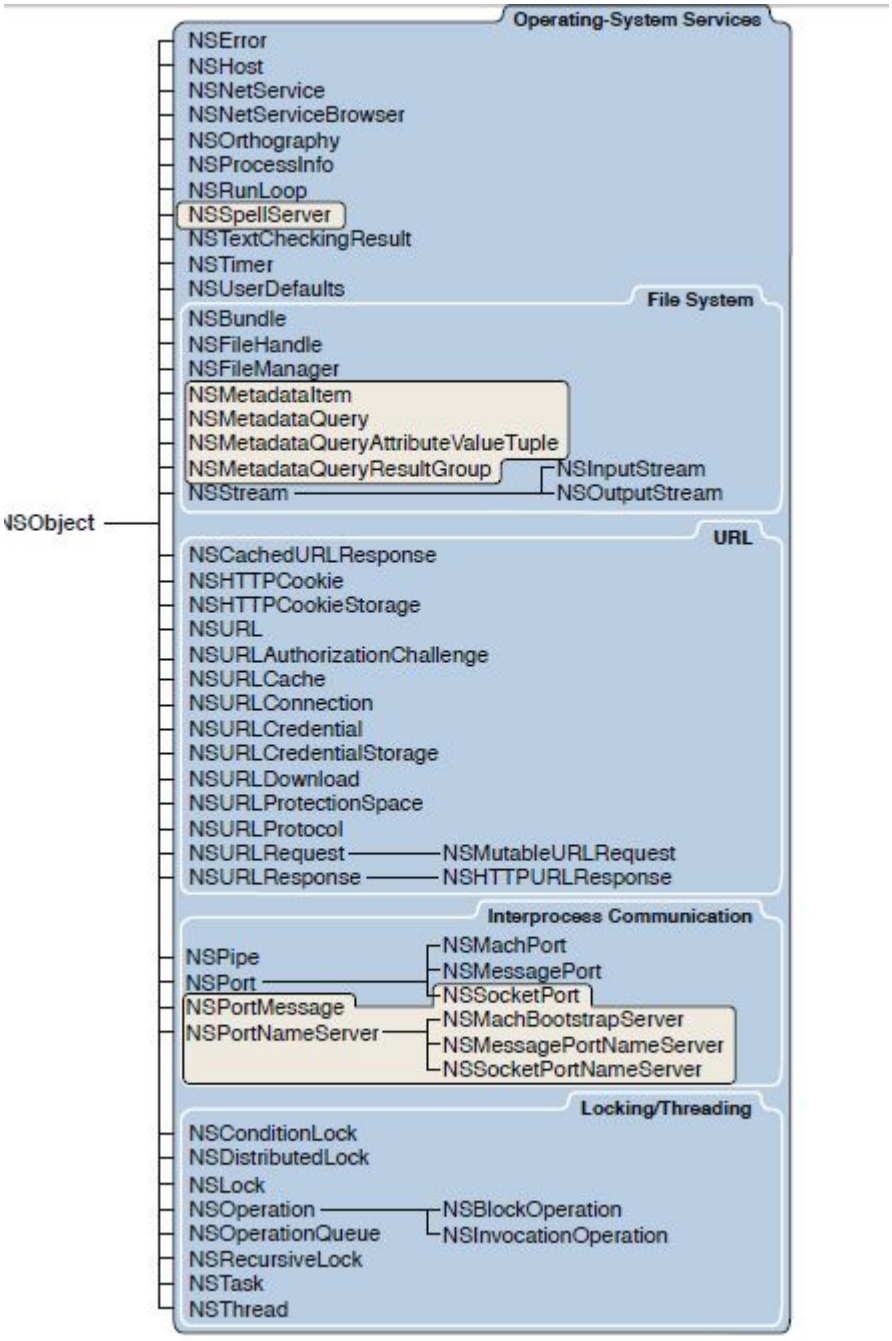
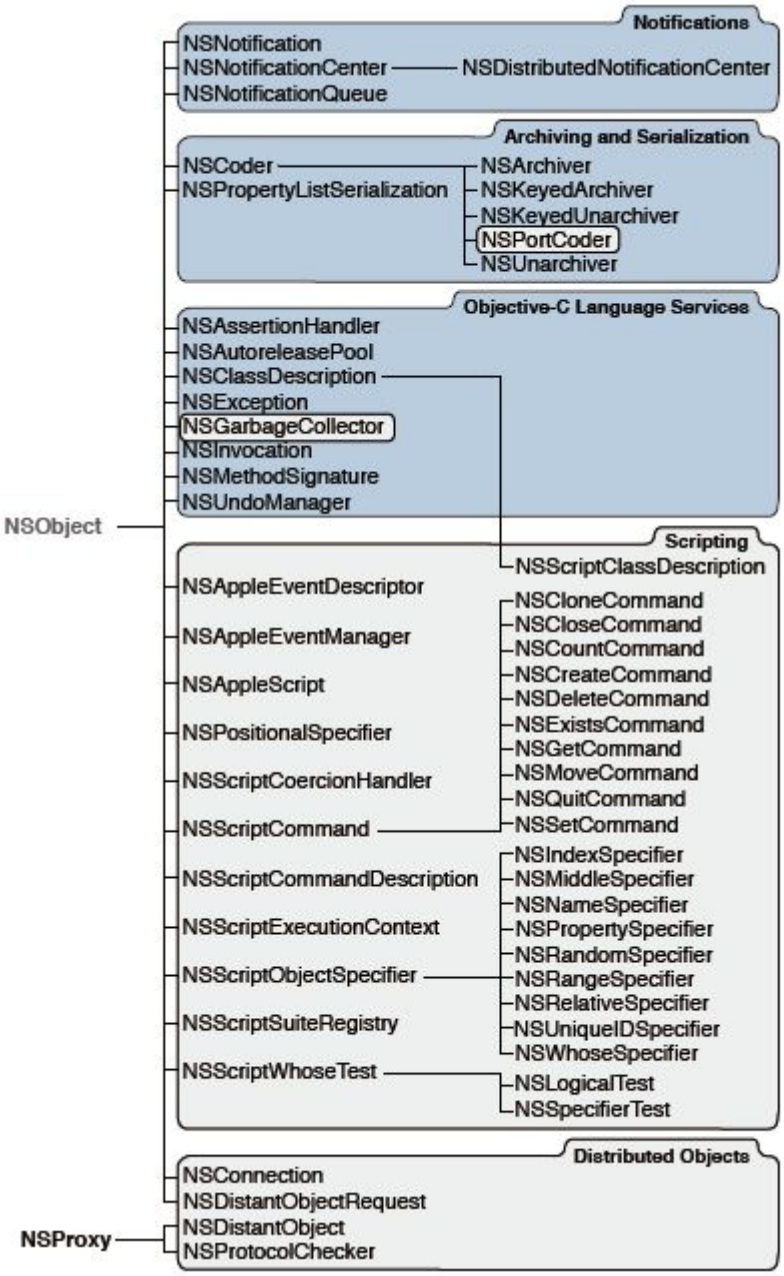


图 1-8 Foundation 类层次—Objective-C（第三部分）



上面这些框图按照如下范畴（和下文提到的其它关联关系）将 Foundation 框架中的类进行逻辑分类：

- **值对象**。值对象封装了各种类型的数据，提供对数据进行访问和各种操作的途径。因为它们都是对象，所以可以对它们（和它们包含的数值）进行归档和分发。`NSData` 类为字节流提供面向对象的存储空间，而 `NSValue` 和 `NSNumber` 类则为简单的标量值数组提供面向对象的存储空间。`NSDate`、`NSCalendarDate`、`NSTimeZone`、`NSCalendar`、`NSDateComponents`、和 `NSLocale` 类提供代表时间、日期、日历、和地域设置（locales）的对象。它们包含的方法可以用于计算日期和时间差、以各种格式显示日期和时间、以及调整世界上各个位置时间和日期。

- **字符串**。`NSString` 是另一类值对象，负责为以 null 结尾的、具有特定编码的字节数组提供面向对象的存储空间。它支持对 UTF-16、UTF-8、MacRoman、和很多其它编码的字符串之间进行转换。`NSString` 还提供对字符串进行检索、组合、和比较、以及对文件系统路径进行操作的方法。您可以

用 NSScanner 对象来对 NSString 对象中的数字和词进行解析。NSCharacterSet（显示在框图中的集合类部分）代表可以在各个 NSString 和 NSScanner 方法中使用的一组字符。

- **集合**。集合是以一定的顺序存储和访问其它对象（通常是数值）的对象。NSArray 的索引从 0 开始，NSDictionary 使用键-值对，而 NSSet 则负责对象的随机存储（NSCountedSet 类使集合具有唯一标识）。通过 NSEnumerator 对象，您可以访问一个集合中的元素序列。集合对象是属性列表的必要元素，和其它所有对象一样，它也可以被归档和分发。

- **操作系统服务**。 很多 Foundation 类为访问各种底层的操作系统服务提供便利，同时又把开发者从操作系统的具体特性隔离开来。举例来说，您可以通过 NSProcessInfo 类查询应用程序运行的环境；通过 NSHost 类得到主机系统在网络中的名称和地址；通过 NSTimer 对象，您可以按指定的时间间隔向其它对象发送消息；NSRunLoop 可以帮您管理应用程序或其它类型程序的输入源；而 NSUserDefaults 则为存储全局（主机级别）和用户级缺省值（预置）的系统数据库提供编程接口。

  - **文件系统和 URL**。 NSFileManager 为诸如创建、重命名、删除、和移动文件这样的文件操作提供统一的接口。NSFileHandle 则可以进行较为底层的文件操作（比如文件内查找操作）。NSBundle 可以寻找存储在程序包中的资源，可以动态装载某些资源（比如 nib 文件和代码）。您可以用 NSURL 和 NSURLHandle 类来表示、访问、和管理源于 URL 的数据。

  - **进程间通讯**。 这个范畴中的大部分类代表不同的系统端口、套接字、和名字服务器，对实现底层的 IPC 很有用。NSPipe 代表一个 BSD 管道，即一种进程间的单向通讯通道。

  - **线程和子任务**。 NSThread 类使您可以创建多线程的程序，而各种锁（lock）类则为彼此竞争的线程在访问进程资源时提供各种控制机制。通过 NSTask，您的程序可以分出一个子进程来执行其它工作或进行进度监控。

- **通告**。请见 "Foundation的范式和策略"部分中的有关通告类总结。

- **归档和序列化**。这个范畴中的类使对象分发和持久保留成为可能。NSCoder 及其子类和 NSCoding 协议一起，可以以独立于架构的方式来表示对象中包含的数据，可以将类信息和数据一起存储。

- **表达式和条件判断**。条件判断类，即 NSPredicate、NSCompoundPredicate、和 NSComparisonPredicate 类，负责对获取或过滤对象的逻辑约束条件进行封装。NSExpression 对象则代表条件判断中的表达式。

- **Spotlight 查询**。NSMetadataItem、NSMetadataQuery 和相关的查询类对文件系统的元数据进行封装，使元数据的查询成为可能。

- **Objective-C 语言服务**。NSException 和 NSAssertionHandler 类为代码中的断言和例外处理提供了面向对象的封装。NSInvocation 对象是 Objective-C 消息的静态表示，您的程序可以对它存储，并在之后用于激活另一个对象的消息。undo 管理器（NSUndoManager）和分布式对象（Distributed Objects）系统都用到了这种对象。NSMethodSignature 对象负责记录方法的类型信息，可以用于信息的推送。NSClassDescription 则是一个抽象类，用于定义和查询类的关系和属性。

- **脚本**。这个范畴中的类可以帮助您实现对 AppleScript 脚本和 Apple Event 命令的支持。

- **分布式对象**。 您可以通过分布式对象类来进行同一台电脑或一个网络中的不同电脑上的进程间通讯。其中的两个类－NSDistantObject 和 NSProtocolChecker 的根类（NSProxy）和 Cocoa 其它部分的根类不同。

- **网络**。`NSNetService`和`NSNetServiceBrowser`类支持称为Bonjour的零配置网络架构。Bonjour是在 IP 网络上发布和浏览服务的强大系统。

Application Kit

Application Kit 框架包含实现图形的、事件驱动的用户界面需要的所有对象：窗口、对话框、按键、菜单、滚动条、文本输入框—这个列表还在不断增加。Application Kit 帮助您处理所有的细节，它可以高效地进行屏幕描画、和营建设备及屏幕缓冲区进行通讯，在描画之前清除屏幕上的区域，以及对视图进行裁剪。Application Kit 框架中的类数量咋一看好像很吓人，但是大多数的 Application Kit 类都是支持类，您不必直接使用。您还可以选择在哪个级别上使用 Application Kit：

- 使用 Interface Builder 创建从用户界面对象到应用程序控制器对象的连接，控制器对象负责管理用户界面，协调用户界面和内部数据结构之间的数据流。为此，您可能会用到 off-the-shelf 控制器对象（用于 Cocoa 绑定），可能需要实现一个或更多的定制控制器类—特别是使用那些类的动作和委托方法。举例来说，您需要实现一个方法，使之在用户选择某个菜单项时被调用（如果该菜单项没有可接受的缺省实现的话）。
- 以编程的方式控制用户界面，这需要对 Application Kit 的类和协议更加熟悉。举例来说，支持用户将图标从一个窗口拖拽到另一个窗口需要一些编程工作，而且熟悉 `NSDragging...` 协议。
- 通过子类化 `NSView` 或其它类实现您自己的对象。在子类化 `NSView` 时，需要用图形函数来编写自己的描画函数。子类化要求对 Application Kit 的工作机制有更深的理解。

Application Kit 概述

Application Kit由超过 125 个类和协议组成。所有的类最终都从Foundation框架的`NSObject`类继承而来。图 1-9 和 图 1-10 的框图显示了Application Kit类的继承关系。

图 1-9 Application Kit 的类层次—Objective-C（第一部分）

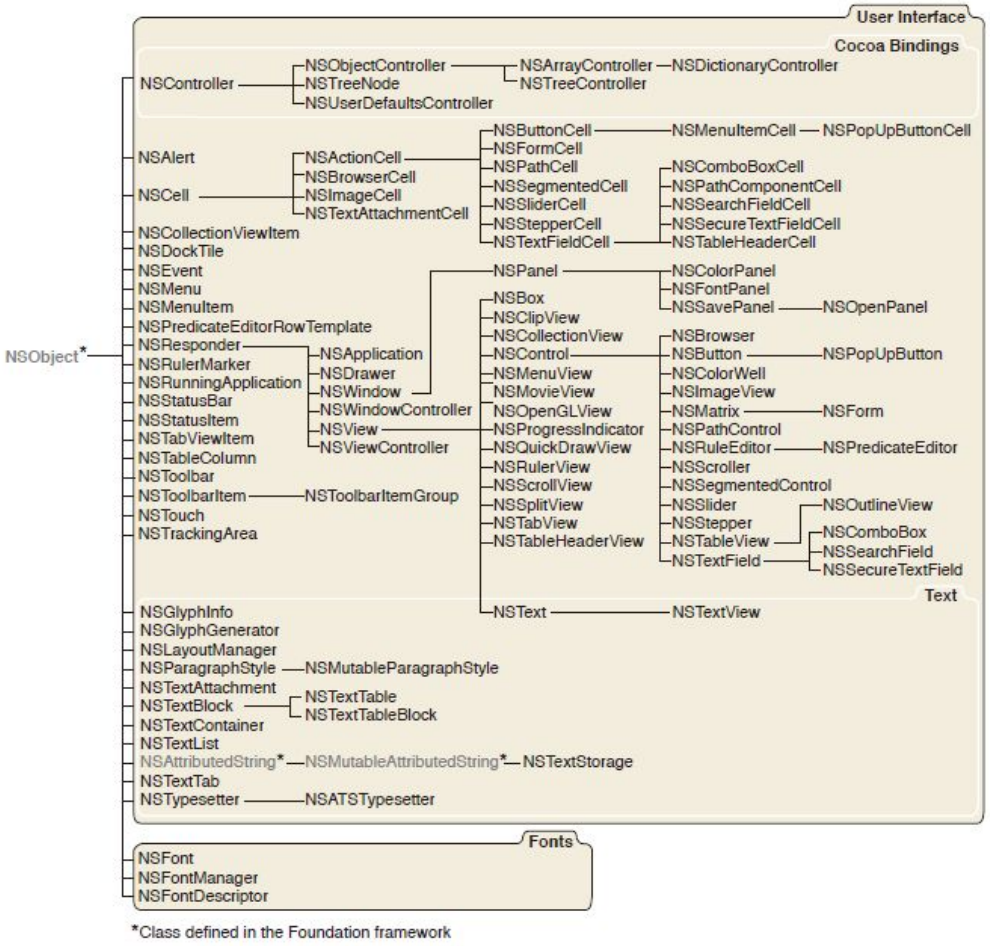
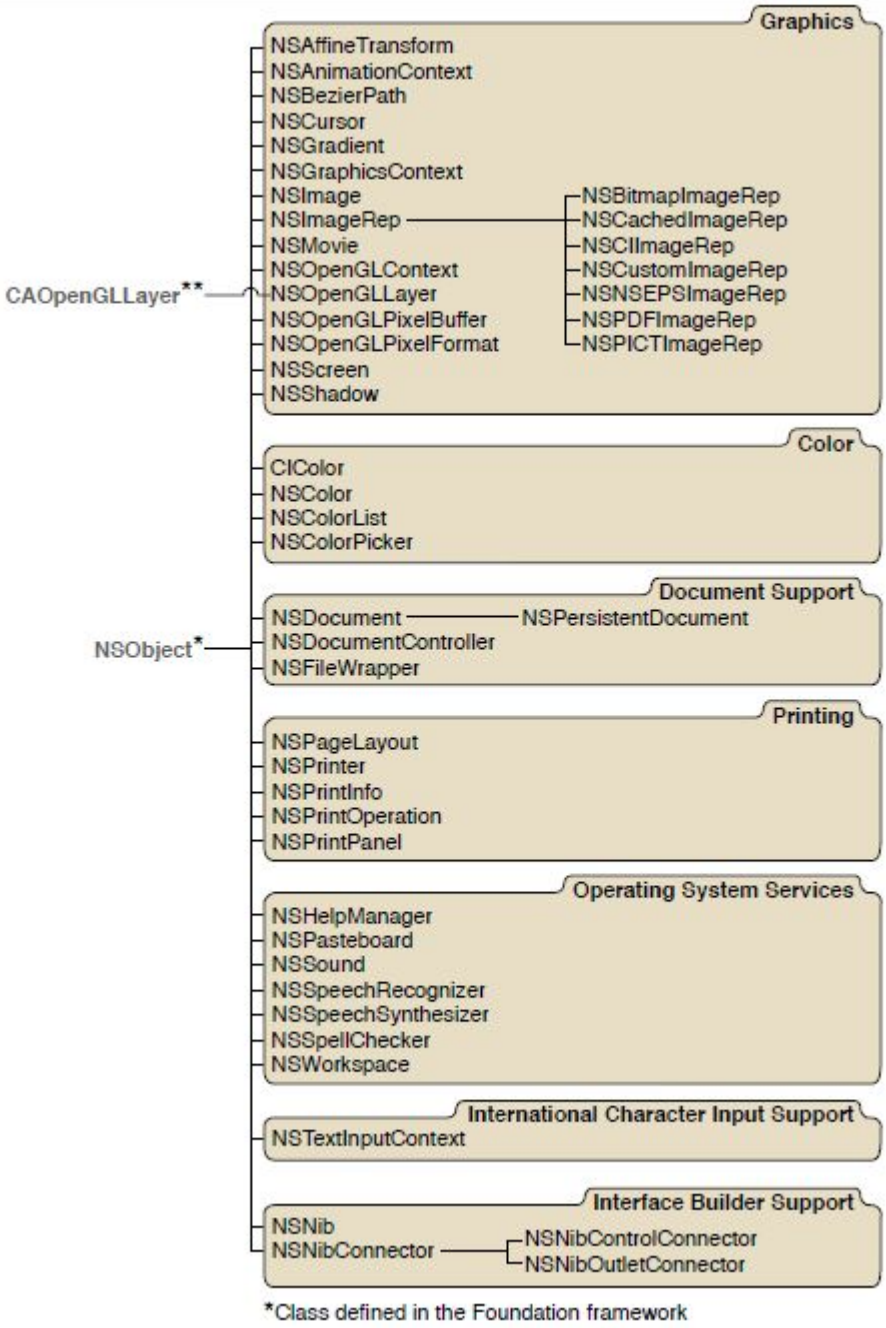


图 1-10 Application Kit 的类层次—Objective-C（第二部分）



如您所看到的那样，Application Kit 的类层次树既宽又浅，类层次中最深的类从根类开始只有五个超类，而大多数类的层次都浅得多。类层次中的一些主要分支特别有意思。

Application Kit中最大分支的根是NSResponder类，它负责定义响应者链，即对用户事件进行响应的有序对象列表。当用户进行按键或鼠标点击时，系统就会产生一个事件，并沿着响应者链向上传递，寻找可以响应该事件的对象。任何处理事件的对象都必须继承自NSResponder类。核心的Application Kit类—NSApplication、NSWindow、和NSView—都继承自NSResponder。阅读 "核心应用程序架构"部分可以得到有关响应者类的更多信息。

Application Kit类的第二大分支继承自NSCell类。值得注意的是，这组类和NSControl类（继承自NSView）的派生类有大体上的映像关系。对于负责响应用户动作的用户界面对象，Application Kit采用的架构将它们的工作分为控件（control）对象和单元（cell）对象。NSControl和NSCell类以及它们的子类定义了一组常见的用户界面对象，比如按键（button）、滑块（slider）、和浏览器（browser），用户可以通过图形

化的操作控制应用程序的某些方面。大多数的控件对象和一个或多个单元对象相关联，单元对象负责实现描画细节和事件的处理。举例来说，一个按键是由一个NSButton对象和一个NSButtonCell对象构成的，进一步的信息请参见 "控件和单元架构"部分。

控件和单元的实现机制基于 Application Kit 的一个重要设计模式：目标-动作（target-action）机制。单元对象可以保留用户点击（或在单元上进行某种动作）时应该发送给特定对象的消息标识信息。当用户操作一个控件时（比如用鼠标点击控件），控件就从它的单元对象抽出必要的信息，并向目标对象发送动作消息。目标-动作机制使您可以指定目标对象及应该调用的方法，从而赋予用户动作某种意义。您通常可以使用 Interface Builder 来对目标和动作进行设置，只要按住 Control 键，同时将鼠标指针从控件对象拖拽到应用程序或其它对象就可以了。您也可以通过编程的方式来设置目标和动作。

Application Kit中另一个基于设计模式的机制是委托（delegation）机制。用户界面上的很多对象，比如文本框和表视图，都定义了委托。委托对象代表被委托对象进行各种动作，或者与之相互协作，因此可以在用户界面操作中实现特定的应用程序逻辑。有关委托、目标-动作、以及Application Kit的其它范式和机制的更多信息，请参见 "和对象进行通讯"部分。有关这些范式和机制的基础设计模式的讨论，请参见 "Cocoa的设计模式"部分。

下面部分将简要地描述Application Kit的某些能力及其架构的某些方面，还有一些类和协议。在描述时按照 图 1-9 和 图 1-10 所示的类层次图对类进行分组。

通用的用户界面类

在用户界面的总体功能方面，Application Kit 提供了如下几个类：

- **全局应用程序对象**。 每个应用程序都使用一个NSApplication类的单件实例来控制主事件循环、跟踪应用程序的窗口和菜单、将事件分发给恰当的对象（即应用程序本身或者它的一个窗口）、建立高级别的自动释放池、以及接收应用程序级别的事件通告。NSApplication对象有一个委托对象（由您来分配），在应用程序启动或终止、被隐藏或被激活、即将打开用户选择的文件等的时候，委托对象会得到通知。通过设置NSApplication对象的委托对象并实现相应的委托方法，您就可以定制应用程序的行为，而不必生成NSApplication的子类。"核心应用程序架构"部分对这个单件应用程序对象进行详细的讨论。

- **窗口和视图**。窗口和视图类，即 NSWindow 和 NSView，继承自 NSResponder 类，可以对用户动作进行响应。NSApplication 对象内维护着一个 NSWindow 对象的列表—应用程序的每个窗口都有一个对应的对象，而每个 NSWindow 对象都维护一些具有一定层次结构的 NSView 对象。视图层次用于窗口内部的描画和事件处理。NSWindow 对象负责处理窗口级别的事件，将其它事件分发给窗口中的视图对象，并为视图对象提供一个描画区域。NSWindow 对象也有一个委托，用于定制窗口的行为。

  NSView 是显示在窗口中的所有对象的超类。所有的 NSView 子类都需要借助图形函数来实现自己的描画方法；drawRect:是一个基本方法，在创建新的 NSView 时，需要重载这个方法。

  "核心应用程序架构"部分也对NSView和NSWindow对象进行描述。

- **Cocoa绑定的控制器类**。NSController是一个抽象类，它的具体子类有NSObjectController、NSArrayController、和NSTreeController，它们是Cocoa绑定实现的一部分。这个技术可以自动同步存储在对象中的应用程序数据和该数据在用户界面上的表现。有关这些类型的控制器对象的描述请参见 "模型-视图-控制器设计模式" 部分。

- **面板（对话框）**。NSPanel 是 NSWindow 的子类，用于显示一些短暂的、全局的、或紧急的信息。举例来说，您可以使用一个 NSPanel（而不是 NSWindow）的实例来显示错误信息，或请求用户对特

殊或不正常的情况进行响应。Application Kit 为您实现一些常用的对话框，比如 Save、Open、和 Print 对话框，用于保存、打开、和打印文档。将这些对话框用于各种应用程序的公共操作，可以给用户一个统一的观感。

- **菜单和光标**。NSMenu、NSMenuItem、和 NSCursor 类负责定义应用程序显示给用户的菜单和光标的行为和外观。
- **分组和滚动视图**。 NSBox、NSScrollView、和 NSSplitView 类用于为窗口中的视图集合或其它视图对象提供图形"附件"。您可以通过 NSBox 类将窗口中的元素分组，并为整组元素描画一个边界。NSSplitView 类可以在垂直或水平方向附加一些视图，并为每个视图分配一定的公共区域，用户可以通过滑动控制条来重新分配视图的区域。NSScrollView 类及其辅助类，NSClipView，为用户提供一个滚动机制，以及让用户初始化和控制滚动的图形对象。NSRulerView 类则可以为一个滚动视图添加标尺和标志。
- **表视图和大纲视图**。NSTableView 类以行列的方式显示数据。NSTableView 可以很好地（但不仅限于）用于显示数据库记录，在这种场合下，一行对应于一条记录，列则包含记录属性。用户可以对单独的单元进行编辑，以及重新排列各个列。您可以通过设置委托和数据源对象来控制 NSTableView 对象的行为和内容。大纲视图（即 NSOutlineView 的实例，是 NSTableView 的子类）提供了另一种显示表格数据的方法。通过 NSBrowser 类，您可以为用户创建一个显示和漫游层次数据的对象。

文本和字体

NSTextField 类实现了一个简单的可编辑文本输入框，NSTextView 类则为更大的文本体提供更为广泛的编辑特性。

NSTextView 是抽象类 NSText 的子类，定义了扩展文本系统的接口。NSTextView 支持富文本、添附文件（图形文件及其它）、输入管理和按键绑定、以及标识文本属性，可以和 Font 窗口及 Font 菜单、标尺及段落风格、Serivces 工具、还有剪贴板（Clipboard）等组件互相协作。NSTextView 还允许通过委托和通告来进行定制—您很少需要从 NSTextView 派生出子类，也很少需要以编程的方式创建 NSTextView 的实例，因为 Interface Builder 选盘上的一些对象，比如 NSTextField、NSForm、和 NSScrollView，已经包含了 NSTextView 对象了。

通过 NSTextStorage、NSLayoutManager、NSTextContainer、和其它相关的类，还可能实现更为强大、更具创造力的文本操作（比如在一个圆里进行文本编辑）。Cocoa 文本系统还支持列表、表格、和非连续的文本选择。

NSFont 和 NSFontManager 类用于封装和管理字体的家族、尺寸、和变体。对于每种不同的字体，NSFont 类定义一个对象与之对应。这些对象都可以表示很多数据，为了提高效率，它们可以在应用程序中的所有对象之间共享。NSFontPanel 类定义了在用户界面上显示的 Font 窗口。

图形和颜色

NSImage 和 NSImageRep 负责封装图形数据，您可以通过这些类轻松而高效地访问存储在磁盘文件或显示在屏幕上的图像。每个 NSImageRep 的子类都知道如何描画特定类型的源数据代表的图像。NSImage 类可以为一个图像提供多种表示，还实现了诸如缓存这样的行为。Cocoa 的图像处理和描画的能力都集成在 Core Image 框架中。

Cocoa 的颜色处理由 NSColor、NSColorSpace、NSColorPanel、NSColorList、NSColorPicker、和 NSColorWell 类来支持。NSColor 和 NSColorSpace 类支持一组丰富的颜色格式和表示，包括定制颜色。其它类大多是接口类，负责定义和显示的面板及视图，使用户可以选择和应用颜色。举例来说，用户可以将颜色从 Color 窗口拖拽到任意的颜色井（color well）。NSColorPicking 协议可以用于扩展标准的 Color 窗口。

NSGraphicsContext、NSBezierPath、和 NSAffineTransform 类可以实现向量描画，支持图形变换，比如缩放、旋转、和转换等。

打印和传真

NSPrinter、NSPrintPanel、NSPageLayout、和 NSPrintInfo 类一起，可以将显示在窗口或视图上信息进行打印和传真，还可以创建 NSView 的 PDF 表示。

文档和文件系统支持

NSFileWrapper 类用于创建与磁盘文件或目录相对对应的对象。NSFileWrapper 将文件的内容保留在内存中，以便对其进行显示、修改、以及将它传输给其它应用程序。它还提供一个图标，用于拖拽该文件或将文件表示为附件。您也可以通过 Foundation 框架中的 NSFileManager 类来访问或枚举文件和目录内容。NSOpenPanel 和 NSSavePanel 类还提供了便利和熟悉的文件系统界面。

NSDocumentController、NSDocument、和 NSWindowController 类为创建基于文档的应用程序定义了一个架构（在类的层次框图中，NSWindowController 类显示在用户界面组中）。这类程序可以生成包含方式相同、但具有独特排布方式的数据，这些数据可以存储在文件中。在保存、打开、复原、关闭、以及管理这些文档方面，它们具有一些内置或易于得到的能力。

国际化和字符输入支持

如果一个应用程序要在世界其它地方使用，则可能需要根据语言、国家、或文化地域对其资源进行定制或本地化。举例来说，一个应用程序可能需要有独立的日语、英语、法语、和德语版本的字符串、图标、nib 文件、或上下文帮助。特定语言的资源文件被存放在程序包目录下的一个子目录下（就是那些带有 .lproj 扩展名的目录)。您通常可以通过Interface Builder来建立本地化资源文件。有关Cocoa国际化支持的更多信息，请参见 "Nib文件和其它应用程序资源"部分。

NSInputServer 类、NSInputManager 类、和 NSTextInput 协议一起，为您的应用程序提供访问文本输入管理系统的通道。该系统负责对不同国际化键盘产生的按键进行解释，并将正确的文本字符或 Control-key 事件递送给文本视图对象（通常由文本类和这些类进行交互，您不必介入）。

操作系统服务

下面这些 Application Kit 类为您的应用程序提供操作系统支持：

- **和其它应用程序共享数据**。NSPasteboard 类定义了剪贴板，可以存储从应用程序拷贝出来的数据，并使其它希望使用该数据的应用程序可以访问。NSPasteboard 实现了大家熟悉的剪切-拷贝-粘贴操作。通过剪贴板，NSServicesRequest 协议为应用程序间的数据传递定义一种基于注册服务的通讯机制（剪贴板在用户界面上实现为 Clipboard）。

- **拖拽**。只需要少量的编程工作，定制的视图对象就可以被拖拽到任意地方。只要遵循 NSDragging...协议，对象就可以变成拖拽机制的一部分；可拖拽的对象遵循 NSDraggingSource 协议，而目的对象（拖拽对象的接受者）则遵循 NSDraggingDestination 协议。Application Kit 隐藏了所有的光标跟踪和拖拽图像显示的细节。
- **拼写检查**。您可以通过 NSSpellServer 类来定义一个拼写检查服务，并将它作为服务提供给其它应用程序。通过 NSSpellChecker 类可以将您的应用程序连接到拼写检查服务上。NSIgnoreMisspelledWords 和 NSChangeSpelling 协议用于支持拼写检查机制。

Interface Builder 支持

NSNibConnector 类是一个抽象类，它和两个具体子类 NSNibControlConnector 和 NSNibOutletConnector 一起，表示 Interface Builder 上的连接。NSNibControlConnector 负责管理 Interface Builder 中的动作连接，NSNibOutletConnector 则管理插座变量连接。

带有 Cocoa API 的其它框架

作为标准的 Mac OS X 安装的一部分，苹果系统中还包含（除了 Foundation 和 Application Kit 框架外）一些使用 Cocoa 编程接口的框架（它们也可能使用 Carbon 或其它类型的编程接口）。您可以通过这些辅助性的框架来为应用程序实现一些期望但不是必须的能力。这些重要的辅助性框架包括：

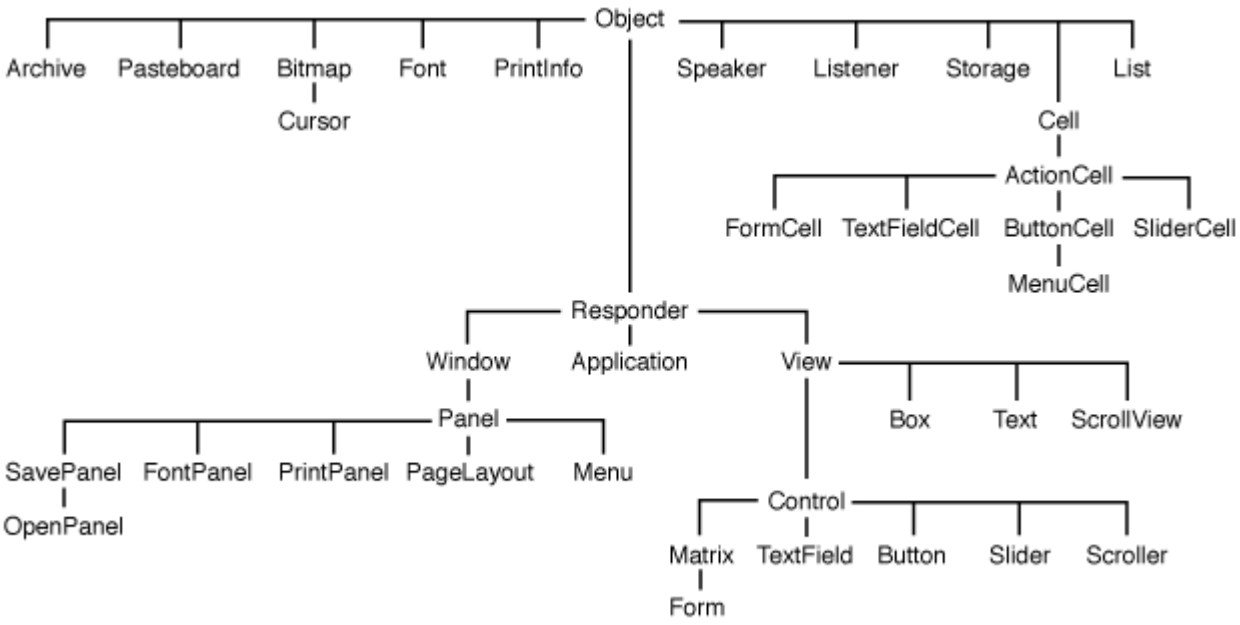
- Core Data—Core Data框架可以帮助应用程序管理模型对象图的整个生命周期，包括关系数据库或平坦文件中数据的持久存储。该框架还包括一些其它特性，比如undo和redo的管理、值的自动正当性检查、将对象的状态改变通知其它对象、以及与Cocoa绑定的集成。更多信息请参见 "其它Cocoa架构"部分和*Core Data编程指南*文档。
- Sync Services—用 Sync Services 可以将联系人、日历和书签结构、还有您自己的应用程序数据同步起来。您还可以扩展现有的结构。更多信息请参见 *Sync Services 编程指南*一书。
- Address Book—这个框架为联系人和其它个人信息实现了一个中心数据库。使用 Address Book 框架的应用程序可以和其它应用程序分享这些联系信息，包括苹果的 Mail 和 iChat。更多信息请参见 *Address Book 编程指南*一书。
- Preference Panes—您可以通过这个框架来创建应用程序动态装载的插件，实现用于录入用户偏好设置的用户界面。这个框架可以应用到您自己或系统级的应用程序。更多信息请参见*预置面板*文档。
- Screen Saver—Screen Saver 框架可以帮助您创建 Screen Effects 模块，该模块可以通过系统预置（System Preferences）程序来装载和运行。更多信息请参见 *Screen Saver 框架参考*一书。
- Web Kit—Web Kit 框架中提供一组在窗口中显示万维网内容的核心类。它缺省实现了一些功能，比如显示用户点击的连接。更多信息请参见 *Web Kit Objective-C 编程指南*一书。

一点历史

Cocoa 在很多年前被称为 NeXTSTEP。NeXT 电脑公司在 1989 年开发并发布了 NeXTSTEP 的 1.0 版本，并在之后不久推出 2.0 和 3.0 版本（分别在 1990 和 1992 年）。在这个早期的阶段，NEXTSTEP 更像一个应用程序环境；这个说法指的是整个操作系统，包括窗口和图像处理系统（基于 Display PostScript）、Mach 内核、设备驱动程序等等。

在那个时候还没有 Foundation 框架。事实上根本就没有框架。相反，软件库（动态共享）被称为工具箱，那些工具箱中最突出的就是 Application Kit。现在 Foundation 框架中的很多工作都实现为函数、结构、常量、和其它类型。Application Kit 自身有一组比现在小得多的类。图 1-11 显示的就是 NeXTSTEP 0.9 (1988) 的类层次图。

图 1-11 1988 年的 Application Kit 类层次



除了 Application Kit，早期的 NeXTSTEP 还包含 Sound Kit 和 Music Kit，这些库中包含一组丰富的 Objective-C 类，通过这些类可以对 Display Postscript 层进行高级别的访问，实现音频和音乐合成。

在 1993 年早期，NeXTSTEP 3.1 被移植（并发售）到 Intel、Sparc、和 Hewlett-Packard 计算机上。NeXTSTEP 3.3 也标志了一个新的大方向，因为它包含了 Foundation 的初级版本。大约在这个时间（1993），OpenStep 的创建者也采用了表单（form）。OpenStep 是 Sun 和 NeXT 的合作产物，目的是将高级别的 NeXTSTEP（特别是 Application Kit 和 Display PostScript）移植到 Solaris 系统上。名字中的“Open” 指的是两个公司联合发布的开放 API 规范。官方的 OpenStep API 在 1994 年 9 月发布，它第一次将 API 分割为 Foundation 和 Application Kit，并且第一次使用“NS”的前缀。

在 1996 年 6 月，NeXT 已经移植并发售了 OpenStep 4.0 版本，可以运行在 Intel、Sparc、和 Hewlett-Packard 计算机上，同时还有一个可以运行在 Windows 系统上的 OpenStep 运行环境。Sun 也完成了 OpenStep 在 Solaris 系统上的移植，并将它作为网络对象计算环境（Network Object Computing Environment）的一部分发售。但是 OpenStep 一直没有成为 Sun 公司总体策略的重要部分。

苹果公司在 1997 年收购 NeXT Software 公司（那时的叫法）的时候，OpenStep 变成一个 Yellow Box，并包含在 Mac OS X Server（也称为 Rhapsody）和 Windows 系统上。之后，随着 Mac OS X 策略的演化，它最终将名字改为“Cocoa”。

一个简单的 Cocoa 命令行工具

让我们从一个简单的命令行程序开始吧。给定一系列随机的词作为参数，由该命令行程序将多余的词删除，并将剩下的词按字母排序，打印在标准输出上。列表 2-1 显示该程序的一个典型执行结果。

列表 2-1 一个简单的 Cocoa 工具的输出

```
localhost> SimpleCocoaTool a z c a l q m z

a

c

l

m

q

z
```

列表 2-2 显示该程序的 Objective-C 代码。

列表 2-2 使输入的词唯一并将其排序的工具的 Cocoa 代码

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



int main (int argc, const char * argv[]) {

    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    NSArray *args = [[NSProcessInfo processInfo] arguments];

    NSCountedSet *cset = [[NSCountedSet alloc] initWithArray:args];

    NSArray *sorted_args = [[cset allObjects]

        sortedArrayUsingSelector:@selector(compare:)];

    NSEnumerator *enm = [sorted_args objectEnumerator];

    id word;

    while (word = [enm nextObject]) {

        printf("%s\n", [word UTF8String]);

    }
```

```
    [cset release];
    [pool release];
    return 0;
}
```

这段代码创建并使用了几个对象：一个自动释放池，用于内存管理；集合对象（多个数组和一个集合），用于对指定的词进行唯一性处理和排序；一个枚举对象，用于遍历最终数组的元素以及将它们打印在标准输入上。

关于代码，您可能注意到的第一件事是代码很短，可能比同样功能的 ANSI C 程序短得多。虽然很多代码可能看起来有些奇怪，但代码中的很多元素又和 ANSI C 类似，比如赋值操作符、流程控制语句（`while`）、对 C 语言连接库例程（`printf`）的调用、以及基本的标量类型等等。Objective-C 的基础显然是 ANSI C。

本章的剩余部分将考察这段代码中的 Objective-C 元素。这些元素将作为例子，用于讨论从消息发送机制到内存管理技术的各种主题。如果您之前从未见过 Objective-C 的代码，那么这个例子可能看起来很可怕，又绕又晦涩，但是这个印象很快就会消失。Objective-C 实际上是一种简单而优雅的编程语言，易于学习，编程也很直接。

用 Objective-C 进行面向对象的编程

从 Cocoa 事件驱动架构的机制和采用的范式可以看出它广泛地使用了面向对象的方法。Objective-C 是 Cocoa 的主要开发语言，也是完全面向对象的语言，尽管它的基础是 ANSI C。它为消息的分发提供运行环境支持，也为定义新类指定了语法规则。Objective-C 支持绝大多数其它面向对象编程语言（比如 C++ 和 Java）具有的抽象和机制，包括继承、封装、重用性、和多态。

但是，Objective-C 在一些重要的方面又和其它面向对象的语言不同。举例来说，Objective-C 和 C++不同，不支持操作符重载、模板、或多重继承。Objective-C 也不象 Java 那样，具有“垃圾收集”机制，可以自动释放不再需要的对象（虽然它有机制和规则可以完成同样的任务）。

虽然 Objective-C 没有这些特性，但是它作为一种面向对象编程语言的能力可以进行补偿和超越。本文接下来的部分将探讨 Objective-C 的特殊能力，同时概要介绍 Java 版本的 Cocoa。

进一步阅读：本部分的很多内容是Objective-C权威指南—*Objective-C编程语言*—一书上的概括。有关Objective-C的详细描述，请查阅该文档。

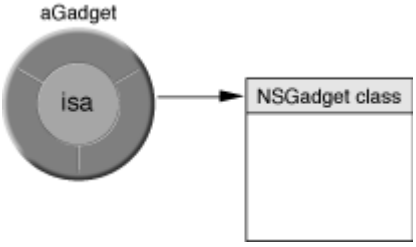
本部分包括如下内容：

Objective-C 的优点

如果您是一个面向过程的编程人员，对面向对象的概念不熟悉，则可以首先将对象的本质考虑为一个结构体加上关联的函数，这可能有助于理解本文的内容。这个概念和现实相差不太远，特别是在运行环境的实现方面。

每个 Objective-C 对象都隐藏着一个数据结构，它的第一个成员变量—或者说是实例变量—是“isa 指针”（大多数剩下的成员变量由对象的类或超类来定义）。顾名思义，isa 指针指向的是对象的类，这个类也是一个对象，有自己的权限（参见图 2-1），是根据类的定义编译而来的。类对象负责维护一个方法调度表，该表本质上是由指向类方法的指针组成的；类对象中还保留一个超类的指针，该指针又有自己的方法调度表和超类（还有所有通过继承得到的公共和保护的实例变量）。isa 指针对消息分发机制和 Cocoa 对象的动态能力很关键。

图 2-1 对象的 isa 指针



对隐藏在对象表面下的工作机制的惊鸿一瞥只是使您简单了解 Objective-C 运行环境如何支持消息分发、继承、和一般对象行为的其它方面。但是它对于理解 Objective-C 的主要能力—动态能力是很必要的。

动态能力

Objective-C 是一种非常动态的语言。这种动态能力使程序可以突破编译和连接时的约束，将更多符号辨识的工作转移到处于受控状态的运行环境上。Objective-C 比其它编程语言具有更强的动态能力，这种能力来源于如下三个方面：

- 动态类—在运行时确定对象的类
- 动态绑定—在运行时确定要调用的方法
- 动态装载—在运行时为程序增加新的模块

Objective-C为动态类型引入了一个称为id的数据类型，用于表示任意的Cocoa对象。列表 2-2 中的代码实例显示了这种基本对象类型的典型用法：

```
id word;

while (word = [enm nextObject]) {

// etc....
```

id 数据类型使我们有可能在运行时进行任意的对象替换。您因此可以在代码中用运行时的因子指定希望使用的对象类型。动态类型使对象中的关联可以在运行时确定，而不需要在静态设计时强制指定对象类型。编译时的类型检查可以确保更加严格的数据完整性，但是作为交换，动态类型则给您的程序更大的灵活性。而且，通过对象的内省（比如询问动态类型转换后的匿名对象所属的类），您仍然可以在运行时确认对象的类型，并验证它是否可以进行特定的操作（当然，您总是可以在需要的时候进行静态类型检查）。

动态类型为 Objective-C 的第二种动态能力—动态绑定—提供了物质基础。正如动态类型将对象类的确定推迟到运行时一样，动态绑定将调用方法的确定也推迟到运行时。在编译时，方法的调用并不和代码绑定在一起，只有在消息确实发送出来之后，才确定被调用的代码。通过动态类型和动态绑定技术，您的代码每次执行都可以得到不同的结果。运行时因子负责确定消息的接收者和被调用的方法。

运行时的消息分发机制为动态绑定提供支持。当您向一个动态类型确定了的对象发送消息时，运行环境系统会通过接收者的 isa 指针定位对象的类，并以此为起点确定被调用的方法，方法和消息是动态绑定的。而且，您不必在 Objective-C 代码中做任何工作，就可以自动获取动态绑定的好处。您在每次发送消息时，特别是当消息的接收者是动态类型已经确定的对象时，动态绑定就会例行而透明地发生。

动态装载是最后一种动态能力。它是 Cocoa 的一个特性，依赖于 Objective-C 的运行环境支持。通过动态装载，Cocoa 程序可以在需要的时候才装载执行代码和资源，而不是在启动的时候装载所有的程序组件。可执行代码（在装载之前就连接好了）通常包含一些新的、会被集成到应用程序运行时映像的类。代码和本地化资源（包括 nib 文件）被包装在程序包中，可以通过 Foundation 框架中的 `NSBundle` 类中定义的方法来显式装载。

这种程序代码和资源的“迟缓装载（lazy-loading）”机制降低了对系统内存的要求，从而提升了程序的整体性能。更重要的是，动态装载使应用程序变得可扩展。您可以考虑在应用程序中采用插件架构，使自己和其它开发者可以通过附加的模块进行定制，应用程序可以在发布数月或数年后动态装载附加的模块。如果设计是正确的，这些类就不会和已经存在的类发生冲突，因为每个类都封装了自己的实现并拥有自己的名字空间。

语言扩展

Objective-C 在基本语言上做了两个扩展：范畴（categories）和协议（protocols），它们是强大的软件开发工具。这两个扩展引入了声明方法并将它们关联到某个类的技术。

范畴

范畴提供一种为某个类添加方法而又不必制作子类的途径。范畴中的方法会变成类的一部分（在您的应用程序的作用域内），并为该类的所有子类所继承。在运行时，原始方法和通过范畴添加的方法之间没有差别，您可以向类（或者它的子类）实例发送消息，以调用范畴中定义的方法。

范畴不仅是一种为类添加行为的便利方法，还可以对方法进行分组，将相关的方法放在不同的范畴中。范畴对于组织规模大的类特别方便，例如当几个开发者同时在一个类上工作时，您甚至可以将不同的范畴放在不同的源文件中。

范畴的声明和实现很象子类。在语法上，唯一的区别是范畴的名称需要跟在`@interface` 或 `@implementation` 导向符之后，且放在园括号中。举例来说，假定您希望为 `NSArray` 类增加一个方法，以便用更加结构化的方式打印集合的描述。那么您可以在范畴的头文件中书写如下的声明代码：

```
#import <Foundation/NSArray.h> // if Foundation not already imported


@interface NSArray (PrettyPrintElements)

- (NSString *)prettyPrintDescription;
```

```
@end
```

然后在实现文件中书写如下代码：

```
#import "PrettyPrintCategory.h"


@implementation NSArray (PrettyPrintElements)

- (NSString *)prettyPrintDescription {

    // implementation code here...

}

@end
```

范畴有一些限制。您不能通过范畴为类添加新的实例变量。虽然范畴方法可以覆盖现有的方法，但这并不是推荐的做法，特别是当您希望对现有行为进行增强的时候。一个原因是范畴方法是类接口的一部分，因此无法通过向 `super` 发送消息来获取类中已经定义的行为。如果您需要改变一个类的现有方法的行为，更好的方法是生成一个该类的子类。

您可以通过范畴来为根类—NSObject—添加方法。通过这种方式添加的方法可以用于与该代码相连接的*所有*实例和类对象。非正式的协议—Cocoa 委托机制的基础—在 `NSObject` 类中声明为范畴。然而，这种在使用上的广泛适用也有它的风险。您通过范畴向 NSObject 添加的行为可能会有意料不到的结果，可能导致崩溃，数据损坏，甚至更坏的结果。

协议

Objective-C 的另一个扩展称为协议，它非常象 Java 中的接口。两者都是通过一个简单的方法声明列表发布一个接口，任何类都可以选择实现。协议中的方法通过其它类实例发送的消息来进行调用。

协议的主要价值和范畴一样，在于它可以作为子类化的又一个选择。它们带来了 C++多重继承的一些优点，使接口（如果不是实现的话）可以得到共享。协议是一个类在声明接口的同时隐藏自身的一种方式。接口可以暴露一个类提供的所有（通常是这种情况）或部分服务。类层次中的其它类都可以通过实现协议中的方法来访问协议发布的服务，不一定和协议类有继承关系（甚至不一定具有相同的根类）。通过协议，一个类即使对另一个类的身份（也就是类的类型）一无所知，也可以和它进行由协议定义的特定目的的交流。

有两种类型的协议：正式和非正式协议。非正式协议在 "范畴"部分中已经简单介绍了，它们是NSObject 类中定义的范畴。因此每个以NSObject为根类的对象（和类对象）都隐式采纳了范畴中发布的接口。和正式协议不同，一个类不必实现非正式协议中的每个方法，而是只实现它感兴趣的方法就可以了。为了使非正式协议正确工作，声明非正式协议的类在向某个目标对象发送协议消息之前，必须首先向它发送 `respondsToSelector:` 消息并得到肯定的回答（如果目标对象没有实现相应的方法，则产生一个运行时例外）。

Cocoa 中的“协议”通常指的是正式协议。它使一个类可以正式地声明一个方法列表，作为向外提供服务的接口。Objective-C 语言和运行系统支持正式协议；编译器可以根据协议进行类型检查，对象可以在运行时进行内省，以确认是否遵循某个协议。正式协议有自己的专用术语和语法。术语方面，提供者和客户的意义有所不同：

- 提供者（通常是一个类）*声明*正式的协议。

- 客户类*采纳*正式协议，表示客户类同意实现协议中所有的方法。
- 如果一个类采纳某协议或者是从采纳该协议的类派生出来的（协议可以被子类继承），则可以说该类*遵循*该协议。

在 Objective-C 中，声明和采纳协议都有自己的语法。协议的声明必须使用编译导向符`@protocol`。下面的例子显示了 `NSCoding` 协议（在 Foundation 框架的 `NSObject.h` 头文件中）的声明方式：

```
@protocol NSCoding

- (void)encodeWithCoder:(NSCoder *)aCoder;

- (id)initWithCoder:(NSCoder *)aDecoder;

@end
```

协议的声明类不需要实现这些方法，但应该对遵循该协议的对象方法进行调用。

如果一个类要采纳某个协议，需要在在`@interface` 导向符后、紧接着超类的位置上指定协议的名称，并包含在尖括号中。一个类可以采纳多个协议，不同的协议之间用逗号分隔。下面是 Foundation 框架中的 `NSData` 类采纳三个协议的方式：

```
@interface NSData : NSObject <NSCopying, NSMutableCopying, NSCoding>
```

通过采纳这些协议，`NSData` 许诺自己要实现协议中声明的所有方法。范畴也可以采纳协议，对协议的采纳将成为类定义的一部分。

Objective-C 通过类遵循的协议和类继承的超类来定义类的类型。您可以通过发送 `conformsToProtocol:` 消息来检查一个类是否遵循特定的协议：

```
if ([anObject conformsToProtocol:@protocol(NSCoding)]) {

       // do something appropriate

}
```

在类型声明—方法、实例变量、或函数中，您可以将遵循的协议作为类型的一部分来指定。这样您就可以通过编译器来得到另一个级别的类型检查，这种检查比较抽象，因为它不和特定的实现相关联。您可以使用与协议采纳相同的语法规则，即把协议的名称放在尖括号中，通过这种语法可以在类型中指定遵循的协议。您常常会看到在这些声明中使用了动态对象类型 `id`，例如：

```
- (void)draggingEnded:(id <NSDraggingInfo>)sender;
```

这里，参数中引用的对象可以是任意类型的类，但是必须遵循 `NSDraggingInfo` 协议。

除了目前为止已经提到的协议之外，Cocoa还提供了几个协议的例子。一个有趣的例子就是 `NSObject`协议。可以想象得到的是，`NSObject`类采纳了这个协议，还有一个根类—`NSProxy`—也采纳了这个协议。通过这个协议，`NSProxy`类可以和Objective-C运行环境的一部分进行交互，包括引用计数、内省、和对象行为的其它基础部分。

正式协议有其自己的限制。如果协议声明的方法列表随着时间而增长，协议的采纳者就会不再遵循该协议。因此，Cocoa 中的正式协议被用于稳定的方法集合，比如 `NSCopying` 和 `NSCoding`。如果您预期协议方法会增多，则可以声明为非正式协议，而不是正式协议。

使用 Objective-C

在面向对象的程序中，完成工作的方式是通过消息，即一个对象向另一个对象发送消息。通过消息，发送对象可以向接收对象（接收者）发出请求，请求接收者执行某些动作，返回某些对象或值，或者同时执行两者。

Objective-C在消息传递方法采用了独特的语法形式。列表 2-2 的语句来自SimpleCocoaTool工程的代码：

```
NSEnumerator *enm = [sorted_args objectEnumerator];
```

消息表达式位于赋值符号的右边，包含在方括号中。消息表达式中最左边的部分是接收者。它是一个变量，代表送出消息的对象。在这个例子中，接收者是 `sorted_args`，即 `NSArray` 类的一个实例。紧接着接收者的是消息体，在这个例子中就是 `objectEnumerator`（这里我们要专注的是消息语法，而不是深入探讨这个 SimpleCocoaTool 中的消息或其它消息实际上做些什么）。`objectEnumerator` 消息调用 `sorted_args` 对象中名为 `objectEnumerator` 的方法，该方法会返回一个对象的引用，并由赋值符号左边的 `enm` 变量来保存。`enm` 变量的类型被静态地定义为 NSEnumerator 类的一个实例。您可以将这个语句图解为：

NSClassName *variable* = [*receiver* **message**];

消息通常有参变量，或者称为参数。仅带一个参数的消息在消息名称后面附加一个冒号，并将参数直接放在冒号后：

NSClassName *variable* = [*receiver* **message:** *argument*];

和函数的参变量一样，参数的类型必须和方法声明中指定的类型相匹配。作为例子，请看如下 SimpleCocoaTool 工程中的表达式：

```
NSCountedSet *cset = [[NSCountedSet alloc] initWithArray:args];
```

这里 `args` 也是 `NSArray` 类的一个实例，它是 `initWithArray:`消息的参数。

如果消息有多个参数，则消息名称就有多个部分，每个部分都以冒号结束，冒号后面是新的参数：

NSClassName *variable* = [*receiver* **message:** *arg1* **anotherArg:** *arg2*];

上面引用的 `initWithArray:`例子很有意思，它说明了嵌套的使用。在 Objective-C 中，您可以将一个消息嵌套到另一个消息内部，将一个消息表达式返回的对象用作将它包围在内的另一个消息表达式的接收者。因此，为了解释嵌套的消息表达式，可以从最里面的表达式开始，然后向外延伸。下面的语句可以解释为：

1. 将 `alloc` 消息发送给 `NSCountedSet` 类，以创建（通过为其分配内存）一个未初始化的类实例。

   请注意：Objective-C 类自身也是对象，因此您也可以象它们的实例一样，向它们发送消息。在消息表达式中，类消息的接收者总是一个类对象。

2. 将 initWithArray:消息发送给未初始化的类实例，以根据 args 数组对对象本身进行初始化，并返回一个自身的引用。

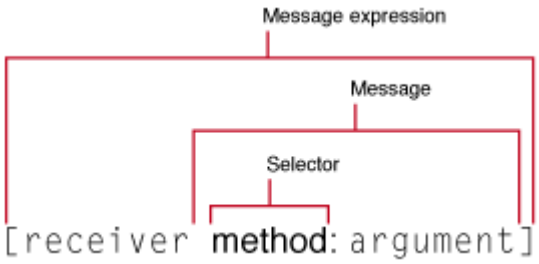
接下来考虑 SimpleCocoaTool 工程中 main 例程中的如下语句：

