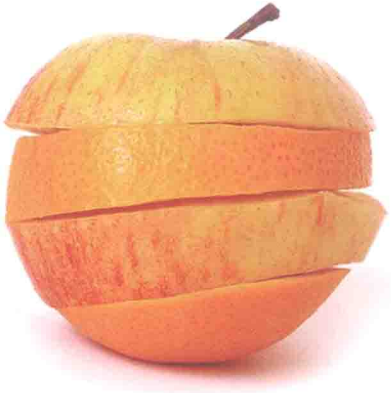
TURING 图灵程序设计丛书

日本Objective-C圣经级教材



Objective-C 编程全解

（第3版）

[日] 荻原刚志 著　唐璐 翟俊杰 译

你所能找到的最细致、最全面的Objective-C教程

人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

Objective-C 编程全解

（第3版）

[日] 荻原刚志 著　唐璐 翟俊杰 译

人民邮电出版社
北　京

图书在版编目（CIP）数据

Objective-C编程全解：第3版 /（日）荻原刚志著；唐璐，翟俊杰译. --北京：人民邮电出版社，2015.1
（图灵程序设计丛书）
ISBN 978-7-115-37719-7

Ⅰ. ①O… Ⅱ. ①荻… ②唐… ③翟… Ⅲ. ①C语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字（2014）第277728号

内 容 提 要

本书结合理论知识和实例程序，全面而系统地介绍了Objective-C编程的相关内容，包括类和继承、对象的类型和动态绑定、基于引用计数的内存管理、垃圾回收、属性声明、类NSObject和运行时系统、Foundation框架中常用的类、范畴、抽象类与类簇、对象的复制及存储、块对象、消息发送模式、图像视图、异常和错误、并行编程、键值编码等。

本书适合iOS应用和Mac OS X开发初学者系统入门、有经验的开发者深入理解语言本质，也适合开发团队负责人、项目负责人作为综合性的Objective-C参考书阅读。

◆ 著　　[日]荻原刚志
译　　唐　璐　翟俊杰
责任编辑　乐　馨
执行编辑　杜晓静
责任印制　杨林杰
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京天宇星印刷厂印刷
◆ 开本：800×1000　1/16
印张：28.75
字数：731千字　　2015年1月第1版
印数：1－3 000册　　2015年1月北京第1次印刷
著作权合同登记号　图字：01-2013-2656号

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第0021号

译者序

Objective-C 近年来一直备受关注，它诞生于 1983 年，设计之初和 C++ 一样都是为给 C 语言加入面向对象的功能。在 iPhone 发布之前，Objective-C 一直都是一门默默无闻的小众语言，而 C++ 则长期占据编程语言排行榜的前几名。随着 iPhone 销量的爆发性增长，Objective-C 语言的份额也迅速增长。在 TIOBE 编程语言排行榜上，Objective-C 只用了短短的 3 年时间就从 2009 年 8 月的第二十名增长到了 2012 年 7 月的第三名（仅次于 C 语言和 Java），截至今天仍然排名第三。

因为 Objective-C 流行的时间还比较短，所以就算是开发过多个上架 App 的 iOS 程序员也仍可能不完全了解 Objective-C 语言。例如，不知道消息转发背后的工作机制，不了解 Objective-C 几种内存管理之间的区别，不清楚该如何选择 Delegate、Notification 和 KVO 等。

本书是一本专门讲解 Objective-C 的图书，虽然作者并不会教你如何使用 Xcode 开发一个运行在 iPhone 上的 App，但用了 20 章的内容深入讲解了 Objective-C 的各个特性。

2014 年苹果公司在 WWDC 2014 开发者大会上发布了用于 Mac OS X 和 iOS 编程的新一代编程语言 Swift，相信很多读者一定在为应该学习 Objective-C 还是应该直接学习 Swift 而感到困惑。Swift 于 2014 年才发布，而目前市面上至少有上百万个 App 和类库都是用 Objective-C 编写的，如果你未来的工作是为现有 App 添加新的功能（这种工作的可能性很大），那你一定要懂得 Objective-C。

除了讲解 Objective-C 语言本身外，本书还介绍了与其密不可分的 Foundation 框架，介绍了字符串、数组、字典等常用类。

本书的作者荻原刚志教授从 iPhone 还没诞生的 2001 年就开始写 Mac OS X 上的 Objective-C 编程入门的书籍，是 Objective-C 方面真正的专家。作者写这本书时把日本人认真严谨的天性发挥到了极致，用了 400 多页来讲解 Objective-C。本书已经再版了三次，本次翻译的就是最新的第三版。作者在讲述每个知识点的时候都精心配置了示例代码，所有的示例代码均可运行。相信通过对本书的学习，你一定能够打下坚实的 Objective-C 基础，开发出更美妙的 App。

本书由唐璐、翟俊杰翻译。在翻译的过程中得到了图灵编辑的大力帮助，在此深表谢意。由于时间仓促，译者水平有限，错误与疏漏之处在所难免，敬请读者谅解并批评指正。

序言

Objective-C 是一门为 C 语言增加了面向对象功能的语言，是开发 Mac OS X、iPhone、iPod touch 和 iPad 应用的主要语言。

在为 C 语言加入了面向对象功能的语言中，C++ 是最有名的一种。Objective-C 和 C++ 完全不同。Objective-C 和大家所熟知的 Java、C# 和 Ruby 也有所不同，是一门比较独特的语言。

Objective-C 最大的特点是支持面向对象编程，具备很多动态语言才有的动态特征，同时在效率上还可以媲美 C 语言。学习过其他面向对象语言的人可能会对 Objective-C 为 C 语言添加的功能之少感到惊讶。

随着 Mac OS X 和 iOS 的逐步更新，Objective-C 运行的系统环境也做了同步升级。同时 Objective-C 语言本身也引入了不少新的特性，包括一种新的内存管理方式——ARC。另外属性声明和代码块（block）的使用范围也得到了扩大。Objective-C 的编程风格这些年一直在不停地更新。

本书以开发 Mac OS X 或 iOS 应用为目的来介绍 Objective-C，默认读者具备 C 语言基础，但并不要求精通 C 语言。

Objective-C 和苹果公司的产品和运行环境紧密相关，介绍 Objective-C 的时候无法脱离具体的操作系统或框架。本书在介绍语言本身的同时也介绍了 Foundation 框架中的主要类，同时也尽可能指出了 Mac OS X 和 iOS 的不同之处。

另外，本书并不是一本讲解 Mac OS X 和 iPhone 图形界面编程的书，不会涉及 GUI 控件的使用，所以并不是说读了本书就能立刻做出一个具有优美界面的应用程序。已经有太多的优秀书籍介绍 GUI 编程的方方面面，请参考这些图书的内容。

Xcode 是苹果公司向开发人员提供的集成开发环境，用于开发 Mac OS X 和 iOS 的应用，其中自带了 Objective-C 的编译器。到笔者写作本书为止，Xcode 可以免费下载安装（只能安装在苹果系统中，没有 Windows 版）。而且只要加入苹果公司的开发者计划，注册成为 Apple Developer，就可免费获取创建 iOS 应用和 Mac 应用的资源，包括开发工具、示例代码、技术文档等。如果想在 iPhone 或 iPad 的真机上测试自己开发的应用并发布到 Apple Store，则需要付费加入苹果公司的 iOS 开发者计划。

本书中的代码都是终端类型的程序，只要安装了 Xcode，不需要对 Xcode 作任何设置，就可以编译、运行示例程序。对学习的内容有任何疑问时都可以通过运行程序来找到答案。

本书是《Objective-C 编程全解》的全新修订版本，新增了以下内容。

- 详细介绍了 Objective-C 新引入的内存管理方式 ARC 的方方面面，从工作原理到编程时的各种注意事项。以使用 ARC 为前提对示例程序做了大幅修改，同时简化了手动内存管理和垃圾回收方面的说明。
- 介绍了新的属性声明方法，并使用了新的方法重写了示例程序。
- 追加了使用 ARC 时 Core Foundation 对象和 Objective-C 对象之间如何进行类型转换的内容。

本书中讲解的所有内容都经过了实际编码测试。本书的代码在 Mac OS X 10.7 和 iOS 5 以上的环境下能正常执行。书中源代码可以从以下链接下载。

http://download.sbcr.jp/getDLService.php?id=c2e364e9c1c1181de9c303520a68493a

另外，本书还提供了以下三部分内容作为附录，帮助读者理解。

- 附录 A　Foundation 框架的概要
- 附录 B　Core Foundation 框架概要
- 附录 C　编码原则

请读者在开始阅读本书前自行下载：

http://www.ituring.com.cn/book/download/9f2a581b-81f3-4873-8b4c-82addf7211ab

Mac OS X 和 iOS 的软件开发环境的可扩展性非常好，也很人性化，只要具备基础知识就能够开发出简单的应用。但如果要开发一个真正的商用应用，还需要具备系统化的 Objective-C 知识，因为运行环境是为了最大限度地发挥 Objective-C 的特征而设计的。通过本书学习了 Objective-C 的基础知识之后，你也就可以更加得心应手地使用其他框架了。

开始享受你的 Objective-C 编程之旅吧！

2011 年 10 月 5 日，Apple 公司（NeXT 公司）的缔造者乔布斯逝世。乔布斯总能像魔法师一样给我们展示一些令人难以置信的新产品，真心感谢他带给我们的一切。一路走好！

版权声明

目录

第1章

面向对象的编程

本章介绍面向对象的基本概念，讲述基于面向对象的建模和对象模块化方面的内容。

1.1 对象的概念

1.1.1 面向功能的软件设计的局限性

很多编程语言，例如 C、Pascal、Basic 等，都是面向过程的。面向过程的程序设计语言主要使用顺序、选择和循环三种基本结构来编写程序。顺序指按照时间轴顺序完成每个处理；选择指根据条件的成立与否执行不同的条件分支；循环指根据一定的条件反复执行同样的代码。

在面向对象技术问世之前，程序设计被看为一个个功能系统的集合。程序员根据设计文档实现各个函数，完成目标软件。所以，那个年代软件开发最关心的是如何从需求中提取出要实现的功能，决定数据格式，并将其组合在一起。举一个简单的例子，假设我们需要实现一个用于管理和采购办公用品的 OA 系统。在这个 OA 系统中，我们假设某部门提出采购要求后，首先要检查该部门的预算是否够用。通常的做法是设计申请购买商品的登录页面和申请的流程，决定使用的数据格式，定义检查数据正确性的函数，定义预算数据的格式和访问它的接口。简而言之，整个系统设计的过程就是将需求分解成一个个小的功能，同时定义每个功能所需要的数据格式。

但是，需求是无时无刻不在变更的。例如，假设需要从纸制文件的购买申请变更为通过网页来申请，或者需要新增能根据部门、物品类别来查看购买历史记录的功能，这时就需要对基于传统做法生成的软件进行大幅的修改。软件的核心是功能，而需求又非常容易变更，所以围绕功能设计软件，会比较难于应对需求的变更，维护成本会比较高。

1.1.2 面向对象的模块化

从 20 世纪 80 年代后半期开始，**面向对象**（object-oriented）的编程方法渐渐引起了人们的关注。

首先，让我们超出计算机的范畴广义地考虑，**对象**（object）指的就是人能够识别的东西，从你手里拿的书、随身的笔记本，到桌子、手表、摩托车、汽车、收据、工资单、图书馆和区政府等，这些都是对象。

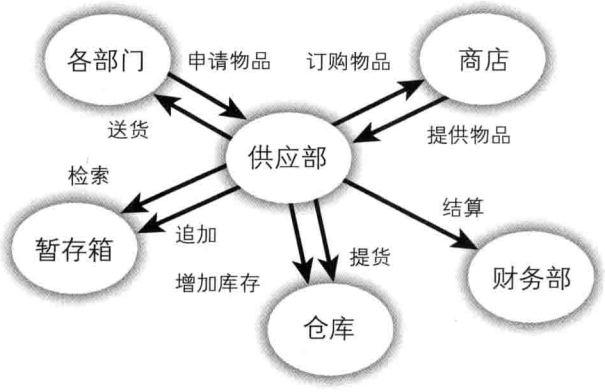
而且，对象有**属性**（attribute），并且能够接收**消息**（message）来进行相应的处理。属性指的是对象的性质和所具有的信息，例如汽车能装多少个人，是什么颜色的，现在行驶的速度是多少等。属性也可以称为状态。消息是指对象和对象之间的信息传递，在信息共享、查询、命令请求时使用。

面向对象的编程指的是，以对象作为程序的基本模块来进行软件的分析、设计和开发的一种思考方法。

再让我们来看看刚才提到的那个办公用品管理系统。

某公司的办公用品管理系统对于圆珠笔和复印纸这类消耗品都会有一部分预留库存。每个部门想要这些东西的时候，如果库存够用，就会直接从库存中划拨，并提交结算单给财务部门。如果库存不足，则会从商店购买，并把各部门提交的申请单放入暂存箱中。商品送来之后，把商品交给需要的部门，并同时把结算单提交给财务部门。

图 1-1 总务部门的业务



只读文字可能不太好懂，参照图 1-1 来看一下会更容易理解。图里面的椭圆形就是对象，箭头代表的是消息的传递。图中虽没有标明对象的属性，但例如库存中肯定会有圆珠笔和复印纸的数量，暂存箱中肯定会有还没有处理的订单信息，这些分别是仓库和暂存箱的属性。另外，订单本身也可以用一张纸质订单这样的对象来表示。通常人们在整理多个概念之间的关系时，会画类似于图 1-1 这样的图来理顺关系。因此，把这个图中的每个对象作为一个模块来实现一个软件会是个不错的选择。

例如，总务部告诉供应部："我想要 10 支圆珠笔。"供应部会调用仓库的属性，即圆珠笔的库存情况。如果库存不足的话，供应部会向商店发出购买圆珠笔的请求。

这张图并没有表现出功能性的细节，所以就算是申请办公用品的方式基于网页，或需要添加查看购买的历史记录，上图中描绘的各个对象之间的关系也不会有太大的变化。基于面向对象的软件开发，比较接近于人的思考方式，更善于应对需求变更。

1.1.3 消息传递

消息是对象之间通信的唯一手段。请求、查询、应答和异常通知等，所有的通信和控制都是通过收发消息完成的。

对象收到消息后，会对消息进行解析，完成相应的处理并返回结果。具体的处理方法和这个对象的内部实现相关，这里叫作**方法**（method）。

方法中写明了程序的各种操作的实现和规则。

消息传递的时候可以使用对象或者基本数值作为参数。另外，消息处理的结果也可以返回一个对象或基本数值。

送信的对象称为**发送者**（sender），收信的对象称为**接收者**（receiver）。

通过消息协调各个对象之间的消息发送，使其作为一个整体运行，这就是面向对象的软件的运行模式。

1.1.4 模块的抽象化

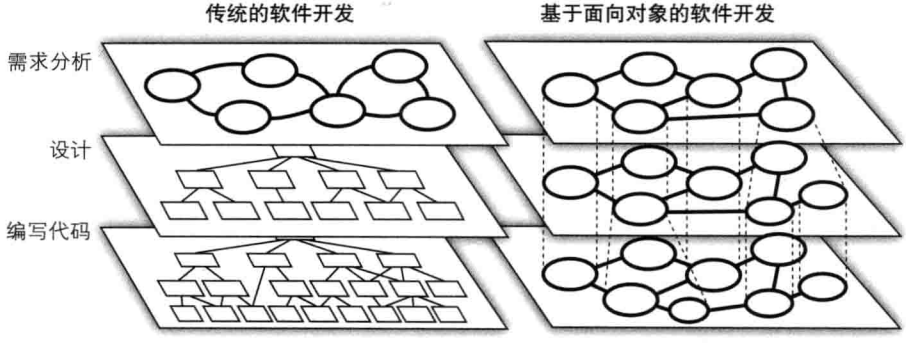
综上所述，具备以下特征的东西可称为对象。

- **可人为分辨出这是一个对象**
- **拥有属性**
- **能够向其他对象发送消息**
- **能够接收消息，并作出相应的处理**
- **消息的处理是通过对象的方法完成的**

像这种使用对象的概念对问题进行抽象化的方法叫作面向对象。使用对象的概念来分析如何做一个软件叫作**面向对象的分析**（OOA，Object-Oriented Analysis）。以对象为基础来设计软件叫作**面向对象的设计**（OOD，Object-Oriented Design）。编程过程中使用面向对象的概念叫作**面向对象的程序设计**（OOP，Object-Oriented Program）。另外，以消息通信构成的鼠标、键盘或用户界面的按钮等同程序之间的接口叫作面向对象的接口。

通过使用面向对象的语言，可在面向对象的分析和设计的基础上来编程。它不像传统的以功能为核心的软件开发，需要明确指明每个函数所对应的功能。面向对象的软件开发，从需求分析、设计到编程都使用统一的模型，所以更善于应对需求变更。

图 1-2 模块的一致性



把一个事物作为对象考虑时，并不需要把真实世界中这个事物的所有属性和构成全部放到对象中，只需要考虑和要实现的模型有关的属性和动作即可。例如现实中供应部的工作肯定不会像我们举例这么简单，但在设计的时候也不需要考虑要订的纸是不是 A4 的，商店是不是早九晚五上班等过于琐碎的细节。

抽象化（abstraction）指的是尽可能地不考虑相关细节，只关注对象的核心和本质。对于现实世界中的事物，你越观察、分析就会发现越来越多的细节。通过抽象可以用简单概念的集合来描述一个复杂的对象，这是非常重要的。

如何抽象出一个对象，是根据要实现的模型和软件的性质、功能来决定的。其实我们人类对于事物的认识，在不知不觉中也是一个抽象化的过程。理解模型中的对象，就和理解人类日常生活中

的对象是非常相似的。

目前为止，好像给大家留下了一个“万物皆对象”的感觉，到底有没有什么不适合作为对象的东西呢？适合作为对象和不适合作为对象的东西有什么区别呢？

实际上并没有明显区别，只要构建出来的模型是没有问题的，万物都可以被当作对象来处理。

真实世界中某个独立存在的事物、一个整体的某个组成部分、有重要作用的人和部门等，都比较适合作为对象。特别是如果某个东西能够拟人化，也比较适合作为对象。另外，多个素数、行列这些和数学有关的操作也可以被看作对象。

相反，时间、空间、知识、伦理、感动等等这些过于抽象，不适合用数字来定性的东西，不适合作为对象。另外难于被归类成“收到消息，进行处理”这个过程的，也不太适合当作对象。

那么“数字”到底该不该被看作对象呢？对此有正反两方面的意见。整数或实数这种单纯的数字类型，可被看作对象，也可被看作单纯的数值。让我们来看看在编程语言中是如何做的。Smalltalk（面向对象编程语言的始祖）中所有的东西都是对象，所以数字也不例外。而 C++ 中数字并不是对象。Objective-C 和 C++ 一样，并不把数字看作对象，后面的章节中将会有具体的说明。

专栏：面向对象的方方面面 COLUMN

“面向对象”是一个经常被提及的词，但很多以“面向对象”开头的词并没有一个被广泛认可的定义。例如，到底什么样的编程语言才算是一个面向对象的语言，就有各种不同的定义。

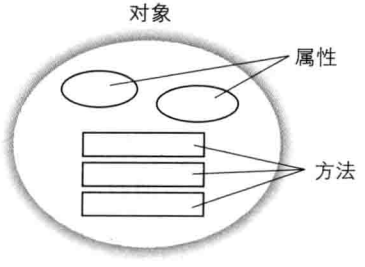
同样，语言中很多基本的定义、概念和惯用名在不同的编程语言或不同的设计方法中存在不同的解释，有点混乱。更麻烦的是，把资料翻译成中文的时候，又新“发明”了很多说法。

Objective-C 来源于面向对象语言的始祖 Smalltalk，所以 Objective-C 中和面向对象相关的各种表达的本质和 Smalltalk 是基本一致的，但和其他面向对象语言有可能存在不同。阅读本书的时候要注意这点。

1.1.5 对象的属性

让我们从属性的角度来重新认识一下对象这个概念。图 1-3 是一个包含了属性和方法的对象。

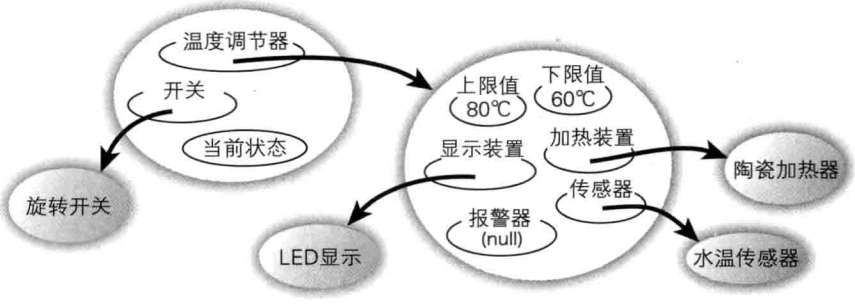
图 1-3 对象的概念图



对象拥有属性（也可以说是状态），但属性是怎么被定义的呢？对象的属性一般被定义为指向其他对象的指针，这个指针叫作**实例变量**（instance variable，简称为 ivar）或变量。变量可能指向一个空的对象（null），另外变量也不一定必须是一个指针，也可以是数值类型。

让我们来看一下图 1-4，这是一个油箱（或水箱）保温系统的例子。左上角的对象是整个系统的总控部分，它有三个属性，分别是系统开关、温度调节器和当前状态（停止中、低温保温中、高温保温中等）。开关和温度调节器这两个属性分别指向其他两个类的对象。

图 1-4 对象的属性



温度调节器本身也拥有若干个属性，例如用于获取水温的温度传感器、用于给水加热的加热器、用于显示当前温度的 LED 显示设备，这些属性都是其他对象的实例。温度调节器还有一个属性是报警器——用于在水温过高或过低时报警，目前还没有指向某个具体的对象。另外水温的上限值和下限值都已经设置好了。

对象和对象之间一般是通过一个对象的某个属性是另外一个对象的变量来建立关系的。没有引用关系的两个对象之间无法发送消息。Objective-C 中把连接对象的变量称为**输出口**（outlet）。GUI 的设计工具 Interface Builder（Mac OS X 平台下用于设计和测试 GUI 的应用程序）中也使用了 outlet 这种说法。各个对象之间就好像插口一样，互相连接在一起。

1.1.6 类

图 1-4 展示了管理油箱温度用的多个对象之间的关系。但对于一个大型设备来说，可能有多个需要管理的油箱。这时如果为每个油箱单独设计的话，就可能会非常麻烦。

针对这种情况，我们可以把具备相同变量和方法的对象提炼出来，做成“模板”。这样以后就可以使用“模板”来创建各个具体的对象。这种“模板”就是**类**（class）。

类包含了一组特定对象的共有特性。例如就汽车这个对象来说，虽然世界上有各种各样的汽车，但汽车的基本结构和驾驶方法都是相同的。于是我们就可以不考虑汽车构造和驾驶方面的各种细节，抽象出“汽车”这个概念。虽然每个具体型号的汽车有着不一样的细节，但整体上都符合我们抽象出来的汽车的概念。类就是舍弃了每个具体对象的各种细节，把所有对象都具备的共同的部分抽象出来。

所有类的对象都可以共享该类中定义的变量和方法。不同对象之间的差异就在于变量值的不同。拿汽车来举例的话，变量就包含颜色、外观、引擎和轮胎等，根据这些条件就可以确定一台汽车了。

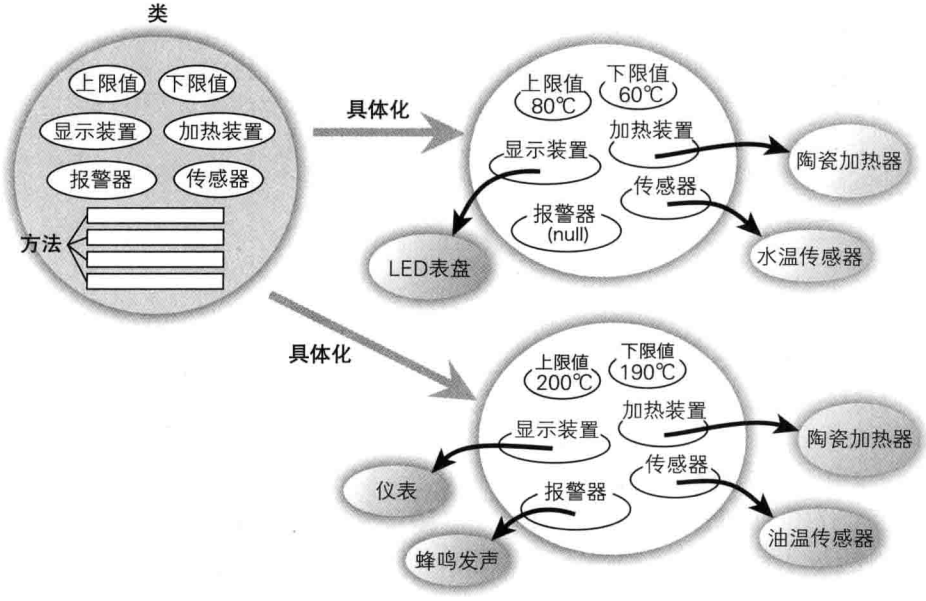
用类创建对象的过程叫作**实例化**（instantiation），生成的对象叫作**实例对象**（instance object），或简称为**实例**（instance）。

不同实例对象的变量各不相同，或指向不同的对象，或有不同的数值。实例所拥有的变量称为实例变量。类决定了需要定义几个、定义什么类型的实例变量。不同实例对象的实例变量可以各不相同。

方法是在类中统一定义的，同一个类不同实例对象的方法都是相同的。

在图 1-5 中，管理水温的对象和管理油温的对象是同一个类的不同实例。管理水温和油温的方法是一致的，但各自用到的上限值和下限值各不相同，水温和油温用的感应器和加热器也各不相同。很多面向对象语言中，对象都只能由类的定义来实例化。

图 1-5 对象和实例对象的一个例子



既然提到了类，就不得不提到一个非常重要的概念——**继承**。但一次讲太多的话比较容易引起混乱，关于继承的详细内容就留到第 3 章详细说明。接下来我们从软件构成的角度来继续说明一下对象。

专栏：Cocoa 和 Objective-C 的历史 COLUMN

Mac OS X 是一套基于 Unix 内核的图形化操作系统，主要技术来源于苹果公司 1996 年底收购的由史蒂夫·乔布斯创立的 NeXT。早在 1985 年，乔布斯离开苹果公司后成立了 NeXT 公司，并于 1988 年推出了 NeXT 电脑，使用 NeXTSTEP（后期为对应多平台版本，名字改为了 OPENSTEP）作为操作系统。在当时，NeXTSTEP 是相当先进的系统，它以 Unix 为基础，同时集成了很先进的、超越 Macintosh 的 GUI。那个时候，Windows 还只是一个登不上大雅之堂的玩具。NeXTSTEP 就是用 Objective-C 开发的。苹果公司收购 NeXT 之后，作为苹果电脑下一代操作系统的基础，OPENSTEP 便演变成为 MacOS X 的 Cocoa 环境。Cocoa 是 Mac OS 上面应用软件的核心类库。Cocoa API 中开头的“NS”，就是 NEXTSTEP 的缩写。

想要开发出优秀的程序，必须熟悉标准 API。Cocoa 以框架（framework）的形式提供了很多类，Cocoa 框架的设计和开发中就应用了很多面向对象技术。使用 Cocoa 开发一段时间后，开发者就会逐步理解 Cocoa 设计的精巧之处了。

2006 年，苹果公司的 Mac 电脑更换了 CPU，从 PowerPC 公司更换到了 Intel 公司的 CPU。但操作系统 Mac OS X 的代码基本没怎么修改，只需重新编译一遍就能够在 Intel 平台上运行。这不仅得益于 Mac OS X 的前身 OPENSTEP 本身就是多平台（Intel、Motorola、Sun、HP）的，还和 Cocoa API 良好的移植性密不可分。

苹果公司 2007 年发布了 iPhone，2010 年发布了 iPad。iPhone 和 iPad 使用的操作系统是 iOS（最初称为 iPhone OS）。iOS 是以 Mac OS X 为核心面向移动设备特殊定制的操作系统。iOS 上面提供的应用程序类库叫作 Cocoa touch。iOS 平台上使用的编程语言是 Objective-C。Objective-C 是在 C 语言基础上扩展而成的，它在面向对象的可扩展性和最大发挥硬件性能两方面做到了良好的平衡。

未来苹果公司新开发的设备和系统都会以 Mac OS X 为基础，同时使用 Objective-C 作为编程语言[①]。

1.2 模块和对象

上一节中我们简单说明了基于面向对象的软件设计和开发。本书在说明时使用了比较简单的概要图，而真正的基于面向对象的分析和设计则会使用 UML（Unified Modeling Language，统一建模语言）这种图形化的表示方法。至于如何基于对象来建立模型，如何用图形或文档来描述定义好的模型等，都会被作为各种各样的软件开发方法来详细讨论。

接下来我们来讲讲如何基于面向对象来设计软件。封装是软件设计中一个非常重要的概念，不仅仅是 Objective-C，所有的面向对象语言都将使用到封装这一概念。

1.2.1 软件模块

无论是面向对象还是面向过程的软件开发，都需要把要完成的系统分解成若干个小的**模块**，先

① 在 WWDC2014 大会上，苹果新发布了编程语言 Swift。它比 Objective-C 更简洁，运行更轻快。——译者注

独立开发每个模块，然后再组装成整个软件。那么到底什么是模块呢？通常，模块是工业制品中的概念，指的是整体中的某一部分，它具备独立的功能，更换时不会影响到其他部分，例如电源模块。

软件开发中的模块也是一个功能单位，构成一个软件的各个相互独立的部分叫作模块。一个模块由变量、方法甚至其他模块构成。所以模块具备层次性。

我们这里提到的模块所提供的功能包括：过程和函数的调用、变量的重新赋值等。

1.2.2 高独立性的模块

长久以来，人们一直在寻找软件模块划分的最佳方案，并为此提出了各种各样的概念和方法。但简而言之，独立性高（高耦合，低内聚）的模块划分是最佳的划分方式。模块的独立性，指的是每个模块之间的交集应该尽可能地小。这样，模块的内部无论如何变化，对其他模块的影响都能减少到最小。

如果一个模块中实现的内容需要参照其他模块的内容才能理解，或一个模块发生了变动之后其他模块都需要相应地改变，这样的模块就是独立性比较差的模块。

我们可以从 What 和 How 这两个角度来观察一个模块。What 指的是这个模块提供了什么功能。How 指的是这个模块如何实现这些功能。一个独立性高的模块会把 What 和 How 清楚地分开。相反，独立性低的模块往往无法良好地区分 What 和 How，说不清楚它所提供的功能。

就独立性高的模块而言，我们只需要知道它所提供的功能就能够良好地使用，不需要了解其内部是如何实现的。以独立性高的模块为基础，就算是大型软件也可以很容易地完成。独立性高的模块的声明和实现是分开的，只要声明保持一致，具体的实现可以随时被更换成性能更高的实现。独立性高的模块一旦发生问题，只需要更改这个模块即可，这大大地提高了软件的可维护性。

相反，如果一个模块封装得不好，其他模块直接使用了这个模块内部的一些实现，那么未来无论是想更新模块内部的某个方法的实现还是想替换这个模块都会很麻烦。

1.2.3 模块的信息隐蔽

模块独立性的划分原则是只对外提供最小限度的接口信息，内部实现不对外公开。也就是把模块做成一个黑盒（black box）。

这个原则叫作**信息隐蔽**（information hiding）或**封装**（encapsulation）。

如果提供方提供的模块是一个黑盒，那么调用方就只能使用该模块所提供的功能来开发软件。提供方也只需要按照事先的约定实现功能即可。这样的话，一旦提供方发现了更快的算法或更有效率的内存管理方式，就可以很容易地对内部实现进行替换，这也符合之前说明的对于高独立性模块的要求。另外模块调用方也不需要了解模块的内部实现细节，只需按照事先约定好的接口来调用函数即可，这样更容易制作出可重用的软件。

传统的面向过程语言的封装功能有限，很难制作出高独立性的模块。在这些语言中，数据和对数据的操作是分离的，从语言层面上无法提供模块化的解决方案，也不能对外隐蔽模块的内部实现。

从 1980 年开始，人们开始思考如何从语言层面上来提供模块化编程的支持，例如以 Pascal 为基础开发出了 Modula-2 的模块、Ada 的包等。

C 语言本身是以文件为模块的，难以实现多层次的模块。通过使用 static 函数或特别构造头文件等方法也能够做出高独立性的模块，但这对程序员的要求比较高，一不小心就可能弄出低独立性、不好理解的程序。

1.2.4 类的定义和接口

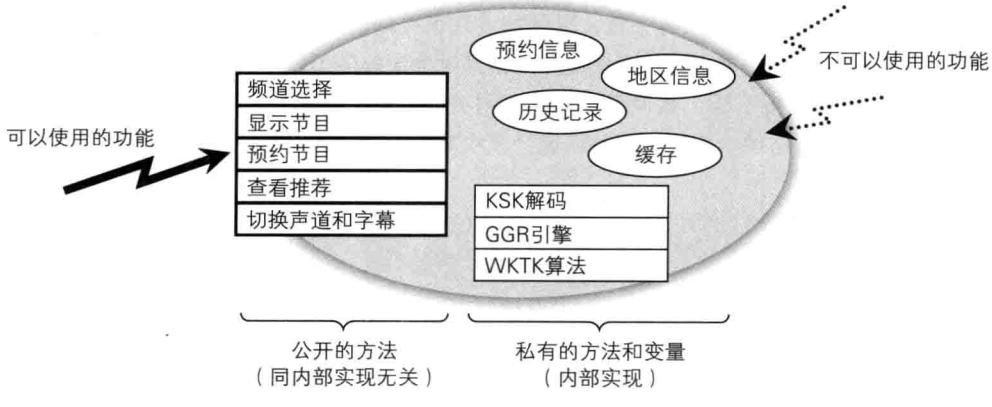
在绝大多数基于面向对象语言开发的程序中，对象是由类的定义来描述的，实例对象是在程序运行的时候动态生成的。也就是说，类的定义和实现文件就是构成程序的模块。

类是由实例变量和方法构成的，所以类的定义中包含了实例变量和方法的定义。对象的使用者只关心到底该如何使用这个对象，而不关心它的内部实现。

类公开给外部的、关于如何使用这个类的信息叫作**接口**（interface）。接口中定义了这个类所包含的实例变量和可接收的消息。

从类的外部也就是类的使用者的角度来看，只能看到接口中定义的信息，无法看到类内部的实现和数据定义等。图 1-6 用一个电视的例子说明了接口的概念，用户只需要知道电视的使用方法（接口）就可以了，不需要了解电视内部的各种实现细节。

图 1-6 接口的概念



如上面的例子所示，类把对外公开的方法记录在接口中，除此之外各种详细的实现对外均不可见。一个设计良好的类只会把必须公开的信息记录在接口中，这样才能够加强独立性。

就类的定义而言，有把接口和实现分开写的语言，也有写在一起的语言，Objective-C 属于前者。

1.2.5 消息发送的实现

一个程序通常由多个对象构成，而至于每个对象到底该如何完成预设的功能，则有各种各样的

实现方法。

其中一种实现的方式就是多个对象同时执行，对象之间的消息通信也并行进行（异步）。就好像很多人在参加一个宴会一样，每个人都可以和其他人说话。另一种方式通常是一个对象发送消息触发其他对象的消息处理函数，整个过程好比一个顺序发言的研讨会。

现在在很多面向对象的语言中，消息的发送就好像函数调用一样，可以将给某个对象发送消息看作调用这个对象的一个函数。虽然在消息的发送方法、响应方法的确定等方面和函数调用还有很多不一样的地方，但在发送方发送消息后需要等到接收方处理完成才能返回这点上和函数调用是一样的。

另一方面，进程间通信、不同主机间的网络通信也可以抽象地看成对象间的消息通信。把这些情况都考虑在内的话，就会出现很多仅仅使用函数调用这种形式无法解决的情况。例如为了高效率地实现进程间通信，Objective-C 特别定义了一种接收方无需返回应答的消息。

专栏：C 语言的新标准 COLUMN

本书是在假定读者已经知道如何编写 C 语言程序的基础上创作的。Objective-C 最初的设计目标是实现一种可替代 C 语言的语言，所以读者一定要具备 C 语言的基础。与此同时，C 语言标准的变化也给 Objective-C 带来了影响。

我们先来介绍一下 **K&R**。K&R 是指 Brian W. Kernighan 和 Dennis M. Ritchie 两人，他们是 C 语言之父，同时也是《C 程序设计语言》（*The C Programming Language*）一书的作者，这是一本广为人知的书，许多人都亲切地称之为“C 语言圣经”。这本书书末的参考指南（Reference Manual）给出了当时 C 语言的完整定义，成为那时 C 语言事实上的标准，人们称之为 K&R C。在此之后，符合 ANSI 标准的《C 程序设计语言》第二版出版了。

从 1989 年到 1990 年，ANSI（American National Standards Institute，美国国家标准学会）和 ISO（International Standard Organized，国际标准化组织）先后通过了关于 C 语言的标准草案 C89 和 C90。除了在印刷编排上的某些细节不同外，C89 和 C90 在技术上完全一样。目前各种书籍中提到的“ANSI C”指的都是这个标准。

1999 年，ANSI 和 ISO 又通过了最新版本的 C 语言标准和技术勘误文档，该标准被称为 **C99**。2011 年 12 月，ANSI 采纳了 ISO/IEC 9899:2011 标准，即 C11，它是 C 语言的最新标准。目前大多数编译器都能够兼容 C99 的绝大部分功能。其中，inline 函数、可变长度的数组、通过优化编译器提高数据访问效率的 restrict 标识符等是几个最主要的功能。C99 之前规定变量必须定义在一个程序块的开始，C99 之后开始允许在块中定义变量，但必须被定义在需要使用它的代码前面。

Xcode 4.2 之前，Mac OS X 提供的编译器一直是基于 GNU 的 **gcc**，在 Xcode 4.2 之后被更换为了 **LLVM** 的编译器 **clang**。无论是哪个编译器，都能够编译 C、Objective-C 和 C++ 的程序，也都能够编译生成 Mac OS X、iPhone、iPad 的应用程序。这些编译器都兼容 C99 标准，所以无论是 C 程序还是 Objective-C 程序都能够使用 C99 中的新功能。另外，编译器中也加入了苹果公司的独有的扩展，例如后面将详细讲到的 block 对象等。

第2章

Objective-C 程序设计

本章介绍如何用 Objective-C 定义一个简单的类。虽然很多概念会比较模糊，但让我们暂且不管这些，先试着编写一个真正的 Objective-C 程序吧。

Objective-C 是在 C 语言的基础上加入面向对象的特性扩充而成的。这意味着除了新加入的功能之外，基本上可以使用 C 语言的写法。

2.1 对象和消息

2.1.1 消息表达式

在 Objective-C 中，id 类型是一个独特的数据类型，id 类型的变量可以存放任何数据类型的对象（相当于 Java 中的 Object）。定义一个 id 类型变量 obj 的方法如下所示。

```
id  obj;
```

向 obj 发送 msg 消息的语法如下所示。

语法	消息表达式
[obj msg]	

这就是**消息表达式**（message expression）的语法。其中，消息接收者（obj）是一个对象，消息（msg）告诉它要去做什么。请求一个实例执行某个操作的时候，你就可以给它发送一个消息。消息表达式和 C 语言中的函数调用类似。消息表达式可作为一个变量嵌套在语句中，也可以单独执行返回 void。

Objective-C 允许嵌套消息表达式，每个消息表达式的返回值都可以作为消息的接收者，继续接收消息。例如，可用一个消息表达式来代替上面的 obj。

```
[[ obj msg1 ] msg2 ];
[[[ obj msg1 ] msg2 ] msg3 ];
```

[] 在 C 语言中是数组用的修饰符，而在 Objective-C 中则是消息表达式用的修饰符。[] 是否为数组用的修饰符，可以通过 [] 的左边是否存在数组名或指针等别的表达式来进行区分。下面展示了将一个消息表达式的返回值作为数组索引来使用的例子。

```
element = table[[ obj count ]];
```

消息是由**消息关键字**（message keyword）组成的，消息关键字的命名规则同变量名的命名规则一样。同函数调用一样，消息中也可以带有参数。没有参数的消息就只有消息名。让我们来看看下面的几个例子。

```
[aString copy];
width = [node width];
[[doc filename] retain];
```

在第二个例子中，变量名和消息关键字都是 width，这样也没有关系，消息表达式中会进行区分。

消息关键字的末尾有“:”时，表示这个消息带有参数。“:”的后面紧跟着的就是实际的参数。和函数调用一样，参数可以是一个变量，甚至也可以是一个消息表达式。

```
[printInfo setLeftMargin: 60.0];
[[[cw window] firstResponder] copy: sender];
[doc isSameDirectory:[info objectAtIndex: ++num]];
```

发送带有多个参数的消息的时候，可通过多个“关键字 : 变量”的写法完成调用（例如：方法名关键字 1: 变量 1 关键字 2: 变量 2）。另外，也可以省略关键字而只通过“:”将变量值连接起来（例如：方法名 : 变量 1: 变量 2）。

```
cell = [albumview cellAtRow:i column:j];
[manager fileExistsAtPath:dirname isDirectory:&isdir];
[view lineTo: 1.4142 : (y + 1.0)];
```

2.1.2 消息选择器

函数是通过函数名来区分的，消息则是通过消息名来区分的。消息名又称为**消息选择器**（message selector）、**选择器**（selector）或**方法**（method）。

下面我们来看看上述消息表达式例子中每个消息的选择器。另外，带有参数的消息选择器中要包含“:”，`copy` 和 `copy:` 是两个不同的选择器。

```
copy
retain
firstResponder
copy:
objectAtIndex:
cellAtRow:column:
fileExistsAtPath:isDirectory:
lineTo::
```

消息名的命名规则同变量名的命名规则一样，一般都习惯使用小写字母开头。如果选择器包含多个单词，则第一个单词以小写字母开头，后面的每个单词的首字母都要大写。使用这种方式命名，即使单词之间不留有空格，也能够明白这个单词串的意思[①]。Objective-C 借鉴了很多 Smalltalk 的特征，上述这种消息的表示方式也是从 Smalltalk 中借鉴而来的。

带有参数的消息，通常把消息的关键字按照英文的语序来组织。另外，有多个参数的情况下，消息名中会指出每个参数的含义。例如`cellAtRow:column:`这个选择器指的就是一个指定行列的表格，选择器清楚地指明了哪个参数用于行，哪个参数用于列。这个方法名翻译成中文就是“在某行某列的表格”。

如果消息名不省略参数的关键字，选择器就会变得很长，但这提高了程序的可读性，程序员在

① 更多关于命名规则的内容请参考附录 C 中的内容。

编程的时候就不需要再不断地查看参考手册以确认第三个参数是干什么用的等。另一方面，编程的时候不能写错选择器的名字，为此，使用复制和粘贴是一个好办法。一个好消息是 Mac OS X 下的 Xcode[①]（一个用于开发 Objective-C 程序的 IDE）中提供了方法名自动补全的功能，大家可以对其加以灵活应用。

另外，由多个消息关键字组成的选择器，其关键字是不能颠倒顺序的，例如：`dragFile:fromRect:slideBack:event:` 和 `dragFile:event:slideBack:fromRect:` 是两个完全不同的选择器。

2.1.3 实例变量的生成和初始化

只声明一个 id 类型的变量时，该变量并不指向任何对象。因此，在使用一个对象之前，首先就需要调用类的构造函数以生成对象。

Objective-C 通过向类发送消息来创建一个对象。向类发送消息的详细介绍请参考 4.5 节中的内容。目前只需要记住像下面这样给类名发送一个`alloc`消息就可以生成一个这个类的对象。

语法	实例的生成
`[ 类名 alloc ]`	

alloc 执行后即可完成对象所需要的存储空间的分配，但还没有对对象进行初始化。初始化对象用的方法叫作**初始化方法**（initializer）。不同的类会提供不同的初始化方法，同一个类也可能提供多个初始化方法。

Cocoa 中的初始化方法通常都是`init`或以 `init` 开头的函数。下面的语句完整地创建并初始化了 Cocoa 中某个类的一个对象。

语法	Cocoa 中某个类的对象的生成
`[[ 类名 alloc ] init ]`	

通常嵌套调用`alloc`和`init`来生成一个对象。

另外，对象生成后只调用一次初始化方法，来为对象中的各个变量设置初值。初始化方法并不具备将对象中的的信息重置（reset）的功能。如果需要将对象中的属性重置（使实例变量的值恢复为初始值），则需要额外实现一个重置专用的函数。

另外，有的类也可以不通过 alloc 而使用别的方法来生成对象，有的类也有可能会返回一个初始化好的对象。

① Mac OS X 的 X 和英语 ten 的发音一样，而 Xcode 则读为 [Xkod]。

2.2 类的定义

2.2.1 类的接口声明

下面我们来看看 Objective-C 中的类是如何定义的。Objective-C 中**接口**（interface）和实现（implementation）是分离开的，我们先来看一下类的接口部分。

类的接口部分定义了类的实例变量和方法。类的接口声明通常声明为头文件，提供给要调用这个类的模块引用。接口的定义如下所示，@end 后面不需要加“;”。

语法 类的定义

```
@interface 类名 : 父类名
{
    实例变量的定义;
     ...
}
方法声明;
  ...
@end
```

在 Objective-C 里面，类接口的声明以编译指令 @interface 开头并以 @end 结束，所有的 Objective-C **编译指令**（compiler directive）都是以 @ 字符开头，以便和 C 语言的字符串区分。

Objective-C 的类的命名规则和 C 语言变量的命名规则一样。但 Objective-C 的类名习惯首字母大写。如前所述，方法名（选择器名）和实例变量名通常都以小写字母开头。

类名不能和变量名以及方法名相同。

更多关于“父类”的概念可以参考 3.1 节，目前这个例子里面我们使用 NSObject 作为父类，NSObject 是 Objective-C 中的根类。

实例变量的声明和 C 语言中定义变量的方法一样，使用“变量类型 变量名”这样的语法。id 类似于 Java 的 Object 类，可以转换为任何数据类型，也就是说，id 类型的变量可以存放任何数据类型的对象。类中实例变量的名字要尽量避免使用 i、aaa 这种没有任何意义的名字。

方法的声明和 C 语言中函数的属性声明是一样的。

首先我们来看一个没有参数且返回值是对象的方法的声明。在 Objective-C 里，方法的返回类型需要用圆括号“()”包住，当编译器看到减号或者加号后面的括号之后，就会认为这是在声明方法的返回类型。假设下面这个方法的方法名是`delegate`。

```
- (id)delegate;
```

参数的类型指定也是一样，用“()”括起类型名并放置在参数之前。一个具有两个整数型的参数，且返回值是 id 对象的方法`cellAtRow:column:`的定义如下所示，row 和 col 是参数。

```
- (id)cellAtRow:(int)row column:(int)col;
```

同 C 语言一样，当一个方法没有任何返回值时，我们用 void 来表示。

```
- (void)setAutodisplay:(BOOL)flag;
```

这个例子里面的 BOOL 是 Objective-C 的布尔类型，它的值为真值常量 YES（非 0）和假值常量 NO（0）。Objective-C 的 if 和 while 语句等都可以使用 BOOL 类型的表达式。

专栏：各种各样的布尔类型 COLUMN

早期的 C 语言中是没有布尔类型的，直到 C99 标准才增加了对布尔类型的支持，关键字为 _Bool。虽然 _Bool 也被用在 Core Foundation 框架中（详情请参考附录 A），但 _Bool 和 Objective-C 中的 BOOL 是不同的。另外，C++ 中也定义了自己的布尔类型——bool，编程的时候应避免混用这些不同的布尔变量。

Objective-C 的 BOOL 并不是一个基本类型，它是无符号 char 的一个 typedef（别名）。不过我们在编程的时候并不需要了解这些细节问题。

下面，让我们来看一个简单的接口的定义吧。

```
@interface Prawn : NSObject
{
    id  order;
    int currentValue;                /* 注释 */
}
- (id)initWithObject: (id)obj;
- (void)dealloc;
- (int)currentValue;
- (void)setCurrentValue:(int)val;
- (double)evaluation:(int)val;       // 注释
@end
```

如例中所示，类的方法名和实例变量名可以相同，这样的情况在 Objective-C 中很常见。

Objective-C 中的**注释**写法和 ANSI C 一样，有单行注释和成对注释两种。单行注释以双斜线（//）开头，双斜线后直到这行结尾的任何字符都是注释，将被编译器忽略。另一种写法是使用符号对（/* */），这种注释以 /* 开始，以 */ 结尾，开始 /* 和结束 */ 之间的所有字符都被作为注释来处理。

专栏：不指定方法的返回值 COLUMN

Objective-C 中声明方法时也可以省略方法的返回值，这时编译器会给没有指定返回值的方法加上一个默认的返回值，它的类型是 id。也就是说下面两个方法是等价的。

```
- (id)initWithObject:(id)obj;
- initWithObject: obj;
```

这种写法在 NEXTSTEP 时代很常见，但现在已经不推荐使用了。因此大家应尽量清楚地定义方法的返回值。对于 C 语言来说，如果省略了方法的返回值，编译器会自动为方法加上 int 类型的返回值，但这种写法也存在争议，在未来可能会被废除。

2.2.2 类的实现

类的**实现部分**以编译指令 @implementation 开始，以 @end 结束，具体语法如下所示。

语法 类的实现

```
@implementation 类名
方法的定义
    ...
@end
```

类的实现部分不需要再次声明父类（接口部分声明的时候必须写上父类）。实现部分包含了接口部分中声明的所有方法的实现。如果接口部分中没有定义任何方法，实现部分也可以为空，即不包含任何方法的实现。

方法实现部分的第一行代码和方法声明部分完全相同（方法声明时以“;”结尾，在实现的时候需要删除），之后接上 {}，并将具体的实现方法写在 {} 里面。一个简单的方法实现如下所示。

```
- (double)evaluation:(int)val
{
    double tmp = [order proposedBalance: val];
    if (currentValue > (int)tmp)
        tmp = [order proposedBalance: val * 1.25];
    return tmp;
}
```

方法内部可以自由使用类的实例变量。上例中的 order 和 currentValue 就是实例变量。

方法内部定义局部变量和 C 语言中定义局部变量的方法相同。但如果定义的局部变量和实例变量重名，实例变量就会被覆盖，这时将无法访问实例变量。方法的参数也是一样，如果参数名和实例变量重名，也会发生覆盖，所以要尽量使用不同的名字。

方法中的 self 指的是实例对象自身，是 Objective-C 内置的变量。self 支持赋值等操作，也可以被作为返回值返回。通过 self 可以调用类的实例变量和方法。

2.2.3 一个遥控器的例子

通过遥控器可以控制电视等电器的音量，这里我们就举一个音量类的例子来看看如何定义一个类。音量类共包含四个变量，分别是音量的最小值（min）、音量的最大值（max）、音量变化的幅度值（step）和当前音量（value）。音量类共包括四个方法，分别是用于初始化音量的最小值、最大值和变化幅度值的初始化方法（initWithMin），以及返回当前音量值大小的方法（value）、增大音量 D 的方法（up）和减小音量的方法（down）。

代码清单 2-1　音量类的接口部分

```
@interface Volume : NSObject
{
    int val;
    int min, max, step;
}

- (id)initWithMin:(int)a max:(int)b step:(int)s;
- (int)value;
- (id)up;
- (id)down;
@end
```

代码清单 2-2　Volume 类的实现部分

```
@implementation Volume
- (id)initWithMin:(int)a max:(int)b step:(int)s
{
    self = [super init];
    if (self != nil) {
        val = min = a;
        max = b;
        step = s;
    }
    return self;
}

- (int)value
{
    return val;
}

- (id)up
{
    if ((val += step) > max)
        val = max;
    return self;
}

- (id)down
{
    if ((val -= step) < min)
        val = min;
    return self;
}
@end
```

方法initWithMin:max:step:中第一行的 [super init] 表示调用父类（NSObject）的初始化消息。nil 表示对象的指针指向空（没有东西就是空）。父类如果返回 nil，则不进行初始化。更多详细内容请参考第 3 章。

方法initWithMin:max:step:、up和down的返回值都是 self 对象，所以可以继续接收消息

来调用嵌套方法。例如，假设变量 obj 中存储着类 Volume 的一个实例，则嵌套方法的调用如下所示。

```
[[[obj up] up] up];
```

但是，即使类的声明和实现都完成了，也无法生成可执行程序，同 C 语言一样，有 main() 函数才能生成可执行程序。一个用于测试音量类的 main 函数如下所示。

本书自带的代码中使用了“\”作为换行符，不同字体和编码下“\”有可能会发生变化，请大家注意这一点。

代码清单 2-3　用于测试 Volume 类的 main 函数

```
int main(void)
{
    id v, w;

    v = [[Volume alloc] initWithMin:0 max:10 step:2];
    w = [[Volume alloc] initWithMin:0 max:9 step:3];
    [v up];
    printf("%d %d\n", [v value], [w value]);
    [v up];
    [w up];
    printf("%d %d\n", [v value], [w value]);
    [v down];
    [w down];
    printf("%d %d\n", [v value], [w value]);
    return 0;
}
```

2.3 编译

2.3.1 简单的编译方法

下面我们来执行一下上一节介绍的程序。本书中的所有 Objective-C 程序都是在 Mac OS X 上编写执行的，但实际上就 Mac OS X 和 iOS 来说，Objective-C 程序的写法和执行的原则是一样的。

首先访问苹果的在线商店——Apple Store 下载并安装 Objective-C 的开发环境 Xcode。安装好之后就可以使用以 Xcode 为核心的各种开发工具了。Objective-C 使用 C 语言的编译器 clang[①] 来编译代码。

Xcode 是一个非常强大的集成开发环境，可以在其中创建、编译 Objective-C 的程序。但由于本书集中讲解 Objective-C，因此就不再详细介绍 Xcode 的使用方法了。下面说明终端下编译

① clang 的发音是 [klæŋ]。和其他 UNIX 系的操作系统一样也可以使用 gcc 或者 cc 来编译 Objective-C 的程序，但无法使用后面要介绍的 ARC 等功能。使用 clang 编译的时候，需要让宏定义 __clang__ 有效。

Objective-C 程序的方法。

编写 Objective-C 程序的编辑器有很多种，命令行下可以使用 vi 或 emacs，也可以在图形界面中使用 Mac OS X 自带的文本编辑器。另外，使用 Xcode 来编写代码也是一个不错的选择。

接下来我们开始编译上节中创建的 Volume 类。一种最简单的编译方法如图 2-1 所示，创建一个文件，把代码清单 2-1~2-3 的内容都放入到这个文件中。但一定要注意把接口声明（@interface）部分放在另外两部分的前面。图 2-1 中最前面两行使用 **#import** 引入了头文件，#import 和 C 语言中的 #include 具备相同的功能，都可以引入头文件。#import 优于 #include 的地方在于不会导致重复引入头文件，详情请参考 2.4.4 节。大家可能对 stdio.h 比较熟悉，而这里除了 stdio.h 之外，还要引入一个 Objective-C 用的头文件。

Objective-C 中包含实际代码的文件的扩展名为 .m，m 是模块的意思。图 2-1 中的文件就被保存为了 voltest.m。

Mac OS X 的 Cocoa 和以 Cocoa 为基础扩展而来的 iOS 的 Cocoa Touch 提供了类似的执行环境，本书中统称其为 Cocoa 环境。在 Cocoa 环境中，系统提供的类和函数被封装在**框架**（framework）中，并以动态库的形式被提供给应用程序调用。作为通用的面向对象的函数库，Cocoa 中的 Foundation 框架主要定义了一些基础类，如字符串、数值的管理、容器、文件系统等，在 Mac OS X 和 iOS 中都可以使用它。

图 2-1　单个文件编译方式的例子

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

@interface Volume : NSObject
    /*代码清单2-1的内容*/
@end

@implementation Volume
    /*代码清单2-2的内容*/
@end

int main(void)
{
    /*代码清单2-3的内容*/
}
```

用以下命令就可以编译程序 voltest.m 了。其中要注意的是，编译和链接的时候需要加上 -framework Foundation 的选项，说明你要使用 Foundation 框架中的内容。

```
% clang voltest.m -framework Foundation
```

编译成功后就会生成可执行程序 a.out。因为编译的时候编译器通过扩展名就能够区分要编译的文件到底是 C、C++ 还是 Objective-C 文件，所以并不需要特别指定。可以通过 -o 选项来指定生成的可执行文件名。另外，编译的时候也可以通过 -Wmost 或 -Wall 选项把隐藏的 warning 显示出来。

执行生成的 a.out 后，输出如下。

```
% ./a.out
2 0
4 3
2 0
```

2.3.2 多文件编译

为了简单起见，上面的例子中我们把类的接口部分、实现部分和 main 函数都放到了一个文件中。而通常情况下，Objective-C 中每一个类都会分成 .h 和 .m 文件。

接口文件

文件名为“类名 .h”，内容为类的接口部分。

实现文件

文件名为“类名 .m”，内容为类的实现部分。

也就是说，接口文件起到了头文件的作用。使用某个类时都需要包含该类的接口文件。类实现文件中也需要包含接口文件。

定义接口文件时需要父类的定义，关于这一点，我们将在第 3 章中详细说明。上面的例子中需要包含父类的接口文件 Foundation/NSObject.h。

而含有 main 函数的文件该如何命名呢？上面的 main 函数中虽然没有包含类的定义，但是使用到了消息表达式。因为消息表达式是 Objective-C 所特有的用法，使用了消息表达式的程序无法作为 C 语言编译，所以即使文件中不包含类的定义，但只要使用到了类或消息表达式，也都需要把文件的扩展名保存为 .m。这里我们把 main 函数的文件名保存为 main.m。

多文件编译时文件的构成如图 2-2 所示。请注意每个文件前面都引入了哪些头文件。

同 C 语言一样，编译之后还需要链接。链接模块 main.m 和 Volume.m 生成可执行程序 vol 的方法如下所示。

```
% clang -o vol main.m Volume.m -framework Foundation
```

编译和链接的整个过程如下所示。

```
% clang -c main.m
% clang -c Volume.m
% clang -o vol main.o Volume.o -framework Foundation
```

▶ 图 2-2 将类分成多个文件的定义方式

Volume.h

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

@interface Volume : NSObject
    /*代码清单2-1中的内容*/
@end
```

Volume.m

```
#import "Volume.h"

@implementation Volume
    /*代码清单2-2中的内容*/
@end
```

main.m

```
#import "Volume.h"
#import <stdio.h>

int main(void)
{
    /*代码清单2-3中的内容*/
}
```

2.4 程序的书写风格

2.4.1 混合编程

Objective-C 是在 C 语言的基础上加入了面向对象的功能，因此它既可以完全使用面向对象的程序设计风格，也可以混合使用 C 语言加面向对象的程序设计风格。基于这一特征，Objective-C 也被称为混合编程语言。

Objective-C 是以能够兼容 C 为目标而开发的，所以还可以使用 C 语言的编程风格，但既然加入了面向对象的功能，程序中面向对象的部分还是使用面向对象的编程风格为好。

下面这几种情况下，建议在 Objective-C 中使用 C 语言的函数。

- **想使用成熟的 C 语言函数模块时**
- **想使用以 C 语言声明的接口时，例如 Unix 的系统调用等**
- **和面向对象没有关系，用于数学、计算等时**

  例如一个用于三角函数计算的函数，我们不需要特意将其实现为某个类的方法。

- **类定义时**
- **对速度有较高要求时**

  虽然 Objective-C 的方法调用的速度也不慢，但函数调用的速度更快一些。因此，如果将系统中那些需要被反复调用上万甚至上百万次的功能改成函数调用，速度就会快很多，而除此以外的绝大部分则都没有必要改成函数调用的形式。

Objective-C 允许以面向对象的思维对软件进行整体设计，并在实际开发的时候使用 C 语言来实现其中的关键模块。而至于如何把面向对象的部分和通过函数实现的部分完美统一，那就要看程序员的水平了。

2.4.2 C 语言函数的使用方法

Objective-C 无论是在函数中还是方法中都可以自由地调用 C 语言的函数。另外，如测试 Volume 类的 main 函数所示（代码清单 2-3），在 C 语言的函数中也可以调用 Objective-C 类的方法。而且还可以创建参数和返回值都是 id 类型的函数。

和普通的 C 语言程序一样，把只含有 C 语言的文件命名为 .c 文件即可。但如果源文件中包含了方法的调用或类定义，那么 .c 文件就无法被编译，因此需要将其命名为 .m 文件。同时也需要载入类的头文件。

类的方法中调用 C 语言函数时，只需要能够读取该函数的相关信息即可。这和普通的 C 语言程序完全一样。

扩展名为 .c 和 .m 的文件可以一起编译、链接，如下所示。

```
% clang -o sample main2.m util.c Volume.m -framework Foundation
```

类的实现文件中可以定义 C 语言风格的函数，但仅限于和类的实现有直接关系的函数。如图 2-3 所示，函数可以定义在 @implementation 的前面、@implementation 和 @end 之间或 @end 的后面的任何位置。如果函数的作用域仅限于这个文件，那么给函数加上 static 修饰符比较好。另外，因为上例中的函数 funC 实现比较靠后，所以需要在前面加上前置声明。

类的实现文件中定义的函数不可以直接使用类中定义的实例变量和 self 变量，但可以把这些作为参数传递给函数[①]。访问实例变量和 self 变量的操作都应该被定义在类的方法中。

类中共有的功能更适合被定义成类的方法，而不是一个函数。关于类方法的更多内容，请参考第 4 章。

① @ implementation 和 @end 之间定义的函数，就实例变量的现性而言和方法的处理一样。关于这一点，请参考 4.4 节。

图 2-3 实现文件内定义函数的例子

```
#import "MyExample.h"

static void funcA(int a, char *p)
{
        /*函数定义*/
}

@implementation MyExample

static double funcB(int n)
{
        /*函数定义*/
}

- (id)myMethod
{
        /*方法定义*/
}
    ...
@end

static id funcC(id obj)
{
        /*函数定义*/
}
```

2.4.3 静态变量的定义

在函数或类方法范围外定义的变量以及指定了 static 的变量都是静态变量。静态变量的生命期为从程序开始执行到结束。

类方法中可以使用带有 static 修饰符的静态变量，但需要注意同 C 语言有不一样的地方。

无论生成了多少个对象，都只有一个静态变量存在，也就是说多个对象会共享同一个静态变量。一个对象对静态变量赋值后，在使用这个静态变量之前，如果另外一个对象修改了该静态变量的值，那么不进行同步的话就会发生错误。C 语言中经常使用静态变量来完成一些特殊的功能，但如果不认真区分，反而会导致错误的发生。

利用静态变量的这种性质可以实现对象间的信息共享和消息传递等。关于类的实现文件中将静态局部变量作为类变量使用的详细内容，请参考 4.5 节。

2.4.4 头文件的引入

Objective-C 使用 #import 来引入头文件，#import 和 #include 的基本功能一样，稍有不同的是 #import 不存在嵌套引用的问题。

不仅仅是 Objective-C，在 C 语言中，当软件的规模变大而需要多个头文件时，很有可能就会出现头文件被重复引入的问题。同一个头文件被重复引入后，就会出现二重定义的错误。C 语言中一

般通过在头文件的最开始加上宏定义的方法来避免这个问题，如下例所示。

```
#ifndef __MYLIB_H__
#define __MYLIB_H__
    // 头文件的定义
#endif /* __MYLIB_H__ */
```

__MYLIB_H__ 是这个头文件专用的宏定义。当头文件第一次被引入的时候，因为这个宏名还没有被定义，所以 #ifndef 和 #endif 之间的内容会被读入，同时 __MYLIB_H__ 也会被定义。在这之后，当头文件再次被引入的时候，因为 __MYLIB_H__ 已被定义，所以 #ifndef 和 #endif 之间的内容也就不会被再次加载了。

不过这种方法比较麻烦，而且容易出错。Objective-C 中的 #import 内置了判断同一个文件是否已被引入的功能，据此，不使用宏定义的方法也能够防止同一个文件被重复引用。

专栏：参考文档和 SDK COLUMN

Cocoa 各个类的参考文档可以在苹果开发者联盟（ADC，Apple Developer Connection）上面找到。下面列出了几个常用的网址（网址有可能会发生变化，具体请以苹果公司网站上公布的信息为准）。多数开发文档对非 ADC 会员也是公开的。

Apple Developer Connection Home
http://developer.apple.com/

Mac OS X Developer Library
http://developer.apple.com/library/mac/navigation/

iOS Developer Library
http://developer.apple.com/library/ios/navigation/

中文资源
http://developer.apple.com/cn/resources/

上面的这些参考文档可以直接使用浏览器访问。使用 Xcode 开发 Mac OS X、iPhone 和 iPad 的应用时，也能够方便地使用这些参考文档。在 Xcode 中的“Help”菜单中选择“Documentation and API Reference”就会显示这些文档。另外，Xcode 也提供了很多方便编程的功能，例如输入时能够自动显示选择器备选、在源代码上单击方法和函数的定义就能够跳转到文档中等。

注册为免费的 ADC 会员后，就可以查看 API 文档和例子程序，也可以利用 iPhone 或 iPad 的开发环境将开发好的应用在 Mac 提供的模拟器上运行。但如果想将开发好的应用真机调试或将应用发布到 App Store，就一定要注册为 ADC 的收费会员。

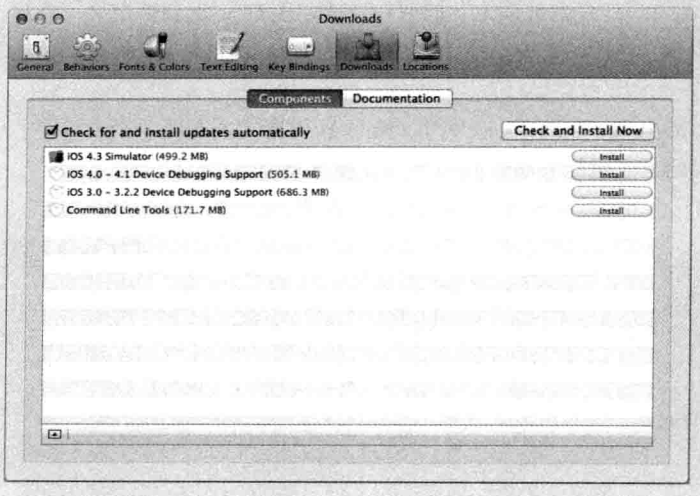
有关免费和收费会员的详细情况请参考下面的 URL。

Register with Apple Developer
http://developer.apple.com/jp/programs/register/

专栏：Xcode 的安装 COLUMN

从 Xcode 4.3 开始，开发工具所在的目录等发生了很大的变化。从命令行启动的各种开发工具也是一样，Xcode 安装之后也必须进行追加安装（需要连接网络）。下面简单地介绍一下安装的过程。

首先启动 Xcode，选择“Preferences..”菜单，点击菜单上半部分的“Downloads”会显示下面的列表，然后点击“Command Line Tools”右边的“Install”按钮就会开始下载安装。如果需要，其他的 iOS 模拟器等也可以用同样的方法安装。



第3章

类和继承

本章讲述面向对象中的一个重要概念——继承，使用继承可以方便地在已有类的基础上进行扩展，定义一个具有父类全部功能的新类。

3.1 继承的概念

3.1.1 父类和子类

我们在定义一个新类的时候，经常会遇到要定义的新类是某个类的扩展或者是对某个类的修正这种情况。如果可以在已有类的基础上追加内容来定义新类，那么新类的定义将会变得更简单。

像这种通过扩展或者修改既有类来定义新类的方法叫作**继承**（inheritance）。在继承关系中，被继承的类称为**父类**（superclass），通过继承关系新建的类称为**子类**（subclass）。

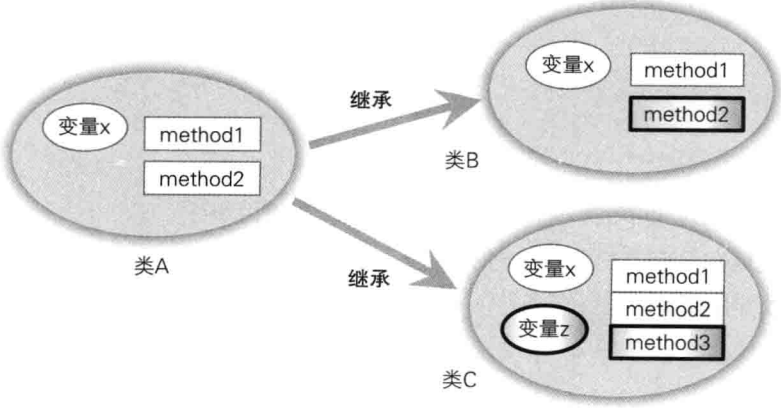
继承意味着子类继承了父类的所有特性，父类的数据成员和成员函数自动成为子类的数据成员和成员函数。除此之外，子类还可以

- **追加新的方法**
- **追加新的实例变量**
- **重新定义父类中的方法**

当然，如果子类中只追加新的实例变量而不变更方法则没有任何意义。子类中重新定义父类的方法叫作**重写**（override）。

让我们来看几个例子。在图 3-1 中，类 B 是类 A 的子类，类 B 继承了类 A 的实例变量和方法，但重写了 method2。类 C 也是类 A 的子类，类 C 中增加了新的实例变量 z 和新的方法 method3。类 B 和类 C 都是类 A 的子类，无论类 A、类 B 和类 C 的任何一个实例变量都能够执行方法 method1 和 method2。

图 3-1 继承的概念



父类和子类是一种相对的称呼。例如，在上例中，如果以类 B 为父类又派生出一个子类 D，那么类 B 相对于类 A 是子类，相对于类 D 却为父类。

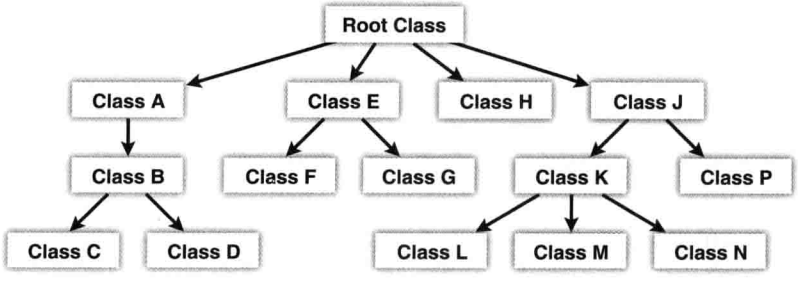
另外，在集合用语中，子集指的是比较小的集合（相对于父集），但在类的情况下子类一般是父类的扩展。为了避免这种命名上的混乱，C++ 中把父类称为**基类**（base class），把子类称为**派生类**或**导出类**（derived class）。考虑到面向对象的程序设计中一般都使用父类、子类的叫法，本书也使用这种叫法。

3.1.2 类的层次结构

假如以某个类为父类生成若干子类，然后再继承这些子类并生成更多的子类，如此循环下去就可能会生成一颗倒立的树，它由通过继承而彼此关联的类组成，这样的树称为**类层次结构**（class hierarchy）。

位于类层次最顶端的类称为**根类**（root class），如图 3-2 所示。

图 3-2 类层次结构的概念



NSObject 是 Cocoa 环境下的根类，Cocoa 中所有的类都直接或间接地继承了 NSObject[①]。新建的任何类都必须是 NSObject 或它的继承类的子类。NSObject 中定义了所有 Objective-C 对象的基本方法。