```
NSArray *sorted_args = [[cset allObjects]
sortedArrayUsingSelector:@selector(compare:)];
```

这个消息表达式中值得注意的是 sortedArrayUsingSelector:消息的参数。该参数要求使用编译器导向符@selector 来创建一个选择器。选择器是一个名称，在 Objective-C 运行环境中用于唯一标识一个接收者的方法，这个名称包含消息名的所有部分，包括冒号，但是不包括其它部分，比如返回类型或参数类型。

让我们暂停一下，回顾一下消息和方法的专用术语。方法本质上就是类定义和实现的函数，消息接收者是该类的实例。消息是一个与参数结合在一起的选择器，消息发送给接收者后导致对方法的调用。消息表达式同时包含接收者和消息。图 2-2 对这些关系进行描述：

图 2-2 消息的专用术语



Objective-C 使用了很多在 ANSI C 中找不到的类型和常量（literal）。在某些情况下，这些类型和常量会代替 ANSI C 的对应部分。表 2-1 描述一些重要的类型，包括每个类型允许使用的常量。

表 2-1 Objective-C 定义的重要类型和常量

类型	描述和文字
id	动态对象类型，否定常量为 nil。
Class	动态类的类型，否定常量为 Nil。
SEL	选择器的数据类型（typedef）。和 ANSI C 一样，这种类型的否定常量为 NULL。
BOOL	布尔类型。允许的值为 YES 和 NO。

在程序的控制流程语句中，您可以通过测试正确的否定常量来确定处理流程。举例来说，下面的 while 语句来自 SimpleCocoaTool 工程的代码，它隐式测试了 word 对象，以判断返回对象是否存在（或者从另一个角度看，测试是否不等于 nil）：

```
while (word = [enm nextObject]) {

    printf("%s\n", [word UTF8String]);

}
```

在 Objective-C 中，您可能经常向 nil 发送消息而没有副作用。运行环境保证发给 nil 的消息的返回值和其它类型的返回值对象一样是可以工作的。

SimpleCocoaTool 代码中最后需要注意的是一些 Objective-C 的初学者不容易注意到的东西。请对比下面的语句：

```
NSEnumerator *enm = [sorted_args objectEnumerator];
```

和：

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
```

表面上它们是做同样的事情，两者都返回一个对象的引用。然而，在返回对象的所有权以及由此引出的谁负责释放该对象的问题上有一个重要的语义差别。在第一个语句中，SimpleCocoaTool程序并不拥有返回对象；而在第二个语句中，程序创建了对象，并因此拥有了该对象，程序最后需要做的是向已创建的对象发送 release信息，从而释放该对象。其它只有一个显式创建的对象（NSCountedSet实例）也在程序的结束部分显式释放了。有关对象所有权和对象清理的策略，以及使用什么方法执行这种策略的信息，请参见 "Cocoa对象的生命周期"部分。

根类

仅凭 Objective-C 语言和运行环境并不足以构造哪怕是最简单的面向对象的程序，至少是不容易的。还缺少一些东西：即所有对象公有的基本行为和接口的定义。根类正是提供了这些定义。

之所以叫根类，是因为它位于整个类层次（这里是指 Cocoa 的类层次）的根上。根类不从其它类继承，但是类层次中的所有其它类都最终从根类继承下来。根类连同 Objective-C 语言，是 Cocoa 直接访问 Objective-C 运行环境或与之交互的基本途径。Cocoa 对象的大部分对象行为能力都是从根类得到的。

Cocoa 提供了两个根类：NSObject 和 NSProxy。Cocoa 将后者定义为抽象类，用于表示其它对象的替身对象。因此 NSProxy 类在分布式对象架构中是很重要的。由于作用比较特别，NSProxy 在 Cocoa 程序中出现频率很低。Cocoa 开发者在提到根类时，几乎总是指 NSObject。

本部分将讨论 NSObject 类，看看它如何与运行环境进行交互，以及它为所有 Cocoa 对象定义的基本行为和接口。其中主要是它为对象的内存分配、初始化、内存管理、内省、以及运行环境支持所声明的方法。这些概念是理解 Cocoa 的基础。

本部分包含如下内容：

NSObject

NSObject 是大多数 Objective-C 类层次的根类，它没有超类。其它类从 NSObject 继承访问 Objective-C 语言运行时系统的基本接口，它们的实例可以得到对象行为的能力。

虽然 NSObject 不是一个严格的抽象类，但它是个虚类。仅凭一个 NSObject 实例除了作为一个简单的对象外，不能完成任何有用的工作。为了在您的程序中加入特有的属性和逻辑，必须创建一个或多个从 NSObject 或其派生类继承下来的类。

NSObject采纳了NSObject协议（参见 "根类—和协议"部分）。NSObject协议支持多个根对象。举例来说，NSProxy是另一个根类，它不是继承自NSObject，但采纳了NSObject协议，以便和其它Objective-C对象共用一个公共的接口。

NSObject 和 java.lang.Object 一起，是 Java 版本的 Cocoa 中所有类的根类，包括 Foundation 和 Application Kit。

根类—和协议

NSObject 不仅仅是一个类的名称，还是一个协议的名称。两者对于定义一个 Cocoa 对象都是必要的。NSObject 协议指定了 Cocoa 中*所有*根类必须的基本编程接口，因此不仅 NSObject 类采纳了这个同名的协议，其它根类也采纳这个协议，比如 NSProxy。NSObject 类进一步指定了不作为代理对象的 Cocoa 对象的基本编程接口。

NSObject 及类似的协议用于 Cocoa 对象的总体定义（而不是在类接口中包含那些协议），使多个根类成为可能。每个根类共用一个由它们采纳的协议定义的公共接口。

在另一种意义上，NSObject 不仅仅是个“根”协议。虽然 NSObject 类没有正式采纳 NSCopying、NSMutableCopying、和 NSCoding 协议，但它声明和实现了与那些协议相关的方法（而且，包含 NSObject 类的 NSObject.h 头文件中也包含上面提到的所有四个协议的定义）。对象拷贝、编码、和解码是对象行为的基本部分。很多子类（如果不是绝大多数的话）都希望采纳和遵循这些协议。

请注意：其它 Cocoa 类可以（而且确实是）通过范畴将方法添加到 NSObject 中。这些范畴通常是一些非正式的协议，在委托中使用。它们允许委托对象选择实现范畴中的部分方法。然而，NSObject 的范畴并不被认为是基本对象接口的一部分。

根类方法概述

NSObject 根类和它采纳的 NSObject 协议及其它“根”协议一起，为所有不作为代理对象的 Cocoa 对象指定了如下的接口和行为特征：

- **分配、初始化、和复制**。NSObject 类中的一些方法（包括一些来自协议的方法）用于对象的创建、初始化、和复制：
  - alloc 和 allocWithZone:方法用于从某内存区域中分配一个对象内存，并使对象指向其运行时的类定义。
  - init 方法是对象初始化原型，负责将对象的实例变量设置为一个已知的初始状态。initialize 和 load 是两个类方法，它们让对象有机会对自身进行初始化。
  - new 是一个将简单的内存分配和初始化结合起来的便利方法。
  - copy 和 copyWithZone:方法用于拷贝实现这些（由 NSCopying 协议定义的）方法的类的实例。希望支持可变对象拷贝的类则需要实现 mutableCopy 和 mutableCopyWithZone:（由 NSMutableCopying 协议定义）方法。

  更多信息请参见 "对象的创建"部分。
- **对象的保持和清理**。下面的方法对面向对象程序的内存管理特别重要：

- retain 方法增加对象的保持次数。
- release 方法减少对象的保持次数。
- autorelease 方法也是减少对象的保持次数，但是以推迟的方式。
- retainCount 方法返回对当前的保持次数。
- dealloc 方法由需要释放对象的实例变量以及释放动态分配的内存的类实现。

更多信息请参见"Cocoa 对象的生命周期"。

- **内省和比较**。NSObjec 有很多方法可以查询对象的运行时信息。这些内省方法有助于找出对象在类层次中的位置，确定对象是否实现特定的方法，以及测试对象是否遵循某种协议。这些方法中的一部分仅实现为类方法。
  - superclass 和 class 方法（实现为类和实例方法）分别以 Class 对象的形式返回接收者的超类和类。
  - 您可以通过 isKindOfClass:和 isMemberOfClass:方法来确定对象属于哪个类。后者用于测试接收者是否为指定类的实例。isSubclassOfClass:类方法则用于测试类的继承性。
  - respondsToSelector:方法用于测试接收者是否实现由选择器参数标识的方法。instancesRespondToSelector:类方法则用于测试给定类的实例是否实现指定的方法。
  - conformsToProtocol:方法用于测试接收者（对象或类）是否遵循给定的协议。
  - isEqual:和 hash 方法用于对象的比较。
  - description 方法允许对象返回一个内容描述字符串；这个方法的输出经常用于调试（“print object”命令），以及在格式化字符串中和“%@”指示符一起表示对象。

  更多信息请参见 "内省"部分。
- **对象的编码和解码**。下面的方法和对象的编解码（作为归档过程的一部分）有关：
  - encodeWithCoder:和 initWithCoder:是 NSCoding 协议仅有的方法。前者使对象可以对其实例变量进行编码，后者则使对象可以根据解码过的实例变量对自身进行初始化。
  - NSObject 类中声明了一些于对象编码有关的方法：classForCoder:、replacementObjectForCoder:、和 awakeAfterUsingCoder:。

  进一步信息请参见 *Cocoa的归档和序列化编程指南*一文。
- **消息的转发**。forwardInvocation:和相关的方法允许一个对象将消息转发给另一个对象。
- **消息的派发**。以 performSelector...开头的一组方法使您可以在指定的延迟后派发消息，以及将消息从辅助线程派发（同步或异步）到主线程。

NSObject 还有几个其它的方法，包括一些处理版本和姿态（后者使一个类在运行时将自己表示为另一个类）的类方法，以及一些访问运行时数据结构的方法，比如方法选择器和指向方法实现的函数指针。

接口规范

某些NSObject方法只是为了被调用，而另一些方法则是为了被重载。举例来说，大多数子类不应该重载allocWithZone:方法，但必须实现init方法—至少需要实现一个最终调用根类的init方法（请参见 "对象的创建"部分）的初始化方法。对于那些期望子类重载的方法，NSObject的实现或者什么也不做，或者返回一个合理的值，比如self。这些缺省实现使我们有可能向任意的Cocoa对象—甚至是没有重载这些方法的对象—发送诸如init这样得基本消息，而又不必冒运行时例外的风险。在发送消息之前，不必进行检查（通过respondsToSelector:方法）。更加重要的是，NSObject的这些“占位”方法为Cocoa对象定义了一个公共的结构，并建立了一些规则，如果所有的对象都遵循这些规则，对象间的交互将更加可靠。

实例方法和类方法

运行环境系统以一种特殊的方式处理根类定义的方法。根类定义的实例方法可以由实例对象和类对象执行，因此所有类对象都可以访问根类定义的实例方法。对于任何类对象，如果对象中不包含同名的类方法，就可以执行根类的所有实例方法。

举例来说，一个类对象可以通过发送消息来执行 NSObject 的 respondsToSelector:和 performSelector:withObject:实例方法：

```
SEL method = @selector(riskAll:);



if ([MyClass respondsToSelector:method])

    [MyClass performSelector:method withObject:self];
```

请注意，只有根类中定义的实例方法才可以在类对象中使用。在上面的例子中，如果 MyClass 重新实现了 respondsToSelector:或者 performSelector:withObject:方法，则那些新的版本将只能用于实例对象。MyClass 的类对象只能执行 NSObject 类定义的版本（当然，如果 MyClass 将 respondsToSelector:或 performSelector:withObject: 实现为类方法，而不是实例方法，则该类对象可以执行这些新的实现）。

Cocoa对象的生命周期

Cocoa 对象的生命周期（至少是潜在地）跨越不同的阶段。它需要被创建、初始化、和使用（就是其它对象向它发送消息），它可能被保持、拷贝、或者归档，并最终被释放和销毁。下面的讨论将给出一个典型对象的生命周期框图，但仍然不涉及太多的细节。

让我们从最后开始，即从清理对象的方式开始讨论。和其它编程语言不同，Objective-C 没有自动释放不再使用的对象的“垃圾收集”设施。垃圾收集开销大而且不灵活，取而代之的是，Cocoa 和 Objective-C 选择一种主动的、策略驱动的例程来保持对象，并在不再需要的时候进行清理。

这种例程和策略建立在引用计数的基础上。每个 Cocoa 对象都带有一个整数，用于指示对其持久性感兴趣的其它对象（甚至是例程代码的现场）的数目。这个整数被称为对象的*保持数*（“保持” 是为了避免和“引用”重复）。当您通过 alloc 或者 allocWithZone:类方法创建对象的时候，Cocoa 做了一些非常重要的工作：

- 它将对象的isa指针—NSObject类中唯一的公共实例变量—指向对象的类，因此将对象集成到类层次的运行时视图中（进一步信息请参见 "对象的创建"部分）。

- 它将对象的保持数—一个隐藏的实例变量，所有对象都有—设置为 1（这里假定对象的创建者对其持久性感兴趣）。

为对象分配内存之后，您通常需要将其实例变量设置为合理的初始值，以便进行初始化（NSObject 声明了 init 方法作为这个目的的原型）。这样对象就可以使用了，您可以向它发送消息，将它传递给其它对象，等等。

请注意：由于除了显式分配的对象之外，初始化方法也可以返回一个对象，因此习惯上将 alloc 消息嵌套在 init 消息（或其它初始化方法）中—举例来说：

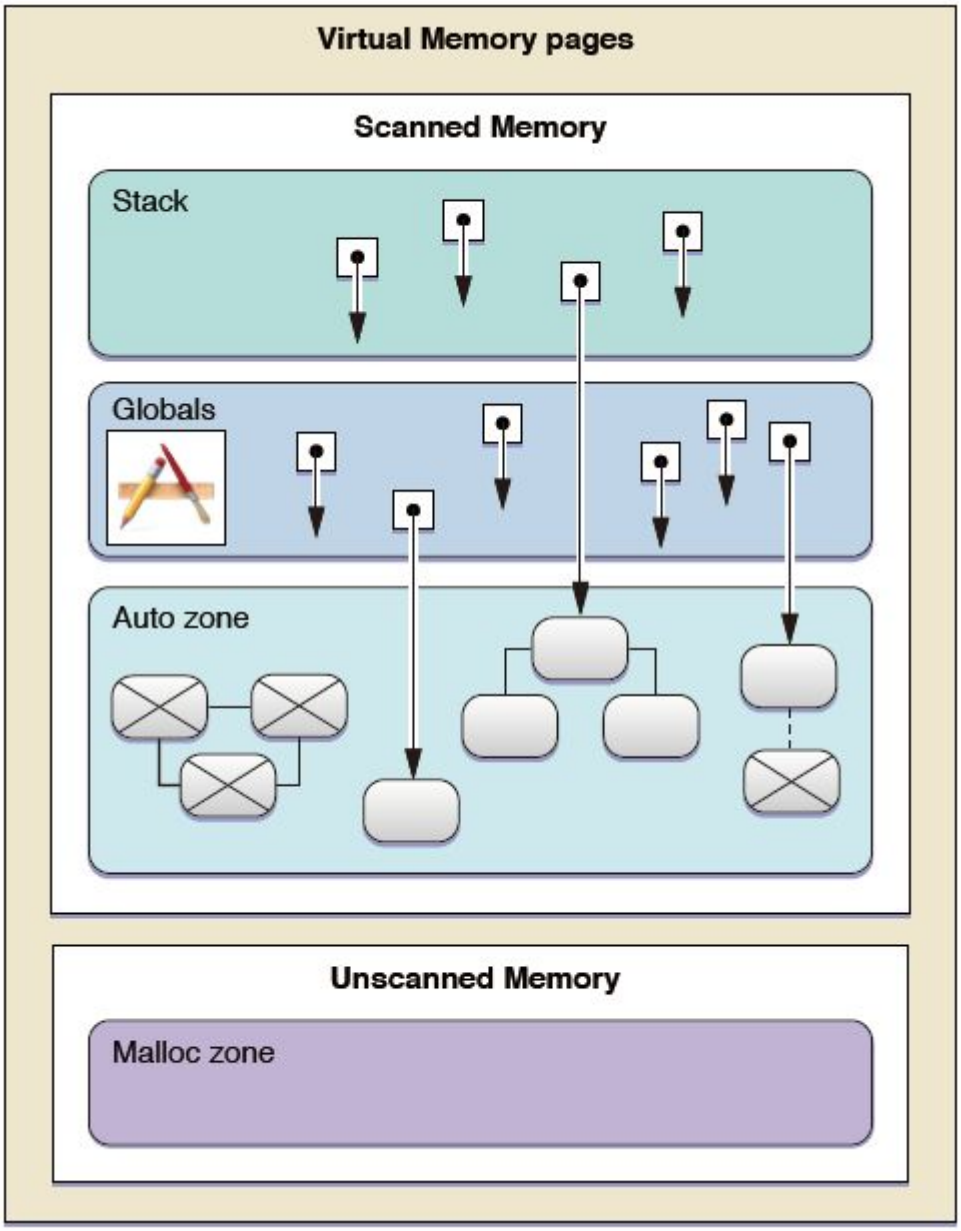
```
id anObj = [[MyClass alloc] init];
```

当您释放一个对象—也就是向对象发送一个 release 消息时—NSObject 会减少它的保持数。如果保持数从 1 下降到 0，对象就会被解除分配。对象的解除分配有两个步骤：首先是对象的 dealloc 方法被调用，以释放实例变量和动态分配的内存；然后是操作系统将对象的本身销毁，并回收对象占用的内存。

重要提示：您永远不应该直接调用对象的 dealloc 方法。

如果您不希望对象很快消失，该怎么办呢？如果您在创建对象之后向它发送一个 retain 消息，对象的保持数就会增加到 2。这样，就需要两个 release 消息才能导致对象的释放。图 2-3 描述了这种极为简单的场景。

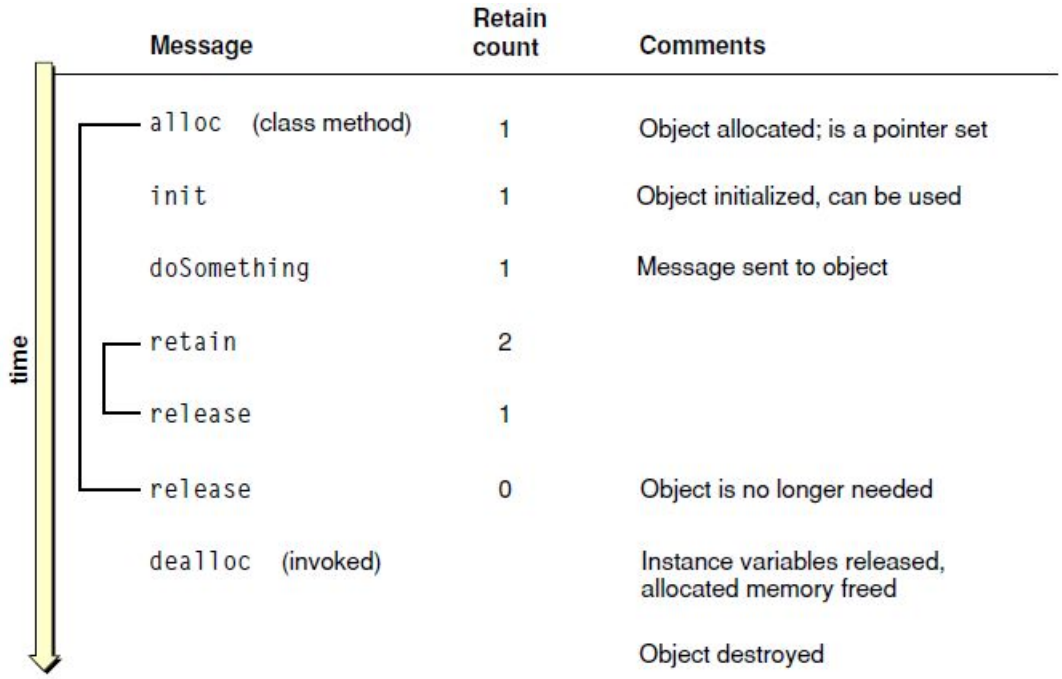
图 2-3 对象的生命周期—简化视图



当然，在这个场景下，对象的创建者不需要保持对象，它已经拥有对象了。但是，如果这个创建者要将对象传递给消息中的另一个对象，则情况就不一样了。在 Objective-C 程序中，人们假定从其它对象接收到的对象在其被得到的作用域内总是正当的。负责接收的对象可以向被接收的对象发送消息，而且可以将它传递给其它对象。这个假设要求对象的发送者“行为规矩”，不要在客户对象仍然拥有被发送对象的引用时将它过早释放。

如果客户对象在程序的作用域之外希望保持接收到的对象，则可以保持该对象—也就是向它发送一个 `retain` 消息。保持一个对象会增加该对象的保持数，从而表示希望拥有该对象。客户对象有责任在一段时间后释放该对象。如果对象的创建者将该对象释放，但同时有一个客户对象已经保持了该对象，则该对象会一直持续到客户对象将它释放为止。图 2-4 说明了这个序列：

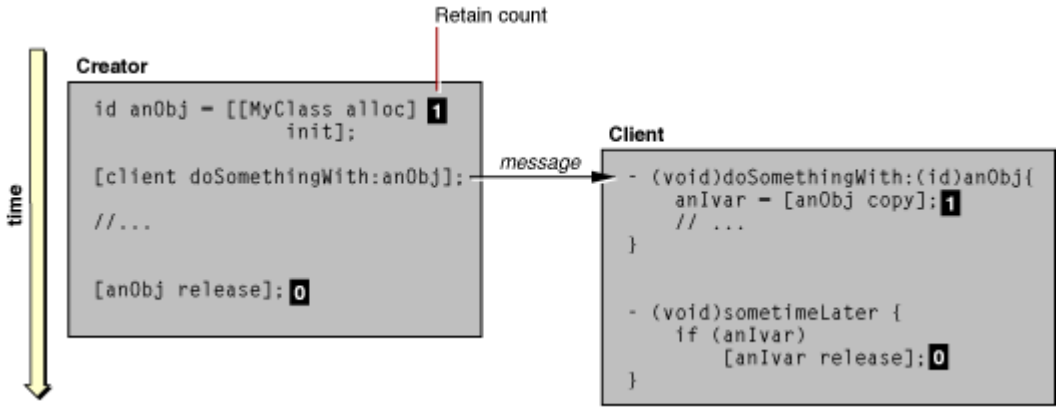
图 2-4 保持接收到的对象



您可以不保持对象，而是通过发送 copy 或 copyWithZone:消息来对其进行拷贝（很多子类—如果不是大多数的话—都封装了某种数据采纳方法，或遵循这种协议）。拷贝一个对象不仅仅是对其进行复制，而且几乎总是将它的保持数设置为 1（请参见图 2-5）。根据对象的本质和可能的用法，拷贝可以是浅拷贝，也可以是深拷贝。深拷贝将对象复制为被拷贝对象的一个实例变量，而浅拷贝只是复制那些实例对象的引用。

在用法方面，copy 和 retain 的区别在于前者要求成为对象新的、唯一的拥有者；新的拥有者可以修改拷贝后的对象，而不考虑其原始对象。一般地说，您需要对值对象（即对某些简单的值进行封装的对象）进行拷贝，而不是保持。特别是当对象是可变的时候，比如一个 NSMutableString 对象。对于不可变对象，copy 和 retain 可能是等价的，其实现方法也是类似的。

图 2-5 拷贝接收到的对象

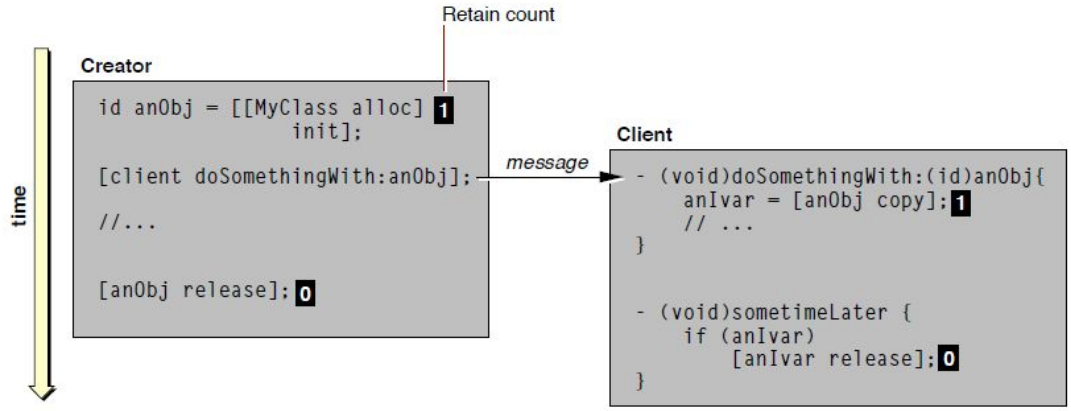


您可能已经注意到，用这种策略管理对象的生命周期有一个潜在的问题，就是创建一个对象并将它传递给另一个对象的对象本身并不总是知道什么时候可以安全地释放对象。在调用堆栈中可能有多个该对象的引

用，某些引用可能来自创建者不知道的对象。如果创建者对象释放了其所创建的对象，而其它对象向这个已经被销毁的对象发送消息，程序就会崩溃。为了解决这个问题，Cocoa 引入了一种延迟对象释放的机制，称为自动释放（autoreleasing）机制。

自动释放机制通过自动释放池（由NSAutoreleasePool类定义）来实现。自动释放池是位于显式定义的作用域内的一个对象集合，该作用域被标志为最后释放。自动释放池可以嵌套。当您向一个对象发送一个 autorelease消息时，Cocoa就会将该对象的一个引用放入到最新的自动释放池。它仍然是个正当的对象，因此自动释放池定义的作用域内的其它对象可以向它发送消息。当程序执行到作用域结束的位置时，自动释放池就会被释放，池中的所有对象也就被释放（参见图 2-6）。如果您正在开发应用程序，可能不需要建立一个自动释放池，Application Kit会自动建立一个自动释放池，其作用域为为应用程序的事件周期。

图 2-6 自动释放池



到目前为止，对象生命周期的讨论主要关注整个周期中的对象管理机制。但是，指导如何使用这些机制的是对象的所有权策略。这个策略可以概括如下：

- 如果您通过分配和初始化（比如[[MyClass alloc] init]）的方式来*创建*对象，您就拥有这个对象，需要负责该对象的释放。这个规则在使用 NSObject 的便利方法 new 时也同样适用。
- 如果您*拷贝*一个对象，您也拥有拷贝得到的对象，需要负责该对象的释放。
- 如果您*保持*一个对象，您就部分拥有这个对象，需要在不再使用时释放该对象。

反过来，

- 如果您从其它对象那里接收到一个对象，则您不拥有该对象，也不应该释放它（这个规则有少数的例外，在参考文档中有显式的说明）。

和其它规则一样，这些策略也有一些例外和经常出错的地方:

- 如果您通过类工厂方法来创建对象（比如 NSMutableArray arrayWithCapacity:方法），则可以假定您接收到的对象已经自动被放到自动释放池了。您不应该自行将它释放，如果您需要保持该对象，则应该保持（retain）它。
- 为了避免循环引用，子对象不能保持它的父对象（父对象是该子对象的创建者，或者将该子对象作为实例变量持有的对象）。

请注意：在上面的原则中提到的“释放”是指向对象发送一个 release 或 autorelease 消息。

如果您没有遵循这个所有权的策略，则可能导致您的 Cocoa 程序出现两种不好的结果：由于没有释放自己创建、拷贝、或保持的对象，您的程序会发生内存泄露；或者，由于向已经解除分配的对象发送消息，您的程序发生崩溃。而且还会有进一步的问题：调试这些问题可能相当费时间。

对象的生命周期中可能发生的另一个基本事件是归档。归档是将组成一个面向对象程序中的相关对象形成的网状结构—对象图—转化为一种可持久的形式（通常是一个文件），该形式可以保存对象图中对象的标识和彼此之间的关系。在解档时，可以通过这个档案重新构造出程序的对象图。为了参与归档（和解档），对象必须支持通过 NSCoder 类定义的方法对实例变量进行编码（和解码）。为了这个目的，NSObject 采纳了 NSCoding 协议。有关对象归档的更多信息，请参见"对象的归档"部分。

进一步阅读： 这个 Cocoa 对象的生命周期概述揭示了这个主题的一些表面的东西。关于 Cocoa 对象内存管理的更详细讨论，请参见 *Objective-C 编程语言*一书中的“Objective-C 运行系统”部分，以及 *Cocoa 内存管理编程指南*中的内容。

对象的创建

Cocoa 对象的创建总是分成两个阶段：对象分配和初始化，缺少其中的任何一个步骤，对象通常都不可用。虽然在几乎所有的情况下，初始化总是紧接在对象分配之后，但是在建立对象的过程中，这两个操作的作用是不同的。

本部分包含如下内容：

分配一个对象

当您分配一个对象时，Cocoa 进行的工作可以部分地由“分配”这个术语看出来。Cocoa 会从应用程序的虚存区中为对象分配足够的内存。在计算需要分配多少内存时，Cocoa 会考虑对象的实例变量，包括它们的类型和顺序，这些信息由对象的类来定义。

为了进行对象分配，您需要向对象的类发送 alloc 或 allocWithZone:消息。在消息的返回值中可以得到一个“生的”（未初始化的）类实例。alloc 方法使用应用程序缺省的虚存区。区是一个按页对齐的内存区域，用于存放应用程序分配的对象和数据。有关区的更多信息请参见 *Cooca 内存管理编程指南*。

除了分配内存之外，Cocoa 的分配（allocation）消息还进行其它一些重要的工作：

- 将对象的保持数设置为 1（如 "Cocoa对象的生命周期"部分所述）。
- 使初始化对象的 isa 实例变量指向对象的类。对象类是一个根据类定义编译得到的运行时对象。
- 将其它所有的实例变量初始化为 0（或者与 0 等价的类型，比如 nil、NULL、和 0.0）。

对象的 isa 实例变量是从 NSObject 继承下来的，因此所有的 Cocoa 对象都有。在将 isa 指针指向对象类之后，对象就被集成到继承层次的运行时视图和构成程序的对象（类和实例）网络中了。其结果是对象可以找到它所需要的所有运行时信息，比如其它对象在继承层次上的位置，它们遵循的协议，以及在响应消息时可以执行的方法实现的位置。

总之，分配过程不仅进行对象的内存分配，而且还初始化对象的两个小而又非常重要的属性：即它的 isa 实例变量和保持数。它还将所有剩下的实例变量设置为 0。但是分配完成的对象还是不可用，还需要调用象 init 这样的初始化方法来进行对象自有的初始化，才能返回一个可用的对象。

初始化一个对象

初始化过程将对象的实例变量设置为合理而有用的初始值，还可以分配和准备对象需要的其它全局资源，并在必要时装载诸如文件这样的资源。声明实例变量的所有对象都应该实现一个初始化方法－除非将所有变量都置为 0 的缺省初始化已经足够。如果一个对象没有实现自己的初始化方法，Cocoa 就会调用其最近的祖先对象的方法。

初始化方法的形式

NSObject 声明了 init 方法作为初始化方法的原型，它是一个实例方法，返回一个类型为 id 的对象。对于不需要初始化其它数据的子类，重载 init 方法就可以了，但是常见的情况是初始化阶段需要根据外部的数据来设置对象的初始状态。举例来说，假定您有一个 Account 类，正确地初始化一个 Account 对象需要一个唯一的账号，而这个号码必须提供给初始化方法，这样初始化方法可能需要接收一或多个参数。唯一的要求是初始化方法必须以“init”字母开头（有时用格式规则描述 init... 来表示初始化方法）。

请注意：子类可以不采用带参数的初始化方法，而是实现一个简单的 init 方法，并在初始化后马上使用“set”存取方法，将对象设置为有用的初始状态（存取方法通过设置和获取实例变量的值，强制进行对象数据的封装）。

Cocoa 有很多带参数的初始化方法，下面是几个例子（括号里面是对应的类）：

- - (id)initWithArray:(NSArray *)array; (from NSSet)
- - (id)initWithTimeInterval:(NSTimeInterval)secsToBeAdded sinceDate:(NSDate *)anotherDate; (from NSDate)
- - (id)initWithContentRect:(NSRect)contentRect styleMask:(unsigned int)aStyle backing:(NSBackingStoreType)bufferingType defer:(BOOL)flag; (from NSWindow)
- - (id)initWithFrame:(NSRect)frameRect; (from NSControl and NSView)

这些初始化方法都是以“init”字母开头，返回类型为 id 的动态类型对象的实例方法。此外，它们还遵循 Cocoa 的多参数方法规则，通常在第一个和最重要的参数之前使用 With*Type*：或者 From*Source*：名称。

初始化方法的问题

虽然 init... 方法的方法签名要求它们返回一个对象，但是返回的并不一定是最近分配的对象，即不一定是 init... 消息的接收者对象。换句话说，您从初始化方法得到的对象可能不是您认为的、正在被初始化的对象。

有两种情况提示初始化方法的返回值不是刚刚分配的对象，第一种情况需要两个条件：必须是单件实例，或者对象的定义属性必须是唯一的。某些Cocoa类，比如NSWorkspace，在一个程序中只能有一个实例。在这种情况下，类（在初始化方法或者更可能在类工厂方法中）必须保证只创建一个实例，并在其它对象请求新的实例时将它返回（实现单件对象的信息请参见 "创建一个单件对象" 部分）。

当一个对象需要保证某个属性唯一的时候，也会出现类似的情况。回忆一下我们在早些时候假定的 Account 类，所有类型的账号都必须有唯一的标识。如果这个类的初始化方法，假定是 initWithAccountID:，传入一个已经和某个对象相关联的标识，那么它必须做下面的两个工作：

- 释放刚刚分配的对象。
- 返回先前用这个唯一标识初始化的 Account 对象。

这样，初始化方法既确保了标识唯一性，又提供了被请求的对象，即一个具有指定标识的 Account 实例。

有些时候，init...方法不能执行其它对象请求的初始化。举例来说，initFromFile:方法希望根据一个文件的内容来初始化对象，文件的路径作为参数传入。但是如果在指定的地方不存在该文件，该对象就不能被初始化。如果传给 initWithArray:方法的是一个 NSDictionary 对象，而不是 NSArray 对象，也会发生类似的问题。 当一个 init...方法不能对对象进行初始化时，它应该：

- 释放刚刚分配的对象。
- 返回 nil。

从初始化方法返回 nil 表示不能创建被请求的对象。在创建对象时，您通常应该在处理之前检查返回值是否为 nil：

```
id anObject = [[MyClass alloc] init];

if (anObject) {

    [anObject doSomething];

    // more messages...

} else {

    // handle error

}
```

由于 init...方法可能返回 nil 或者不同于显式分配的对象,所以使用 alloc 或 allocWithZone:方法而不是初始化方法返回的对象是有危险的。考虑下面的代码：

```
id myObject = [MyClass alloc];

[myObject init];

[myObject doSomething];
```

这个例子中的 init 方法可能返回 nil，或者将刚刚分配的对象代替为不同的对象。由于您可以向 nil 发送消息而不引起例外， 所以在前面的代码中不会出现问题，如果不考虑（可能）需要进行痛苦的调试的话。但您总是应该依赖于初始化过的实例，而不是一个“生的”、刚刚分配完成的实例。我们推荐您将分配和初始化消息嵌套在一起，并在处理之前测试初始化方法返回的对象。

```
id myObject = [[MyClass alloc] init];

if ( myObject ) {
```

```
    [myObject doSomething];
} else {
    // error recovery...
}
```

一旦对象被初始化了，就不应该再进行初始化。如果您试图进行重复初始化，实例化对象的框架类通常会产生一个例外。举例来说，下面这个例子中的初始化会导致程序产生 NSInvalidArgumentException 例外。

```
NSString *aStr = [[NSString alloc] initWithString:@"Foo"];
aStr = [aStr initWithString:@"Bar"];
```

实现一个初始化方法

实现一个init... 方法，使之作为类的唯一初始化方法或者具有多个初始化方法的类的*指定初始化方法*时（参见"多个初始化方法和指定初始化方法"部分的描述），有如下几个关键步骤：

- 总是*首先*调用超类（super）的初始化方法。
- 检查超类返回的对象。如果是 nil，则初始化不能进行，需要向接收者对象返回 nil。
- 在初始化实例变量时，如果它们是其它对象的引用，则在必要时进行保留和拷贝。
- 将实例变量设置为正当的初始值之后，就返回 self，除了下列的情况：
  - 需要返回一个代替对象，在这种情况下，首先释放新分配的对象。
  - 某些问题导致不能成功初始化，这时需要返回 nil。

列表 2-3 中的 init... 方法说明了这些步骤：

列表 2-3 初始化方法的实例

```
- (id)initWithAccountID:(NSString *)identifier {
    if ( self = [super init] ) {
        Account *ac = [accountDictionary objectForKey:identifier];
        if (ac) { // object with that ID already exists
            [self release];
            return [ac retain];
        }
        if (identifier) {
            accountID = [identifier copy]; // accountID is instance variable
```

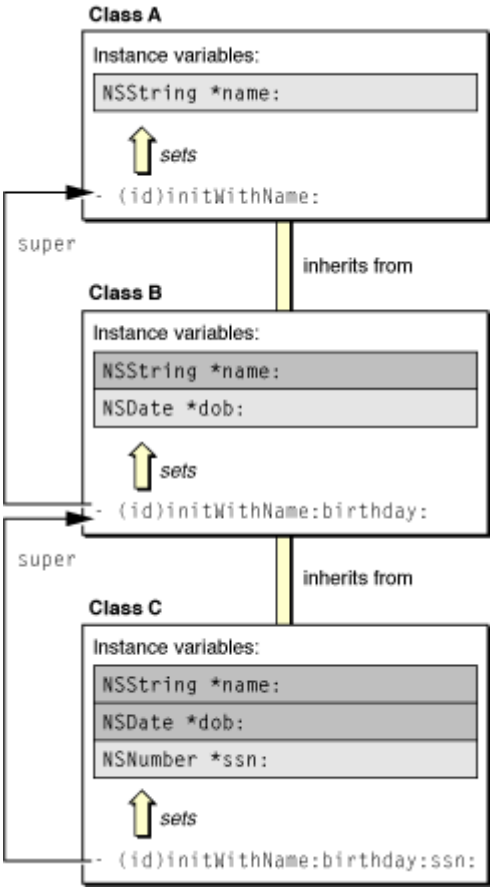
```
            [accountDictionary setObject:self forKey:identifier];
            return self;
        } else {
            [self release];
            return nil;
        }
    } else
        return nil;
}
```

请注意：虽然为了描述的简洁，这个例子在参数为 nil 的时候返回 nil，但是更好的 Cocoa 实践时产生一个例外。

没有必要显式初始化对象中所有的实例变量，只要初始化对象正常工作必须的那些变量就可以了。在对象分配时将实例变量设置为 0 的缺省初始化通常就足够了。请务必根据具体的需要对实例变量执行保持或拷贝操作。

首先调用超类的初始化方法是非常重要的。回忆一下，对象不仅封装了其所属类定义的实例变量，而且也封装了所有祖先类定义的实例变量。首先调用 super 的初始化方法可以确保继承链上方的类定义的实例变量都率先得到初始化。对象的直接超类在其初始化方法中会调用它自身超类的初始化方法，而该方法又会调用超类的主 init... 方法，以此类推（参见图 2-7）。按正确的顺序进行初始化是很关键的，因为超类后来进行的初始化可能建立在之前超类定义的实例变量已经被初始化为合理值的基础上。

图 2-7 继承链的初始化



在创建子类时需要关注通过继承得到的初始化方法。有些时候，超类的 init... 方法已经为您的类做好足够的初始化，但是更多的可能是没有做好的，因此您应该对其进行重载。如果没有重载，被调用的是超类的实现。由于超类完全不了解您创建的类，所以您的类实例可能没有被正确初始化。

多个初始化方法和指定初始化方法

一个类可以定义多个初始化方法。有些时候，多个初始化方法让客户类可以以不同形式的输入进行同样的初始化。举例来说，NSSet 类就为其客户提供几个可以接受不同形式数据的初始化方法，其中一个可以接受 NSArray 对象，另一个可以接受数目确定的元素列表，还有一个可以接受以 nil 结尾的元素列表：

```
- (id)initWithArray:(NSArray *)array;

- (id)initWithObjects:(id *)objects count:(unsigned)count;

- (id)initWithObjects:(id)firstObj, ...;
```

某些子类提供一些便利的初始化方法，为具有完整初始化参数表的初始化方法提供缺省值。那种初始化方法通常是指定的初始化方法，也是类中最重要的初始化方法。举例来说，假定有一个 Task 类用如下的方法签名声明了一个指定初始化方法：

```
- (id)initWithTitle:(NSString *)aTitle date:(NSDate *)aDate;
```

Task 类可能会包含一些辅助或便利的方法，这些方法简单地调用指定的初始化方法，并为辅助初始化方法中没有显式要求的参数传入缺省值（列表 2-4）。

列表 2-4 辅助初始化方法

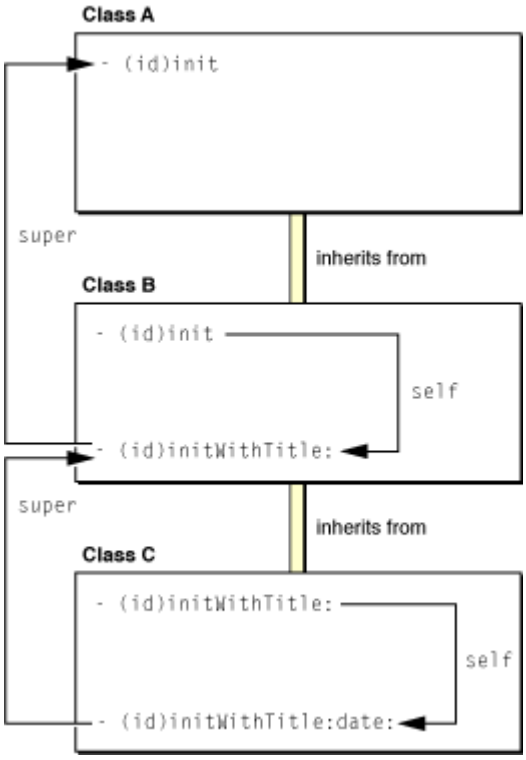
```
- (id)initWithTitle:(NSString *)aTitle {

    return [self initWithTitle:title date:[NSDate date]];

}



- (id)init {

    return [self initWithTitle:@"Task"];

}
```

指定初始化方法对类是很重要的，它确保被继承的实例变量可以通过调用 super 的初始化方法来进行初始化。它通常是具有最多参数、执行最多初始化工作的 init... 方法，也是辅助初始化方法通过向 self 发送消息进行调用的初始化方法。

您在定义子类时必须能够识别超类的指定初始化方法，并通过向 super 发送消息来对其进行调用。如果您觉得有必要，也可以提供便利的初始化方法。在设计类的初始化方法时，请记住，指定初始化方法是通过发向 super 的消息彼此链接在一起的，而其它的初始化方法则通过发向 self 的消息和其所属类的指定初始化方法链接在一起。

让我们举个例子来更清楚地说明这个问题。假定有三个类： A、B、和 C，B 继承自 A，C 又继承自 B。每个子类都以实例变量的形式增加一个属性，并实现一个 init... 方法（即指定初始化方法）来对这个实例变量进行初始化；同时还在必要时定义一些辅助初始化方法，并确保通过继承得到的初始化方法被重载。图 2-8 说明了所有三个类的初始化方法以及它们之间的关系。

图 2-8 辅助初始化方法和指定初始化方法之间的交互



每个类的指定初始化方法都是覆盖面最大的初始化方法，负责对子类新增的属性进行初始化。指定的初始化方法也是通过向 super 发送消息来调用超类的指定初始化方法的 init... 方法。在这个例子中，类 C 的指定初始化方法是 initWithTitle:date:方法，它调用了超类的指定初始化方法 initWithTitle:，该方法又调用了类 A 的 init 方法。在创建子类时，重要的一点是需要知道超类的指定初始化方法。

继承链上的指定初始化方法就这样通过发给 super 的消息链接起来了，同时辅助初始化方法也通过发给 self 的消息和其所属类的指定初始化方法相链接。辅助初始化方法（如这个例子所示）经常是通过继承得到的初始化方法的重载版本。类 C 重载了 initWithTitle:方法，以便调用自己的指定初始化方法，传入缺省的日期；而这个指定初始化方法反过来又调用类 B 的指定初始化方法 initWithTitle:，而该方法也就是被重载方法。如果您向类 B 和类 C 的对象发送 initWithTitle:消息，就会调用不同的方法实现。另一方面，如果类 C *没有*重载 initWithTitle:方法，您向类 C 的实例发送消息时调用的将是类 B 的实现，结果类 C 的实例就没有被完全初始化（因为缺少一个日期值）。在创建子类时，确保所有继承得到的初始化方法都被覆盖时很重要的。

有些时候，有可能超类的指定初始化方法对子类就足够了，因此子类没有必要实现自己的指定初始化方法。另一些时候，一个类的指定初始化方法可能是超类指定初始化方法的重载版本。当子类需要对超类的指定初始化方法所做的工作进行补充的时候，即使子类没有增加任何自己的实例变量（或者增加了自己的实例变量但不需要显式的初始化），也常常需要进行重载。

dealloc方法

在很多方面，一个类的 dealloc 方法都是和 init... 方法（特别是指定初始化方法）相呼应的。初始化方法在对象分配之后马上被调用，而 dealloc 在对象的销毁之前被调用；初始化方法确保对象的实例变量被正确初始化，而 dealloc 方法确保该对象的实例变量被释放，以及确保动态分配的内存被释放。

两者的最后一点相似之处在于必须调用各自的超类实现。在初始化方法中，您首先调用超类的指定初始化方法；在 dealloc 方法中，您则在*最后*一步调用超类的 ealloc 实现。这样做的原因是与初始化方法相反像，子类应该在祖先类的实例变量被释放之前释放自己拥有的实例变量。

列表 2-5 说明应该如何实现这个方法。

列表 2-5 dealloc 方法的实例

```
- (void)dealloc {

    [accountDictionary release];

    if ( mallocdChunk != NULL )

        free(mallocdChunk);

    [super dealloc];

}
```

请注意，在这个例子中，准备释放之前检查一个通过 malloc 得到的实例变量是否不为 NULL 有些过于谨慎。由于您可以安全地向一个 nil 对象发送消息，这样的谨慎是没有必要的。

类工厂方法

类工厂方法的实现是为了向客户提供方便，它们将分配和初始化合在一个步骤中，返回被创建的对象，并进行自动释放处理。这些方法的形式是+ (*type*)*className*...（其中 *className* 不包括任何前缀）。

Cocoa 提供很多类工厂的实例，特别是在“数值” 类中。NSDate 包括下面的类工厂方法：

```
+ (id)dateWithTimeIntervalSinceNow:(NSTimeInterval)secs;

+ (id)dateWithTimeIntervalSinceReferenceDate:(NSTimeInterval)secs;

+ (id)dateWithTimeIntervalSince1970:(NSTimeInterval)secs;
```

NSData 提供下面的工厂方法：

```
+ (id)dataWithBytes:(const void *)bytes length:(unsigned)length;

+ (id)dataWithBytesNoCopy:(void *)bytes length:(unsigned)length;

+ (id)dataWithBytesNoCopy:(void *)bytes length:(unsigned)length

      freeWhenDone:(BOOL)b;

+ (id)dataWithContentsOfFile:(NSString *)path;

+ (id)dataWithContentsOfURL:(NSURL *)url;

+ (id)dataWithContentsOfMappedFile:(NSString *)path;
```

工厂方法可能不仅仅为了方便使用。它们不但可以将分配和初始化合在一起，还可以为初始化过程提供对象的分配信息。举例来说，假定您必须根据一个属性列表文件来初始化一个集合对象（NSString 对象、NSData 对象、或者 NSNumber 对象等），属性列表文件包含任意数目的、经过编码的集合元素。在工厂方法确定应该为集合类分配多少内存之前，必须先读取文件并对属性列表进行解析，确定有多少元素，每个元素的类型是什么。

类工厂方法的另一个目的是使类（比如 NSWorkspace）提供单件实例。虽然 init... 方法可以确认一个类在每次程序运行过程只存在一个实例，但它需要首先分配一个“生的”实例，然后还必须释放该实例。工厂方法则可以避免为可能没有用的对象盲目分配内存（参见列表 2-6）。

列表 2-6 单件实例的工厂方法

```
static AccountManager *DefaultManager = nil;



+ (AccountManager *)defaultManager {

    if (!DefaultManager) DefaultManager = [[self allocWithZone:NULL] init];

    return DefaultManager;

}
```

进一步阅读：更多与 Cocoa 对象的分配和初始化有关的讨论请参见 *Objective-C 编程语言*中的“Objective-C

运行时系统”部分。

内省

内省（Introspection）是面向对象语言和环境的一个强大特性，Objective-C 和 Cocoa 在这个方面尤其的丰富。内省是对象揭示自己作为一个运行时对象的详细信息的一种能力。这些详细信息包括对象在继承树上的位置，对象是否遵循特定的协议，以及是否可以响应特定的消息。NSObject 协议和类定义了很多内省方法，用于查询运行时信息，以便根据对象的特征进行识别。

明智地使用内省可以使面向对象的程序更加高效和强壮。它有助于避免错误地进行消息派发、错误地假设对象相等、以及类似的问题。下面的部分将介绍如何在代码中有效地使用 NSObject 的内省方法。

本部分包括如下内容：

评估继承关系

一旦您知道一个对象属于什么类，就可能已经相当了解这个对象了。您可以知道它具有什么能力、哪些属性、以及可以响应哪些消息。即使在内省之后不能了解对象所属的类，也可以知道该对象不能响应特定的消息。

NSObject协议声明了几个方法，用于确定对象在类层次中的位置。这些方法在不同粒度上进行操作，比如class和 superclass实例方法分别返回代表类和超类的Class对象。使用这些方法需要将一个Class对象和另一个进行对比。列表 2-7 给出了一个简单（可能是没有价值）的用法实例。

列表 2-7 使用类和超类的方法

```
// ...

while ( id anObject = [objectEnumerator nextObject] ) {

    if ( [self class] == [anObject superclass] ) {

        // do something appropriate...

    }

}
```

请注意：有些时候您需要通过 class 或 superclass 方法得到正确的类消息接收者。

更加常见的是检查对象类的从属关系，这种情况下您需要向该对象发送 isKindOfClass:或 isMemberOfClass:消息。前一个方法返回接收者是否为给定类或其继承类的实例，isMemberOfClass:消息则告诉您接收者是否为指定类的实例。isKindOfClass: 方法通常更有用，因为通过它可以知道是否可以向该对象发送一系列消息。考虑列表 2-8 中的代码片断：

列表 2-8 使用 isKindOfClass:方法

```
if ([item isKindOfClass:[NSData class]]) {

    const unsigned char *bytes = [item bytes];

    unsigned int length = [item length];

    // ...

}
```

确定 *tem* 对象是 NSData 类的继承类的实例之后，代码就知道可以向它发送 NSData 的 bytes 和 length 消息。假定 *item* 是 NSMutableData 类的一个实例，则 isKindOfClass:和 isMemberOfClass:之间的差别就变得更加明显。如果您调用的是 isMemberOfClass:，而不是 isKindOfClass:，条件控制块中的代码将永远不会被执行，因为 *item* 并不是 NSData 类的实例，而是其子类 NSMutableData 的实例。

方法实现和协议遵循

NSObject还有两个功能更加强大的内省方法，即 respondsToSelector:和 conformsToProtocol:。这两个方法分别告诉您一个对象是否实现特定的方法，以及是否遵循指定的正式协议（即该对象是否采纳了该协议，且实现了该协议的所有方法）。

在代码中，您可以在类似的情况下使用这些方法。通过这些方法，您可以在将消息或消息集合发送给某些潜在的匿名对象之前，确定它们是否可以正确地进行响应。在发送消息之前进行检查可以避免由不能识别的选择器引起的运行时例外。在实现非正式协议（这种协议是委托技术的基础）时，Application Kit 就是在调用委托方法之前检查委托对象是否实现该方法（通过 respondsToSelector:方法）。

列表 2-9 显示了如何在代码中使用 respondsToSelector:方法。

列表 2-9 使用 respondsToSelector:方法

```
- (void)doCommandBySelector:(SEL)aSelector {

    if ([self respondsToSelector:aSelector]) {

        [self performSelector:aSelector withObject:nil];

    } else {

        [_client doCommandBySelector:aSelector];

    }

}
```

列表 2-10 显示如何在代码中使用 conformsToProtocol:方法：

列表 2-10 使用 conformsToProtocol:方法

```
// ...

if (!([((id)testObject) conformsToProtocol:@protocol(NSMenuItem)])) {
```

```
    NSLog(@"Custom MenuItem, '%@', not loaded; it must conform to the
        'NSMenuItem' protocol.\n", [testObject class]);
    [testObject release];
    testObject = nil;
}
```

对象的比较

hash和 isEqual:方法虽然不是严格的内省方法，但是可以发挥类似的作用，是进行对象的识别和比较时不可或缺的运行时工具。它们并不向运行环境查询对象信息，而是依赖于具体类的比较逻辑。

hash 和 isEqual:方法都在 NSObject 协议中声明，且彼此关系紧密。实现 hash 方法必须返回一个整型数，作为哈希表结构中的表地址。两个对象相等（isEqual:方法的判断结果）意味着它们有相同的哈希值。如果您的对象可能被包含在象 NSSet 这样的集合中，则需要定义 hash 方法，并确保该方法在两个对象相等的时候返回相同的哈希值。NSObject 类中缺省的 isEqual:实现只是简单地检查指针是否相等。

isEqual:的使用相当直接，它将消息的接收者和通过参数传入的对象进行比较。对象的比较常常可以在运行时决定应该对对象做些什么。如列表 2-11 所示，您可以通过 isEqual:来确定是否执行某一个动作。在这个例子中，动作是指保存被修改了的预置信息。

列表 2-11 使用 isEqual:方法

```
- (void)saveDefaults {
    NSDictionary *prefs = [self preferences];
    if (![origValues isEqual:prefs])
        [Preferences savePreferencesToDefaults:prefs];
}
```

如果您正在创建子类，则可能需要重载 isEqual:方法，以进一步检查对象是否相等。子类可能定义额外的属性，当两个实例被认为相等时，属性的值必须相同。举例来说，假定您创建一个名为 MyWidget 的 NSObject 子类，类中包含两个实例变量：name 和 data。当 MyWidget 的两个实例被认为是相等时，这些变量必须具有相同的值。列表 2-12 显示如何在 MyWidget 类中实现 isEqual:方法。

列表 2-12 重载 isEqual:方法

```
- (BOOL)isEqual:(id)other {
    if (other == self)
        return YES;
    if (!other || ![other isKindOfClass:[self class]])
```

```
        return NO;

    return [self isEqualToWidget:other];

}


- (BOOL)isEqualToWidget:(MyWidget *)aWidget {

    if (self == aWidget)

        return YES;

    if (![(id)[self name] isEqual:[aWidget name]])

        return NO;

    if (![[self data] isEqualToData:[aWidget data]])

        return NO;

    return YES;

}
```

isEqual:方法首先检查指针的等同性，然后是类的等同性，最后调用对象的比较器进行比较。比较器的名称指示出参与比较的对象的类名称。这种类型的比较器对传入的对象进行强制类型检查，是 Cocoa 中常见的约定，NSString 的 isEqualToString:和 NSTimeZone 的 isEqualToTimeZone:就是两个这样的例子。特定类的比较器（在这个例子中是 isEqualToWidget:）负责执行 name 和 data 变量的等同性。

在 Cocoa 框架的所有 isEqualTo*Type*:方法中，nil 都不是正当的参数，这些方法的实现在接收到 nil 参数时会抛出例外。然而为了向后兼容，Cocoa 框架中的 isEqual:方法可以接收 nil 值，在这种情况下返回 NO。

对象的可变性

Cocoa 对象或者是可变的，或者是不可变的。一旦创建之后，不可变对象封装的值就是不可改变的，且在整个对象的生命周期中都保持不变。但是对于可变对象，您可以随时修改它封装的值。下面部分将解释一个对象类型为什么会有可变和不可变两种变体，描述对象可变性的特点和副作用，并向您推荐当对象的可变性成为问题时如何进行最佳的处理。

本部分包含如下内容：

为何要有可变和不可变两种对象变体？

对象缺省都是可变的。大多数对象都允许您通过“setter”存储方法改变其封装的数据。例如，您可以改变一个 NSWindow 对象的尺寸、位置、标题、缓冲行为、和其它特征。一个设计良好的模型对象，比如一个表示客户记录的对象，*必须提供* setter 方法，以便修改它的实例数据。

Foundation 框架通过引入一些具有可变变体和不可变变体的类明确了一些细微的差别。可变的子类通常是其不可变变体的子类，且在类名上嵌入了“Mutable”字样。这些类包括：

- NSMutableArray
- NSMutableDictionary
- NSMutableSet
- NSMutableIndexSet
- NSMutableCharacterSet
- NSMutableData
- NSMutableString
- NSMutableAttributedString
- NSMutableURLRequest

请注意：除了 NSMutableParagraphStyle在Application Kit框架中定义之外，所有显式命名的可变类都在Foundation框架中定义。但是所有的Cocoa 框架都可能有自己的可变的和不可变的类变体。

虽然这些类都有一个典型的名称，但是和相应的不可变体相比，它们更接近可以改变的正常体。为什么这么复杂？为可变对象定义一个不可变的变体有什么目的呢？

考虑在一个所有对象都可以被改变的场景中，您的应用程序调用某个方法，并返回一个代表一个字符串的对象引用。您在用户界面中用这个字符串来标识一片特定的数据。现在，程序中的另一个子系统也得到了同一个字符串的引用，并决定对它进行修改。这样，您的标签就会被不知不觉地修改了。在某些情况下事情会变得更加可怕，比如您得到一个数组得引用，并用在表视图控件中。用户选择一个与数组中某个对象相对应的行，而该对象已经被程序中其它地方的代码删除了—问题就因此出现了。对象的不可变性可以保证对象在使用时不会被意外地改变。

对离散值的集合进行封装、或者包含的值被存储在缓冲区（其自身也是集合，或者是字符集合，或者是字节集合）中的对象，都是实现不可变版本的良好候选者。但并不是所有的“值”对象都一定受益于可变版本，只包含一个简单值的对象，比如 NSNumber或 NSDate的实例，并不是实现可变版本的良好候选者。当它们表示的值在这些类中发生改变时，以新的实例替换旧的实例显得更为合理。