由于这种类的层次关系，Objective-C 的所有对象都继承了 NSObject 类中定义的各种属性。Objective-C 的对象能够作为对象来使用，就是因为类 NSObject 中定义了对象的基本功能。

在面向对象的语言中，有的和 Objective-C 一样有唯一根类，例如 Java 和 Smalltalk 等；有的则不存在唯一根类，如 C++。

更多关于根类的介绍，请参考 8.1 节的内容。

① 实际上，除了 NSObject 之外，Cocoa 环境中还有一个根类 NSProxy（详细内容请参考第 19 章）。

3.2 利用继承定义新类

3.2.1 继承的定义

如果想通过继承为某个类定义一个子类，该怎么办呢？

Objective-C 在子类的接口部分声明继承关系。在 2.2 节中我们已经说明了如何定义类的接口，这里再介绍一遍。

语法 类定义

```
@interface 类名 : 父类名
{
    实例变量的声明;
      ...
}
方法的声明;
  ...
@end
```

定义父类 A 的子类 B 的时候，“类名”是新类 B，冒号后面的“父类名”是需要继承的类 A。

至此为止本书中的父类都使用了 NSObject，这是因为 Objective-C **中所有的类都要继承根类，而 NSObject 是 Objective-C 中所有类的根类**。如果子类有想继承的类，就要直接指明该类为父类，否则就需要指定 NSObject 为父类。前文中定义 Volume 类的时候，因为 Volume 类并没有特别想继承的类，所以直接使用了 NSObject 作为父类。

实例变量的声明中只需要声明新增的变量。如果没有新增的变量，则只需要加上 {} 即可，有时甚至连 {} 都可以省略。

方法的声明中只需要追加新增的方法。如果要覆盖父类中已声明的方法（重写），则需要在接口中对方法重新声明。通常我们会给重写的方法加上注释，以便理解。

下面展示了定义类 A 的子类 B 时接口部分的情况。变量 x 和方法 method1 继承于类 A，所以不需要重新声明，方法 method2 的声明也可以被省略。

```
@interface B : A
- (void)method2;    // 这个方法被覆盖了
@end
```

图 3-1 中定义类 C 时的接口部分如下所示，需要对变量 z 和方法 method3 进行声明。

```
@interface C : A
{
    id    z;
}
- (void)method3;
@end
```

3.2.2 类定义和头文件

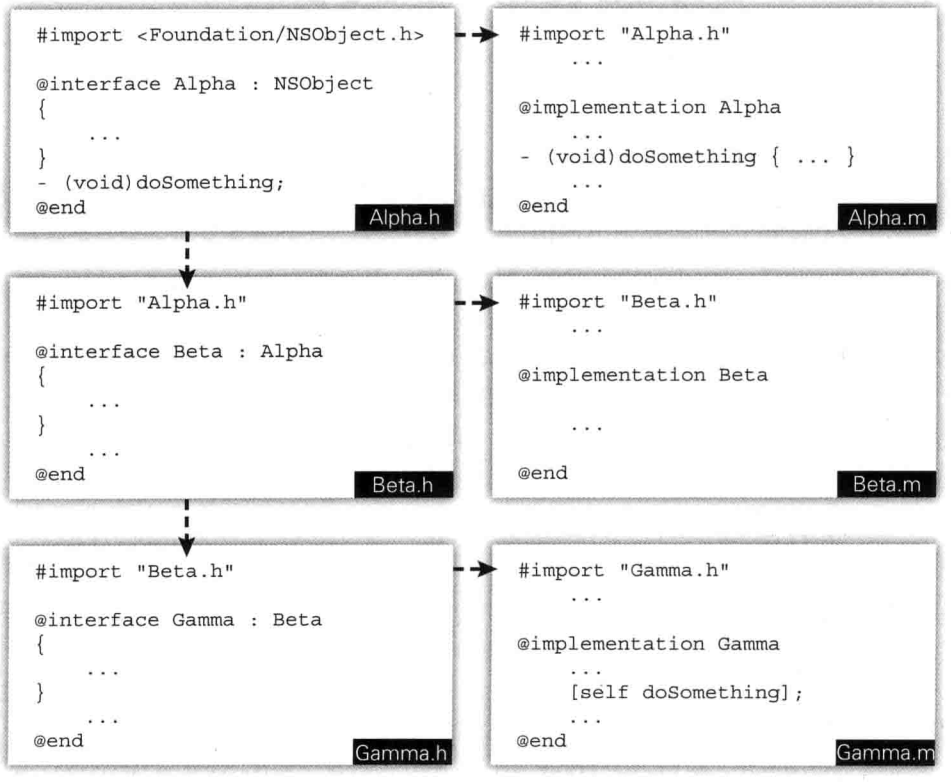
上一章介绍多文件编译时我们提到过接口部分通常都被声明为一个头文件，而这对继承来说也是很重要的。

假设有一个已经定义好了的类 Alpha，那么头文件 Alpha.h 就应该已经存在。要定义类 Alpha 的子类 Beta 的时候，头文件 Beta.h 中必须包含 Alpha.h。不知道父类定义的话是无法定义子类的。所以**包含父类接口的头文件是必须的**。

类的实现部分必须引入包含类的接口部分的头文件。实现部分需要包含新增和重写的方法的实现。当然实现部分也可以定义各种局部函数和变量。

图 3-3 的文件 Gamma.m 的方法中调用了方法doSomething，这个方法是从类 Alpha 继承而来的。文件 Gamma.m 引入的头文件 Gamma.h 中引入了 Beta.h，Beta.h 中又引入了 Alpha.h，所以 Gamma.m 可以调用方法doSomething。

图 3-3 继承和头文件的关系



类的定义可以不断地使用继承向下扩展，但无论怎么扩展，只要保证了这种头文件的引入方式，任何一个派生类中就都能使用父类中定义的变量和方法。

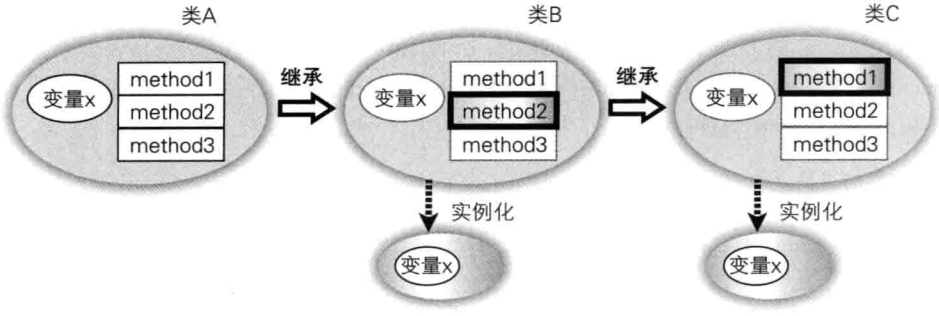
3.2.3 继承和方法调用

子类中定义的方法，除了能够访问新追加的实例变量外，也能够访问父类中定义的实例变量。

另外，因为继承的原因，子类也可以响应父类中定义的消息。但如果子类中重写了父类的方法，就需要注意实际运行中到底哪个方法（父类的还是子类的）被执行了。

如图 3-4 所示，类 A 包含方法 method1、method2、method3。类 B 是类 A 的子类，类 B 中重新定义了 method2。类 C 是类 B 的子类，类 C 中重新定义了 method1。

图 3-4 继承和方法



我们来看看给类 B 的实例变量发送消息时的情况。首先，假设向类 B 的实例对象发送了对应 method1 的消息，即进行了方法调用。虽然类 B 中没有 method1 的定义，但因为类 B 的父类类 A 中定义了 method1，所以会找到类 A 的 method1，调用成功。消息 method3 的情况下也是同样的道理，类 A 中定义的 method3 会被执行。method2 同前两个消息不同，类 B 中定义了 method2，所以会使用自身定义的 method2 来响应这个消息。

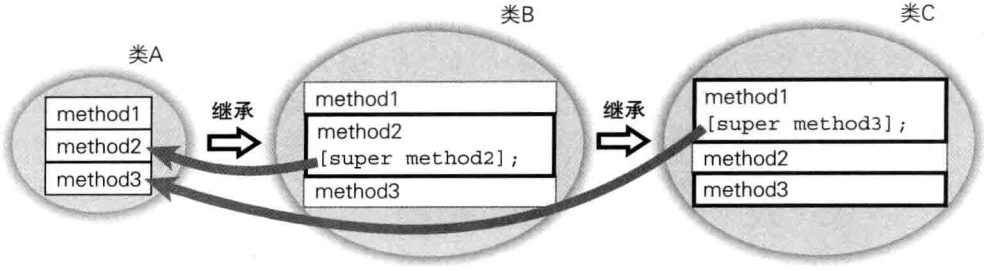
而给类 C 的实例发送消息的话会怎么样呢？类 C 中有 method1 的定义，所以会直接使用类 C 中定义的 method1 来响应这个消息。类 C 中没有 method2 的定义，所以调用的时候会使用类 B 中定义的 method2 来响应。类 C 和类 B 中都没有定义 method3，所以类 A 中的定义 method3 会被调用。

3.2.4 调用父类的方法

子类继承了父类之后，有时就可能会希望调用父类的方法来执行子类中定义的其他处理，或者根据情况进行和父类一样的处理或子类中单独定义的处理。让我们来看看图 3-4 中的例子，如果要在类 B 的 method2 的定义中调用类 A 的 method2，那么该怎么办呢？通过 self 调用 method2 的话，就会变成递归调用自身定义的 method2。

如果子类中想调用父类的方法，可以通过 super 关键字来发送消息。使用 super 发送消息后，就会调用父类或父类的父类中定义的方法。如图 3-5 所示，类 C 中定义了 method1 和 method3。类 C 的 method1 中通过 super 调用了 method3，这时被调用的 method3 是类 A 中定义的 method3。

图 3-5 使用 super 发送消息



super 和 self 不同，并不确定指向某个对象。所以 super 只能被用于调用父类的方法，不能通过 super 完成赋值，也不能把方法的返回值指定为 super。

3.2.5 初始化方法的定义

新追加的实例变量有时需要被初始化。另外，子类也可能需要同父类不同的初始化方法。这些情况下就需要为子类定义自己的初始化方法。

子类中重写 init 初始化方法的时候，通常按照以下逻辑。其他以 init 开头的初始化方法也是同理。

```
- (id)init
{
    self = [super init];   /* 一定要在第一行调用父类的 init 方法 */
    if (self != nil) {     /* 父类返回了初始化好的实例时 */
      ...                  /* 子类专有的初始化操作 */
    }
    return self;
}
```

请注意第一行调用了父类的init方法，父类的init方法会初始化父类中定义的实例变量。下面是子类专有的初始化操作。

如果所有的类的初始化方法都这样写，那么根类 NSObject 的init方法就一定会被执行。否则生成的对象就无法使用。与此同时，这样做也可以防止漏掉父类中定义的实例变量的初始化。

执行的时候父类的初始化方法可能会出错。出错时则会返回 nil，这种情况下子类也不需要再进行初始化，直接返回 nil 就可以了。

如果父类是 NSObject，则基本上不可能初始化出错，因此不判断这个返回值也是可以的。使用传入的参数或通过从文件读入变量进行初始化时，因为值的类型错误或读取文件失败等原因，初始化有可能会失败。这种情况下，需要确认父类的初始化方法的返回值。另外，上例中对 self 进行了赋值，关于这个赋值的含义我们会在第 8 章中详细说明，这里只需要记住这是初始化方法的一种固定写法即可。

生成实例对象的方法alloc会把实例对象的变量都初始化为 0（后面会提到的实例变量 isa 除外）。所以，如果子类中新追加的实例变量的初值可以为 0，则可以跳过子类的初始化。但是为了明

确是否可以省略，最好为初值可为 0 的变量加上注释。

从程序的书写角度来说，设定初始值的方法有两种，即可以在初始化方法中一次性完成实例变量的初始化，也可以在初始化方法中先设置实例变量为默认值，然后再调用别的方法来设置实例变量的值。例如，类 Volume 也可以通过先调用初始化方法`init`，然后再调用`setMax:`等方法来设定音量的最大值、最小值和变化幅度。原则上来说，初始赋值之后值不再发生变化的变量和需要显示设定初值的变量，都需要通过带参数的初始化方法来进行初始化。

3.3 使用继承的程序示例

3.3.1 追加新方法的例子

我们来定义一个带有静音功能的类 MuteVolume。该类只有一个功能，即当收到`mute`消息时，设置音量为最小。

类 MuteVolume 的定义非常简单，父类是已经定义好的类 Volume。子类 MuteVolume 除了可以使用父类 Volume 中定义的所有实例变量和方法之外，还新增加了一个 mute 方法。

代码清单 3-1 文件 MuteVolueme.h - 版本 1

```
#import "Volume.h"

@interface MuteVolume : Volume  /* 父类是 Volume*/
- (id)mute;
@end
```

这里使用了 Volume 作为父类，并引入了头文件 Volume.h。Volume 的父类是 NSObject，所以 MuteVolume 也是 NSObject 的派生类。因为 Volume.h 中已经引入了 Foundation/NSObject.h（图 2-2），所以就不需要再进行指定了。

没有定义新的实例变量，意味着子类中没有要追加的实例变量。

代码清单 3-2 文件 MuteVolume.m - 版本 1

```
#import "MuteVolume.h"

@implementation MuteVolume

- (id)mute
{
    val = min;
    return self;
}

@end
```

代码清单 3-3 用于测试类 MuteVolume 的 main 程序

```
#import "MuteVolume.h"
#import <stdio.h>

int main(void)
{
    id v;
    char buf[8];

    v = [[MuteVolume alloc] initWithMin:0 max:10 step:2];
    while (scanf("%s", buf) > 0) {
        switch (buf[0]) {
        case 'u': [v up]; break;
        case 'd': [v down]; break;
        case 'm': [v mute]; break;
        case 'q': return 0;
        }
        printf("Volume=%d\n", [v value]);
    }
    return 0;
}
```

该测试程序的功能是从终端读入输入的字符串，并根据字符串的第一个字符来决定如何设置音量。具体来说，第一个字符为 u 时表示提高音量，d 表示降低音量，m 表示静音，q 表示退出程序。

编译子类的时候，需要连同父类一起编译和链接，否则就无法使用父类中定义的方法。本例中编译所需要的文件一共有 5 个，即 Volume.h、Volume.m、MuteVolume.h、MuteVolume.m、main.m。

```
% clang main.m Volume.m MuteVolume.m -framework Foundation
```

3.3.2 方法重写的例子

上面通过继承实现静音功能类的例子非常简单，让我们来看一个更实用的例子。

假设该例子要实现两个功能。第一个功能是，当再次收到mute消息时，音量会恢复原值；第二个功能是，在静音状态下收到up或down消息时，会返回最小音量值，同时改变音量值。

实现这些功能的方法有很多，这里我们增加一个 BOOL 类型的变量 muting，同时修改方法initWithMin:max:step:和方法value的实现。

代码清单 3-4 文件 MuteVolume.h - 版本 2

```
#import "Volume.h"

@interface MuteVolume : Volume  /*MuteVolume 的父类是 Volume*/
{
    BOOL   muting;
}
```

```
/* override */
- (id)initWithMin:(int)a max:(int)b step:(int)s;
- (int)value;

- (id)mute;
@end
```

代码清单 3-5 文件 MuteVolume.m - 版本 2

```
#import "MuteVolume.h"

@implementation MuteVolume

/* override */
- (id)initWithMin:(int)a max:(int)b step:(int)s
{
    self =[super initWithMin:a max:b step:s];
    if (self != nil)
        muting = NO;
    return self;
}

/* override */
- (int)value
{
    return muting ? min : val;
}

- (id)mute
{
    muting = !muting;
    return self;
}

@end
```

初始化方法initWithMin:max:step:首先调用了父类的初始化方法，然后对新增的实例变量muting进行了初始化。如前所述，子类的初始化一定要在父类的初始化之后进行。

value方法根据当前是否为静音状态返回不同的值。静音状态下，返回最小值min。mute方法中只需要改变实例变量muting的状态来标识是否静音，不需要更改音量值val。

编译的情况和上一节一样。main.m直接使用上一节的即可。

3.4 继承和方法调用

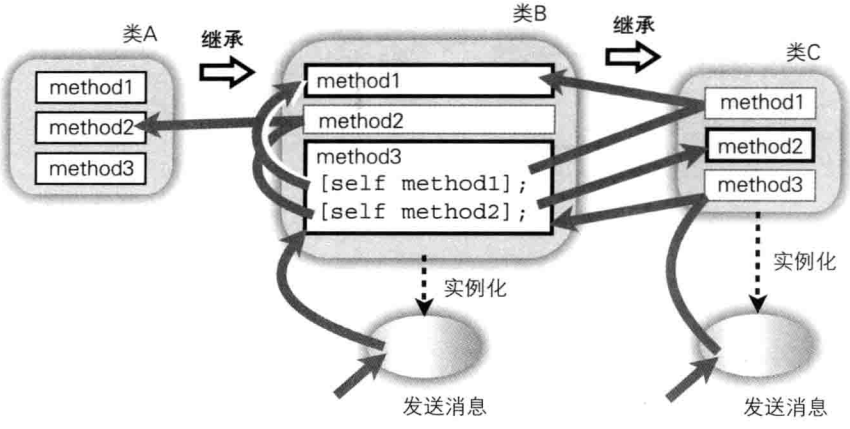
3.4.1 使用 self 调用方法

如果想在一个方法中调用当前类中定义的方法，可以利用self。但如果存在继承关系，通过self调

用方法时要格外注意。

在图 3-6 的例子中，有三个类 A、B、C。类 A 中定义了 method1、method2 和 method3 三个方法。类 B 继承了类 A，重写了 method1 和 method3。类 C 继承了类 B，重写了 method2。

图 3-6 类的继承和 self



假设类 B 的方法 method3 想调用 method1 和 method2，通过 self 调用了 method1 和 method2。我们来分析一下这个过程中到底哪个函数被调用了。对类 B 的实例对象调用 method3 方法时，首先会通过 self 调用 method1，这个 method1 就是类 B 自身定义的 method1。接着，method3 通过 self 调用 method2，因为类 B 中并没有 method2 的定义，所以就会调用从类 A 中继承而来的 method2。

而如果是类 C 的实例对象调用方法 method3 的话会怎么样呢？我们首先来看看 method3，因为类 C 中并没有定义 method3，所以调用的是类 B 中定义的 method3。要注意这个时候 self 指的是类 C 的实例对象，当 `[self method1]` 执行时，因为类 C 中没有定义 method1，所以调用的是类 B 中定义的 method1。然后，当 `[self method2]` 执行时，因为类 C 中定义了 method2，所以执行的是类 C 中定义的 method2，而不是上例中类 A 中定义的 method2。另外还有一点需要注意，就算类 B 中定义了 method2，调用的也是类 C 中定义的 method2。

也就是说，self **指的是收到当前消息的实例变量**，因此，就算是同一个程序，根据实例的类的不同，实际调用的方法也可能不相同。

使用 self 的时候要一定小心，要仔细分辨到底调用了哪个类的方法。即便如此，利用 self 的特性来编程也是很常见的，更多详细内容请参考 11.1 节的内容。

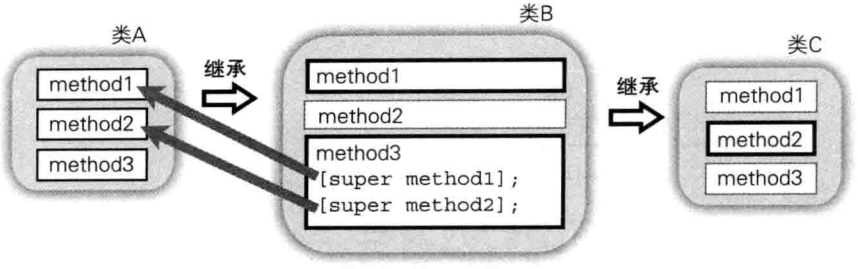
3.4.2 使用 super 调用方法

而如果不使用 self 而使用 super，程序执行的结果会怎样呢？

图 3-7 是用 super 替代图 3-6 中的 self 的情况。使用 super 调用方法时，最后被调用的方法是类 B 的父类中定义的方法。所以无论是类 B 还是类 C 的实例变量调用了 method3，最后调用到的都是类 A 中定义的 method1 和 method2。

super 调用的是父类的方法，而至于到底调用了哪个方法则是**由编译时类的继承关系决定**的。

图 3-7 类的继承和 super



3.4.3 测试程序

我们用一个简单的程序来验证一下上面所描述的内容。这个程序本身并没有太大的意义，仅仅是用来测试方法调用的。

测试程序中有三个类 A、B、C。类 A 中定义了方法 method1 和 method2。类 B 中对 method1 进行了重写，通过 self 调用了 method1，通过 super 调用了 method2。类 C 重写了 method1。

代码清单 3-6 文件 testself.m

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

@interface A: NSObject
- (void)method1;
- (void)method2;
@end

@implementation A
- (void)method1 { printf("method1 of Class A\n"); }
- (void)method2 { printf("method2 of Class A\n"); }
@end

@interface B: A
- (void)method2;
@end

@implementation B
- (void)method2 {
    printf("method2 of Class B\n");
    printf("self --> ");
    [self method1];
    printf("super--> ");
    [super method2];
}
@end
```

```
@interface C: B
- (void)method1;
@end

@implementation C
- (void)method1 { printf("method1 of Class C\n"); }
@end

int main(void)
{
    id x = [[B alloc] init];
    id y = [[C alloc] init];
    printf("--- instance of B ---\n");
    [x method1];
    [x method2];
    printf("--- instance of C ---\n");
    [y method1];
    [y method2];
    return 0;
}
```

程序执行之后输出如下。可以看出，类 B 和类 C 的实例分别调用了不同的方法。

```
--- instance of B ---
method1 of Class A
method2 of Class B
self --> method1 of Class A
super--> method2 of Class A
--- instance of C ---
method1 of Class C
method2 of Class B
self --> method1 of Class C
super--> method2 of Class A
```

3.5 方法定义时的注意事项

3.5.1 局部方法

实现接口声明中的方法时，可把具备独立功能的部分独立出来定义成子方法。一般情况下，这些子方法都只供内部调用，不需要包含在类的接口中对外公开。

这种情况下，局部方法可以只在实现部分（通常是 .m 文件）中实现，而不需要在接口部分中进行声明。这样一来，就算其他模块引用了接口文件，也无法获得这个方法的定义，无法调用这个方法，从而就实现了局部方法。但这里只是说无法从接口中获得这个方法的定义，这个方法本身还是

存在的，只要发送了消息，就能够执行。

让我们来看一个简单的例子，类 ClickVolume 是类 Volume 的一个子类，它的主要功能是当音量发生变化（提高或降低）时发出提示音。提高或降低音量时发出提示音使用一个共同的方法playClick，定义如下所述。因为这个功能不会在其他地方使用到，所以我们把它定义成一个局部方法，不在接口文件中声明。

```
@interface ClickVolume: Volume
- (id)up;              //playClick 方法没有在这里声明
- (id)down;
@end

@implementation ClickVolume
- (void)playClick { // 类内部定义的局部方法
    /* 发出提示音 */
}
- (id)up {
    [self playClick];
    return [super up];
}
- (id)down {
    [self playClick];
    return [super down];
}
@end
```

未在接口中声明的局部方法和没有进行属性声明的 C 语言函数一样，只能被定义在局部方法之后的方法调用。在上面的例子中，playClick就必须定义在up和down的前面。定义顺序方面出现的问题，可以使用第 10 章介绍的“范畴”（category）来解决。

编程的时候使用局部方法可以增强程序的可维护性，但在继承的时候可能会出现问题。例如，子类新追加的方法可能并不知道父类已经实现了局部方法而去重新实现一个父类的局部方法。

为了避免这一问题，苹果公司建议为局部方法名添加固定的前缀（详情请参考附录 C）。

3.5.2 指定初始化方法

前面已经介绍过了如何定义初始化方法，但还有一些要注意的地方。

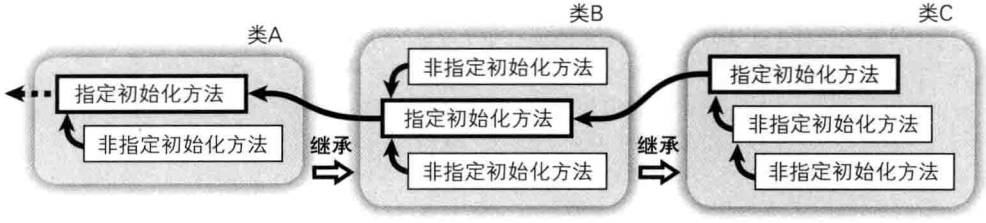
根据需求有时可能需要为一个类定义多个不同的初始化方法。例如，既需要提供一个可指定每个参数初始值的初始化方法，又需要提供一个每个参数都直接使用默认值的初始化方法；既需要提供一个用内存变量进行初始化的初始化方法，又需要提供一个能从文件读入变量完成初始化的初始化方法等。**指定初始化方法**（designated initializer）就是指能确保所有实例变量都能被初始化的方法，这种方法是初始化的核心，类的非初始化方法会调用指定初始化方法完成初始化。通常，接收参数最多的初始化方法就是指定初始化方法。

子类的指定初始化方法通常都是通过向 super 发送消息来调用超类的指定初始化方法。除此之外，还有一些通过封装来调用指定初始化方法的方法叫作**非指定初始化方法**（secondary initializer）。图 3-8

展示了指定初始化方法的概念，箭头指明了调用关系。图中每个类都只有一个指定初始化方法，实际上也可以存在多个。

子类的指定初始化方法，必须调用超类的指定初始化方法。如图 3-8 中所示，按照类层次从底向上，各个类的指定初始化方法会被连锁调用，一直到最上层的 NSObject 的指定初始化方法——`init` 为止。

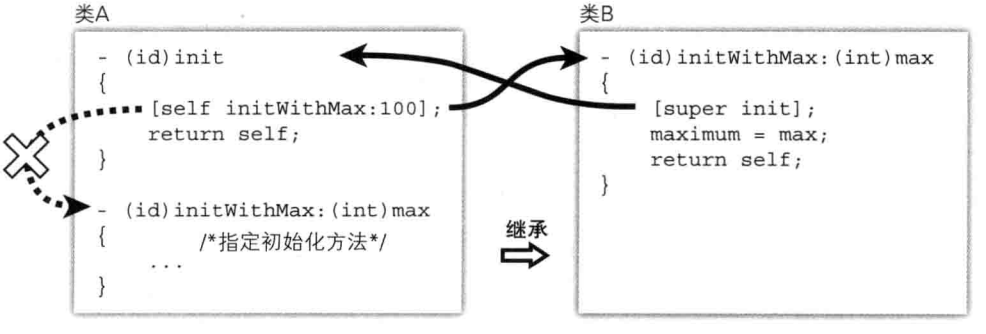
图 3-8 类的继承与指定初始化方法



如果子类中想重写父类中的指定初始化方法，就一定要调用父类的指定初始化方法，而不能调用父类的非指定初始化方法。原因是非指定初始化方法内部会调用指定初始化方法，造成递归循环调用，无法终止。

请看图 3-9 中的例子，类 A 的指定初始化方法是`initWithMax:`。`init`是类 A 的非指定初始化方法。类 B 是类 A 的子类，在 B 中重写了指定初始化方法`initWithMax:`。`initWithMax:`中调用了父类类 A 的 init。如图所示，如果类 A 的 init 中通过 self 调用了`initWithMax:`，那么，当初始化对象是类 B 的实例时，就又会调用到类 B 的`initWithMax:`，这样就变成了一个递归循环，调用永远无法结束。

图 3-9 继承时重写指定初始化方法的错误



再让我们回头看一下图 3-8，图 3-8 中类的非指定初始化方法都调用了指定初始化方法来进行初始化，同时父类的非指定初始化方法也可以被继承，但定义的时候一定要注意，否则也会出现循环调用的问题。

Objective-C 没有特殊的语法或关键字来表明哪个方法是指定初始化方法，所以通常需要通过文档或注释来标明指定初始化方法。Cocoa API 文档中的绝大多数类都标明了哪个方法是指定初始化方法。

专栏：Objective-C 与开源软件 COLUMN

20 世纪 80 年代末，布莱德·确斯（Brad Cox）发明了 Objective-C 并创建了公司 Stepstone。后来 NeXT Software 公司获得了 Objective-C 语言的授权，1996 年苹果公司宣布收购 NeXT 公司，拥有了 Objective-C 语言的所有权。Objective-C 自身的规范是公开的，编译器也是开源的。

NeXTstep 公司的 Objective-C 的编译器是基于 GUN 的 gcc 编译器扩展而来的。后来 NeXTstep 公司又把这些扩展贡献出来，所以现在的 gcc 是能够编译 Objective-C 程序的。但是 gcc 和 Cocoa 使用不同的类库，所以本书中的例子程序无法在 gcc 环境下执行。

因为 gcc 的授权问题，很多开源社区把编译器由 gcc 变为了 llvm。苹果公司也为 LLVM（http://llvm.org/）项目提供了支援，为 clang 增加了 Objective-C 的新功能（例如，ARC 自动引用技术等）。

苹果公司的 Cocoa 并没有开源，但 **OPENSTEP**（Cocoa 的基础）被 NeXT 公司开源了（OpenStep 是一个开放的操作系统的规范，OPENSTEP 是基于 OpenStep 规范的操作系统的名字，由 NeXT 公司开发）。

OpenStep 在自由软件基金会的实现叫作 **GNUStep**（http://www.gunstep.org/）。这个项目的界面采用了 OPENSTEP 风格，并提供了两个相当于 Mac OS X Foundation 和 Application Kit 的主要程序库，应用程序使用 Objective-C 来编写。GUNStep 可运行在 Unix 系和 Windows 上，也有人尝试着把 Cocoa 的应用程序移植到 GUNStep 上面。

第4章

对象的类型和动态绑定

Objective-C 的一个重要特征就是动态性，本章将对 Objective-C 的动态类型（dynamic typing）和动态绑定（dynamic bingding）进行说明。使用动态绑定能够让程序变得更加灵活，但如果使用不当也会造成程序不稳定。另外，本章也会对类型检查、访问实例变量和类对象等内容进行说明。

4.1 动态绑定

4.1.1 什么是动态绑定

实际的程序会使用各种各样的类的实例对象，所有这些对象都可以用 id 类型来表示，因为 id 是通用的对象类型，可以用来存储任何类的对象。但这样一来，程序中就会出现无法区分某个实例对象到底是哪个类的对象的情况。

让我们来看看下面这个例子。

代码清单 4-1 动态绑定的例子

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

@interface A : NSObject
- (void)whoAreYou;
@end

@implementation A
- (void)whoAreYou { printf("I'm A\n"); }
@end

@interface B : NSObject
- (void)whoAreYou;
@end

@implementation B
- (void)whoAreYou { printf("I'm B\n"); }
@end

int main(void)
{
    id obj;
    int n;

    scanf("%d", &n);
    switch (n) {
    case 0:  obj = [[A alloc] init]; break;
    case 1:  obj = [[B alloc] init]; break;
    case 2:  obj = [[NSObject alloc] init]; break;
    }
    [obj whoAreYou];
    return 0;
}
```

该程序会根据从终端读入的数字的不同，让 obj 指向不同的类的对象。类 A 和类 B 中都实现了 whoAreYou 方法，NSObject 中没实现这个方法，但编译时没给出任何警告信息，正常地生成了可执

行程序。

执行该程序，输入 0 时终端显示 I'm A，输入 1 时终端显示 I'm B，这些和事先预想的都一样。但输入 2 时，程序就会出错，有如下异常。

```
*** Terminating app due to uncaught exception 'NSInvalidArgumentException', reason:
'-[NSObject whoAreYou]: unrecognized selector sent to instance 0x103f00'
```

出错的原因在于类 NSObject 的实例对象中没有实现 whoAreYou 方法，所以出现了运行时错误。而编译时之所以没出错，是因为编译时无法确定存储在 id 中的对象的类型。

Objective-C 中的消息是在运行时才去绑定的。运行时系统首先会确定接收者的类型（动态类型识别），然后根据消息名在类的方法列表里选择相应的方法执行，如果没有找到就到父类中继续寻找，假如一直找到 NSObject 也没有找到要调用的方法，就会报告上述不能识别消息的错误。

动态绑定（dynamic binding）指的就是在程序执行时才确定对象的属性和需要响应的消息。

C 语言不支持动态绑定，在程序执行前基本上都已经绑定好了各种方法。而通过 C 语言中的函数指针也能模拟动态绑定的实现，但这需要程序员额外做很多工作，并不是一个优雅的方法。

4.1.2 多态

在面向对象的程序设计理论中，**多态**（polymor phism）是指，同一操作作用于不同的类的实例时，将产生不同的执行结果。即不同类的对象收到相同的消息时，也能得到不同的结果。

让我们通过一个画图的例子来看看软件开发中是如何使用多态的。这个软件的功能是使用鼠标来画图，可以画直线、圆、矩形等。对于画好的图形，还可以通过鼠标的拖曳来改变形状。

按照面向过程的程序设计风格来设计这个程序时，鼠标移动的处理如下所示。

```
switch (target->kind) {
case Line: // 直线
        lineDragged(direction);
        break;
case Circle: // 圆
        circleMove(direction);
        break;
case Rectangle: // 矩形
        RectangleMove(direction);
        break;
    ...
}
```

而按照面向对象的编程风格，即使用多态的情况下，则只需要下面这样一条语句就可以完成上面的操作。将各个图形定义成一个类，其中实现自己的 direction 方法，并根据 target 实际指向的对象的不同，来调用不同的 direction 操作。

```
[target move:direction];
```

单纯从这个例子来看，你可能会感到这两种写法并没有太大的不同，但如果从程序扩展性方面来考虑就大不相同了。假设我们现在需要为这个画图程序追加椭圆形、星形和心形。

使用 switch 时，每新追加一个图形都需要增加新的分歧处理，而且更重要的是，程序中所有判断图形类型的地方都需要追加新的处理语句。

而如果用多态来实现，就不需要以上处理。只要确定对象是对同一个消息做出的处理，消息发送方就不需要做任何修改。虽然需要增加类来描述新追加的图形，但程序不需要在各种地方都做修改。多态的另外一个优点就是：利用继承可以更容易地定义新的图形。

多态是面向对象编程的一个重要特征，大大增强了软件的灵活性和扩展性。

4.2 作为类型的类

4.2.1 把类作为一种类型

至今为止我们声明过的对象都是 id 类型，除了使用 id 类型外，我们还可以把定义好的类作为对象的类型。例如，用以下语句可以声明 Foo 类的一个对象 Foo *。

```
Foo *
```

类的类型既可以被用作变量的类型，也可以作为方法或函数的参数和返回值的类型使用。下面是声明两个类型变量的例子。

```
Volume *v;
MuteVolume *mute;
```

虽然这个例子中把对象用指针的形式表示了出来，但并不需要纠结指针所指向的内容。大家只要能想到整数或实数的类型同指针类型有巨大差别就可以了。让我们看看下面这个例子。

```
Volume *v1, *v2;

v1 = [[Volume alloc] initWithMin:0 max:10 step:1];
v2 = v1;
[v1 up];
printf("%d\n", [v2 value]);
```

进行赋值操作之后会发生什么呢？ v1 和 v2 都是指针类型，所以赋值操作之后，v2 指向了 v1 所指向的对象，printf() 的输出值并不是 0 而是 1。

同样，当消息的参数是对象的时候，实际上传递的是指向这个对象的指针而不是对象本身，根据消息处理中操作的不同，有可能会更改对象的值。

4.2.2 空指针 nil

Objective-C 中，nil 表示一个空的对象，这个对象的指针指向空。nil 是指向 id 类型的指针，值为 0。初始化方法失败的时候通常会返回 nil。

新生成一个实例变量的时候，alloc方法会把数值类型的实例变量初始化为 0，id 和其他类型的指针变量也会被初始化为 nil。

返回值为 id 类型的方法中，如果处理出错的话一般也会返回 nil。调用端会采用如下语句来判断方法调用是否成功。

```
if ([list entryForKey:"NeXT"] != nil) ...
```

另外，因为在 C 语言中有时会把 NULL 当作 0 来使用，因此判断语句中可以省略“!=NULL”，Objective-C 中也可以省略“!=nil”，如下所示。

```
if ([list entryForKey:"NeXT"]) ...
```

而如果给 nil 变量发送消息会怎么样呢？虽然这种写法是完全有效的，但是运行时不会有任何作用，消息也不会被发送。

让我们看看下面这个例子。

```
if ((val = [list entryForKey:"NeXT"]) != nil)
        [val increment];
```

给 nil 变量发送消息，消息也不会被发送，所以上面的程序就等价于下面的程序。

```
val = [list entryForKey:"NeXT"];
[val increment];
```

第二种写法虽然使程序变得更简洁了，但却使程序变得更难理解了。因为程序的本意是返回值为 nil 时不做任何处理。另外，不注意的话这种写法也会带来各种各样的错误，让我们来看看下面这段程序。

```
if ((val = [list entryForKey:"NeXT"]) != nil)
        [val setValue: n++];
```

虽然 val 是 nil 时消息不会被发送，但是并不意味着 [val setValue: n++] 不会被执行。如果上面这段代码中省略了 val!=nil 的判断，n++ 就仍然会被执行，这点同判断语句中的“短路”（关于“短路”的详细介绍请参考接下来的专栏）有所不同。

那么，如果向 nil 发送消息，这个消息的返回值是什么呢？一般来说，如果消息对应的方法的返回值是一个对象，那么将返回 nil；如果消息对应的方法的返回值是指针类型，则将返回 NULL；如

果消息对应方法的返回值是整数类型，则将返回 0。而如果返回值的类型是以上这几种类型以外的类型，例如结构体或者实数，那么实际返回值则同 Mac OS X 的版本以及结构体的大小等相关。也就是说，如果方法返回值不是上述提到的几种情况，那么发送给 nil 的消息的返回值将是未定义的。

专栏：关系表达式 COLUMN

C 语言中使用逻辑操作符（|| 代表逻辑与，&& 代表逻辑和）时，我们会遇到一种“短路”现象。即一旦能够准确无误地确定整个表达式的值，就不再计算表达式的剩余部分。因此，整个逻辑表达式中靠后的部分就有可能不会被运算。

例如

```
if (a == 0 && b == 1)
```

这个逻辑表达式，如果 a 不是 0，就不会对 b 的部分进行判断，而直接认为这个表达式的值为假。同样，

```
if (a == 0 || b == 1)
```

对于这个逻辑表达式，如果 a 为 0，也不会对 b 的部分进行判断，而直接认为这个表达式的值为真，程序不会查询 b 的值。

C 语言中经常利用这个性质对指针进行判断。例如，判断字符指针 p 既不是 NULL 也不指向空字符的语句可以写成

```
if (p && *p)
```

如果 p 是 NULL，由于表达式的前半部分为 false，因此不会执行对 *p 的判断。如果 p 为 NULL，则执行 *p 会出错，这个表达式就是短路模式的一个典型应用，Pascal 中就不能这么写。

利用这种性质虽然会使程序看起来更简洁一些，但不注意的话也容易出错，让我们来看看下面这个判断表达式。

```
if (a == 0 && b++ == 1)
```

根据 a 的值的不同，b++ 的操作可能会被执行，也可能不被执行。虽然这种写法有可能是事先设计好的，但很多情况下这种写法容易导致错误。

条件表达式 ?: 也同样容易让人误解。

```
a = (b > 0) ? b++ : func(b);
```

b 为正数的时候，func(b) 会被执行吗？ b 为负数的时候，b++ 的操作会被执行吗？

条件表达式中未被选择的项不会被执行。在上例中，如果 b 为正数则 func(b) 不会被执行，b 为 0 或负数的时候 b++ 的操作也不会被执行。

4.2.3 静态类型检查

虽然在 Objective-C 中 id 数据类型可以用来存储任何类型的对象，但绝大多数情况下我们还是将一个变量声明为特定类的对象，这种情况称为静态类型。使用静态类型时，编译器在编译时可以进行类型检查，如果类型不符会提示警告。

在 Objective-C 中，id 类型是一种通用的对象类型，类似于 C 语言的（void*），可以用来存储任何类的对象。在程序执行期间，这种数据类型真正的优势就会体现出来，使用 id 类型结合多态，可以使程序具备更大的灵活性和可伸缩性。但程序变灵活的同时，也需要付出代价，编译器不会对 id 类型变量的类型进行检查，这会导致程序更容易出错。

在 Objective-C 中，如果已经确定了要使用的类，我们就可以使用具体的类名来为变量静态赋予类型而不是泛泛地使用 id 类型。使用静态类型时，编译器可以在编译时检查接收者是否可以响应收到的消息，如果接收者没有与消息中的方法名相对应的方法，编译器就会发出警告。

到底是应该使用 id 类型来追求程序的灵活性，还是该使用静态类型来追求程序的严密性呢？其实并没有一个统一的准则，严谨一些的程序员喜欢使用静态类型，追求灵活性的程序员喜欢使用动态类型。

让我们通过下面这个例子来看看 Objective-C 的类型检查。

代码清单 4-2 类型检查的一个例子

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

@interface A : NSObject
- (void)whoAreYou;
@end

@implementation A
- (void)whoAreYou { printf("I'm A\n"); }
@end

@interface B : A    /*B 是 A 的子类 */
- (void)whoAreYou;
- (void)sayHello;
@end

@implementation B
- (void)whoAreYou { printf("I'm B\n"); }  /* override */
- (void)sayHello { printf("Hello\n"); }
@end

@interface C : NSObject    /*C 和 A、B 不存在继承关系 */
- (void)printName;
@end

@implementation C
- (void)printName { printf("I'm C\n"); }
@end
```

```
int main(void)
{
    A *a, *b;
    C *c;

    a = [[A alloc] init];
    b = [[B alloc] init];
    c = [[C alloc] init];
    [a whoAreYou];
    [b whoAreYou];
    [c whoAreYou];
    return 0;
}
```

类 A 和类 C 是相互独立的两个类，类 B 是类 A 的子类。main 函数中所有的变量都采用了静态定义，明确指明了变量的类型。

编译这段程序，会有如下警告。

```
warning: `C' may not respond to 'whoAreYou'
```

由于类 C 中并没有定义 whoAreYou 这个方法，因此上面提示的警告说明编译器类型检查是有效的。我们再试试把 main 函数中的类型改成动态类型 id。

```
id  c;
```

这次编译的时候没有提示任何警告，但执行的时候却出错了。我们修改一下程序中方法调用的地方，再重新编译执行一下。

```
[a whoAreYou];
[b whoAreYou];
[c printName];
```

这次程序正常执行了，执行的结果如下所示。

```
I'm A
I'm B
I'm C
```

虽然变量 a 和 b 都被声明为了类 A 类型的变量，但编译器并没有提示警告。原因在于，如果仅仅使用父类中定义的功能，则变量的类型声明为父类也是没有问题的。

下面我们来修改一下变量 b 调用的方法。

```
[b sayHello];
```

这次编译之后，会提示如下警告。

```
warning: 'A' may not respond to 'sayHello'
```

虽然有警告提示，但程序可以执行。如果不想编译器提示警告，声明的时候直接声明 b 为类 B 类型的变量即可，或者在调用方法时进行下面的转换。

```
[(B *)b sayHello];
```

通过使用强制类型转换，使编译器认为变量 b 就是类 B 的对象。虽然这种简单的程序根本不需要强制类型转换，但希望大家记住这种方法，它在一些复杂的程序中很有用。而强制转换的时候也一定要注意类型，否则就会引发运行时错误。

下面再来看最后一个例子。变量 a 是类 A 的一个实例，如果对其进行强制类型转换会怎么样呢？

```
[(B *)a whoAreYou];
```

执行之后，输出了 I'm A。也就是说，强制类型转换无效，实际执行的方法是由对象的类型决定的。

4.2.4 静态类型检查的总结

我们来总结一下编译时进行的类型检查的要点。

(1) **对于 id 类型的变量，调用任何方法都能够通过编译（当然调用不恰当的方法会出现运行时错误）**
id 数据类型可以用来存储任何类型的对象。正是由于这个原因，编译器并不知道 id 中存储的是哪个类的变量，所以无法通过->来获取类的实例变量或方法，也就没法完成类型检查。

(2) **id 类型的变量和被定义为特定类的变量之间是可以相互赋值的**
这里的赋值是一个广义的含义，包括方法或函数的参数的传递、返回值的接收等。

(3) **被定义为特定类对象的变量（静态类型），如果调用了类或父类中未定义的方法，编译器就会提示警告**

(4) **若是静态类型的变量，子类类型的实例变量可以赋值给父类类型的实例变量**
需要注意的是，如果这个变量中调用了子类特有的方法，如 (3) 所示，会提示警告信息。

(5) **若是静态类型的变量，父类类型的实例变量不可以赋值给子类类型的实例变量**
因为父类类型的变量无法对应子类中特有的方法，所以这种赋值会提示警告信息。

(6) **若要判断到底是哪个类的方法被执行了，不要看变量所声明的类型，而要看实际执行时这个变量的类型**

(7) **id 类型并不是 (NSObject*) 类型**
id 类型和其他类之间并没有继承关系。如果想更多地了解 id 类型，请参考头文件 /usr/include/objc/objc.h。

Objective-C 的静态类型检查是在编译期完成的。向一个静态类型的对象发送消息时，编译器可以确保接收者可以响应该消息，否则会发出警告；当把一个静态类型的对象赋值给一个静态类型的变量时，编译器可以确保这种赋值是兼容的，否则会发出警告。运行时实际被执行的方法同变量定义时的类型无关，只和运行时这个变量的实际对象有关。长期使用 C 或 Pascal 等静态语言的人，可能不容易理解这种思想。

面向对象的语言中 Smalltalk 是一种弱类型语言，程序中不做变量类型说明，系统也不做类型检查。C++ 和 Java 是强类型语言，编译时会进行类型检查，以保证类型兼容。

C++ 和 Java 的多态是基于类层次结构的，可以通过子类重写父类中的方法来实现多态。这种类型的语言无法实现代码清单 4-1 中那种没有继承关系的多态。

Objective-C 除了在运行时检查接收者是否可以响应所收到的消息之外，还在编译时进行基于类层次的静态类型检查。而就结果而言，也能够实现基于类层次的多态。

4.3 编程中的类型定义

4.3.1 签名不一致时的情况

消息选择器（message selector）中并不包含参数和返回值的类型的信息，消息选择器和这些类型信息结合起来构成**签名**（signature），签名被用于在运行时标记一个方法。接口文件中方法的定义也叫作签名。

```
- (id)cellAtRow:(int)row column:(int)col;
```

Cocoa 提供了类 NSMethodSignature，以面向对象的方式来记录方法的参数个数、参数类型和返回值类型等信息。这个类的实例也叫作方法签名，详情请见 15.5 节。

方法通过消息选择器被调用，选择器并不包含参数和返回值的信息。那么，如果选择器相同而参数和返回值不一样的话，会怎么样呢？

让我们来看看下面这个例子，接口定义如下。

```
@interface X : NSObject
- (int)value;
@end

@interface Y : NSObject
- (float)value;
@end

@interface Z : X
- (const char *)value;
@end
```

这时，下面的代码会被编译执行，而返回值则是无法确定的，因为 X 或 Z 的 value 方法都有可能被执行。

```
id obj;
...
a = [obj value];
```

再看看下面的例子。

```
X *foo;
Y *bar;
...
a = [foo value];
b = [bar value];
```

[bar value] 的返回值类型是确定的，但 [foo value] 的返回值类型无法确定。因为 foo 既有可能被赋值为类 X 的对象，也有可能被赋值为类 Z 的对象。类 X 的 value 方法返回 int，类 Z 的 value 方法返回 const char *。

如果消息接收者和参数的类型是运行时确定的，那么消息签名不唯一的话编译就会出错。

也就是说，Objective-C 中**选择器相同的消息，参数和返回值的类型也应该是相同的**。尤其是上例中类 X 和类 Z 这种带有继承关系的类要特别注意这种情况。

不仅仅是我们在编程时定义的类，如果 Cocoa 提供的类之间也要查看选择器相同的方法的参数和返回值的话，那就太麻烦了。实际上，Foundation 和 Application 框架内存在一些选择器相同但签名不同的方法，所以对于这点，大家也不要过于追求完美了。

编译的时候，如果编译器发现了签名不一致的情况，就会提示警告，并显示出有问题的地方。大家可采用更改方法名，或更改方法类型为静态类型来屏蔽警告。

```
signature.m:41:9: warning: multiple methods named 'value' found
```

专栏：重载 COLUMN

对于绝大多数编程语言（C 或 Pascal 等）来说，1+2 和 1.0+2.2 这种写法都是可以的。但是你是否考虑过根据加法运算符处理的变量类型的不同（整数的加法和浮点数的加法），来分别实现不同的加法操作呢？

重载（overloading）指的就是一个函数、运算符或方法定义有多种功能，并根据情况来选择合适的功能。

以 C++ 为例来说明一下函数的重载。下面是 maxValue() 的三个实现，编译器会根据参数的个数和类型的不同，来决定到底调用哪个函数。

```
double maxValue(double a, double b);
double maxValue(double a, double b, double c);
double maxValue(int length, double arr[]);
```

Objective-C 是动态语言，参数的类型是在运行时确定的，所以不支持这种根据参数类型的不同来调用不同函数的重载。Objective-C 可以通过动态绑定让同一个消息选择器执行不同的功能来实现重载。

4.3.2 类的前置声明

当我们定义一个类的时候，有时会将类实例变量、类方法的参数和返回值的类型指定为另外一个类。这种情况该如何定义呢?

一种方法是，在新定义的类的接口文件中引入原有类的头文件，如下例所示。

```
#import <Foundation/NSObject.h>
#import "Volume.h"     // 引入类 Volume 的接口头文件

@interface AudioPlayer : NSObject {
    Volume *theVolume;
    ...
}
- (Volume *)volume;
...
```

以上这种方法肯定是可行的，但也有一些缺点。首先是头文件中除了类名之外，还有各种各样的其他信息的定义。另外，引入的头文件中还有可能还引入了其他类的头文件，如此循环会大大加大编译时的负担。

如果仅仅是在类型定义的时候使用一下类名，则还有一种新的解决方法。

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

@class Volume;     // 声明要使用类 Volume

@interface AudioPlayer : NSObject
...
```

通过编译指令 **@class** 告知编译器 Volume 是一个类名。这种写法被叫作类的**前置声明**（forward declaration）。

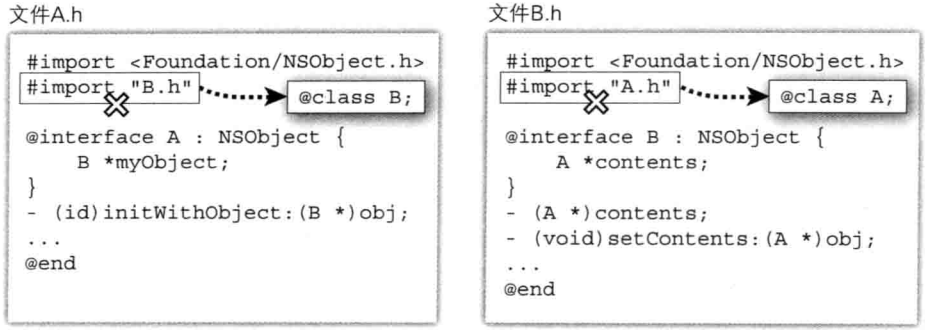
class 指令的后面可以一次接多个类，不同的类之间用“,”来分割，最后用“;”来标识前置声明的结束。前置声明可以声明多次，如下所示。

```
@class NSString, NSArray, NSMutableArray;
@class Volume, MuteVolume;
```

通过使用 @class 可以提升程序整体的编译速度。但要注意的是，如果新定义的类中要使用原有类的具体成员或方法，就一定要引入原有类的头文件。

@class 的另外一个好处是，当多个接口出现类的嵌套定义时，如果只是相互包含对方的头文件无法解决，则只能通过类的前置声明来解决（图 4-1）。

图 4-1 使用 @class 的类的前置声明



4.3.3 强制类型转换的使用示例

上面已经说明了父类和子类在类型方面的关系，但有些情况下必须使用强制类型转换。一个典型的例子就是，父类类型的指针实际指向了子类类型的变量。

让我们用类 AudioPlayer 的例子来说明一下。类 AudioPlayer 中有一个 Volume 类型的实例变量。

```
@interface AudioPlayer : NSObject {
    Volume *theVolume;
    ...
}
...
@end
```

假设这个类的派生类 CDPlayer 想拥有静音的功能，使 Volume 类型的实例变量 theVolume 指向 Volume 类的子类 MuteVolueme 的对象。

```
@implementation CDPlayer  // //AudioPlayer 的派生类
- (id)init {
    if ((self = [super init]) != nil) {
        theVolume = [[MuteVolume alloc] init];
        ...
}
```

AudioPlayer 类中的绝大多数地方使用 theVolume 都是没有问题的。但在 mute 这个函数里面使用 theVolume 的时候则需要强制类型转换。

```
- (void)mute:(id)sender {
    [(MuteVolume *)theVolume mute];
}
```

还有一种不需要强制类型转换的方法就是利用 id 类型的变量临时中转一下，但这样一来代码的

可读性就变差了。

```
- (void)mute:(id)sender {
    id m = theVolume;
    [m mute];
}
```

虽然强制类型转换的功能很强大，但会让编译器的类型检查变得没有意义，所以要尽量少用。不得不使用的情况下，要重新思考设计是否合理。

4.4 实例变量的数据封装

4.4.1 实例变量的访问权限

目前为止所举的例子都是对象自身来访问、修改自己的实例变量，那么对象能获取、修改另外一个对象的实例变量吗?

Objective-C 原则上不允许从对象外直接访问对象的实例变量。但类 A 的方法中可以直接访问类 A 中包含 self 以外的其他实例的实例变量。同类型检查一样，能不能访问对象的实例变量也需要检查，这个检查在编译时完成。因此，只能访问使用静态类型定义的实例对象的内部变量。

下面是访问 obj 对象中实例变量 myvar 的语句写法。

```
obj->myvar
```

可以看出，该语法和通过指针访问结构体中内部成员的写法是一样的。

下面这个例子定义了一个代表三原色的类，类中有一个计算两种颜色的混合色的方法。另外，这里面还展示了如何使用局部函数。

代码清单 4-3 三原色类的接口部分 (RGB.h)

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

@interface RGB : NSObject
{
    unsigned char  red, green, blue;
}

- (id)initWithRed:(int)r green:(int)g blue:(int)b;
- (id)blendColor:(RGB *)color;
- (void)print;

@end
```

代码清单 4-4 三原色类的实现部分 (RGB.m)

```
#import "RGB.h"
#import <stdio.h>

static unsigned char roundUChar(int v)
{
    if (v < 0) return 0;
    if (v > 255) return 255;
    return (unsigned char)v;
}

@implementation RGB

- (id)initWithRed:(int)r green:(int)g blue:(int)b
{
    if ((self = [super init]) != nil) {
        red = roundUChar(r);
        green = roundUChar(g);
        blue = roundUChar(b);
    }
    return self;
}

- (id)blendColor:(RGB *)color
{
    red = ((unsigned int)red + color->red) / 2;
    green = ((unsigned int)green + color->green) / 2;
    blue = ((unsigned int)blue + color->blue) / 2;
    return self;
}

- (void)print {
    printf("(%d, %d, %d)\n", red, green, blue);
}

@end
```

代码清单 4-5 测试三原色类用的 main 程序

```
#import "RGB.h"

int main(void)
{
    RGB *u, *w;

    u = [[RGB alloc] initWithRed:255 green:127 blue:127];
    w = [[RGB alloc] initWithRed:0 green:127 blue:64];
    [u print];
    [w print];
    [[u blendColor: w] print];
    return 0;
}
```

方法blendColor:中直接通过->访问了参数 color 内部的实例变量。之所以可以访问，是因为参数 color 的类型和 blendColor 所在类的类型一致。如果 color 是别的类（就算是父类）的对象，则不可以用这种方法直接访问 color 内部的实例变量。另外，变量 color 声明类型时必须使用静态类型 RGB。

4.4.2 访问器

Objective-C 不允许从外部直接访问和修改实例对象的属性。而仅仅可以访问同一个类的其他实例对象的变量。我们通常会定义专门的方法来访问或修改实例变量。

例如，类中有一个 float 类型、变量名叫 weight 的属性，从类外部访问这个属性的**方法应和属性同名**，如下所示。

```
- (float) weight
```

定义修改该属性的方法时，可以**用 set 作为前缀，之后接要更改的属性的名称，属性名的第一个字母要求大写**。

```
- (void) setWeight: (float) value
```

这种用于读取、修改实例对象属性的方法称为**访问器**或访问方法。读取属性值的方法称为 getter 方法，修改属性值的方法称为 setter 方法。这两种方法也可以简称为 getter 和 setter。setter 方法有一个 void 类型的参数。

有些面向对象的语言会用 get 作为 getter 方法的前缀，而 Cocoa 的 Objective-C 则直接使用属性名作为方法名（详情请参考附录 C）。

下面的例子是三原色类中 red 属性对应的 getter 和 setter 方法。

```
- (unsigned char)red { return red; }

- (void)setRed:(unsigned char)newvalue {
    red = newvalue;
}
```

为什么不允许直接访问成员属性，而要通过 getter 和 setter 方法来完成操作呢？

一切都是为了封装。类中包含哪些实例变量以及怎么使用这些实例变量都和类的实现紧密相关。而如果允许直接访问类的实例变量，那么当类的实现发生了变化，实例变量被删除或者作用发生了变化时，所有调用这个类的外部模块都需要更改。

而如果使用了 getter/setter 的形式，那么当类的实现发生了变化时则只需要更改 getter/setter 接口，外部调用部分不需要做任何改变。

还是拿三原色的类作为例子来说明。可以看出，目前使用了 RGB 来表示颜色，而如果想用 CMYK[①] 来表示颜色的话该怎么办呢？这种情况下，通过采用 getter/setter，只需要修改 getter/setter 方

① 是一种色彩模式，和 RGB 类似，CMY 是 3 种印刷油墨名称的首字母：青色 Cyan、品红色 Magenta、黄色 Yellow。而 K 取的是 black 最后一个字母，之所以不取首字母，是为了避免与蓝色（Blue）混淆。

法中的实现就可以了。外部调用模块不需要做任何修改。

同样的道理，父类和子类之间也要尽可能地多利用 getter/setter 方法。

虽然子类的方法可以直接访问父类的实例变量，但我们要养成一个好的习惯，即尽量使用 getter/setter 方法来访问父类中的实例变量，这样可以使程序做到尽可能地低耦合。

举一个实际的例子，在 NeXTstep 时代，API 的参考文档中不光提供类方法，还包括类中主要的实例变量。而到了 OPENSTEP 时代，类库（框架）的参考文档中就不再包括类中实例变量的信息了。于是，这些类的派生类中就必然只能通过 getter/setter 方法来访问父类中的实例变量。养成这样的习惯后，就算框架中类的实现方法有大幅改变，子类也不容易受影响。

4.4.3 实例变量的可见性

如前所述，虽然一般情况下不允许从外部直接访问对象的实例变量，但如果一定要访问的话，也可以通过其他办法来访问。反之，我们知道默认条件下子类是可以访问父类的实例变量的，而有没有办法不允许子类访问父类的实例变量呢？

能否从外部访问实例变量决定了访问的**可见性**（visibility）。Objective-C 中有四种可见性修饰符。

- **@private**

  只能在声明它的类内访问，子类中不可以访问。可以在方法中通过->来访问同一个类的实例对象。

- **@protected**

  能够被声明它的类和任何子类访问。类方法中可以通过->来访问本类实例对象的实例变量。没有显式指定可见性的实例变量都是此属性。

- **@public**

  作用范围最大，本类和其他类都可以直接访问。

- **@package**

  类所在的框架（参见 8.3 节）内，可以像 @public 一样访问。而框架外则同 @private 一样，不允许访问。

表 4-1 实例变量的访问可见性

	@private	@protected	@public
同一个类内	○	○	○
同一个类内使用 -> 访问	○	○	○
子类	×	○	○
子类中使用 -> 访问	×	×	○
任意地点都可访问	×	×	○

类接口定义中声明实例变量的同时设置变量的访问权限。可分别为每个实例变量指定访问权限，也可以一次性为一组变量指定访问权限。为一组变量指定访问权限时，前一个声明到下个声明（或声

明结束）之间所定义的全部变量都属于该声明的权限。访问权限声明的顺序和次数都没有限制。

例如：

```
@interface TableOfColors : HashTable
{
    id      delegate;                   // protected
@public
    BOOL    empty;                      // public
@private
    id      cache;                      // private
    int     cache_entries;              // private
@protected
    int     entries;                    // protected
}
...
@end
```

代码实现部分（@implementation 和 @end 之间）中定义的函数的可见性是在声明的时候决定的。图 2-3 中的函数 funcB 可以使用 -> 符号来访问 @private 属性的实例变量。

4.4.4 在实现部分中定义实例变量

Xcode 4.2 之后的编译器 clang，**允许在实现部分中定义类的实例变量**。

代码清单 4-3、4-4 的三原色的例子，可以采用如下的方法定义。

```
// 接口部分
@interface RGB : NSObject
- (id)initWithRed:(int)r green:(int)g blue:(int)b;
- (id)blendColor:(RGB *)color;
- (void)print;
@end

// 实现部分
@implementation RGB
{
    unsigned char  red, green, blue; // 实例变量
}

- (id)initWithRed:(int)r green:(int)g blue:(int)b
    ...
@end
```

使用这种方法定义类后，因为实例变量定义在了实现文件中，因此，即便外部模块拿到接口文件，也不知道类中定义了哪些实例变量。如果想追求更高层次的封装，可以采用这种方式。这样一来，就算重新定义了实例变量，也不需要更改接口头文件。

有一点要注意的是，采用这种方法定义后，子类就无法访问父类的实例变量了。这种情况下，子类如果想访问父类的实例变量，就需要利用访问方法或第 7 章中介绍的属性声明的方法（@property）。

实例变量也可以在接口和类文件中分别定义。下面的例子中，类 RGB 有 4 个实例变量：red、green、blue 和 gamma，其中只有 gamma 是对外不可见的。

```
// 接口部分
@interface RGB : NSObject {
    unsigned char  red, green, blue;
    // 实例变量（默认的可视性是 @protected）
}
    ...
@end

// 实现部分
@implementation RGB {
    double gamma; // 追加了外部不可见的实例变量
}
```

让一个变量对外不可见有两种方法，一种是把变量的可见性属性设为 @private，另一种就是把变量定义在实现文件中。相比之下第二种方法封装性更好一点，同时也可以清楚地表明这些变量对外是不可见的。

实现文件中定义的实例变量的可见性默认是 @private。也可以用 @public、@protected 来重设可见性。如果有一个新类定义在了同一个文件中，新类的方法也可以访问可见性为 @public 的实例变量。

4.5 类对象

4.5.1 什么是类对象

面向对象的语言中对类有两种认识，一种是认为类只作为类型的定义，程序运行时不作为实体存在；另外一种是认为类本身也作为一个对象存在。我们把后一种定义中类的对象叫作**类对象**（class object）。这种情况下类定义就分为了两部分，一部分定义所生成的实例的类型，另外一部分则定义类自身的行为。Objective-C 和 Smaltalk 都把类作为对象看待，而在 C++ 中，类只被作为类型定义使用。

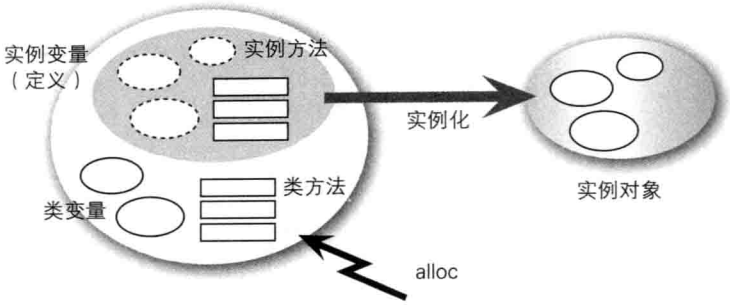
类对象有自己的方法和变量，分别被称为**类方法**和**类变量**。至今为止我们一直把类的实例变量和方法称为实例变量和**实例方法**，这样可以和类变量和类方法予以区分。

Objective-C 中只有类方法的概念，没有类变量的概念，关于这一点后面会详细说明。另外，Objective-C 中类对象也称为 factory，类方法称为 factory method。大家在看书或者文档的时候有可能会碰到这种叫法。

类方法的一个典型操作就是创建类的实例对象。类对象收到`alloc`这种消息之后会生成类的实例，如图 4-2 所示。至今为止我们只介绍了通过给类名发送`alloc`消息来生成类实例，其实这个地

方真正调用的是类方法。

图 4-2 类对象的概念



通过向类发送消息可以生成实例对象，那么类对象自身是什么时候生成的呢？类对象是程序执行时自动生成的，每个类只有一个类对象[①]，不需要手动生成。

而类方法alloc定义在哪个类里面呢？它定义在根类 NSObject 中，所有的类都继承了 NSObject，所以不用担心使用alloc生成类对象的问题。

也可以通过为类对象定义新的方法来完成实例变量的生成和初始化。每个类的所有实例对象都可以使用类方法。类方法可以访问类对象管理的变量。

4.5.2 类对象的类型

假设把类对象也作为对象来处理，那么可以为类对象进行赋值等操作吗？答案是肯定的。

id 类型可以表示任何对象，类对象也可以用 id 类型来表示。Objective-C 中还专门定义了一个 Class 类型用来表示类对象，所有的类对象都是 Class 类型。Class 和 id 一样都是指针类型，只是一个地址，并不需要了解实际指向的内容。Nil 被用来表示空指针（是 Class，而不是对象），实际的值是 0。

NSObject 中定义了类方法class，所有的类都可以使用这个方法来获取类对象。下面是类方法class的一种用法。

```
Class theClass = flag ? [Volume class] : [MuteVolume class];
id    v = [[theClass alloc] initWidthMin:0 max:100 step:5];
```

将类名定义为消息接收者是类对象特有的功能，除此之外类名只能在类型定义时使用。例如，上面的例子如果采用以下写法，就会出错。

```
Class theClass = flag ? Volume : MuteVolume;
```

我们再举一个类方法的例子。NSObject 中有一个实例方法isMemberOfClass:。这个方法的参

① 或者在执行文件以外的别的文件中放置类的代码，并在执行的时候动态加载（具体信息请参考第 16 章）。

数是一个类对象，返回值是一个 BOOL 值。该方法可被用来确定调用方法的对象是否是该类的成员，如下所示。

```
BOOL isMember = [someobj isMemberOfClass:[Volume class]];
```

下面这种写法是错误的。

```
BOOL isMember = [someobj isMemberOfClass: Volume];
```

除了 class 类方法之外，类 NSobject 还有一个`class`实例方法。所有的实例对象都可以使用`class`实例方法，这个方法返回的是对象所属类的类对象。

4.5.3 类方法的定义

实例方法在接口声明和实现文件中都以“-”开头，类方法则与此相反，以“+”开头。`alloc`方法的定义如下。

```
+ (id)alloc;
```

类方法的方法名（消息选择器）、参数等的写法都和实例方法完全一样。类方法的方法名可以和实例方法甚至和实例变量的名字一样。

继承的情况下，子类可以访问父类的类方法。

类方法的语法比较简单，但实际使用时要注意以下几点。

首先，类方法中不能访问类中定义的实例变量和实例方法。类对象只有一个，类的实例对象可以有任意个。所以，如果类对象可以访问实例变量，就会不清楚访问的到底是那个实例对象的变量。大家稍微思考一下就会觉得这种规则确实是合理的。同理，类方法中也不能访问实例方法。

其次，类方法在执行时用 self 代表了类对象自身，因此可以通过给 self 发送消息的方式来调用类中的其他类方法。同实例方法一样，也要注意 self 实际指向的类（详情请参考 3.4 节）。

调用父类的类方法时，可以使用 super。

4.5.4 类变量

Objective-C 不支持类变量（也就是静态成员变量）。既然对象同时拥有方法和变量，那么为什么可以定义类方法，而不支持类变量呢？

Objective-C 通过在实现文件中定义**静态变量**（static variables）的方法来代替类变量。

C 语言的函数和在函数外部定义的变量（也称为全局变量）的作用域是整个源程序，程序中任何地方都可以使用这些函数和变量。如果给函数或变量加上 static 修饰符，函数或变量的作用域就会变为只在其所在的文件内有效，在同一程序的其他文件中则不能使用它。通常情况下，我们会利用

static 的这一特性来定义一些不想对外公开的函数和变量。

Objective-C 在实现文件中定义了静态变量后，该变量的作用域就变为只在该文件内有效。也就是说，只有类的类方法和实例方法可以访问这个变量。

但这种方法有一个问题，就是在继承的情况下子类如何访问 static 变量。

在继承的情况下，因为 static 变量的作用域仅限于定义它的文件内，所以子类无法访问父类中定义的 static 变量。

可以通过定义类方法来解决这个问题，即为静态变量定义访问（getter）和设置（setter）类方法。这样一来，就算在继承的情况下，因为子类会继承父类中定义的类方法，所以通过这些类方法也可以访问父类中定义的 static 变量。通过使用这种方法，当利用 Cocoa API 进行编程时，就可以自由地访问和设置实例变量，而不会感到任何不便。

Objective-C 中类变量原则上只在类的内部实现中使用，在进行设计时要充分考虑到这一点。

4.5.5 类对象的初始化

Objective-C 中实例对象的生成一般分为两步，第一步是通过`alloc`为对象分配内存，第二步是对内存进行初始化，也就是对对象的各个成员赋予初值。可以通过给实例对象发送`init`消息来完成第二步的初始化。同实例对象一样，类对象中也有要初始化的变量。那么如何对类对象进行初始化呢？由于类对象在程序执行的时候就已经生成了，因此不能通过发送消息的方法来进行初始化。

Objective-C 的根类 NSObject 中存在一个`initialize`类方法，可以通过使用这个方法来为各类对象进行初始化。在每个类接收到消息之前，为这个类调用一次`initialize`，调用之前要先调用父类的`initialize`方法。每个类的`initialize`方法只被调用一次。

因为在初始化的过程中会自动调用父类的`initialize`方法，所以子类的`initialize`方法中不用显式调用父类的`initialize`方法。

让我们看看下面这个例子。

代码清单 4-6 类对象初始化测试

```
#include <Foundation/NSObject.h>
#include <stdio.h>

@interface A : NSObject
+ (void)initialize;
@end

@implementation A
+ (void)initialize {  printf("I'm A\n"); }
@end

@interface B : A
+ (void)initialize;
+ (void)setMessage:(const char *)msg;
- (void)sayHello;
@end
```

```
static const char *myMessage = "Hello";

@implementation B
+ (void)initialize { printf("I'm B\n"); }
+ (void)setMessage:(const char *)msg { myMessage = msg; }
- (void)sayHello { printf("%s\n", myMessage); }
@end

int main(void)
{
    id obj = [[B alloc] init];
    [obj sayHello];
    [B setMessage: "Have a good day!"];
    [obj sayHello];
    return 0;
}
```

程序的执行结果如下所示。

```
I'm A
I'm B
Hello
Have a good day!
```

通过程序的输出结果可以看出，父类 A 和子类 B 的initialize方法被执行后，sayHello方法被执行了。要注意子类 B 中不需要显式调用父类 A 的initialize方法。

该程序把 myMessage 当作了类变量来使用，通过类方法setMessage:可设置 myMessage 的值。而一旦设置之后所有类 B 的实例对象的 myMessage 都会改变。

如果一个类中没有实现initialize方法，其父类的initialize方法就会被调用两次，面向自己一次，面向子类一次。所以实现initialize方法时要能确保该方法可以被重复调用。例如，上例中 initialize 的实现如下。