性能也是使用不可变对象表示某些数据（比如字符串和字典）的一个原因。这种数据的可变对象自身会产生更多的开销，因为它们必须动态管理一个可变的辅助存储—在必要时分配或解除分配内存块—所以比相应的不可变版本效率低。

虽然不可变的特性在理论上可以保证对象的值是稳定的，但在实践中这个保证并不总是确定的。类的方法可能会将不可变变体的返回类型下面的可变对象取出，并在之后决定对其进行改变，这可能侵犯接收者根据之前的值做出的假定和选择。在经历各种变形的过程中，对象自身的可变性也可能发生变化。举例来说，对一个属性列表进行序列化（通过 NSPropertyListSerialization类）并不保留对象的可变性信息，而是只保留它们的一般类型—比如字典、数组等等。因此，当您反序列化这个属性列表时，结果对象可能和原始对象不同，比如最初的 NSMutableDictionary对象现在可能变成一个 NSDictionary对象。

用可变对象编程

在碰到对象可变性的问题时，最好采纳一些防御性的编程实践。下面是几个一般性的原则：

- 当您需要在对象创建之后频繁或不断地对其内容进行修改时，请使用对象的可变变体。
- 有些时候，用一个不可变对象取代另一个可能更好。比如，大多数保留字符串的实例变量都应该被赋值为一个不可变的 NSString 对象，而这些对象则用“setter”方法来进行替换。
- 依靠返回类型来进行可变性提示。
- 如果您不能确定一个对象是可变的，则将它当成不可变的处理。

本部分将探讨这些规则中存在的灰色区域，讨论用可变对象编程的过程中通常需要做的选择，同时还将概要介绍 Foundation 框架中创建可变对象及在对象的可变变体和不可变变体之间进行转换的方法。

创建和转换可变对象

您可以通过标准的 alloc-init 嵌套消息来创建一个可变对象，比如：

```
NSMutableDictionary *mutDict = [[NSMutableDictionary alloc] init];
```

然而，很多可变对象都提供初始化器和工厂方法，用于指定对象的初始或可能的容量，比如 NSMutableArray 类的 arrayWithCapacity: 类方法。

```
NSMutableArray *mutArray = [NSMutableArray arrayWithCapacity:[timeZones count]];
```

这种线索使得可变对象数据的存储更为有效（类工厂方法约定俗成地将返回对象放到自动释放池中，因此如果您希望在代码中使用该返回对象，请务必对其进行保留）。

您也可以通过为现有的一般类型对象制作可变拷贝的方式来创建一个可变对象。要实现这个目的，只需要调用 Foundation 框架中可变类的不可变超类实现的 mutableCopy 方法就可以了：

```
NSMutableSet *mutSet = [aSet mutableCopy];
```

反过来，您也可以通过向一个可变对象发送 copy 消息来得到该对象的一个不可变拷贝。

很多同时带有可变和不可变变体的 Foundation 类都含有在不同变体之间进行转换的方法，包括：

- *type*With*Type*:—比如 arrayWithArray:
- set*Type*:—比如 setString:（只有不可变类）
- initWith*Type*:copyItems:—比如 initWithDictionary:copyItems:

存储和返回可变实例变量

Cocoa 开发中的一个常见问题是是否应该让一个实例可变或不可变。对于一个值可以改变的实例变量，比如字典或字符串，什么时候将它变为可变变体比较合适？还有，什么时候将对象转换为不可变变体，并在它的值发生变化时将它替换为另一个对象比较好？

一般地说，当一个对象的内容需要发生彻底变化时，使用不可变对象更好一些。字符串(NSString)和数据对象(NSData)通常属于这个范畴。如果对象是逐步发生变化，则采用可变变体比较合理。诸如数组和

字典这样的集合类对象都属于这个范畴。然而，变化的频率和集合的尺寸也是应该考虑的因素。举例来说，如果是一个很少发生变化的小数组，则采用不可变变体比较好。

在确定由一个实例变量表示的集合的可变性时，还有一些其它因素需要考虑：

- 如果您有一个可变的集合，经常需要变化，而且经常需要提供给客户代码（也就是说，直接在“getter”存取方法中直接返回该对象）。这样，客户代码引用的对象就有被改变的风险。如果这种风险是可能的，该实例变量就应该是不可变的。
- 如果实例变量的值经常改变，但是很少通过 getter 方法将它返回给客户代码，则您可以使用该实例变量的可变变体，但是在存取方法中返回一个放入自动释放池的不可变拷贝（参见列表 2-13 的实例）。

列表 2-13 返回一个可变实例变量的不可变拷贝

```
@interface MyClass : NSObject {

    // ...

    NSMutableSet *widgets;

}

// ...

@end



@implementation MyClass

- (NSSet *)widgets {

    return (NSSet *)[[widgets copy] autorelease];

}
```

处理返回给客户的可变集合的一个复杂方法是维护一个标志，以记录对象当前是可变还是不可变。在对象需要发生变化时，将对象转换为可变，并对其进行修改；在将集合提供给其它代码时，则在返回对象之前将它转换为不可变（必要的话）。

接收可变对象

方法的调用者对返回对象的可变性感兴趣有如下两个原因：

- 它希望知道是否可以改变对象的值
- 它希望知道它所引用的对象是否会意外地发生变化

使用返回类型，而不是通过内省获取类型

接收者必须依赖返回值的正式类型来确定是否可以修改接收到的对象。举例来说，如果它接收到一个类型为不可变的数组对象，那就不应该试图改变它。基于对象所属的类来确定对象是否可以修改并不是一个可接受的编程实践，例如：

```
if ( [anArray isKindOfClass:[NSMutableArray class]] ) {

    // add, remove objects from anArray

}
```

由于实现上的原因，这里的 isKindOfClass: 返回的信息可能不是精确的。除此之外，还有其它一些原因，使您不应该根据所属的类来判定对象是否可以改变，而是应该将产生对象的方法签名上的可变性声明作为唯一的依据。如果您不能确定一个对象是否可变，就假定它是不可变的。

下面的几个例子可能有助于说明这个指导原则的重要性：

- 您从一个文件读取一个属性列表。当 Foundation 框架处理这个列表时，它注意到属性列表中的多个子集是一样的，并因此创建一组对象，使之在那些子集中共享。之后，您看到创建好的属性列表对象，并决定修改其中的一个子集。这样，在不知情的情况下，您就突然改变了多个地方的数据树了。
- 您向 NSView 请求它的子视图对象（subviews 方法），它返回一个声明为 NSArray 的对象，该对象的类型在内部可能是 NSMutableArray。然后您将该数组传递给其它代码，而那些代码通过内省的方法确定数组是可以改变的，并对其进行修改。这样，NSView 的内部数据结构也就被改变了。

因此，请不要根据内省方法的信息来判定对象的可变性。对象是否可以改变应该根据您在 API 边界上进行的定义（也就是根据返回类型）来判定。如果您在将对象传递给客户代码时需要明确标识对象是否可变，则可以将这个信息作为一个标志量，和对象一起传递。

为接受到的对象制作快照

如果您希望确保从某个方法接收到的、被认为是不可变的对象不会在不知情的情况下发生变化，可以在局部对它进行拷贝，为它制作一个“快照”，然后不时地将对象的存储版本和最新版本进行比较。如果对象被改变了，您可以调整依赖于先前版本对象的程序。列表 2-14 显示了这种技术的一个可能的实现方法。

列表 2-14 制作潜在可变对象的快照

```
static NSArray *snapshot = nil;

- (void)myFunction {

    NSArray *thingArray = [otherObj things];

    if (snapshot) {

        if ( ![thingArray isEqualToArray:snapshot] ) {

            [self updateStateWith:thingArray];

        }

    }

    snapshot = [thingArray copy];

}
```

为对象制作快照以备后用的问题在于开销太昂贵，您需要为同一个对象制作多个拷贝。一个更有效的选择是使用键-值观察协议，该协议的概要信息请参见 "键-值观察"部分。

集合中的可变对象

将可变对象存储在集合对象中可能会导致问题。某些集合可能会因为其包含的对象发生变化而被破坏或变成无效，因为那些改变可能影响到对象放置到集合的方式。在第一种情况中，对象的属性是诸如 NSDictionary 或 NSSet 对象这样的哈希集合的键，如果被改变了，且被改变的属性影响到对象的 hash 或者 isEqual:方法的结果，则会导致集合被破坏（如果集合中对象的 hash 方法不依赖于它们的内部状态，则集合被破坏的可能性就小一些）；第二种情况是，如果顺序集合（比如经过排序的集合）种对象的属性发生改变，可能会影响该对象和数组中其它对象比较的方式，并因此使集合的顺序变成无效。

类簇

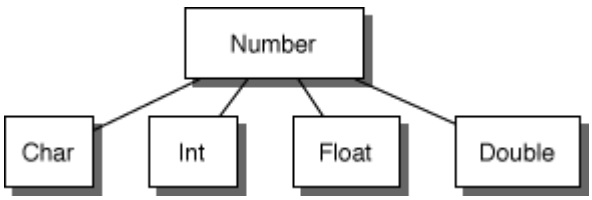
类簇是Foundation框架中广泛使用的设计模式。类簇将一些私有的、具体的子类组合在一个公共的、抽象的超类下面，以这种方法来组织类可以简化一个面向对象框架的公开架构，而又不减少功能的丰富性。类簇基于抽象工厂设计模式，其详细讨论请参见 "Cocoa设计模式"部分。

本部分包括如下内容：

简单的概念，复杂的接口

为了说明类簇的架构及其好处，我们来考虑构造一个定义不同类型数值（char、int、float、double）的类层次的问题。由于不同类型的数值有很多公共的特性（比如它们可以从一个类型转换为另一个类型，可以表示为字符串），因此可以表示为一个单一的类。然而，它们的存储要求并不一样，因此用同一个类来表示的效率不高。这就需要如下的架构：

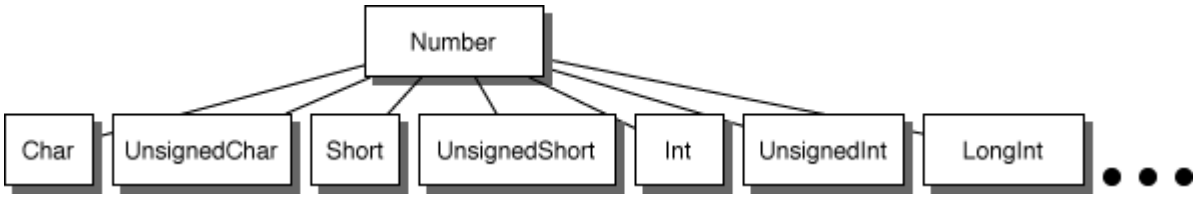
图 2-9 数字类的简单架构



Number 是抽象超类，负责声明子类的公共操作，但是不声明用于存储数字的实例变量。它的子类负责声明实例变量，并共享 Number 类定义的编程接口。

到目前为止，这种设计相当简单。但是如果考虑广泛使用的 C 语言基本类型，该框图应如下所示：

图 2-10 更完整的数值类层次

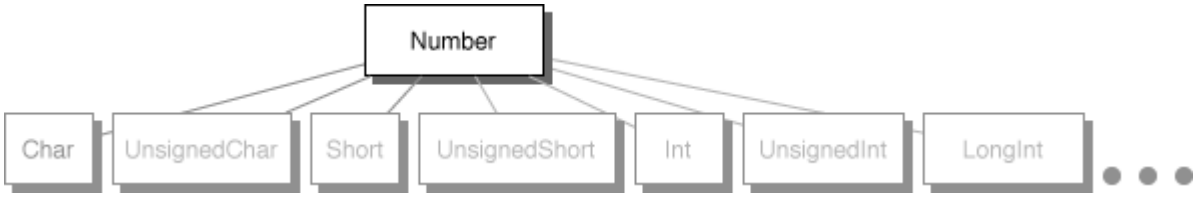


简单的概念—创建一个保存数值的类—可以很容易地产生出一打还多的类。类簇架构代表一个反映简单概念的设计。

简单的概念，简单的接口

将类簇的设计模式应用到具体问题中，就会产生下面的架构（私有类用灰色表示）：

图 2-11 应用到数字类的类簇架构



这个架构的用户只看到一个公共类，即 Number 类。那么，它怎么可能分配出正确子类的实例呢？答案是通过抽象超类来处理实例化。

创建实例

类簇中的抽象超类必须声明创建其私有子类的方法。根据您调用的创建方法分配正确类型的对象是超类的责任，您不必也不能选择实例的类。

在 Foundation 框架中，您通常通过调用+ classname... 或 alloc... 和 init... 方法来创建对象。以 Foundation 框架的 NSNumber 类为例，您可以发送如下的消息来创建数字对象：

```
NSNumber *aChar = [NSNumber numberWithChar:'a'];

NSNumber *anInt = [NSNumber numberWithInt:1];

NSNumber *aFloat = [NSNumber numberWithFloat:1.0];

NSNumber *aDouble = [NSNumber numberWithDouble:1.0];
```

(这种风格的实例化创建的对象会被自动地被取消分配—更多信息请参见 "类工厂方法"部分。很多类也提供标准的alloc... 和init... 方法来创建对象，您需要自行释放这种方法创建的对象）。返回的每个对象—aChar、anInt、aFloat、和 aDouble—可能属于不同的私有子类（实际上就是如此）。虽然每个对象所属的类是被隐藏的，但是其接口是公开的，即由抽象超类 NSNumber 声明的接口。虽然不是完全正确，但是将 aChar、anInt、aFloat、和 aDouble 对象考虑为 NSNumber 类的一

个实例是很便利的，因为它们都是由 NSNumber 类的方法创建的，且通过 NSNumber 声明的实例方法来访问。

带有多个公共超类的类簇

在上面的例子中，一个抽象公共类负责声明多个私有子类的接口。这就是最纯粹意义上的类簇。通过两个（或者可能更多）抽象公共类来声明类簇的接口也是可能的（而且常常是可取的）。这在 Foundation 框架中也是很明显的，包括如下这些类簇：

类簇	公共超类
NSData	NSData
	NSMutableData
NSArray	NSArray
	NSMutableArray
NSDictionary	NSDictionary
	NSMutableDictionary
NSString	NSString
	NSMutableString

这种类型的其它类簇也是存在的，但是这些已经清楚地说明了两个抽象结点如何结合在一起，共同声明一个类簇的编程接口。在上面的每个类簇中，一个公共结点声明所有类簇对象都能响应的方法，另一个则声明内容可以被改变的对象才能响应的方法。

这种类簇接口的重构有助于使面向对象框架的编程接口更加清楚。作为例子，假定一个 Book 对象声明了如下方法：

```
- (NSString *)title;
```

这个对象可以返回它自己的实例变量或创建一个新的字符串对象并返回—这没什么关系。这个声明很清楚地说明返回的字符串不能被修改。任何修改返回对象的尝试都会引起编译器的警告。

在类簇中创建子类

类簇架构是简洁性和可扩展性之间的一个折衷。用几个公共类代表多个私有类可以使框架衷的类易学易用，但是在某种程度上增加创建类簇子类的难度。然而如果很少需要创建子类，则类簇架构则有明确的好处。Foundation 框架只在这种情况下使用类簇。

如果您发现某个类簇没有提供您的程序需要的功能，则可能适合引入一个子类。举例来说，假定您希望创建一个数组对象，其存储是基于文件的，而不是如 NSArray 类簇那样是基于内存的。由于您需要改变类的存储机制，所以需要创建子类。

另一方面，在某些情况下可能定义一个拥有类簇对象的类就够了（而且更加容易）。假定您的程序需要在某些数据被修改的时候得到通知，则为 Foundation 框架中定义的数据对象创建一个简单的包装类可能是最好的方法。该类的对象可以干预修改数据的消息，将它截获并进行必要的动作，然后再转发给嵌入的数据对象。

总的来说，如果您需要管理对象的存储，就创建一个真的子类。否则，就创建一个合成对象，即将标准的 Foundation 框架对象嵌入到您自己设计的对象中。下面的部分将更为详细地讨论这两种方法。

真正的子类

在类簇中创建的新类必须：

- 是类簇抽象超类的子类
- 声明自己的存储
- 重载超类的基本方法（下文进行描述）

由于类簇的抽象超类是类簇层次中唯一公共可见的结点，所以上述的第一点是显而易见的。这意味着新的子类继承类簇的接口，但没有实例变量，因为抽象超类没有声明。这样就引出了上述的第二点：子类必须声明自己需要的所有实例变量。最后，子类必须重载它继承的、直接访问对象实例变量的所有方法，这样的方法称为*基元方法（primitive methods）*。

类的基元方法是其接口的基础。以 NSArray 类为例，它为管理对象数组的对象声明了接口。在概念上，数组负责存储一些数据项，每个数据项都可以通过索引来访问。NSArray 通过两个基元方法来描述这个抽象的概念， 即 count 和 objectAtIndex:方法，在这些方法的基础上可以实现其它的方法—即*衍生方法*，比如：

衍生方法	可能的实现
lastObject	通过向数组对象发送如下消息寻找数组的最后一个对象：[self objectAtIndex: ([self count] ?1)]。
containsObject:	通过向数组对象反复发送 objectAtIndex:消息来寻找对象，每次递增索引，直到检查完数组中所有的对象。

在接口上区分基元方法和衍生方法使子类的创建更加容易。您的子类必须重载通过继承得到的基元方法，而重载基元方法之后就可以保证继承而来的衍生方法可以正确地工作。

基元方法和衍生方法的区别适用于完全初始化了的对象的接口。我们还需要考虑在子类中应该如何处理 init...方法的问题。

一般来说，类簇的抽象超类会声明一些init...和+ className方法。如 "创建实例"部分描述的那样，抽象类根据您调用的init...或+ className方法来确定应该实例化哪个具体的子类。您可以认为抽象类声明这些方法是为了方便子类。由于抽象类没有实例变量，所以不需要初始化方法。
您的子类应该声明自己的 init...（如果它需要初始化自己的实例变量）和可能的+ className 方法，而不应该依赖于继承而来的那些方法。为了维护它连接的初始化链，它应该在自己的指定初始化方法中调用超类的指定初始化方法（有关指定初始化方法的讨论请参见 *Objective-C 编程语言*一书中的“指定初始化方法”部分）。在类簇中，抽象超类的指定初始化方法总是 init 方法。

真正的子类：一个例子

讨论一个例子有助于阐明上面的讨论。假定您希望创建一个 NSArray 的子类，类名为 MonthArray，返回与给定索引位置对应的月份名称。然而，MonthArray 对象并不将月份名称数组实际存储为一个实例变量，而是在负责返回给定索引位置的月份名称的方法（objectAtIndex:）中返回常量字符串。因此，无论应用程序中存在多少个 MonthArray 对象，都只分配十二个字符串。

MonthArray 类的声明如下：

```
#import <foundation/foundation.h>

@interface MonthArray : NSArray

{

}



+ monthArray;

- (unsigned)count;

- (id)objectAtIndex:(unsigned)index;



@end
```

请注意，MonthArray 类并没有声明 init... 方法，因为它没有需要初始化的实例变量。count 和 objectAtIndex:方法只是简单地封装继承得到的基元方法，如上所述。

MonthArray 类的实现大致如下：

```
#import "MonthArray.h"



@implementation MonthArray



static MonthArray *sharedMonthArray = nil;

static NSString *months[] = { @"January", @"February", @"March",

    @"April", @"May", @"June", @"July", @"August", @"September",

    @"October", @"November", @"December" };



+ monthArray
```

```
{
    if (!sharedMonthArray) {
        sharedMonthArray = [[MonthArray alloc] init];
    }
    return sharedMonthArray;
}


- (unsigned)count
{
 return 12;
}


- objectAtIndex:(unsigned)index
{
    if (index >= [self count])
        [NSException raise:NSRangeException format:@"***%s: index
            (%d) beyond bounds (%d)", sel_getName(_cmd), index,
            [self count] - 1];
    else
        return months[index];
}


@end
```

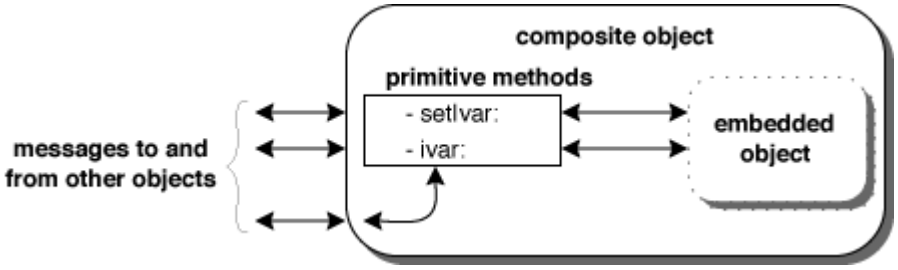
由于 MonthArray 重载了继承的基元方法，所以相应的衍生方法没有被重载也可以正确工作。NSArray 的 lastObject、containsObject:、sortedArrayUsingSelector:、objectEnumerator 及其它方法在 MonthArray 对象中可以正常使用。

合成对象

您可以将一个私有的类簇对象嵌入到自己设计的对象中，从而创建一个合成对象。这种合成对象在基本功能上可以依赖于类簇对象，而只需要截取希望特殊处理的消息。使用这种方法可以减少必须编写的代码量，而且可以利用 Foundation 框架提供的测试良好的代码。

您可以以如下方式查看一个合成对象：

图 2-12 嵌入一个类簇对象



合成对象必须将自己声明为类簇抽象结点的一个子类。作为子类，它必须重载超类的基元方法。它也可以重载衍生方法，但并不是必须的，因为衍生方法是基于基元方法的。

以 NSArray 的 count 方法作为例子，干预对象对重载方法的实现可以很简单，如下所示：

```
- (unsigned)count

{

    return [embeddedObject count];

}
```

当然，您的对象也可以将自己需要的代码放到重载方法的实现中。

合成对象：一个例子

为了说明合成对象的用法，假定您有一个可变的数组对象，并希望在允许改变数组内容之前，检查即将发生的变化是否符合正当性条件。本文接下来的例子描述一个称为 ValidatingArray 的类，它包含一个标准的可变数组对象。ValidatingArray 重载 NSArray 和 NSMutableArray 超类声明的所有基本方法，声明了 array、validatingArray、和 init 方法，可以用于创建和初始化实例：

```
#import <foundation/foundation.h>



@interface ValidatingArray : NSMutableArray

{

    NSMutableArray *embeddedArray;

}



+ validatingArray;

- init;
```

```
- (unsigned)count;

- objectAtIndex:(unsigned)index;

- (void)addObject:object;

- (void)replaceObjectAtIndex:(unsigned)index withObject:object;

- (void)removeLastObject;

- (void)insertObject:object atIndex:(unsigned)index;

- (void)removeObjectAtIndex:(unsigned)index;


@end
```

实现文件中显示在 ValidatingArray 类的 init 方法中如何创建嵌入对象，并将它分配给 *embeddedArray* 变量。对数组进行访问但不修改其内容的消息被中继给嵌入对象；需要对内容进行改变的消息则被审查（这里用伪代码表示），而且只有通过理想的正当性测试之后才进行中继。

```
#import "ValidatingArray.h"


@implementation ValidatingArray


- init
{
    self = [super init];
    if (self) {
        embeddedArray = [[NSMutableArray allocWithZone:[self zone]] init];
    }
    return self;
}


+ validatingArray
{
    return [[[self alloc] init] autorelease];
```

```
}

- (unsigned)count
{
    return [embeddedArray count];
}

- objectAtIndex:(unsigned)index
{
    return [embeddedArray objectAtIndex:index];
}

- (void)addObject:object
{
    if (/* modification is valid */) {
        [embeddedArray addObject:object];
    }
}

- (void)replaceObjectAtIndex:(unsigned)index withObject:object;
{
    if (/* modification is valid */) {
        [embeddedArray replaceObjectAtIndex:index withObject:object];
    }
}

- (void)removeLastObject;
{
```

```
    if (/* modification is valid */) {
        [embeddedArray removeLastObject];
    }
}
- (void)insertObject:object atIndex:(unsigned)index;
{
    if (/* modification is valid */) {
        [embeddedArray insertObject:object atIndex:index];
    }
}
- (void)removeObjectAtIndex:(unsigned)index;
{
    if (/* modification is valid */) {
        [embeddedArray removeObjectAtIndex:index];
    }
}
```

创建一个单件实例

Foundation 和 Application Kit 框架中的一些类只允许创建单件对象，即这些类在当前进程中的唯一实例。举例来说，NSFileManager 和 NSWorkspace 类在使用时都是基于进程进行单件对象的实例化。当您向这些类请求实例的时候，它们会向您传递单一实例的一个引用，如果该实例还不存在，则首先进行实例的分配和初始化。

单件对象充当控制中心的角色，负责指引或协调类的各种服务。如果您的类在概念上只有一个实例（比如 NSWorkspace），就应该产生一个单件实例，而不是多个实例；如果将来某一天可能有多个实例，您可以使用单件实例机制，而不是工厂方法或函数。

创建单件实例时，您需要在代码中执行下面的工作：

- 声明一个单件对象的静态实例，并初始化为 nil。
- 在该类的类工厂方法（名称类似于“sharedInstance”或“sharedManager”）中生成该类的一个实例，但仅当静态实例为 nil 的时候。
- 重载 allocWithZone:方法，确保当用户试图直接（而不是通过类工厂方法）分配或初始化类的实例时，不会分配出另一个对象。

- 实现基本协议方法：copyWithZone:、release、retain、retainCount、和 autorelease ，以保证单件的状态。

列表 2-15 显示如何实现一个单件：

列表 2-15 实现一个单件

```
static MyGizmoClass *sharedGizmoManager = nil;



+ (MyGizmoClass*)sharedManager

{

    @synchronized(self) {

        if (sharedGizmoManager == nil) {

            [[self alloc] init]; // assignment not done here

        }

    }

    return sharedGizmoManager;

}



+ (id)allocWithZone:(NSZone *)zone

{

    @synchronized(self) {

        if (sharedGizmoManager == nil) {

            sharedGizmoManager = [super allocWithZone:zone];

            return sharedGizmoManager;  // assignment and return on first allocation

        }

    }

    return nil; //on subsequent allocation attempts return nil

}



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

```
{
    return self;
}

- (id)retain
{
    return self;
}

- (unsigned)retainCount
{
    return UINT_MAX;  //denotes an object that cannot be released
}

- (void)release
{
    //do nothing
}

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

当您需要一个单件实例（由类工厂方法来创建和控制），又需要通过分配和初始化创建其它实例时，就可能出现问题。在这种情况下，您不必重载allocWithZone: 及其后的其它方法，如 列表 2-15 所示。

为Cocoa程序添加行为

当您用 Objective-C 开发程序的时候，很多工作不必自己做。您要关注苹果和其它人已经完成的工作，关注他们已经开发完成并封装在 Objective-C 框架的类。这些框架为您提供一组独立的类，而这些类构成了您的程序的一部分—通常是很重要的一部分。

本章将描述使用 Cocoa 框架编写一个 Objective-C 程序是什么样的过程，并为您提供制作一个框架类的子类需要知道的信息。

本部分包括如下内容：

开始

使用 Objective-C 框架中的类及其方法和使用一个 C 函数库不同。使用 C 函数库时，您可以根据程序的具体需求灵活选择使用什么函数及何时使用。但是另一方面，框架可以将一种设计加入到您的程序，或者至少加入到您的程序需要解决的某个问题空间中。在面向过程的程序中，您在必要时调用库函数来完成程序需要完成的工作；使用框架时，您也必须调用框架的方法来完成程序中的很多工作，但您还需要对框架进行定制，即实现框架在合适的时候需要调用的方法，使其适合您的需要。那些方法是一种“钩子”，将您的代码加入到框架实现的程序结构中，为其增加您的程序需要的行为。在某种意义上，通常理解的程序和库的作用被反过来了，不是将库的代码结合到您的程序中，而是将您的程序代码结合到框架中。

通过考察一个 Cocoa 应用程序开始执行的时候发生的事，可以对定制代码和框架之间的关系有一些认识。

在main函数里发生了什么

和 C 程序一样，Objective-C 程序从 main 函数开始执行。在一个复杂的 Objective-C 程序中，main 函数的工作相当简单，由两个步骤组成：

- 建立一个核心的对象组。
- 将程序的控制权转交给那些对象。

在程序运行时，核心组的对象可能会创建其它对象。有些时候，程序也可能装载类，装载归档了的实例，连接远程的对象，以及寻找需要的资源。但在开始时，需要的是一个足以处理程序初始任务的结构—即一个够用的对象网络。main 函数将这个初始结构建立起来，并使它们为即将到来的工作做好准备。

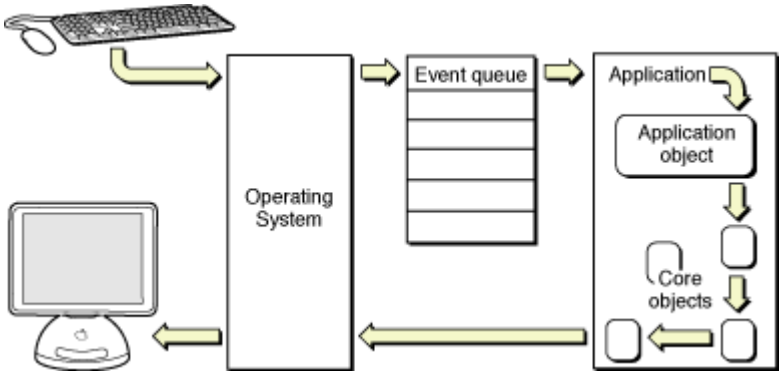
典型情况下，核心对象中应该有一个是负责监视程序或控制其输入的对象。在核心结构准备好了之后，main 函数会启动监视对象的工作。如果程序是一个命令行工具或后台运行的服务器，这个步骤承担的工作可能是简单地传递命令行参数或打开一个远程连接。但是最常见的 Cocoa 程序类型是应用程序，这种情况下， main 函数需要参与更多的工作。

对于应用程序，main 函数建立的核心对象组必须包含一些描画用户界面的对象。当用户启动应用程序时，用户界面，或者至少是用户界面的一部分（比如应用程序菜单）必须出现在屏幕上。一旦初始的用户界面被显示在屏幕上，应用程序随即就由外部的事件驱动了，最重要的是那些由用户发起的外部事件：比如点击一个按键，选择一个菜单项，拖拽一个图标，在输入域中键入等等。每个这样的事件都和很多用户动作发生的环境信息一起，报告给应用程序—比如哪个键被按下，是否有鼠标被按下或释放，光标的位置在哪里，以及哪个窗口受到影响等。

应用程序获取一个事件，对其进行查看并做出响应（经常是通过描画部分的用户界面），然后等待下一个事件。如果事件由用户或某些其它的事件源（比如定时器）发起，应用程序就会逐个地进行获取。从启动到终止运行，应用程序所做的几乎所有事情，都由用户动作驱动，形式上表现为事件。

事件的获取和响应机制就是主事件循环（称为“主循环”是因为应用程序可以为某个短暂的时间段建立从属的事件循环）。事件循环本质上是一个带有一个或多个输入源的运行循环。核心组中的一个对象负责运行主事件循环—获取事件，并将事件派发给最适合的一个或多个对象，然后获取下一个事件。在 Cocoa 应用程序中，这个负责协调的对象就是全局的应用程序对象，即 NSApplication 类的一个实例。图 3-1 显示了就是这个主事件循环。

图 3-1 主事件循环



在几乎所有的 Cocoa 应用程序中，main 函数都是极为简单的，因为它只有一个函数调用（参见列表 3-1）。NSApplicationMain 函数的作用是创建一个应用程序对象，建立一个自动回收池，从主 nib 文件中装载初始的用户界面，以及将应用程序运行起来，使其开始处理主循环接收到的事件。

列表 3-1 Cocoa 应用程序中的 main 函数

```
#import <AppKit/AppKit.h>


int main(int argc, const char *argv[]) {

    return NSApplicationMain(argc, argv);

}
```

"核心应用程序架构"部分更为详细地描述主事件循环、全局的NSApplication实例、以及其它核心应用程序对象。

使用Cocoa框架

库函数很少对使用它们的程序进行限制，您可以在任何需要的时候进行调用。另一方面，面向对象的库或框架中的方法和类的定义紧密相关，如果您没有创建或保留可以访问那些定义的对象，就不能对其进行调用。而且，在大多数程序中，对象必须至少和一个对象相连接，才能在程序网络中发挥作用。一个类只负责定义一个程序组件，为了访问类提供的服务，您必须将它连接到应用程序结构中。

也就是说，框架类生成一些行为类似于一组库函数的实例。您简单地创建一个实例，对其进行初始化，然后或者向它发送消息使其完成某个任务，或者将它插入到应用程序中某个设计好的插槽中。举例来说，您可以用 NSFileManager 类来执行各种文件系统操作，比如移动、拷贝、和删除文件。如果您需要显示一个警告对话框，则可以创建一个 NSAlert 类的实例，并向它发送正确的消息。

然而在一般情况下，象 Cocoa 这样的环境并不仅仅是一些提供服务的、彼此独立的类集合。它们是由一些面向对象的框架和面向特定问题空间并提出完整解决方案的类集合组成的。框架不是提供一些在需要时可以调用、彼此不相关的服务（象函数库那样），而是制订您的代码必须适应的整个程序结构—或者说程序模型。由于这个程序模型是具有一般意义的，您可以对它进行具体化，以满足特定的程序需要。您要做的不是设计一个调用库函数的程序，而是将您自己的代码插入到框架提供的设计中。

要使用框架，就必须接受框架定义和使用的程序模型，而且需要定制一些类，使面向具体应用场合的程序可以和该模型相适应。这些类相互依赖，以一个组而不是单独类的形式出现。乍一看，在程序代码中采纳框架的模型需要做的工作比较有限，但事实却相反。框架为您提供了很多改变和扩展其一般行为的途径，它只是简单地要求您接受所有 Cocoa 程序的基本行为方式，因为它们都基于同样的程序模型。

本部分包含如下内容：

框架类的类型

Cocoa 框架中的类以四种方式发布它们的服务：

- **复活方式**。一些类定义了复活（off-the-shelf）对象以方便用户使用。您可以简单地创建这些类的实例，并根据需要对其进行初始化。NSControl 的子类，比如 NSTextField、NSButton、和 NSTableView（和与之相关联的 NSCell 类一起）都属于这个范畴。虽然您也可以通过编程的方式来创建和初始化复活对象，但是通常还是用 Interface Builder 来进行这些工作。
- **幕后方式**。在程序运行时，Cocoa 会在“幕后”为其创建一些框架对象。您不需要显式分配和初始化这些对象，框架会自动替您完成这些工作。这些类通常是私有的，但是要实现期望的行为。
- **一般类的方式**。框架中有一些一般类，或者说抽象类。框架可能提供一些一般类的具体子类，不经修改就可以使用。然而您可以—在某些情况下是*必须*—定义您自己的子类，并重载特定的方法。NSView、NSDocument、和 NSFormatter 类就是这种类的例子。
- **委托者和通告者方式**。很多框架对象将自己的动作通知给其它对象，甚至将特定的责任委托给其它对象。传递这种信息的机制就是委托和通告机制。委托者对象需要公布一个被称为非正式协议的接口，客户对象则必须首先注册为委托，然后实现该接口中的一个或多个方法。发布通告的对象要公布自己广播的通告列表，而所有的客户对象都可以自由监听其中的一个或多个公告。NSApplication、NSText、和 NSWindow 就是一些委托者类，而很多框架类都可以广播通告。

某些类提供不止一种类型的服务。举例来说，您可以将一个准备好的 NSWindow 对象从 Interface Builder 的选盘中拖出，并在少量的初始化之后进行使用。这样，NSWindow 类就为您提供了复活实例。但是一个 NSWindow 对象也需要向它的委托发送消息，以及向其它对象发布通告。如果需要，比如您希望有一个圆角的窗口，您甚至可以生成 NSWindow 的子类。

采用最后两种服务方式—即一般类方式和委托者/通告者方式—的Cocoa类为将程序的具体代码集成到框架提供的结构中提供最大的可能性。"Cocoa类的继承"部分就如何创建框架类、特别是一般类的子类进行一般性的讨论，有关委托、通告、以及程序网络中的其它对象间通讯机制的信息请参见 “和对象进行通讯”部分。

Cocoa API的约定

在您开始使用 Cocoa 框架中的类、方法、和其它 API 的时候，需要知道一些约定，它们都是为了确保使用效率和一致性而制订的。

- 返回对象的方法通常通过返回 nil 来表示“创建失败”或者“没有对象可以返回”。这种方法并不返回状态码。

  返回 nil 的约定通常用于表示运行时错误或其它非例外的条件。Cocoa 框架通过抛出例外（由最顶级的例外处理代码处理）来处理诸如“数组索引越界” 或 “不能识别方法选择器”这样的错误，如果方法签名有要求的话，同时返回 nil。

- 某些可能返回 nil 的方法可以通过最后一个参数以引用的方式返回错误信息。

  上述的最后一个参数是一个 NSError 对象的指针；对于执行失败的方法（也就是说方法返回 nil），您可以通过考察返回的错误对象来确定错误的原因，或者将错误显示在对话框上。

  以 NSDocument 类的一个方法为例：

```
- (id)initWithType:(NSString *)typeName error:(NSError **)outError;
```

- 类似地，执行某些系统操作的方法（比如文件读写）通常返回一个 Boolean 值，以指示执行成功还是失败。

  这些方法也会将一个 NSError 对象指针作为最后一个引用参数。以 NSData 类的一个方法为例：

```
- (BOOL)writeToFile:(NSString *)path options:(unsigned)writeOptionsMask
error:(NSError **)errorPtr;
```

- Cocoa 用空的容器对象来表示缺省值或没有值—nil 通常不是正当的对象参数。

  很多对象封装了对象的值或集合，比如 NSString、NSDate、NSArray、和NSDictionary类的实例。将这些对象作为参数的方法可以接收表示没有值或缺省值的“空”对象（比如@""）。比如，下面的消息通过指定一个空的字符串，将一个窗口的关联文件名设置为“没有值”：

```
[aWindow setRepresentedFilename:@""];
```

  请注意：Objective-C 的@“*characters*”构造器用于创建一个包含文字*字符*的 NSString 对象，因此@“ “会创建一个不包含字符的字符串对象—或者说是一个空字符串。更多信息请参见 *Cocoa 的字符串编程指南*。

- Cocoa 框架要求在字典键、通告和例外名称、以及一些用字符串作为参数的方法上使用全局字符串常数，而不是一个字符串文字。

  在可以进行选择的时候，您应该总是选择字符串常量，而不是字符串文字。使用字符串常量时，编译器可以帮助您进行拼写检查，这样可以避免运行时错误。

- Cocoa 框架在类型使用上是一致的，各组 API 之间可以进行较好的匹配。

  举例来说，Cocoa框架用float来表示坐标的值，用CGFloat类型表示图形和坐标值，用NSPoint（由两个CGFloat值组成）来表示坐标系统中的一个位置，用 NSString对象来表示字符串的值，

用NSRange来表示范围（起始点和偏移量），分别用NSInteger和NSUInteger来表示有符号和无符号的整数值。当您设计自己的API时，应该尽量保持类似的一致性。

相当一部分 Cocoa API 约定关注于类、方法、函数、常量、和其它符号的命名。在您开始设计自己的编程接口时，需要知道这些约定。一部分较为重要的命名约定如下所示：

- 在类名和与类相关联的符号，比如函数和 typedef 定义的类型上，使用前缀。

  使用前缀可以避免命名冲突，帮助区分不同的函数域。前缀的命名约定是使用两个或三个唯一的大写字母，比如 ACCircle 中的“AC”。

- 在 API 的命名上，清楚比简洁更重要。

  举例来说，removeObjectAtIndex:的功能很容易理解，但是 remove:就有点模糊。

- 避免模菱两可的命名。

  比如displayName 就模菱两可，因为我们不清楚它的功能是显示一个名称还是返回一个显示名称。

- 在代表动作的函数或方法名上使用动词。

- 如果一个方法返回一个属性或经过计算的值，则直接使用属性名作为方法名。

  这些方法就是所谓的“getter”存取方法。举例来说，如果属性是背景颜色，则 getter 方法应该命名为 backgroundColor。返回 Boolean 值的 getter 方法的命名有细微的区别，采用“is”或“has” 前缀—比如 hasColor。

- 对于负责设置属性值的方法（也就是“setter” 存取方法），则其名称以“set”开头，后接属性名称。

  属性名称的第一个字母是大写字母—比如 setBackgroundColor:。

  请注意：有关如何实现setter和getter方法的详细讨论，请参见 *模型对象实现指南*文档中的 “存取方法” 部分。

- 不要在 API 名称上使用缩写，如果不是众所周知的缩写的话（比如 HTML 或 TIFF）。

如果需要 Objective-C 编程接口命名风格的完整资料，请参见 *Cocoa 编码指南*。

还有一个一般性的、具有决定作用的API约定，是关于对象所有权的。简单地说，这个约定就是，如果一个客户代码进行对象的创建（通过分配、初始化）、拷贝、和保持（通过发送 retain消息），就拥有该对象的所有权。对象的所有者在不需要使用该对象时，要向它发送 release或者 autorelease消息进行清除。这个主题的更多信息请参见 “Cocoa对象的生命周期”部分和*Cocoa的内存管理编程指南*。

Cocoa类的继承

象 Application Kit 这样的框架都定义某种程序模型。由于这个模型具有一般性，很多不同类型的应用程序都可以共享。也由于这个模型具有一般性，框架中的某些类是抽象类或有意没有完成也并不奇怪。一个类通常会完成很多低级别的、公用的代码，而将工作的相当一部分留下来，或者以安全而又一般的“缺省”方式来完成。

应用程序通常需要创建子类来填充超类留下的缺口，提供框架类缺少的东西。子类是向框架添加具体应用程序行为的基本途径。定制子类的实例在框架定义的对象网络中代替其超类的位置，并通过继承从超类得到与框架中其它对象协同工作的能力。举例来说，如果您创建了一个 NSCell 的子类，则这个新类的实例

可以出现在 NSMatrix 对象中，就象 NSButtonCell、NSTextFieldCell、以及其它框架定义的 cell 对象一样。

在制作子类时，一个主要的任务就是实现一组由超类（或者超类采纳的协议）声明的具体方法。重新实现超类的方法被称为对该方法进行*重载*。

本部分包含如下主要内容：

何时进行方法的重载

框架类中定义的大多数方法都是完全实现的，您可以对其进行调用，以得到它们提供的服务。您很少需要重载这种方法，而且也不应该试图这样做。依赖于这些类的框架只是做它们应该做的事—既不多，也不少。在某些场合下，您可以对这些方法进行重载，但是没有真正的原因需要这么做，框架版本的方法已经足够了。但是，正如您可能实现您自己的字符串比较函数、而不是使用 strcmp 函数那样，如果您愿意，可以选择重载框架的方法。

然而，有些框架方法的设计目的就是为了被重载的，您可以通过这种方式向框架加入程序的具体行为。这些方法在框架中的实现对应用程序通常价值很小，或者没有价值，但会在其它框架方法发出的消息中被调用。应用程序必须实现自己的版本，为这些方法加入新的内涵。

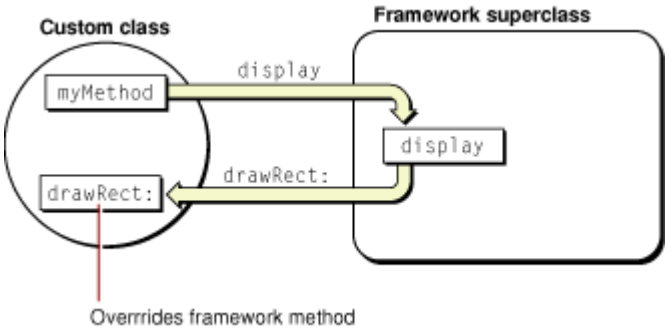
调用还是重载？

一般来说，您自己并不调用，至少不直接调用在子类中重载的框架方法。您只要简单地重新实现这些方法，然后将它留给框架就好了。实际上，越是那些实现应用程序具体行为的版本，您自己的代码对它调用的可能性就越小。这有一个很好的原因。在一般意义上，框架类负责声明一些公共方法，您作为开发者可以有两种使用方式：

- 调用这些方法，使类提供的服务为您所用
- 对这些方法进行重载，将您的代码引入到框架定义的程序模型中

有些时候，一个方法会同时符合上述两种情况，既可以通过被调用提供有价值的服务，也可以被策略性地重载。但是一般来说，一个方法如果可以被调用，就已经由框架完全定义好了，不需要在您的代码中进行精化；如果该方法需要在子类中重新实现，则说明框架为该方法分派了特殊的工作，而且会在恰当的时候对其进行调用。图 3-2 显示了这两种一般类型的框架方法。

图 3-2 调用一个框架方法，该方法又通过消息调用一个重载了的方法



使用 Cocoa 框架进行面向对象编程的大部分工作是实现一些方法，而您的程序只是间接地、通过框架安排的消息使用这些方法。

重载方法的类型

您可以选择在子类中定义几个不同类型的方法：

- 某些框架方法是完全实现的，且其设计的目的是被别的框架方法调用。换句话说，即使您重新实现这些方法，通常也不在其它代码的其它地方调用。它们提供特定的服务—数据或行为，这些服务是程序执行过程中某些地方的代码要求的。这些方法存在于公共接口中只有一个原因—就是让您在需要的时候可以对其进行重载，这使您有机会用自己的算法来替代框架使用的算法，或者对框架的算法进行修改和扩展。

  这种类型的方法的一个例子是 NSMenuView 类定义的 trackWithEvent:方法。NSMenuView 类实现这个方法是为了满足看得见的需求—处理菜单跟踪和菜单项的选择，但是如果您希望实现不同的行为，则可以对其进行重载。

- 另一类方法负责做一些与具体对象有关的决定，比如是否打开某个属性，或者是否让特定的策略起作用。框架为这种方法实现一个缺省版本，从而提供一种工作方式，如果您需要有所改变，就必须实现自己的版本。在大多数情况下，实现就是简单地返回 YES 或者 NO，或者对某个值进行计算，而不是使用缺省值。

  NSResponder 类的 acceptsFirstResponder 方法就是一个典型的例子。系统向视图对象发送消息中包含 acceptsFirstResponder 消息，用于询问它们是否响应按键或鼠标点击事件。缺省情况下，NSView 对象在这个方法中返回 NO—大多数视图对象并不接收按键输入。但是某些视图对象却是可以的，因此它们必须重载 acceptsFirstResponder 方法，使之返回 YES。

- 某些方法必须被重载，但只是增加一些处理，而不是完全取代框架的实现。这种方法的子类版本对超类版本的行为进行增强。您的程序在实现这种方法时，很重要的一点是要吸收被重载方法，即向 super（超类）对象发送消息，调用框架为该方法定义的版本。

  这类方法通常是继承链中的每个类都希望有所贡献的。举例来说，可以自行归档的对象必须遵循 NSCoding 协议，并且实现 initWithCoder:和 encodeWithCoder:方法。但是，一个类在对自己特有的实例变量进行编解码的时候，必须调用相应方法的超类版本。

  有些时候，方法的子类版本希望"重用"超类的行为，然后在最后的结果中加入一些小变化。比如 NSView 类的 drawRect:方法，执行某些复杂描画的视图子类可能希望在描画结果中加上一个边界，这样就要首先调用 super 版本的方法。

- 某些框架方法什么事情都不做，或者只是返回一些试验性的缺省值（比如 self），避免运行时或编译时的错误。这些方法的设计目的就是为了被重载。即便是最基本的行为，框架也无法为它们定义，因为它们执行的任务全部和具体程序相关。对于这种方法，没有必要通过向 super 发送消息来调用框架的实现。

  子类重载的大部分方法都是这种类型。比如 NSDocument 类的 dataOfType:error:和 readFromData:ofType:error:（还有其它）方法，在您创建基于文档的应用程序时必须被重载。

对一个方法进行重载并不一定很难。通过认真地重写方法中的一两行代码，您常常就能显著改变超类的行为。在实现自己版本的方法时，也不是完全从头开始，您可以借助 Cocoa 框架提供的类、方法、和类型。

什么时候需要使用子类

和了解类的哪些方法需要重载—并真正地实施重载—一样重要的是，识别哪些类需要被继承。有些时候，这些决定可能是很明显的，而在另一些时候，做这样的决定则相当不简单。下面的一些设计上的考虑可以指导您做这样的选择。

首先，连接框架。您应该熟悉每个框架类的目的和能力。您希望做的事情可能已经在某个类中实现了，或者如果您发现希望完成的任务在某个类中已经*差不多*完成了，那就很幸运了，那个类很可能是您需要的定制类的超类。子类化是重用现有的类、并根据需要将它具体化的过程。有些时候，一个子类需要做的所有工作，就是对一个方法进行重载，并使它的行为和超类版本轻微不同。其它子类可能在超类的基础上增加一两个属性（以实例变量的形式），然后实现一些访问和操作这些属性的方法，从而将它们集成到超类的行为中。

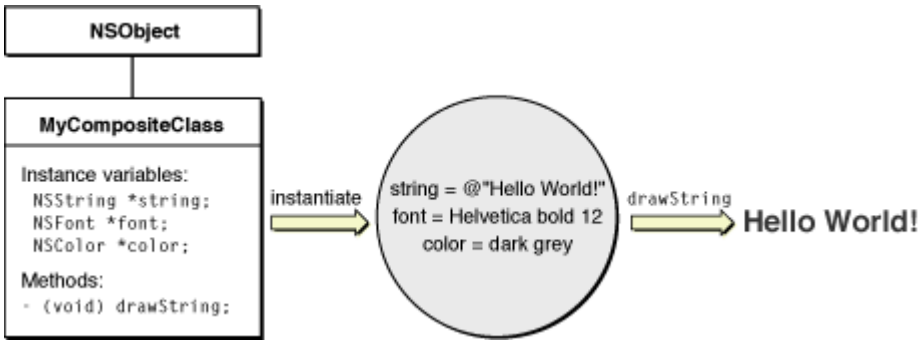
在决定子类在类层次中的位置时，还有其它一些有益的考虑。您希望开发的应用程序、或者应用程序的一部分的本质是什么？有些 Cocoa 架构对子类有些要求。举例来说，如果您开发的是一个多文档的应用程序，则 Cocoa 基于文档的架构就要求您生成 NSDocument 类的子类，可能还有其它类。如果要让您的应用程序可以通过脚本进行控制（也就是说，可以响应 AppleScript 命令），可能必须生成诸如 NSScriptCommand 这样的脚本类的子类。

另一个因素是子类的实例在应用程序中发挥的作用。模型-视图-控制器模式是Cocoa的主要设计模式，它将对象的角色做如下分配：出现在用户界面上的对象属于视图对象，模型对象负责保存应用程序数据（和对该数据进行操作的算法），控制器对象则负责协调视图对象和模型对象（详细信息请参见 "模型-视图-控制器设计模式"部分）。了解一个对象的作用可以收窄其超类的选择范围。如果您的类实例是实现定制描画和事件处理的视图对象，可能应该选择NSView作为超类；如果您的应用程序需要一个控制器类，则可以使用某个复活类（比如NSObjectController类），或者如果您希望有不同的行为，也可以从 NSController或NSObject派生出子类；如果您的类是一个典型的模型类—比如代表一个电子表格数据中的行的类—则可能应该从NSObject派生出子类，或者使用Core Data框架。
然而生成子类有时并不是解决问题的最好办法，可能有更好的办法可以选择。如果您只是希望为某个类增加一些便利方法，就可以通过创建范畴来实现，而不需要生成子类；或者，您也可以借助基于Cocoa开发"工具箱"资源的很多其它设计模式之一来实现，比如委托、通告、和目标-动作模式（在 "和对象进行通讯"部分中描述）。在确定使用某个候选超类之前，先扫描一下它的头文件（或参考文档），看看是否有什么委托方法、通告、或者其它机制可以实现您需要的功能，而又不需要生成子类。
类似地，您也可以考察一下框架协议的头文件或文档。通过采纳协议，您既可以完成目标，又可以规避复杂子类的困难工作。举例来说，假定您希望管理菜单项的激活状态，则可以在定制的控制器子类中采纳 NSMenuValidation 协议，而不必从 NSMenuItem 或 NSMenu 派生子类来得到这个行为。

和某些框架方法不用于被重载一样，一些框架类（比如NSFileManager、NSFontPanel、和NSLayoutManager）也不用于生成子类。如果您确实希望有这样的子类，则应该谨慎处理。某些框架类的实现是相当复杂的，和其它类的实现、甚至是操作系统的不同部分紧密结合在一起。通常情况下，我们很难正确复制框架方法的行为，或者预期这种方法可能有的依赖性和效果。您对一些方法实现的修改，可能带来深远的、不可预见的、以及不希望的结果。

在某些情况下，您可以通过对象的合成来克服这种困难。对象的合成是一种将多个对象装配到一个“宿主”对象中的通用技术，宿主对象负责管理这些对象，并获得复杂而又高度定制的行为（参见图 3-3）。您不必直接从一个复杂的框架超类继承子类，而是创建一个定制类，然后将超类的实例作为类的一个实例变量。定制类自身可能相当简单，可能直接从NSObject根类继承就可以了。虽然从继承的角度来看是简单了，但是该类负责对嵌入的实例进行操作、扩展、和增强。对于客户对象来说，该类在某些方面就象是复杂超类的子类，虽然它可能并不共享超类的接口。Foundation框架中的NSAttributedString类就是对象合成的一个实例。NSAttributedString以实例变量的形式保有一个NSString对象，并通过string方法将它暴露给客户代码。NSString是一个具有复杂行为的类，包括字符串编码、字符串检索、以及路径处理。NSAttributedString则在这些行为的基础上加入了新的能力，可以将字体、颜色、对齐、以及段落风格这样的信息附加到某个范围的字符中，而且在不生成NSString子类的前提下实现这个增强。

图 3-3 对象合成



有些时候，看起来很明显的超类候选者其实并不是最好的选择。您可能知道，Cocoa 在大多数情况下都用 NSView 对象来进行描画。但是如果您正在设计的描画程序或 CAD 程序可能具有成千上万个图形元素，则应该考虑使用您自行定制的图形元素类，而不是从 NSView 继承。从对象尺寸的角度看，一个 NSView 对象携带很多实例数据，而您定制的类的图形元素实例可以是“轻量级”的，但又包含 NSView 对象中用于描画的所有信息。

基本子类设计

所有设计良好的子类都有一些共同的特征，不管它继承自什么类、作用是什么。设计不好的子类容易出现错误、不好用、难于扩展、以及性能不好，而设计良好的子类则恰恰相反。本文将为设计高效、强壮、既易于使用又可以重用的子类提供一些建议。

进一步阅读：虽然这篇文档也描述一些制作子类的技巧，但主要还是关注基本设计。文中没有描述如何使用Xcode和Interface Builder开发工具来自动化类定义的部分工作。如果要学习使用这些工具，请阅读 *Xcode 快速游指南（Xcode Quick Tour Guide）*。这个部分描述的设计信息特别适用于模型对象。*模型对象实现指南*更为详细地讨论模型对象的正确设计和实现。

本部分包含如下主要内容：

子类定义的形式

Objective-C 类是由一个接口部分和一个实现部分组成的。根据约定，这两个类定义部分分别放在两个文件中。接口文件（和 ANSI C 头文件一样）的扩展名是.h；实现文件的扩展名为.m。文件名中除了扩展名之外的部分通常是类的名称。这样，对于名为 Controller 的类，相应的文件为：

```
Controller.h

Controller.m
```

接口文件中包含一个构成类公共接口的方法声明的列表，还有实例变量、常量、全局字符串、以及其它数据类型的声明。@interface 导向符用于引出基本的接口声明，@end 导向符则用于终止接口的声明。@interface 导向符特别重要，因为它标识了类的名称以及直接的继承类，它的形式如下：

@interface*ClassName* ： *Superclass*

列表 3-2 给出了先前假定的 Controller 类在加入任何声明之前的接口文件。

列表 3-2 The 接口文件的基本结构

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



@interface Controller : NSObject {


}
```

```
@end
```

为了完成类接口，您必须在接口结构的恰当地方加入必要的声明，如图 3-4 所示。

图 3-4 在接口文件的什么地方加入声明

有关这些声明类型的更多信息，请参见 "实例变量"和 "函数、常量、和其它C类型"部分。

开始时，您需要为接口中出现的类型导入对应的 Cocoa 框架和头文件。#import 预处理命令在合并指定的头文件方面和#include 类似。但是，#import 的效率更高，在编译时，它只包含之前没有直接或间接包含的头文件。命令之后的尖括号(<...>)中的内容标识头文件所在的框架，斜线之后是头文件自身。这样#import 语句的语法形式如下：

#import <*Framework*/*File*.h>

框架必须位于框架的标准系统位置上。在 列表 3-2 的实例中，类接口文件导入了Foundation框架中的Foundation.h文件。和这个例子一样，Cocoa约定框架的同名头文件包含一个#import命令的列表，列表中又包含框架的所有公共接口（以及其它公共头文件）。顺便提一下，如果您定义的类是应用程序的一部分，则只需要导入Cocoa雨伞框架中的Cocoa.h文件就可以了。

如果您导入的头文件是工程的一部分，需要使用引号而不是尖括号来作为分隔符，例如：

```
#import "MyDataTypes.h"
```

类实现文件的结构更加简单，如列表 3-3 所示。重要的一点是，在类实现文件开始部分要导入类接口文件。

列表 3-3 实现文件的基本结构

```
#import "Controller.h"