```
+ (void)initialize
{
    static BOOL nomore = NO;

    if ( nomore )
        return;
    printf("I'm B\n");
    nomore = YES;
}
```

上面的例子中通过调用initialize方法进行了初始化操作，但并不是每个类都需要initialize方法。如果没有任何需要被初始化的变量，则只需要对类对象发送一个self消息即可。self方法定义在 NSObject 中，只返回消息接收者自身，没有多余的操作。

4.5.6 初始化方法的返回值

让我们回顾一下上一章中的类 Volume 的例子。在那个例子中，初始化方法的返回值被定义为了 id 类型。然而，既然返回的是 Volume 类的实例变量，为什么不能把返回值定义为（Volume *）类型呢？

之所以将返回值定义为 id 类型是因为考虑到初始化方法的返回值不是具体的类类型，而是可变的，取决于上下文。例如，假设类 Volume 提供了一个返回值为（Volume *）的初始化方法。这时，向类 MuteVolume（Volume 的派生类）的对象发送初始化消息后，返回的就是 Volume 类型的对象，而不是 MuteVolume 类型。所以一般情况下，**初始化方法的返回值都应该设为 id 类型**。

可见，在继承存在的情况下，我们要尽可能地避免使用静态类型把代码写“死”。让我们看一个如何给代码增加弹性的例子。下面是调用类方法初始化的一行语句。

```
[[Volume alloc] initWithMin:a max:b step:s];
```

比起该写法，一种更好的写法是

```
[[[self class] alloc] initWithMin:a max:b step:s];
```

新写法中使用 [self class] 替代了具体的类名 Volume。class 是类方法，返回 self 所属的类对象。这样修改之后，子类也可以原封不动地使用这行代码。

第5章

基于引用计数的内存管理

本章和下一章中将介绍 Objective-C 如何对实例对象进行内存管理。本章中主要对基于引用计数的内存管理方式和 Xcode 4.2 之后可以使用的自动引用计数（ARC，Automatic Reference Counting）的管理方式进行说明。

ARC 是 Mac OS X 10.7 和 iOS 5 引入的新特性，也是苹果公司推荐使用的内存管理方法。启用 ARC 后，编译器会在适当的地方自动加入 retain、release、autorelease 等语句，来简化 Objective-C 编程在内存管理方面的工作量。

5.1 动态内存管理

5.1.1 内存管理的必要性

C 语言中需要手动利用 malloc() 和 free() 对内存进行管理。当程序运行结束时，操作系统会释放为其分配的内存。如果是很小、运行时间很短的程序，就算是内存没释放也不会有问题，程序结束时操作系统会自动释放内存。而对于长时间运行的程序，则需要程序员释放不再使用的内存，否则程序就会崩溃。

如果程序没能妥善管理内存，运行过程中就不但不能释放不再使用的的内存，而且还会不停地分配内存，这样所占用的内存就会越来越多，程序速度也会越来越慢，最后甚至会因内存耗尽而崩溃。

就好像漏水一样，程序未能释放已经不再使用的内存叫作**内存泄漏**（memory leak）。C 语言中要特别注意内存的动态分配和释放，以防内存泄漏。有效地管理内存，会提高程序的执行效率。

如果访问了已经被释放的内存，则会造成数据错误，严重时甚至会导致程序异常终止。在指针所指向的对象已被释放或收回的情况下，该指针就称为**悬垂指针**（dangling pointer）或野指针。继续使用这种指针会造成程序崩溃。

Objective-C 会通过向类对象发送 `alloc` 消息来生成实例对象，alloc 的作用就是分配内存。alloc 方法的返回值是 id 类型，我们前面介绍过 id 其实就是指针类型，而其所指向的就是为实例对象分配的内存。生成的实例对象用完之后如果不被释放的话，就会发生内存泄漏。另一方面，如果给已经释放了的实例对象发送消息，运气好的话会得到警告提示，告诉你向已经释放的对象发送了消息。运气不好的话则会造成程序错误甚至异常终止，所以 Objective-C 的程序中一定要注意内存管理。

在面向对象的语言中，对象是程序的核心。而对象也有生命周期，既有从头到尾一直存在的对象，也有生命期短暂的临时对象。对象之间也有可能互相引用，构成结构复杂的数据结构。同面向过程的语言相比，面向对象的语言的内存管理要更复杂一些。

5.1.2 引用计数、自动引用计数和自动垃圾回收

Cocoa 环境的 Objective-C 提供了一种动态的内存管理方式，称为**引用计数**（reference counter）。这种方式会跟踪每个对象被引用的次数，当对象的引用次数为 0 时，系统就会释放这个对象所占用的内存。本书中把这种内存管理方式称为**基于引用计数的内存管理**[①]。

比引用计数内存管理更高级一点的就是**自动引用计数**（Automatic Reference Counting，简写为

① 苹果公司的文档中把基于引用计数的内存管理称为 managed memory。这种说法容易引起误解，让人误认为内存已经被自动管理好了，也就是使用了垃圾回收。和 ARC 相对，手动引用计数内存管理称为 MRC（Manual Reference Counting），或简写为 MRR（Manual Retain/Release）。

ARC[①]）的内存管理。自动引用计数使开发者不需要考虑何时该使用 retain、release、autorelease 来管理内存，它提供了自动评估对象生存期的功能，在编译期间会自动加入合适的管理内存的方法。为了同自动引用计数进行区分，本书中将引用计数内存管理称为手动引用计数内存管理。

除了 ARC 外，Objective-C 2.0 还引入了另外一种自动内存管理机制——**垃圾回收**（详见第 6 章）。使用垃圾回收时，就不再需要通过引用计数来管理创建的对象，系统会自动识别哪些对象仍在使用，哪些对象可以回收。

程序员可以从手动引用计数管理、ARC 和垃圾回收中选择任意一种内存管理方式来开发程序。为 Mac OS X 10.7 Lion 和 iOS 5 之后的系统开发程序时，强烈建议使用 ARC。三种内存管理方式的对比如表 5-1 所示。

表 5-1 几种内存管理方式的比较

内存管理方式	难易度	Mac	iOS	备注
手动引用计数	较难	支持	支持	默认方式
自动引用计数	容易	支持	支持	强烈建议新开发项目采用
垃圾回收	容易	支持	不支持	

5.2 手动引用计数内存管理

本节将说明如何基于引用计数来管理内存。手动引用计数是自动引用计数的基础，就算程序使用自动引用计数的内存管理，也需要了解手动引用计数的原理。

5.2.1 引用计数

Cocoa 环境的 Objective-C 使用了一种叫作引用计数的技术来管理对象所占用的内存。每个对象都有一个与之相关的整数，称作它的引用计数。当某段代码需要使用一个对象时，就将该对象的引用计数器值加 1。当这段代码不再使用这个对象时，则将对象的引用计数器值减 1。换言之，引用计数就是指程序中到底有多少个地方需要访问这个对象。

使用`alloc`和初始化方法创建一个对象时，该对象的引用计数的初始值为 1。假设有一个类 A 在进行某些处理的过程中需要使用到实例 B，为了防止实例 B 被别的对象随意释放，类 A 会事先给实例 B 发送一个`retain`消息。这样，每执行一次`retain`，实例 B 的引用计数就会加 1。

反之，不再需要某个对象时，可以通过发送`release`消息，使对象的引用计数减 1。

实际上，释放内存的并不是`release`，而是`dealloc`方法。同`alloc`不同，`dealloc`不是类方

① ARC 的发音为 ɑ:k。

法而是一个实例方法。每收到一个release消息，对象的引用计数值就会减一。当对象的引用计数值达到0的时候，系统就知道这个对象不再需要了。这时，Objective-C会自动向对象发送一条dealloc消息来释放内存。通常不允许在程序内直接调用dealloc方法。

retain、release和dealloc的定义如下所示。retain的返回值是接收消息的对象。关键字oneway的详细含义请参考第19章。

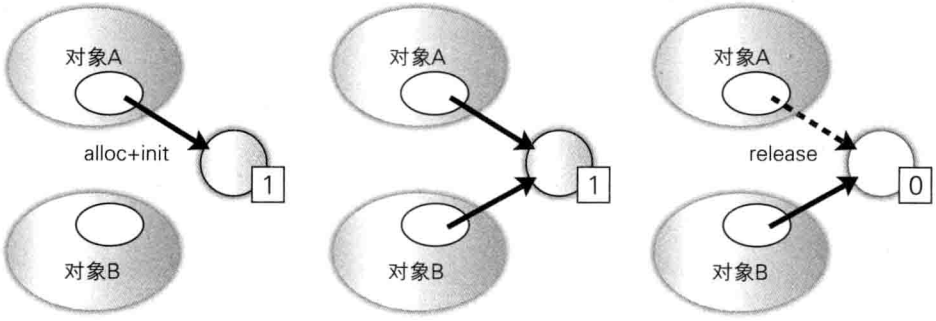
```
- (id)retain;
- (oneway void)release;
- (void)dealloc;
```

retain是“保持”的意思，给一个对象发送retain消息，就意味着“保持”这个对象。生成对象或通过给对象发送retain消息来保持对象这种状态，都可以说是拥有这个对象的**所有权**（ownership）。拥有实例所有权的对象叫作**所有者**（owner）。

这里需要注意的是，所有权是一个虚拟的概念。既无法通过语法标记，也无法通过对象中的某个属性表示出来。程序在运行时没法确认某个对象的所有者是谁。所有权仅仅是人们分析阅读程序时，为了说明对象之间的关系而加上的一个属性。

通过引用计数能够表现出一个对象有几个所有者。只要某个对象的引用计数大于0就表示这个对象有所有者。引用计数变为0的时候，说明这个对象没有所有者，会被释放。

图5-1 没使用引用计数的例子



（1）对象A中生成一个实例　（2）把对象A中生成的实例赋值给对象B　（3）对象A不再使用这个实例

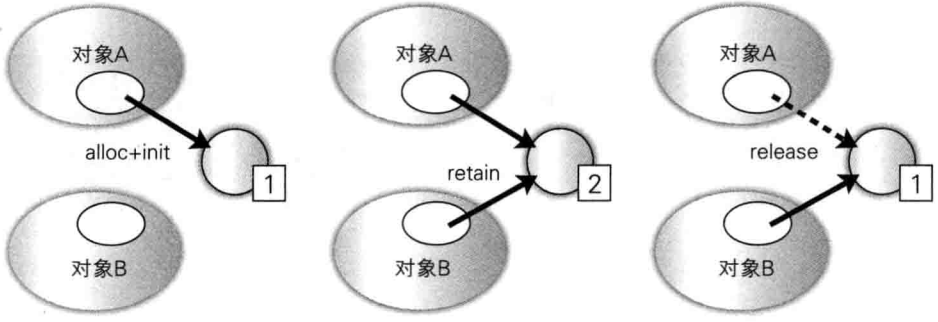
让我们通过图5-1的例子来说明一下基于引用计数的内存管理。首先，假设图（1）中对象A的方法把一个新生成的一个实例对象赋值给了A的实例变量。这个时候新生成的对象的引用计数为1，它的所有者是对象A。然后，图（2）中把这个对象的指针赋值给了对象B的某个实例变量。因为对象B并没有发送retain消息，所以并不是这个对象的所有者，这个对象的引用计数还是1。最后，图（3）中对象A不再使用这个对象时，给它发送了release消息。于是，虽然对象B还在使用这个对象，但该对象也被释放了。而此时如果对象B给已经释放的对象再发送消息，就会发生运行时错误，程序会异常终止。

为了防止这样的情况发生，一定要给动态生成的对象发送retain消息来增加它的引用计数。只

要对象的引用次数大于零，系统就不会释放它。

图 5-2 中，对象 B 把新生成的对象赋值给另外一个变量的同时给它发送retain消息，实例对象的引用计数变为 2。retain操作之后，即使图（3）中对象 A 给新生成的对象发送release消息，对象也不会被释放，对象 B 还可以继续使用这个对象。

▶ 图 5-2 使用引用计数的例子



（1）对象A中生成一个实例　（2）把对象A中生成的实例赋值给对象B　（3）对象A不再使用这个实例

5.2.2 测试引用计数的例子

让我们通过一个例子来看看引用计数到底是如何工作的。

retain和release是类 NSObject 的实例方法，方法retainCount可以获得对象引用计数的当前值。retainCount方法并没有太大的实用价值，一般在调试程序的时候使用。

代码清单 5-1 的程序显示了对象生成后收到retain、release 消息时引用计数的变化。retainCount 的返回值是 NSUInteger 类型，使用 printf 输出返回值的时候需要进行类型转换。详情请参考 8.4 节的内容。

▶ 代码清单 5-1 内存管理测试

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

int main()
{
    id obj = [[NSObject alloc] init];
    printf("init: %d\n", (int)[obj retainCount]);
    [obj retain];
    printf("retain: %d\n", (int)[obj retainCount]);
    [obj retain];
    printf("retain: %d\n", (int)[obj retainCount]);

    [obj release];
    printf("release: %d\n", (int)[obj retainCount]);
    [obj release];
```

```
    printf("release: %d\n", (int)[obj retainCount]);
    [obj release];

    return 0;
}
```

程序的输出如下所示。可以看出，对象刚生成时引用计数的值为1，每收到一次retain消息，引用计数的值就加1；而每收到一次release消息，引用计数的值就减1。

```
init: 1
retain: 2
retain: 3
release: 2
release: 1
```

5.2.3 释放对象的方法

在自定义类的时候，如果类的实例变量是一个对象类型，那么，在销毁类的对象的时候，也要给类的实例变量发送release消息。

通过给对象发送release消息可以放弃对这个对象的所有权，但如前所述，真正释放对象所占用的内存的是dealloc方法。

释放一个类的实例对象时，为了彻底释放该实例对象所保持的所有对象的所有权，需要为该类重写dealloc方法，在其中释放已经分配的资源，放弃实例变量的所有权。因为最终释放内存的是dealloc方法，所以不能重写release方法，如下例所示。

```
- (void)dealloc   // 重写的是 dealloc 方法而不是 release 方法
{
    /*
        这里通过 release 方法放弃子类中所有实例变量的所有权。
        其他用于释放前的善后操作也都写在这里。
    */
    [super dealloc];
}
```

在重写的dealloc方法中，在释放自身之前，首先要做好“善后工作”（释放所有需要释放的资源）。一般情况下，“善后工作”包括通过使用release放弃自身的实例变量的所有权。销毁对象的时候，不允许直接用dealloc，而是使用release。release会让引用计数减1，只有当引用计数等于0时，系统才会调用dealloc真正销毁这个对象。

子类的“善后工作”完成后，调用父类的dealloc方法来释放父类中定义的实例变量。这样，内存的释放会从子类一直向上直到NSObject，最终这个对象就会被彻底释放掉。

下面的分数计算器的例子中展示了如何重写dealloc。

5.2.4 访问方法和对象所有权

在通过访问方法等改变拥有实例变量所有权的对象时，必须注意实例变量引用计数的变化，合理安排release和retain的先后顺序。

让我们看一下 setMyValue 这个方法。

```
- (void)setMyValue:(id)obj
{
    [myValue release];
    myValue = [obj retain];
}
```

绝大多数情况下这个方法是没问题的，只有在参数 obj 和 myValue 是同一个对象的时候才会出错。这种情况下，在最开始进行release时，对象会被释放掉。针对这个问题，我们可以把release改为autorelease，或者改成图 5-3 中的任意一种写法。

图 5-3 安全的 setter 方法的写法

```
- (void)setMyValue:(id)obj
{
    [obj retain];
    [myValue release];
    myValue = obj;
}
```

（a）retain和release

```
- (void)setMyValue:(id)obj
{
    if (myValue != obj) {
        [myValue release];
        myValue = [obj retain];
    }
}
```

（b）使用if

如果不考虑对象所有权，而只是单纯赋值的话，则不需要保持和释放。不考虑所有权的 setter 方法如下所示。

```
- (void)setMyValue:(id)obj
{
    myValue = obj;
}
```

专栏：静态对象 COLUMN

C 语言中定义一个整数或结构体时，允许编译器在栈上为变量分配内存，如下所示。

```
struct node  rootnode;
```

那么，Objective-C 的变量能否不通过alloc来动态分配内存，而在栈上分配内存呢？

例如，我们打算生成类 Volume 的一个实例对象 v。

```
Volume  v;
```

编译上面的代码，会出现如下错误。

```
error: interface type cannot be statically allocated
```

实践证明，Mac OS X 的 Objective-C 不允许在栈上分配内存。

5.2.5 自动释放

在实际编程的时候，我们会遇到很多只用了一次就不再使用的对象。而如果这种对象也需要逐个释放，那将是一件很麻烦的事情。

Cocoa 环境的 Objective-C 提供了一种对象**自动释放**（autorelease）的机制。这种机制的基本思想是把所有需要发送release消息的对象都记录下来，等到需要释放这些对象时，会给这些对象一起发送release消息。其中，类 NSAutoreleasePool（自动释放池）就起到了记录的作用，下面，我们就对 NSAutoreleasePool 的使用方法进行详细说明。

首先让我们生成一个 NSAutoreleasePool 的实例对象。当向一个对象发送autorelease消息时，实际上就会将该对象添加到 NSAutoreleasePool 中，将它标记为以后释放。这个时候，因为这个对象没有被释放，所以还可以继续使用，对象引用计数的值也没有变化。但发送autorelease消息和发送release消息一样，相当于宣布放弃了对象的所有权。这样一来，当自动释放池被销毁时，池中记录的所有对象就都会被发送release消息。

autorelease方法的定义如下。

```
- (id) autorelease
```

其返回值是接收消息的对象。

本书中把自动释放池中登录的实际上相当于放弃了所有权的对象称为**临时对象**。

自动释放池的典型用法如下所示。

```
id pool = [[NSAutoreleasePool alloc] init];
/*
    在此进行一系列操作。
    给临时对象发送 autorelease 消息。
*/
[pool release];  /* 销毁自动释放池，自动释放池中所有的对象也被销毁 */
```

5.2.6 使用自动释放池时需要注意的地方

autorelease虽然是NSObject类的方法，但必须和类NSAutoreleasePool一起使用。

自动释放池中不存在对象时，不可以给对象发送autorelease消息，否则会出现运行时错误。我们这里讲解的自动释放池的例子都是在main()函数的最开始生成一个自动释放池，并在程序结束之前终止自动释放池。其实这样做没什么意义，因为就算不释放这些对象，程序结束之后操作系统也会回收这些内存。

某些需要长时间运行的代码段或大量使用临时对象的代码段可以通过定义临时的自动释放池来提高内存的利用率。例如，一个大量使用临时变量的循环中，经常会在循环开始时创建自己的自动释放池，在循环结束时释放这个自动释放池。

```
while ( ... ) {
    id pool = [[NSAutoreleasePool alloc] init];
    /*
        在此进行一系列操作。
    */
    [pool release];  /* 释放对象 */
}
```

但要注意的是，如果在循环过程中通过continue或break跳出循环的话，将可能导致自动释放池本身没被释放掉。另外，循环外也可以使用循环内生成的临时对象，但需要事先在循环内给对象发送retain消息，同时在循环外给对象发送autorelease消息。

可以给实例对象多次发送retain消息，相应的也可以给实例对象发送多次autorelease消息。这种情况就相当于同一个对象被重复登录到自动释放池中，只要autorelease和retain被成对发送就没有问题。

5.2.7 临时对象的生成

当你使用alloc、init方法创建一个对象时，该对象的初始引用计数为1。当不再使用该对象时，你要负责销毁它。

除了这种标准的创建对象的方法外，还有一种创建临时对象的方法。通过这种方法创建的对象都是临时对象，生成之后会被直接加入到内部的自动释放池，你不需要关心如何销毁它。

例如，Cocoa里用于处理字符串的类NSString，由UTF-8编码的C风格字符串生成NSString对象的方法有两个。

- (id) **initWithUTF8String:** (const char *) bytes
  alloc生成的实例对象的初始化方法，生成的实例对象的初始引用计数为1。

+ (id) **stringWithUTF8String:** (const char *) bytes
  生成临时变量的类方法，生成的实例对象会被自动加入到自动释放池中。

Objective-C 中的很多类都提供了这种生成临时对象的类方法。这种类方法的命名规则是，不以init开头，而以要生成的对象的类型作为开头。例如上例中的类方法以string开头。在表示数组的 NSArray 类中，生成临时对象的方法以array开头。

同综合使用alloc、init创建对象的方法相比，通过这种方法生成的对象的所有者不是调用 stringWithUTF8String 类方法的对象。

这种生成临时对象的类方法，在 Objective-C 中称为**便利构造函数**（convenience constructor）或**便利构造器**。一些面向对象的语言中会把生成对象的函数叫作构造函数，以和普通的函数进行区分。把在内部调用别的构造函数而生成的构造函数叫作便利构造函数。

在 Objective-C 语言中，便利构造函数指的就是这种**利用 alloc、init 和 autorelease 生成临时对象的类方法**。在后面要介绍的垃圾回收机制中，因为不需要区分临时对象，所以alloc和init组合在一起的类方法也称为便利构造函数。另外，对于需要参数的初始化方法，有时也会提供使用具有代表性的参数来生成实例对象的类方法，这些方法也称为便利构造函数。

5.3 节中将会展示如何定义便利构造函数。

5.2.8 运行回路和自动释放池

典型的图形界面应用程序（GUI 程序）往往会在休眠中等待用户操作，例如，操作鼠标或键盘点击菜单、按钮等。在用户发出动作之前，程序将一直处于空闲状态。当点击事件发生后，程序会被唤醒并开始工作，执行某些必要的操作以响应这一事件。处理完这一事件后，程序又会返回到休眠状态并等待下一个事件的到来。这个过程被叫作**运行回路**（runloop），更多详细内容请参考 15.1 节的内容。

Cocoa 在程序开始事件处理之前会隐式创建一个自动释放池，并在事件处理结束后销毁该自动释放池。所以，程序员在进行 Cocoa 的 GUI 编程时，就算不手动创建自动释放池，也可以使用临时对象。

5.2.9 常量对象

内存中常量对象（类对象，常量字符串对象等）的空间分配与其他对象不同，他们没有引用计数机制，永远不能释放这些对象。给这些对象发送消息retainCount后，返回的是 NSUIntegerMax[①]（其值为 0xffffffff，被定义为最大的无符号整数）。

常量对象的生成和释放操作和一般对象有所不同，有时需要重写retain和release方法的实现。但考虑到 ARC 和垃圾回收中无法重写这些方法，因此，从兼容性的角度来看，不推荐重写retain和release方法。

有的时候，我们可能会需要某个类仅能生成一个实例，程序中访问到这个类的对象时使用的都是同一个实例对象。在设计模式中这种情况称为**单例模式**（singleton）。Cocoa 框架中有很多单例模式的应用。例如控制程序运行的 NSApplication，除此之外调色板和字体设置也都只有一个实例存在。这些类通过以shared开头的类方法返回唯一的实例对象。

① NSUIntegerMax 的所有位都为 1，所以如果把它赋值给一个有符号数，这个有符号数的值就为 -1。

单例模式的实现如下所示，建议定义一个无论何时都只返回同一个实例对象的类方法。但要注意的是，在继承和多线程的情况下，这个实现还需要完善。

```
+ (MyComponent *)sharedMyComponent {
    static MyComponent *shared;
    if (shared == nil)
        shared = [[MyComponent alloc] init];
    return shared;
}
```

专栏：常量修饰符 const

COLUMN

ANSI C 的最新标准中增加了对 const 的支持。实际上，不仅仅是 C 语言，翻看一下 Objective-C 的 API 文档，你就会发现很多地方都使用到了 const。const 有很多使用方法，这里我们介绍一下 const 修饰指针时的情况。

首先，我们用 const 声明一个常量数组 days，要求不能改变数组的地址——days 的值，也不能修改 days 中的内容，只能使用 days 中的内容。

```
static const int days[] = { 31, 28, 31, 30, 31, 30 };
```

数组地址的定义如 p 所示，是一个指向常量的指针。没加 const 修饰符的指针 r 不能指向加了 const 修饰的数组的地址 days，否则编译的时候就会提示警告。

因为 p 是一个指针常量，所以不能改变 p 指向的内容。不仅不能修改常量数组 days 的内容，当 p 指向了一个不加 const 修饰符的普通数组的时候，也不能修改 p 所指向的内容。

```
static const int days[] = { 31, 28, 31, 30, 31, 30 };
static int temp[] = { 100, 500, 1000 };
const int *p;
int *r;

p = days;
p[1] = 0;       // 赋值非法，不能修改常量数组
r = days;       // 赋值非法，不能把一个常量指针赋值给一个非常量指针

r = temp;
r[1] = 0;
p = temp;
p[1] = 0;       // 不能改变常量指针所指向的内容
```

const 也可以修饰函数的传递参数。一个指针类型的参数加了 const 后，就表明无法改变指针所指向的内容。C 语言的标准库函数中很多地方都使用了 const，如下面的 strcpy 的例子所示，第二个参数就是 const 类型的指针。

```
char *strcpy(char *to, const char *from);
```

这个声明的含义是，第一个参数 to 所指向的内容是可以修改的，第二个参数 from 所指向的内容无法修改。

使用 const 能够表明指针所指向的内容是否可以修改，在一定程度上可以提高程序的安全性和可靠性。因此，在 Objective-C 中也请尽量使用 const。

5.3 分数计算器的例子

5.3.1 分数类 Faction

下面让我们在手动引用计数的内存管理方式下，定义一个简单的分数类。并基于这个类做一个支持分数计算的计算器。

让我们首先定义分数类 Fraction，它的接口部分如下所示。

代码清单 5-2 分数类的接口部分（Fraction.h）

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

@class NSString;

@interface Fraction : NSObject
{
    int sgn;    // sign（符号位）
    int num;    // numerator（分子）
    int den;    // denominator（分母）
}

+ (id)fractionWithNumerator:(int)n denominator:(int)d;
- (id)initWithNumerator:(int)n denominator:(int)d;
- (Fraction *)add:(Fraction *)obj;
- (Fraction *)sub:(Fraction *)obj;
- (Fraction *)mul:(Fraction *)obj;
- (Fraction *)div:(Fraction *)obj;
- (NSString *)description;

@end
```

让我们说明一下这个类中各个变量和方法的意义。实例变量 sgn 代表符号位，值可以为 1 或 -1。num 和 den 分别表示分子和分母。

类方法`fractionWithNumberator:denominator:`用于生成分数类的临时对象，有两个参数分别用于指定分子和分母。方法`initWithNumerator:denominator:`是指定初始化函数，两个参数分别用于初始化分子和分母。分子或分母是负数时，整个分数的符号位就为负。

四个类方法`add`、`sub`、`mul`、`div`用于分数的四则运算，计算结果会返回 Fraction 类的一个新对象。四则运算类方法的消息接收者和参数在运算过程中都不会发生变化。消息`description`返回用 NSString（NSString 是 Cocoa 环境中用于表示字符串的类）表示的分数值。

下面让我们看一下这个类的实现部分。

▶ 代码清单 5-3 分数类的实现部分（Fraction.m）

```
#import "Fraction.h"
#import <Foundation/NSString.h>
#import <stdlib.h>

@implementation Fraction

static int gcd(int a, int b) // Greatest Common Divisor : 最大公约数
{
    if (a < b)
        return gcd(b, a);
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

/* Local Method① */
- (void)reduce
{
    int d;

    if (num == 0) {
        sgn = 1;
        den = 1;
        return;
    }
    if (den == 0) { // infinity
        num = 1;
        return;
    }
    if ((d = gcd(num, den)) == 1)
        return;
    num /= d;
    den /= d;
}

/* 生成临时对象 */
+ (id)fractionWithNumerator:(int)n denominator:(int)d
{
    id f = [[self alloc] initWithNumerator:n denominator:d];
    return [f autorelease];
}

#define  SIGN(a)    ( ((a) >= 0) ? 1 : (-1) )

/* 指定初始化函数 */
- (id)initWithNumerator:(int)n denominator:(int)d
{
    if ((self = [super init]) != nil) {
        sgn = SIGN(n) * SIGN(d);
        num = abs(n);
        den = abs(d);
        [self reduce];
    }
```

① 用于约分的局部方法。——译者注

```
    return self;
}

- (Fraction *)add:(Fraction *)obj
{
    int n, d;

    if (den == obj->den) {
        n = sgn * num + obj->sgn * obj->num;
        d = den;
    }else {
        n = sgn * num * obj->den + obj->sgn * obj->num * den;
        d = den * obj->den;
    }
    return [Fraction fractionWithNumerator:n denominator:d];
}

- (Fraction *)sub:(Fraction *)obj
{
    Fraction *tmp;
    int n = -1 * obj->sgn * obj->num;
    tmp = [Fraction fractionWithNumerator:n denominator:obj->den];
    return [self add: tmp];
}

- (Fraction *)mul:(Fraction *)obj
{
    int n = sgn * obj->sgn * num * obj->num;
    int d = den * obj->den;
    return [Fraction fractionWithNumerator:n denominator:d];
}

- (Fraction *)div:(Fraction *)obj
{
    int n = sgn * obj->sgn * num * obj->den;
    int d = den * obj->num;
    return [Fraction fractionWithNumerator:n denominator:d];
}

- (NSString *)description
{
    int n = (sgn >= 0) ? num : -num;
    return (den == 1)
        ? [NSString stringWithFormat:@"%d", n]
        : [NSString stringWithFormat:@"%d/%d", n, den];
}

@end
```

静态函数 gcd() 使用了辗转相除法求最大公约数。接着，reduce 方法中使用了 gcd 的结果进行约分。reduce没有在接口部分中定义，也是一个局部方法。分母为 0 的情况会被当作无限大来处理，没什么实用意义。

类方法fractionWithNumerator:denominator:先调用初始化方法生成一个分数对象，然

后向其发送autorelease消息把这个对象加入自动释放池中。这就是上节中介绍到的便利构造函数的一个实际用例。

初始化函数initWithNumberator:denominator:被用于初始化实例对象。分子和分母都是非负整数，整个分数的符号位由 sgn 来表示。同时也会对分子分母进行约分。生成的对象的所有者就是调用这个函数的对象。

再让我们来看看add:方法。add:方法的参数也是一个 Fraction 类型的对象，所以可以通过->直接访问它的实例变量。变量 n 代表加法运算后得到的新分数的分子，d 代表新分数的分母。如果双方的分母一致，就直接对分子进行带符号的加法运算。而如果双方的分母不一样，则如同我们在小学时学到的一样，先对双方的分母进行通分，然后再进行加法运算。最后通过方法fractionWithNumber:denominator:返回分数对象，分数对象的分子和分母分别是 n 和 d。在方法fractionWithNumber:denominator:中会自动进行约分操作。

sub:方法会反转输入分数的符号位，然后再利用上面的add:方法做减法运算。mul:和div:方法则分别完成分数的乘法和除法运算。

description方法的功能是生成一个 NSString 类型的结果，来表示分数类的分数。类方法stringWithFormat:能够像 printf() 格式化输出一样，按照指定的格式生成字符串。如上节所介绍的stringWithFormat:就是以 string 开头的，所以生成的对象是临时对象。

5.3.2 保存计算结果的 FracRegister 类

计算器一般都有一个液晶显示屏，来显示计算的结果。这里我们也来实现一个具有相同功能的类 FracRegister，并尝试为其增加输入错误时可以显示上次的计算结果这一功能（也就是 undo 功能[①]）。

代码清单 5-4 分数寄存器的接口部分（FracRegister.h）

```
#import <Foundation/NSObject.h>
#import "Fraction.h"

@interface FracRegister : NSObject
{
    Fraction *current;
    Fraction *prev;
}

- (id)init;
- (void)dealloc;
- (Fraction *)currentValue;
- (void)setCurrentValue:(Fraction *)val;
- (BOOL)undoCalc;
- (void)calculate:(char)op with:(Fraction *)arg;

@end
```

① 此处的 undo 功能和第 15 章中的 undo 没有任何关系。

变量 current 和 prev 分别表示当前的计算结果和上次的计算结果，并为 current 变量定义 get 和 set 访问方法。方法calculate用于计算，计算的类型用 char 字符来表示。

代码清单 5-5 分数寄存器的实现部分（FracRegister.m）

```
#import "FracRegister.h"
#import <stdio.h>

@implementation FracRegister

- (id)init {
    if ((self = [super init]) != nil)
        current = prev = nil;
    return self;
}

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

- (Fraction *)currentValue { return current; }

- (void)setCurrentValue:(Fraction *)val
{
    [val retain];
    [current release];
    current = val;
    [prev release];
    prev = nil;
}

- (BOOL)undoCalc
{
    if (prev == nil)
        return NO;
    [current release];
    current = prev;
    prev = nil;
    return YES;
}

- (void)calculate:(char)op with:(Fraction *)arg
{
    Fraction *result = nil;

    if (current != nil && arg != nil)
        switch (op) {
        case '+':
            result = [current add: arg];
            break;
        case '-':
            result = [current sub: arg];
            break;
```

```
        case '*':
            result = [current mul: arg];
            break;
        case '/':
            result = [current div: arg];
            break;
        default: // Error
            break;
        }
    if (result != nil) {
        [result retain];  // 保存运算结果
        [prev release];
        prev = current;
        current = result;
    }else
        printf("Illegal Operation\n");
}

@end
```

初始化函数init中把 current 和 prev 的值都设为了 nil。因为即使不通过 init 函数初始化，current 和 prev 的初始值也会是 0，所以这个init函数不是必须的。方法dealloc中释放了 current 和 prev。方法undoCalc执行 undo 操作，如果 prev 不为 nil，则释放当前值 current，并把 prev 赋值给 current。如果 prev 为 nil，则无法完成 undo 操作，返回 NO。

setCurrentValue:是变量 current 的 setter 方法，用于设置 current 的值，同时将 prev 设为 nil。这里也可以把 prev 设为 current 的原值。

下面来看一下方法calculate:with:。参数 op 可以是 +、-、*、/ 中的任何一个。如果不是这四个符号，程序就会提示出错。代表当前分数值的 current 和代表新输入的分数值的 arg 是 nil 时，程序也会提示出错。输入一切正常的情况下，会进行两个分数类的四则运算，结果存入 result。result 是临时变量，所以要给其发送retain消息。此外，为了实现 undo 功能，要先释放 prev 中保存的内容，然后把 current 的值赋给 prev，最后再把计算的结果赋值给 current。

5.3.3 主函数和执行示例

代码清单 5-6 中演示了使用以上定义的类来进行运算的 main 函数。

main 函数的执行流程是：首先输入一个分数（或整数），接着输入一个运算符（+、-、*、/）和一个分数。这时程序会输出运算结果。然后再输入一个运算符和分数，程序再次输出运算结果，如此循环。除了可以输入运算符之外，还可以输入代表控制命令的英文字母。例如，输入 q 代表程序结束；输入 c 代表 clear，即清除当前的计算结果并重新开始；输入 u 代表 undo 操作，即使用上次的计算结果代替当前的计算结果。需要注意的是，undo 操作只支持一次，不可连续进行 undo 操作。

函数 getFraction() 用于从命令行读入分数值，支持的输入格式有分数、整数和带分数。分数的输

入格式为 1/2。带分数的输入格式为 1'1/2，代表“1 又 1/2”，也就是 3/2。getFraction() 函数会使用 sscanf() 循环测试所输入的字符串是否符合以上 3 种格式中的任何一种。sscanf() 和 scanf() 使用起来比较简单，但这两个函数在对缓冲区的处理方面有缺陷，因此，在生产系统中使用的时候一定要注意。

函数 readFraction 逐行读入用户的输入，直到读入了能够读入的内容为止。

代码清单 5-6 分数计算器的 main 函数

```
#import "Fraction.h"
#import "FracRegister.h"
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSString.h>
#import <stdio.h>

#define BUFSIZE  80

static Fraction *getFraction(const char *buf)
{
    int a, b, c;

    if (sscanf(buf, "%d'%d/%d", &a, &b, &c) == 3)
        b = (a < 0) ? (a * c - b) : (a * c + b);
    else if (sscanf(buf, "%d/%d", &b, &c) != 2) {
        if (sscanf(buf, "%d", &b) == 1)
            c = 1;
        else
            return nil;
    }
    return [Fraction fractionWithNumerator:b denominator:c];
}

static Fraction *readFraction(FILE *fp)
{
    char buf[BUFSIZE];
    Fraction *frac = nil;

    for ( ; ; ) {
        if (fgets(buf, BUFSIZE, fp) == NULL) // EOF is input
            return nil;
        if ((frac = getFraction(buf)) != nil)
            break;
    }
    return frac;
}

int main(void)
{
    char com[BUFSIZE], cc;
    BOOL contflag = YES;
    NSAutoreleasePool *pool, *tmppool;
    FracRegister *reg;
    Fraction *val;

    pool = [[NSAutoreleasePool alloc] init];
    reg = [[[FracRegister alloc] init] autorelease];
```

```
    while (contflag) { ❶
        tmppool = [[NSAutoreleasePool alloc] init];
        printf("? ");
        if ((val = readFraction(stdin)) != nil)
            [reg setCurrentValue:val];
        else
            contflag = NO;

        while (contflag) { ❷
            if (fgets(com, BUFSIZE, stdin) == NULL
                || (cc = com[0]) == 'q' || cc == 'Q') { ❸
                contflag = NO;
                break;
            }
            if (cc == 'c' || cc == 'C') // Clear
                break;
            if (cc == '+' || cc == '-' || cc == '*' || cc == '/') {
                if ((val = getFraction(com+1)) == nil) ❹
                    val = readFraction(stdin);
                if (val == nil) { // EOF
                    contflag = NO;
                    break;
                }
                [reg calculate:cc with:val];
            }else if (cc == 'u' || cc == 'U') { // Undo
                if (![reg undoCalc])
                    printf("Can't UNDO\n");
            }else {
                printf("Illegal operator\n");
                continue;
            }
            printf("= %s\n",
                [[[reg currentValue] description] UTF8String]);
        }
        [tmppool release]; ❺
    }
    [pool release];
    return 0;
}
```

main 函数中先创建了一个 FraceRegister 类的对象 reg。代码中的 ❶ 和 ❷ 处是一个循环，当 contflag 变量为真的情况下处理继续，否则程序终止。❷ 处的循环在程序进行运算或执行 undo 操作期间会继续，并在循环的最后在类 FracRegister 的实例 reg 中显示当前的计算结果。方法 `description` 返回的是 NSString 类型的字符串，这里使用了`UTF8String`方法以将其转换为 C 风格的字符串。关于 UTF8String 的详细介绍，请参考本书第 9 章中关于 NSString 的说明。

❸ 处首先读入一行输入，然后判断首字符是控制命令还是运算符。如果是运算符的话就继续判断运算符之后是否是分数，如果是分数就调用 getFraction 进行计算，如果不是则调用 readFraction 继续读入。

在 ❷ 处，如果输入了控制命令 c、q，或者用 contronl-D 结束了程序，就会跳出循环。控制命令 c 代表清空 register 的值，q 代表退出程序。跳出循环后，❺ 处会释放自动释放池 tmppool。由于

tmppool 是在 ❶ 之后生成的，因此，当清空 register 的值或退出程序的时候，到此为止生成的临时对象都会被释放掉。

释放临时对象的时机是可变的，既可以按测试程序中写的这样操作，也可以每进行一次内循环就释放一次。如果你也觉得手动管理对象的引用计数麻烦的话，也可以不使用临时对象。

该例子的执行结果如下所示。

```
% ./fraction
? 5/12
* 8/7
= 10/21
+ 1/3
= 17/21
- 2/21
= 5/7
c
? 2'3/4
+ 1'1/4
= 4
- 1'2/3
= 7/3
* 1'1/2
= 7/2
q
```

5.4 ARC 概要

5.4.1 什么是 ARC

采用引用计数方式管理内存时需要程序员管理所有生成对象的**所有权**（object ownership）。程序员需要清楚地了解获得 / 放弃对象所有权的时机，并在适当的位置插入`retain`、`release`或`autorelease`函数。

ARC（Automatic Reference Counting，自动引用计数）是一个编译期技术，利用此技术可以简化 Objective-C 在内存管理方面的工作量。ARC 通过在编译期间添加合适的 `retain/release/autorelease` 等函数，来确保对象被正确地释放。编译器会根据传入的变量是局部变量还是引用变量，返回对象的方法是不是初始化方法（initialize）等信息来推断应当在何处加入 `retain/release/autorelease` 等函数。这样一来，程序员就可以不再关心是否通过 `retain` 或 `release` 正确地释放了每个使用过的对象，而将更多的精力用于开发程序的核心功能。

现在，ARC 只能管理 Obective-C 的对象，不能管理通过 malloc 申请的内存。

本书虽然不会详细说明 ARC 到底在何处加入了什么样的代码，但为了让大家更好地理解 ARC

的工作原理，下面将举例说明一下 ARC 是如何工作的。

现在有一个指向某个对象的变量 s，假设将一个 w 对象赋值给 s。

```
s = w;
```

赋值操作之后，s 原来指向的对象将不能通过 s 来访问，s 需要放弃该对象的所有权。与此同时，s 需要获得新赋值的对象 w 的所有权。编译以上代码的时候，ARC 相当于生成了以下代码。

```
[w retain];          // 获得 w 的所有权
id _old = s;         // _old 用于临时缓存 s
s = w;               // 赋值之后，s 放弃原来对象的所有权
[_old release];      // ( 要考虑到对象的释放再次涉及 s 时的情况 )
```

让我们来看一个更实际的例子。下面是上节中的分数计算器中分数类 FracRegister 的 `setCurrentValue:` 方法。这个方法的功能是重设代表计算器当前值的实例变量 current 为 val，同时把代表上一个计算结果的 prev 设为空。这段赋值的代码在 ARC 有效的情况下编译后，ARC 会自动添加被注释掉的行，程序员就不需要手动添加这些代码了。

```
- (void)setCurrentValue:(Fraction *)val
{
    // [val retain];            可省略
    // [current release];       可省略
    current = val;
    // [prev release];          可省略
    prev = nil;
}
```

如上面的代码所示，编译器会根据赋值操作、变量的初始化、变量的生命周期等因素，在合适的位置自动加入保持和释放 (`retain/release`) 相关的代码。至于在什么位置加入什么样的代码，编译器会自己判断，并进行优化，删除多余的`retain/release`对。虽然最后插入的代码以及代码插入的位置和程序员的预想可能会有差别，例如并没有使用`release`而使用了`autorelease`等，但程序员并不需要了解这些实现的细节。

为了让对象能够像程序员预先设计的那样被保持或销毁，需要给编译器正确的信息。也就是说，程序员在编程时需要遵守一定的规则。

下面将说明利用 ARC 时程序员需要遵守的规则和一些新增的概念和语法。

5.4.2 禁止调用引用计数的相关函数

ARC 有效的程序，不能调用以下这些跟引用计数相关的方法。

```
retain
release
```

```
autorelease
retainCount
```

当然也不能使用这些函数的 selector（例如 @selector（retain）等，详情请参考 8.2 节）。

5.4.3 管理自动释放池的新语法

ARC 中禁止使用 NSAutoReleasePool，而是使用新语法 @autoreleasepool 来管理自动释放池。

手动管理内存的情况下，自动释放池的使用方法如下所示。

```
id pool = [[NSAutoreleasePool alloc] init];
    /* 进行一系列操作 */
    /* 此处不可以使用 break、return、goto 之类的语句 */
[pool release];  /* 释放对象 */
```

新的语法如下所示。

```
@autoreleasepool {
    /* 在此进行一系列操作 */
    /* 可以使用 break、return,、goto 等语句 */
}
```

旧的写法中，在自动释放池被初始化至被释放期间，不可以使用 break、return 和 goto 等跳转语句，否则对象就有可能无法被成功释放。

而新的语法的情况下，因为运行到 @autoreleasepool 块外的时候进行对象的释放，所以可以使用跳转语句。

另外，@autoreleasepool 在非 ARC 模式下也能使用，并且使用 @autoreleasepool 比使用 NSAutoReleasePool 性能更好效率更高，所以无论是否使用 ARC，都推荐使用这种新的语法。

5.4.4 变量的初始值

在 ARC 中，未指定初始值的变量（包括局部变量）都会被初始化为 nil。

但是，对于用 _autoreleasing 和 _unsafe_unretained 修饰的变量来说，初始值是未定的。

而对象以外的变量的初值则和以前是一样的。

5.4.5 方法族

采用引用计数方式管理内存时，创建对象时就会拥有这个对象的所有权。例如，使用以alloc开头的类方法生成对象，并且使用以init开头的类方法来初始化对象的时候，就会获得这

个对象的所有权。另外，使用名称中包含new、copy或mutableCopy的方法复制对象的时候，也会获得这个对象的所有权。

采用引用计数方式管理内存时，如果不使用alloc/init/new/copy/mutableCopy这些方法、或者不使用retain来保留一个对象，就不能成为对象的所有者。另外，只有使用release或者autorelease，才能够放弃这个对象的所有权。

这些规定被叫作**所有权策略**（ownership policy）。换句话说，对象实例该由谁来释放，并不取决于编程语言方面的要求，而是由编程逻辑决定的。

另一方面，由于 ARC 允许混合链接手动内存管理和自动内存管理的代码，所以针对到底哪个方法同对象的生成和复制相关这一问题，不能使用“init ...方法名”这种命名方式来区分，而需要定义能够让编译器明确区分的方法。同对象生成相关的方法集合叫作**方法族**（method family）。

一个方法要属于某个方法族，除了需要满足返回值和方法的类别方面的要求外，也需要满足以下命名规则，即选择器同方法族名相同（开头的 _ 可忽略），或选择器的名字由方法族名加上非小写字母开头的字符串构成。

例如，以下选择器符合 init 方法族的要求。

```
init
initToMemory
initWithData:
_init:locale:
```

而以下这些选择器则不符合要求。

```
initialize                  ……init 的后面不能接小写字母
initwithString:locale:      ……init 的后面不能接小写字母
do_initWithData:            ……不以 init 开头
```

目前为止一共定义了 5 个方法族，具体定义和要求如下。**可被保持的对象**（retainable object）指的是 Objective-C 的对象或第 14 章中说明的 block 对象。

alloc 方法族

以alloc开头的方法表示调用者对被创建的对象拥有所有权，返回的对象必须是可以被retain的。

copy 方法族

以copy开头的方法表示调用者对被创建的对象拥有所有权，返回的对象必须是可以被retain的。

mutableCopy 方法族

以mutableCopy开头的方法表示调用者对被创建的对象拥有所有权，返回的对象必须是可以被retain的。

new 方法族

以new开头的方法表示调用者对被创建的对象拥有所有权，返回的对象必须是可以被retain的。

init 方法族

以init开头的方法必须被定义为实例方法，它一定要返回 id 类型或父类、子类的指针。

除了以init开头的方法之外，以alloc/new/copy/mutableCopy开头的方法都既可以是类方法也可以是实例方法。依照 Objective-C 中的命名惯例，调用以 alloc/new/copy/mutableCopy/init 开头的方法时，需要将对象所有权返回给调用端，由调用端release生成的对象。另一方面，如果你想由调用端来release一个方法返回的对象，而这个方法的名字又不是以上述关键字开头的话，ARC 就有可能不会释放这个对象，从而造成内存泄漏。

给方法命名时必须遵循命名规则。大家可能会在无意中使用 new 或 copy 开头的名字作为方法名，例如给一个换行的方法命名为newLine，给一个获取版权的方法命名为copyRight等。这些命名都是不正确的，有可能会造成误释放，也有可能在编译时提示警告或错误。因此要切记**严守内存管理相关的函数命名规则**。

5.4.6 方法 dealloc 的定义

手动管理内存的情况下，当变量的引用计数为 0 的时候，dealloc就会被调用并释放内存。所以，实例对象被释放之前的所有“善后操作”都被定义在了dealloc中。

ARC 中，当对象被释放的时候，对象的dealloc函数也会被调用。但有一些需要注意的地方。

首先，要被释放的对象的实例变量被另外一个对象保持的情况下，ARC 会自动进行释放，因此不需要做任何处理（手动内存管理时，需要执行release操作让变量的引用计数减 1）。

其次，ARC 编程的时候不能显式调用dealloc方法，包括使用 @selector（dealloc）等的隐式调用。手动内存管理方式下编程时的一条重要规则是，除了调用父类的dealloc方法之外，不允许显式调用dealloc方法。ARC 有效的时候，不允许调用父类的dealloc。尽管你可以创建一个定制的dealloc方法来释放资源而不是实例变量，但也不要调用 [super dealloc]，因为编译器会自动处理这些事情。如果你在代码中调用了 [super dealloc]，就会出现编译错误（error: ARC forbids explicit message send of ‘dealloc’）。

让我们来看一个例子。Fig5-4（a）中定义了一个类 MyBuffer。图（b）是手动管理内存时 dealloc 的实现，图（c）是 ARC 有效时dealloc的实现。

图 5-4 dealloc 方法的定义

```
@interface MyBuffer : NSObject {
    NSString *filename;
    NSView   *monitorView;
    char     *buffer;
    FILE     *fp;
}
...
```

（a）类MyBuffer的接口

```
- (void) dealloc {
    [filename release];
    [monitorView release];
    if (buffer != NULL)
        free(buffer);
    if (fp != NULL)
        fclose(fp);
    [super dealloc];
}
```

（b）手动管理内存方式下dealloc的实现

```
- (void) dealloc {
    if (buffer != NULL)
        free(buffer);
    if (fp != NULL)
        fclose(fp);
}
```

（c）ARC有效时dealloc的实现

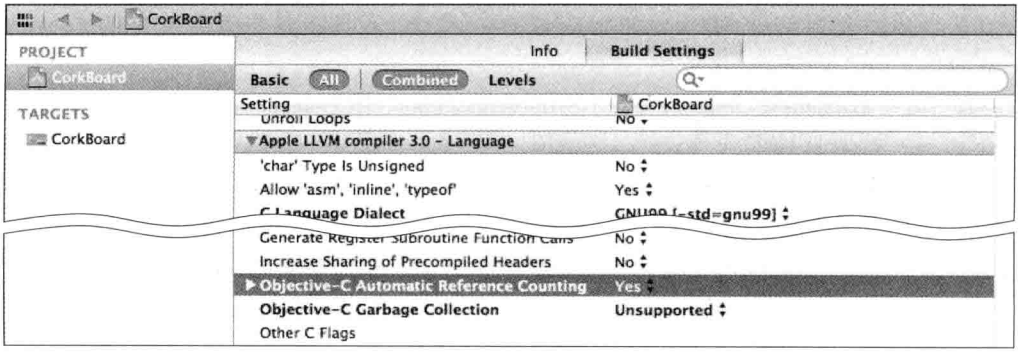
5.4.7 使用 ARC 的程序的编译

启用 ARC 编译代码时，不能使用 gcc 而要使用 clang。同时，编译选项中需加上 -fobjc-arc。与此相反，也可以通过设置编译选项 -fno-objc-arc 来明确告诉编译器不想使用 ARC。

自 MacOS X 10.7 与 iOS 5 开始可以使用 ARC。在 MAC OS X 10.6 雪豹中也可以使用 ARC，但有一些限制，例如不支持弱引用（weak references，更多关于弱引用的内容请参考后面的介绍）。

在 Xode 中创建一个新的工程时，可以在项目初期设定中选择 ARC 是否有效，也可以在项目的编译设定中选择使用哪种内存管理方式。图 5-5 是 Xcode4.2 截图的一部分。

图 5-5 设置 Xcode 4.2 的编译选项



如前所述，使用 ARC 的代码和不使用 ARC 的代码可以混合链接。

5.4.8 ARC 的基本注意事项

下面总结一下使用 ARC 编程时的注意事项。总结虽然并不全面，但基本上能够解决绝大多数使用 ARC 编程时碰到的问题。

- **不能在程序中定义和使用下面这些函数：`retain`、`release`、`autorelease`和`retainCount`**
- **使用 @autoreleasepool 代替 NSAutoreleasePool**
- **方法命名必须遵循命名规则，不能随意定义以 alloc/init/new/copy/mutableCopy 开头且和所有权操作无关的方法**
- **不用在 dealloc 中释放实例变量（但可以在 dealloc 中释放资源），也不需要调用 [super dealloc]**
- **编译代码时使用编译器 clang，并加上编译选项 -fobjc-arc**

5.4.9 使用 ARC 重构分数计算器

让我们使用 ARC 来重构一下分数计算器的例子。

首先来看一下类 Fraction 的类方法`fractionWithNumerator:`。ARC 有效的情况下，需要删除方法中使用的`autorelease`。因为方法`fractionWithNumberator:`并不以 `alloc/new/copy/mutableCopy/init`开头，所以这个方法生成的对象是一个临时对象，编译器会将生成的对象自动放入 autoReleasePool 中。

```
+ (id)fractionWithNumerator:(int)n denominator:(int)d
{
    return [[self alloc] initWithNumerator:n denominator:d];
}
```

下面让我们来看看实现文件 FraceRegister.m 中都有哪些需要修改的地方。首先是`dealloc`方法。因为当前的`dealloc`方法只`release`了类 FraceRegister 的实例变量，所以可以将整个`dealloc`方法删除掉。其次是方法`setCurrentValue:`，按照 ARC 中的要求，需要删除其中的 `retain` 和 `release` 方法。最后是方法`undoCalc`和`calculate:with:`，同样需要删除其中的 `retain` 和`release`方法。

main 函数中需要用 @autoreleasepool 替换 NSAutoreleasePool。因为 @autoreleasepool 中允许使用 break 或 goto 语句，所以可以删除掉原来的二重循环，只保留一重循环（原来之所以使用二重循环是为了不脱离 NSAutoreleasePool 的作用域）。

```
int main(void)
{
    //…省略，删除 NSAutoreleasePool 的变量

    @autoreleasepool {
        reg = [[FracRegister alloc] init];

        while (contflag)
            @autoreleasepool {
                printf("? ");
                //…省略…
            }
    }
    return 0;
}
```

编译方面只需要新增编译选项 -fobjc-arc 即可，下面是编译和链接的操作。

```
clang -Wall -fobjc-arc -c -o Fraction.o Fraction.m
clang -Wall -fobjc-arc -c -o FracRegister.o FracRegister.m
clang -Wall -fobjc-arc -c -o main.o main.m
clang -o fraction Fraction.o FracRegister.o main.o -framework Foundation
```

这里我们通过自己动手重构代码完成了 ARC 转换。实际上，Xcode 为我们提供了一个 ARC 自动转换工具，可以帮你将代码转为支持 ARC 的代码。在 Xcode 4.2 的菜单中依次选择 Edit → Refactor，就能找到这个小工具 Convert to Objective-C ARC。使用了垃圾回收的代码不能被自动转换为支持 ARC 的代码。

5.5 循环引用和弱引用

如前所述，在简单的程序中，只需要删除操作引用计数的函数，就可以让程序过渡到 ARC 模式下。但是，除了简单的程序之外，还有很多复杂的程序，这种情况下就需要更精巧的处理。本节中将介绍 ARC 内存管理时存在的循环引用问题，以及为了解决这个问题而引入的弱引用的概念。

5.5.1 循环引用

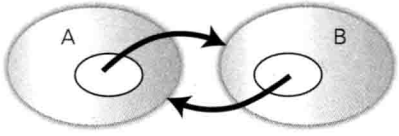
让我们以一个简单的类 People 为例来说明一下什么是循环引用。为了便于理解，这里没有使用 ARC 而使用了手动内存管理的方法，但两者的原理是一样的。

```
@interface People : NSObject {
    id friend;
}
- (void)setFriend:(id)obj;
@end

@implementation People
- (void)setFriend:(id)obj {
    d tmp = friend;  // 相当于 5-4 中的赋值操作
    [obj retain];
    friend = obj;
    [tmp release];
}
- (void)dealloc {
    [friend release];
    [super dealloc];
}
@end
```

让我们看一下 People 的两个实例对象（A 和 B）互相引用的情况，如图 5-6 所示。

```
    [A setFriend: B];
    [B setFriend: A];
```

图 5-6 互相引用的两个对象



这种情况下，就算想将 A 和 B 都释放掉，但按照规则，也只有等到释放 B 之后才有可能释放 A（否则 A 的引用计数为 1，无法被释放），同样，只有释放 A 后才可能释放 B（原因同前），而当双方都在等待对方释放的时候，就形成了循环引用，结果两个对象都不会被释放，这样就会造成内存泄漏。虽然通过设定 A 的 friend 为 nil，手动打破 A 和 B 的循环引用关系可以完成内存释放，但这种做法比较麻烦，而且容易出错。

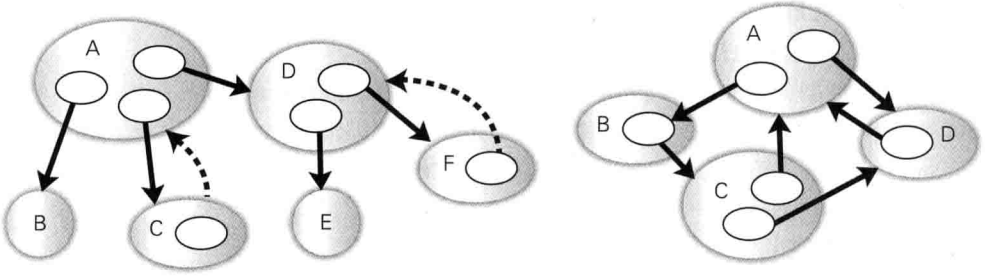
像这种两个对象互相引用，或者像 A 持有 B、B 持有 C、C 持有 A 这样多个对象的引用关系形成了环的现象，叫作**循环引用**或**循环保持**（retain cycle）。循环引用会造成内存泄漏，只有打破循环引用关系才能够释放内存。

5.5.2 所有权和对象间的关系

程序中有多个对象的时候，应该明确对象间的所有权关系。但根据程序复杂程度的不同，所有权关系有时很容易明确，有时则很难确定。

图 5-7 展示了两种比较常见的对象间的关系。（1）中对象间的关系类似于树形结构，父节点是子节点的所有者。用实际的例子来比喻的话，就像窗口上有几个控件对象，而其中每个控件本身又是由另外一些小控件组成的。（1）中，对象 A 是 B、C、D 的所有者，D 是 E、F 的所有者。释放 A 的时候要先释放 B、C、D，释放 D 的时候要先释放 E 和 F。

图 5-7　对象间的引用关系



（1）层次引用关系　　（2）网状引用关系

图 5-7（1）中，对象 C 指向 A、对象 F 指向 D 的虚线，就是指向自己父节点的指针，也被叫作反向指针（backpointer）。为了避免循环引用的发生，可以使用无关所有权的指针来实现这种关系。

图 5-7（2）中各个对象之间的引用关系更像是一种网状关系，这种情况下比较容易发生循环引用的问题。使用和所有权无关的指针时要注意的是，当你需要使用对象时，对方可能已经在不知不觉中被释放了。

总而言之，拥有所有权的实例变量和只通过指针指向、不拥有所有权的变量在内存方面的处理截然不同，如果处理不当就会造成内存泄漏。

5.5.3　弱引用

到目前为止，我们介绍 ARC 时提到的实例变量都是拥有所有权的实例变量（强引用类型，默认属性）。但为了避免循环引用的出现，我们还需要另外一种类型的变量，这种变量能够引用对象，但不会成为对象所有者，不影响对象本身的回收。

为了实现这个目的，ARC 中引入了**弱引用**（weak reference）的概念。弱引用是通过存储一个指向对象的指针创建的，且不保留对象。Objective-C 中用 __weak 修饰符来定义弱引用，如下例所示。

```
__weak id temp;
__weak NSObject *cacheObj;
```

通常声明的未加 __weak 修饰符的变量都是**强引用**（strong reference）类型的变量，声明时也可以通过加上 __strong 修饰符来明示变量是强引用类型。函数和方法的参数也是强引用类型。

声明变量的时候，__weak 或者 __strong 可以出现在声明中的任意位置，如下面的例子所示。但有一点要注意的是，最后一个声明的变量 f 前不可省略 __weak。

```
__weak NSObject *a, *b;
NSObject __weak *c, *d;
NSObject * __weak e, * __weak f;
```

对强引用变量赋值之后，编译器会自动插入变量引用计数加 1 的代码。当强引用变量不再使用某个对象时，编译器会自动插入变量引用计数减 1 的代码。与此相对，弱引用的情况下，无论是对变量赋值还是解除引用，变量的引用计数都不会发生变化。

强引用和弱引用都会被隐式地初始化为 nil。

这种用于修饰指针类型变量的修饰符被叫作**生命周期修饰符**（lifetime qualifier）或所有权修饰符。生命周期修饰符一共有四种，除了我们已经介绍的 strong、__weak 之外，还有 __autoreleasing、__unsafe_unretained。我们会在后面介绍另外两种修饰符的使用方法。

5.5.4 自动 nil 化的弱引用

弱引用会在其指向的实例对象被释放后自动变成 nil，这就是弱引用的**自动 nil 化**功能。也就是说，即使弱引用指向的实例对象在不知不觉中被释放了，弱引用也不会变成野指针。

让我们通过代码清单 5-7 中的例子来看看弱引用自动 nil 化是如何工作的。

例子中的 People 类有一个弱引用类型的实例变量 friend。方法 nameOfFriend 用于返回 friend 指向的对象的 name 实例变量，如果 friend 为 nil，则返回 none。

main 函数中会声明 People 类的两个实例对象 a 和 b，变量 b 会在从自动释放池中退出的时候被释放掉。

代码清单 5-7 使用弱引用的一个例子（friend.m）

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

@interface People : NSObject {
    const char *name;
    __weak People *friend;
}
- (id)initWithName:(const char *)p;
- (void)setFriend:(id)obj;
- (const char *)nameOfFriend;

@end

@implementation People

- (id)initWithName:(const char *)p {
    if ((self = [super init]) != nil) {
        name = p;
        friend = nil;
    }
    return self;
}
```

```
- (void)setFriend:(id)obj { friend = obj; }

- (const char *)nameOfFriend {
    if (friend == nil) return "none";
    return friend->name;
}

@end

int main(void)
{
    People *a = [[People alloc] initWithName:"Alice"];
    printf("Friend: %s\n", [a nameOfFriend]);
    @autoreleasepool {
        People *b = [[People alloc] initWithName:"Bob"];
        [a setFriend: b];
        printf("Friend: %s\n", [a nameOfFriend]);
        b = nil;// 变量b的对象被释放
    }
    printf("Friend: %s\n", [a nameOfFriend]);
    return 0;
}
```

程序执行后的输出如下所示。

```
Friend: none
Friend: Bob
Friend: none
```

最后一行的输出表示对象 a 的实例变量 friend 已经变成了 nil，也就是说，弱引用 friend 指向的实例变量被释放之后，friend 自动变成了 nil。

在 ARC 条件下编程时最需要注意的就是不要形成循环引用。通过使用弱引用既可以防止生成循环引用，又可以防止对象被释放后形成野指针。虽然弱引用有很多好处，但也不能滥用。

由于 ARC 中只有强引用才能改变对象的引用计数，保持住对象，因此，如果想保持住某对象，至少要为其赋值一个强引用类型的变量。下面这行代码是一个极端的例子，因为赋值了一个弱引用，所以生成的对象会被立刻释放掉。

```
    __weak People *w = [[People alloc] initWithName:"Lucy"];
```

5.5.5 对象之间引用关系的基本原则

面向对象的编程中，一个对象的实例变量引用着另外的对象，对象之间是通过引用连接在一起的。如果把这些关系画出来，就可能得到一张图（关于图的具体概念请参考计算机图论），因此，我们就把这些对象（和它们之间的联系）称为**对象图**（object graph）。在对象图中，一个对象可能被多

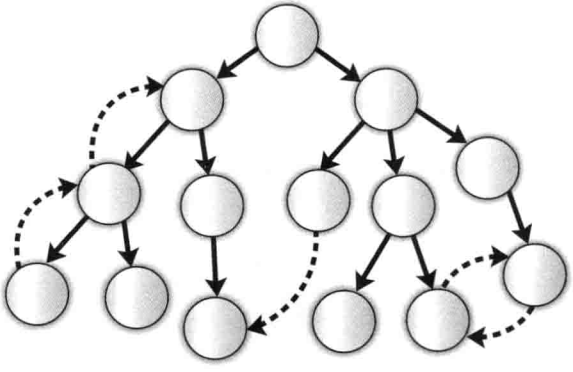
个地方引用，对象之间的引用关系可能存在环路。

对象图中的环路就是循环引用产生的原因。使用 ARC 的时候**应该尽量保证对象之间的关系呈树形结构**，避免一个对象同时被两处引用（环形成的必要条件）。这样的话，当一个对象被释放的时候，这个对象引用的对象也会被自然地释放掉，如图 5-8 所示。

在使用反向指针指向父节点，或者两个对象之间互相引用、进行跨越子树间的引用等时，容易产生循环引用。这种情况下，可以使用弱引用，或者通过手动给一方赋值为 nil 来打破循环引用关系。

如果能留意到以上这些产生循环引用的原因，并在设计阶段加以注意的话，就可以最大程度地减少释放对象时的各种问题。不光是无 GUI 的类之间有这样的问题，GUI 的各种类之间也有同样的问题。更多详细内容请参考第 16、17 章。

图 5-8 树形结构的对象图



5.6 ARC 编程时其他一些注意事项

5.6.1 可以像通常的指针一样使用的对象

使用 ARC 的时候，如果既不想保持赋值的对象，也不想赋值的对象在释放后被自动设为 nil，可以使用生命周期修饰符 __unsafe_unretained。__unsafe_unretained 所修饰的变量称为非 nil 化的弱指针，也就是说，如果所指向的内存区域被释放了，这个指针就是一个野指针了。

下面让我们来看一个 __unsafe_unretained 的例子。

```
People *s, *a;
__unsafe_unretained People *u;
s = [[People alloc] initWithName:"Shelly"];
u = s;    // s 被赋值给 u，但所有权不会加 1
a = u;    // 变量 a 是强引用类型，所以赋值操作之后所有权会加 1
u = nil; // 所有权不发生变更
```

这里首先声明了强引用类型的变量 s 和 a，以及一个 __unsafe_unretained 类型的变量 u。然后，生成一个实例对象并赋值为 s，因为 u 是弱引用类型的变量，所以当 u=s 时，s 的所有权不变；接着，把 u 赋值给 a，因为 a 是强引用类型的变量，所以赋值之后 a 的所有权加 1；最后，把 nil 赋值给 u，因为 u 是弱引用类型的变量，所以 u 的所有权也不会发生变化。另外，因为 u 是弱引用类型，不会保持对象，所以不能把生成的对象直接赋值给 u，否则生成的对象就会被立刻释放掉。

和 C 语言中的指针一样，在没有被初始化的情况下，__unsafe_unretained 类型的变量的初始值是不确定的，释放之后变量的指向也是不确定的。ARC 中不会管理（保持 / 释放）这种类型的变量，所以就和“__unsafe_unretained”字面上的意思一样，这种类型的变量是不安全的，它有可能会变成野指针。

还有一点要注意的是，手动内存管理时也有可能在赋值时没对变量进行`retain`操作，这就相当于在 ARC 中用 _weak、__unsafe_unretained 修饰了变量的行为。因此，将这种代码迁移到 ARC 环境的时候一定要小心，需要重新考虑所有权方面的问题。

有一些类的实例不能使用自动化 nil 的弱引用[①]。如果属性变量是这些类的实例，可以使用 __unsafe_unretained 来替代 __weak，但要注意这样的变量是危险的，因为当其指向的对象被释放的时候，指针不会被自动置为空，它会变成野指针。

在使用 ARC 的程序中，id 类型和 void* 类型之间不能进行转型。就算加了 __unsafe_unretained 修饰符，转型操作在编译时也会报错。这是因为，iOS 世界中主要有两种对象：Objective-C 对象和 Core Foundation 对象。其中，Core Foundation 类型的对象不在 ARC 的管理范畴内。因此，当转换这两种类型（一种有 ARC 管理，一种没有 ARC 管理）时，就需要告诉编译器怎样处理对象的所有权。为了解决这一问题，可以使用 __bridge 修饰符来实现 id 类型与 void* 类型的相互转换（更多详细内容请参考附录 B）。

5.6.2 setter 方法的注意事项

ARC 有效的情况下，使用 setter 方法时也有一些需要注意的地方，如代码清单 5-7 中的`setFriend:`方法。

如前所述，ARC 有效的情况下，变量默认为 __strong 类型。用 __weak 修饰的变量不会进行保持操作，其指向的对象被释放后，变量会自动变为 nil。

当访问上述用 __unsafe_unretained 修饰的变量时，以及在手动内存管理模式下使用可能被编译的模块时，可能会发生问题。setter 方法无法仅从接口的定义来判断是否该对传入的对象进行保持操作。而如果不对对象执行保持操作，就有可能出现野指针的情况。15.2 节中介绍的 delegate 就是一个例子，其中绝大多数都没有进行保持操作。

有一种减少野指针出现的方法是，当不再使用传入的对象时，将其赋值为 nil。典型的做法就是

① Lion ReleaseNote 中的 NSWindows、NSTextView、NSFont、NSImage 等类的实例不能使用自动 nil 化的弱引用。以上这些类的实例未来是否支持自动 nil 化的弱引用，以及当前支持自动 nil 化弱引用的类以后还能不能继续支持这种功能等，目前还没有明确的答案。

在dealloc方法中进行如下处理。

```
[someone setFriend: nil];
[controller setDelegate: nil];
```

5.6.3 通过函数的参数返回结果对象

当一个函数或方法有多个返回值时，我们可以通过函数或方法的参数传入一个指针，将返回值写入指针所指向的空间。C 语言中把这种方法叫作**按引用传递**（pass-by-reference）。Objective-C 的 ARC 中也有类似的方法，但采用了和 C 语言不同的实现方式，叫作**写回传**（pass-by-writeback）。

写回传经常被用于当一个方法在处理过程中出现错误时，通过指向 NSError 的二重指针返回错误的种类和原因（关于错误处理的详细内容，请参考 18.5 节）。下面这个声明是 NSString 的一个初始化函数，它会从指定的文件读入内容来完成 NSString 的初始化。如果初始化失败，则通过 error 返回错误的种类和原因。error 是二重指针类型，*error 指向的是 NSError* 类型的变量。

```
- (id)initWithContentsOfFile:(NSString *)path
        encoding:(NSStringEncoding)enc
        error:(NSError **)error
```

ARC 的编译器会自动为函数的二重指针变量加上 __autoreleasing 修饰符。在 ARC 有效的情况下，上面声明的函数可被编译为

```
- (id)initWithContentsOfFile:(NSString *)path
        encoding:(NSStringEncoding)enc
        error:(__autoreleasing NSError **)error
```

__autoreleasing 的根本目的是获得一个延迟释放的对象。比如，假设你想传递一个未初始化的对象的引用（二重指针的形式）到一个方法中，并在此方法中实例化此对象，而且希望方法返回时这个对象会被加入到自动释放池中，那么你就应该使用 __autoreleasing 关键字。调用 initWithContentsOfFile 时的代码如下所示。

```
NSError *error = nil;
NSString *string = [[NSString alloc] initWithContentsOfFile:@"/path/to/file.txt"
encoding:NSUTF8StringEncoding error:&error];
```

编译器会把这段代码转为以下代码。

```
NSError __strong * error = nil;
NSError __autoreleasing * tmpError = error;
NSString *string = [[NSString alloc] initWithContentsOfFile:@"/path/to/file.txt"
encoding:NSUTF8StringEncoding error:&tmpError];
error = tmpError;
```