@implementation Controller




@end
```

您必须在@implementation 和@end 导向符之间书写所有的方法实现。根据约定，与类相关的函数实现也应该放在这两个导向符之间，但实际上可以放在文件的任何地方。私有类型（函数、结构等）的声明通常放在#import 命令和@implementation 导向符之间。

重载超类方法

顾名思义，定制类就是以一些面向具体程序的方式对超类行为进行修改的类。对超类的方法进行重载是做这种修改的常用方式。在设计子类时，一个基本的步骤是识别希望重载的方法，以及考虑如何重新实现这些方法。

虽然 "什么时候对方法进行重载"部分提供了一些一般的指导原则，但是您还是需要对类进行调查，详细阅读类的头文件和文档，才能识别需要重载的超类方法，还需要找出超类的指定初始化方法，因为您必须在子类的指定初始化方法中调用相应的超类版本，才能成功进行初始化（详细信息请参见 "对象的创建" 部分）。
一旦您识别了要重载的超类方法，就可以开始—纯粹从实践的角度看—将这些方法声明拷贝到您的接口文件中，再将同样的声明拷贝到.m 文件中作为方法实现的框架，并用花括号代替结尾的分号。

请记住，如 "什么时候对方法进行重载"部分讨论的那样，Cocoa框架（不包含您自己的代码）通常会调用您重载的框架方法。在某些情况下，您需要让框架知道应该调用的是您重载的版本，而不是原始的方法。Cocoa为此提供了不同的解决方法，比如在Interface Builder的Info (查看器)窗口中，您可以用自己的类来代替（兼容的）框架类。如果您创建了一个定制的NSCell类，则可以通过NSControl的setCell:方法将它关联到特定的控件上。

实例变量

创建定制类的原因除了修改超类的行为之外，还有增加新的属性。这里提到的“属性”既指子类实例的属性，也指该实例对其它对象的引用（也就是它的关系）。假定有一个 Circle 类，如果您要生成一个子类来为形状加入颜色，则子类在某些程度上必须携带颜色属性。为此，您可能要在类接口中加入一个 color 实例变量（可能是类型为 NSColor 的对象）。您的定制类的实例需要封装这个新的属性，将它作为一个持久的特征数据加以保有。

Objective-C 的实例变量是指对象、结构、以及作为类定义一部分的其它数据类型声明。如果是对象，则类型的声明可以是动态（使用 id）的，也可以是静态的。下面的例子显示了两种声明风格：

```
id delegate;

NSColor *color;
```

一般说来，当对象所属的类不确定或不重要时，就使用动态类型来声明实例变量。以实例变量保有的对象需要进行创建、拷贝、或显式的保持—如果父对象没有对其进行保持的话（使用委托对象也是一样的）。如果对象实例是通过解档生成的，则应该在initWithCoder:方法进行解码和赋值的时候保持对象实例变量（更多对象归档和解档的信息请参见 "入口点和出口点"部分）。
实例变量的命名规则是使用小写字符串，不包含标点符号和特殊字符。如果变量名称包含多个词，就直接把这些词连起来，且第二个及之后的词的首字大写。比如：

```
NSString *title;

NSColor *backgroundColor;

NSRange currentSelectedRange;
```

实例变量声明之前的 IBOutlet 标识表示一个带有连接的插座变量，该连接信息存储在 nib 文件中。IBOutlet 标识也使 Interface Builder 和 Xcode 可以协调自己的活动。从 Interface Builder 的 nib 文件解档的插座变量是自动保持的。

实例变量不仅仅可以保有提供给对象客户的对象属性。有些时候，实例变量也可以保有一些私有数据，用于支持对象执行某些任务，后备存储器或缓存就是这样的例子（如果数据不是基于实例，而是在类的多个实例之间共享，则需要全局变量，而不是实例变量）。

当您为子类增加实例变量时，下面这些原则值得注意：

- 只加入一些绝对必要的实例变量。您加入的实例变量越多，实例的尺寸越大。而且，您的类实例创建得越多，对开销的关注就越大。在可能的情况下，尽量从现有的实例变量中计算出一个关键值，而不是增加新的实例变量。
- 出于同样的经济上的原因，尽量有效使用类的实例数据。举例来说，如果您希望将一些标志指定为实例变量，则可以用位域来代替一系列布尔声明（然而需要知道，位域的归档复杂一些）。您可以通过 NSDictionary 对象来把一些互相关联的属性整合成键-值对。如果采取这种方法，需要确保键有良好的文档说明。
- 赋给实例变量正确的作用域。永远不要将变量设置为@public，因为这违反了封装的原则。如果您定义的类（比如您的应用程序类）的子类可能需要使用且需要进行高效访问，可以使用 @protected。否则，@private 就是合理的选择，这可以很大程度上隐藏实现的细节。对于框架输出的、被应用程序或其它框架使用的类，隐藏实现细节特别重要；这样可以在修改类实现时，不必重新编译所有客户代码。
- 确保类基本属性对应的实例变量有存取方法（存取方法用于获取和设置实例变量的值）。这个主题的更多信息请参见 "存取方法"部分。
- 如果您希望将子类公开化—也就是说，您希望其它人基于您的子类派生新的类—可以在实例变量列表最后补丁一些保留子段，通常使用id作为类型。如果在将来的某个时候，您需要在类中加入另一个实例变量，保留字段有助于保证二进制的兼容性。更多为公共类做准备的信息，请参见 "类什么时候是公共的"部分。

入口点和出口点

Cocoa 框架会在对象生命周期的不同时候向对象发送信息。几乎所有对象（包括类，它们本身实际上也是个对象）在运行时生命周期开始时和对象被析构之前，都会收到特定的消息。这些消息调用的方法（如果实现的话）是一些“勾子”，使对象可以在特定时机上执行一些任务。这些方法（按照调用顺序）如下：

1. initialize 这个类方法使类可以在自身或其实例接收任何其它消息之前，自行进行初始化。超类在子类之前接收到这个消息。使用老的归档机制的类可以通过实现初始化方法来设置类的版本。其它可能的初始化工作包括注册某些服务，以及初始化那些所有实例都使用的全局状态。然而，有些时候更好的做法是将这些工作推迟到某个实例方法来实现（也就是在首次使用的时候），而不是在初始化时实现。
2. init(或其它初始化方法）如果超类的指定初始化方法所做的工作不足以初始化您的类，就必须实现init或其它基本初始化方法，以便对类实例状态进行初始化。有关初始化方法，包括指定初始化方法的信息，请参见 "对象的创建"部分。
3. initWithCoder:如果您希望类对象可以被归档—比如当您的类是模型类的时候—则应该采纳（必要的话）NSCoding 协议，并实现其中的两个方法：initWithCoder:和 encodeWithCoder:，并在这两个方法中分别将对象的实例变量编码和解码为适合归档的形式。为此，您可以使用 NSCoder 类的解码和编码方法，以及 NSKeyedArchiver 和 NSKeyedUnarchiver 类提供的键-归档的支持。当对象通过解档的方式生成、而不是显式创建时，

initWithCoder:方法（而不是其它的初始化方法）就会被调用。在这个方法中，您可以在解码实例变量值之后将它们赋给相应的变量，并在必要时对其进行保持或拷贝。有关这个主题的更多信息请参见 *Cocoa 的归档和序列化编程指南*。

4. awakeFromNib 应用程序在装载 nib 文件时，会向从档案中装载的对象发送一个 awakeFromNib 消息，但只是发送给可以响应该消息的对象，且在档案中的所有对象都装载和初始化完成后发送。当对象接收到 awakeFromNib 消息时，Cocoa 会保证对象中所有的插座实例变量都准备好了。典型情况下，拥有 nib 文件的对象（File's Owner）会实现 awakeFromNib 方法，以执行插座变量和目标-动作连接准备好之后才能执行的初始化工作。

5. encodeWithCoder:如果您的类实例需要支持归档，则需要实现这个方法。这个方法在对象析构之前被调用。请参见上面的 initWithCoder:方法的描述。

6. dealloc您可以实现这个方法，以释放实例变量，解除类实例占用的其它所有内存。在这个方法返回后不久，实例就会被销毁。有关dealloc方法的进一步信息，请参见 "对象的创建"部分。

一个应用程序的全局应用程序对象（由 NSApp 表示）在应用程序生命周期的开始和结束时也会向相应的对象发送消息—如果该对象是 NSApp 的委托并实现了恰当的方法的话。在应用程序启动后，NSApp 会向它的委托对象发送 applicationWillFinishLaunching:和 applicationDidFinishLaunching:消息（前者在任何被双击的文档打开之前发送，后者则在文档被打开之后发送）。委托对象可以在这两个方法中任何一个实现应用程序在运行时存在之前需要的一次性全局逻辑。NSApp 在终止运行之前会向其委托对象发送 applicationWillTerminate:消息，您可以在这个方法中保存文档和应用程序状态，以便优雅地终止应用程序。

Cocoa框架为您提供了很多其它的事件挂钩，从窗口的关闭到应用程序的激活，以及从活动状态变为不活动状态。这些挂钩通常实现为委托消息，要求您的对象成为框架对象的委托，并实现必要的方法。有些时候，挂钩也可以是通告（更多关于委托的信息，请参见 "和对象进行通讯"部分）。

初始化还是解码?

如果您希望类的对象可以支持归档和解档，则该类必须遵循 NSCoding 协议；必须实现对对象进行编码（encodeWithCoder:）和解码（initWithCoder:）的方法。和初始化方法或其它方法不同的是，调用 initWithCoder:方法是为了初始化解档得到的对象。

由于类的初始化方法和 initWithCoder:可能会做很多同样的工作，一个合理的做法是将一些共同的工作实现为一个辅助方法，然后在初始化方法和 initWithCoder:中调用。举例来说，如果一个对象在配置例程中需要指定拖拽类型和拖拽源，则可能需要按列表 3-4 所示的方式来实现：

列表 3-4 辅助的初始化方法

```
(id)initWithFrame:(NSRect)frame {

    self = [super initWithFrame:frame];

    if (self) {

        [self setTitleColor:[NSColor lightGrayColor]];

        [self registerForDragging];

    }
```

```
    return self;
}
- (id)initWithCoder:(NSCoder *)aCoder {
    self = [super initWithCoder:aCoder];
    titleColor = [[aCoder decodeObject] copy];
    [self registerForDragging];
    return self;
}

- (void)registerForDragging {
    [theView registerForDraggedTypes:
        [NSArray arrayWithObjects:DragDropSimplePboardType, NSStringPboardType,
       NSFilenamesPboardType, nil]];
    [theView setDraggingSourceOperationMask:NSDragOperationEvery forLocal:YES];
}
```

类的初始化方法和 initWithCoder: 在角色上的并行性也有例外。在为框架类创建定制子类、而框架类的实例出现在 Interface Builder 的选盘上时，在工程中定义子类的一般过程是将对象从 Interface Builder 选盘中拖到界面上，然后通过 Interface Builder 的 Info 窗口中的 Custom Class（定制类）面板把对象和定制子类关联起来。但是在这种情况下，子类的 initWithCoder: 方法在解档时并不被调用，取而代之的是发送一个 init 消息。定制对象的所有特殊的配置任务都应该在 awakeFromNib 方法中执行，该方法在 nib 文件中所有对象解档完成后被调用。

存取方法

存取方法（或者简单地称为“存取器”）负责获取或设置实例变量的值。它们是设计良好的类接口的必要组成部分，相当于通向对象属性的一道门槛，负责提供对象属性的访问通道，并强制对对象实例数据的封装。

由于命名规范的原因—也由于这种命名规范使类得以遵循键-值编码（参见 "键-值机制"部分）的约定—存取方法必须以指定的形式命名。对于返回实例变量值的方法（有时也称为*getter*），就简单地使用实例变量名；对于为实例变量设置值的方法（也称为*setter*），其名称以“set”开头，后面紧接着实例变量的名称（首字母大写）。例如，如果您有一个名为“color”的实例变量，则其getter和setter存取方法的声明应为：

```
- (NSColor *)color;

- (void)setColor:(NSColor *)aColor;
```

如果实例变量的类型为C的数值类型，比如int或者float，则存取方法的实现就趋于非常简单。假定有一个实例变量名为currentRate，类型为float，列表 3-5 显示如何为其实现存取方法。

列表 3-5 为非对象实例变量实现存取方法

```
- (float)currentRate {

    return currentRate;

}



- (void)setCurrentRate:(float)newRate {

    currentRate = newRate;

}
```

如果实例变量保存的是对象，情况就有些细微的区别。由于是实例变量，所以这些对象必须可以持久，因此在赋值时必须进行创建、拷贝、或保持。setter 方法在改变实例变量的值时，不仅要保证它的可持久性，还要正确处理原来的值；getter 方法则是将实例变量的值传给发出请求的对象。两类存取方法的操作在内存管理方面都有源自 Cocoa 对象所有权策略的两个隐含假定：

- 方法（比如 getter 存取方法）返回的对象在调用该方法的对象的作用域内是正当的。换句话说，需要保证对象在该作用域内（如果没有其它文档说明的话）不被释放或修改。
- 调用对象从某个方法（比如存取方法）接收到对象时，不应该对其进行释放，除非它先前进行显式的保持（或拷贝）。

记住这两个假定之后，让我们看一个名为 title、类型为 NSString 的实例变量的 getter 和 setter 存取方法的两种可能的实现。列表 3-6 显示了第一种实现方式。

列表 3-6 为对象实例变量实现存取方法—好技术

```
- (NSString *)title {

    return title;

 }

 - (void)setTitle:(NSString *)newTitle {

    if (title != newTitle) {

        [title autorelease];

        title = [newTitle copy];

    }

 }
```

请注意，getter 方法只是简单地返回实例变量的引用。相比之下，setter 方法更忙一些，在确认传入值和当前值不一样之后，它就自动释放当前值，然后将新值拷贝到实例变量上（向对象发送 autorelease 消息比发送 release 消息更加“线程安全”一些）。然而，这种方法仍然有潜在的危险。如果一个客户正在

使用 getter 方法返回的对象，与此同时 setter 方法自动释放了老的 NSString 对象，且该对象在之后很快被释放和销毁，那会有什么结果呢？客户对象对实例变量的引用将不再是正当的了。

列表 3-7 显示了存取方法的另一种实现，通过在 getter 方法中保持和自动释放实例变量的值，使这个问题得到解决。

列表 3-7 为对象实例变量实现存取方法—更好技术

```
- (NSString *)title {
    return [[title retain] autorelease];
}
- (void)setTitle:(NSString *)newTitle {
    if (title != newTitle) {
        [title release];
        title = [newTitle copy];
    }
}
```

在上面两个 setter 方法的实例（列表 3-6 和列表 3-7）中，新的 NSString 实例变量是通过拷贝，而不是通过保持得到。为什么不使用保持的方式呢？一般的规则是：当赋给实例变量的对象是一个值对象时—也就是说，该对象代表的是一个诸如字符串、日期、数字、或组合记录这样的属性—您就应该进行拷贝。您感兴趣的是保留该属性的值，不希望它的值被潜在的程序修改。换句话说，您希望自己拥有该对象的拷贝。但是，如果要存储和访问的是一个实体对象，比如 NSView 或者 NSWindow 对象，则应该加以保持。实体对象聚合度更高，关系更为复杂，拷贝起来开销更大一些。确定一个对象是值对象还是实体对象的方法之一是弄清楚您感兴趣的是对象的值，还是对象的本身。如果您感兴趣的是对象的值，则该对象可能是一个值对象，应该进行拷贝（当然，我们假定对象遵循 NSCopying 协议）。

确定在setter方法中应该保持实例对象还是进行拷贝的另一种方法是确定实例变量是一个属性还是一种关系。这对表示应用程序数据的模型对象来说尤其正确。属性和值对象基本是相同的：它表示的是自己封装的对象的某个定义特征，比如对象的颜色（NSColor对象）或标题（NSString对象）；另一方面，关系仅仅是指对象本身和一个或多个其它对象之间的一种关系（或者引用）。一般地说，在setter方法中，您对属性进行拷贝，而对关系进行保持。然而关系有一个基数，可能对一对一的，也可能是一对多的。一对多的关系通常由集合对象来表示，比如NSArray或NSSet的实例，可能需要在setter方法中做更多的工作，而不是简单地对实例变量进行保持。详细信息请参见 *模型对象实现指南*。更多有关对象特性的信息，无论是属性还是关系，请参见 "键-值机制"部分。

如果一个类的 setter 方法以列表 3-6 或列表 3-7 所示的方式来实现，则在类的 dealloc 方法中对实例变量进行解除分配时，只需要调用合适的 setter 方法，并传入 nil 就可以了。

键-值机制

名称中带有“键-值”的几个机制都是 Cocoa 的基本部分：键-值绑定、键-值编码、和键-值观察。它们都是 Cocoa 技术的必要成分，比如对象间自动进行值的通讯和同步的绑定技术就是基于这些技术的；它们也至

少为实现应用程序的脚本控制（就是使应用程序可以响应 AppleScript 命令）提供部分的基础设施。键-值编码和键-值观察在定制子类的设计时特别重要。

“键-值” 这个术语指的是将属性名称作为键，并通过这个键取得相应的值。这个术语是对象建模模式中使用的词汇，而对象建模模式反过来又是从描述关系数据库用到的实体-关系模型衍生出来的。在对象建模的模式中，对象—特别是模型-视图-控制器模式下与数据有关的模型对象—拥有一些特性（properties），它们在形式上通常（但并不总是）表现为实例变量。特性可以是一个属性（attribute），比如名称或者颜色，也可以是指向一个或多个其它对象的引用。这些引用称为关系，关系可以是一对一的，也可以是一对多的。一个程序中的对象网络通过它们之间的关系组成一个对象图。在对象模型中，您可以使用键路径—就是由若干个键组成的字符串，键与键之间用圆点分隔的字符串—来遍历对象图中的关系，以及访问对象的特性。

请注意：对象建模的完整描述，请参见 "对象建模"部分。

键-值绑定、键-值编码、和键-值观察是支持这种遍历的机制。

- 键-值绑定（Key-value binding，简称 KVB）机制负责建立对象间的绑定关系，以及移除和公布这种绑定关系。它用了几个非正式的协议。属性的绑定必须指定一个对象和一个指向该属性的键路径。
- 键-值编码（Key-value coding，简称 KVC）机制通过实现名为 NSKeyValueCoding 的非正式协议，使开发者可以通过键直接设置和获取对象属性，而不需要调用对象的存取方法（Cocoa 为该协议提供了缺省的实现）。键通常和被访问对象中的实例变量或存取方法的名称相对应。
- 键-值观察（Key-value observing，简称 KVO）机制通过实现名为 NSKeyValueObserving 的非正式协议，其作用是使对象可以将自己注册为其它对象的观察者。当被观察对象的属性之一发生改变时，会直接通知相应的观察者。Cocoa 为遵循 KVO 的对象的每个属性都实现了自动观察者通知机制。

为使子类的每个属性都遵循键-值编码的要求，需要进行如下的工作：

- 对于名为 *key* 的属性或者目标基数为一(to-one)的关系，实现名为 *key* (getter)和 set*Key*: (setter) 的存取方法。举例来说，如果您有一个名为 salary 的属性，则需要实现名为 salary 和 setSalary:的存取方法。
- 对于一个目标基数大于一（to-many）的关系，如果其属性对应的实例变量是一个集合（比如一个 NSArray 对象）或者返回集合的存取方法，则将 getter 方法命名为属性名（比如 employees）。如果该属性是可以改变的，而 getter 方法并不返回一个可变的集合（比如 NSMutableArray），您必须实现 insertObject:in*Key*AtIndex:和 removeObjectFrom*Key*AtIndex:方法。如果实例变量不是集合类型，且 getter 方法不返回集合，则必须实现其它的 NSKeyValueCoding 方法。

在满足自动观察者通知的基础上，简单地保证您的对象遵循 KVC 就可以使之遵循 KVO。然而，如果您选择实现手工的键-值观察，就需要额外的工作。

进一步阅读：更多的技术信息请阅读 *键-值编码编程指南*和 *键-值观察编程指南*。有关Cocoa绑定技术（使用KVC、KVO、和KVB）的概括性描述，请参见 "和对象进行通讯"中的 "绑定"部分。

对象的基础设施

如果一个子类是设计良好的，它的实例就应该以 Cocoa 对象期望的方式工作。使用这种对象的代码可以将它和类的其它实例相比较，探索它的内容（比如在调试器中），以及用该对象执行类似的基本操作。

定制子类应该实现绝大多数（如果不是全部的话）根类和基本协议的方法：

- isEqual:和hash 实现这些NSObject方法可以在对象比较时执行某些对象的具体逻辑。举例来说，如果您的类实例根据序列号的不同进行区分，则可以将序列号作为相等比较的基础。更多信息请参见 "内省"部分。
- description 通过实现这个 NSObject 方法来返回简要描述对象属性和内容的字符串。这个信息在 gdb 调试器中可以通过 print object 命令返回，在格式化字符串中则可以由对象的%@指示符来使用。举例来说，假定您有一个 Employee 类，带有姓名（name）、雇佣日期（date of hire）、部门（department）、和位置 ID（位置 ID）等属性，则类的描述方法可能如下：

```
- (NSString *)description {

    return [NSString stringWithFormat:@"Employee:Name = %@,

        Hire Date = %@, Department = %@, Position = %i\n", [self name],

        [[self dateOfHire] description], [self department],

        [self position]];

}
```

- copyWithZone:如果您希望类的客户代码对类实例进行拷贝，则应该实现这个 NSCopying 协议中的方法。值对象，包括模型对象，是拷贝操作的典型候选者；而诸如 NSWindow 和 NSColorPanel 这样的对象则不是。如果您的类实例是可变的，则应该相应地遵循 NSMutableCopying 协议。
- initWithCoder:和encodeWithCoder: 如果您希望自己设计的类的实例可以支持归档（比如一个模型类），则应该采纳NSCoding协议（必要的话），并实现上面这两个方法。NSCoding方法的更多信息请参见 "入口点和出口点"部分。

如果您设计的类的所有祖先类都采纳了某个正式协议，则必须保证您的类也正确遵循该协议。也就是说，如果在您的类中协议方法的超类实现是不充分的，则应该对其进行重新实现。

错误处理

正确地进行错误处理是不言自明的编程纪律。然而，怎样处理才是“正确”的呢？这取决于不同的编程语言、应用程序环境、以及其它因素。Cocoa 对于子类代码的错误处理有自己的一套约定。

- 如果在方法实现中碰到的错误是系统级错误或 Objective-C 的运行时错误，可以在必要时创建和抛出一个例外。如果可能的话，则最好就在当时进行处理。

  在Cocoa中，例外通常是为编程时或*意料之外*的运行时错误保留的，比如集合的越界访问、试图改变不可修改的对象、发送不正当的消息、以及丢失窗口服务器连接发生的错误。您通常是在编写应用程序时（而不是运行时）通过例外处理这些错误。Cocoa预定义了几个例外，您可以通过例外处理器来捕捉。更多有关这些预定义例外，以及抛出和处理例外的例程和API的信息，请参见 *Cocoa的例外编程主题*。

- 对于其它类型的错误，包括*意料之中*的运行时错误，请向调用者返回 nil、NO、NULL、或一些恰当形式的零值。这种错误的例子包括不能进行文件的读写、对象的初始化错误、不能建立网络连接、或者不能定位集合中的对象。如果您觉得有必要向调用者返回错误的补充信息，请使用 NSError 对象。

NSError 对象封装了错误的有关信息，包括一个错误代码（这个代码可能专门用于 Mach、POSIX、或者 OSStatus 域）和一个包含具体程序信息的字典。直接返回的负值（nil、NO 等等）应该是错误的基本指示器，如果您确实需要表现更多的具体错误信息，则可以通过方法的参数间接返回一个 NSError 对象。

- 如果错误需要用户输入一个选择或动作，则显示一个警告对话框。

  请使用NSAlert类的实例（以及相关的设施）来显示警告对话框和处理用户响应。更多信息请参见*对话框和特殊面板*部分。

有关NSError对象、处理错误、和显示错误警告的更多信息，请参见 *Cocoa错误处理编程指南*。

资源管理和其它效率问题

您可以通过很多方面来增强对象以及负责组合和管理对象的应用程序的性能，包括采纳多线程技术、优化图形的描画、以及使用减少代码印迹的技术。您可以通过阅读 *Cocoa性能指南*和其它性能文档来了解更多这方面的技术。

然而，在采纳更为先进的性能技巧之前，您可以通过遵循下面三个简单的、常识性的指导原则，显著提供对象的性能：

- 在真正需要之前，不要装载资源或分配内存。

  如果您装载了程序的资源，比如一个 nib 文件或一幅图像，而等到之后很久才需要使用，或者根本不需要使用，这就是严重的效率低下。您的程序的内存印迹膨胀了，却没有很好的理由。您应该在马上需要使用的时候，才进行资源的装载和内存的分配。

  举例来说，如果您的应用程序的偏好设置窗口是一个独立的 nib 文件，则在用户首次从应用程序菜单中选择 Preferences（预置）命令之前，不要对其进行装载。为某些任务分配内存也需要同样的谨慎，在需要使用之前，且慢分配内存。这种迟缓装载（lazy-loading）或迟缓分配（lazy-allocation）机制是很容易实现的。举例来说，您的应用程序有一幅图像，且希望在用户首次请求时对其进行装载，以便显示在用户界面上。列表 3-8 显示了一种方法，即在图像的 getter 方法中装载。

列表 3-8 资源的迟缓装载

```
- (NSImage *)fooImage {

    if (!fooImage) { // fooImage is an instance variable

        NSString *imagePath = [[NSBundle mainBundle] pathForResource:@"Foo"
ofType:@"jpg"];

        if (!imagePath) return nil;

        fooImage = [[NSImage alloc] initWithContentsOfFile:imagePath];

    }

    return fooImage;