编译器生成了一个临时变量 tmpError，执行完函数之后又将临时变量赋值给了 error。error 被加入到了自动释放池中，会一直存在到自动释放池释放为止。

只可以把 nil 或临时变量的指针用于写回传，不可以把静态变量的指针、数组首地址的指针或内部变量的指针用于写回传。

将二重指针用于方法的参数时，可以给二重指针加上 out 修饰符。例如，类 NSURL（见 9.7 节）中有这样一个方法。

```
-(BOOL)getResourceValue:(out id *)value
                 forKey:(NSString *)key
                  error:(out NSError **)error;
```

out 修饰符原本是被用于提高调用分布式对象的效率的，详情请参考 19.5 节。使用写回传的情况下，方法调用和方法返回的时候都会发生值传递。而通过使用 out 修饰符，就可以使函数只在返回的时候发生值传递，从而省略调用函数时的值传递。

5.6.4 C 语言数组保存 Objective-C 对象

ARC 有效的程序中可以用 C 语言数组保存 Objective-C 的对象。让我们看一下代码清单 5-8 中的这段程序，其中类 People 的定义请参考代码清单 5-7。

代码清单 5-8 保存 Objective-C 对象的 C 数组（array.m）

```
int main(void)
{
    People *a[4];
    static const char *const names[] = {
        "Laura", "Donna", "James", "Audrey" };
    @autoreleasepool {
        for (int i = 0; i < 4; i++)
            a[i] = [[People alloc] initWithName:names[i]];
        [People makeFriends: &a[0]]; //a[0] 和 a[1] 互为好朋友
        [People makeFriends: &a[2]]; //a[2] 和 a[3] 互为好朋友
        a[0] = nil; // // 释放掉 a[0]
    }
    [People printFriends:a number:4];
    return 0;

}
```

这里给 People 类追加了两个类方法（List5-9）。

代码清单 5-9 为 People 类追加的类方法

```
+ (void)makeFriends:(People *__strong [])p {
    [p[0] setFriend: p[1]];
    [p[1] setFriend: p[0]];  // 因为 friend 是弱引用类型，所以就算互相引用也不会产生循环引用的问题
}

+ (void)printFriends:(People *const [])p number:(int)n {
    for (int i = 0; i < n; i++)
        printf("%d: %s\n", i, [p[i] nameOfFriend]);
}
```

请注意这两个类方法中分别对参数 p 添加了 __strong 和 const 修饰符。

如前所述，ARC 中会自动对函数参数中的二重指针添加 __autoreleasing 修饰符。而通过为函数的参数增加其他修饰符，可以阻止编译器自动添加 __autoreleasing 修饰符。makeFriends 函数中的 p 就是一个二重指针（p 是一个数组的地址，数组中每个元素都是指向类 People 的指针），但 p 的目的并不是写回传，所以在定义时为 p 显式增加了 __strong 修饰符。另一方面，因为 printfFriends 函数中只输出数组的内容，并不会更改数组的内容，所以为 p 增加了 const 修饰符。

声明函数 printFriends 的参数时必须按照 Peole *const[]（一个指向指针常量的数组）这种形式，否则编译器就会提示警告。而声明函数makeFriends:时，参数则可以使用下面任何一种声明方法。

```
(People __strong **)  或者  (People __strong *[])
(People *__strong *)  或者  (People *__strong [])
(__strong People **)  或者  (__strong People *[])
```

程序的输出如下所示。

```
0: (null)
1: none
2: Audrey
3: James
```

也可以使用动态分配内存的方式为数组分配内存。代码清单 5-10 中就使用了动态内存分配的方式来为数组分配内存，但有几点要注意的地方。

代码清单 5-10 动态分配内存的例子

```
int main(void)
{
    People *__strong *a; /* 必须添加 __strong 修饰符，__strong 修饰符的位置也可以在最前面 */
    static const char *const names[] = {
        "Laura", "Donna", "James", "Audrey" };
    a = (People *__strong *)calloc(sizeof(People *), 4);
    /* 因为生成的对象的所有成员变量都要被初始化为 nil，所以使用了 calloc()。动态分配完内存后，自动将该内
       存空间初始化为零 */
    /* 赋值的时候需要进行类型转换 */
    @autoreleasepool {
        /* 省略 */
    }
```

```
    [People printFriends:a number:4];
    for (int i = 0; i < 4; i++)
        a[i] = nil; /* 为了释放空间，需要将指针赋值为 nil*/
    free(a);
    return 0;
}
```

首先，必须显式给变量 a 加上 __strong 修饰符，以防止编译器自动给变量加上 __autoreleasing 修饰符，详细分析请参考上面。其次，ARC 要求变量初值必须为 nil，所以应使用 calloc 来分配内存。另外，为了使不再使用的内存不会被突然释放掉，需要将数组中的每个指针对象赋值为 nil，以彻底释放这些对象。

最后，ARC 中不可以使用 memset()、bzero()、memcpy() 等操作内存的函数，因为 ARC 会监视这些函数的行为，所以使用了这些函数后就有可能造成内存段错误。

5.6.5 ARC 对结构体的一些限制

ARC 有效的情况下，不可以在 C 语言的结构体（或共用体）中定义 Objective-C 的对象。原因是编译器不能自动释放结构体（或共用体）内部的 Objective-C 对象。例如，下面这种定义将出现编译错误“error: ARC forbids Objective-C objs in structs or unions”。

```
struct Element {
    id       person;    // 不能定义 id 类型的对象
    NSString *address;  // 也不能定义 NSString 类型的对象
    int      age;
}
```

一种比较常见的解决方法就是使用 Objective-C 中的类来代替结构体（共用体）。如果因为效率或其他原因无论如何都要利用结构体的话，可以使用 __unsafe_unretained 修饰符来修饰结构体中的 Objective-C 变量。这样一来，编译器就不会管理这个变量的内存，所以需要完全手动地管理内存（引用计数也不可用）。

```
struct Element {
    __unsafe_unretained id       person;
    __unsafe_unretained NSString *address;
    int      age;
}
```

我们经常会用准备好的结构体数组来初始化程序。NSString 类型的常量字符串可以直接放入 C 语言的静态数组中，下面的例子是 ARC 有效的情况下结构体数组的声明方法。@ 开头的字符串表示的是 NSString 类型的常量字符串。如果数组仅仅是为了初始化用，而不需要修改数组中的内容的话，可以给结构体中的对象加上 const__unsafe__unretained 修饰符，来回避内存管理方面的各种问题。

```
static struct { /* 初始化用的结构体，包括名字和年龄 */
    const __unsafe_unretained NSString *name;
    int    age;
} initialData[] = {
    { @"Laura", 17 }, { @"Donna", 17 }, ...
};
```

5.6.6 提示编译器进行特别处理

未使用 ARC 的时候你可能没有按照 ARC 中的命名规则来为方法起名，而这种情况下如果因为某些原因没法修改这些方法的名字，那么将这些代码迁移到 ARC 环境中就会有问题。这时可以通过给方法加上事先定义好的一些宏来告诉编译器应该如何对这个方法的返回值进行内存管理[①]。

宏的声明需要放置到方法末尾，让我们来看几个常用的宏。

NS_RETURNS_RETAINED

指明这个方法和init或copy开头的方法一样，由调用端负责释放返回的对象。

NS_RETURNS_NOT_RETAINED

指明这个方法不属于内存管理方面的方法，调用端无需释放返回的对象。

例如 newMoon 这个方法，如果从命名规则的角度来考虑，它属于 new 方法族。但如果希望这个方法不改变返回值的所有权，就可以为方法加上 NS_RETURNS_NOT_RETAINED，声明如下。

```
+ (FishingDate *)newMoon NS_RETURNS_NOT_RETAINED;
```

但要注意的是，大家还是应该尽可能地修改方法名以使其符合命名规则，实在无法更改方法名的时候才能使用这种方法。

编译时，这些宏会被替换为**注释**（annotation）。宏被定义在 NSObjCRuntime.h 中。除了上面提到的这两个宏之外，文件 NSObjCRuntime.h 中还定义了很多其他的宏，这些都不属于语言的一部分，是编译器特有的扩展功能。

可以使用条件编译来区分 ARC 是否有效，编译器可以按条件编译程序中的不同部分，生成不同的目标文件。区分 ARC 是否有效的编译条件是 #if（__has_feature（objc_arc）），结果为真时表明 ARC 有效，否则就表示 ARC 无效。

① 其实如果不是为了应对 ARC 和手动引用计数内存管理混合编程的需求，根本就不需要通过方法名来区分是否进行内存管理，完全可以通过简单的方式来实现 ARC。

第6章

垃圾回收

垃圾回收（garbage collection）指的是在程序运行过程中不定时地检查是否有不再使用的对象，如果存在就释放它们占用的内存空间。为了更好地利用垃圾回收，需要了解垃圾回收的原理和内部机制，本章中将对此进行说明。

目前，垃圾回收只能用在 Mac OS X 的软件中，Mac OS X 10.7 和 iOS 5 以上的系统推荐使用上一章中介绍的 ARC 来管理内存。

6.1 垃圾回收的概要

垃圾回收是 MacOS X 10.5 Leopard 之后开始引入的内存管理方法，iOS 目前还不支持垃圾回收机制。Mac OS X 10.7 和 iOS 5 以上的系统强烈建议使用上一章中讲到的 ARC 来管理内存，垃圾回收仅仅是作为一种备选方案而存在的

本章将介绍 Mac OS X 中垃圾回收的概要和使用方法，只想使用 ARC 的读者只需要了解一下垃圾回收是怎么工作的就足够了。

6.1.1 查找不再使用的对象

垃圾回收指的是在程序运行过程中，检查是否有不再使用的对象，并自动释放它们所占用的内存，通常被简称为 GC。内存的检查和回收都是由**垃圾收集器**（garbage collector）完成的。

Mac OS X 中垃圾回收的目标是不再使用的实例对象。C 风格的变量、C 风格的结构体以及 C 风格申请的内存都不属于垃圾回收的范围（后文将详细介绍哪些内存属于可回收的范围）。id 类型、以类名作为类型（例如：NSobject * 类型）的实例对象是垃圾回收的目标。

在 Objective-C 2.0 中，垃圾回收首先进行的工作就是识别不允许被回收的对象。首先，全局变量和静态变量引用的对象不允许被回收。另外，栈内临时变量引用的对象也不允许被回收。这些对象称为**根集合**（root set）。如果一个对象的实例变量引用了不允许被回收的对象或根集合中的对象，那么这个对象也不允许被回收。

也就是说，**通过全局变量、静态变量或者栈内变量的引用而查找到的对象都不可以被回收**，如图 6-1 所示。图 6-1 中的任何一个对象都可以从根集合出发通过引用关系找到，所以都不可以被回收。Cocoa 中的 GUI 应用一定会包含类 NSApplication 的实例，它是一个名为 NSApp 的全局变量（详情请参考第 15 章）。这个全局变量属于根集合，不能够被回收，所以所有引用了 NSApp 的窗口和菜单对象都不能够被回收。

反之，只要是从根集合出发无法到达的对象都属于垃圾回收的目标。如图 6-2 所示，可被垃圾回收的对象用深色的椭圆形表示。虽然这些对象也都被别的对象引用着，但只要从根集合出发找不到它们，它们就属于垃圾回收的目标。

图 6-1 不允许被回收的对象

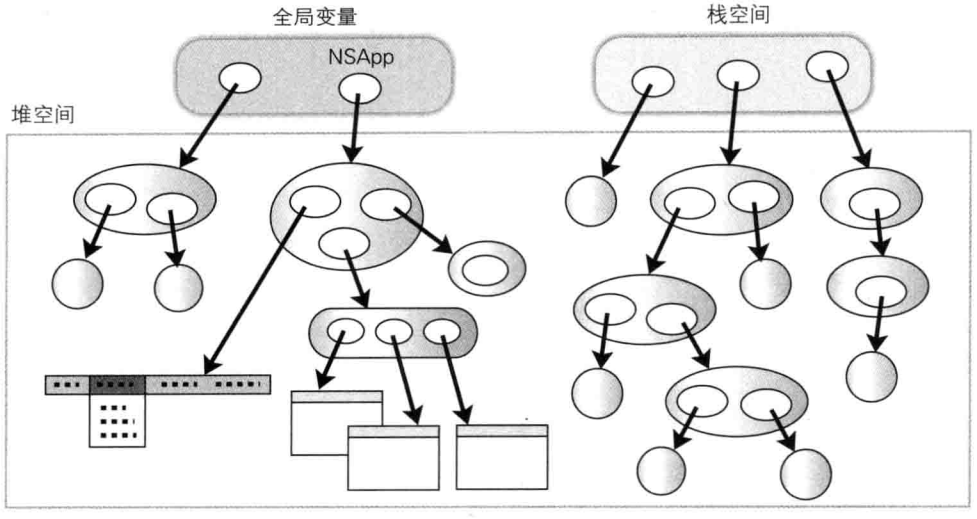
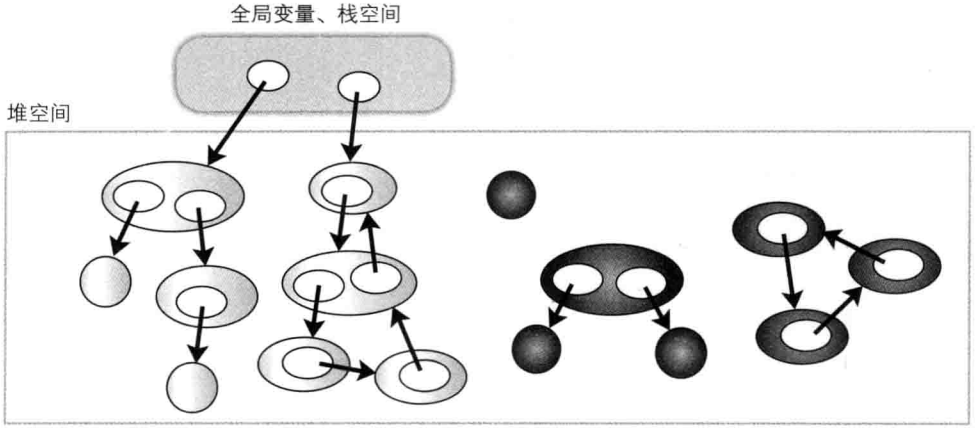


图 6-2 允许被回收的对象



6.1.2 编程时的注意事项

让我们看看图 6-3 中的例子，假设程序执行到了methodB中注释的位置时垃圾回收器启动了，这时都有哪个对象会被回收呢？首先，因为 objB 是一个栈内对象，所以 objB 所指向的对象不会被回收。全局变量 sharedObj 最开始指向了类 Gamma 的实例，之后又指向了类 Zeta 的实例。因为全局变量指向的对象不会被回收，所以 sharedObj 指向的类 Zeta 的实例不会被回收，但类 Gamma 的实例因为没被任何对象引用，所以会被回收。另外，因为方法 methodA 的执行已经完毕，所以方法methodA中类 Alpha 的实例也会被回收。

▶ 图 6-3 被回收的对象

```
id sharedObj = nil;

- (void)methodA {
    id objA = [[Alpha alloc] init];
    [objA doSomething];
}

- (void)methodB {
    id objB = [[Beta alloc] init];
    sharedObj = [[Gamma alloc] init];
    [self methodA];
    sharedObj = [[Zeta alloc] init];
    /*执行到这里会发生什么? */
}
```

手动管理内存时，类的 setter 方法的写法如下所示（详情请参考 5.4 节中的内容）。

```
- (void)setHelper:(id)obj {
    [obj retain];              // 必须首先保持参数
    [helper release];          // 不要忘记释放原对象
    helper = obj;
}
```

垃圾回收有效的情况下，setter 方法只需要一个赋值语句就可以了。这时就不需要再对 obj 和 helper 执行保持和释放操作，垃圾回收机制会自动判断变量是否可以被回收。

```
- (void)setHelper:(id)obj {
    helper = obj;
}
```

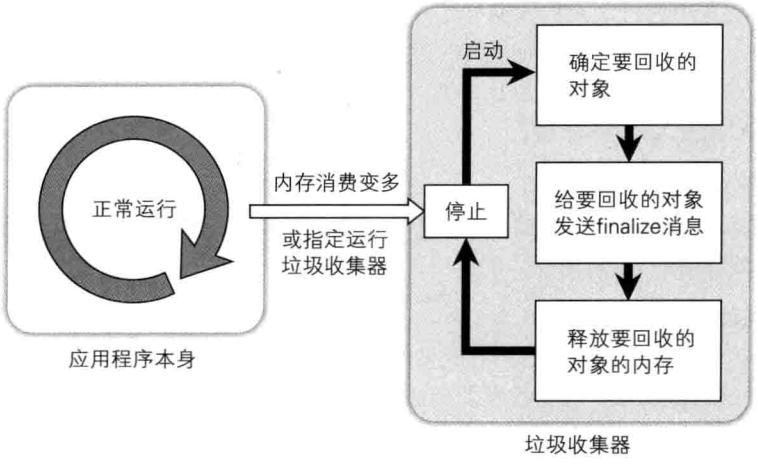
手动内存管理时有可能会出现多个对象循环引用的情况，而使用垃圾回收管理内存时，就算有循环引用也不会影响内存的释放。互相引用的对象集合也是如此，只要不能被根集合中的变量引用到，就属于垃圾回收的目标。

虽然使用垃圾回收的情况下不需要手动释放每个对象，但如果不再使用的对象属于根集合，这个对象就不会被回收。对于不再使用的对象，我们可以通过为其赋值 nil 来明确表示其已不再被使用，据此通知垃圾收集器回收该对象。这种做法不仅有助于垃圾回收，也是一种良好的编程风格。

6.1.3 垃圾收集器

垃圾收集器的运行是自适应的，既不以固定的频率运行，也不能通过程序调用运行。当程序运行过程中分配的内存超过一定量时，垃圾收集器就会被自动触发运行。垃圾收集器通常作为一个单独的线程运行，并对已经不再被使用的对象进行回收。整个回收过程完全是根据程序占用内存的多少自动完成的（如图 6-4）。

图 6-4 垃圾收集器的运行



一个程序只有一个垃圾收集器，通常在 mainthread 中运行。每个线程都有自己的栈空间，这些栈空间中的变量不属于垃圾回收的范畴。垃圾收集器运行的时候有可能会暂停其他线程来完成垃圾收集，但不会暂停所有线程的运行。

虽然垃圾收集器会根据内存的情况自动运行，但我们也可以通过给类 NSGarbageCollector 发送collectIfNeeded消息来启动垃圾收集器。特别是当我们使用了大量的临时对象时，可以通过这种方法来释放内存。通过主动启动垃圾回收，不仅可以控制内存的消费，又可以避免垃圾回收突然启动时程序反应变慢的问题。

垃圾收集器并不是一找到不再使用的对象就立刻释放它，而是会首先找到所有不再使用的对象，然后再给这些对象发送finalize消息。等所有对象响应了finalize之后，才释放这些对象。垃圾收集环境下释放对象时，对象的dealloc方法不会被执行。

6.1.4 finalize 方法的定义

对象中定义了finalize方法的情况下，在对象被回收释放之前，finalize方法会被执行。

finalize方法被定义在 NSObject 中，典型的定义如下所示。除了子类中调用父类的 finalize 之外，程序中不允许直接调用finalize方法。

```
- (void)finalize
{
    ... // 释放之前的处理
    [super finalize];
}
```

引用计数的内存管理方式下，在对象被释放的时候，dealloc方法会被执行。finalize可能会被看作是dealloc的替代，但实际上并不一样。原则上并**不推荐轻易定义finalize**，只有碰到不通

过finalize就无法实现的功能时才建议定义finalize。finalize方法会作为垃圾回收的一部分被执行。垃圾回收在一个独立的线程中进行，同软件本身是并行的关系，如果finalize处理过于复杂的话就会影响软件本身的执行速度。另外，要被释放的对象的finalize方法的执行是没有先后顺序的，不要依赖finalize方法的执行顺序来做任何事情。

实现finalize方法的时候要注意下面这些事项。

（1）finalize 方法内不用担心其他对象的释放

和dealloc不一样，**不用考虑对象所有权的问题**。一个对象是否被释放是在处理中（所有权的数值）决定的，垃圾收集器会释放不再使用的对象。

（2）要注意 finalize 方法中对象的赋值

finalize方法中，除了自动变量之外，对其他所有变量的赋值操作都要小心。因为如果被垃圾收集器判断为无用的对象在finalize方法中被再次赋值的话，该对象就有可能不允许被回收，这个过程叫作**复活**（resurrection）。一部分编程语言允许复活，Objective-C 2.0 中则不允许这种操作，否则会发生执行时错误。因为finalize中无法知道赋值的对象需不需要被回收，所以冒然进行复活操作是不安全的。

（3）finalize 的执行顺序和对象的释放顺序都是无法确定的

假设有两个对象 A 和 B，它们都是垃圾回收的目标。这时，给 A 和 B 发送finalize消息的顺序是无法确定的，而且同 A 引用 B 与否无关。对象 A 在finalize中给 B 发送消息的时候，对象 B 的finalize有可能已经执行完了，也有可能还没有执行，这些都是不确定的。

（4）finalize 不是线程安全的

线程安全是指某个函数或函数库在多线程环境中被调用时，也能够正确地处理各个线程的局部变量，使程序功能正常完成。特别是在处理多个实例共有的对象或资源的时候，要特别注意线程安全问题。

对象的“善后”处理应该在这个对象不再使用的时候进行，而不要在finalize方法中进行。特别是文件之类的资源对象，不要在finalize方法中进行关闭处理。进行善后处理之后，可以将不再使用的对象赋值为别的对象或者 nil，以取消对该对象的引用，并告知垃圾收集器该对象可以被清理了。

反之，在多个地方释放对象时，就不得不用到finalize方法。

6.1.5 编译时的设定

垃圾回收只能在 Mac OS X 10.5（Leopard）以后的版本中使用，在此之前的 Mac OS 系统和 iOS 中目前还不支持垃圾回收机制。

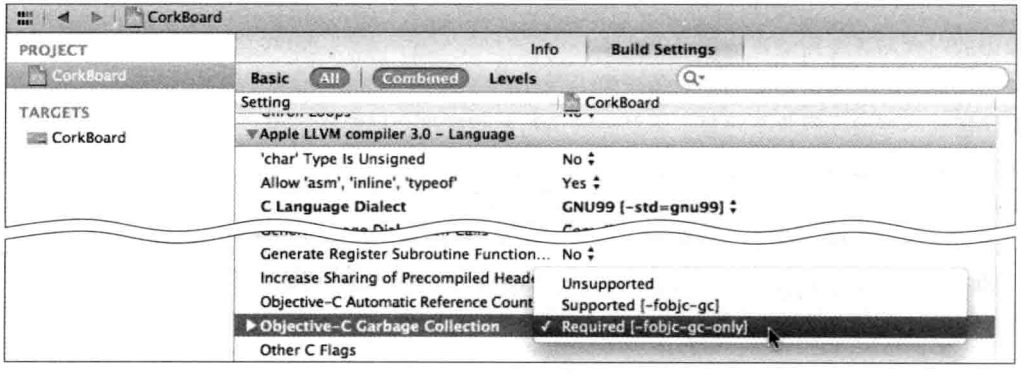
使用垃圾回收时需要设置编译选项。命令行下的编译选项如表 6-1 所示，这些选项无论对 cc 还是对 clang 都是有效的。

表 6-1 垃圾回收的编译选项

选项	含义	动作
无	不支持 gc	基于引用计数的内存管理，不使用垃圾回收
-fobjc-gc	支持 gc	同时使用垃圾回收和引用计数进行内存管理
-fobjc-gc-only	只使用 gc	只使用垃圾回收（推荐）

如果要在新工程中使用垃圾回收，可以指定选项 -fobjc-gc-only。-fobjc-gc 是混用垃圾回收和引用计数的意思。当程序根据第三方类库或插件的情况而不得不混用上述两种内存管理方式时，建议使用该选项。

图 6-5 Xcode 4.2 的画面



使用 Xcode 编译之前也需要同样的设定。图 6-5 展示了 Xcode 4.2 中设定垃圾回收的编译选项的过程。

6.1.6 引用计数管理方式中方法的处理

启用垃圾回收后，以下这些和引用计数有关的方法都将无效。

```
retain
release
autorelease
dealloc
retainCount
```

把使用引用计数管理内存的代码迁移为以垃圾回收管理内存时，就算调用了以上这些方法也不会有任何效果。对象被释放时dealloc方法也不会被执行。

伴随着垃圾回收功能的引入，类 NSAutoreleasePool 增加了新的方法。

- (void) **drain**

引用计数的内存管理中，drain方法和release方法具备同样的功能，即释放自动释放池。

垃圾回收的内存管理中，drain方法表示申请进行垃圾回收，和后面提到的类NSGarbageCollecotr的collectIfNeeded方法具备同样的功能。

以上这些方法在引用计数内存管理和垃圾回收模式下的动作各不相同，无论如使用哪种方式编译，都能生成想要的代码。但是要在具有一定规模的代码中毫无差错地混用这两种模式是非常困难的，不推荐这样做。

6.1.7 使用垃圾回收编程小结

使用垃圾回收编程的时候，不需要什么高深的技巧，只需要注意对象什么时候不再使用就足够了。这里我们来简单地总结一下。

- 编译的时候需要使用编译选项 -fobjc-gc-only
- 使用局部变量给方法中的局部对象赋值
- 如果暂时不想垃圾回收某对象，需要保证该对象能够被全局变量引用
- 原则上垃圾回收只负责回收 id 类型或类类型的变量
- 不用考虑所有权的问题，也不用考虑实例变量的问题
- 释放对象时的一些“善后”操作可以写在 finalize 方法中，但除了不写在 finalize 中就无法实现的功能，要尽量减少通过 finalize 完成的操作
- 引用计数相关的方法和 dealloc 方法不会被执行

Cocoa 的应用程序中可以把 GUI 控件的连接和配置都压缩到 nib 文件中，并在执行的过程中加载。通常这么做是没有问题的，但在处理和 nib 文件关联的内存管理时有需要注意的地方，请参考 16.2 节。

6.2 垃圾回收的详细功能

6.2.1 分代垃圾回收

启用垃圾回收之后，会在变量赋值、改写变量时建立**写屏障**（write barrier）。Objective-C 2.0 利用修改变量时的信息采用**分代垃圾回收**（generational GC）的方式来进行内存管理。

在一般的程序处理中，绝大部分的变量都是被临时生成使用的，也就是说，当进行完相应的处理之后，这些变量就没用了。针对这种情况，分代垃圾回收并不会每次都对整个堆空间进行遍历，而是以新生成的对象为中心，以尽可能快速地收集那些生命周期短的对象。通过采用这种分代的方法，既可以减轻垃圾回收的处理负荷，又能有效释放空间。但也要小心这种方法会积攒太多的陈旧对象。

Objective-C 2.0 的垃圾回收不会把对象向别的内存区域移动。因为 Objective-C 中指向对象的指针也可以被作为 C 语言的指针来使用，所以对象在内存中的位置不会发生变化。

6.2.2 弱引用

能够通过引用从根集合连接到的对象都不属于垃圾回收的目标。而 Objective-C 的垃圾回收中引入了弱引用的概念。这里的弱引用是指，即使一个对象被引用了，也不影响该对象被回收。同 ARC 一样，与弱引用相对的是**强引用**。

因此，准确地说，垃圾回收规则应该是，**从根集合只通过强引用连接到的对象都不属于垃圾回收的目标**。无法连接到根集合的对象或不通过弱引用就连接不到根集合的对象都是垃圾回收的目标。

图 6-6 弱引用和回收的对象

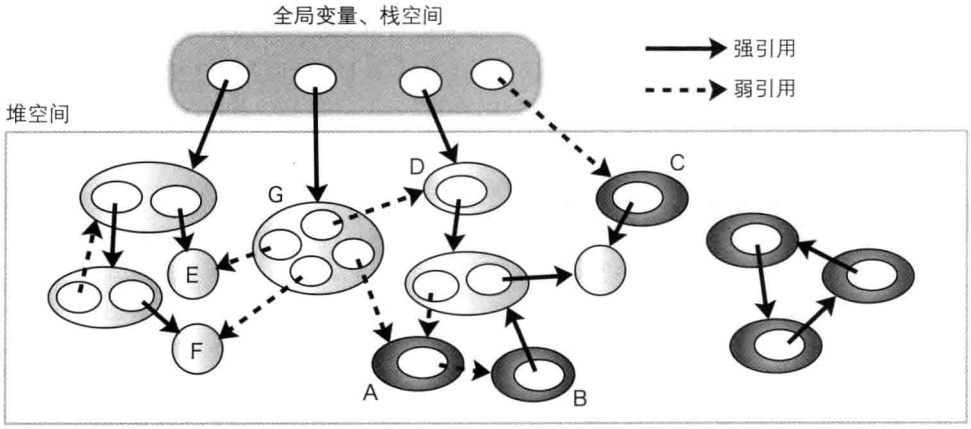


图 6-6 中，实线箭头代表强引用，虚线箭头代表弱引用。

对象 A、B 和 C 无法只通过强引用连接到根集合，所以是垃圾回收的目标。

通过在声明的时候加上 __weak 修饰符，可以把一个对象声明为弱引用类型。和 ARC 不一样的是，栈内的临时变量不能声明为弱引用类型。同样，也可以使用 __strong 修饰符来声明一个变量是强引用类型。__weak 和 __strong 修饰符不仅可以声明对象，还可以声明指针。除了这两个修饰符之外，__autoreleasing 和 __unsafe_unretained 等 ARC 的其他生命期修饰符在垃圾回收中都不可用。

6.2.3 自动 nil 化

垃圾回收中弱引用的变量也会自动 nil 化，当其指向的对象被垃圾收集器释放的时候，就会被自动赋值为 nil。强引用类型的对象因为不是垃圾回收的目标，所以不会被自动 nil 化。图 6-6 中，对象 G 有四个弱引用类型的实例变量，当这四个变量被回收、释放之后，实例变量就会被自动设为 nil。

如果被释放的对象中定义了`finalize`方法，那么在对象被置为 nil 之前，`finalize`方法会被执

行。图 6-6 中的对象 A 和 B 同时被释放的情况下，当对象 A 的finalize执行的时候，引用对象 B 的实例变量就会被设为 nil。

ARC 中是为了解决对象之间循环引用的问题才引入弱引用这个概念的。如图 6-6 所示，垃圾回收中就算有循环引用也不会影响对象的回收，那么为什么要引入弱引用这个概念呢？

使用强引用的情况下，只要该引用存在，被引用的对象就不能被回收。而弱引用则没有这个问题，它的作用是引用一个对象，但是并不阻止该对象被回收。在垃圾收集器运行的时候，如果一个对象的所有引用都是弱引用，该对象也会被回收。弱引用的作用在于解决强引用所带来的对象之间在存活时间上的耦合关系。

通常情况下，编程的时候不使用弱引用也不会有什么问题。弱引用的一个使用场合就是给多个对象同时发送消息（第 15 章中有更详细的说明）。在图 6-6 中，假设 G 想给 A、D、E、F 同时发送消息，对 G 来说，接收消息的对象是否存在都没有关系，如果存在就发送消息，不存在就不发送消息。对象 A 被释放了之后，指向 A 的引用变量就会被置为 nil，之后也就不会再被发送消息了。整个过程就好像商店给顾客发送广告，如果对方的地址不存在了，就从地址本中将其抹掉（赋值为 nil）即可。

弱引用也可以被用来作为使用后就删除的临时缓存，但这都要基于垃圾回收才能够完成，所以请尽量不用使用这种依赖于垃圾回收的运行频率和时机的编程方法。

6.2.4 通过垃圾回收回收动态分配的内存

目前我们只介绍了使用垃圾回收回收对象，实际上通过函数 NSAllocateCollectable 动态分配的内存也可以被垃圾回收回收。NSAllocateCollectable 被定义在 Foundation/NSZone.h 中。

```
void *__strong NSAllocateCollectable(
        NSUInteger size, NSUInteger options);
```

NSUInteger 是无符号整数类型，size 是以 byte 为单位的内存大小。第二个参数 options 一定要指定为常量 NSScannedOption。返回值是分配好的内存空间的首地址，如果内存分配失败就返回 NULL。

通过这个函数分配的内存和实例变量一样，如果无法通过根集合到达，这块内存就可以被回收。函数返回的指针类型用了 __strong 来修饰，也就是说不只限于对象类型，指针也可以用 __strong 和 __weak 来修饰。

根据指针声明的不同，垃圾回收的规则也不同，让我们来看看下面这 3 个例子。

```
static void *p = NSAllocateCollectable(SZ, NSScannedOption);
static __weak void *w = NSAllocateCollectable(SZ, NSScannedOption);
static __strong void *s = NSAllocateCollectable(SZ, NSScannedOption);
```

变量 p 不是对象类型，当 p 所指向的内存被回收后，p 有可能会变成一个野指针。变量 w 被声明为弱指针类型，当 w 所指向的内存被回收后，w 会被自动赋值为 nil。变量 s 被声明为强指针类型，

它所指向的内存不会被回收。

表 6-2 根据指针类型的不同，使用不同的垃圾回收方法

声明类型	是否能够被回收	是否会被自动赋值为 nil
局部变量 void *	可以	方法执行结束后指针变无效
void *	可以	不会，有可能变成野指针
__weak void *	可以	会
__strong void *	不可以	—

指针变量指向栈内的临时空间时，内存空间会在方法执行结束后被自动释放，所以不需要考虑内存释放的问题。而如果指针变量指向的空间不是栈内的临时空间，则声明变量的时候就需要加上 __weak 或 __strong 修饰符。之间的差别请参考表 6-2。

通过 NSAllocateCollectable() 分配的内存因为可以通过垃圾回收被自动收回，所以可以不考虑内存释放的问题，比使用 malloc 和 free 编程更方便。但指向除对象之外的动态内存的指针变量都需要用 __strong 来修饰。结构体中的成员为指针变量的情况下也一样。另外，程序的效率有可能会降低，所以并不需要特意把原来使用 malloc/free 的代码都改为使用 NSAllocateCollectable。

6.2.5 __strong 修饰符的使用方法

下面对 __strong 修饰符的使用方法进行一下总结。

（1）修饰对象类型的变量

声明对象类型的变量时，如果不加上 __weak 修饰符，都默认是 __strong 类型。

（2）修饰指针类型的变量

被 __strong 修饰的指针所指向的内存可以被垃圾回收回收，但要等到 __strong 类型的指针不再使用这块内存之后。

（3）用于修饰函数或方法返回的指针类型

指针指向的内存属于垃圾回收的范畴，但如果被赋值给一个强引用类型的变量，就暂时不能被回收。

6.2.6 NSGarbageCollector 类

类 NSGarbageCollector 是用于对垃圾收集器进行设定的类，程序编译的时候可以指定垃圾回收的各种选项。这个类的接口被定义在 Foundation/NSGarbageCollector.h 中。

+ (id) **defaultCollector**

返回现在有效的垃圾收集器的实例，如果垃圾收集无效，则返回nil。

- (void) **collectIfNeeded**

申请进行垃圾回收。虽然垃圾回收是根据内存使用量自动运行的，但通过这个方法也可以指定希望启动垃圾回收的地方。这个方法在被调用之后会判断当前的内存使用情况，如果觉得需要进行垃圾回收就会启动垃圾收集器。典型的调用方式如下。

```
[[NSGarbageCollector defaultCollector] collectIfNeeded];
```

- (void) **collectExhaustively**

申请进行垃圾回收。希望使用深度遍历以尽可能地释放更多的内存时使用。该方法在被调用之后会判断当前的内存使用情况，并在需要时启动垃圾收集器。

- (void) **disableCollectorForPointer:** (void *) ptr

参数ptr是指向一块内存或指向实例变量的指针，调用这个函数后，这个指针所指向的内存不会被回收。也就是说指针指向的对象会变成根集合中的对象。如果想让对象变为可以被再次回收的话，需要调用下面的enableCollectorForPointer:。