}
```

- 使用 Cocoa 的 API，不要挖掘更底层的编程接口。

  为了使 Cocoa 框架的实现尽可能强壮、安全、和高效，苹果公司已经付出了很多努力。而且，这些实现处理了框架中的可能不需要您知道的相互依赖性。如果您决定在实现某个任务时不使用 Cocoa API，而是“回退”到底层的接口，则最后您很可能要编写更多代码，并因此增加了发生错误和降低效率的风险。另外，使用 Cocoa 可以更好地为利用未来的增强做好准备，而在底层的实现发生变化时又可以更好地隔离。因此，如果可能的话，请使用 Cocoa 的方法来完成编程任务。举例来说，您可以不使用 BSD 例程来对打开的文件进行查找，而是使用 NSFileHandle 类；或者，您可以使用 NSBezierPath 和 NSColor 方法来进行描画，而不是调用相应的 Quartz（Core Graphics）函数。

- 使用好的内存管理技巧

  为了提高定制对象的效率和强壮性，您能做的最重要的一件事可能就是使用好的内存管理技巧。确保每个对象分配(alloc)、拷贝(copy)、和保持(retain)操作都有一个匹配的释放(release)操作。熟悉正确的内存管理策略和技术，并对其进行实践、实践、再实践。完全的细节请参见 *Cocoa 内存管理编程指南*。

函数、常量、和其它C类型

由于 Objective-C 是 ANSI C 的超集，因此可以在代码中使用任何 C 类型，包括函数、typedef 结构、enum 常量、和宏。一个重要的设计问题是如何在定制类的接口和实现中使用这些类型。

下面的列表为您提供一些在定制类的定义中使用 C 类型的指导原则：

- 将经常使用而又不需要在子类中进行重载的功能定义为函数，而不是方法。这样做是因为性能上的考虑。在这种情况下，最好将代码定义为私有函数，而不是类 API 的一部分。您也可以把和其它类不相关（因为它是全局的）的行为或者对简单类型（C 的原始类型或结构）的操作实现为函数。然而，对于全局的功能，创建一个产生单件实例的类可能更好（因为扩展性的考虑）。

- 在满足下面的条件时，定义一个结构类型比定义一个简单的类更好：

  - 您不希望在字段列表中增加新的成员。
  - 所有的字段都是公共的（由于性能上的原因）。
  - 所有字段都不是动态的（动态字段可能要求特殊的处理，比如保持或释放）。
  - 您不倾向于使用面向对象的技术，比如子类化。

  即使所有的条件都满足，对于 Objective-C 代码中的结构的最基本判断还是性能。换句话说，如果没有显著的性能上的好处，定义一个简单类还是更好一些。

- 声明 enum 常量，而不是使用#define 常量。前者更适合类型判断，您可以在调试器中看到常量的值。

开发公共类的时候

在为自己定义一个定制类，比如应用程序的控制类时，您有很大的灵活性，因为您充分了解这个类，在必要时可以进行重新设计。但是，如果您的定制类可能被其它开发者用作超类—换句话说，它是个公共类—其它人对该类的期望要求您必须更加认真地进行类设计。

下面是给公共的 Cocoa 类开发者的一些原则：

- 确保子类需要访问的实例变量的作用域是@protected。
- 加入一两个保留的实例变量，这有助于将来版本的类的二进制兼容性（参见 "实例变量"部分）。
- 实现自己的存取方法，以正确处理提供给客户的对象的内存管理（参见 "存取方法"部分）。您设计的类的客户代码从getter存取方法得到的对象应该是自动释放的。
- 参照公共Cocoa接口中的命名规则，如 *Cocoa的编码指南*文档中推荐的那样。
- 为类提供文档技术，至少在接口文件中加入注释。

Cocoa的设计模式

Cocoa 环境的很多架构和机制都有效地使用了设计模式：即为特定上下文中反复出现的问题提供解决方案的抽象设计。本文描述设计模式在 Cocoa 框架中的主要实现方式，特别是模型-视图-控制器和对象建模模式，主要目的是使您对 Cocoa 的设计模式有更好的认识，鼓励您在自己的软件工程中利用这些模式。

本部分有如下主要内容：

什么是设计模式？
Cocoa如何适配设计模式
模型-视图-控制器设计模式
对象建模

什么是设计模式？

设计模式是一种设计模板，用于解决在特定环境中反复出现的一般性的问题。它是一种抽象工具，在架构、工程、和软件开发领域相当有用。下面的部分将概要说明什么是设计模式，解释为什么设计模式在面向对象的设计中非常重要，并讨论一个设计模式的实例。

本部分包含如下主要内容：

特定环境下的特定问题的解决方案
例子：命令模式

特定环境下的特定问题的解决方案

作为一个开发者，您可能已经熟悉面向对象编程中的设计模式的概念。对设计模式首次进行权威描述和分类的是*设计模式：可重用的面向对象的软件中的元素（Design Patterns: Elements of Reusable Object-Oriented Software）*一书，作者是 Erich Gamma、Richard Helm、Ralph Johnson、和 John Vlissides（被称为“四人组”）。这本书最初出版于 1994 年，之后不久，就有其它书籍和文章对面向对象系统中的设计模式进行进一步的探索和论述。

设计模式的简单定义是“特定环境下的特定问题的解决方案”，让我们按从前向后的顺序解读一下这个定义。特定环境是指一个经常出现的环境，是设计模式应用的地方；问题是指在这个环境以及和这个环境与生俱来的约束下您希望达成的目标；而解决方案则是您关心的：一个可以在这个环境下达成目标、解决约束的一般性设计。

已经被时间证明为有效的具体设计在结构上有些关键的地方，设计模式对其进行抽象。每个模式都有一个名称，并通过这个名称标识出参与这个模式的类和对象、它们的责任、以及它们之间如何协作。模式还需要说明结果（开销和可以得到的好处）以及在什么情况下可以应用。设计模式是某种特定设计的模板或指导原则，在某种意义上，具体设计是设计模式的一个“实例化”。设计模式并不是绝对的，如何应用模式是有一些灵活性的，而且象编程语言和现有架构这样的因素通常可以决定设计模式的应用方式。

有几个设计原则或主题会影响到设计模式，它们是构筑面向对象系统的重要规则，比如“对变化的系统结构的各个方面进行封装”，和“面向接口进行编程，而不是面向实现”。这些原则表达了一些重要的观点。举例来说，如果您将系统中变化的部分独立出来并进行封装，这些部分就可以独立于其它部分发生变化。特别是，如果您为那些部分定义的接口不和具体实现紧密相联，就可以在之后对其进行修改和扩展，而不影响系统的其它部分。您因此消除了各个部分之间的依赖性，减少它们之间的联结，系统最终可以更加灵活和强壮地适应未来的变化。

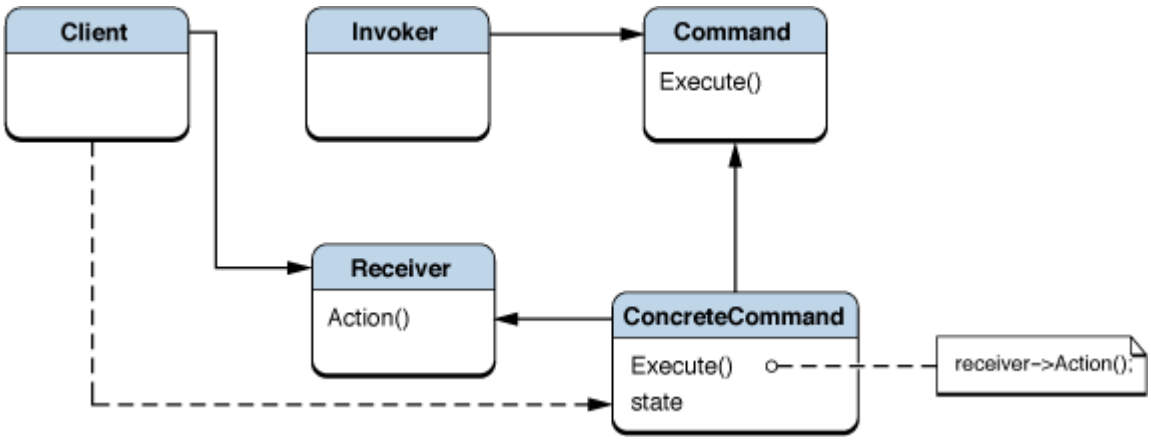
这样的优势使设计模式在编写软件时成为一个重要的考虑。在进行程序设计时，如果您发现、采纳、和使用模式，您的程序—以及程序包含的对象和类—会变得更具可重用性和扩展性，更加易于根据将来需求的改变而改变。而且，基于设计模式的程序通常更加优雅和有效，因为它们可以用更少的代码完成同样的目标。

例子：命令模式

四人组写的那本书大部分是由设计模式的目录组成的。它将模式按作用范围（类或对象）和使用目的（创建、结构、和行为）进行分类。目录中的每个项目讨论一个设计模式的目的、动机、适用性、结构、参与者、协作、结果、和实现，其中的一个模式就是命令模式（一种对象行为模式）。

命令模式陈述的意图是为了“将一个请求封装为对象，您因此可以通过不同的请求使客户进行参数化，对请求进行排队和记录，支持可撤消（undoable）的操作”。这种模式将发送消息的对象和接收及评价那些消息的对象分离开来。消息的始发者（客户）把针对特定接收者的一或多个动作绑定在一起，实现对请求的封装。封装之后的消息可以在对象之间传递，放入队列，或者进行存储，以便以后进行调用，还可以动态地进行修改，以改变接收者或消息的参数。图 4-1 显示了这个模式的结构框图。

图 4-1 命令模式的结构框图



NSInvocation。如这个模式陈述的意图那样，这个类的目的之一是使操作可以撤消。它的对象在 Cocoa 设计中用于撤消（undo）的管理，也用于作为进程间通讯架构的分布式对象管理。命令模式还描述了（虽然不是很完全）Cocoa 的目标-动作机制，用户界面控件对象正是用这种机制封装了用户在激活控件时发出的

消息的目标和动作。

Cocoa的框架类和语言及运行环境已经为您实现了很多类型的设计模式（这些实现在 "Cocoa如何应用设计模式"部分进行描述）。重新使这些设计模式发挥作用可以满足您的很多开发需求。或者，您也可以为自己的问题及其环境约束确定一个全新的、基于模式的设计。重要的是，在您开发软件时，要认识到模式，并正确地将它们用于您的设计。

Cocoa如何应用设计模式

在 Cocoa 中到处都可以找到设计模式的应用，基于模式的机制或架构在 Cocoa 框架和 Objective-C 运行环境及语言中是很常见的。Cocoa 经常把自己与众不同的工作机制建立在模式上，它的设计受到诸如语言能力或现有架构这样因素的影响。

本部分包含*设计模式：可重用的面向对象软件的元素*一书中编目的大多数设计模式的介绍。每个设计模式都有一个总结性的描述，以及该模式的 Cocoa 实现的讨论。文中列出的都是 Cocoa 实现的模式，每个模式的讨论都发生在特定的 Cocoa 环境中。我们推荐您熟悉这些模式，您会发现这些模式在 Cocoa 软件开发中非常有用。

Cocoa 中设计模式的实现有不同的形式。下面部分中描述的一些设计—比如协议和范畴—是 Objective-C 语言的特性；在另外一些场合中，“模式的实例”被实现为一个类或一组相关的类（比如类簇和单件类）；还有一些场合下，模式表现为一个大的框架结构，比如响应者链模式。对于某些基于模式的机制，您几乎可以“免费”使用；而另外一些机制则要求您做一些工作。即使对于 Cocoa 没有实现的模式，我们也鼓励您在条件许可的情况下自行实现，比如在扩展类的行为时，对象合成（装饰模式）技术通常就比生成子类更好。

有两个设计模式没有在下面的内容中进行讨论，即模型-视图-控制器（MVC）模式和对象建模。MVC是一种复合或聚合模式，就是说它基于几种不同类型的模式。对象建模在四人组的分类目录中没有对应类别，它源自关系数据库领域。然而，MVC和对象建模在Cocoa中可能是最重要和最普遍的设计模式或用语，而且它们在很大程度上是相关的。它们在几个技术的设计中发挥关键的作用，包括绑定、撤消管理、脚本控制、和文档架构。要了解更多有关这些模式的信息，请参见 "模型-视图-控制器设计模式"和 "对象建模"部分。

本部分包含如下主要内容：

抽象工厂模式
适配器模式
责任链模式
命令模式
合成模式
装饰模式
表观模式
跌代器模式
仲裁者模式
备忘录模式
观察者模式
代理模式
单件模式
模板方法模式

抽象工厂模式

提供一个接口，用于创建与某些对象相关或依赖于某些对象的类家族，而又不需要指定它们的具体类。通过这种模式可以去除客户代码和来自工厂的具体对象细节之间的耦合关系。

类簇

类簇是一种把一个公共的抽象超类下的一些私有的具体子类组合在一起的架构。抽象超类负责声明创建私有子类实例的方法，会根据被调用方法的不同分配恰当的具体子类，每个返回的对象都可能属于不同的私有子类。

Cocoa将类簇限制在数据存储可能因环境而变的对象生成上。Foundation框架为NSString、NSData、NSDictionary、NSSet、和NSArray对象定义了类簇。公共超类包括上述的不可变类和与其相互补充的可变类NSMutableString、 NSMutableData、NSMutableDictionary、NSMutableSet、和NSMutableArray。

使用和限制

当您希望创建类簇代表的类型的可变或不可变对象时，可以使用类簇中的某个公共类来实现。使用类簇是在简洁性和扩展性之间进行折衷。类簇可以简化类接口，因此使其更易于学习和使用，但是创建类簇抽象类的定制子类则会变得更加困难。

进一步阅读： "类簇" 部分提供有关Cocoa类簇的更多信息。

适配器模式

将一个类接口转化为客户代码需要的另一个接口。适配器使原本由于兼容性而不能协同工作的类可以工作在一起，消除了客户代码和目标对象的类之间的耦合性。

协议

协议是一个编程语言级别（Objective-C）的特性，它使定义适配器模式的实例成为可能（在 Java 中的“接口”和“协议”是同义的）。如果您希望一个客户对象和另一个对象进行交流，但由于它们的接口不兼容而导致困难，您就可以定义一个协议，它本质上是一系列和类不相关联的方法声明。这样，其它对象的类就可以正式采纳该协议，并通过实现协议中的全部方法来“遵循”该协议。结果，客户对象就可以通过协议接口向其它对象发送消息。

协议是一组独立于类层次的方法声明，这样就有可能象类的继承那样，根据对象遵循的协议对其进行分组。您可以通过NSObject的 conformsToProtocol:方法来确认一个对象的协议关系。

除了正式协议之外，Cocoa还有一个非正式协议的概念。这种类型的协议是NSObject类中的一个范畴（category），这样就使所有的对象都成为范畴方法的潜在实现者（参见 "范畴"部分）。非正式协议的方法可以选择性地实现。非正式协议是委托机制实现的一部分（参见 "委托"部分。
请注意，协议的设计和适配器模式并不完全匹配。但它是使接口不兼容的类在得以协同工作的手段。

使用和限制

协议主要用于声明层次结构上不相关的类为了互相通讯而需要遵循的接口。但是，您也可以将协议用于声明对象的接口，而隐藏相应的类。Cocoa框架包括很多正式协议，这些协议使定制子类可以和框架进行特

定目的的通讯。举例来说，Foundation框架中包含 NSObject、NSCopying、和NSCoding协议，都非常重要。Application Kit中的协议包括NSDraggingInfo、NSTextInput、和NSChangeSpelling。

正式协议要求遵循者实现协议声明的*所有*方法。它们也是零碎的，一旦您定义一个协议并将它提供给其它类，将来对协议的修改会使那些类不能工作。

进一步阅读：有关正式协议的更多信息请参见 *Objective-C编程语言*文档中"扩展类"部分的讨论。

责任链模式

通过为多个对象提供处理请求的机会，避免请求的发送者和接收者产生耦合。将接收对象串成链，并将请求沿着接收者链进行传递，直到某个对象对其进行处理。对象或者处理该请求，或者将它传递给链中的下一个对象。

响应者链

Application Kit 框架中包含一个称为响应者链的架构。该链由一系列响应者对象（就是从 NSResponder 继承下来的对象）组成，事件（比如鼠标点击）或者动作消息沿着链进行传递并（通常情况下）最终被处理。如果给定的响应者对象不处理特定的消息，就将消息传递给链中的下一个响应者。响应者在链中的顺序通常由视图的层次结构来决定，从层次较低的响应者对象向层次较高的对象传递，顶点是管理视图层次结构的窗口对象，窗口对象的委托对象，或者全局的应用程序对象。事件和动作消息在响应者链中的确切传递路径是不尽相同的。一个应用程序拥有的响应者链可能和它拥有的窗口（甚至是局部层次结构中的视图对象）一样多，但每次只能有一个响应者链是活动的—也就是与当前活动窗口相关联的那个响应链。

还有一个与响应者链相类似的链，用于应用程序的错误处理。

视图层次的设计应用的是合成模式（参见 "合成模式"部分），它和响应者链密切相关。动作消息—源自控件对象的消息—基于目标-动作机制，是命令模式（参见 "命令模式"部分）的一个实例。

使用和限制

当您通过 Interface Builder 或以编程的方式为程序构造用户界面时，可以"免费"得到一个或多个响应者链。响应者链和视图层次结构一起出现，当您使一个视图对象成为窗口内容视图的子视图时，视图层次结构就自动生成了。如果您将一个定制视图加入到一个视图层次结构中，它就变成响应者链的一部分；如果您实现了恰当的 NSResponder 方法，就可以接收和处理事件及动作消息。定制对象是窗口对象或全局应用程序对象 NSApp 的委托对象，也可以接收和处理那些消息。

您也可以用编程的方式将定制的响应者对象注入到响应者链中，以及通过编程操作响应者在链中的顺序。

进一步阅读： 处理事件和动作消息及程序错误的的响应者链在*Cocoa事件处理指南*和*Cocoa的错误处理编程指南*文档中进行描述。本文档在 "合成模式"部分对视图层次结构有总结性的介绍，在 "核心应用程序架构"部分有更全面的描述。

命令模式

这种模式将请求封装为对象，使您可以用不同的请求来对客户代码进行参数化；对请求进行排队和记录，并支持可撤消（undoable）的操作。请求对象将一或多个动作绑定在特定的接收者上。命令模式将发出请求的对象和接收及执行请求的对象区分开来。

调用对象

NSInvocation 类的实例用于封装 Objective-C 消息。一个调用对象中含有一个目标对象、一个方法选择器、以及方法参数。您可以动态地改变调用对象中消息的目标及其参数，一旦消息被执行，您就可以从该对象得到返回值。通过一个调用对象可以多次调用目标或参数不同的消息。

创建 NSInvocation 对象需要使用 NSMethodSignature 对象，该对象负责封装与方法参数和返回值有关系的信息。NSMethodSignature 对象的创建又需要用到一个方法选择器。NSInvocation 的实现还用到 Objective-C 运行环境的一些函数。

使用和限制

NSInvocation 对象是分布式、撤消管理、消息传递、和定时器对象编程接口的一部分。在需要去除消息发送对象和接收对象之间的耦合关系的类似场合下，您也可以使用。

分布式对象是一种进程间通讯技术，关于这个主题的更多信息请参见 "代理模式"部分。

进一步阅读：调用对象的细节请阅读NSInvocation的类参考文档。您也可以从下面的文档中获取相关技术的信息：*Objective-C编程语言*文档中的*分布式对象*、*撤消架构*、*定时器*以及之后的部分。

目标-动作

目标-动作机制使控件对象—也就是象按键或文本输入框这样的对象—可以将消息发送给另一个可以对消息进行解释并将它处理为具体应用程序指令的对象。接收对象，或者说是目标，通常是一个定制的控制器对象。消息—也被称为动作消息—由一个选择器来确定，选择器是一个方法的唯一运行时标识。典型情况下，控件拥有的单元对象会对目标和动作进行封装，以便在用户点击或激活控件时发送消息（菜单项也封装了目标和动作，以便在用户选择时发送动作消息）。目标-动作机制之所以能够基于选择器（而不是方法签名），是因为 Cocoa 规定动作方法的签名和选择器名称总是一样的。

使用和限制

当您用 Interface Builder 构建程序的用户界面时，可以对控件的动作和目标进行设置。您因此可以让控件具有定制的行为，而又不必为控件本身书写任何的代码。动作选择器和目标连接被归档在 nib 文件中，并在 nib 文件被解档时复活。您也可以通过向控件或它的单元对象发送 setTarget:和 setAction:消息来动态地改变目标和动作。

目标-动作机制经常用于通知定制控制器对象将数据从用户界面传递给模型对象，或者将模型对象的数据显示出来。Cocoa绑定技术则可以避免这种用法，有关这种技术的更多信息请参见 *Cocoa绑定编程主题*文档。

Application Kit中的控件和单元并不保持它们的目标。相反，它们维护一个对目标的弱引用。进一步的信息请参见 "委托、观察者、和目标的所有权" 部分。

进一步阅读：更多信息请参见 "目标-动作机制"部分。

合成模式

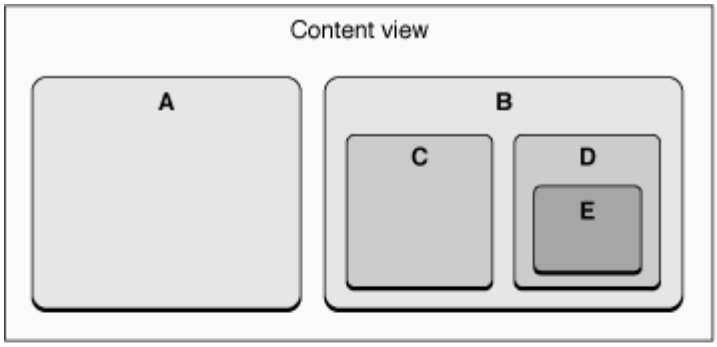
这种模式将互相关联的对象合成为树结构，以表现部分-全部的层次结构。它使客户代码可以统一地处理单独的对象和多个对象的合成结果。

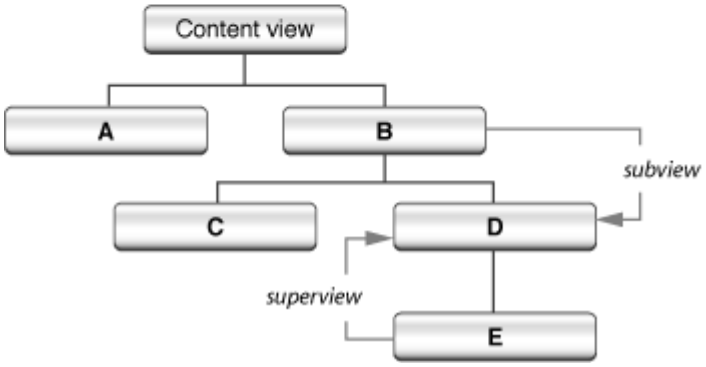
合成对象是模型-视图-控制器聚集模式的一部分。

视图层次结构

一个窗口包含的视图对象（NSView 对象）在内部构成了一个视图层次结构。层次结构的根是窗口对象（NSWindow 对象）和它的内容视图。内容视图就是填充到窗口内容方框中的透明视图，添加到内容视图中的视图都是它的子视图，而这些子视图又会成为下一级视图的父视图。一个视图有一个（且只有一个）父视图，可以有零或多个子视图。在视觉上，您可以将这个结构理解为包含关系：父视图包含它的子视图。图 4-2 显示视图层次的结构以及在视觉上的关系。

图 4-2 视图层次的结构及其在视觉上的关系





视图层次是一个结构方面的架构，在图形描画和事件处理上都扮演一定的角色。一个视图有两个影响图形操作位置的边界框，即边框（frame）和边界（bound）。边框是外部边界，表示视图在其父视图坐标系统中的位置，它负责定义视图的尺寸，并根据视图边界对图形进行裁减。边界则是内部的边界框，负责定义视图对象自身描画表面的内部坐标系统。

当窗口系统要求一个窗口做好显示准备时，父视图会在其子视图之前被要求进行渲染。当您向一个视图发送消息时—比如发送一个重画视图的消息—该消息就会被传播到子视图。因此，您可以将视图层次结构中的一个分支当成一个统一的视图来处理。

响应者链也把视图层次用于处理事件和动作消息。请参见责任链模式部分中（"责任链模式"）关于响应者链的总结描述。

使用和限制

无论以编程的方式，还是通过 Interface Builder，当您将一个视图加入到另一个视图中时，都需要创建或修改视图层次结构。Application Kit 框架自动处理与视图层次结构相关联的所有关系。

进一步阅读：如果需要了解更多视图层次结构的信息，请阅读本文档的 "视图层次结构"部分。*Cocoa描画指南*文档中也对视图层次结构进行讨论。

装饰模式

这种模式动态地将额外的责任附加到一个对象上。在进行功能扩展时，装饰是子类化之外的一种灵活的备选方法。和子类化一样，采纳装饰模式可以加入新的行为，而又不必修改已有的代码。装饰将需要扩展的类的对象进行包装，实现与该对象相同的接口，并在将任务传递给被包装对象之前或之后加入自己的行为。装饰模式表达了这样的设计原则：类应该接纳扩展，但避免修改。

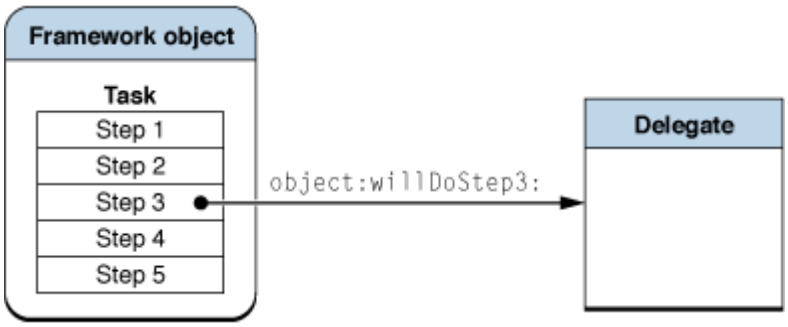
一般性的说明

装饰是用于对象合成的模式。在您自己的代码中应该鼓励使用对象的合成（参见 "什么时候需要生成子类"部分）。然而，Cocoa自己提供了一些基于这种模式的类和机制（在下面的部分进行讨论）。在这些实现中，扩展对象并不完全复制它所包装的对象的接口，虽然具体实现中可以使用不同的技术来进行接口共享。Cocoa 在实现某些类时用到了装饰模式，包括 NSAttributedString、NSScrollView、和 NSTableView。后面两个类是复合视图的例子，它们将其它一些视图类的对象组合在一起，然后协调它们之间的交互。

委托

委托是在宿主对象中嵌入一个指向另一对象（也就是委托对象）的弱引用（一个未保持的插座变量），并不时地向该委托对象发送消息，使其对有关的任务进行输入的机制。宿主对象一般是一个“复活”的框架对象（比如一个 NSWindow 或 NSXMLParser 对象），它寻求完成某项工作，但又只能以一般的方式来进行。委托几乎总是一个定制类的实例，它负责配合宿主对象，在有关任务的特定点（参见图 4-3）上提供与具体程序有关的行为。这样，委托机制使我们可以对另一个对象的行为进行修改或者扩展，而不需要生成子类。

图 4-3 框架对象向它的委托对象发送消息



简而言之，委托就是一个对象将任务委托给另一个对象，它是面向对象编程的常用技术。然而，Cocoa以独特的方式实现委托机制。宿主类用非正式协议—即NSObject中的范畴—来定义委托对象可能选择实现的接口。委托对象不必象采纳正式协议那样实现所有的协议方法。在向委托对象发送消息之前，宿主对象可以首先确定相应的方法是否实现（通过 respondsToSelector:消息），以避免运行时例外。有关正式协议和非正式协议的更多信息，请参见 "协议"部分。

Cocoa 框架中的一些类也向它们的数据源发送消息。数据源在各个方面都和委托一样，除了它的目的是为宿主对象提供数据，以传递给浏览器、表视图、或者类似的用户界面视图。和委托不同的是，数据源还必须实现某些协议方法。

委托不是装饰模式的严格实现。宿主（委托）对象并没有包装它希望扩展的类的实例。相反，委托是对委托框架类的行为进行特殊化。除了框架类声明的委托方法之外，它们也没有公共的接口。

Cocoa中的委托也是模板方法模式（"模板方法模式"）的一部分。

使用和限制

委托是 Cocoa 框架中的一种常用的设计。Application Kit 框架中的很多类都向它们的委托发送消息，包括 NSApplication、NSWindow、和 NSView 的几个子类。Foundation 框架中的一些类，比如 NSXMLParser 和 NSStream，也维护自己的委托。您应该总是使用类的委托机制，而不是生成类的子类，除非委托方法不能完成您的目标。

虽然您可以动态地改变委托，但是同时只能有一个对象可以成为委托。因此，如果您希望当特定的程序事件发生时，有多个对象可以同时得到通知，则不能使用委托。在这种情况下您可以使用通告机制。如果委托对象实现了一或多个框架类声明的通告方法，就会自动接收到其委托框架对象的通告。请参考观察者模式（ "观察者模式"）中有关通告的讨论。
Application Kit框架中的向外委托任务的对象并不保持它们的委托或数据源，而是维护一个弱引用，更多信息请参见 "委托、观察者、和目标的所有权"部分。

进一步阅读：关于委托的进一步信息请参见 "委托和数据源"部分。

范畴

范畴是 Objective-C 语言的一个特性，用于为一个类增加方法（接口和实现），而不必生成子类。类原始声明的方法和通过范畴添加的方法在运行时没有区别—在您的程序的作用范围内。范畴中的方法成为类类型的一部分，并被所有的子类继承。

和委托一样，范畴并没有严格适配装饰模式。它实现了该模式的目的，但采用不同的实现方式。范畴加入的行为是在编译时生成的，而不是动态得到的。而且，范畴并没有封装被扩展的类的实例。

使用和限制

Cocoa框架中定义了很多范畴，大多数都是非正式协议（在 "协议"部分中进行总结）。它们通常使用范畴来对相关的方法进行分组。您也可以在代码中实现范畴，以在不生成子类的情况下对类进行扩展，或者对相关的方法进行分组。但是您需要注意如下两点：

- 您不能为类添加实例变量。
- 如果您对现有的方法进行重载，则应用程序可能产生预料之外的行为。

进一步阅读: 更多有关范畴的信息请参见 *Objective-C编程语言*一文中的“类的扩展”部分。

表观模式

这种模式为子系统中的一组接口提供统一的接口。表观模式定义一个更高级别的接口，通过减少复杂度和隐藏子系统之间的通讯和依赖性，使子系统更加易于使用。

NSImage

NSImage 类为装载和使用基于位图（比如 JPEG、PNG、或者 TIFF 格式）或向量（EPS 或 PDF 格式）的图像提供统一的接口。NSImage 可以为同一个图像保持多个表示，不同的表示对应于不同类型的 NSImageRep 对象。NSImage 可以自动选择适合于特定数据类型和显示设备的表示。同时，它隐藏了图像操作和选择的细节，使客户代码可以交替使用很多不同的表示。

使用和限制

由于 NSImage 支持几种不同的图像表示，因此某些属性可能不能适用。举例来说，您可能不能取得图像中一个像素的颜色，如果潜在的图像表示使基于向量且与设备无关的话。

请注意：NSImage和图像表示的讨论请参见 *Cocoa描画指南*。

迭代器模式

这种模式提供一种顺序访问聚合对象（也就是一个集合）中的元素，而又不必暴露潜在表示的方法。迭代器模式将访问和遍历集合元素的责任从集合对象转移到迭代器对象。迭代器定义一个访问集合元素的接口，并对当前元素进行跟踪。不同的迭代器可以执行不同的遍历策略。

NSEnumerator

Foundation 框架中的 NSEnumerator 类实现了迭代器模式。NSEnumerator 抽象类的私有具体子类返回的枚举器对象可以顺序遍历不同类型的集合—数组、集合、字典（值和键）—并将集合中的对象返回给客户代码。

NSDirectoryEnumerator 是一个不紧密相关的类，它的实例可以递归地枚举文件系统中目录的内容。

使用和限制

象 NSArray、NSSet、和 NSDictionary 这样的集合类都包含相应的方法，可以返回与集合的类型相适用的枚举器。所有的枚举器的工作方式都一样。您可以在循环中向枚举器发送 nextObject 消息，如果该消息返回 nil，而不是集合中的下一个对象，则退出循环。

仲裁者模式

这种模式定义的对象用于封装一组对象的交互机制。仲裁者模式可以避免对象之间显式的互相引用，使对象之间的耦合变得宽松，也使您可以独立地改变它们的交互方式。这些对象也因此可以更具重用性。

仲裁者对象集中了系统中的对象之间的复杂通讯和控制逻辑。这些对象在状态发生改变时会告诉仲裁者对象，反过来，也对仲裁者对象的请求进行响应。

控制器类

模型-视图-控制器设计模式为一个面向对象的系统（比如一个应用程序）中的对象分配不同的角色。它们可以是模型对象，包含应用程序的数据及对那些数据进行操作；可以是视图对象，负责表示数据及响应用户动作；也可以是控制器对象，负责协调模型和视图对象。控制器对象适合于仲裁者模式。

在 Cocoa 中，控制器对象一般有两个类型：仲裁控制器或者协调控制器。仲裁控制器负责仲裁应用程序中视图对象和模型对象之间的数据流。仲裁控制器通常是 NSController 对象。协调控制器则负责实现应

用程序的集中化通讯和控制逻辑，作为框架对象的委托和动作消息的目标。它们通常是 NSWindowController 对象或定制 NSObject 子类的实例。由于协调控制器高度专用于特定的程序，因此不考虑重用。

Application Kit 框架中的 NSController 抽象类和它的具体子类是 Cocoa 绑定技术的一部分，该技术可以自动同步模型对象包含的数据和视图对象中显示及编辑的数据。举例来说，如果用户在一个文本框中编辑一个字符串，绑定技术会将文本框中的变化（通过仲裁控制器）传递给绑定了的模型对象中合适的属性。编程者需要做的就是正确设计自己的模型对象，并通过 Interface Builder 在程序的视图、控制器、和模型对象之间建立绑定关系。

具体的公共控制器类的实例可以在 Interface Builder 的控件选盘上得到，因此是高度可重用的。它们提供一些服务，比如选择和占位符值的管理。这些对象执行下面这些特定的功能：

- NSObjectController 管理一个单独的模型对象。
- NSArrayController 管理一个模型对象数组，以及维护一个选择；还可以在数组中加入或删除模型对象。
- NSTreeController 使您可以在一个具有层次的树结构中添加、删除、和管理模型对象。
- NSUserDefaultsController 为预置（用户缺省值）系统提供一个便利的接口。

使用和限制

您通常可用将 NSController 对象用作仲裁控制器，因为这些对象的设计目的是在应用程序的视图对象和模型对象之间传递数据。在使用仲裁控制器时，您通常是从 Interface Builder 选盘中拖出对象，指定模型对象的属性键，并通过 Interface Builder Info 窗口中的 Bindings 面板建立视图和模型对象之间的绑定关系。您也可以生成 NSController 或其子类的子类，以获得更具具体行为的子类。

几乎任何一对对象之间都可以建立绑定关系，只要它们遵循 NSKeyValueCoding 和 NSKeyValueObserving 这两个非正式协议。但是，我们推荐您通过仲裁控制器来建立绑定，以得到 NSController 及其子类为您提供的各种好处。

协调控制器通过下面的方式集中实现一个应用程序的通讯和控制逻辑：

- 维护指向模型和视图对象的插座变量（插座变量是指向其它被保持为实例变量的连接或引用）。
- 通过目标-动作机制响应用户在视图对象上的操作（参见 "目标-动作"部分）。
- 作为委托对象，处理从框架对象发出的消息（参见 "委托"部分）。

您通常可以在 Interface Builder 中建立上述的连接—插座变量、目标-动作、和委托，它将这些连接归档到应用程序的 nib 文件中。

进一步阅读: 有关仲裁控制器、协调控制器、和与控制器有关的设计决定的讨论，请参见 "模型-视图-控制器设计模式"部分。*Cocoa绑定编程主题*一文中则对仲裁控制器进行详细的描述。

备忘录模式

这种模式在不破坏封装的情况下，捕捉和外部化对象的内部状态，使对象在之后可以回复到该状态。备忘录模式使关键对象的重要状态外部化，同时保持对象的内聚性。

归档

归档将一个程序中的对象以及对象的属性（包括属性和关系）存储到档案上，使之可以保存到文件系统中，或者在不同的处理器和网络间传递。档案将程序的对象图保存为独立于架构的字节流，对象的标识和对象之间的关系都会被保留。由于对象的类型和它的数据一起被存储，从归档的字节流解码出来的对象会被正常实例化，实例化所用的类与原来编码的类相同。

使用和限制

通常情况下，您希望将程序中需要保存状态的对象归档。模型对象几乎总是属于这个范畴。您通过编码将对象写入到档案中，而通过解码将对象从档案中读取出来。通过 NSCoder 对象可以执行编解码操作，在编解码过程中最好使用键化的归档技术（需要调用 NSKeyedArchiver 和 NSKeyedUnarchiver 类的方法）。被编解码的对象必须遵循 NSCoding 协议；该协议的方法在归档过程中会被调用。

进一步阅读: 有关归档的进一步信息，请参见 *Cocoa 的归档和序列化编程指南*。

属性列表的序列化

属性列表是一个简单的、具有一定结构的对象图序列，它仅使用下面这些类的对象：NSDictionary、NSArray、NSString、NSData、NSDate、和 NSNumber。这些对象通常也被称为属性列表对象。Cocoa 中有几个框架类提供了序列化属性列表对象，以及定义录写对象内容及其层次关系的特殊数据流格式的方法。NSPropertyListSerialization 类就提供了将属性列表对象序列化为 XML 或其它优化的二进制格式的类方法。

使用和限制

如果对象图中包含的是简单对象，则在捕捉和外部化对象及其状态时，属性列表序列化是一种灵活的、可移植的、而又非常适当的工具。然而，这种形式的序列化有它的限制，它不保留对象的全部类标识，而只保留一些一般的类型（数组、字典、字符串、等等）。这样，从属性列表恢复出来的对象可能和原来的类不同，特别是当对象的可变性可能发生变化时，这就会带来问题。属性列表序列化也不跟踪在同一对象中被多次引用的对象，这可能导致反向序列化时产生多个实例，而在原来的对象图中却只有一个实例。

进一步阅读: 有关属性列表序列化的更多信息，请参见 *Cocoa的归档和序列化编程指南*。

Core Data

Core Data 是一个管理对象图，并使其留存的框架和架构。正是第二种能力—对象的留存能力—使 Core Data 成为备忘录模式的一种适配形式。

在 Core Data 架构中，中心的对象称为被管理对象上下文，负责管理应用程序对象图中的模型对象。在被管理对象上下文下面是该对象图的持久栈，也就是一个框架对象的集合，负责协调模型对象和外部数据存储，比如 XML 文件或关系数据库。持久栈对象负责建立存储中的数据和被管理对象上下文中的对象之间的映射关系，在有多个数据存储的时候，持久栈对象将这些存储表现为被管理对象上下文中的一个聚合存储。

Core Data 的设计也在很大程度上受到模型-视图-控制器以及对象建模模式的影响。

重要信息： Core Data 框架是在 Mac OS X v10.4 引入的。

使用和限制

Core Data 在开发企业级应用程序时特别有用，这些程序需要定义、管理、以及从数据存储中透明地归档和解档复杂模型对象图。Xcode 开发环境中包含有关的工程模板和设计工具，在创建两种一般类型的 Core Data 应用程序（即基于文档和非基于文档的类型）时，这些模板和工具可以节省一些编程的工作量。Interface Builder 也在其选盘中包含可配置的 Core Data 框架对象。

进一步阅读: 在本文档中，"其它Cocoa架构"部分中包含对Core Data的总结。更多Core Data的信息，请阅读*Core Data编程指南*。 *NSPersistentDocument Core Data教程*和*底层Core Data教程*则一步一步地教您创建基于文档和非基于文档的Core Data应用程序的基本流程。

观察者模式

这种模式定义一种对象间一对多的依赖关系，使得当一个对象的状态发生变化时，其它具有依赖关系的对象可以自动地被通知和更新。观察者模式本质上是个发布-定阅模型，主体和观察者具有宽松的耦合关系。观察和被观察对象之间可以进行通讯，而不需要太多地了解对方。

通告

Cocoa 的通告机制实现了一对多的消息广播，其实现方式符合观察者模式。在这种机制中，程序里的对象将自己或其它对象添加到一或多个通告的观察者列表中，每个通告由一个全局的字符串（即通告的名称）标识。希望向其它对象发送通知的对象－也就是被观察的对象－负责创建一个通告对象，并将它发送到通告中心。通告中心则负责确定通告的观察者，并通过消息将通告发送给观察者对象。通告消息激活的方法必须遵循特定的单参数签名格式，方法的参数就是通告对象，包含通告的名称、被观察的对象、以及一个含有补充信息的字典。

通告的发送是一个同步的过程。在通告中心将通告广播给所有的观察者之前，发送对象不会再得到程序的控制权。对于异步的处理方式，控制权会在您将通告放入通告队列中之后立即返回到发送对象，当通告到达队列的顶部时，通告中心会将它广播出去。

常规的通告－也就是那些由通告中心广播的通告－只能在进程内部传播。如果您希望将通告广播给其它进程，需要使用分布式通告中心及其相关的 API。

使用和限制

使用通告可以有很多原因。例如，借助通告机制，您可以根据程序中其它地方发生的事件改变用户界面元素显示信息的方式。或者，您可以用通告来保证文档中的对象在文档窗口关闭之前保存自己的状态。通告的一般目的是将事件通知给其它程序对象，使它们可以做出恰当的反应。

但是，通告的接收对象只能在事件发生之后进行反应，这和委托机制有显著的不同。被委托的对象有机会拒绝或修改委托对象希望进行的操作。另一方面，观察者对象不能直接影响一个即将发生的操作。

与通告有关的类有 NSNotification（通告对象）、NSNotificationCenter（用于发送通告和添加观察者）、NSNotificationQueue（负责管理通告队列）、和 NSDistributedNotificationCenter。很多 Cocoa 框架都发布和发送通告，其它对象都可以成为这些通告的观察者。

进一步阅读: "通告"部分对通告机制进行更详细地描述，并提供其用法的指南。

键-值观察

键-值观察是使对象可以在其它对象的具体属性发生变化时得到通知的一种机制。它基于名为 NSKeyValueObserving 的非正式协议。被观察的属性可以是简单的属性、一对一的关系、或者一对多的关系。键-值观察在模型-视图-控制器模式中特别重要，因为它使视图对象－通过控制器层－可以观察到模型对象的变化，因此是Cocoa绑定技术的必要组件（参见 "控制器类"部分）。
Cocoa 为很多 NSKeyValueObserving 方法提供了缺省的“自动”实现，使所有遵循该协议的对象都具有属性-观察的能力。

使用和限制

键-值观察和通告机制类似，但在一些重要的方面也有不同。在键-值观察中，没有为所有观察者提供变化通告的中心对象，而是将变化的通告直接传递给观察对象。还有，键-值观察直接和具体的对象属性相关联。而通告机制则更广泛地关注程序的事件。

参与键-值观察（KVO）的对象必须满足特定的要求－或者说是遵循 KVO，记忆起来更为方便。对于自动观察来说，这需要满足键-值编码的要求（也称为遵循 KVC），并使用遵循 KVC 的方法（也就是存取方法）。键-值编码是一个与自动获取和设置对象属性值有关的机制（基于相关的非正式协议）。

您可以禁用自动的观察者通告，并通过 NSKeyValueObserving 非正式协议及相关范畴中的方法实现手工通告，从而对 KVO 通告进行精化。

进一步阅读： 阅读 *键-值观察编程指南*可以进一步了解其机制和潜在的协议。还可以阅读 *键-值编码编程指南*和 *Cocoa绑定技术编程指南*两个相关文档。

代理模式

这种模式为某些对象定义接口，使其充当其它对象的代理或占位对象，目的是进行访问控制。这种模式可以用于为一个可能是远程的、创建起来开销很大的、或者需要保证安全的对象创建代表对象，并在代表对象中为其提供访问控制的场合。它在结构上和装饰模式类似，但服务于不同的目的；装饰对象的目的是为另一个对象添加行为，而代理对象则是进行访问控制。

NSProxy

NSProxy 类定义的接口适用于为其它的、甚至是尚未存在的对象充当代理或占位符的对象。代理对象通常将消息转发给自己代表的对象，但是也可以通过装载其代表的对象或对自身进行改造来对消息进行响应。虽然 NSProxy 是一个抽象类，但它实现了 NSObject 协议及其它根对象应该具有的基本方法，它实际上是和 NSObject 类一样，是一个类层次的根类。

NSProxy 的具体子类可以完成代理模式规定的目标，比如对开销大的对象进行迟缓实例化，或者实现安全守卫对象等。NSDistantObject 就是 Foundation 框架中的 NSProxy 类的一个具体子类，负责为透明的分布式消息传递实现远程的代理对象。 NSDistantObject 对象是分布式对象架构的一部分。这些对象充当其它进程或线程中的对象的代理，帮助实现那些线程或进程中的对象之间的通讯。

NSInvocation对象采纳了命令模式，也是分布式对象架构的一部分（参见 "调用对象"部分）。

使用和限制

Cocoa 只在分布式对象中使用 NSProxy 对象。NSProxy 对象是 NSDistantObject 和 NSProtocolChecker 两个具体子类的特殊实例。分布式对象不仅可以用于进程间（在同一台或不同

的计算机上都可以）的消息传递，还可以用于实现分布式计算或并行处理。如果您希望将代理对象用于其它目的，比如昂贵资源的创建或者安全，则需要实现您自己的 NSProxy 具体子类。

进一步阅读: 要进一步了解 Cocoa 代理对象以及它们在分布式消息传递中发挥的作用，请阅读*分布式对象*部分。

单件模式

这种模式确保一个类只有一个实例，并提供一个全局的指针作为访问通道。该类需要跟踪单一的实例，并确保没有其它实例被创建。单件类适合于需要通过单个对象访问全局资源的场合。

框架类

有几个 Cocoa 框架类采用单件模式，包括 NSFileManager、NSWorkspace、和 NSApplication 类。这些类在一个进程中只能有一个实例。当客户代码向该类请求一个实例时，得到的是一个共享的实例，该实例在首次请求的时候被创建。

使用和限制

使用由单件类返回的共享实例和和使用非单件类的实例没有什么不同，只是您不能对该实例进行拷贝、保持、和释放（相关的方法被重新实现为空操作）。您在条件许可的情况下也可以创建自己的单件类。

进一步阅读: "Cocoa对象" 部分解释如何创建一个单件类。

模板方法模式

这种模式为某个操作中的算法定义框架，并将算法中的某些步骤推迟到子类实现。模板方法模式使子类可以重定义一个算法中的特定步骤，而不需要改变算法的结构。

可重载的框架方法

模板方法模式是 Cocoa 的基本设计，事实上也是一般的面向对象框架的基本设计。Cocoa 中的模式使一个程序的定制组件可以将自己挂到算法上，但何时和如何使用这些定制组件，由框架组件来决定。Cocoa 类的编程接口通常包括一些需要被子类重载的方法。在运行环境中，框架会在自己所执行的任务过程中的某些点调用这些所谓的一般方法。一般方法为定制代码提供一个结构，目的是为当前正在执行且由框架类负责协调的任务加入具体程序的的行为和数据。

使用和限制

为了使用 Cocoa 采纳的模板方法模式，您必须创建一个子类，并重载有关的方法。框架会调用这些方法，并将具体应用程序的信息输入到正在执行的算法中。如果您正在编写自己的框架，则可能应该将这个模式包含到您的设计中。

请注意： "为Cocoa程序添加行为" 部分中讨论了Cocoa对模板方法模式的使用，特别是 "Cocoa类的继承" 部分。

文档架构

Cocoa 的文档架构就是一个特别的（也是重要的）、采纳模板方法模式进行框架方法重载的设计实例。几乎所有创建和管理多个文档、而且每个文档有各自显示窗口的 Cocoa 应用程序都基于文档架构。在这个架构中，有三个互相协作的框架类：NSDocument、NSWindowController、和

NSDocumentController。NSDocument 对象管理模型对象，代表文档的数据，它会根据用户的请求将数据写到文件中，或者重新装载数据并用这些数据重建模型对象；NSWindowController 对象管理负责管理特定文档的用户界面；基于文档的应用程序的 NSDocumentController 对象管理负责跟踪和管理所有打开的文档，或者协调应用程序的活动。在运行时，这些对象从 Application Kit 接收要求执行具体操作的消息。应用程序开发者必须重载很多由这些消息激活的方法，以便添加应用程序的具体行为。

Cocoa 文档架构的设计在很大程度上也受到模型-视图-控制器模式的影响。

使用和限制

您可以在 Xcode 的新建工程（New Project）助手中选择基于文档的 Cocoa 应用程序（Cocoa Document-based Application）模板，创建基于文档的 Cocoa 应用程序工程。然后，您需要实现一个 NSDocument 的定制子类，选择实现 NSWindowController 和 NSDocumentController 的定制子类。Application Kit 为您提供很多应用程序需要的文档管理逻辑。

请注意：本文档的 "其它Cocoa架构"部分中包含一些有关文档架构的概括性描述。如果需要如何应用模板方法模式的权威文档，请参见 *基于文档的应用程序概述*。

模型-视图-控制器设计模式

模型-视图-控制器模式（MVC）是一个相当老的设计模式，它的一些变体至少在 Smarttalk 的早期就出现了。它是一种高级别的模式，关注的是应用程序的全局架构，并根据各种对象在程序中发挥的作用对其进行分类。它也是个复合的模式，因为它是由几个更加基本的模式组成的。

面向对象的程序在设计上采用 MVC 模式会带来几个方面的好处。这种程序中的很多对象可能更具重用性，它们的接口也可能定义得更加良好。程序从总体上更加适应需求的改变—换句话说，它们比不基于 MVC 的程序更加容易扩展。而且，Cocoa 中的很多技术和架构—比如绑定技术、文档架构、和脚本技术—都基于 MVC，而且要求您的定制对象充当 MVC 定义的某种角色。

本部分包含如下主要内容：

MVC对象的作用和关系

MVC 设计模式考虑三种对象：模型对象、视图对象、和控制器对象。模式定义了这三种对象在应用程序中充当的角色，以及它们的通讯路径。在设计应用程序时，一个主要的步骤就是进行这三种对象的选择－或者说为这三种对象创建定制类。三种对象中的每一种都和其它两种按抽象的边界区分，并和其它两种对象进行跨边界的通讯。

模型对象负责包装数据和基本行为

模型对象代表特别的知识和专业技能，它们负责保有应用程序的数据和定义操作数据的逻辑。一个定义良好的 MVC 应用程序会将所有重要的数据封装在模型对象中。任何代表应用程序留存状态的数据（无论该

状态存储在文件中，还是存储在数据库中），一旦载入应用程序，就应该驻留在模型对象中。因为它们代表与特定问题域有关的知识和专业技能，所以有可能被重用。

在理想情况下，模型对象不和负责表示与编辑模型数据的用户界面建立显式的连接。举例来说，如果有个代表一个人的模型对象（假定您在编写一个地址本），您可能希望存储这个人的生日，则将生日存储在您的 Person 模型对象是比较好的做法。但是，日期格式字符串或其它有关日期如何表示的信息可能存储在别的地方比较好。

在实践上，这种分隔并不总是最好的，这里有一定的灵活空间。但一般来说，模型对象不应该关心界面和表示的问题。一个具有合理例外的例子是描画程序，它的模型对象代表要显示的图形。图形对象知道如何描画自身是合理的，因为它们存在的主要原因就是为了定义视觉上的信息。但是即使在这种情况下，图形对象也不应该完全依赖于特定的视图，它们不应该负责描画的具体位置，而应该由希望表示这些图形对象的视图对象发出描画的请求。

进一步阅读：*模型对象实现指南*文档种讨论模型对象的正确设计和实现。

视图对象负责向用户表示信息

视图对象知道如何显示应用程序的模型数据，而且可能允许用户对其进行编辑。视图对象不应该负责存储它所显示的数据（这当然不是说视图永远不存储它所显示的数据。由于性能上的原因，视图可能对数据进行缓存，或使用类似的技巧）。一个视图对象可能负责显示模型对象的一部分或全部，甚至是很多不同的模型对象。视图对象可能有很多变化。

视图对象应该尽可能可重用和可配置，它们可以在不同的应用程序中提供一致的显示。在 Cocoa 中，Application Kit 定义了大量的视图对象，其中很多对象都出现在 Interface Builder 的选盘上。您可以通过重用 Application Kit 的视图对象，比如 NSButton 对象，来保证应用程序中的按键和其它 Cocoa 应用程序的按键行为是一样的，从而保证不同的应用程序在外观和行为上具有高度的一致性。

视图必须正确地显示模型，因此需要知道模型发生的改变。由于模型对象不应该依赖于特定的视图对象，所以需要有一个一般性的方式来指示模型对象发生了变化。

控制器对象连接模型和视图

控制器对象是应用程序的视图对象和模型对象之间的协调者。通常情况下，它们负责保证视图可以访问其显示的模型，并充当交流的管道，使视图可以了解模型发生的变化。控制器对象也可以为应用程序执行配置和协调的任务，管理其它对象的生命周期。

在一个典型的 Cocoa MVC 设计中，当用户通过某个视图对象输入一个值或做出一个选择时，该值或选择会传递给控制器对象。控制器对象可能以应用程序特有的方式对用户输入进行解释，然后或者告诉模型对象如何处理这个输入—比如“增加一个新值”或“删除当前记录”，或者使模型对象在其某个属性上反应被改变的值。基于同样的用户输入，一些控制器对象也可以通知相应的视图对象改变其外观或行为的某个部分，比如禁用某个按键。反过来，当一个模型对象发生变化了—比如加入一个新的数据源—模型对象通常将变化通知控制器对象，由控制器对象要求一或多个视图对象进行相应的更新。

控制器对象可能是可重用的，也可能是不可重用的，取决于它们的一般类型。"Cocoa控制器对象的类型" 部分描述Cocoa中不同类型的控制器对象。

组合角色

我们可以将多个 MVC 角色组合起来，使一个对象同时充当多个角色，比如同时充当控制器和视图对象的角色—在这种情况下，该对象被称为视图-控制器。同样地，您也可以有模型-控制器对象。对于某些应用程序，象这样的角色组合是可接收的设计。

模型-控制器是主要关注模型层的控制器。它“拥有”模型，主要责任是管理模型，并和视图对象进行交流。应用到整个模型的动作方法通常在模型-控制器中实现。文档架构为您提供了一些这样的方法；比如说，NSDocument 对象（文档架构的核心部分）会自动处理和保存文件相关的动作方法。

视图控制器是主要关注视图层的控制器。它“拥有”界面（视图），主要责任是管理界面，并和模型对象进行交流。和视图显示的数据相关的动作方法通常在视图控制器中实现。NSWindowController 对象（也是文档架构的核心部分）就是视图控制器的一个例子。

"MVC应用程序设计指南"中提供一些关于MVC组合角色对象的设计建议。

进一步阅读：*基于文档的应用程序概述*从另一个角度讨论模型控制器和视图控制器之间的区别。

Cocoa控制器对象的类型

"控制器对象连接模型和视图"部分粗略介绍了控制器对象的抽象框架，但是在实践中的情景要复杂得多。在Cocoa中有两种一般类型的控制器对象：仲裁控制器和协调控制器。每种类型的控制器对象都和一组不同的类相关联，并提供不同的行为。

仲裁控制器通常是从 NSController 类继承而来的对象。在 Cocoa 绑定技术中使用了这种对象。它们负责为视图和模型对象之间的数据流提供仲裁或支持。

仲裁控制器通常都是已经准备好了，可以直接从 Interface Builder 选盘中直接拖出。您可以对这些对象进行配置，以在视图和控制器对象的属性之间、进而在控制器属性和模型对象的具体属性之间建立绑定关系。结果，当用户改变视图对象显示的值时，新的值就会通过仲裁控制器自动传递给模型对象；而当模型的属性值发生变化时，那些变化又会传递给视图对象。NSController 抽象类及其具体子类—NSObjectController、NSArrayController、NSUserDefaultsController、和 NSTreeController—提供了诸如提交和丢弃改变的能力，还可以管理选择和占位值的特性。

协调控制器通常是一个 NSWindowController 或 NSDocumentController 对象，或者是一个 NSObject 定制子类的实例。它在应用程序中的角色是检查（或者协调）整个或部分应用程序是否正常工作，比如从一个 nib 文件解档出来的对象是否有效。协调控制器提供如下服务：

- 响应委托消息和对通告进行观察
- 响应动作消息
- 管理自己“拥有”的对象的生命周期（比如在正确的时间释放那些对象）
- 建立对象间的连接，并执行其它配置任务

NSWindowController 和 NSDocumentController 类是 Cocoa 为基于文档的应用程序定义的架构的一部分。这些类的实例为上面列出的几种服务提供了缺省的实现，您也可以通过创建它们的子类来实现更为具体的应用程序行为，甚至可以用 NSWindowController 对象来管理不基于文档架构的应用程序窗口。

协调控制器通常拥有nib文件中的对象，比如File's Owner对象，它不属于nib文件包含的对象，但负责管理nib文件中的对象。它拥有的对象包括仲裁控制器、协调控制器、和视图对象。如果需要进一步了解File's Owner及类似的协调控制器的更多信息，请参见 "MVC是一个复合的设计模式" 部分的内容。

NSObject 的定制子类的实例可能完全适合用作协调控制器。这种类型的控制器对象既有仲裁的功能，也有协调的功能。在仲裁行为方面，它们通过象目标-动作、插座变量、委托、和通告机制来实现视图和模型对象之间的数据移动。它们有可能包含很多"胶水"代码，由于那些代码只用于特定的应用程序，所以它们是应用程序中最不可能被重用的对象。

进一步阅读：如果您需要进一步了解控制器对象作为仲裁者的角色，请参见 "仲裁者"设计模式的信息；如果需要进一步了解Cocoa绑定技术的信息，请参见*Cocoa绑定技术编程主题*。

MVC是一个复合的设计模式

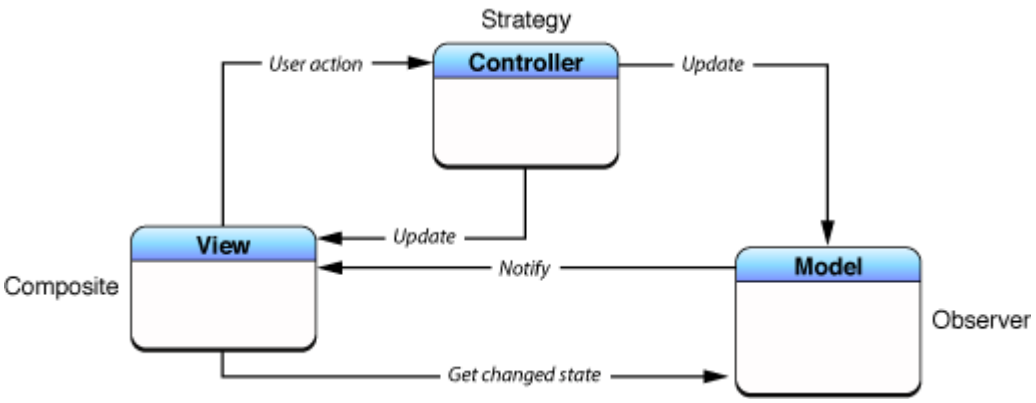
模型-视图-控制器是一个组合了几个更为基本的设计模式的复合设计模式。这些基本的模式一起定义了 MVC 应用程序中特有的功能分割和通信路径。但是和 Cocoa 相比，传统意义上的 MVC 使用了不同的基本模式，主要表现在应用程序的控制器和视图对象的不同作用上。

在原来的（Smalltalk）概念上，MVC 是由合成（Composite）、策略（Strategy）、和观察者（Observer）模式组成的。

- 合成模式：应用程序中的视图对象实际上是一些嵌套视图的集合，这些视图以一种协调过的方式（也就是视图层次结构）在一起工作。这些显示组件包括窗口、复合视图（比如表视图）、以及单独的视图（比如按键）。用户输入和显示可以在复合结构的任意级别上进行。
- 策略模式：一个控制器对象负责实现一或多个视图对象的策略。视图对象将自己限制在视觉效果的维护上，而将与应用程序具体界面行为有关的全部决策委托给控制器。
- 观察者模式：模型对象将状态的变化通知应用程序中感兴趣的对象（通常是视图对象）。

这些模式以图 4-4 所示的方式协同工作：用户操作视图层次中某个级别的视图，结果产生一个事件。控制器对象接收到这个事件，并根据应用程序的具体逻辑对其进行解释－也就是说，它应用了某种策略。这个策略可以是请求（通过消息）模型对象改变其状态，也可以是请求视图对象（位于视图结构的某个级别上）改变其行为或外观。反过来，模型对象在状态发生变化时会通知注册为观察者的所有对象，如果观察者是个视图对象，则可能会因此更新外观。

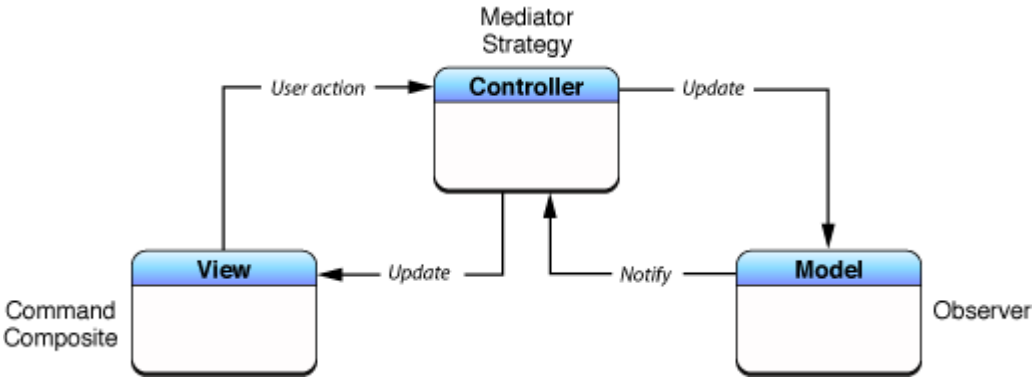
图 4-4 传统版本的 MVC 是一个复合设计模式



Cocoa 版本的 MVC 也是一种复合模式，和传统版本有一些类似之处。事实上，基于图 4-4 的框图构建一个可以工作的应用程序是完全可能的。通过使用绑定技术，您很容易就可以创建一个 Cocoa 的 MVC 程序，让程序中的视图对象直接观察模型对象，以接收状态的改变。然而这个设计有个理论上的问题。视图对象和模型对象应该是程序中最具可重用性的对象。视图对象代表操作系统及操作系统支持的应用程序的“观感”；外观和行为的一致性是很重要的，这就要求对象是高度可重用的。顾名思义，模型对象负责对问题域的关联数据进行封装，以及执行相关的操作。从设计的角度上看，最好让模型对象和视图对象彼此分离，因为这样可以增加它们的可重用性。

在大多数 Cocoa 应用程序中，模型对象的状态变化通告是通过控制器对象传递给视图对象的。图 4-5 显示了这种不同的机制，尽管多用了两个基本设计模式，这种通讯机制显得清晰很多。

图 4-5 Cocoa 版本的 MVC 也是一个复合设计模式



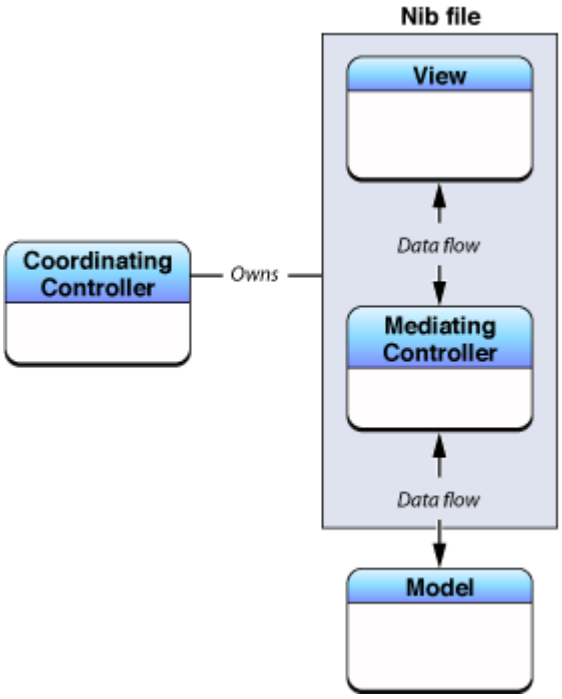
在这种复合模式中，控制器对象结合了仲裁者模式和策略模式，对模型和视图对象之间的数据流实施双向协调。模型状态的变化通过应用程序的控制器对象传递给视图对象。此外，视图对象在目标-动作机制上采纳了命令模式。

请注意：目标-动作机制使视图对象可以和用户输入或选择进行通讯，这种机制可以在协调或仲裁控制器中实现。但是，机制的设计在不同类型的控制器中也有所不同。对于协调控制器，您可以在 Interface Builder 中将视图对象连接到它的目标（即控制器对象）上，并为其指定动作选择器，动作选择器必须遵循特定的签名格式。通过成为窗口和全局应用程序对象的委托，协调控制器也可以进入响应者链。仲裁控制器使用的绑定机制也是将视图对象和目标连接起来，并允许动作方法的签名携带可变数量、任意类型的参数。但是仲裁控制器不在响应者链上。

图 4-5 描述的是改良后的复合设计模式，对其进行改良既有实践上的原因，也有理论上的原因，特别是在使用仲裁者模式的时候。仲裁控制器是从 NSController 的具体子类派生而来的，除了实现仲裁者模式之外，这些类还提供很多应用程序应该加以利用的功能，比如选择和占位值的管理。如果您不喜欢使用绑定技术，则您的视图对象可以使用象 Cocoa 通告中心这样的机制来接收模型对象的通告。但是这要求您创建一个定制的视图子类，以便处理模型对象发出的通告。

在一个设计良好的 Cocoa MVC 程序中，协调控制器对象常常“拥有”归档到 nib 文件的仲裁控制器。图 4-6 显示了这两种控制器类型之间的关系。

图 4-6 协调控制器作为 nib 文件的拥有者



MVC应用程序的设计原则

在设计应用程序的模型-视图-控制器时，可以应用下面这些指导原则：

- 虽然您可以使用 NSObject 定制子类的实例来作为仲裁控制器，但是没有理由重新实现一个仲裁控制器。相反，您可以使用已经准备好的、为 Cocoa 绑定技术设计的 NSController 对象。也就是说，使用 NSObjectController、NSArrayController、NSUserDefaultsController、或者 NSTreeController 实例－或者使用这些 NSController 具体子类的定制子类的实例来作为仲裁控制器。

  然而如果应用程序很简单，而且您对使用插座变量和目标-动作机制来编写仲裁行为所需要的“胶水代码”感觉更好的话，也可以使用 NSObject 定制子类的实例来作为仲裁控制器。在 NSObject 的定制子类中，您也可以以 NSController 的方式来实现仲裁控制器，使用键-值编码、键-值观察、以及编辑器协议。

- 虽然您可以把不同的 MVC 角色合并在一个对象中，但是，在总体上最好的策略还是保持角色的分离。这可以增强对象的可重用性以及使用这些对象的程序的可扩展性。如果您要把不同的 MVC 角色合并到一个类中，则首先为该类选择一个主要的角色，然后（为了便于维护）在相同的实现文件中使用范畴来进行扩展，使其具有其它角色的作用。

- 设计良好的 MVC 应用程序的目标之一应该是尽可能多地使用（至少在理论上）可重用的对象。特别重要的是，视图对象和模型对象应该是高度可重用的（当然，那些准备好的仲裁控制器对象也都是可重用的）。应用程序的具体行为通常尽可能多地集中在控制器对象中。

- 虽然让视图直接观察模型并检测状态的变化是可能的，但是这并不是推荐的做法。视图对象应该总是通过仲裁控制器对象来了解模型对象的变化。这有两层意义：

- 如果您使用绑定机制来使视图对象直接观察模型对象的属性，那么您就忽视了 NSController 及其子类为应用程序提供的各种好处：选择和占位符的管理，还有提交和丢弃修改的能力。
- 如果您不使用绑定机制，则必须从现有的视图类派生出子类，并加入处理模型对象发出的状态变化通告的能力。

- 努力限制应用程序中类代码的依赖关系。一个类对另一个类的依赖越大，就越不具有重用性。具体的推荐规则和两个类在 MVC 中的角色有关：
  - 视图对象不应依赖于模型对象（虽然对于某些定制视图来说可能是不可避免的）。
  - 视图类不必然依赖于仲裁控制器类。
  - 模型类不应该依赖于除了其它模型类之外的类。
  - 协调控制器不应该依赖于模型类（和视图相似，虽然对某些定制的控制器类来说，这种依赖关系是必须的。）
  - 仲裁控制器类不应该依赖于视图类或协调控制器类。
  - 协调控制器类依赖于所有 MVC 类。
- 如果 Cocoa 提供的架构已经将 MVC 角色分配给具体类型的对象，则直接使用该架构。这样做可以使您更为容易地将工程的各个组件集成在一起。文档架构就是这样的一个例子，它包括一个 Xcode 工程模板，并在模板中将 NSDocument 对象（基于 nib 的控制器模型）预先配置为 File's Owner。

Cocoa中的模型-视图-控制器

模型-视图-控制器设计模式使很多 Cocoa 机制和技术的基础。因此，在面向对象的设计中使用 MVC 的重要性已经超过了如何在自己的应用程序中得到更好的可重用性和可扩展性。如果您的应用程序要使用基于 MVC 的 Cocoa 技术，则最好它本身也是遵循 MVC 模式。如果您的应用程序很好地进行 MVC 的分离，在使用这些技术时就应该会相对简单一些；相反，如果没有好的分离，则需要花费更多的努力。

Cocoa 框架中包含下面这些基于模型-视图-控制器的架构、机制、和技术：

- **文档架构。**在这个架构中，一个基于文档的应用程序由一个应用程序级别的控制器对象（NSDocumentController）组成的，每个文档窗口都有一个控制器对象（NSWindowController），每个文档（NSDocument）都有一个结合了控制器和模型角色的对象。
- **绑定技术**。 在之前的讨论中曾经提到过，MVC 是 Cocoa 绑定技术的核心。NSController 抽象类的具体子类提供了一些准备好的控制器对象，您可以对它们进行配置，建立视图对象和模型对象属性之前的绑定关系。
- **应用程序的脚本能力。**在设计应用程序并使其可以支持脚本控制的时候，不仅需要遵循 MVC 设计模式，而且需要正确设计应用程序的模型对象。访问应用程序状态和请求应用程序行为的脚本命令通常应该发送给模型对象或者控制器对象。
- **Core Data**。Core Data 框架负责管理模型对象图，以及将模型对象存储到一个持久的仓库（还有从仓库中取出），以确保这些对象的持久性。Core Data 和 Cocoa 的绑定技术紧密结合在一起。MVC 和对象建模模式是 Core Data 架构的基本决定因素。

- **Undo**。在 Undo 架构中，模型对象又一次发挥中心的作用。模型对象的基元方法（常常是它的存取方法）通常是实现 undo 和 redo 操作的地方。某个动作的视图和控制器对象也可能参与这些操作。举例来说，您可能有一个方法负责处理 undo 和 redo 菜单项的标题，或者 undo 一个文本视图中的选择操作。

进一步阅读："其它Cocoa架构"部分概述了文档架构、应用程序的脚本能力、和Core Data，还包含对这些技术进行详细描述的文档的交叉引用。

对象建模

本部分将定义一些术语，并向您演示一些 Cocoa 绑定技术和 Core Data 框架中用到的对象建模和键-值编码的实例。理解诸如键路径这样的术语对于有效使用这些技术是非常重要的。如果您不熟悉面向对象的设计或键-值编码的话，我们推荐您阅读这个部分。

在使用 Core Data 框架时，您需要一种方法来描述不依赖于视图和控制器的模型对象。在一个具有良好可重用性的设计中，视图和控制器需要一种既能访问模型属性，又不会在它们之间加入依赖关系的方法。Core Data 框架借助了数据库技术的概念和术语解决了这个问题－具体地说是借用了实体-关系模型。

实体-关系建模是一种表现对象的方式，这里的对象通常用于描述数据源的数据结构，使那些数据结构可以被映射为面向对象系统中的对象。请注意，实体-关系建模并不仅仅在 Cocoa 中使用，它和数据库技术中使用的术语和规则一起，是广泛使用的模式。这种对象表示有助于数据源中对象的存储和获取。这里的数据源可能是一个数据库、一个文件、一个 web 服务、或者任何其它的留存仓库。由于它不依赖于任何类型的数据源，所以也可以用于表示任何类型的对象以及对象间的关系。

Cocoa 使用的实体-关系建模在传统的规则上进行了修改，在本文中称为**对象建模**。对象建模在表示模型-视图-控制器（MVC）设计模式中的模型对象时特别有用。这个并不奇怪，因为即使是一个简单的 Cocoa 应用程序，模型通常都是可以留存的－即存储在某些类型的数据容器中，比如文件。

本部分包括如下内容：

实体

在 MVC 设计模式中，模型是应用程序中负责封装特定数据及其操作方法的对象。模型通常是可留存的，但更重要的是，模型并不依赖于数据的显示方式。

在实体-关系模型中，模型被称为**实体**，实体中的组件被称为**属性(attributes)**，对其它模型的引用被称为**关系**。属性和关系合在一起被称为**性质（properties）**。通过这三个简单的组件（实体、属性、和关系），我们可以对任意复杂的系统进行建模。

举例来说，一个对象模型可以被用于描述一个公司的客户数据库、一个图书馆中的图书、或者一个网络中的计算机。图书馆中的图书有自己的属性－比如书的标题、ISBN 号、和版权日期－以及于其它对象的关系－比如作者和图书馆的成员。理论上，如果系统的各个部分都可以被标识出来，则整个系统就可以被表示为一个对象模型。

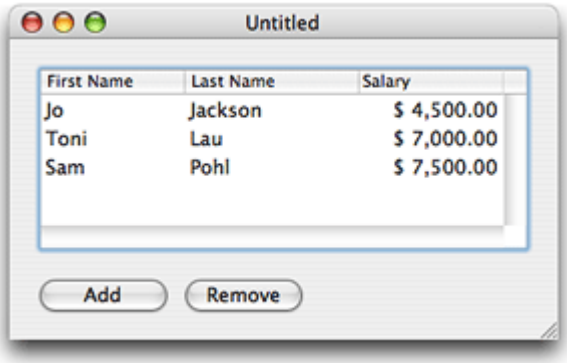
图 4-7 显示的是雇员管理程序中用到的对象模型实例。在这个模型中，Department 表示一个部门的模型，Employee 则代表一个雇员的模型。

图 4-7 雇员管理程序的对象图

属性

属性表示包含数据的结构。对象的属性可能是一个简单的值，比如一个数量（象整型数、浮点数、或者双精度数），但也可以是一个C的结构（比如一个char或者NSPoint的数组），或者一个基元类的实例（比如Cocoa中的NSNumber、NSData、或者NSColor）。象NSColor这样的不可变对象通常也考虑为属性（请注意，Core Data只内建支持一个特定的属性类型集合，具体请见"NSAttributeDescription"部分的描述。但是您可以使用其它的属性类型，具体请见 *Core Data编程指南*中的"非标准属性"部分）。在 Cocoa 中，一个属性通常对应于一个模型的实例变量或存取方法。举例来说，一个 Employee（雇员）有 firstName（名）、lastName（姓）、和 salary（工资）这些实例变量。在一个雇员管理程序中，您可能实现一个表视图来显示一个 Employee 对象及其某些属性的集合，如图 4-8 所示。表格的每一行对应一个 Employee 实例，每一列则对应 Employee 的一个属性。

图 4-8 Employees 的表视图



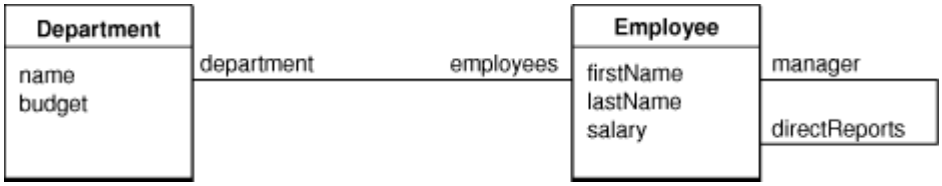
关系

不是所有模型的性质都是属性－一些性质表示与其它对象之间的**关系**。您的应用程序模型通常是由多个类组成的。在运行时，您的对象模型是由彼此关联的对象组成的集合，这些对象构成一个**对象图**。它们通常是一些持久的对象，用户创建这些对象，并在终止应用程序之前（象基于文档的应用程序那样）将它们存储到特定的数据容器或文件中。在运行时，应用程序可以遍历这些模型对象之间的关系，以便访问关联对象的属性。

举例来说，在雇员管理程序中，雇员和其所在的部门之间存在关系，雇员和雇员的经理之间也存在关系。后者是**反身关系**（reflexive relationship，即从一个实体到它自身的关系）的一个实例。

关系本质上是双向的，因此至少在概念上，部门和下面的雇员之间、雇员和其直接汇报的经理之间也存在关系。图 4-9 显示了Department（部门）实体和Employee（雇员）实体之间的关系，以及Employee的反身关系。在这个例子中，Department实体的“employees”关系是Employee实体的“department” 关系的**反面**。然而，某些关系可能只需要按一个方向进行访问－这样就没有反向的关系。假定您永远不需要通过部门对象找出与其相关联的雇员信息，就不需要对该关系进行建模（虽然在一般情况下，Core Data可能对一个的Cocoa对象强加一些额外的约束－不对反向关系进行建模应该是一个非常高级的选项）。

图 4-9 雇员管理系统中的关系



关系的数和所有权

每个关系都有一个**数（cardinality）**：关系的数告诉您有多少目的对象可以（潜在地）确定该关系。如果一个关系只有一个目的对象，则成为 **to-one 关系**。如果一个关系有多个目的对象，则称为 **to-many 关系**。

关系可以是**强制的**，也可以是**可选的**。强制的关系是指必须指定关系的目的，比如说，每个雇员都必须有一个部门。可选的关系则不同，顾名思义，这种关系是可选的－比如说，不是每个雇员都有直接汇报的经理。

为数指定一个范围也是可能的。可选的 to-one 关系的范围是 0-1。一个雇员的直接汇报经理的个数可能是不确定的，或者说是一个指定最小值和最大值的范围，比如 0-15，这也是一个可选的 to-many 关系的例证。

图 4-10 说明了雇员管理程序中的数。Employee 对象 Department 对象之前的关系是一个强制的 to-one 关系—一个雇员必须属于且仅属于一个部门。Department 对象和部门中的 Employee 对象是一个可选的 to-many 关系（用一个“*”来表示）。雇员及其经理的关系是一个可选的 to-one 关系（由范围 0-1 来表示）—最高级的雇员没有经理。

图 4-10 关系的数

也请注意，关系的目的对象有些时候是被拥有的，有些时候则是共享的。

访问性质

为了使模型、视图、控制器之间彼此独立，您必须能够以一种独立于模型具体实现的方式来访问性质。这可以通过键-值对来完成。

键

模型的性质是通过一个简单的**键**（通常是个字符串）来指定的。视图和控制器通过键来查找相应的属性**值**。“为属性提供值”的说法强化了这样的概念：即属性自身并不一定包含数据—它的值可以间接得到。

键-值编码技术用于进行这样的查找—它是一种间接访问（在特定的上下文中可以是自动访问）对象属性的机制。键-值编码的工作机制是用对象性质的名称—通常是其实例变量或存取方法—作为键来访问那些性质的值。

举例来说，您可以通过 name 键来取得一个 Department 对象的名称。如果 Department 对象有一个实例变量或方法的名称叫 name，则该键的值就会被返回（如果没有这样的变量或方法则回导致错误）。类似地，您可以通过 firstName、lastName、和 salary 这些键来取得 Employee 对象的属性。

值

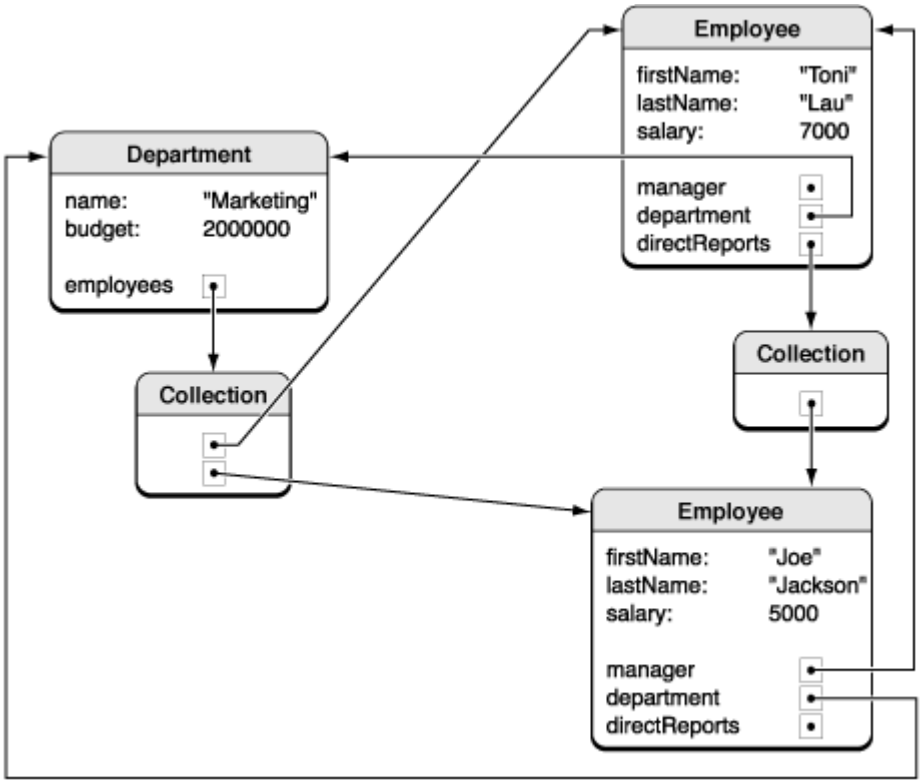
在一个给定的实体中，同一个属性的所有值具有相同的数据类型。属性的数据类型取决于相应的实例变量或存取方法返回值的声明。举例来说，Department 对象中 name 属性的数据类型可能是一个 Objective-C 的 NSString 对象。

请注意，键-值编码只返回对象值。如果为特定的键提供值的存取方法或实例变量的返回值类型或数据类型不是一个对象，则 Cocoa 会为该值创建一个 NSNumber 或 NSValue 对象，并将它返回。如果 Department 的 name 属性为 NSString 类型，则通过键-值编码取得的 Department 对象的 name 键的值就是一个 NSString 对象。如果 Department 的 budget 属性是 float 类型，则通过键-值编码取得的 Department 对象的 budget 键的值就是一个 NSNumber 对象。

类似地，当您通过键-值编码技术进行值的设置时，如果与指定键对应的存取方法或实例变量要求的数据类型不是对象，则 Cocoa 会通过正确的-<type>Value 方法将值从传递过来的对象中抽出来。

to-one 关系的值比较简单，就是该关系的目的对象。举例来说，Employee 对象的 department 属性值就是一个 Department 对象。to-many 关系的值是一个集合对象（可能是一个集合或一个数组—如果您使用 Core Data，则是个集合，否则通常是个数组），集合或数组中包含该关系的目的对象。举例来说，Department 对象的 employees 属性是一个包含 Employee 对象的集合。图 4-11 展示了雇员管理程序的对象图实例。

图 4-11 雇员管理程序的对象图

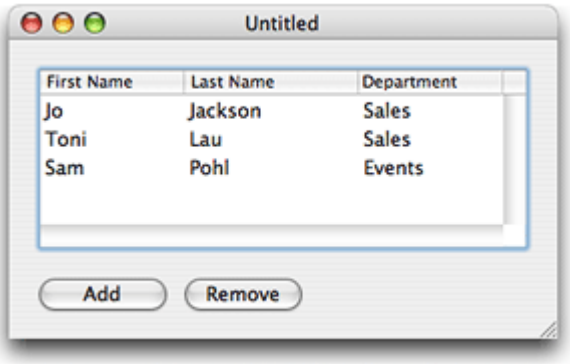


键路径

键路径是一个由用点作分隔符的键组成的字符串，用于指定一个连接在一起的对象性质序列。第一个键的性质是由先前的性质决定的，接下来每个键的值也是相对于其前面的性质。键路径使您可以以独立于模型实现的方式指定相关对象的性质。通过键路径，您可以指定对象图中的一个任意深度的路径，使其指向相关对象的特定属性。

键-值编码机制实现了给定键路径的查找，类似于键-值对。举例来说，在一个账户处理程序中，您可以通过名为 department.name 的键路径，从 Employee 对象访问 Department 的名称。在该路径中，department 是 Employee 的关系，name 是 Department 的属性。如果您希望显示一个目的实体的属性，则键路径非常有用。举例来说，在图 4-12 中，雇员表视图被配置为显示雇员所在部门的名称，而不是部门对象本身。通过 Cocoa 的绑定技术，“Department”列的值被绑定到列表中的雇员对象的 department.name 路径上。

图 4-12 在 Employees 表视图上显示部门名称



并不是键路径上的每个关系都必须有个值。如果雇员是 CEO，则其 manager 关系可能为 nil。在这种情况下，键-值编码机制并不会出现问题—它只是简单地停止路径的遍历，并返回合适的值，比如 nil。

和对象进行通讯

Cocoa 借助几个设计模式来实现应用程序中对象间的通讯机制。这些机制和范式包括委托、通告、目标-动作，和绑定技术。本章将对这些机制和范式进行描述。

本部分包括如下内容：

面向对象程序中的通讯

通过 Cocoa 及其使用的面向对象编程语言（Objective-C 和 Java）为程序加入具体行为的一种方式是继承。您可以为现有的类创建一个子类，然后为该类的实例增加属性或行为，或者以某种方式对其进行修改。但是，还有一些为程序添加特有逻辑的其它方式，以及一些重用和扩展 Cocoa 对象能力的其它机制。

在一个程序中，对象之间的关系不止存在于一个维中。除了继承层次结构关系，程序中的对象还动态地存在于一个网络中。在运行时，网络中的对象必须进行通讯，以完成程序的工作。和管弦乐队中的音乐家之间的方式类似，程序中的每个对象都有一个角色，即对象为整个程序实现的一个有限行为集合。角色可以显示一个可以响应鼠标点击的椭圆形表面，或者管理一个对象的集合，或者协调窗口生命周期中的主要事件。它只做设计时规定的工作，不做别的。但是为了在程序中发挥作用，它必须能够和其它对象进行通讯，比如能够向其它对象发送消息，或者接收其它对象的消息。

在您的对象可以向其它对象发送消息之前，必须拥有对其它对象的引用，或者可以依赖于某种分发机制。Cocoa提供了很多对象间通讯的方式。这些机制和技术基于 "Cocoa设计模式"部分中描述的设计模式，使我们可以高效地构建强壮的应用程序。它们中有简单的，也有稍微复杂一些的，通常是比子类化更好的选择。您可以通过编程的方式对其进行配置，有时也可以通过Interface Builder进行图形化的配置。

插座变量

对象的合成是一种动态的模式，要求对象设法得到其委托者的引用，以便向它们发送消息。它通常以实例变量的方式保有其它对象。这些变量必须在程序执行的某些点上，用正确的引用进行初始化。

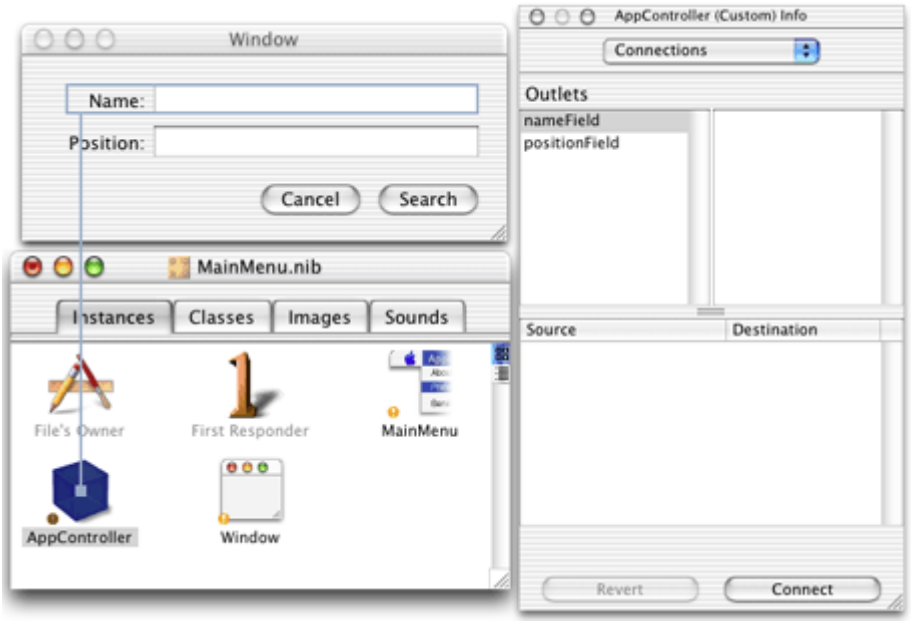
插座变量就是这样的一种对象实例变量，它的特别之处在于，其对象的引用是由 Interface Builder 来配置和归档的。每次包含对象从 nib 文件解档时，它与插座变量之间的连接都需要重新建立。包含对象以实例变量的方式保有插座变量，其类型限定符为 IBOutlet。例如：

```
@interface AppController : NSObject

{

    IBOutlet NSArray *keywords;

}
```

插座变量是一个实例变量，因此也成为对象封装数据的一部分。但是插座变量不仅仅是个简单的实例变量。对象与其插座变量之间的连接会被归档到 nib 文件中，在 nib 文件被装载的时候，每个连接都会被解档和保持，因此在需要向其它对象发送消息时，插座对象总是可用的。

类型限定符 IBOutlet 是一个标签，用于实例变量的声明。通过这个限定符，Interface Builder 程序在开发过程中可以和 Xcode 同步插座变量的显示和连接。换句话说，您可以为某个定制对象添加插座变量并建立连接，然后生成带有这个插座变量的头文件。或者，您可以在 Xcode 中声明插座变量（使用 IBOutlet 限定符），Interface Builder 就能识别这些新的声明，使您可以建立连接，并将连接存储到 nib 文件中。图 5-1 显示了如何在 Interface Builder 中连接插座变量。

图 5-1 在 Interface Builder 中连接插座变量



应用程序通常在其定制的控制器对象和用户界面对象之间设置插座变量连接，但是这种连接可以在 Interface Builders 中代表实例的任何对象之间，甚至使两个定制对象之间建立。和其它实例变量一样，您应该可以判断在类中包含插座变量的正当性；一个对象包含的实例变量越多，内存开销就越大。如果有其

它方式可以得到对象的引用，比如通过其在矩阵中的索引位置进行查找，或者将对象作为函数的参数进行传递，或者通过使用标签（一个分配好的数字标识），则您应该使用其它的方法。

委托和数据源

委托是一种对象，当向外委托任务的对象遇到程序中的事件时，它的委托可以代表它对事件进行处理，或者和它进行协调。向外委托任务的对象通常是一个响应者对象—即继承自 NSResponder 的对象—负责响应用户事件。委托则是受托进行事件的用户界面控制，或者至少根据应用程序的具体需要对事件进行解释的对象。

为了更好地理解委托的价值，让我们考虑一个复活的 Cocoa 对象，比如一个窗口（NSWindow 的实例）或者表视图（NSTableView 的实例）。这些对象的设计目的是以一般的方式实现一个具体的角色；举例来说，窗口对象负责响应窗口控件的鼠标操作，处理象关闭窗口、调整尺寸、以及移动窗口的位置这样的事件；这个受限而又具有一般性的行为必然限制该对象认识一个事件对应用程序其它地方的影响，特别是当被影响的行为只存在于您的应用程序的时候。委托为您的定制对象提供一种方法，使它可以就应用程序特有的行为和复活对象进行通讯。

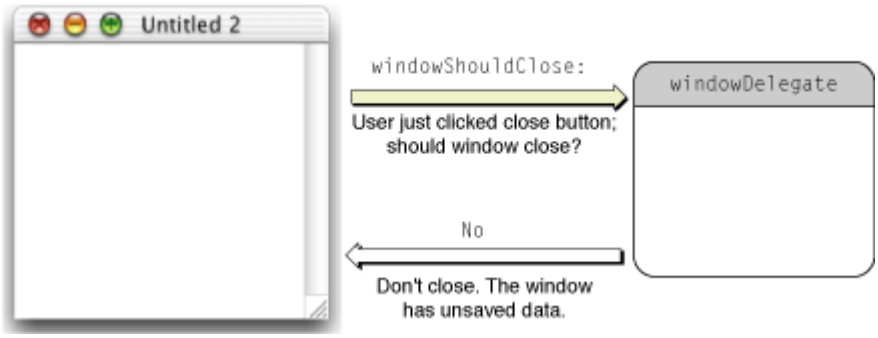
委托的编程机制使对象有机会对自己的外观和状态、以及程序在其它地方发生的变化进行协调，这些变化通常是由用户动作触发的。更重要的是，委托使一个对象有可能在没有进行继承的情况下改变另一个对象的行为。委托几乎总是您的一个定制对象，它通过定义将应用程序具体逻辑结合到程序中，而这些逻辑是具有一般性的，是向外委托任务的对象自身不可能知道的。

本部分包括如下主要内容：

委托是如何工作的

委托机制的设计是很简单的（图 5-2）。希望向外委托任务的类需要有一个插座变量，通常命名为 delegate，并包含对该插座变量进行设置和访问的方法。它还需要声明一或多个方法，构成一个非正式的协议，但不进行实现。非正式协议通常是希望向外委托任务的类的一个范畴，和正式协议的不同之处在于，它不需要实现协议中的所有方法。在非正式协议中，委托只实现希望进行协调或对缺省行为实施影响的方法。

图 5-2 委托的机制



非正式协议的方法标记着进行任务委托的对象需要处理或预期发生的重大事件。该对象希望就这些事件和委托进行交流，或者就即将发生的事件向委托请求输入或批准。举例来说，当用户点击一个窗口的关闭按键时，窗口对象会向委托发送 windowShouldClose: 消息；这就使委托有机会否决或推迟窗口的关闭，如果必须保存窗口关联数据的话（参见图 5-3）。

图 5-3 一个更接近现实的、涉及委托的序列

向外委托任务的对象只将消息发送给实现了相应方法的委托。在发送消息之前，它会首先对委托调用 NSObject 的 respondsToSelector: 方法，确认委托是否实现该方法。这种预先检查是非正式协议设计的关键。

委托消息的形式

委托方法有个命名约定，即以进行委托的 Application Kit 对象的名字作为开头—如应用程序、窗口、控件等，名字是小写的，且没有“NS”前缀；这个对象名后面通常（但并不总是）紧接着一个辅助的动词，指示被报告的事件在时间上的状态，换句话说，这个动词指示事件是即将发生（“Should”或者“Will”），还是刚刚发生（“Did”或者“Has”）。这个时间上的区别可以帮助我们区分那些希望得到一个返回值和不需要返回值的消息。程序清单 5-1 列出了期望得到返回值的 Application Kit 委托方法。

程序清单 5-1 带有返回值的委托方法示例

```
- (BOOL)application:(NSApplication *)sender openFile:(NSString *)filename;

- (BOOL)textShouldBeginEditing:(NSText *)textObject;

-         (NSApplicationTerminateReply)applicationShouldTerminate:(NSApplication
```

```
*)sender;

-              (NSRect)windowWillUseStandardFrame:(NSWindow                *)window
defaultFrame:(NSRect)newFrame;
```

实现这些方法的委托可以阻塞（前面的两个方法可以通过返回 NO 来实现）、延迟 (通过在 applicationShouldTerminate: 方法中返回 NSTerminateLater）即将发生的事件，或者改变 Cocoa 建议的参数（比如最后一个方法中的外形方框）。

另外一类委托方法是由不期望返回值的消息调用的，因此返回类型为 void。这些消息纯粹是信息性的，且方法的名称通常包含“Did” 或其它指示事件已经发生的词。程序清单 5-2 列出了一些这类委托方法的实例。

程序清单 5-2 返回 void 的委托方法示例

```
-                    (void)                    tableView:(NSTableView*)tableView
mouseDownInHeaderOfTableColumn:(NSTableColumn *)tableColumn;

- (void)applicationDidUnhide:(NSNotification *)notification;

- (void)applicationWillBecomeActive:(NSNotification *)notification;

- (void)windowDidMove:(NSNotification *)notification;
```

有一些和最后一组消息有关的事项需要注意。第一是辅助动词“Will” (如第三个方法）并不一定意味着需要返回值。在这种情况下，事件即将发生且不能被阻塞，而这个消息使委托有机会为事件做好准备。

另一个有意思的点是关于最后三个方法的。这三个方法都只有一个参数，即一个NSNotification对象，意味着调用这些方法是特定的通告发出的结果。举例来说，windowDidMove:方法和NSWindow的NSWindowDidMoveNotification方法是相互关联的。"通告"部分对此进行详细的讨论。但是在这里，理解通告和委托消息的关系是很重要的。向外委托的对象自动使自己的委托成为自己发出的所有通告的观察者。委托需要做的是实现相关联的方法，以获取通告。

为了使您的定制类成为 Application Kit 对象的委托，只要简单地在 Interface Builder 中将实例和 delegate 插座变量相连接就可以了。或者，也可以在程序中调用向外委托的对象的 setDelegate: 方法来进行设置，这样的设置最好早做，比如在 awakeFromNib 方法中进行。

委托和Application Kit

应用程序中向外委托的对象通常是一个 NSApplication、NSWindow、或者 NSView 对象。委托在典型情况下是个对象，但也不是一定如此。它通常是一个定制对象，负责控制应用程序的一部分（也就是说，是协调控制器对象）。表 5-1 列出了定义委托的 Application Kit 类。

表 5-1 带有委托的 Application Kit 类

NSApplication	NSFontManager	NSSplitView	NSTextField
NSBrowser	NSFontPanel	NSTableView	NSTextView
NSControl	NSMatrix	NSTabView	NSWindow

NSDrawer	NSOutlineView	NSText	

向外委托的对象并不（且不应该）保持自己的委托。但是，它的客户对象（通常是应用程序）需要负责保证它们的委托可以接收委托消息。为此，它们可能需要对委托进行保持。这个警示同样适用于数据源、通告的观察者，以及动作消息的目标。

某些 Application Kit 类有一种更为严格的委托，称为模式委托。这些类的对象（比如 NSOpenPanel）会弹出模式对话框，当用户点击对话框的 OK 按键时，指定委托中的处理函数就会被调用。模式委托的应用限制在模式对话框操作的范围内。

委托的存在有一些其它的编程用法。举例来说，通过委托，一个程序中的两个协调控制器可以很容易地找到彼此，并进行通讯。比如，控制整个应用程序的对象可以通过类似如下的代码找到应用程序的查看器窗口（假定它时当前的关键窗口）：

```
id winController = [[NSApp keyWindow] delegate];
```

您的代码也能通过执行下面的代码找到应用程序控制器对象—它定义为全局应用程序实例的委托：

```
id appController = [NSApp delegate];
```

数据源

数据源很像委托，区别在于委托处理的是用户界面的控制，而数据源处理的是数据的控制。数据源是由 NSView 对象保有的插座变量，举例来说，表视图和大纲视图都需要一个提供可视数据的源。视图的数据源和委托通常是同一个对象，但也可以是其它对象。和委托对象一样，数据源必须实现某个非正式协议中的一个或多个方法，以便为视图提供其所需要的数据，在更高级的实现中，它还可以处理用户在视图中直接编辑的数据。

和委托一样，数据源对象必须可以接收由请求数据的对象发出的消息。使用数据源对象的应用程序必须确保该对象的持久性，在必要的时候对其进行保持操作。

数据源负责保证它们分发给用户界面对象的数据对象的持久性。换句话说，它们负责那些对象的内存管理。然而，每当视图对象（比如大纲视图或表视图）对数据源的数据进行访问时，会在需要使用该数据的时候对其进行保持。但是它们并不使用很长时间，通常只是足以对该数据进行显示就可以了。

实现一个定制类的委托

通过下面的步骤可以为您的定制类实现一个委托：

- 在您的类头文件中声明一个委托存取方法：

```
- (id)delegate;

- (void)setDelegate:(id)newDelegate;
```

- 实现该存取方法。请注意，在 setter 方法中，您应该仅拥有委托的弱引用，避免循环保持（也就是说，不要对委托进行保持或拷贝）：

```
- (id)delegate {
```

```
    return delegate;
}


- (void)setDelegate:(id)newDelegate {
    delegate = newDelegate;
}
```

- 声明一个包含委托编程接口的非正式协议。非正式协议是 NSObject 类的范畴。

```
@interface NSObject (MyObjectDelegateMethod)
- (BOOL)operationShouldProceed;
@end
```

- 请参见 "委托消息的格式"，该部分给出了对您自己的委托方法进行命名的建议。
- 在调用一个委托方法时，请向委托发送 respondsToSelector:消息，确认其是否实现该方法。

```
- (void)someMethod {
    if ( [delegate respondsToSelector:@selector(operationShouldProceed)] ) {
        if ( [delegate operationShouldProceed] ) {
            // do something appropriate
        }
    }
}
```

目标-动作机制

委托、绑定、和通告机制在处理程序中特定形式的对象间通讯是很有用的。但是，对于大多数可视的通讯形式，它们并不特别合适。一个典型应用程序的用户界面是由一些图形对象组成的，最常见的对象可能就是控件。控件是真实世界或逻辑设备的对等物（比如按键，滑块，检查框等）。和真实世界中的控制接口（比如收音机调谐器）一样，您可以用控件来传达您对其所在系统的控制意图－在这里，系统是一个应用程序。

用户界面中的控件的作用很简单：它对用户的意图进行解释，并指示其它对象执行相应的请求。当用户对控件进行动作，比如点击控件或按下回车键，硬件设备就会产生一个未经加工的事件。控件接受该事件（在根据 Cocoa 的需要进行恰当的封装之后），并将它翻译为应用程序的具体指令。然而，事件的本身并没有给出很多关于用户意图的信息，它们只是告诉您用户点击了鼠标键，或者按下了一个按键。因此，程序需要某些机制来进行事件和指令的翻译。这个机制就是目标-动作机制。

Cocoa 通过目标-动作机制来实现控件和对象的通讯。这个机制使控件和它的单元（可能有多个）可以把向恰当对象发送应用程序具体指令所需要的信息封装起来。接受对象－通常是个定制类的实例－被称为目标（target）。动作是控件发送给目标的消息（action）。对用户事件感兴趣的对象－即目标－也就是为用户事件给出意义的对象，这个意义通常在动作的名称中反应出来。

本部分包含如下内容：

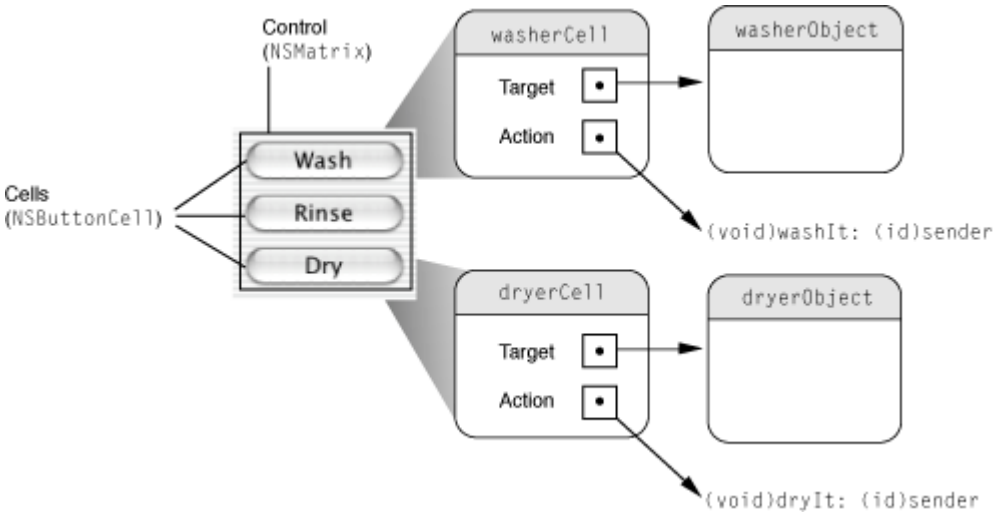
控件、单元、和菜单项

大多数控件和对象都是从 NSControl 类继承下来的。虽然控件最初的任务是将动作消息发送给目标，但它很少携带发送消息需要的信息，因此通常需要依赖于它的单元。

一个控件几乎总是有一个或多个与之相关联的单元－也就是从NSCell类继承下来的对象。为什么需要有这种关联呢？控件是相对“重”的对象，因为它继承了其祖先类，包括NSView和NSResponder类的所有实例变量。由于控件开销比较大，我们就通过单元将其在屏幕上的空间划分成不同的功能区域。单元是轻量级的对象，我们可以将它考虑为覆盖全部或部分控件的区域。但它不仅是区域的分割，还是对任务的分割。单元负责一些本来由控件描画的工作，而且保有一些本来由控件保有的数据，其中的两项就是目标和动作的实例变量。图 5-4 描述了控件-单元的机制。
作为抽象类，NSControl 和 NSCell 都没有完全处理目标和动作实例变量的设定。缺省情况下，NSControl 只是简单地将信息设置到其关联的单元中，如果有关联单元存在的话（NSControl 只支持其自身和一个单元之间的一对一的映射关系；NSControl 的某些子类，比如 NSMatrix，支持多个单元）。而 NSCell 在其缺省实现中只是简单地抛出一个例外。您必须沿着继承链进一步往下才能找到真正实现了目标和动作设置的类：即 NSActionCell。

从 NSActionCell 派生出来的对象为其控件提供相应的目标和动作值，使控件可以产生一个动作消息，并发送给正确的接收者。NSActionCell 对象通过强调显示相应的区域来处理鼠标（光标）跟踪，并辅助其对应的控件将动作消息发送给指定的目标。在大多数情况下，NSControl 对象的外观和行为都由相应的 NSActionCell 对象来负责（NSMatrix 及其子类 NSForm 属于不遵循这个规则的 NSControl 子类）。

图 5-4 目标-动作机制是如何工作的



当用户从菜单中选择一个项时，一个动作就会被发送给相应的目标。然而，菜单（NSMenu 对象）及其菜单项（NSMenuItem 对象）在架构上和控件及单元是完全分开的。NSMenuItem 类为自己的实例实现了目标-动作机制；每个 NSMenuItem 对象都有代表目标和动作的实例变量（及相应的存取方法），并在用户选择时将动作消息发送给目标。

请注意："控件和菜单"部分更为详细地讨论了控件-单元和菜单的架构。您还可以参见*Cocoa的控件和单元编程主题*和*Cocoa的应用程序菜单和下拉列表编程主题*部分。

目标

目标是动作消息的接收者。一个控件，或者更为常见的是它的单元，以插座变量（参见 "插座变量"部分）的形式保有其动作消息的目标。虽然目标可以是任何实现恰当的动作方法的Cocoa对象，但通常是您的定制类的一个实例。

您也可以将一个单元或控件的目标插座变量设置为 nil，使目标对象在运行时才确定。当目标为 nil 时，NSApplication 对象会以预先定义好的顺序检索合适的接收者：

1. 以键盘焦点窗口中的第一个响应者开始，然后通过 nextResponder 方法沿着响应者链检索，一直到 NSWindow 对象的内容视图。

   请注意：键盘焦点窗口负责响应应用程序的键盘输入，且是菜单和对话框的消息接收者。应用程序的主窗口是用户动作的主要焦点，通常也是键盘焦点窗口。

2. 接着尝试 NSWindow 对象，然后是窗口对象的委托。

3. 如果主窗口不是键盘焦点窗口，就接着从主窗口的第一个响应者开始，沿着主窗口的响应者链一直到 NSWindow 对象及其委托。

4. 接下来，NSApplication 对象会确认自己是否响应。如果不能响应，就尝试自己的委托。NSApp 及其委托是最后的接收者。

控件对象并不（且不应该）保持它们的目标。然而，发送动作消息的控件的客户要负责保证它们的目标可以接受动作消息。为此，它们必须对目标进行保持。在委托和数据源中，也同样需要注意这一点。

动作

动作是控件发送给目标的消息，或者从目标的角度看，它是目标为了响应动作而实现的方法。控件，或者更为常见的是它的单元，将动作存储为 SEL 类型的实例变量。SEL 是一种 Objective-C 的数据类型，用于指定消息的签名。动作消息必须有一个简单而清楚的签名，消息调用的方法没有返回值，且只有一个类型为 id 的参数。该参数约定的名称为 sender。下面是一个从 NSResponder 类摘出的、定义一些动作方法的实例：

```
- (void)capitalizeWord:(id)sender;
```

动作方法也可以有如下相似的签名形式：

```
- (IBAction) deleteRecord:(id)sender;
```

在这种形式中，IBAction 并不是指定一个返回值的数据类型；这里没有返回值。IBAction 是一个类型限定符，在应用程序开发过程中，Interface Builder 会通过这个限定符来同步以编程方式加入的动作和内部为工程定义的动作方法列表。

sender 参数通常标识发送动作消息的控件（虽然也可以用另一个对象来代替实际的发送者）。这个设计背后的想法类似于明信片上的返回地址。如果需要的话，目标可以向消息的发送者查询更多信息。如果实际的发送对象用另一个对象来代替 sender，您也应该以同样的方式来对待该对象。举例来说，您有一个文本输入框，当用户输入文本时，目标中的 nameEntered: 方法（这里随意进行命名）就会被调用：

```
- (void)nameEntered:(id) sender {

    NSString *name = [sender stringValue];

    if (![name isEqualToString:@""]) {

        NSMutableArray *names = [self nameList];

        [names addObject:name];

        [sender setStringValue:@""];

    }

}
```

这里，响应方法抽出文本输入框的内容；将该字符串加入到数组中，以实例变量的形式进行缓存；然后清空文本输入框。其它可能发给发送者对象的查询包括查询 NSMatrix 对象中选定的行（[sender selectedRow]），查询 NSButton 对象的状态（[sender state]），以及查询与某个控件相关的单元的标签（[[sender cell] tag]），这种标签可能是任意的标识符。

Application Kit定义的动作

Application Kit 不仅包含很多 NSActionCell-它通过控件来发送动作消息，还在很多类中定义了动作方法。当您创建 Cocoa 应用程序工程时，一些动作就被连接到缺省的目标上。举例来说，应用程序菜单的 Quit 命令会被连接到全局应用程序对象（NSApp）的 terminate: 方法上。

NSResponder 类也为公共的文本操作定义了很多缺省的动作信息（也称为标准命令）。这也使得 Cocoa 文本系统可以在应用程序的响应者链中发送这些动作消息。响应者链是一个具有一定层次结构的事件处理

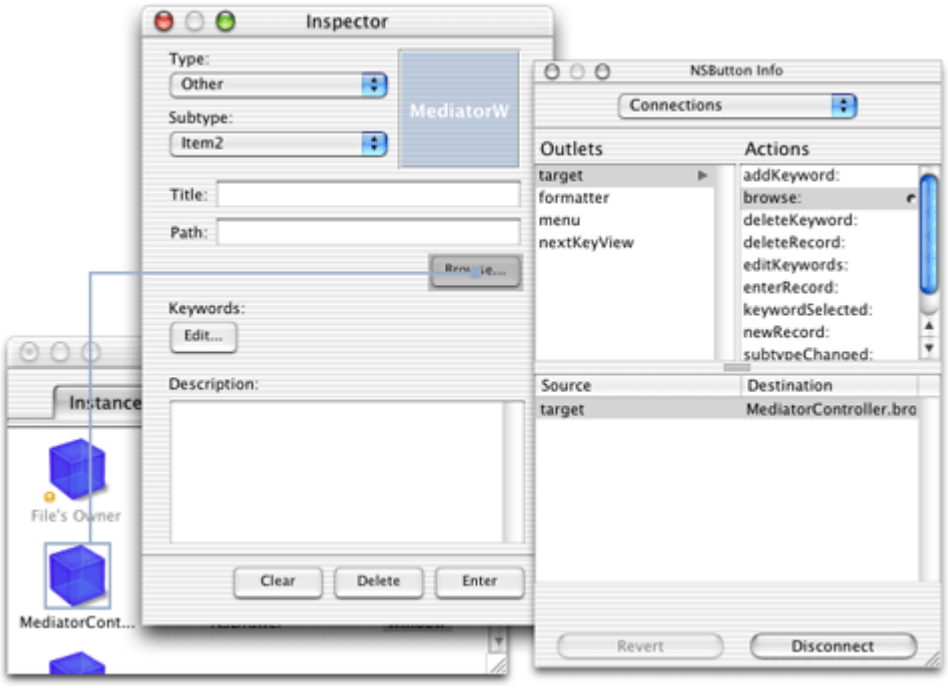
对象序列，这些动作消息会被响应者链中第一个实现对应的处理方法的 NSView、NSWindow、或者 NSApplication 对象。

设置目标和动作

您可以在程序中设置单元和控件的目标和动作，也可以在 Interface Builder 中进行。对于大多数的开发者和大多数的使用场合，Interface Builder 是更好的方法。当您使用 Interface Builder 设置控件和目标时，它可以提供视觉上的确认，并且可以支持锁定连接，以及将连接归档到 nib 文件中。用 Interface Builder 进行设置的流程比较简单，在开发工具文档中有更详细的讨论：

1. 在您的定制类中定义动作方法。
2. 为您的定制类生成一个实例（如果这种实例尚未存在的话）。
3. 按住 Control 键，同时拖拽出一条*从*控件或单元对象*到*代表定制类实例的连接线。
4. 在查看器窗口（Interface Builder 会自动显示该窗口）的 Connections 面板中选择动作方法，并点击 Connect 键。

图 5-5 在 Interface Builder 中设置目标和动作



如果动作是由您的定制类的超类或复活的 Application Kit 类处理，则可以跳过第一步。当然，如果您自行定义了动作方法，就必须确保提供相应的实现。

如果要以编程的方式设置动作及其目标，可以通过下面的方法来将消息发送给控件或单元对象：

```
- (void)setTarget:(id)anObject;

- (void)setAction:(SEL)aSelector;
```

下面的实例显示了这些方法的可能用法：

```
[aCell setTarget:myController];

[aControl setAction:@selector(deleteRecord:)];

[aMenuItem setAction:@selector(showGuides:)];
```

在程序中设置目标和动作确实有自己的优点，在某些情况下，它还是唯一可行的设置方法。例如，您可能希望根据某些运行时条件，比如网络连接是否存在或查看器窗口是否被装载，来改变目标和动作。再比如，当您动态确定弹出式菜单的菜单项，并希望每个弹出式菜单都有自己的动作时。

绑定

绑定（Bindings）是一种Cocoa技术，用于同步应用程序中数据的显示和存储，是Cocoa工具箱中激活对象间通讯的重要工具。这种技术和模型-视图-控制器及对象建模两种设计模式相适配（"模型-视图-控制器设计模式"部分在对控制器对象的讨论中介绍了绑定技术）。它使您可以在显示某个值的视图对象属性和存储该值的模型对象属性之间建立起一个仲裁连接；当连接一端的值发生变化时，另一端可以自动反应该变化。负责连接仲裁的控制器对象会提供其它支持，包括选择的管理，占位符的值，以及支持排序的表格。

本部分包括如下内容：

绑定是如何工作的

绑定借用了模型-视图-控制器（MVC）和对象建模两种设计模式定义的概念空间。MVC 程序赋以各个对象一般性的角色，并根据这些角色区分不同的对象。对象可以是视图对象，模型对象，或者控制器对象，它们的角色可以简要描述如下：

- 视图对象显示应用程序的数据。
- 模型对象负责封装和操作应用程序数据。它们通常是一些留存对象，用户在应用程序运行时对其进行创建和保存。
- 控制器对象负责协调视图和模型对象之间的数据交换，同时还为应用程序执行“命令和控制”服务。

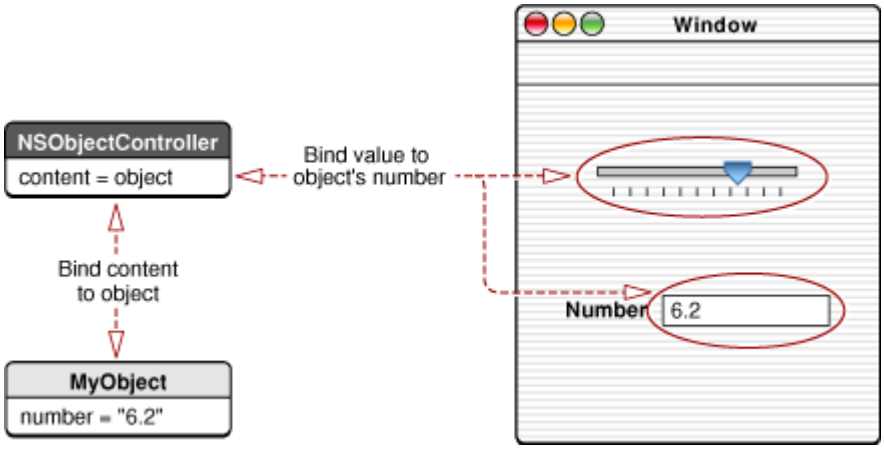
所有的对象都可以定义称为性质的组件或特征，但最重要的是模型对象。有两种类型的性质：属性－指像字符串、数量、和数据结构，和与其它对象之间的关系。关系又有两种类型：一对一和一对多，它们可以是双向或者反身的。因此，应用程序中的对象之间具有不同类型的关系，这个对象的网络称为对象图。性质有一个标识名，称为键。通过键路径－也就是以句点分割的键序列，人们可以遍历对象图中的关系，进而访问相关对象的属性。

绑定技术使用这个对象模型来建立应用程序中的视图、模型、和控制器对象之间的绑定关系。通过绑定，您可以将关系的网络从模型对象的对象图扩展到应用程序中的控制器和视图对象。您可以在视图对象的属性和模型对象的性质之间建立绑定（通常是通过控制器对象的仲裁性质）。界面上显示的属性值的任何变化都可以自动地通过绑定传播到存储该值的性质；该性质的值的任何内部变化，也会被传递到显示视图中。

举例来说，图 5-6 显示了一个简化的绑定集合，该绑定存在于滑块控件的显示值、文本输入框（那些视图对象的属性）、及模型对象（MyObject）的number 属性之间，通过控制器对象的content性质来实

现。在这些绑定关系建立之后，如果用户移动滑块，则滑块的值会被应用到number属性，然后再传回文本框进行显示。

图 5-6 视图、控制器、和模型对象之间的绑定



绑定的实现是基于键-值编码，键-值观察，和键-值绑定的激活机制。这些机制的概述及相关的非正式协议的信息，请参见 "键-值机制"部分。"Cocoa设计模式"中有关观察者模式的讨论部分也对键-值观察进行描述。您可以在任意两个对象之间建立绑定关系。唯一的要求是对象要遵循键-值编码和键-值观察的规则。然而，您通常需要*通过*仲裁控制器来建立绑定，因为这种控制器提供了与绑定有关的服务，比如选择管理，占位符值，还提供提交或抛弃修改的能力。仲裁控制器是几个NSController子类的实例；在Interface Builder的Controllers（控制器）选盘上提供（参见下文中的"如何建立绑定"部分）。您也可以创建定制的仲裁控制器，以获得更多具体的行为。如果需要有关仲裁控制器和NSController对象的讨论，请参见"模型-视图-控制器设计模式"中的 "Cocoa控制器对象的类型" 部分，以及 "Cocoa设计模式"中的仲裁者模式部分。

进一步阅读：如果需要进一步了解上面概括的设计模式，请参见 "模型-视图-控制器设计模式"和 "对象建模"部分。

如何建立绑定

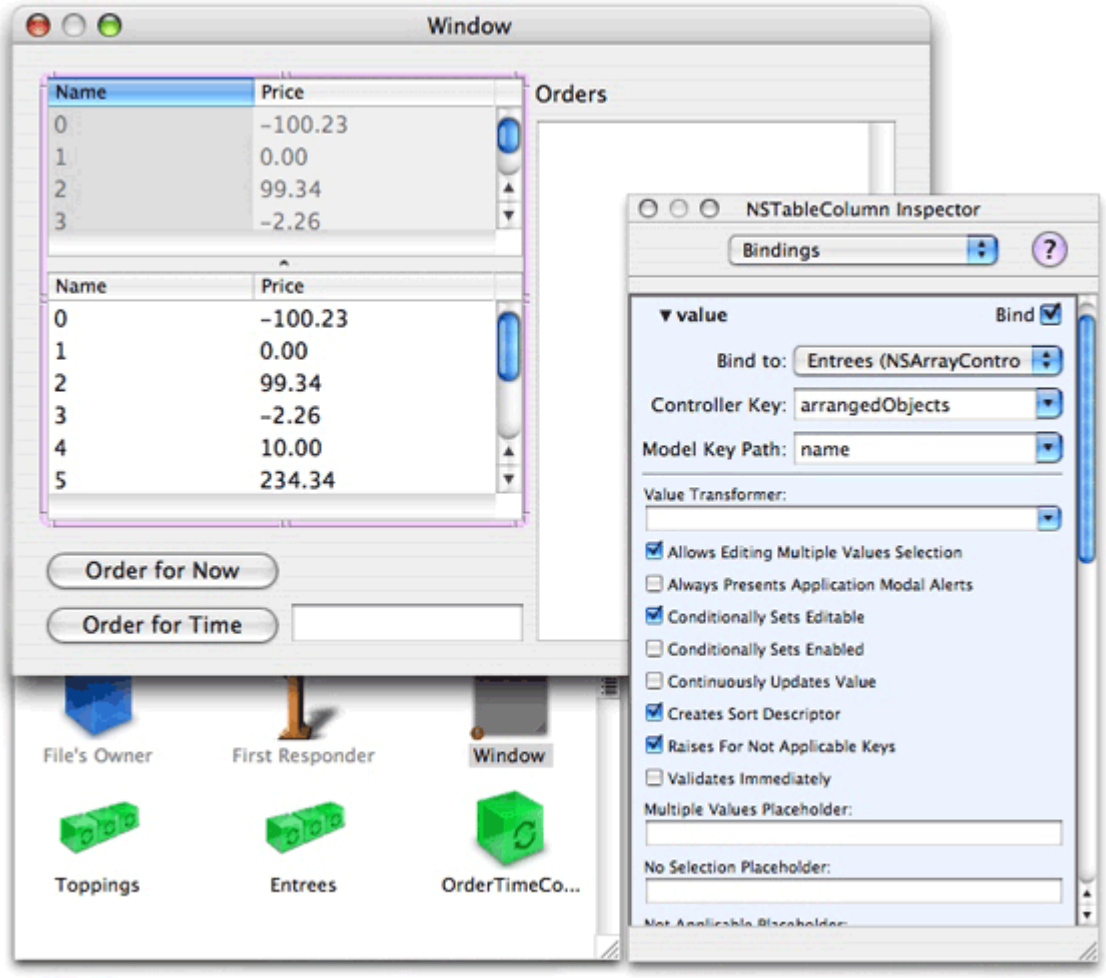
如果您的应用程序中只有模型类是定制类，则只需要让那些类中需要绑定的性质遵循键-值编码的约定，就可以建立绑定了。如果您使用了定制的视图或控制器，则还应该确保它们遵循键-值观察的约定。遵循键-值编码和键-值观察需要满足的要求，请参见 "键-值机制"部分。

请注意：Cocoa 框架中的大部分类都是遵循键-值编码和（如果使用的话）键-值观察的。

您也可以在程序中建立绑定，但在大多数情况下，还是使用 Interface Builder 来建立绑定。在 Interface Builder 中，您可以首先将 NSController 对象从选盘中拖拽到 nib 文件中，然后在 Info 窗口的 Bindings 面板中指定应用程序中的视图、控制器、模型对象的性质之间的关系，以及希望绑定的属性。

图 5-7 给出了一个绑定的实例。在图中，位于上部的表视图对象的"Name"列通过 Entrees 控制器（一个 NSArrayController 对象）的 arrangedObjects 性质绑定到名为 name 的模型属性。arrangedObjects 性质本身也绑定到模型对象的一个数组（图中没有显示）。

图 5-7 在 Interface Builder 中建立绑定



进一步阅读：阅读*Cocoa绑定编程主题*，进一步学习绑定技术和如何用Interface Builder建立绑定。还可以参见 *键-值编码编程指南*和 *键-值观察编程指南*，了解资料中有关这些机制的完整描述。

通告

在对象间传递信息的标准方法是消息传递－即一个对象调用另一个对象的方法。然而，消息传递要求发送消息的对象知道消息的接收者，以及它可以响应什么消息。这个要求对于委托消息和其它类型的消息是可以的。有些时候，我们不希望两个对象之间具有这种紧密的耦合－特别值得注意的原因是它会把本来独立的子系统联结在一起。而且这种要求也是不切实际的，因为它需要把应用程序中很多全然不同的对象之间建立硬编码的连接。

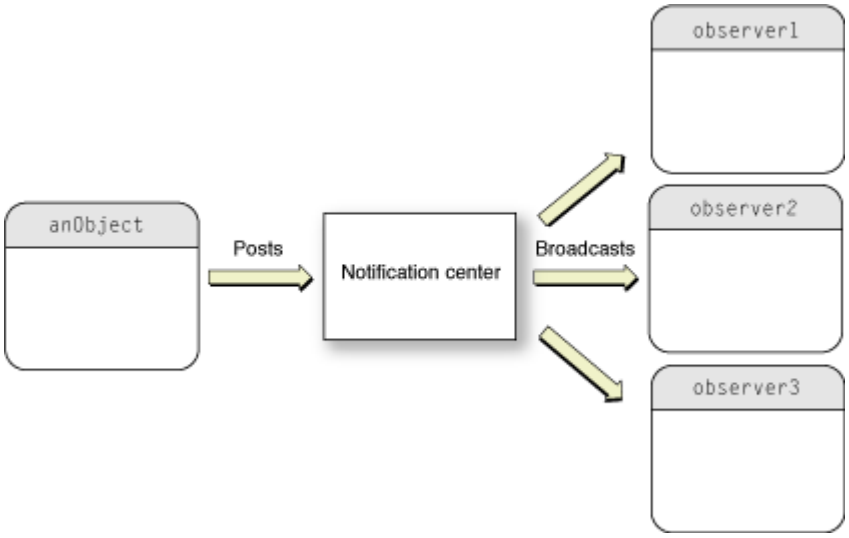
对于不能使用标准的消息传递的场合，Cocoa提供了通告广播模型。通过通告机制，一个对象可以通知其它对象自己正在干什么。在这个意义上，通告机制类似于委托，但是它们之间的区别是很重要的。委托和通告的关键区别在于前者是一对一的通讯路径（在向外委托任务的对象和被委托的对象之间）。而通告是潜在的一对多的通讯方式－也就是一种广播。一个对象只能有一个委托，但可以有很多*观察者*，因为通告的接收者是未知的。对象不必知道那些观察者是什么对象。任何对象都可以间接地通过通告来观察一个事件，并通过调整自己的外观、行为、和状态来响应事件。通告是一种在应用程序中进行协调和聚合的强大机制。

通告机制是如何工作的？这在概念上相当直接。在进程中有一个称为通告中心的对象，充当通告的信息交换和广播中心。在应用程序的其它地方，需要知道某个事件的对象在通告中心进行注册，让它知道当该事

件发生时，自己希望得到通知。这种场景的一个例子是当一个弹出式菜单被选择时，控制器对象需要知道这个事件，以便在用户界面上对反映这个变化。当事件发生时，处理该事件的对象向通告中心发出一个通告，然后通告中心会将它派发给所有的相关的观察者。图 5-8 描述了这种机制。

请注意：通告中心同步地将通告派发给它的观察者。发出通告的对象直到所有的通告被发出后，才重新获得程序的控制权。如果需要以异步的方式发送通告，必须使用通告队列（参见 "通告队列"）。通告队列在对特定的通告进行延迟处理，并根据某些具体的条件将类似的通告进行组合之后，才将通告发给通告中心。

图 5-8 公布和广播通告



任何对象都可以发出通告，也可以在通告中心注册为通告的观察者。发出通告的对象、通告中包含的对象、还有通告的观察者可以是不同的对象，也可以是同一个对象（用同一个对象作为通告的发送者和观察者有它的用处，比如用于空闲处理方面）。发送通告的对象不需要知道通告观察者的任何信息。另一方面，观察者至少需要知道通告的名称和通告对象中封装的字典的键（"通告对象" 部分描述了通告对象的是有什么组成的）。

进一步阅读： 如果需要有关通告机制的完全讨论，请参见 *Cocoa的通告编程主题*。

本部分包含如下主要内容：

何时以及如何使用通告

和委托一样，通告机制是实现应用程序中的对象间通讯的好工具。它使应用程序中的对象可以了解其它地方发生的改变。一般地说，一个对象注册为通告的观察者，是因为它希望在相应的事件发生后或即将发生时进行调整。举例来说，如果一个定制视图希望在窗口调整尺寸的时候改变自己的外观，则可以观察窗口对象发出的 NSWindowDidResizeNotification 通告。通告也允许在对象间传递信息，因为通告中可以包含一个与事件相关的字典。

但是，通告和委托之间是不同的，这些差别也导致这两种机制应该用于不同的地方。在早些时候提到，通告模型和委托模型的主要区别在于前者是广播机制，而委托是一对一的关系。每种模型都有自己的优点，通告机制的优点如下：

- 发出通告的对象不需要知道观察者的标识。
- 应用程序并不受限于 Cocoa 框架声明的通告；任何类都可以声明通告，其实例可以发布通告。
- 通告并不限于应用程序内部的通讯；通过分布式通告，一个进程可以将发生的事件通知另一个进程。

但是，一对一的委托模型也有自己的优点。委托有机会通过将值返回给进行委托的对象来影响事件。另一方面，通告的观察者必须发挥更为被动的作用，在响应事件时，它只能对自身及其环境产生影响。通告方法必须具有如下形式的签名：

```
- (void)notificationHandlerName:(NSNotification *);
```

这个要求使观察对象无法以任何直接的方式影响原来的事件。但是，委托通常可以影响进行委托的对象对事件的处理方式。而且，Application Kit 对象的委托自动注册为其通告的观察者。您只需要实现框架类定义的通告方法，就可以接收通告。

在 Cocoa 中，通告机制并不是观察对象状态变化的唯一选择，在很多情况下甚至都不是最好的选择。Cocoa 绑定技术，特别是为其提供支持的键-值观察（KVO）和键-值绑定（KVB）协议，也可以使应用程序中的对象观察其它对象性质的变化。绑定机制比通告机制更为有效。在绑定机制中，被观察对象和观察对象直接进行通讯，不需要像通告中心这样的中间对象。而且，绑定机制不会像常见的通告那样，因为处理无观察者的变化而对性能产生不利影响。

但是在某些场合中，选择通告比选择绑定更为合理。您可能希望观察事件，而不是对象性质的改变；或者，有些时候遵循 KVO 和 KVB 是不适合实际情况的，特别是当需要发送和观察的通告很少的时候。

即使在适合使用通告的场合下，您也应该知道它对性能的潜在影响。通告发出之后，最终会通过本地的通告中心同步地派发给观察对象。不管通告的发送是同步的还是异步的，这个过程都是要发生的。如果有很多观察者，或者每个观察者在处理通告时都做很多工作，您的程序就会有明显的延迟。因此，您应该小心，不要过度或低效地使用通告。最后，下面这些关于通告用法的原则应该有帮助：

- 对应用程序应该观察的通告有所选择。
- 注册通告时，要具体到通告的名称和发送的对象。
- 尽可能高效地实现处理通告的方法。
- 避免添加或移除很多观察者；通过一些“中间的”观察者将通告的结果传递给它们可以访问的对象要好得多。

进一步阅读：如果您需要有效使用通告的详细信息，请参见 *Cocoa性能指南*中的"通告"部分。

通告对象

通告是一个对象，是 NSNotification 的一个实例。该对象封装了与事件有关的信息，比如某个窗口获得了焦点，或者某个网络连接关闭了。当事件发生时，负责处理该事件的对象会向通告中心发出一个通告，通告中心则立刻将通告广播给所有注册的对象。

NSNotification 对象中包含一个名称、一个对象、还有一个可选的字典。名称是标识通告的标签；对象是指通告的发送者希望发给通告观察者的对象（它通常是通告发送者本身），类似于委托消息的发送者对象，通告的接收者可以向该对象查询更多的信息；字典则用于存储与事件有关的信息。

通告中心

通告中心负责发送和接受通告。它将通告通知给所有符合条件的观察者。通告信息封装在 NSNotification 对象中。客户对象在通告中心中将自己注册为特定通告的观察者。当事件发生时，对象向通告中心发出相应的通告。而通告中心会向所有注册过的观察者派发消息，并将通告作为唯一的参数进行传递。通告的发送对象和观察对象有可能是同一个对象。

Cocoa 框架中包含两种类型的通告中心：

- 通告中心（NSNotificationCenter 的实例），管理单任务的通告。
- 分布式通告中心（NSDistributedNotificationCenter 的实例），管理单台计算机上多任务之间的通告。

请注意，NSNotificationCenter 和很多其它的 Foundation 类不同，不能和其在 Core Foundation 中对应的结构（CFNotificationCenterRef）进行自由桥接。

NSNotificationCenter

每个任务都有一个缺省的通告中心，您可以通过NSNotificationCenter的defaultCenter类方法来进行访问。通告中心在单任务中处理通告。如果需要在同一个机器的不同任务之间进行通讯，可以使用分布式通告中心（参见 "NSDistributedNotificationCenter"部分）。
通告中心同步地将通告发送给观察者。换句话说，在发出一个通告时，在所有的观察者接收和处理完成通告之前，程序的控制权不会返回给发送者。如果需要异步发送通告，可以使用通告队列，这在 "通告队列" 部分中进行描述。
在多线程的应用程序中，通告总是在发送的线程中传送，这个线程可能不同于观察者注册所在的线程。

NSDistributedNotificationCenter

每个任务都有一个缺省的分布式通告中心，您可以通过NSDistributedNotificationCenter的defaultCenter类方法来访问。这个分布式通告中心负责处理同一个机器的不通任务之间的通告。如果需要实现不同机器上的任务间通讯，请使用分布式对象（参见 *分布式对象*部分）。

发送分布式通告是一个开销昂贵的操作。通告会被发送给一个系统级别的服务器，然后再分发到注册了该分布式通告的对象所在的任务中。发送通告和通告到达另一个任务之间的延迟是很大的。事实上，如果发出的通告太多，以致于充满了服务器的队列，就可能被丢弃。

分布式通告通过任务的运行循环来分发。任务必须运行在某种“常见”模式的运行循环下，比如 NSDefaultRunLoopMode 模式，才能接收分布式通告。如果接收通告的任务是多线程的，则不要以通告会到达主线程作为前提。通告通常被分发到主线程的运行循环上，但是其它线程也可以接收通告。

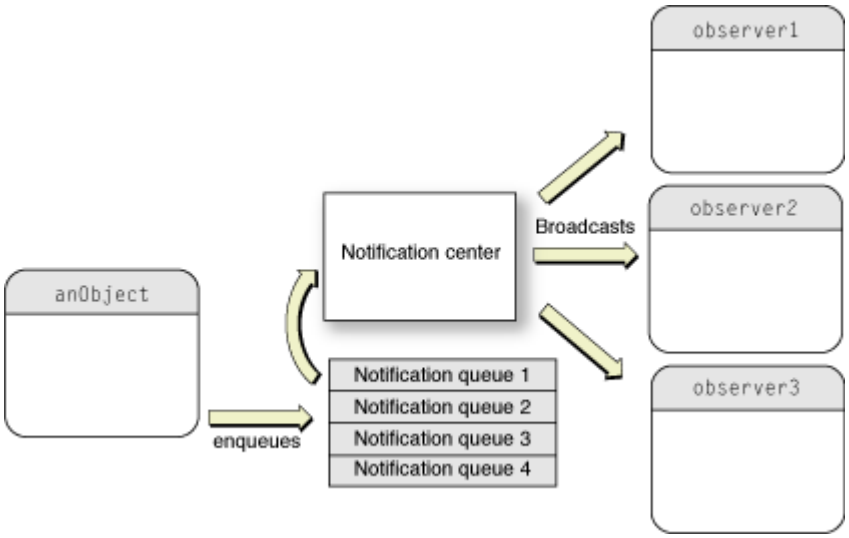
尽管常规的通告中心允许任何对象作为通告对象（也就是通告封装的对象），分布式通告中心只支持将 NSString 对象作为它的通告对象。由于发出通告的对象和通告的观察者可能位于不同的任务中，通告不能包含指向任意对象的指针。因此，分布式通告中心要求通告使用字符串作为通告对象。通告的匹配就是基于这个字符串进行的，而不是基于对象指针。

通告队列

NSNotificationQueue 对象（或者简单称为通告队列）的作用是充当通告中心（NSNotificationCenter 的实例）的缓冲区。通告队列通常以先进先出（FIFO）的顺序维护通告。当一个通告上升到队列的前面时，队列就将它发送给通告中心，通告中心随后将它派发给所有注册为观察者的对象。

每个线程都有一个缺省的通告队列，与任务的缺省通告中心相关联。图 5-9 展示了这种关联。您可以创建自己的通告队列，使得每个线程和通告中心有多个队列。

图 5-9 通告队列和通告中心



聚结的通告

NSNotificationQueue 类为 Foundation Kit 的通告机制增加了两个重要的特性：即通告的聚结和异步发送。聚结是把和刚进入队列的通告相类似的其它通告从队列中移除的过程。如果一个新的通告和已经在队列中的通告相类似，则新的通告不进入队列，而所有类似的通告（除了队列中的第一个通告以外）都被移除。然而，您不应该依赖于这个特殊的聚结行为。

您可以为 enqueueNotification:postingStyle:coalesceMask:forModes:方法的第三个参数指定如下的一或多个常量，指示简化的条件：

- NSNotificationNoCoalescing
- NSNotificationCoalescingOnName
- NSNotificationCoalescingOnSender

您可以对 NSNotificationCoalescingOnName 和 NSNotificationCoalescingOnSender 常量进行位或操作，指示 Cocoa 同时使用通告名称和通告对象进行聚结。那样的话，和刚刚进入队列的通告具有相同名称和发送者对象的所有通告都会被聚结。

异步发送通告

通过 NSNotificationCenter 类的 postNotification:方法及其变体，您可以将通告立即发送给通告中心。但是，这个方法的调用是同步的：即在通告发送对象可以继续执行其所在线程的工作之前，必须等待通告中心将通告派发给所有的观察者并将控制权返回。但是，您也可以通过 NSNotificationQueue 的 enqueueNotification:postingStyle:和 enqueueNotification:postingStyle:coalesceMask:forModes:方法将通告放入队列，实现异步发送，在把通告放入队列之后，这些方法会立即将控制权返回给调用对象。

Cocoa 根据排队方法中指定的发送风格和运行循环模式来清空通告队列和发送通告。模式参数指定在什么运行循环模式下清空队列。举例来说，如果您指定 NSModalPanelRunLoopMode 模式，则通告只有当运行循环处于该模式下才会被发送。如果当前运行循环不在该模式下，通告就需要等待，直到下次运行循环进入该模式。

向通告队列发送通告可以有三种风格：NSPostASAP、NSPostWhenIdle、和 NSPostNow，这些风格将在接下来的部分中进行说明。

尽快发送

以 NSPostASAP 风格进入队列的通告会在运行循环的当前迭代完成时被发送给通告中心，如果当前运行循环模式和请求的模式相匹配的话(如果请求的模式和当前模式不同,则通告在进入请求的模式时被发出)。由于运行循环在每个迭代过程中可能进行多个调用分支（callout），所以在当前调用分支退出及控制权返回运行循环时，通告可能被分发，也可能不被分发。其它的调用分支可能先发生，比如定时器或由其它源触发了事件，或者其它异步的通告被分发了。

您通常可以将 NSPostASAP 风格用于开销昂贵的资源，比如显示服务器。如果在运行循环的一个调用分支过程中有很多客户代码在窗口缓冲区中进行描画，在每次描画之后将缓冲区的内容刷新到显示服务器的开销是很昂贵的。在这种情况下，每个 draw...方法都会将诸如“FlushTheServer” 这样的通告排入队列，并指定按名称和对象进行聚结，以及使用 NSPostASAP 风格。结果，在运行循环的最后，那些通告中只有一个被派发，而窗口缓冲区也只被刷新一次。

空闲时发送

以 NSPostWhenIdle 风格进入队列的通告只在运行循环处于等待状态时才被发出。在这种状态下，运行循环的输入通道中没有任何事件，包括定时器和异步事件。以 NSPostWhenIdle 风格进入队列的一个典型的例子是当用户键入文本、而程序的其它地方需要显示文本字节长度的时候。在用户输入每一个字符后都对文本输入框的尺寸进行更新的开销是很大的（而且不是特别有用），特别是当用户快速输入的时候。在这种情况下，Cocoa 会在每个字符键入之后，将诸如“ChangeTheDisplayedSize”这样的通告进行排队，同时把聚结开关打开，并使用 NSPostWhenIdle 风格。当用户停止输入的时候，队列中只有一个“ChangeTheDisplayedSize”通告（由于聚结的原因）会在运行循环进入等待状态时被发出，显示部分也因此被刷新。请注意，运行循环即将退出（当所有的输入通道都过时的时候，会发生这种情况）时并不处于等待状态，因此也不会发出通告。

立即发送

以 NSPostNow 风格进入队列的通告会在聚结之后，立即发送到通告中心。您可以在不需要异步调用行为的时候 使用 NSPostNow 风格（或者通过 NSNotificationCenter 的 postNotification:方法来发送）。在很多编程环境下，我们不仅允许同步的行为，而且希望使用这种行为：即您希望通告中心在通告

派发之后返回，以便确定观察者对象收到通告并进行了处理。当然，当您希望通过聚结移除队列中类似的通告时，应该用 enqueueNotification...方法，且使用 NSPostNow 风格，而不是使用 postNotification:方法。

委托、观察者、和目标的所有权

向外委托任务的对象并不拥有委托或数据源对象的所有权。类似地，控件和单元不拥有其目标对象的所有权，通告中心也不拥有通告观察者的所有权。因此，这些框架对象都遵循*不*保持其目标、观察者、委托、和数据源对象的约定，而是保持这些对象的一个弱引用－也就是说，存储一个对象的指针。

按照对象所有权策略的推荐，对被其它对象拥有的对象应该进行保持和无条件归档，而对被引用（但不被拥有）的对象则不进行保持，但进行有条件地归档。这个所有权策略的实际目的是为了避免循环引用，即避免两个对象互相引用的情况。保持对象会创建一个该对象的强引用，而在所有的强引用释放之前，对象不能被解除分配。如果两个对象彼此互相保持，则它们将永远不能被解除分配，因为它们之间的关联不能被打断。

如果您创建一个带有委托、数据源、观察者、和目标的 Cocoa 框架类的子类，则在该子类中永远不应显式保持这些对象，而应该创建这些对象的弱引用（未经保持），以及有条件地进行归档。

进一步阅读：有关所有权策略、弱引用、和循环引用的更多信息，请参见 *Cocoa内存管理编程指南*文档中的"对象的所有权及其去除"部分。

核心应用程序架构

当用户启动一个 Cocoa 应用程序时，一个对象网络就会被置入内存。这个运行时网络包含不同类型的 Application Kit 对象，每个对象都扮演特定的角色。这些对象以不同的方式互相关联，其关联方式由所有权、依赖、和协作来定义。本章将讨论这种应用程序架构，考察各种核心对象扮演的角色，它们的基本属性，以及对象之间的关系。

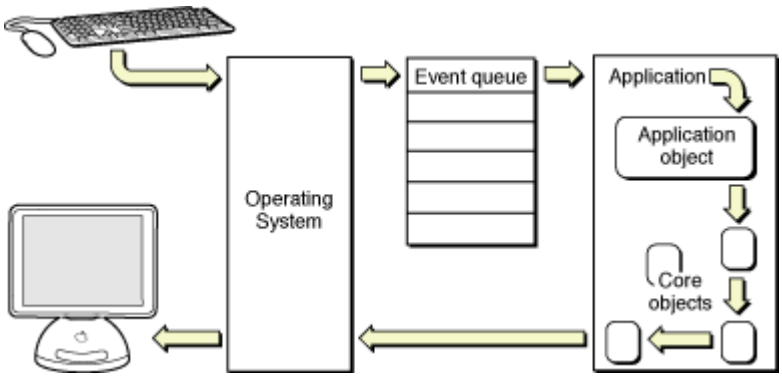
本部分包括如下内容：

事件-描画周期的再次讨论
全局的应用程序对象
窗口
视图
响应者及响应者链
控件和菜单
Nib文件和其它应用程序资源

重温事件-描画周期

对象网络的总体目标是快速处理事件和描画周期的工作。图 6-1（您可以回忆 "为Cocoa应用程序添加行为"部分的内容）展示了这个周期。

图 6-1 事件和描画周期



应用程序代表一个图形用户界面；用户通过鼠标或键盘输入数据或进行选择，从而与该界面进行交互；而这种输入被转换为事件传递给应用程序，并置入其事件队列。对于该队列中的每个事件，应用程序负责为其找到最合适的处理对象；在事件处理完成后，应用程序可能适当地修改显示视图。然后应用程序获取队列中的下一个事件，开始新的周期。

参与这个架构的核心对象是 NSApplication、NSWindow、和 NSView 类的直接或间接的后代。

全局的应用程序对象

每个Cocoa程序都由一个NSApplication对象来管理，该对象是一个全局变量，名为NSApp。这个 NSApplication（或其定制子类）的单件实例的主要责任是：获取目标为该应用程序的用户或系统事件，并派发给恰当的对象。它还负责管理应用程序的窗口，包括跟踪键盘焦点窗口或主窗口（参见 "窗口状态"）的状态。

本部分包含如下主要内容：

主事件循环
事件派发的进一步讨论
窗口管理
处理Apple事件

主事件循环

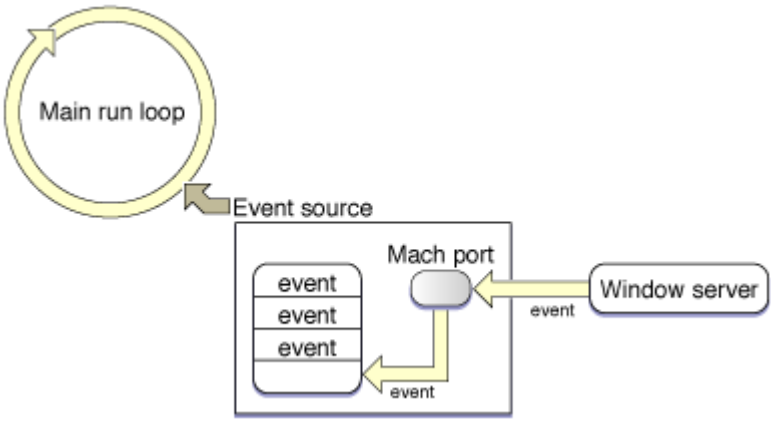
NSApp 做的绝大部分工作发生在应用程序的主事件循环中，该循环是事件和描画循环的基础。考察 Cocoa 应用程序中的 main 入口函数的逻辑有助于理解主事件循环的工作机制。在一个标准的 Xcode Cocoa 工程中，main 会调用 NSApplicationMain 函数来处理三个重要的任务，其顺序如下：

1. 调用 sharedApplication 类方法来获取共享的应用程序对象（NSApp）
2. 将应用程序的主 nib 文件载入内存
3. 将应用程序运行起来（[NSApp run]）。

让我们从一些背景知识开始，更细致地考察一下这些步骤。一个运行着的应用程序是一个进程，每个进程都有一个主线程，可能还有一个或更多的辅助线程。每个线程都有一个运行循环，用于监控进程的输入源，并在输入源做好准备的时候将控制权派发给它们。

除了保证单件实例之外，sharedApplication 方法主要做什么呢？它还负责建立一些程序的基础设施，以便接收和处理来自窗口服务器的事件。在初始化全局应用程序对象的过程中，NSApplication 通过生成一个接收事件的事件源建立一个与窗口服务器的连接（实现为一个 Mach 端口），并建立一个应用程序事件队列，即一个 FIFO（先进先出）的机制，使事件在到达事件源时被取出，并放入队列等待处理。最后，NSApplication 将事件源作为输入源，对主运行循环，即主线程的运行循环，进行初始化（参见图 6-2）。

图 6-2 主事件循环及事件源



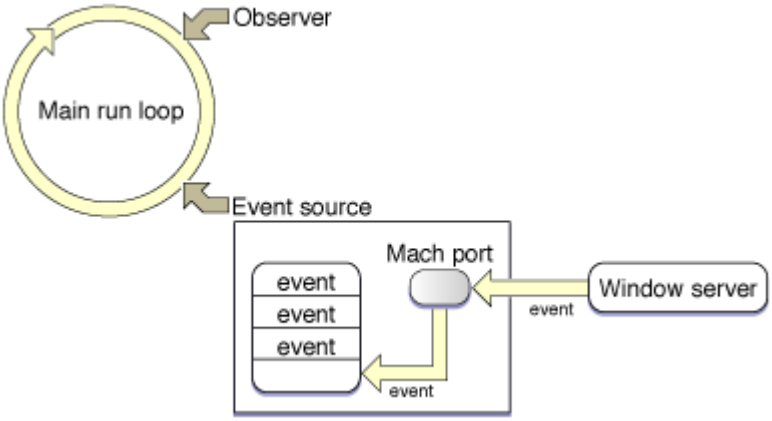
窗口服务器从 I/O Kit 设备驱动程序接收事件，并派发到恰当的进程中。而进程则在运行循环中接收来自事件源的事件，并将它们放入队列中。

在应用程序装载主 nib 文件时，文件中的对象及对象之间的连接会被解档。主 nib 文件中总是包含应用程序菜单，还可能包含一个或多个窗口对象（以及其中的视图对象）。其它的 nib 文件也可能在启动的时候被解档，比如基于文档的应用程序，当用户在 Finder 中双击一个文件时，与文档有关的 nib 文件可能需要被解档。装载初始的图形用户界面是必需的，只有这样用户才能发起请求，比如选择一个菜单项，并最终产生一个需要应用程序处理的事件。

NSApplication类的run方法负责主事件循环的主要工作，它首先为应用程序注册Apple Events的处理器（请参见 "处理Apple Event"部分），然后在一个封闭的while循环中进行如下工作，直到应用程序终止：

1. 作为运行循环中挂起的窗口-显示的观察者（参见图 6-3，其结果是重画窗口中被标识为“弄脏”的区域）。

图 6-3 主事件循环及运行循环的观察者



2. 从事件队列中获取下一个事件（nextEventMatchingMask:untilDate:inMode:dequeue:）。
3. 将事件派发给下一个可以进行处理的对象，绝大多数情况下就是NSWindow对象（sendEvent:）。为了弄清楚这个问题，请参见 "事件派发的进一步讨论"部分。

   最终可能会有很多对象参与事件的处理，调用堆栈深度可能会显著增长。在事件处理完成后，控制权就返回 run 方法。

主事件循环的过滤点是 nextEventMatchingMask:untilDate:inMode:dequeue:方法。当事件队列中有事件时，它可以将位于队列最顶部的事件取出，并将它转换为一个 NSEvent 对象。如果队列中没有事件，该方法就会阻塞等待。在该方法阻塞期间，来自窗口服务器的新事件会被处理，并放入队列中。而队列中的新事件就会“唤醒”nextEventMatchingMask:untilDate:inMode:dequeue:方法，并返回队列中第一个匹配的事件。

如果窗口的auto-display（自动显示）特性处于激活状态，且窗口中有标识为需要显示的视图对象，run方法也会在运行循环中安装一个观察者（即一个输入源）。在NSApp对象处理队列中的下一个事件之前，该观察者就会被触发，从而引起相应视图对象的重画。有关窗口内容自动显示的更多信息，请参见 "窗口和描画"部分。

事件派发的进一步讨论

在 sendEvent:方法的实现中，NSApp 查看了传入的事件类型，并根据该类型进行事件的派发。通常情况下，派发的目标是某个应用程序的窗口，应用程序通过调用该 NSWindow 对象的 sendEvent:方法来进行事件的传递。对于键盘和鼠标这两大类输入事件，NSApp 对象通过不同的方法来确定事件的目标窗口。

当用户按下键盘中的按键时，就会产生键盘事件。NSApp 会将这些事件传递给键盘焦点窗口—即当前正在接收键盘输入的应用程序窗口（在某些情况下，比如等价键，应用程序会对 NSKeyDown 事件进行特殊处理，而不是对其进行派发）。

当用户用鼠标点击窗口中的对象时，比如点击图形描画程序中的一个图形形状，就会产生事件。NSApp 会将鼠标事件派发给发生事件的窗口。

如果被点击或进行其它操作的对象是一个控制对象，比如一个按键或滑块，则该对象还会向应用程序额外发送一种消息—即动作消息。这种消息会调用 NSApplication 的 sendAction:to:from: 方法，如果没有指定消息目标，该方法就会检索应用程序的键盘焦点窗口和（如果必要的话）主窗口的响应链，以便寻找合适的目标，然后将消息发送给该目标。

NSApp 在 sendEvent:方法中处理其它类型的事件。某些事件是面向应用程序自身的（比如程序的激活和失效），NSApp 就自行处理；其它事件和一个或多个窗口相关—比如某个窗口被暴露出来，或者屏幕的分辨率发生变化。对于这些事件，NSApp 会调用被影响的窗口对象中合适的方法。

进一步阅读：响应者链在 "响应者和响应者链"部分中描述。有关键盘焦点窗口的更多信息，请参见 "窗口状态"部分。有关NSControl对象及其如何与NSCell对象协同工作的讨论，请阅读 "控件和菜单"部分。

窗口的管理

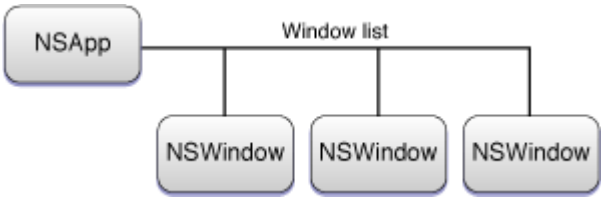
全局应用程序对象的部分工作是管理应用程序的窗口。NSApp 负责如下的窗口管理任务：

- 跟踪当前的键盘焦点窗口和主窗口
- 窗口的隐藏和取消隐藏
- 在应用程序终止时对窗口进行解除分配
- 管理应用程序切换过程中窗口的激活和不激活的状态

如 "事件派发的进一步讨论"部分介绍的那样，NSApp还负责把面向窗口的事件和动作消息派发给受影响的窗口。此外，它也负责维护应用程序的Window菜单，以及管理模式窗口和面板（具体描述参见 "模式窗口"）。

为了找到窗口，应用程序对象保有一个窗口的引用，称为窗口列表（参见图 6-4）。在进行某些窗口管理任务（比如窗口对象的解除分配）时，需要遍历这个列表（您随时都可以向 NSApp 对象发送 windows 消息，以得到窗口列表）。应用程序对象也可以从当前的 NSEvent 对象得到窗口号码，然后通过调用 windowWithWindowNumber:方法将窗口号码转化为窗口的引用。

图 6-4 应用程序的窗口列表



请注意：NSApp不针对屏幕上的窗口分层（Z-order）做任何事。这是由窗口服务器管理的。有关窗口Z-order的讨论请参见 "窗口的Z-Order和级别"。

处理Apple Event

并不是所有应用程序必需处理的事件都来自事件队列。Mac OS X 系统上的其它进程，比如 Finder 和 Launch Services，可以通过 Apple Event 来和其它进程进行通讯。它们有时会向应用程序发送 Apple Event 进行某些通知，比如当用户双击 Finder 窗口中的某个文档时，或者用户在 Apple 菜单上选择 Shut Down 要求应用程序终止时。

当一个 Cocoa 应用程序启动时，所做的第一件事就是注册一些 Apple Event 的处理器。在其它进程向该程序发送 Apple Event 时，相应的处理器就会被调用。Cocoa 应用程序为下面的 Apple Event 注册处理器：

Apple 事件 ID	描述

kAEOpenApplication	启动一个应用程序。
kAEReopenApplication	重新打开一个应用程序。比如，当用户点击 Dock 上的某个正在运行的应用程序的图标时，就会发送这个事件。
kAEOpenDocuments	向应用程序提供一个需要打开的文档列表。通常当用户在 Finder 中选择并双击一或多个文档时，发送这个事件。
kAEPrintDocuments	向应用程序提供一个需要打印的文档列表。通常当用户在 Finder 中选择一或多个文档，并从 File 菜单中选择 Print 时，发送这个事件。
kAEOpenContents	向应用程序提供拖拽的内容，比如文本或图像。这个事件通常发生在用户拖拽一个文件并放在 Dock 上的应用程序图标的时候。
kAEQuitApplication	要求终止应用程序。

请注意：有关这个主题的更多信息，请参见"Cocoa应用程序如何处理Apple Event "部分。

窗口

应用程序通过窗口来得到一个屏幕区域，并在其中显示内容，以及响应用户动作。窗口是描画和事件处理的必要元素。

进一步阅读：*Cocoa窗口编程指南*文档中对本部分涉及的很多主题进行进一步的讨论。

本部分包括如下主要内容：

应用程序窗口

虽然开发一个带有不可见窗口（比如一个后台程序）的应用程序是可能的，但这种程序很少见。根据潜在可能显示的窗口个数，通常可以把应用程序分成两类：

- 基于文档的应用程序—这种应用程序可以创建多个文档，每个文档有自己的窗口，比如字处理程序和描画程序。在基于文档的窗口中，用户可以通过选择一个菜单项（通常是File > New菜单）来创

建一个新文档。Cocoa中的大多数基于文档的程序都是基于它提供的文档架构（您可以在 "其它Cocoa架构"部分中看到这种架构的概述）。

- 单窗口应用程序—这种应用程序在任何时候都只显示一个窗口，比如 Mac OS X 的 iSync 和 Font Book。单窗口程序在启动的时候显示其窗口。关闭窗口则通常导致应用程序的终止。

所有的应用程序都可以有一些辅助窗口，也称为对话框和面板。这些窗口从属于当前文档窗口，在单窗口程序中则是从属于主窗口，并以各种方式支持文档窗口或主窗口—比如允许字体和颜色的选择，允许从选盘上选取工具，或者显示警告信息。辅助窗口通常是模式的，更多信息请参见 "面板"部分。

NSWindow和窗口服务器

在 Cocoa 中，一个 NSWindow 对象代表一个物理窗口。窗口服务器创建一个物理窗口，并最终在屏幕上管理它们。它为每个窗口分配一个唯一的数字，作为标识。NSWindow 对象和它对应的物理窗口之间的连接是通过其窗口号建立起来的。

当窗口服务器创建一个窗口时，会取得一个窗口的图形上下文，并对其图形状态栈进行初始化。窗口服务器还会创建一个窗口的背板存储区，即用于存放像素值、最终被置入显示设备的帧缓冲区的内存区域。

请注意：创建一个没有背板存储区的窗口是可能的，这种窗口称为迟延窗口。迟延窗口的背板存储区仅在需要被显示时才会被创建。

窗口的缓冲

屏幕上的多个窗口之间常常会有覆盖。当一组窗口被逐个叠放在一起时，用鼠标点击一个部分被遮掩的窗口通常会使该窗口显示在最前面。在这个过程中，该窗口被遮掩的部分就会变为可见。如果窗口有一个显示缓冲区，则窗口服务器会自动显示其内容，否则应用程序必需对其进行重画。

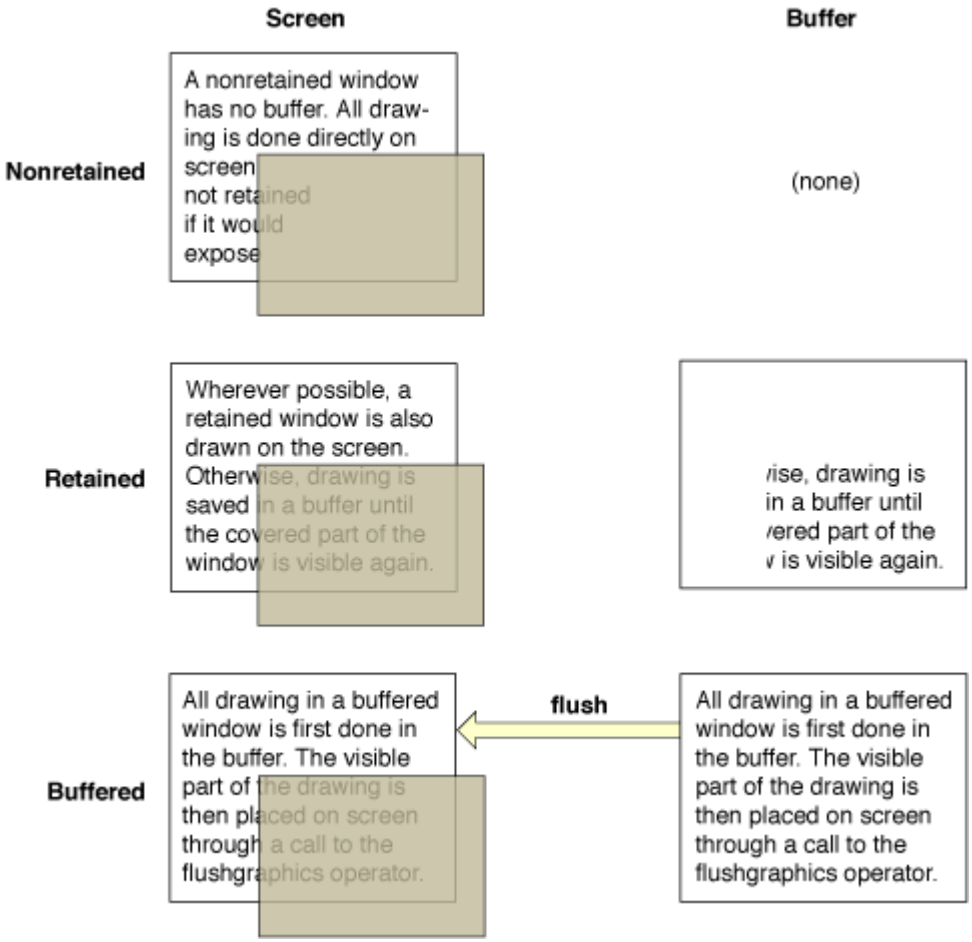
窗口缓冲机制有两种类型。窗口的显示缓冲区或者存储整个窗口的像素值，或者存储窗口中当前被遮掩部分的像素值。当一个带有显示缓冲区的被遮掩窗口被放在屏幕的最前面时，窗口服务器会将显示缓冲区中需要暴露的部分拷贝到屏幕上。在创建 NSWindow 对象时，您可以选择如下三种缓冲方案，图 6-5 对这些方式进行描述。

- **无保持模式** 不提供显示缓冲区。所有的描画都直接发生在屏幕上—也就是说，像素值会被直接写入到帧缓冲区。如果窗口的一部分被另一个窗口覆盖，则该部分的窗口内容对应的位就会丢失。当被遮掩的窗口暴露出来时，应用程序必需重画被遮掩的部分；如果不进行重画，该部分就显示为窗口的背景颜色。

- **有保持模式** 虽然被保持的窗口有显示缓冲区，但是在可能的情况下，描画还是直接在屏幕上进行。显示缓冲区只在窗口被遮掩的时候受影响；在这种情况下，被遮掩区域的像素值会被写到显示缓冲区中。当被遮掩的区域显露出来时，显示缓冲区的内容就被拷贝到屏幕上了。

- **有缓冲模式** 缓冲区中含有一个背板存储区中内容的精确拷贝。事实上，描画在显示缓冲区上进行。如果与之交叠的窗口具有阴影或者透明效果的话，描画内容还会和它进行合成，然后再刷新到屏幕上。如果一个窗口被遮掩，然后被置为最前面的窗口，则整个显示缓冲区都会被拷贝到屏幕上。

图 6-5 窗口缓冲方式



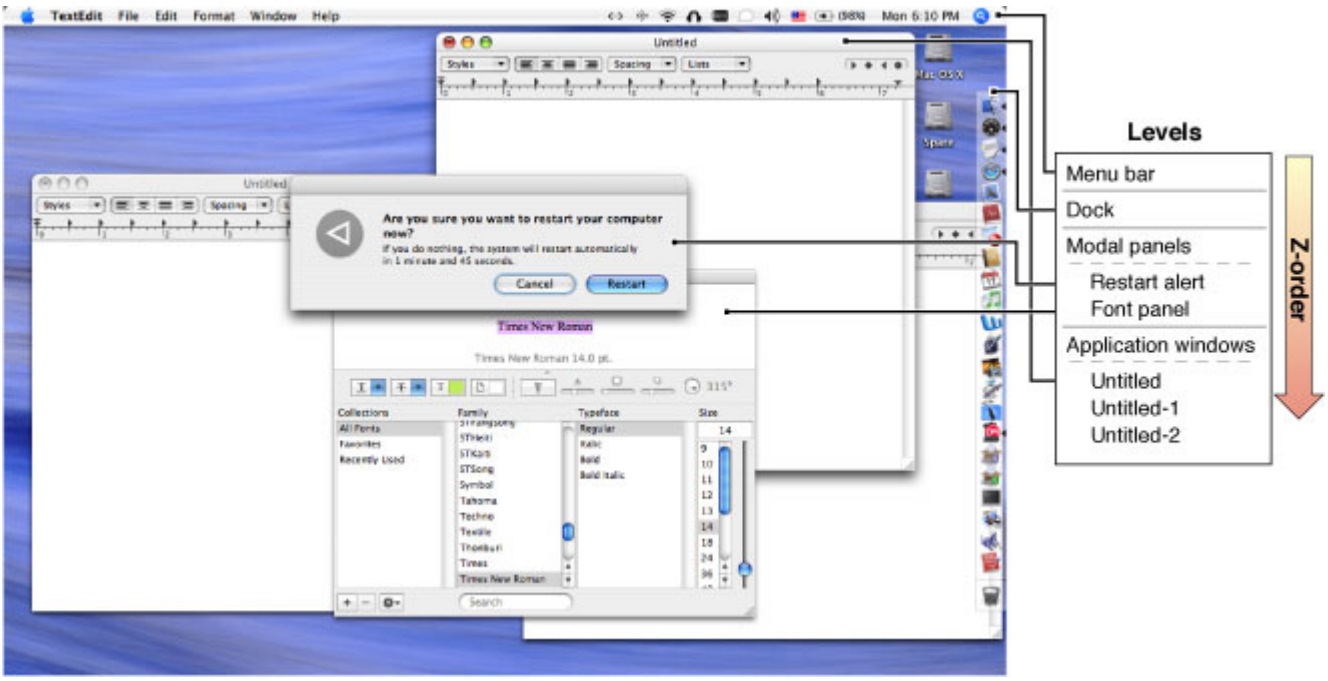
由于有缓冲的窗口对于透明和非矩形的形状是必需的，所以几乎所有的 Cocoa 窗口都是有缓冲的。无保持的窗口适用于临时的图像或简单的连接线，比如 Interface Builder 中的目标-动作连接。

窗口的Z-Order和级别

窗口服务器以从前而后的顺序维护显示在屏幕上的窗口，我们称为屏幕列表或 Z-order。每个窗口在列表中都有一个唯一的位置。一个窗口可以位于列表的顶部—也就是最前窗口，也可以位于列表的底部— 即这个窗口可能被其它任何窗口遮掩，还可以位于其它任何窗口的上面或下面。用户每次点击一个窗口时，Z-order 就会发生变化，使该窗口成为最前窗口，而原来的最前窗口则位于它的后面。

窗口的级别是由窗口分层的概念发展而来的，它指由特定功能类型的窗口组成的、经过分层的子集。窗口级别是有层次顺序的，级别高的窗口显示在级别低的窗口之上。这个策略保证了某个级别的窗口总是显示在另一个级别窗口的上面或下面。举例来说，一个模式的系统窗口总是出现在所有应用程序窗口的上面。因此，Z-order 实际上是根据窗口级别来维护的（参见 图 6-6）。

图 6-6 窗口的级别和 Z-order



窗口服务器支持几种窗口级别，其顺序如下：

1. 屏幕保护（屏幕保护程序适用大小和屏幕相同但没有标题栏的窗口来显示内容）。屏幕保护程序的窗口显示在所有窗口的上面。
2. 菜单栏（包括应用程序菜单）
3. Dock
4. 模式窗口和面板（请参见 "模式窗口"部分）
5. 上下文菜单
6. 浮动窗口（比如描画程序的选盘窗口）
7. 任何其它类型的窗口，包括应用程序窗口

窗口可以被显式地从屏幕列表中拿掉，在这种情况下，我们称之为离屏窗口。当一个窗口从列表中移走时，会从屏幕上消失；而当它被放入列表中时，就会再次回到屏幕上。事件不被派发到离屏的窗口上。应用程序对窗口进行隐藏，就是基于从屏幕列表中移走窗口来实现的。离屏窗口必需被缓冲或者保持，才能支持描画。

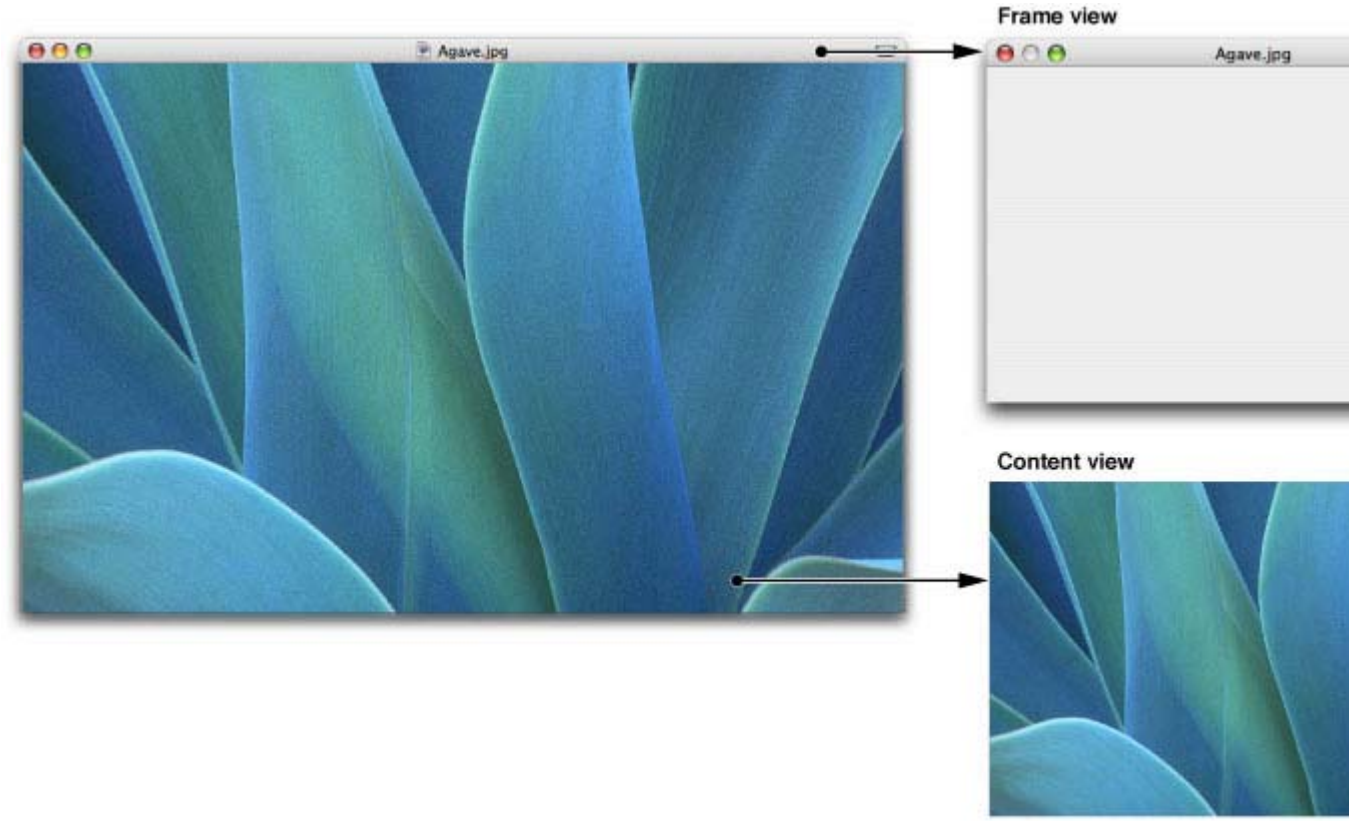
请注意：您也可以将无边界窗口的边框坐标设置为屏幕可视坐标的外部，从而把它从屏幕上移走。然而，您不能对其它类型的窗口进行这样的操作，Application Kit 要求那些窗口至少在屏幕边界内部分可见。NSWindow 类中定义了一些操作应用程序窗口的 Z-order、将窗口从屏幕列表中移走或放回、以及设置窗口级别的方法。详细信息请参见 NSWindow 的类参考。

窗口的组成部分

一个窗口有两个主要部分：框架区域和内容区域，如图 6-7 所示。这些区域都是视图—具体地说，是NSView子类的实例（参见 "视图"部分）。框架视图围绕整个窗口区域，并负责描画窗口的边界和标题栏，它是

NSWindow创建的私有对象，不能通过子类化来改变。然而，当您创建窗口的时候，可以指定在框架视图上需要的控件和其它特性—比如关闭按键、最小化按键、调整尺寸用的三角区域、以及标题栏。

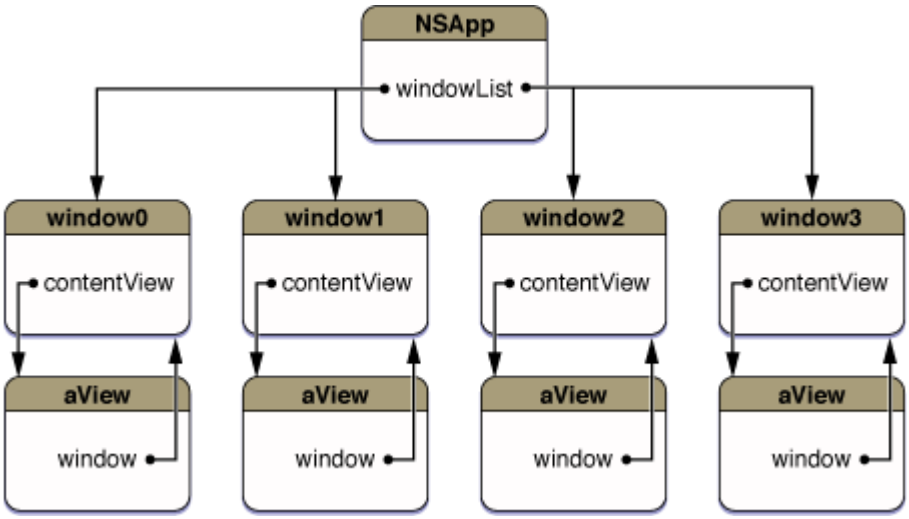
图 6-7 窗口的框架视图和内容视图



内容视图位于框架视图的内部，通常占据除了标题栏和窗口边界之外的整个区域。图 6-7 显示内容视图和框架视图的相对关系（尽管开发者可以创建一个没有标题栏和可视边界的窗口，但它让然有框架视图）。内容视图是框架视图唯一公开的子视图，它不是私有的对象，所以如果愿意的话，您可以用自己定制的视图加以替换。虽然内容视图有一个父视图（也就是在视图层次中拥有和包围内容视图的视图），但该父视图是一个私有对象。因此内容视图是窗口视图层次的根视图。有关视图层次的更多信息，请参见 "视图"部分。

和其它所有视图对象一样，内容视图保有一个窗口的引用，您可以通过 window 方法来访问这个引用。图 6-8 显示了 NSApp 及其窗口列表、列表中每个窗口的内容视图，以及这些对象之间的关系。

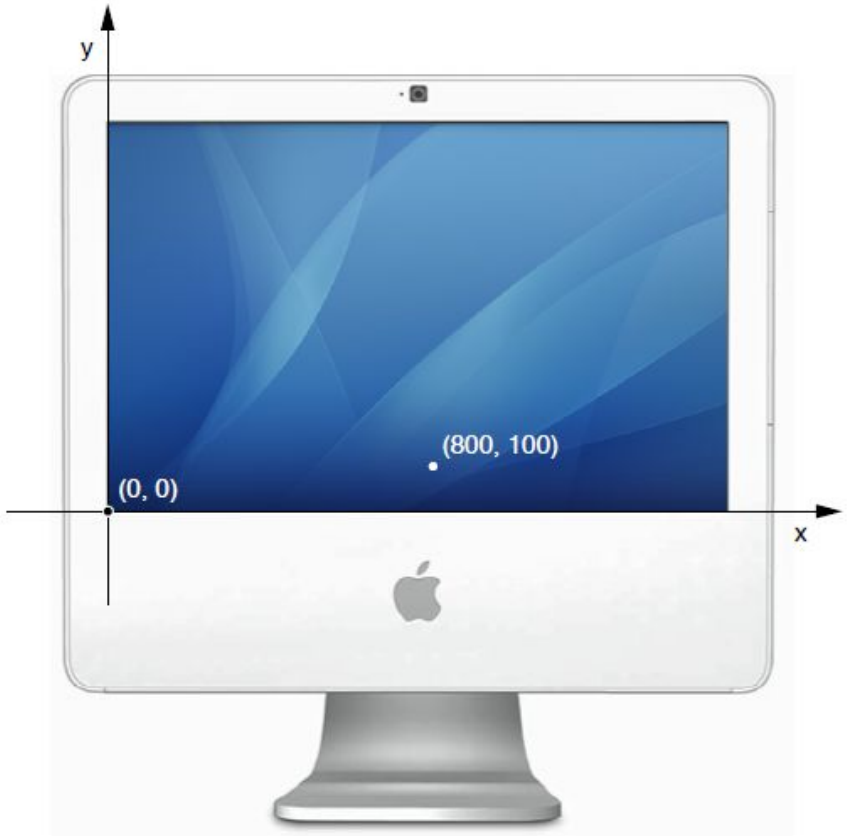
图 6-8 NSApp、窗口、和内容视图之间的关系



窗口坐标

窗口坐标是相对于屏幕坐标而言的。整个屏幕可以考虑为一个位于右上象限的两维坐标格，原点在左下角，水平向右是正向的 X 轴，垂直向上是正向的 Y 轴（如图 6-9）。您可以引用这个栅格来定位屏幕上的一个点。

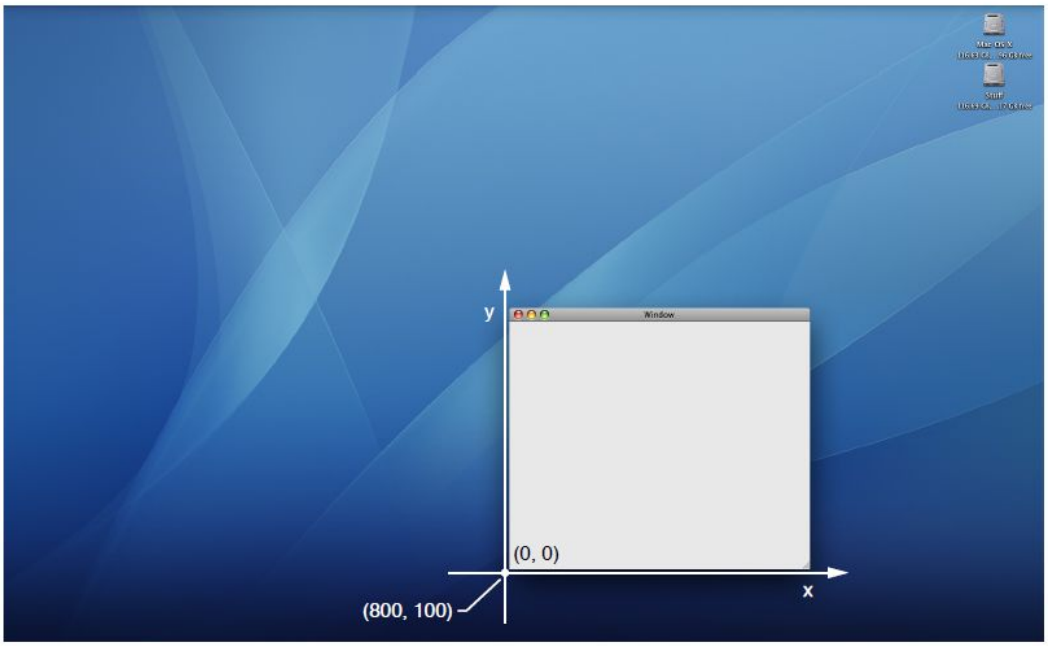
图 6-9 屏幕坐标



屏幕坐标系统的基本功能是定位屏幕上的窗口。当应用程序创建一个新的窗口并放到屏幕上时，必须指定窗口的初始尺寸和在屏幕坐标上的位置。然而，在窗口中*描画*时使用的是不同的坐标系统，该坐标系统是专用于窗口的：即窗口的基准坐标系统（参见图 6-10）。这个坐标系统和屏幕坐标系统在两个方面有所不同：

- 它只应用于某个特定的窗口，每个窗口都有自己的基准坐标系统。
- 坐标的原点是窗口的左下角，而不是屏幕的左下角。如果窗口移动了，原点和整个坐标系统也会跟着移动。窗口中的图像保持的是它在基准坐标系统中的位置，而与窗口的位置无关。

图 6-10 基准坐标系统



基准坐标系统是定义窗口中各个视图坐标系统的参考点。框架视图将窗口的边界和标题栏直接画在基准坐标系统中，内容视图及其子视图则描画在将基准坐标系统进行坐标变换之后的坐标系统中。

窗口和描画

窗口本身并不进行描画，这个工作留给了窗口包含的视图对象。然而，NSWindow 对象在协调视图描画中发挥着重要作用。

通常情况下，应用程序的对象在事件循环中可能会将视图（或视图中的区域）标识为需要重画。NSWindow 对象会将这些"弄脏"了的视图的引用收集起来，按照视图层次中的顺序，放在一个列表中。之后不久—通常是在事件循环的最后—Application Kit 会从最上面的视图（也就是最靠近内容视图的视图）开始，遍历这个列表，并要求每个视图对自身进行重画。这样，位于其它视图背后的视图就会首先被描画。

窗口视图的这种自动描画只有当窗口的auto-display特性被打开时才会发生，这个特性缺省是打开的（请参见setAutodisplay:方法）。如果您关闭了这个特性，则应用程序必须负责在必要的时候更新窗口内容。此外，您可以通过向NSView对象发送display,displayRect:或

displayRectIgnoringOpacity:消息来旁路这个自动显示机制，这些消息会使目标视图及其子视图立刻重画。进一步信息请参见 "显示一个视图"部分。
您可以通过 NSWindow 的 display 和 displayIfNeeded 方法来重画整个窗口。这些方法会强制立即显示窗口中的视图对象，尽管上面的第二个方法只遍历无效的视图列表。display 方法会使窗口视图层次中的每个视图进行自我描画，从内容视图开始。在一次性的窗口（即背板存储在窗口离屏的时候会被释放的窗口）显示在屏幕上之前，会调用这个方法。此外，调用 setFrame:display: 并在第二个参数传入 YES，也会导致窗口调整自己的尺寸，并重画所有的视图。

窗口更新是一个与窗口显示有关的机制。在每个事件循环中，NSApp 会往窗口列表中的每个窗口发送 update 消息。update 的缺省实现是什么都不做，但 NSWindow 的子类可以通过重载这个方法来考察应用程序的状态，并根据需要改变窗口的外观或行为。

窗口状态

应用程序的每个窗口都有一个与用户交互相关的状态，窗口的外观是这个状态的指示器。不活动的窗口是打开的，也可能是可见的，但不显示为前景窗口，其标题栏上的控件和标题是灰色的。如果用户没有通过点击将这种窗口转为前景窗口，就不能和它进行交互。

活动窗口是当前用户输入或关注的焦点。它们是前景窗口，窗口中的控件是有颜色的，标题的字体为黑色字体。活动窗口可以有两种状态：主窗口和键盘焦点窗口。充当用户当前关注焦点的活动窗口就是主窗口。在很多时候主窗口也是键盘焦点窗口，当前接收键盘事件的窗口就是键盘焦点窗口。

然而，有些时候主窗口和键盘焦点窗口是不同的窗口，主窗口仍然是用户关注的焦点，但键盘事件的输入焦点是另一个窗口。键盘焦点窗口必须有一个可以接收键盘输入字符的对象，比如文本输入框。在这种情况下，键盘焦点窗口通常是一个对话框或面板（比如 Find 对话框），使用户可以输入一些与主窗口相关的数据。

应用程序对象（NSApp）负责维护应用程序窗口的主窗口状态和键盘焦点状态。窗口的状态通常决定其接收和派发的事件（请参见 "事件派发的进一步讨论"部分）。

进一步阅读：更多有关主窗口和键盘焦点窗口的讨论，请参见 *苹果人机界面指南*文档中的窗口行为部分。

窗口和事件处理

NSWindow 以两种主要的方式参与事件和事件的处理。在事件派发的活动中，它主动参与；在其它活动中，它则是有约束的事件流的被动接收者。

事件派发

如 "事件派发的进一步讨论"部分中描述的那样，应用程序将接收到的大多数事件通过sendEvent:消息发给事件“所属”的NSWindow对象中。然后轮到窗口对象，它定位应该接收事件的NSView对象，并将恰当的NSResponder消息发送给该视图，传入NSEvent对象。举例来说，如果是键盘按下（key-down）事件，它就向视图对象发送keyDown:消息；如果是鼠标拖动（mouse-dragged）事件（左键），则发送mouseDragged:消息。对于键盘和鼠标事件，窗口对象定位目标视图的方法通常是不一样的：

- 键盘事件会被发送给视图层次中的第一响应者。
- 鼠标事件会被发送给发生鼠标事件的视图对象。

键盘按下(NSKeyDown)和鼠标左键按下(NSLeftMouseDown)事件要求最大限度地进行处理。
NSWindow 在试图将 keyDown:消息发送给第一响应者视图之前，会将键入的字符推送给系统输入管理

器，由它将输入解释为文本或可执行的命令。在发送 mouseDown:消息之前，NSWindow 会尝试使目标视图成为第一响应者。如果同时有修正键按下，则它根本就不发送消息，而是显示上下文帮助，或者上下文菜单。

模式窗口

通常情况下，应用程序主要根据用户动作发生的地方，将事件分发给所有的窗口。但是，有些时候应用程序会运行在模式窗口的状态下，要求用户在退出窗口之前完成某个任务—比如选择一个文件、输入一个名称、或者甚至是点击一下 OK 按键。模式窗口在 Mac OS X 上是很常见的，包括错误信息对话框，以及打开或打印文档的面板。

在模式窗口的机制中，NSWindow 对象是被动的参与者；模式行为是由应用程序以编程的方式发起和管理的。NSApp 使用正常的事件循环来运行模式窗口，但是将输入限制在特定的窗口或面板上。在循环中，它首先取出事件，如果该事件不满足特定的条件，最重要的是需要与模式窗口相关联，就会被丢弃。

NSApplication 提供了一些运行模式窗口的方法：

- 阻塞方式：应用程序被阻塞，直到用户解除模式窗口。
- 非阻塞方式（模式会话）：应用程序发出模式会话，并在一次事件循环中运行模式窗口。模式会话代码可以继续运行模式窗口，直到满足某些条件为止。

  重要信息：在非阻塞的循环中，您的代码应该在两次运行模式窗口的期间做一些工作。否则会使应用程序处于一个紧密的轮询循环，而不是阻塞循环。

其它 Application Kit 类也提供运行模式窗口和面板的方法。

面板

面板是为应用程序或文档窗口提供支持的辅助窗口。人们常常将它成为对话框。在 Cocoa 中，面板是 NSPanel 或 NSPanel 子类的实例。面板有一些适合作为辅助窗口的特殊行为。它们可能变成键盘焦点窗口，但永远不能充当主窗口。缺省情况下，当应用程序处于不活动状态时，它们会被从屏幕上移走；当应用程序变为活动时，又会被重新显示在屏幕上（警告对话框是这种行为的一个例外）。而且，由于面板的目的是重复使用，在关闭的时候不会释放。您也可以将面板配置为浮动窗口，比如工具窗口，这种窗口的级别位于应用程序其它窗口之上。

视图

NSView 对象（或者简单地说成视图）占有窗口中的一个矩形区域。在 Cocoa 中，视图是 NSView 子类的实例。它们是 Application Kit 中使用最为广泛的对象类型，您在 Cocoa 应用程序中看到的几乎所有对象都是视图。视图是描画和事件处理的前线，因此是最重要的对象类型之一。

您可以把在屏幕上进行视图描画考虑成视图对象本身的可视表示。从非常真实的意义上看，视图是在描画自己。它还提供一个表面，负责接收鼠标、键盘、或者其它输入设备的输入。

进一步阅读：用NSView对象进行描画的概念和任务的详细描述，请参见 *Cocoa描画指南*文档。还有，*Cocoa视图编程指南*文档中也描述了与视图操作有关的各种任务。

本部分包含如下内容：

视图的变体

NSView 是负责定义基本描画、事件处理、和应用程序打印架构的类。NSView 自身并不描画内容或响应用户事件，因此您通常不和 NSView 的直接实例进行交互。相反，您使用的是 NSView 定制子类的实例。这种定制子类继承自 NSView 类，并对很多方法进行重载，Application Kit 会自动调用这些方法。

如果您考察Application Kit的类层次（图 1-9），就会注意到很多类都是直接或间接地继承自NSView类。这些类可以分为下面几个范畴：

- **控件:** 用户通过操作控件类视图（比如点击或拖拽），向应用程序表明自己的选择。按键、滑块、文本框、和步进器都是控件。控件通常（但并不总是）和单元(cell)对象一起工作。单元对象不是NSView的子类，有关控件的详细信息请参见 "控件和菜单"部分。
- **容器视图:** 有些视图用于包围和展示其它视图或更多元数据。这种视图可能用于为数据提供编辑支持，或者更有效地展示某个用户界面。这种视图包括 NSTextView、NSImageView、NSBox、NSSplitView、和 NSTabView 对象。
- **复合视图:** 有些视图是由其它视图组成的。当您在 Cocoa 应用程序中看到一个文本视图时，它不仅仅包含一个 NSTextView 对象，而且包含一个 NSClipView 对象和一个 NSScrollView 对象（该对象又包含一个 NSScroller 对象）。另一个例子是表视图（即 NSTableView 类的实例），它是由表头和列对象构成的（列对象不是视图）。
- **包装视图:** 还有少数视图是用于为 Mac OS X 技术提供 Cocoa"宿主"的。这些对象的例如 NSOpenGLView 和 NSMovieView 的实例。

这些分类中的某些对象是重叠的。举例来说，NSTableView 对象是一个复合对象，但也是个控件。

视图的层次结构

您可能还记得我们早些时候关于窗口的讨论（"窗口"部分），每个视图对象和显示该视图的窗口是关联在一起的。一个窗口中的所有视图对象可以通过**视图层次**图连在一起。每个视图都有一个其它视图作为自己的**超视图**，每个视图也可以有任意数量的**子视图**。视图层次结构的顶部是窗口的内容视图，它没有公开的超视图（但确实有一个私有的超视图）。视图层次在视觉上的关键特征就是它们之间的包围关系：即子视图包围在超视图中，其位置也是相对于超视图的。图 6-11 展示了这种包围关系。

图 6-11 视图的层次结构

将视图对象组织在一个层次结构中对于视图的描画和事件处理都是有好处的。在描画方面有三个好处：

- 这使我们可以在其它视图之外构造复杂视图。比如，一个图形化的键盘可以由一个容器视图和多个分立的子视图组成，每个子视图代表一个按键。
- 它也使得每个视图有自己的坐标系统，便于描画。视图的位置是相对于其超视图的坐标系统的，因此当一个视图对象被移动或者其坐标系统被变换时，它的所有子视图都会同样被移动或变换。由于视图是在自己的坐标系统中进行描画的，所以不管它的位置在那里，或者它的超视图被移动到屏幕上的什么位置上，它的描画指令都可以保持不变。
- 它还可以用于设置在同一个描画过程中被渲染的视图的分层顺序（参见 "显示一个视图"部分）。

请注意：请不要混淆视图*实例*层次和视图*类*继承层次这两个不同的概念。视图继承层次是基于共享的属性、接口、和行为的不同类之间的关系。而视图实例层次则是基于包围关系的特定视图实例之间的关系。

视图层次（现在指视图实例层次）在事件处理中发挥重要的作用，因为它是响应者链的主要部分。有关响应者链的更多信息请参见 "响应者和响应者链"部分。

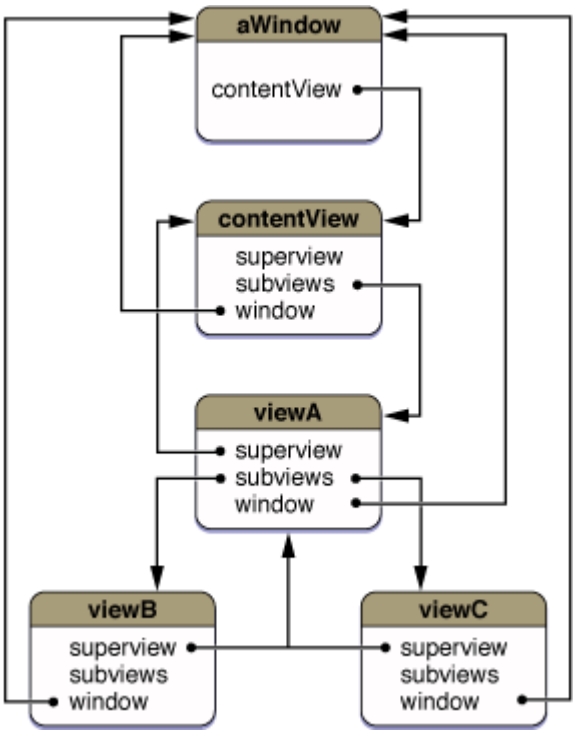
视图层次是动态的：在应用程序运行的同时，您可以对视图进行重新排列，或者进行添加和移除操作。您可以将视图从一个窗口移动到另一个窗口中，也可以在特定层次移动视图。

NSView 有三个关系属性，可以帮助您定位视图在视图层次上的位置：

- window—视图所在的窗口（NSWindow 对象）
- superview—在视图层次中位于上面的视图
- subviews—视图包含的视图列表（这个列表可以是零个或多个视图）

图 6-12 用图表展示窗口对象及其视图层次的关系，同时反映这些属性的关系。

图 6-12 视图层次中对象之间的关系



视图的几何尺寸和坐标

视图的几何尺寸很大程度是由与视图相关联的两个矩形，即视图的边框(frame)和边界(bound)来定义的。虽然这两个矩形包围的是同样的区域，但其目的是不同的。它们一起定义了视图的位置和尺寸，以及视图描画和进行事件响应所在的坐标系统。

请注意: 这些矩形描述的尺寸和位置通过浮点数来表达。

边框

边框矩形定义了视图的可供描画区域。如果您将视图考虑为窗口中的一个矩形区域，则边框指定了这个矩形的尺寸和在窗口中的位置。视图只能在其边框内进行描画，在缺省情况下，Application Kit 会强制将视图上描画的内容按视图边框进行裁剪。

如图 6-13 所示，视图的边框矩形通常位于其超视图的边框矩形中。但这并不是必须的，视图的边框矩形可以扩展到其超视图边框的外部，只是其描画内容会按照包含该视图的上级视图链的边框进行裁剪。只有位于所有超视图的边框矩形之内的视图部分，才能被显示在屏幕上。

图 6-13 层次视图

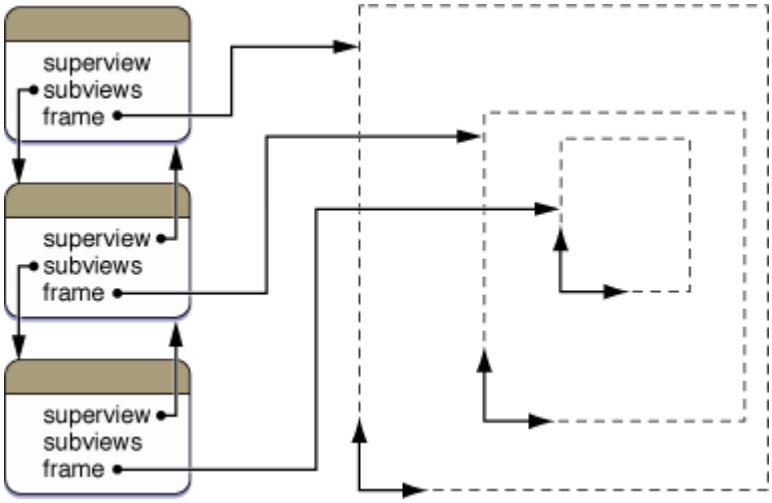
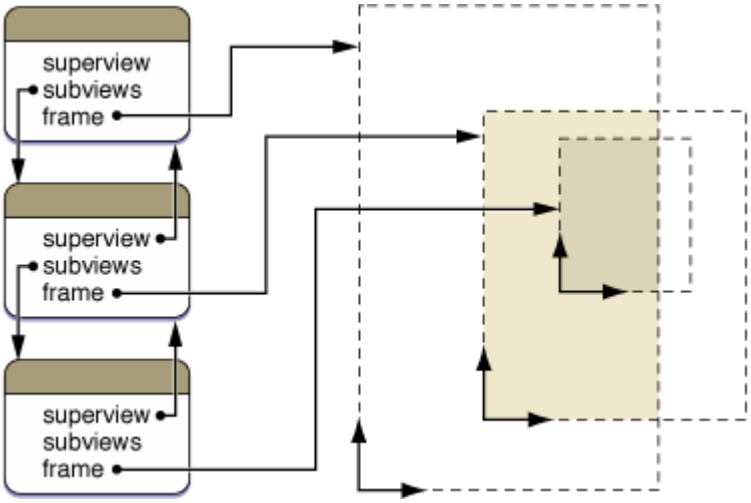


图 6-14 也显示了三个在层次结构上有关系的视图。但是在这种情况下，中间的那个视图部分位于其超视图的边框矩形之外。对于最底层的视图，虽然整个视图都位于其直接超视图的内部，但还是有一部分位于其更上级视图的外面，因此只有带颜色的部分是可见的。

图 6-14　被超视图裁剪之后的视图

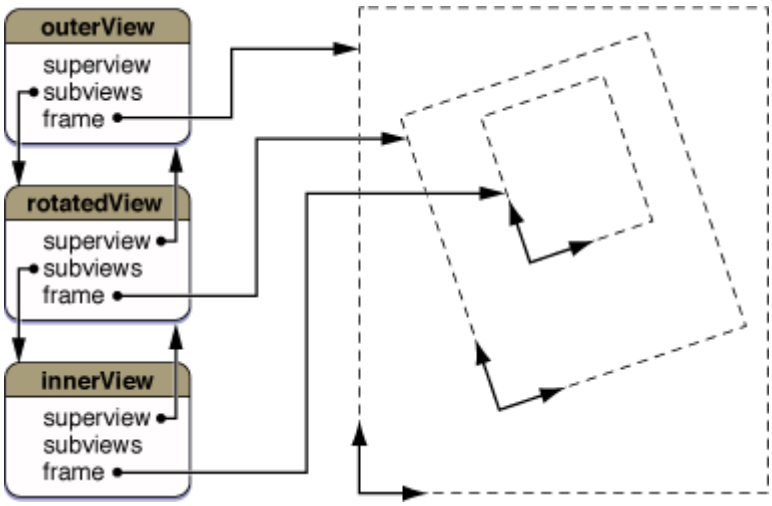


某些视图可能包含比窗口的显示空间更多的资料－比如一个很长的文档。这样的视图可能被处理为另一个较小的、只能显示部分文档的视图的子视图。这个较小的视图被称为剪裁视图（是 NSClipView 类的一个实例）。在剪裁视图包含的滚动视图（NSScrollView）合滚动条（NSScroller）的协助下，用户可以控制控制剪裁视图中文档的可视部分。当子视图移动到不同位置的时候，文档的不同部分就会被滚动到该视图中。

您可以通过重置视图边框的原点来移动视图的位置，通过改变边框矩形的尺寸来改变视图的大小。由于这些值都是基于父视图的坐标系统的，如果父视图的坐标改变了，视图在屏幕中的位置和大小也会跟着发生变化。

您可以围绕边框的原点来旋转视图。旋转并不影响视图的形状和尺寸，即使视图被旋转到边框的边不再和父视图的x轴和y轴平行的某个角度，它仍然是个矩形。不管边框旋转的角度如何，边框的原点都保持在同样的位置上。被旋转的视图和它的子视图的位置关系保持不变，因此相对于旋转视图的父视图，子视图也旋转了。图 6-15 展示了与 图 6-13 相同的三个视图，只是视图层次中间的视图发生了旋转。

图 6-15 旋转后的视图及其子视图



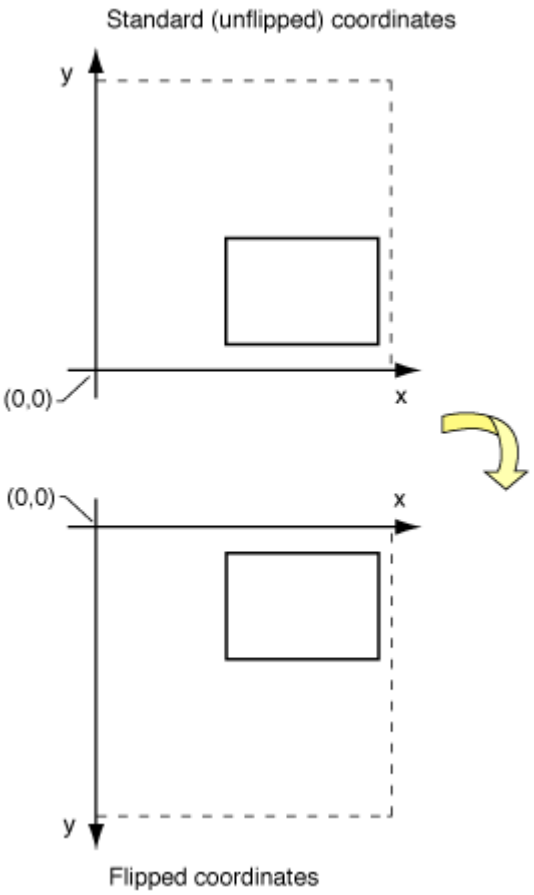
边界

视图的边框是定义视图在其父视图中位置和尺寸的手段，在视图的描画过程中很少用到边框。视图在其本地坐标系统中执行所有的描画和事件处理任务，这个坐标系统由视图的边界矩形来定义。

视图的边界矩形定义了视图的本地坐标系是如何映射到视图的区域上的。虽然它使用的本地坐标系，但描述的物理区域和边框矩形完成一样。缺省情况下，视图的边界矩形的尺寸和其边框是相等的，而其原点的坐标为(0.0, 0.0)。在这种安排下，视图在定位和描画其内容时使用的坐标值都是正的。

然而，如果视图被翻转了，情况就会发生变化。一个视图可以将其坐标系进行翻转，使其描画的原点位于左上角，沿 y 轴向下为正值。图 6-16 显示了翻转之后的坐标系。翻转后的视图对于描画某些语言的文本特别有用，比如英语文本，因为这种语言的文档是从左上角开始向右向下发展的。

图 6-16 翻转一个视图



视图通常使用边界矩形，以便确保那些永远不会渲染在屏幕上的内容不被描画。由于落在任何上级视图外部的内容都会被裁剪，所以边界矩形也仅对那些不会被滚动的、总是显示在其上级视图的边框矩形中的视图是可靠的。NSView 类中提供了其它一些确定描画位置的编程方法，但在所有与描画有关的计算中，边界矩形总是需要的。

描画坐标

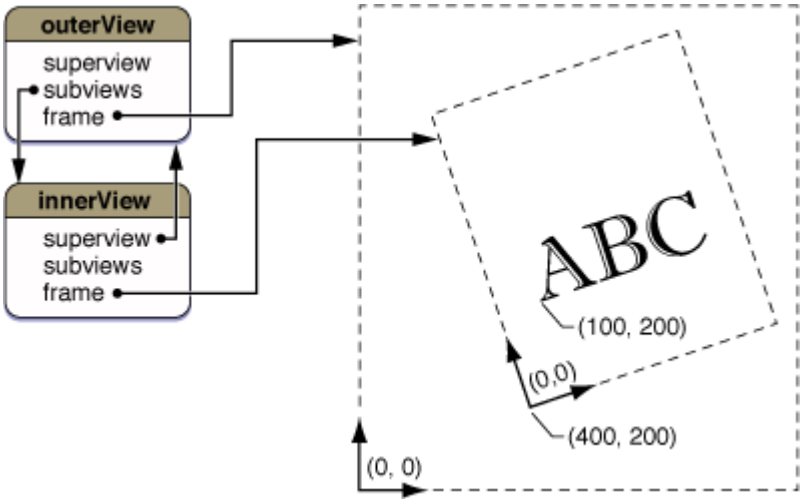
边界矩形提供了视图的描画坐标。在视图对象描画自身的内容之前，其坐标系会被设置为应用程序当前的坐标系（详细信息请参见 "视图如何被描画" 部分）。您可能可以回想起来，除了下面这些区别，一个视图的缺省坐标系和其父视图是一样的：

- 在父视图中定位视图边框的点是该视图边界矩形的坐标原点(0.0, 0.0)，因此也是视图描画坐标的原点。
- 如果视图的边框被旋转，其描画坐标系也跟着旋转；且两者的 x 轴和 y 轴保持平行。

然而，如果边界经过了转换处理，上面的细节可能会发生变化。

图 6-17 展示了一个视图的缺省坐标系与其父视图坐标系之间的关系。图中的 innerView 对象位于父视图坐标系的(400.0, 200.0)点上。如图中的例子，当 innerView 在(100.0, 200.0)点上描画文本时，这个点相对于其自身的原点，而不是其父视图的原点。即使 innerView 发生了如图所示的旋转，边界的轴和边框旋转，但视图描画坐标的原点保持不变。

图 6-17 视图坐标系与其父视图坐标系之间的关系



一个视图可以通过下面的几种方式修改其缺省的本地坐标系：

- 它可以将原点转换为除了边框原点之外的另一个点。
- 它可以将边界轴的尺寸单位缩放为不同于其父视图的值。
- 它可以将边界矩形围绕者边界原点进行旋转，使其不再紧挨着边框的轴。

这些修改改变了视图描画的坐标系，也可能影响描画结果的外观，但不会改变描画发生的区域。换句话说，它们不会影响视图的边框矩形。

视图如何被描画

在 Cocoa 应用程序中，视图是负责描画窗口内容的基本对象。其它一些 Application Kit 对象也能描画（比如 NSCell、NSBezierPath、和 NSAttributedString 对象），但它们都需要一个 NSView 对象作为"宿主"，以提供描画的表面及对描画进行协调。本文的下面部分将对 Application Kit 如何协调视图的描画给出一个高级别的概述。

显示一个视图

通常情况下，视图并不是在任何时候都随时进行描画。它必须首先被显式标志为无效（或者“变脏”），才有被显示的需要。然后，视图的描画可能马上发生，或者如果NSWindow的auto-display特性被打开，描画可能被延迟到主事件循环当前周期的最后。窗口的auto-display特性缺省是打开的（auto-display在 "主事件循环" 和 "窗口和描画"部分中描述)。
您可以通过 NSView 的显示方法（之所以这样称呼是因为这些方法的名称中都包含了"display"这个词）来使视图或视图的一部分立即被重画。这些方法在很多方面都有区别，但是都会触发 Application Kit 发生如下操作：

1. 锁定无效视图的焦点（在 "锁定焦点"部分中进行描述）
2. 调用视图的drawRect: 方法（在 "drawRect:方法中发生了什么"部分中进行描述）
3. 在描画过程的最后，刷新与视图相关联的窗口（如果该窗口的后备存储是双缓冲的话）

在大多数情况下，我们并不推荐立即显示视图，而是推荐使用 auto-display 机制。在事件循环中将视图（或部分视图）标识为需要显示之后，与视图相关联的 NSWindow 会将这些做过标识的视图收集起来，放在一个列表中，并根据其在视图层次结构中的位置进行排序，在层次结构中处于最上面的视图排在列表的最

前面；在事件循环的最后，它会在一个描画过程中递归地处理这个列表，依次锁定每个视图的焦点，让视图自行描画标识为需要显示的整个或部分区域；当所有视图都描画完成后，窗口（如果有缓冲的话）就会被刷新。

Application Kit 可能会请求应用程序描画被显式标识为需要重新显示之外的其它视图或视图区域，它可能自行认定需要额外的描画才能完全更新指定的窗口区域。那是因为视图的描画中有一个重要的方面，即视图的不透明性。一个视图不一定需要描画其表面的每一位像素，但是如果希望这样做的话，需要将自己声明为不透明（通过实现 isOpaque 方法并返回 YES 来实现）。当您将一个视图标识为需要显示时，Application Kit 会检查改视图的不透明性，如果它不是不透明的（也就是说，是部分透明的），Application Kit 会沿着视图层次结构向上检查，直到找到一个不透明的父视图；然后，计算该不透明的视图中被最初考察的视图覆盖的部分；最后，从这个不透明的视图开始描画，沿着视图层次结构一直向前，一直到最初被标识为无效的视图。如果您希望 Application Kit 在描画一个视图之前，不要寻找第一个不透明的父视图，则可以使用下面的几个“显示时忽略不透明性”的方法（在表 6-1 和表 6-3 中列出）。
您也可以将视图或其子视图的一部分标识为需要显示，然后马上对其进行重画，而不是等待 auto-display 机制来触发。NSView 中具有这种特性的显示方法的名称都以 displayIfNeeded... 最为开头。这些方法虽然是立即显示，但也比向孤立的视图发送 display 或 displayRect: 方法效率更高。

表 6-1 列出了 NSView 的显示方法，用于立即显示单独的视图或视图区域的。

表 6-1 NSView 的显示方法－立即显示

显示区域和不透明属性	显示方法
整个视图	display
部分视图	displayRect:
整个视图，忽略不透明性	无
部分视图，忽略不透明性	displayRectIgnoringOpacity:

表 6-2 列出的方法用于将视图或视图区域标识为需要使用 auto-display 特性进行重画。

表 6-2 NSView 的显示方法－延迟显示

显示区域	显示方法
整个视图	setNeedsDisplay:
部分视图	setNeedsDisplayInRect:

表 6-3 列出的方法可以强制地使那些通过表 6-2 列举的方法标识为无效的视图（或者部分视图）立即显示。

表 6-3 NSView 的显示方法－立即显示标识为无效的视图

显示区域和不透明性	显示方法
整个视图	displayIfNeeded
部分视图	displayIfNeededInRect:
整个视图，忽略不透明性	displayIfNeededIgnoringOpacity

部分视图，忽略不透明性	displayIfNeededInRectIgnoringOpacity:

值得再次强调的是，立即显示视图在效率上比使用 auto-display 特性要差，而立即显示标识为无效的视图则在某种程度上处于两者之间。此外，将视图区域标识为无效通常会比将整个视图标识为无效效率更高。显示方法对于应用程序的开发是非常便利的。一个视图可以锁定焦点，描画自身，然后解锁焦点。但是这种做法只在特定的场景下推荐，比如在定时器的回调函数中对视图内容进行动画显示。一般地说，您不应该绕过 Application Kit 的显示机制。

锁定焦点

当 Application Kit－或者您的代码－锁定了视图的焦点，NSView 会完成下面的这些步骤：

- 将父视图的坐标系统转换到当前视图的坐标系统，并将它作为应用程序当前的坐标系统。
- 构建一个剪裁路径，以定义一个矩形区域，使在这个矩形区域外不能进行描画。
- 激活当前图形状态的其它参数，建立视图描画环境。

您可以通过向视图发送 lockFocus（或者相关的 lockFocus...）消息来锁定它的焦点。在焦点视图的描画完成之后，您可以通过发送 unlockFocus 消息来解锁。当您向视图发送 drawRect:消息时，Application Kit 会自动锁定和解锁焦点。

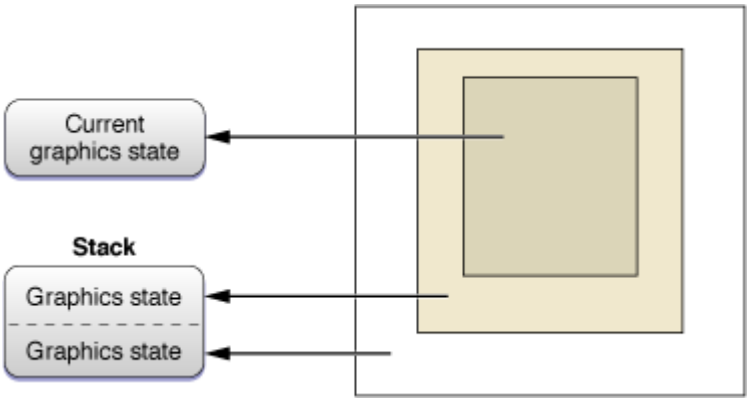
焦点视图是指当前描画的视图。每个窗口都有一个窗口图形上下文，用于定义描画的窗口服务器目标。这个图形上下文包含一个或更多的图形状态，每个状态都将描画分隔到一个特定的视图上。当前图形状态包含有效的坐标系统和为最近的焦点视图设置的其它描画参数。

图形状态中包含当前的转换矩阵，它是一个数学结构，用于将一个点从一个坐标空间射到另一个坐标空间，其中的操作包括旋转、缩放、平移。图形状态还定义了当前裁剪区域以及在渲染图像时需要的其它参数，包括如下参数：

- 填充或划线操作的颜色
- Alpha 值（透明）
- 线的属性，包括宽度、连接、线帽、破折号、和尖角限量
- 抗锯齿值
- 当前颜色空间
- 文本属性：字体、字体大小、字符间隔
- 混合模式

在其它视图拥有焦点时对视图进行焦点锁定也是可能的。事实上，Application Kit 在描画过程中也是这么做的。由于每个视图都跟踪其自己的坐标系统，将它作为其父视图坐标系统的变体，视图通常在其父视图之后才进入焦点。在一组被标识为无效的视图中，Application Kit 首先锁住最上层视图的焦点，并沿着视图层次结果向下，嵌套调用一系列 lockFocus 方法。在每个后续视图的焦点被锁定的同时，当前图形状态被存储在由图形上下文维护的堆栈中（参见图 6-18）。而在视图焦点被解锁时，堆栈顶端的图形状态被“弹出”，并恢复为当前图形状态。

图 6-18 嵌套的焦点视图和图形状态堆栈



drawRect:方法中发生了什么？

Application Kit 在锁定视图焦点之后，随即向它发送 drawRect:消息。视图类负责实现这个方法，并在这个方法中进行视图的描画。在调用 drawRect:方法时会传入一个矩形，定义在视图坐标系中要描画的区域。这个矩形可能对应于视图的边界，也可能不是。它可能是视图中所有标识为无效的矩形区域的并集，或者这个并集的超集。

向窗口服务器发送描画指令和数据是有代价的，应该尽可能避免这些开销，特别是避免那些最终不可见的描画。Application Kit 的一个主要的优化是限制描画所在的区域，特别是当描画很复杂的时候。视图可以选择描画整个区域（这是效率最低的做法），或者仅描画传入矩形定义的区域。对于视图来说，一个潜在的更为有效的方法是得到无效区域的列表（通过 NSView 的 getRectsBeingDrawn:count:方法），并选择性地依次描画这些区域。

在 drawRect:方法的实现中，视图类调用不同的函数和方法来指定传递给窗口服务器的指令和数据。Application Kit 提供了下面这些高级别的描画函数和方法：

- 在矩形中执行绘画、填充、擦除、和执行其它操作的描画函数（在 NSGraphics.h 文件中声明）
- 用巴塞尔路径（NSBezierPath 类）构造线和形状的方法
- 创建和应用远交变换的方法，用于执行平移、缩放、和旋转操作（NSAffineTransform 类）
- 颜色和颜色空间方法（NSColor 和 NSColorSpace）
- 创建和合成图像的方法（NSImage 和各种图像表示类）
- 描画文本的方法（NSString 和 NSAttributedString 类）

Application Kit 使用 Core Graphics (Quartz)的函数和类型来实现这些方法和函数。在描画视图的时候，视图类也可以使用这些 Core Graphics 函数。这些 Quartz 客户端连接库的函数可以直接映射为窗口服务器的渲染操作，最终生成栅格（位图）图像，对于窗口内描画来说，这些图像是窗口后备存储的一部分。

线程和描画

视图的描画不一定需要发生在主线程；应用程序的每一个线程都有能力锁定视图的焦点并进行描画。然而有下面这些限制：

- NSView 对象属性（比如它的边框矩形）的改变应该只发生在主线程。

- 当NSView的显示方法被调用时，Application Kit会设置一个锁，以便在接收视图的窗口中描画；在显示方法返回之前，您不能年执行任何定制的描画。这意味着，同时只有一个线程可以在指定窗口中进行描画。

视图和打印

视图是Cocoa打印架构的基础。他们提供了打印的内容，就像提供屏幕的显示内容一样。打印和显示的一般过程是一样的：Application Kit锁定视图的焦点，调用其drawRect:方法；视图则描画需要打印的内容，然后焦点被解锁。您可以调用视图的print:方法来打印视图本身的内容。

视图和事件

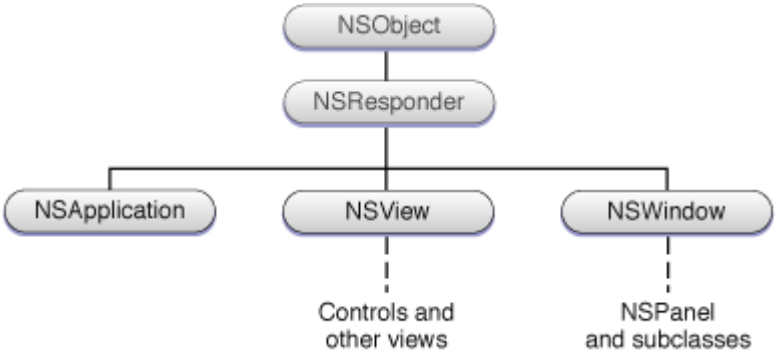
在应用程序中，视图直接响应大部分诸如鼠标点击和按键按下这样的用户事件。视图几乎总是用户事件发生所在的表面。因此，对事件消息进行处理的第一个机会就被赋以给这些第一线的对象。

如在"窗口和事件处理"部分讨论的那样，窗口会将用户事件转送给在视图层次中应该接收该事件的视图。鼠标事件会被发送给事件发生时光标下面的视图。键盘事件则发送给第一响应者，它通常是拥有键盘焦点的视图对象。为了接收键盘事件，视图必须声明接收第一响应者状态（也就是重载NSResponder的acceptsFirstResponder方法，并返回YES）。
视图是响应者对象，也就是说它们继承了NSResponder类的编程接口。事件消息会调用这个类声明的方法，像mouseDown:、mouseMoved:、和keyUp:都是这样的消息。为了处理事件，视图类必须实现恰当的NSResponder方法，在自己的实现中考察传入的NSEvent对象（它封装了事件的有关信息）并进行相应的处理。如果视图没有对事件进行处理，则Cocoa会将机会传给响应者链上的下一个响应者－通常是该视图的父视图。"响应者和响应者链"部分描述了响应者的信息，以及事件如何在响应者链上传递。
视图在动作消息的处理上也发挥主要作用。动作消息通常是在事件消息被发送给NSControl对象后产生的。控件对象在处理事件消息时，会向目标对象发送一个动作消息。如果没有指定目标对象，应用程序会沿着响应者链向上搜索可以响应该动作消息的对象。更多有关NSControl对象及其使用的NSCell对象，请参见"控件和菜单"部分。

响应者和响应者链

核心的应用程序对象－NSApplication、NSWindow、和NSView－都是响应者，它们是NSResponder（参见图6-19）的直接或间接子类的实例。这个抽象类定义了能够响应事件的对象的接口和期望的行为。NSResponder的子类完全或部分实现了这个行为。

图6-19 NSResponder及其直接子类



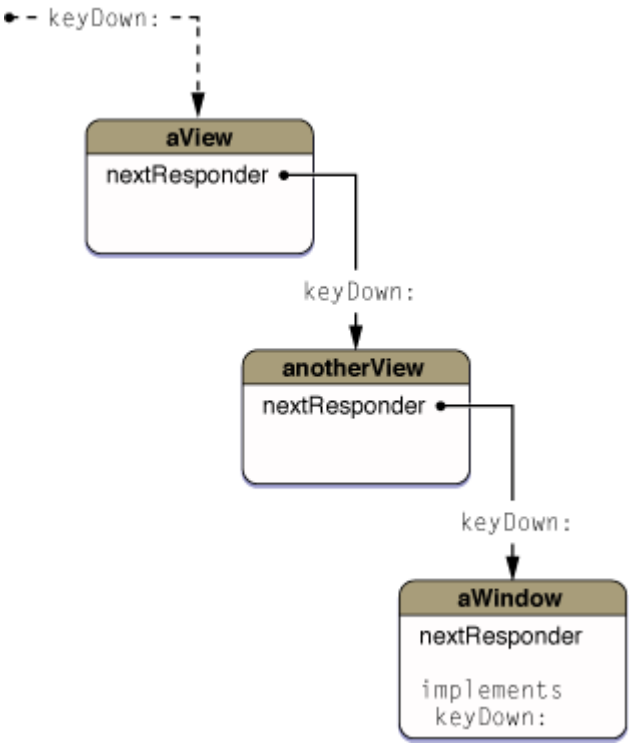
请注意：还有其它两个Application Kit类直接继承自NSResponder：NSWindowController 和NSDrawer。虽然NSWindowController是Cocoa文档架构的一部分（参见 "其它Cocoa架构"部分的描述），这两个类都不是核心Cocoa应用程序架构的中心。

NSResponder 类为核心应用程序架构的三个主要模式或机制定义了一个接口：

- 它声明了一些处理**事件消息**（也就是源自用户事件的消息，比如象鼠标点击或按键按下这样的事件）的方法。
- 它声明了数十个处理**动作消息**的方法，它们和标准的键绑定（比如那些在文本内部移动插入点的绑定）密切相关。动作消息会被派发到目标对象；如果目标没有被指定，应用程序会负责检索合适的响应者。
- 它定义了一套在应用程序中指派和管理响应者的方法。这些响应者组成了我们所知道的**响应者链**－即一系列响应者，事件或动作消息在它们之间传递，直到找到能够对它们进行处理的对象。

响应者链是 Application Kit 事件处理架构的中心机制。它由一系列链接在一起的响应者对象组成，事件或者动作消息可以沿着这些对象进行传递。如图 6-20 显示的那样，如果一个响应者对象不能处理某个事件或动作－也就是说，它不响应那个消息，或者不认识那个事件，则将该消息重新发送给链中的下一个响应者。消息沿着响应者链向上、向更高级别的对象传递，直到最终被处理（如果最终还是没有被处理，就会被抛弃）。

图 6-20 响应者链



当 Application Kit 在应用程序中构造对象时，会为每个窗口建立响应者链。响应者链中的基本对象是 NSWindow 对象及其视图层次。在视图层次中级别较低的视图将比级别更高的视图优先获得处理事件或动作消息的机会。NSWindow 中保有一个**第一响应者**的引用，它通常是当前窗口中处于选择状态的视图，窗口通常把响应消息的机会首先给它。对于事件消息，响应者链通常以发生事件的窗口对应的 NSWindow 对象作为结束，虽然其它对象也可以作为下一个响应者被加入到 NSWindow 对象的后面。

对于动作消息，响应者链则比较复杂。有以下两个因素决定了动作消息的响应者链：

- 如果应用程序当前既有主窗口，也有键盘焦点窗口，则两个窗口的响应者链都会参与，其中键盘焦点窗口的响应者链首先获得处理动作的机会。在每一个窗口链的最后，Cocoa 会给 NSWindow 的委托对象响应动作的机会；在合并的响应者链的最后是 NSApp 以及它的委托对象。
- 应用程序的类型－是简单的基于文档的程序，还是使用窗口控制器的程序－这决定链中的响应者对象的类型和位置。

NSResponder 类也包含有错误表示和恢复、消息派发、应用程序帮助、以及实现其它功能的方法。

进一步阅读：如果需要了解响应者和响应者链的更多信息，请参见"关于响应者链"部分。

控件和菜单

您在应用程序中看到的很多对象的设计目的，是让您对其操作，以表达您的意图。这些对象包括按键、检查框、滑块、表视图、文件系统浏览器、和菜单（包括应用程序菜单和弹出式菜单）。在 Cocoa 中有两类用于选择的对象：控件和菜单，实现这两类对象的架构是类似的。在这些架构中，不同类型的对象（包括 NSView 对象）互相协作，使用户的选择或意图得以揭示。

本部分包含如下主要内容：

控件与单元架构

控件是一种用户界面对象，用于响应诸如鼠标点击这样的用户事件，响应的方式是向应用程序的其它对象发送消息。常见的控件类型有按键、滑块、和文本框（这种控件通常在用户按下回车键时发送消息）。其它不那么显而易见的控件有表格视图、数据浏览器、和颜色选择控件。

控件是NSControl抽象类的子类的实例，通常管理一个或多个单元，单元是另一个抽象类NSCell的子类的实例。如果您考察Application Kit的类层次（图 1-9），会注意到NSView的子类NSControl是很多控件类的根，如NSButton、NSStepper、NSTextField等等；而在类层次的另一个完全不同的位置（在NSObject下面），NSCell是很多单元类的起始点，这些类中绝大多数都和控件类相对应，如NSButtonCell、NSStepperCell、NSTextFieldCell等等。

请注意：下面这两个控件类和单元类完全分离：NSScroller 和 NSColorWell。这两个类的实例和其它控件一样，都是让用户进行选择的界面元素，但是它们的行为和其它非控件视图一样，即自行负责描画，且在响应用户事件时没有使用单元。

管理多个单元的控件

Application Kit 的大多数控件都只管理一个单元。在这种情况下，控件让它管理的单元做几乎所有的实际工作，将发给自己的消息转发给它的单元。但是也有一些控件管理多个单元。Application Kit 采取两种一般的方法来管理控件中的多个单元：

- 将一个单元实例用作描画的模板。每当控件需要描画一个单元的时候，就通过这个实例复制每个单元表示，然后只要改变单元的内容就可以了。NSTableView 和 NSOutlineView 类在描画表视图的单元和大纲视图（outlie view）的列时，都使用这种方法。

- 用一个单元实例表示控件中的每个单元区域。NSMatrix 类在请求其单元实例进行自我描画时采用这种方法。NSBrowser 和 NSForm 对象的工作方式类似：它们的单元也是一个单独的实例。

  一个 NSMatrix 对象（或者说是矩阵）可以管理大多数单元类型。它将单元安排在任意维数的栅格中。当您在 Interface Builder 中构建一个单元矩阵的时候，那些单元都是给定单元原型的拷贝。然而，您可以在程序中将单元设置为不同 NSCell 子类的实例。NSBrowser 和 NSForm 控件在管理的单元类型方面有更多的限制。

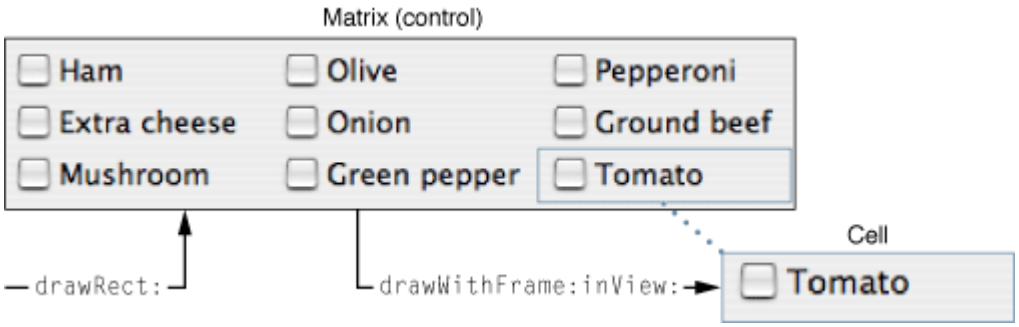
单一单元控件可能看起来象是一个多余的设计，但是这种设计的价值在于单元的使用，而不是控件。具体的解释请参见 "控件-单元架构的基本原理"部分。

控件如何管理单元

控件是完全功能的 NSView 对象，它们可以被标识为需要显示，并负责描画自身所在的区域；也可以响应用户事件，属于响应者链中的对象。在运行时，控件的初始行为和其它视图是一样的，Application Kit 通过调用它的 drawRect:方法来使其描画自己的外观。如果有事件消息被派发到响应者链上，且链中有控件实现了合适的 NSResponder 方法，则该控件就可能获得处理事件的机会。然而，控件不是自己进行描画和事件处理，而是将这些工作传递给它管理的单元。如果有多个单元，则控件负责协调这些单元的行为。

如图 6-21 所示，控件通常在其 drawRect:方法中向它管理的单元发送 drawWithFrame:inView:消息，要求它描画位于传入矩形中的部分。由于焦点已经被锁定在控件上了，单元可以用其“宿主”视图的皮肤进行描画。在 mouseDown:事件方法的实现中，控件会向发生事件的单元发送一个 trackMouse:inRect:ofView:untilMouseUp:消息。典型情况下，该单元会跟踪其边界内部的鼠标事件，直到鼠标移除或者接收到 mouseUp:事件，然后进行正确的响应（其它事件类型也以类似的方式进行处理）。

图 6-21 控件协调单元的描画



单元在描画自己的外观时，必须包括两个方面。首先是单元在一般意义上的外观，这在同一个类的各个单元中是一致的。这种外观包括单元的形状、背景色、风格、和单元状态的指示（比如表示单元选择状态的检查框，或者指示单元不可用的灰色文本）。第二个方面是单元的个性化内容。在内容方面，单元通常有一个标题（即一个字符串）、一副图像、或者在某些情况下两者都有，标题还可能有指定的字体（某些单元有自己定制的内容，而不是使用图像或标题，比如 NSSliderCell）。

为了描述内容，每个单元都有一个对象值和可能的表示对象。对象值必须是一个可以格式化为字符串并因此可以显示为单元标题的对象－比如一个封装了float值的NSNumber对象，虽然本身不能被显示，但可以和单元相关联；再比如一个标题为“红色”的按键，可以有一个类型为NSColor的表示对象。有关对象值和表示对象的更多信息，请参见 “表示对象”部分。

单元的另一个个性化内容是它们为动作消息封装的信息。目标－动作机制（在 "目标-动作机制"部分中讨论）使控件可以在用户激活的时候（比如点击按键或在文本框中输入回车键）将消息发送给指定的对象。动作是一个选择器，标识要被调用的方法；目标则是指定的对象（目标可以为nil，告诉应用程序需要在响应者链中检索能够处理相应消息的对象，参见 "响应者和响应者链"部分）。NSActionCell是一个抽象类，定义了存储和取得动作选择器及目标对象引用的接口，大多数NSCell子类都继承自NSActionCell类。

控件-单元架构的基本原理

Application Kit 的控件-单元架构有很深的历史根源，可以追溯到 NeXTSTEP 的早期。但是，这个架构的必要性乍看起来很令人费解。为什么让控件完全管理单元？为什么控件不能自行完成所需要的工作？

控件没有理由不能自行完成需要的工作，而且如果您正在设计一个定制控件，减少使用单元也确实是正确的方法。但是控件-单元架构可以带来一些好处：

- 这个架构扩展了控件的有效性。诸如表视图或矩阵这样的控件可以有效地管理很多不同类型的单元，而不需要了解每种类型的具体信息。像 NSButtonCell 对象这样的单元可能被设计为配合 NSButton 控件工作的，但是它也可以和表视图和矩阵对象协同工作。

  单元是一种抽象，用于简化视图中不同类型的图形对象的管理。它使我们得到一种插件设计，使控件可以作为不同图形对象的宿主，这些图形对象可以有自己的标识，包括目标和动作信息。

- 控件-单元架构使控件和单元之间可以建立紧密的耦合关系，这种关系比视图和子视图集合之间的关系更为紧密。子视图很大程度上是独立于其父视图之内的；而控件可以更好地充当各个单元的协调者。举例来说，在收音机按键的矩阵中，控件可以确保在任何时候只有一个按键被选择。

- NSTableView 描画单元区域的模型是将大量的单元实例作为描画的“橡皮图章”进行重用，这种模型是很有效的，特别是必须管理没有潜在数量限制的子区域的时候。

- 即使对于那些不适合使用单元实例作为描画模板的场合，使用单元通常也比使用子视图更具性能优势。无论在内存开销还是计算方面，视图都是比较重量级的对象。例如，跟踪和在视图层次中传递失效的视图区域可能需要重大的开销。

和使用子视图相比，使用单元进行子区域的描画也有一些折衷的地方。由于没有视图失效的机制，控件必须自行计算每个单元需要描画什么。但是由于视图是用于一般目的的对象，而控件是具有专门目的的，通常可以更有效地完成所需的计算。

请注意：有关控件、单元、和使其协同工作的架构的更多信息，请参见 *Coca 的控件和单元编程主题*部分。

菜单的特征及架构

除了控件和单元，用户还可以通过菜单来向应用程序（以及操作系统本身）指示自己的意图。菜单是一个用简练词语表示的选项（或者叫菜单项）列表，菜单项可能带有嵌套菜单（称为子菜单）。用户通常通过点击（虽然其它方法也是支持的）来选择一个菜单项，之后 Cocoa 就会发出一个命令，使应用程序或操作系统进行相应的动作。

Cocoa 支持不同类型的菜单，其中最基本的是应用程序菜单，在 Cocoa 中大家都称之为主菜单。在运行时，应用程序菜单包括*本*应用程序菜单（即指示应用程序名称的那个菜单）及其右边的所有菜单，一直到 Help

菜单结束。应用程序菜单和 Apple 菜单及 menu extras 共享一个菜单条，menu extras 是指应用程序菜单右边专用于服务的菜单。Applicadtion Kit 负责自动创建和管理某些专门用于应用程序的菜单，比如服务（Services）菜单，字体（Font）菜单，窗口（Windows）菜单，及帮助（Help）菜单。Cocoa 应用程序管理的其它菜单类型还包括弹出式菜单、上下文菜单、和 dock 菜单。

菜单，特别是菜单项有一些有趣的特征。菜单和菜单项都有标题，菜单的标题是一个字符串，出现在菜单条上。在某些条件下，菜单项的标题左边可能有一个图像，或者直接用图像代替标题。菜单标题可以是一个属性字符串，可以使用不同的字体，甚至是文本附件（这种对象允许图像出现在菜单项内容区域的任何地方）。我们也可以为菜单项分配一个的热键，称为等价键，同时按下修正键（Shift 键除外）和等价键的效果等同于鼠标点击。菜单项可以被激活或禁用，也可以指示打开或关闭状态，当菜单项处于打开状态时，其标题的左边就出现一个选中的记号。

Cocoa 的菜单和菜单项分别是 NSMenu 和 NSMenuItem 类的实例。在 Cocoa 应用程序中，菜单（在一般意义上）基于一个简单的设计，即让 NSMenu 和 NSMenuItem 对象发挥补充的作用。一个 NSMenu 对象负责管理和逐一描画一个菜单项集合，它包含一个代表菜单项集合的 NSMenuItem 对象数组。NSMenuItem 对象封装了菜单项的所有特征数据，但本身不进行描画或事件处理。NSMenu 对象使用每个 NSMenuItem 对象的数据来（在菜单的边界内）描画相应的菜单项、跟踪菜单中各个菜单项的位置、以及当用户选择菜单项时将动作消息发送给目标对象。NSMenu 对象在描画时使用 NSMenuItem 对象的标题和图像；跟踪时使用菜单项的索引；在发送动作消息时，则使用 NSMenuItem 对象中存储的动作选择器和目标对象。

弹出式菜单使用基本的菜单架构。然而由于它们出现在应用程序用户界面的内部，所以有额外的设计。在用户点击之前，弹出式菜单的外观像一个按键对象，这个对象是 NSPopUpButton 类的实例，该类管理着一个 NSPopUpButtonCell 类的实例。换句话说，控件-单元架构也被用于这个弹出式菜单的初始表示中。NSPopUpButtonCell 对象还包含一个 NSMenu 对象及其封装的 NSMenuItem 对象。在用户点击弹出式按键时，这个内嵌菜单就显示出来了。

菜单中的菜单项可以根据当前的上下文被设置为有效。相反，如果菜单项和当前上下文不相关，则可以被禁用。NSMenu 类中包含一个自动激活功能，可以自动执行这个正当性检查。在菜单被显示之前，它会在响应者链中检索可以响应菜单项动作消息的对象；如果不能找到这样的对象，它就禁用这个菜单项。应用程序可以通过实现 NSMenuValidation 非正式协议来精化菜单正当性的控制。

上下文菜单的实现方式类似于弹出式菜单。您可以通过 NSResponder 的 setMenu:方法来将 NSMenu 对象附加到视图上。上下文菜单列出专门用于其关联视图的命令（可以根据上下文进行有效性控制）。当用户按住 Control 键同时点击鼠标左键，或者点击鼠标右键时，这种菜单就被显示。

进一步阅读：*Cocoa 的应用程序菜单和弹出式列表编程主题*对 Cocoa 菜单和菜单项进行更为详细的讨论。

表示对象

单元和菜单项可以有一个表示对象，即一个与之关联的对象。动作消息的目标可以向被点击的菜单项或单元（sender）请求相应的表示对象，然后对取得的对象进行显示、装载、或执行希望的操作。单元或菜单项允许客户访问其表示对象，并负责对该对象进行归档和恢复，但除此之外不做其它用途。

为了理解如何使用表示对象，让我们考察几个例子。考虑一个矩阵，矩阵中有一些单元，用于设置文本视图的背景颜色。这些单元有诸如"浅蓝"、"浅灰"、"粉红"等名字；每个菜单项的表示对象是一个 NSColor 对象，封装了相应颜色的 RGB 值。您的应用程序可以采用下面的方式来使用这个表示对象，如清单 6-1 所示。

清单 6-1 使用表示对象

```
- (void)changeColor:(id)sender {

    NSColor *repObj = [sender representedObject];

    [textView setBackgroundColor:repObj];  // textView is an outlet

}
```

另一个例子是 Info 窗口中的弹出式菜单，用于改变当前显示的设置面板。分配给每个菜单项的表示对象是一个 NSView 对象，视图中含有需要的文本框、控件、等等。

表示对象和控件或单元的对象值不同。表示对象是任意关联的对象，而对象值则是单元或控件显示的内容后面的值。举例来说，对于内容为"05/23/2006"的文本框，其对象值是一个表示其显示内容的 NSDate 或 NSCalendarDate 对象。一个单元或控件的格式器必须能够"理解"对象值（格式器是指一个 NSFormatter 对象）。

表示对象并不严格限制在单元和菜单项，比如 NSRulerMarker 对象也可以有自己的表示对象。您也可以使您自己定制的视图具有表示对象。

Nib文件和其它应用程序资源

到目前为止，本章的讨论主要集中在 Cocoa 的基本应用程序架构，描述一个应用程序的核心对象在运行时如何协同工作，从而为事件处理和内容描画提供便利。但是，现在我们要做一点转变，把焦点转为关注运行着的 Cocoa 应用程序的全局。应用程序在启动时，很少"从无到有"地创建所有的组成对象。这些对象中很多（如果不是绝大多数的话）都以对象图档案的形式存储在应用程序包中。这些对象图可以代表模型对象，用于封装应用程序数据，在用户退出应用程序之前一直有效；也可以是窗口、视图、和其它构成应用程序用户界面的对象的编码表示。应用程序在运行时装载和解档这些对象档案，复活原来的对象。

一个应用程序不仅包括对象和代码，也包括程序包中的其它资源，比如图像和本地化字符串。本部分还将总结 NSBundle 类的实例在定位和装载所有类型的应用程序资源（包括本地化和非本地化的资源）过程中发挥的作用。

本部分包括如下内容：

对象档案

程序中的对象之间存在一种网状的关系。一个对象可以拥有特定的对象或对象集合，可以依赖于其它对象，也可以保有程序中其它对象的引用以向其发送消息。这些相互交织的对象组成的网称为对象图。对象图可以变得非常复杂。

档案是存储对象图的一种手段。它通常以文件的形式存在，但也可以是在进程间传递的流。档案保留对象图中每个对象的标识，以及对象与图中其它对象之间的所有关系。它将每个对象的类型和数据一起进行编码，在解档对象图的时候，每个解码后的对象通常都和最初编码时的对象具有相同的类。对象的实例数据

也被编码，用于重建对象。对象图中的对象之间的关系也会被恢复。结果，一个解档后的对象图应该几乎总是原始对象图的复制品。

请注意：很多Cocoa应用程序都使用档案作为其模型对象的留存方式。然而，Core Data框架（一个Cocoa框架）为对象的留存提供一个更为复杂的机制。更多信息请参见 "其它Cocoa架构"（包含在本文档中）和 *Core Data编程指南*。

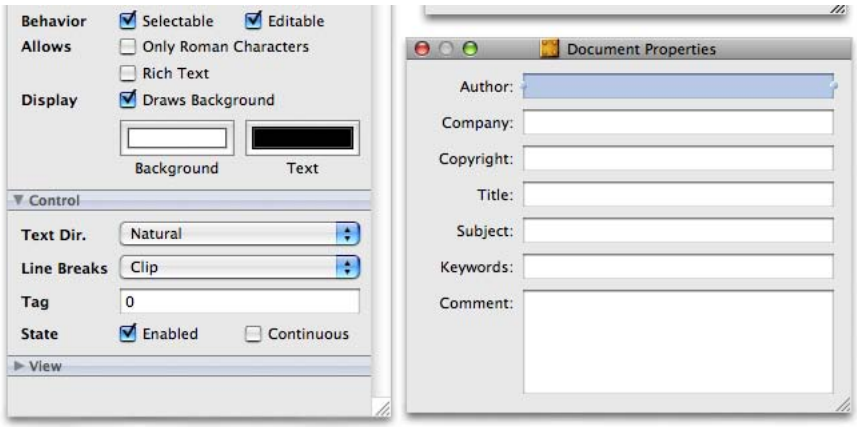
归档是请求对象图中的每个对象将自己编码为一个流的操作；解档则是相反的操作，请求每个对象自行进行解码。这两个操作都由一个发向对象图根对象的消息发起。对象如果希望被保存到档案中，就必须在被请求的时候对自己进行编码；如果希望被恢复，则必须能够自行解码。Cocoa 有两种类型的归档，即顺序归档和键归档，用以反映编解码对象的不同风格。顺序归档要求对象以同样的顺序对其实例数据进行编解码；键归档（这是更为现代的方法）则允许通过键（也就是一个标识字符串）来存储和提取每一片实例数据。一个类如果希望对其实例进行归档，就必须遵循 NSCoding 协议。顺序归档的对象使用 NSCoder 类的编码和解码方法；对于键归档（和解档），则必须使用 NSKeyedArchiver 和 NSKeyedUnarchiver 类的方法。

进一步阅读：如果希望了解更多对象归档和解档的信息，请参见 *Cocoa 的档案和序列化编程指南*。

Nib文件

几乎所有的 Cocoa 开发者都用 Interface Builder 来构建应用程序的用户界面（开发过程不要求使用 Interface Builder，但它是开发工作变得非常容易）。图 6-22 显示了 Interface Builder 窗口的一个典型排布情况，您可能可以从前面的章节回想起来。在创建用户界面时，您可以将诸如文本视图、按键、和表视图这样的控件从选盘中拖出，放到窗口中，然后调整对象的位置和尺寸，设置它们的其它属性。您还可以在这些对象之间建立各种类型的连接－插座连接、目标-动作连接、和绑定。在 Interface Builder 中还可以进行定制模型和控制器类的初始定义，以便建立这些类和其它代理实例的连接。您还可以用 Interface Builder 来指定分配给定制 NSView 子类的占位对象。当您在 Interface Builder 中完成界面设计时，可以将它们保存为一个 **nib 文件**，在工程中作为本地化的资源。在连编工程的时候，nib 文件会被拷贝到应用程序包的本地化（.lproj）文件夹中。

图 6-22 Interface Builder



Nib 文件是描述全部或部分用户界面的对象图的档案。这些图代表用户界面中复杂的关系，包括窗口的视图层次和对象间的各种连接。这些对象图使用 XML 作为描述语言（您永远不应该试图直接编辑 nib 文件中的 XML）。

对象图的根对象出现在nib文件窗口的Instances（实例）面板中（图 6-22 的左下窗口）。在这个例子中，Panel实例就是一个根对象（因为它包含一个视图层次）。nib文件中也可以包含其它的根对象，比如NSController对象（用于绑定）和定制类的代理实例。此外，Cocoa应用程序的每个nib文件窗口都有两种特殊类型的实例：

- **File's Owner。**这个对象拥有 nib 文件并负责管理文件中的对象。File's Owner 对象必须位于 nib 文件的外部，您可以借助 File's Owner 对象来建立 nib 文件内部对象与外部对象之间的连接。
- **First Responder。**这个对象代表响应者链中的第一响应者（参见 "响应者和响应者链"部分）。在目标-动作的连接中，您可以将First Responder对象指定为目标；当某个控件或菜单发出一个动作消息时，应用程序会检索响应者链（从第一响应者开始），直到找到可以处理该消息的对象。

Interface Builder 选盘中的标准对象在构建用户界面的时候被分配和初始化，在将对象保存到 nib 文件的时候被归档。当 nib 文件被解档时，这些对象就被恢复了。如果您基于标准的选盘对象类创建一个定制对象，则 Interface Builder 在归档对象的时候会对其超类进行编码，并在解档该对象的时候会将它换为定制类。在上述的两种情况，解档对象的初始化方法都不会被调用。但是如果您生成的是 NSView 的定制类（由 Custom View 选盘对象来表示），则在 Custom View 对象被解档的时候，类的初始化方法会被调用。无论是哪种情况，在 nib 文件中所有对象都被解档后，应用程序都会向与该 nib 文件关联的每个定制对象发送一个 awakeFromNib 消息；这个消息使那些类有机会建立连接或执行其它准备性的任务。

每个 Cocoa 应用程序都有一个主 nib 文件，包含应用程序菜单和可能的一个或多个窗口。NSApp 是主 nib 文件的 File's Owner。当应用程序启动时，NSApp 会装载主 nib 文件，对其进行解档，并显示主菜单和初始窗口。很多应用程序有一些用于文档和面板的辅助 nib 文件；这些 nib 文件是按需装载的（也就是说，当用户需要 nib 文件中对象提供的行为时，才被装载）。

进一步阅读： 阅读*资源编程指南*可以找到更多有关 nib 文件的信息（包括如何进行动态装载）。

装载应用程序资源

Nib 文件和图像文件、声音文件、帮助文件、及其它类型的数据一样，是一种应用程序资源。应用程序资源是可以被本地化的，也就是说，它们可以适用于多种语言和地域环境的。对于 nib 文件来说，本地化基本意味着对用户界面出现的字符串进行翻译。当然其它的修改可能也是需要的，文本框、按键、和其它用户界面对象的尺寸也可能需要调整，以包容新的字符串；日期和数值的格式也可能需要改变。

国际化是指用于支持本地化的开发设施。在国际化一个软件产品时，您必须将各种语言或地域环境的资源放到程序包的 Resources 目录内的特定位置中。这个位置就是一个具有特定名称的文件夹，其名称标识一种语言或地域环境，可以是广为人知的语言字符串或者遵循 ISO 639-1、ISO 639-2、和（面向地域环境的）ISO 3166-1 规范的缩写。本地化文件夹的扩展名是.lproj。用户在系统预置（System Preferences）的国际化（International）面板中调整本地化偏好的列表，而应用程序则从.lproj 文件夹中选择于第一个匹配的本地化设置对应的资源。Xcode 为应用程序资源的国际化提供支持，它负责创建程序包结构，在程序包中包含了.lproj 文件夹，并自动对其进行管理。

应用程序资源也可以是非本地化的。非本地化的资源简单地放在应用程序包的 Resources 目录下，在所有.lproj 目录的外面。

在运行时，应用程序可以定位程序包内的应用程序资源，并将它们装载到内存中，NSBundle 类的实例可以用于这个目的。在给定资源的名称和扩展名的条件下，这个类的方法可以返回给资源的文件系统路径。Application Kit 在 NSBundle 上定义的范畴类使应用程序可以定位和装载 nib 文件、帮助文件、和声音及图像文件。举例来说，NSBundle 的类方法 loadNibNamed:owner:可以找到和装载具有指定名称且

由被引用的对象拥有的 nib 文件。以这种方式动态装载资源是提高应用程序整体效率的编程实践。应用程序应该仅在用户需要资源时才将它们装载到内存中。

应用程序不仅可以动态装载基于文件的资源，还可以在程序中装载特定上下文中的本地化字符串（比如需要显示不同内容的对话框）。本地化字符串是从一个"strings"文件（即一个扩展名为.strings 的文件）取得的，该文件存储在程序包的某个.lproj 目录下。应用程序的主程序包也可以包含辅助的程序包，称为可装载程序包。可装载程序包可以包含自己的代码（和资源），应用程序可以通过 NSBundle 对象将它们动态装载到内存中。可装载程序包使应用程序的行为可以通过一种插件架构灵活地进行扩展（Automator 就是这种架构的一个例子，它动态装载动作，而动作是一些可装载程序包）。

进一步阅读：有关本地化和国际化的更多信息，请参见*国际化编程主题*。NSBundle 和程序包的更多信息则请参见*资源编程指南*和 *Cocoa 代码装载编程主题*。

其它Cocoa架构

从模型-视图-控制器模式的角度可以看到，Cocoa 的核心应用程序架构关心的是应用程序的视图层。它主要解决事件处理和内容描画的基本结构，在如何向用户展示自己及如何与用户交互方面为应用程序提供一个有效的模型。但是在实际的应用程序中，这些部分并不能存在于真空中。应用程序的其它 MVC 对象－控制器和模型对象，也在内容描画和事件处理中发挥重要作用。

请注意：MVC设计模式在 "模型-视图-控制器设计模式"部分中进行描述。

Cocoa为软件开发者提供几个其它的架构。这些架构的关注点几乎都在应用程序的控制器和（特别是）模型对象上。它们基于模型-视图-控制器模式之外的几个设计模式，特别是基于对象建模模式（在 "对象建模"部分中进行描述）。这些架构还有一个共同点，就是使Cocoa开发工作更加容易。

本部分包含如下主要内容：

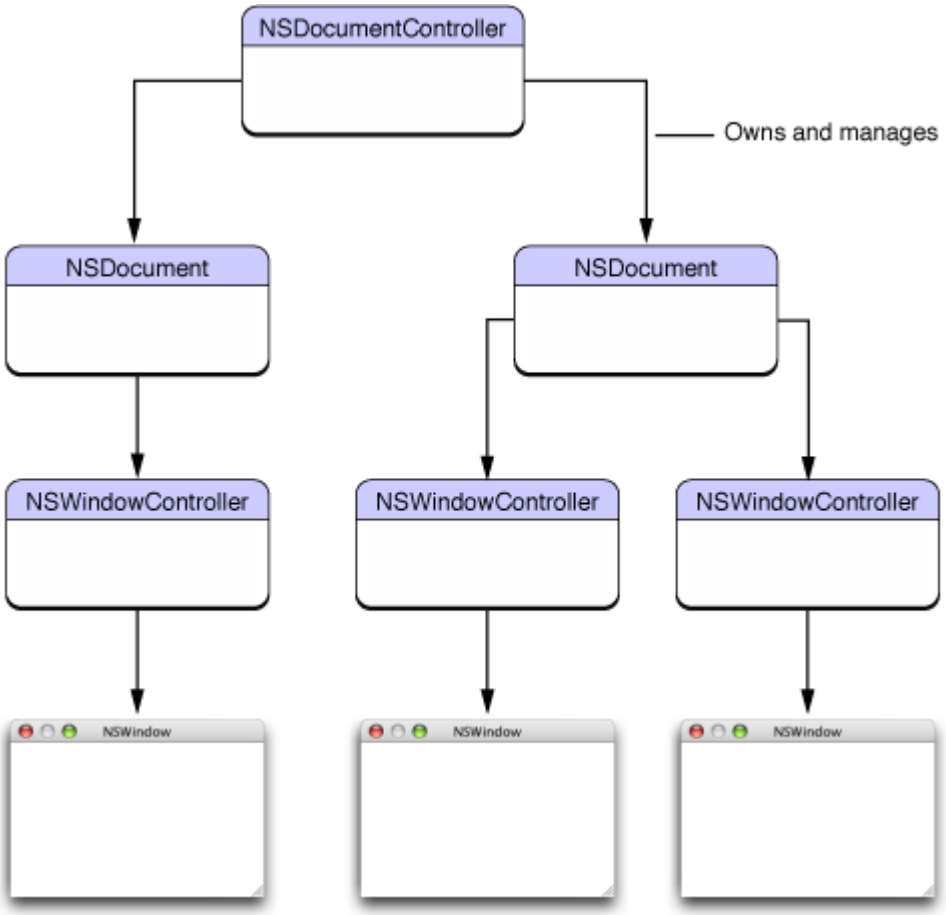
文档架构
应用程序的脚本能力
Core Data

文档架构

很多应用程序让用户创建和编辑文档。文档是指以相同的用户界面表现在窗口中的、独特的数据集合。字处理器、照片-图像编辑器、和 web 浏览器都是基于文档的应用程序的例子。这些程序具有类似的特征，即能让用户创建新文档、能将文档保存为文件、在随后可以打开这些文档。基于文档的应用程序还需要管理菜单项的正当性、监视每个文档的编辑状态、管理文档窗口、以及正确响应应用程序级别的事件（比如应用程序终止事件）。文档数据在这种应用程序中通常有不同的内部表现形式。

Cocoa 为实现基于文档的、具有这些特性的应用程序提供了一种架构，以减少开发这类程序所需要的工作。这个架构的基本组件包括三个 Application Kit 类：、NSDocumentController、和 NSWindowController。在基于文档的 Cocoa 应用程序中，这三个类的对象有自己不同的责任范围，彼此之间有一连串基于所有权和对象管理的级联关系(图 7-1)。

图 7-1 文档类之间的所有权关系



基于文档架构的 Cocoa 应用程序都有一个单独的 NSDocumentController 对象，它拥有和管理一个或多个 NSDocument 对象；这些 NSDocument 对象依次创建和管理一个或多个 NSWindowController 对象；而每个 NSWindowController 对象都和一个文档窗口相关联（一个文档可以有多个窗口）；NSWindowController 对象负责管理文档的表示。

Cocoa 文档架构的这三类对象都有明确的责任，这从它们的 MVC 角色可以看出来：

- **NSDocumentController 对象负责管理应用程序的文档。**它发起和协调创建新文档、保存修改过的文档、以及打开保存过的文档。在整个应用程序中，它充当协调控制器的 MVC 角色。NSDocumentController 对象是 Cocoa 自动为基于文档的应用程序提供的。您一般不需要它的子类。

  NSDocumentController 从文档类型元数据中获取管理文档需要的信息，元数据由应用程序的信息列表（Info.plist）指定，包括文档的扩展名、HFS 类型代码、MIME 类型、图标、以及应用程序能读写的各种文档的 NSDocument 子类（如果有的话）。

- **NSDocument 对象负责管理文档的模型对象。**用 MVC 的术语来说，NSDocument 对象是一个模型控制器，其管理功能向模型层倾斜。NSDocument 是一个抽象类，您必须创建一个能识别档数据的定制子类，文档数据封装在模型对象中。子类的实例必须能够从文件中读取数据，并根据这些数据重建文档模型对象，而且必须能够将文档当前包含的模型对象的集合保存到文件中。它偶尔可能还必须维护多个文档数据的内部表示。一个 NSDocument 对象还拥有和管理一个或多个 NSWindowController 对象。

- **NSWindowController 对象负责管理文档在窗口中的表示。**用 MVC 的术语来说，NSWindowController 对象是一个视图控制器。它的关注点是如何管理显示文档的窗口。对于简单的应用程序，您可能不需要创建 NSWindowController 的定制子类，缺省实例就可以处理基本的窗口管理事务，而视图层的信息可以合并到 NSDocument 对象中。但是在大多数情况下，您应该将文档的视图对象信息加入到一个 NSWindowController 子类中。通常情况下，该子类的一个实例是文档 nib 文件的 File's Owner。对于多窗口的文档，可以生成几个不同的 NSWindowController 对象，每个窗口对应一个控制器对象。

Xcode 为基于文档架构的 Cocoa 应用程序提供了一个工程模板。如果您用这个模板创建工程，Xcode 会自动为您生成一个 NSDocument 的子类（类名是 MyDocument），类的实现文件中含有需要重载的方法存根；还有一个 File's Owner 设置为 MyDocument 的文档 nib 文件。

进一步阅读：文档架构的完整描述请参见*基于文档的应用程序概述*。

应用程序的脚本能力

AppleScript 是一种脚本编程语言，也是很多 Macintosh 的深度用户熟悉的进程间通讯技术。AppleScript 脚本被编译和运行之后，可以通过向应用程序发送命令来对其进行控制，同时接收返回的数据。

为了支持 AppleScript 命令，应用程序必须被赋以脚本控制的能力。具有脚本能力的应用程序通过响应 AppleScript 产生的进程间消息来提供自己的行为和数据，这种消息称为 Apple Event。具有产品品质的应用程序一般都应该具有脚本能力，人们总是希望将自己的应用程序的功能提供给尽可能多的用户，包括脚本编程人员和 Automator 程序的用户。

Cocoa 为具有脚本能力的应用程序提供运行时的支持。当系统首次需要应用程序的脚本能力信息时，Application Kit 会装载这些信息，并自动为支持的命令注册 Apple Event 处理器。当应用程序接收到 Apple Event，请求执行某个注册了的命令时，Application Kit 会实例化一个脚本命令对象，根据应用程序脚本能力信息对其进行初始化。应用程序的脚本能力信息使 Application Kit 可以找到执行该命令的应用程序中具有脚本能力的对象。然后，Application Kit 就执行该命令，具有脚本能力的对象则执行相应的工作。如果那些对象有返回值，则 Application Kit 负责将它包装在 Apple Event 中，返回给发起命令的脚本。

为了使这个运行时支持更为有效，您必须使应用程序具有脚本能力。这个任务包括如下几个方面：

- **您必须提供应用程序的脚本能力信息。**

  脚本能力信息中指定一些词语，用于定位脚本命令的目标应用程序和程序中的对象；还包含另外一些信息，用于说明应用程序如何实现对这些术语的支持。您可以用两种格式来说明脚本能力，首选的一种是基于 XML 的脚本定义－也称为 sdef 格式，因为这种文件的扩展名是.sdef；第二种格式比较老，由一对属性列表文件组成，即一个脚本套件文件和一个脚本术语文件组成。

  两种格式都包含通用的脚本能力信息。它们都定义了可以在脚本中使用的术语，以及应用程序中具有脚本能力的对象的描述信息，包括类、命令、常量、和 AppleScript 及 Cocoa 执行 AppleScript 命令需要的其它信息。

- **您必须实现支持脚本能力需要的所有类和方法。**

  Cocoa 在 Standard 套件中包含了对常用命令和其它术语（比如 get、set、window、和 document）的支持（在 AppleScript 中，与彼此相关的操作关联的术语被合起来放在套件中），还提供对 Cocoa 文本系统对象进行脚本操作的 Text 套件。如果您的应用程序有些脚本操作不能通过 Standard 或 Text 套件定义的命令来完成，就必须定义额外的命令类。此外，应用程序必须为每个具有脚本能力的类实现一个对象指示方法，由该方法描述类中的一个对象，并负责在应用程序对象的层次结构中定位该对

象。

- **您必须正确设计应用程序的模型和（可能的）控制器对象。**

  具有脚本能力的 Cocoa 应用程序在设计时应该遵循模型-视图-控制器（MVC）模式。这种模式将应用程序的对象分成不同的角色。应用程序的模型对象一般是提供脚本能力的对象（虽然控制器对象有时也是具有脚本能力的对象）。应用程序中具有脚本能力的类应该遵循键-值编码（KVC）。KVC 是使对象属性可以通过一个键间接取得或者设置的机制。在运行时，命令对象通过 KVC 机制找到具有脚本能力的目标对象。

  通过将属性的名称（缺省情况下，这个名称也就是它的键）和保有属性或属性存取方法（类定义中负责取得或设置属性的方法）的实例变量结合起来，就可以使您的类遵循 KVC。在为具有脚本能力的（一般是模型）对象设计类的时候，您也要在应用程序脚本能力信息中指定这些对象的键、这些对象中具有脚本能立的属性、以及它们包含的元素类。

  更多有关KVC基础设计模式的信息，请参见 "对象建模"。

  重要提示：使应用程序具有脚本能力并不要求应用程序对象的设计一定要基于 MVC 和 KVC，但是不采纳这种设计会给程序的实现带来更多的困难。

与为脚本控制提供支持相独立的是，Cocoa还可以自动处理来自系统其它进程的特定的Apple Event。"处理Apple Event"部分描述了Apple Event和Cocoa如何处理这种事件。

进一步阅读： *Cocoa 脚本编程指南*详细介绍了 Cocoa 应用程序的脚本能力。

Core Data

Core Data 是一个 Cocoa 框架，用于为管理对象图提供基础设施，以及为多种文件格式的留存提供支持。管理对象图包含的工作如 undo 和 redo、正当性检查、以及保证对象关系的完整性等。对象的留存意味着 Core Data 可以将模型对象保存到持久存储中，并在需要的时候将它们取出。Core Data 应用程序的持久存储（也就是对象数据的最终归档形式）的范围可以从 XML 文件到 SQL 数据库。Core Data 用在关系数据库的前端应用程序是很理想的，但是所有的 Cocoa 应用程序都可以利用它的能力。

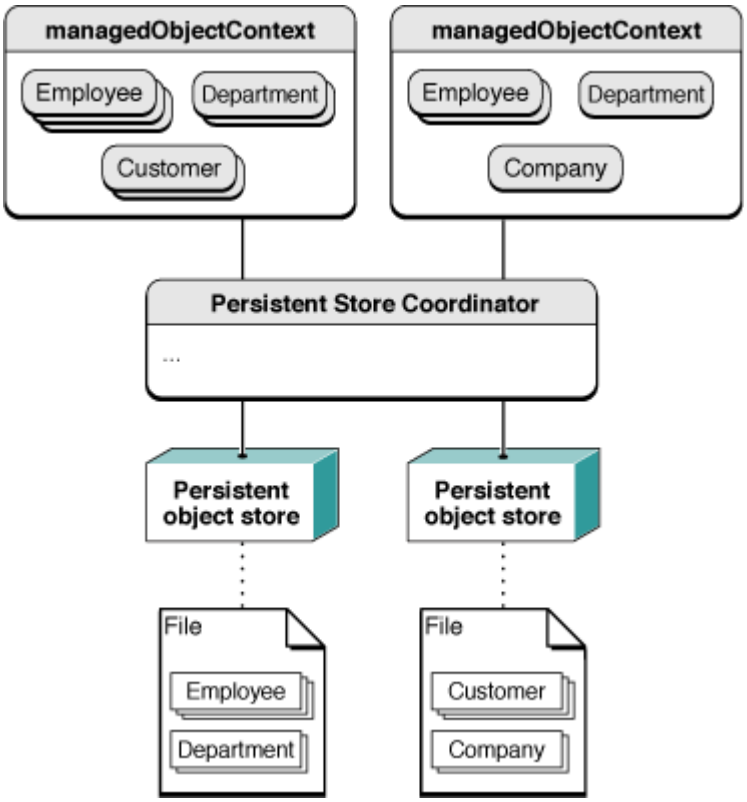
Core Data 的核心概念是托管对象。托管对象是由 Core Data 管理的简单模型对象，但必须是 NSManagedObject 类或其子类的实例。您可以用一个称为托管对象模型的结构（schema）来描述 Core Data 应用程序的托管对象（Xcode 中包含一个数据建模工具，可以帮助您创建这些结构）。托管对象模型包含一些应用程序托管对象（也称为实体）的描述。每个描述负责指定一个实体的属性、它与其它实体的关系、以及像实体名称和实体表示类这样的元数据。

在一个运行着的 Core Data 程序中，有一个称为托管对象上下文的对象负责管理托管对象图。图中所有的托管对象都需要通过托管对象上下文来注册。该上下文对象允许在图中加入或删除对象，以及跟踪图中对象的变化，并因此可以提供 undo 和 redo 的支持。当您准备好保存对托管对象所做的修改时，托管对象上下文负责确保那些对象处于正确的状态。当 Core Data 应用程序希望从外部的数据存储中取出数据时，就向托管对象上下文发出一个取出请求，也就是一个指定一组条件的对象。在自动注册之后，上下文对象会从存储中返回与请求相匹配的对象。

托管对象上下文还作为访问潜在 Core Data 对象集合的网关，这个集合称为留存栈。留存栈处于应用程序对象和外部数据存储之间，由两种不同类型的对象组成，即持久存储和持久存储协调器对象。持久存储位于栈的底部，负责外部存储（比如 XML 文件）的数据和托管对象上下文的相应对象之间的映射，但是它们不直接和托管对象上下文进行交互。在栈的持久存储上面时持久存储协调器，这种对象为一或多个托管对

象上下文提供一个访问接口，使其下层的多个持久存储可以表现为单一一个聚合存储。图 7-2 显示了 Core Data 架构中各种对象之间的关系。

图 7-2 托管对象上下文和留存栈



Core Data 中包含一个 NSPersistentDocument 类，它是 NSDocument 的子类，用与协助 Core Data 和文档架构之间的集成。留存的文档对象创建自己的留存栈和托管对象上下文，将文档映射到一个外部的数据存储；NSPersistentDocument 对象则为 NSDocument 中读写文档数据的方法提供缺省的实现。