- (void) **enableCollectorForPointer:** (void *) ptr

能够让通过disableCollectorForPointer:指定不允许被释放的内存或对象再次允许被释放。

- (void) **enable**
- (void) **disable**

disable方法能够让垃圾收集器暂时停止内存回收。ensable方法则能让垃圾收集器恢复工作。但enable和disable必须成对调用，即调用多少次disable，就必须调用多少次enable。这两个函数一般被用于防止多线程中同时运行垃圾回收。

当垃圾收集器重新开始工作时，垃圾收集器暂停内存回收期间的对象也会被回收掉。

- (BOOL) **isEnabled**

用于判断垃圾回收器是否有效，有效时返回真。

6.2.7 实时 API

Objective-C 提供了实时 API 来控制垃圾回收器的动作（更多关于实时 API 的内容请参考第 8 章）。API 被定义在头文件 /usr/include/objc/objc-auto.h 中，下面来介绍两个主要的。

void **objc_startCollectorThread** (void);

启动一个线程来专门运行垃圾收集器。Cocoa环境下的GUI程序通常不需要调用这个方法。如本章的例子所示，该方法只在使用Foundation框架的程序中使用。

oid ***objc_memmove_collectable**(void *dst, const void *src, size_t size);

复制垃圾回收对象的内存时使用该函数替代memcpy和memmove。

6.3 内存管理方式的比较

6.3.1 引用计数和垃圾回收

垃圾回收使程序员从纷繁复杂的内存管理之中解脱了出来，很多编程语言中也都实现了垃圾回收的功能。特别是最近的面向对象的语言中，绝大多数都把垃圾回收作为一个基本前提。

Objective-C 的一个特点是可以和 C 语言混合编程，所以，和基于引用计数的内存管理方式相比，垃圾回收有一些不优雅的实现。下面对手动引用计数内存管理、ARC 和垃圾回收进行一个对比，看看各种方法都有哪些特点。

首先让我们来看看和手动内存管理相比，ARC 和垃圾回收有哪些优点。

- 不需要再在意对象的所有权
- 可以删除程序中内存管理部分的大部分代码，使程序看起来更清爽
- 可以避免手动内存管理时的错误（内存泄漏等）
- 不需要再在意引用计数内存管理中的一些特殊用法。例如访问方法的定义和临时对象的使用等
- 可以使多线程环境下的编程更简单。例如：不用担心不同的线程之间可能出现的所有权冲突

垃圾回收的缺点如下所示。

- 垃圾收集器运行时会影响程序的速度
- 需要不停地监视内存的使用，同引用计数的方法相比，程序的速度会变慢
- 会影响程序的效率。不经常使用的内存也会被垃圾收集器不时地访问，实际上有可能会占用更多的内存
- 需要使用一些技巧来让对象被回收或不被回收
- 无法使用引用计数管理方式下的一些设计方针。例如：事先准备好一些管理文件或其他资源类的对象，在对象被释放的同时关闭资源

相比较而言，ARC 没有垃圾回收那么多缺点，但也有一些要注意的地方。

- 循环引用一旦形成就不会自己消失
- 为了防止循环引用的形成，需要注意对象间的关系。GUI 的各个类之间也有可能形成循环引用，也要注意防止循环引用的形成
- 理论上可被赋值的对象都可以被自动释放，但在处理结构体、数组、二重指针等类型的变量时有一些限制
- 需要注意 Core Foundation 的对象和 Objective-C 的对象之间的转换（详情请参考附录 B）

6.3.2 更改内存管理方式

5.4 节中提到过 Xcode 提供了一些工具可将手动内存管理的代码变为 ARC 方式的内存管理。但要注意的是，除了特别简单的程序之外，一般情况下变更代码的内存管理方式并不是一件容易的事情。

采用不同的内存管理方式来编写程序的情况下，因为对象的保持和释放方式不同，程序的整体风格和思考方法也不尽相同。例如，即使是一个简单的赋值语句，它在手动内存管理、ARC、垃圾回收中也各不相同，因此这条语句前前后后的操作也都不会相同。

而如果不得不手动迁移的话，就一定要注意以下几个方面。

- **与对象的所有权和生存期相关的处理**
- **释放对象之后的善后处理**
- **防止野指针的生成**
- **有时还需要从对象之间的相互关系来重新考虑**

6.3.3 各种内存管理方式的比较

下面让我们用一个简单的程序来测试一下手动内存管理（MRC）、ARC 和垃圾回收时程序的执行速度。

代码清单 6-2 中的测试程序循环进行了对象的随机生成和释放。测试的方法是采用一个具有 16 个元素的数组，数组中的每个元素都是一个链表的头节点。每次生成两个随机数，第一个随机数被用于指明往数组的哪个元素中添加数据，第二个随机数则起到了标识位的作用。如果生成的第二个随机数是 16 的倍数，则将释放整个链表。

三种不同的内存管理方式的代码有一些不同，代码清单 6-2 中对这些不同的地方加上了注释。阴影部分是手动引用计数内存管理方式的代码。

代码清单 6-1 用于测试垃圾回收速度的程序（speed.m）

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

#define MASS          2000                /*MASS 的值可调整 */
#define ARRAYSIZE     (1 << 6)            /* 数组的大小 */
#define ARRAYMASK     (ARRAYSIZE - 1)     /* 用于生成下标的掩码 */
#define LOOP          1500                /* 确定重复次数 */
#define ACCIDENT      0x0F                /*1/16 的释放概率 */

id buf[ARRAYSIZE];              /* 全部初始化为 nil*/

@interface Cell : NSObject {
    id next;                    /* 指向下一个链表元素 */
    char mass[MASS];            /* 确保实例占用一定的空间 /
}
```

```
+ (Cell *)cellWithNext:(id)obj;
@end

@implementation Cell

- (id)initWithNext:(id)obj {
    self = [super init];
    next = [obj retain];          /* 手动内存管理需要 retain*/
    return self;
}

+ (Cell *)cellWithNext:(id)obj {
    return [[[self alloc] initWithNext:obj] autorelease];
}  /* 手动内存管理需要 autorelease 之后再返回 */

- (void)dealloc { /* 只在手动内存管理时调用 */
    [next release];
    [super dealloc];
}

@end

int main(void)
{
    int i, j;

    srandom(12345);          /* 随机数的种子固定 */
    for (i = 0; i < LOOP; i++) {
      @autoreleasepool { /*GC 不需要自动释放池 */
        for (j = 0; j < LOOP; j++) {
            int idx = random() & ARRAYMASK;
            if (buf[idx] != nil && (random() & ACCIDENT) == 0) {
                /* 满足条件时释放对象 */
                [buf[idx] release]; /* 只在手动内存管理时调用 */
                buf[idx] = nil;
            }else {
                /* 通常会从数组生成链表 */
                id t = buf[idx];
                buf[idx] = [[Cell cellWithNext:t] retain]; /* 手动内存管理时需要 */
                [t release]; /* 只在手动内存管理时调用 */
            }
        }
      } /* GC 在这里促进垃圾回收的执行 */
    }
    return 0;
}
```

在三种内存管理方式下分别编译并执行这一程序，并统计运行时间。可以使用 Unix shell 命令 time 来测试程序的运行时间。bash 中 time 命令的输出如下所示。user 行的数字就是程序占用 CPU 的秒数。

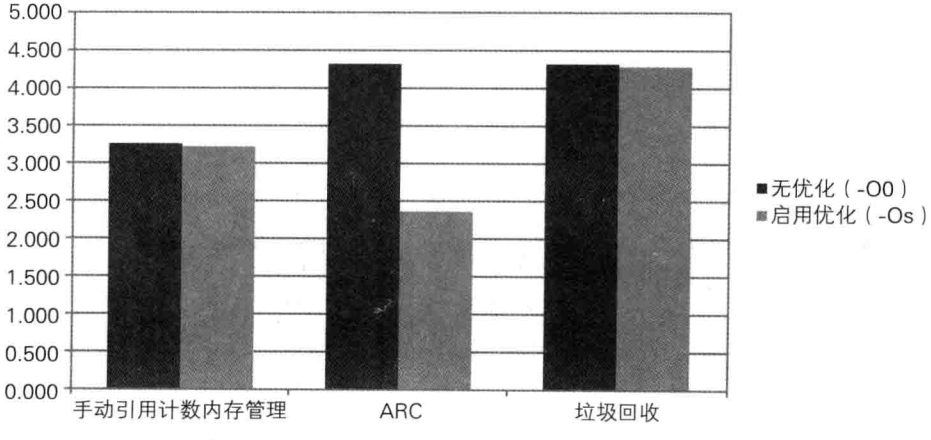
```
% time ./a.out
real    0m3.346s
user    0m3.077s
sys     0m0.239s
```

Mac 上编译的时候分别使用编译选项 -O0 和 -Os 生成了可执行程序[①]。-O0 代表没有优化，-Os 代表对生成的二进制代码进行了尺寸上的优化，同时它还打开了所有 -O2 的优化选项。将每个程序分别执行 3 次，并统计出平均运行时间，结果如表 6-3 所示。可见，使用 ARC 且指定了 -Os 编译选项时程序的运行速度最快，原因是通过优化简化了引用计数操作。而使用垃圾回收的程序的运行速度最慢。虽然无法保证任何情况下使用 ARC 的程序都是最快的，但通过这个简单的测试，我们至少也了解到了一个大概的趋势。

为了便于大家参考，我们还用 C 语言（采用 malloc 和 free 管理内存）和 Java 语言（1.6.0_26）实现了同样的程序，这两个程序的执行时间分别是 0.350 秒和 3.11 秒（Java 中使用了多线程）。

表 6-3 不同内存管理方式下程序的运行速度

内存管理方式	CPU 秒数（sec）	
	无优化（-O0）	启用优化（-Os）
手动引用计数内存管理	3.237	3.203
ARC	4.320	2.343
垃圾回收	4.29	4.283



① Mac mini(Mac OS X 10.7.1, 2.66GH Intel Core 2 Duo)。编译选项 -O0 是大写字母 O 和数字零的组合。编译选项 -Os 是大写字母 O 和小写字母 s 的组合。更多关于优化选项的说明请参考编译手册。

第7章

属性声明

Objective-C 2.0 支持使用简洁代码来调用访问方法，也允许自动生成访问方法。通过这些方便的功能，即使不手动实现访问方法，也能够操作对象的属性值。

7.1 属性是什么

7.1.1 使用属性编程

一般来说，**属性**（property）指的是一个对象的属性或特性。

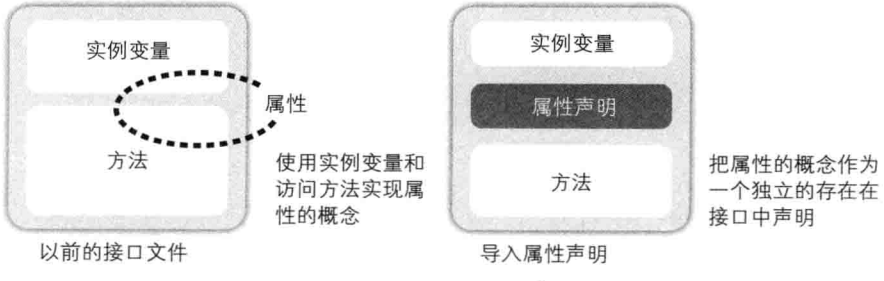
编写一个由多个对象组成的程序时，除了要给对象发送消息使其完成某个操作外，还需要查询或更新对象的状态和属性。对象的实例变量，也就是访问方法的目标一般被称为属性。例如，如果把游戏中的一个人物看作是一个对象的话，他的属性就包括名称、体力、经验、装备等。

本章中要说明的**属性声明**是一种声明变量为属性的语法，该语法同时还引入了一种更简单的访问属性的方法。

在至今为止的说明中，对象的属性值都保存在一个实例变量中，可以直接访问（需要实例变量的访问权限支持）或者通过访问方法访问。也就是说，声明了实例变量或定义了访问方法就相当于实现了属性。

本章中要介绍的属性声明指的就是**在接口文件中声明实例对象到底有哪些属性**，如图 7-1 所示。因为实现方法写在类的实现部分中，所以对外部是不可见的。从功能的角度来看，虽然和使用访问方法并没有什么不同，但可以把访问方法的声明从类的接口文件中删除。

图 7-1 属性的概念和属性声明



随着属性声明的引入，Objective-C 的编程风格也有了很大的变化。

这里将属性声明的一些规则总结如下。

自动生成访问方法

能够为指定的实例变量自动生成访问方法。既可以同时生成 getter 和 setter 方法，也可以只生成 getter 方法。除了自动生成外，也可以手动定义访问方法。

自动生成实例变量

如果不存在同名的实例变量的话，在生成访问方法的同时，也会自动生成同名的实例变量。

更简单地调用访问方法

可以通过点操作符 (.) 来调用访问方法。无论是赋值用的 setter 方法还是返回值用的 getter 方法，

都可以通过点操作符调用。而且点操作符也不仅限于通过属性声明生成的访问方法，只要定义了访问方法（包括手动定义），就都可以使用点操作符来调用。

属性的内省（introspection）

通过内省可以动态查询类中声明的属性以及属性的名称和类型。

本章将首先对属性声明进行说明，然后说明使用点操作符调用访问方法的语法。本书中不介绍内省方面的内容，感兴趣的读者请参考相关书籍。

7.1.2 属性的概念

属性这个词，在不同的上下文中有不同的含义。在以前的 Objective-C 中，属性并没有被用来表示特别的功能，但从 KVC（Key-Value Coding，详情请参考第 20 章）开始，属性就被赋予了“从外部可以访问的对象的属性”这层含义。

Objective-C 2.0 中引入了属性声明和使用点操作符来调用访问方法的手法。使用属性声明可以更简洁地实现访问方法。另一方面，不仅仅是访问方法，KVC 中所有定义的实例变量都可以被当作属性处理。相比较而言，KVC 的属性是一种更广泛的概念。图 7-2 展示了 Objective-C 2.0 中几个属性的概念。

图 7-2 Objective-C 2.0 中的属性的概念

专栏：内省 COLUMN

说明数据的形式和内容的数据被叫作元数据（meta date），说明信息的信息被称为元信息（meta information）。例如，一个文件的原数据是：文件名为“会议记录”；生成时间为昨天 16:15；文件格式为 HTML；文件编码为 Unicode；只读。

一个类包含的方法和属性的相关信息也可以被看作是一种元信息。在面向对象的语言中，通过程序动态访问这些元信息的功能叫作**内省**（introspection）或者**反射**（reflection）。8.2 节中的 respondsToSelector: 就是一个自省的例子，通过它可以检查对象是否包含某个方法。

Objective-C 能够在运行时取出类和协议的相关信息。而 Objective-C 2.0 则能够在运行时从类中取出属性的定义，这也是 Objective-C 2.0 的一个新功能。更多关于运行时的详细信息请参考“Objective-C Runtime Programming Guide”等参考文档。

7.2 属性的声明和功能

7.2.1 显式声明属性

第 4 章中说明了访问类实例变量的访问方法的重要性，这种方法遵循了封装的原则。但如果需要从类的外部访问多个实例变量的话，对每个实例变量都定义访问方法就可能会很麻烦。

Objective-C 2.0 中新增加了**属性声明**（declared property）的功能。这个功能可以让编译器自动生成与数据成员同名的方法，从而就可以省去自己定义读写访问方法的工作。下面就让我们来看看到底该如何使用属性声明。

代码清单 7-1 是类 Creature 的定义。NSString 是表示字符串的类。这里为实例变量 name 定义了 getter 方法，为 hitPoint 定义了 getter 和 setter 方法，但没为 magicPoint 定义任何访问方法。另外，虽然没有 `level` 这个实例变量，但有一个同 getter 方法定义相同的 `level` 方法。关于访问方法的命名规则，请参考 4.4 节和附录 C。

代码清单 7-1 带有访问方法的简单类的例子 (Creature.h)

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

@interface Creature : NSObject
{
    NSString *name;
    int      hitPoint;
    int      magicPoint;
}
- (id)initWithName:(NSString *)str;
- (NSString *)name;                          ❶
- (int)hitPoint;                             ❷
- (void)setHitPoint:(int)val;                ❸
- (int)level;                                ❹
@end
```

在这个类中，我们首先给 int 类型的 hitPoint 定义了读写访问方法。而使用属性声明的话，只需要下面这样一行，就可以为 hitPoint 生成读写访问方法。这里的 @property 是编译器指令，后面紧跟属性的类型信息和名称。

```
@property int hitPoint;
```

属性声明等同于声明了读写两个访问方法，也就是代码清单 7-1 中的 ❷、❸ 两行。

属性声明的时候还可以为属性自定义选项。选项位于圆括号中，前面是 @property 指令。例如，如果想声明一个只读的访问方法，可以在 @property 后面加上（readonly）。下面就是给 NSString 类型的 name 声明一个只读的访问方法。

```
@property(readonly) NSString *name;
```

@property 和是否声明了实例变量无关，❹的level方法也可以用 @property 的方法来实现。用 @property 关键字重写代码清单 7-1 的接口部分，结果就如代码清单 7-2 所示。在这个例子中，我们把所有 @property 的语句都放在了后面，实际上 @property 也可以和方法声明混在一起。

代码清单 7-2 使用 @property 定义的接口文件

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

@interface Creature : NSObject
{
    NSString *name;
    int      hitPoint;
    int      magicPoint;
}
- (id)initWithName:(NSString *)str;
@property(readonly) NSString *name;
@property int hitPoint;
@property(readonly) int level;

@end
```

magicPoint 和 hitPoint 的类型一样，都是 int 类型。为 magicPoint 声明读写方法时，既可以单独写一行，也可以和 hitPoint 写在一起，如下所示。

```
@property int hitPoint, magicPoint;
```

7.2.2 属性的实现

下面说明一下实现文件中的写法。

代码清单 7-3 是代码清单 7-1 中接口部分所对应的实现。可以看到这里为每个属性分别实现了访问方法。本例中的实现都是以使用 ARC 管理内存为前提的。

代码清单 7-3 带有访问方法的类的实现例子 (Creature.m)

```
#import "Creature.h"

@implementation Creature                    // 使用 ARC

- (id)initWithName:(NSString *)str
{
    if ((self = [super init]) != nil) {
        name = str;
        hitPoint = magicPoint = 10;         // 固定值
    }
    return self;
}
```

```
- (NSString *)name {
    return name;
}
- (int)hitPoint {
    return hitPoint;
}
- (void)setHitPoint:(int)val {
    hitPoint = val;
}
- (int)level {
    return (hitPoint + magicPoint) / 10;
}

@end
```

代码清单 7-3 也是代码清单 7-2 中接口文件所对应的实现。除了上述这种写法外，还有一种更简单的写法。如下所示，通过使用 @synthesize，就可以在一行之内自动生成 getter 和 setter 方法。

```
@synthesize hitPoint;
```

上面的这行语句会自动生成属性 hitPoint 的 getter 方法`hitPoint`和 setter 方法`setHitPoint:`。@synthesize 是一种编译器功能，会让编译器为类的实例变量自动生成访问方法。

同样，通过一行语句也能为只读变量 name 生成 getter 方法。

```
@synthesize name;
```

只要将上述语句（@synthesize 加上属性名）写在 @implementation 和 @end 之间，就能自动生成和接口文件中声明的属性一致的访问方法（可读写的 getter 和 setter 方法，或者只读的 getter 方法）。其他方法可以直接在实现文件中实现，而不用在接口文件中声明。但是属性声明的情况下则不允许这种做法，接口部分中如果不使用 @property 进行定义，就无法在实现文件中使用 @synthesize。

也可以不使用 @synthesize 自动生成，而是由自己来实现访问方法，但这种情况下属性名和类型一定要和声明时一致。例如，在代码清单 7-4 中我们就没有对 level 属性使用 @synthesize，而是为其单独定义了 getter 方法。另外，我们还可以通过 @dynamic 关键字告诉编译器自动合成无效，用户会自己生成属性的 getter 和 setter 方法。@dynamic 是可选的[①]。

在代码清单 7-4 中，我们用 @synthesize 重写了代码清单 7-3 中类 Creature 的实现部分。可以看出，和原来相比，代码变得简单多了。代码清单 7-4 中也展示了 @dynamic 的使用方法。

① 利用运行时系统的功能可以动态提供方法的实现。关于这一点，本书中不做详细介绍。

代码清单 7-4 使用属性声明的方法重新实现类 Creature (Creature.m)

```
#import "Creature.h"

@implementation Creature

- (id)initWithName:(NSString *)str
{
    // 省略
}

@synthesize name;
@synthesize hitPoint;

@dynamic level;

- (int)level {
    return (hitPoint + magicPoint) / 10;
}

@end
```

7.2.3 @synthesize 和实例变量

使用 @synthesize 的时候，可以在一行中声明多个变量。

```
@synthesize hitPoint, magicPoint;
```

通常情况下，@property 声明的属性名称和实例变量的名称是相同的，但有时你也可能会需要属性的名称和实例变量的名称不同，这时就可以为实例变量定义其他名称的属性。例如，我们可以通过下面的语句生成名为 value 的访问方法，并将其绑定到实例变量 runningAverage 中。

```
@synthesize value = runningAverage;
```

正如 4.4 节中介绍的那样，我们可以在类的实现部分中声明一部分或全部实例变量，如代码清单 7-5 和代码清单 7-6 所示。这种声明方法可以隐藏是否对变量进行了属性声明。另外，在子类中访问实例变量时也只能通过访问方法来访问，不能直接访问父类的实例变量。

代码清单 7-5 没声明实例变量的接口部分的例子

```
@interface Creature : NSObject
- (id)initWithName:(NSString *)str;
@property(readonly) NSString *name;
@property int hitPoint;
@property(readonly) int level;
@end
```

代码清单 7-6 声明了实例变量的实现部分的例子

```
@implementation Creature
{
    NSString *name;
    int      hitPoint;
    int      magicPoint;
}

- (id)initWithName:(NSString *)str
{
    /* 省略 */
}
    ...
@end
```

7.2.4 通过 @synthesize 生成实例变量

属性声明的变量在接口文件和实现文件中都没有声明的情况下，通过使用 @synthesize，就可以在类的实现文件中自动生成同名同类型的实例变量[①]。

请看代码清单 7-7 和 7-8 的例子。

代码清单 7-7 接口文件中声明属性的例子

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

@interface Creature : NSObject
- (id)initWithName:(NSString *)str;
@property(readonly) NSString *name;
@property int hitPoint, magicPoint;
@property(readonly) int level;
@property int speed;
@property int skill;
@end
```

代码清单 7-8 没有显式声明实例变量的例子

```
@implementation Creature
{
    NSString *name;
    int      hitPoint;
    int      magicPoint;
}
@synthesize name;
@synthesize hitPoint, magicPoint;
@synthesize speed;
@synthesize skill = ability;
```

① 传统运行时系统（Legacy runtime）不能使用这个功能，详情请参考 8.4 节。

```
- (id)initWithName:(NSString *)str
{
    if ((self = [super init]) != nil) {
        name = str;
        hitPoint = magicPoint = 10;
        speed = ability = 5;
    }
    return self;
}

@dynamic level;

- (int)level {
    return (hitPoint + magicPoint + self.skill) / 10;
}

@end
```

新追加的属性 speed 和 skill 在接口文件和实现文件中都没有声明，但对其使用 @synthesize 之后，就会自动生成这两个实例变量。因为实例变量会被生成在类的实现文件中，所以无论是从类的外部还是从子类中都无法访问这两个实例变量。

初始化方法中使用了变量 speed 和 ability，并对其进行了赋值操作，这两个变量的有效区间从 @synthesize 的声明之后开始。

`level`方法中有 self.skill 这样的写法，这是点操作符的写法，会在 7.3 节中进行详细介绍。self.skill 就相当于调用 self 的`skill`方法取出了变量 ability 的值。

代码清单 7-7 中只有方法和属性的定义。可见，通过属性声明的方法也能够同访问方法一样实现封装的目的。

7.2.5 给属性指定选项

在前文中我们提到过用 @property 声明的时候可以给属性指定 readonly 选项，而除了 readonly 之外，还有其他一些选项，如表 7-1 所示。此外，也可以同时给一个变量指定多个选项，选项之间需要用逗号分隔。

表 7-1 @property 可用的选项

种类	选项	说明
指定方法名	getter=getter 方法名 setter=setter 方法名	显式指定 getter 方法和 setter 方法的名字
读写属性	readonly	只读
	readwrite	读写
赋值时的选项	assign	单纯赋值
	retain	进行保持操作

（续）

种类	选项	说明
赋值时的选项	unsafe_unretained	同 assign 一样（用于 ARC）
	strong	同 retain 一样（用于 ARC）
	weak	弱引用（用于 ARC）
	copy	复制对象
原子性操作	nonatomic	非原子性操作、非线程安全

也可以不使用默认的访问方法名，而通过 setter option 来指定访问属性用的方法名。例如，我们可以通过下面这行语句来指定实例变量 hitPoint 的 setter 方法为`setValue:`，注意不要忘写了最后的“:”。

```
@property(setter=setValue:) int hitPoint;
```

这个例子的情况下，我们可以通过后面要讲的点运算符来调用 .hitPoint，但实际上启动的方法是`setValue:`。

readwrite 表示属性是可读写的，这也是默认的选项。readonly 选项则意味着属性是只读的。

7.2.6 赋值时的选项

我们可以为可读写的 @property 设置选项，选项共有 6 种：assign、retain、unsafe_unretained、strong、weak 和 copy。选项之间是排他的关系，可以不设置任何选项或只设置 6 种中的一种。根据所修饰的属性是否是对象类型或者所采用的内存管理方式（手动引用计数、ARC、垃圾回收）的不同，选项的意义也会发生变化。

表 7-2 中对各种情况进行了总结，unsafe_unretained 和 strong 主要被用在使用了 ARC 的情况下，分别和 assign 和 retain 具备同样的功能。

▶ 表 7-2 赋值方法和属性的种类

	基础数据类型	对象类型		
		手动引用计数	ARC	垃圾回收
未指定任何选项	直接赋值	警告	警告	直接赋值（有可能会提示警告）
assign unsafe_unretained	直接赋值	直接赋值	直接赋值	直接赋值
retain strong	出错	赋值并对新值进行 retain 操作	赋值并对新值进行 retain 操作	无特别操作，和 assign 动作相同
weak	出错	无特别操作，和 assign 动作相同	弱引用	无特别操作，和 assign 动作相同
copy	出错	赋值时建立传入值的一份副本	赋值时建立传入值的一份副本	赋值时建立传入值的一份副本

（1）@property 的属性不是对象类型

不是对象类型的属性只可以单纯赋值，因此不需要指定任何选项，或者也可以指定 assign 选项。通过使用 @synthesize，能够生成图 7-3（a）和（b）中的 getter 和 setter 方法[①]。

（2）@property 的属性是对象类型，且手动管理内存

不指定任何选项的情况下，编译的时候会提示警告。指定了 assign 选项的情况下，通过 @synthesize 会生成图 7-3（a）和（b）中的 getter 和 setter 方法。

指定了 retain 选项的情况下，会生成图 7-3（c）中的 setter 方法，在赋值的时候应该对该对象进行保持操作。

指定了 copy 选项的情况下，会生成图 7-3（d）中的 setter 方法，并使用对象的一个副本来进行赋值。也就是说，不使用输入的对象对属性进行赋值，而是生成对象的一个副本，使用这个副本对属性赋值。这种赋值方式只适用于对象类型，并且要求该对象遵循 NSCopying 协议，且能够使用copy方法。更多关于对象复制方面的详细内容请参考第 13 章。

（3）属性是对象类型，且使用 ARC 管理内存

不指定任何选项的情况下，编译的时候会提示警告。

指定了 assign 或者 unsafe_unretained 选项的情况下，只进行单纯的赋值，不进行保持操作。声明属性对应的实例变量时需要加上 __unsafe_unretained 修饰符。因为没有被保持，所以实例变量指向的内容有可能会被释放掉而变成野指针，在使用的时候需要小心。

指定了 strong 或者 retain 选项的情况下，赋值操作之后还会对传入的变量进行保持操作，这同图 7-3（c）中的 setter 方法动作一样。实例变量在声明时需要不加任何修饰符或使用 __strong 修饰符。

指定了 weak 选项的情况下，会生成相当于弱引用赋值的代码。实例变量在声明时需要加上 __weak 修饰符。

指定了 copy 选项的情况下，会使用copy方法（第 13 章）建立传入值的一份副本，并用这份副本给实例变量进行赋值。

（4）属性是对象类型，且使用垃圾回收管理内存

这种情况下，如果不指定任何选项或指定了 assign 选项，@synthesize 会生成图 7-3（a）和（b）中的 getter 和 setter 方法。但有一点要注意的是，对于符合 NSCopying 协议也就是说可以利用copy方法的类实例变量，如果不指定任何选项的话，就会提示警告。

选项 retain 和 weak 没有意义，就算指定了也会被忽略掉，并执行和 assign 同样的动作。

对弱引用类型的实例变量进行属性设定的语句如下所示，@property 需要使用 assign 选项，实例变量需要使用 __weak 选项修饰。

```
@property(assign) __weak NSString *nickname;
```

weak 选项是 ARC 专用的，在垃圾回收的情况下是无效的。

① 图 7-3 中只是对概念进行了说明，严格来说并不一定是这样的。另外也会受到 nonatomic 的有无的影响。

指定 copy 选项后，会生成图 7-3（d）中的 setter 方法。

图 7-3 不同条件下生成的 setter 和 getter

```
- (TYPE)name {
    return name;
}
```

（a）单纯的getter的定义

```
- (void)setName:(TYPE)obj
    name = obj;
}
```

（b）单纯进行赋值操作时的setter方法

```
- (void)setName:(TYPE)obj {
    if (name != obj) {
        [name release];
        name = [obj retain];
    }
}
```

（c）带有保持操作时的setter方法

```
- (void)setName:(TYPE)obj {
    if (name != obj) {
        [name release];  // 垃圾回收的情况下无意义
        name = [obj copy];
    }
}
```

（d）带有copy操作时的setter方法

7.2.7 原子性

表 7-1 中的最后一个选项是 nonatomic，这个选项是在多线程环境下使用的。关于线程和锁的更多详细信息请参考第 19 章。

Nonatomic 表示访问方法是非原子的。原子性是多线程中的一个概念，如果说访问方法是原子的，那就意味着多线程环境下访问属性是安全的，在执行的过程中不可被打断。而 nonatomic 则正好相反，访问方法被 nonatomic 修饰的情况下，就意味着访问方法在执行的时候可被打断。缺省情况下访问方法是原子的。

图 7-3 中的 getter 和 setter 方法都是指定了 nonatomic 选项时的实现。当属性为对象类型，使用了 retain 并且没有指定 nonatomic 时，getter 和 setter 方法的实现如图 7-4 所示。没指定 nonatomic 的时候，访问方法中需要使用 lock 和 unlock 来保证方法的原子性（_ex 是系统提供的锁函数）。

如果在多个线程中同时调用 getter 或 setter，就有可能会出现值丢失或内存泄漏等错误。通过使用 lock 和 unlock，能够保证每次最多有一个线程执行 lock 和 unlock 之间的代码，从而也就保证了原子性（详情请参考 19.2 节）。

另外，图 7-4（a）的 getter 方法中除了被追加了 lock 和 unlock 之外，还增加了`retain`和`autorelease`的操作。getter 方法中没有直接返回实例变量，而是使用`retain`和`autorelease`创建了一个实例变量的副本，最后返回的是这个新建的副本。这么做是为了防止在返回实例变量的时候，另外的线程释放了这个实例变量。没指定 nonatomic 选项的 getter 方法中不需要使用`retain`和`autorelease`。

因为这样的机制能提高访问方法的安全性，所以通常不需要指定 nonatomic 选项。但毕竟 lock 和 unlock 操作对性能有影响，因此，对于使用频繁且不用考虑多线程竞争的访问方法，可以在声明的时候加上 nonatomic。

nonatomic 选项不仅能被用于使用 @synthesize 生成的访问方法，手动定义的访问方法中不存在多线程竞争的情况下，也可以给属性加上 nonatomic 选项。

属性声明中的属性和关联引用（见 10.3 节）中通过策略指定的动作是一致的。

图 7-4 使用了 retain 而没有指定 nonatomic 选项时的情况

```
- (TYPE)name {
    [_ex lock]; // 局部锁
    TYPE rtn = [[name retain]
                  autorelease];
    [_ex unlock];
    return rtn;
}
```

（a）getter方法的定义

```
- (void)setName:(TYPE)obj {
    [_ex lock]; // 局部锁
    if (name != obj) {
        [name release];
        name = [obj retain];
    }
    [_ex unlock];
}
```

（b）setter方法的定义

7.2.8 属性声明和继承

子类中可以使用父类中定义的属性，也可以重写父类中定义的访问方法。但是，父类中属性声明时指定的各种属性（assign、retain 等），或者为实例变量指定的 getter 和 setter 的名称等必须完全一样。

唯一一个特别的情况是，对于父类中被定义为 readonly 类型的属性，子类中可以将其变为 readwrite。虽然不可以在子类中使用 @synthesize 对父类中的实例变量生成访问方法，但可以手动实现对应的访问方法。这是为了防止子类可以轻易地访问父类中隐藏的实例变量。

属性的声明可能会包含范畴（第 10 章）或协议（第 12 章）。这种情况下实现文件中不可以使用 @synthesize，原因是范畴和协议都和实例变量的实现无关，需要在实现文件中实现访问方法。

7.2.9 方法族和属性的关系

使用 ARC 的时候，必须注意方法的命名，不要和方法族发生冲突（见 5.4 节）。

属性声明的时候会默认生成和属性同名的 getter 访问方法，需要注意属性名是否和方法族名冲突。特别要注意以 new 开头的属性名的情况。

7.3 通过点操作符访问属性

7.3.1 点操作符的使用方法

Objective-C 2.0 中新增了使用点操作符来访问属性的功能，我们首先来看一下点操作符的使用方法。

上一节的代码清单 7-1 和代码清单 7-3 中用手动定义访问方法的方式定义了类 Creature。代码清单 7-2、以及代码清单 7-4 到 7-8 的代码中用属性声明的方法重新定义了类 Creature。而无论采用那种定义方式，执行起来都是一样的。代码清单 7-9 展示了如何使用点操作符来访问属性。

代码清单 7-9 中以 @ 开头的 @"Nike" 是字符串常量的表示方式。类 NSString 的 UTF8String 方法可把 NSString 转为 C 语言的字符串类型。更多关于字符串类 NSString 的详细内容请参考 9.2 节。

代码清单 7-9 使用点操作符的例子

```
int main(void)
{
    Creature *a = [[Creature alloc] initWithName:@"Nike"];
    a.hitPoint = 50;
    printf("%s: HP=%d (LV=%d)\n",
        [a.name UTF8String], a.hitPoint, a.level);
    return 0;
}
```

请注意一下这里的变量 a。它像访问结构体中的元素那样使用了点操作符来获取、修改类的属性。程序的执行结果如下所示。

```
Nike: HP=50 (LV=6)
```

Objective 2.0 会**在编译时把使用点操作符访问属性的过程理解为访问方法的调用**。因为调用的是访问方法，所以无论对应的实例变量是否存在，只要访问方法存在，就都可以通过点操作符访问属性。

点操作符只能用于类类型的实例变量，**不能对 id 类型的变量应用点操作符**。因为没指定类型的情况下，编译器无法判断是否存在属性对应的访问方法。在代码清单 7-9 的例子中，如果将变量 a 的类型声明为 id 类型，程序就会出错。除了 id 类型外，void 类型和 C 语言中的数组类型的变量也都不能应用点操作符，而除了以上这些类型的变量之外，则都可以应用点操作符。

另外，点操作符可用于可读写的属性或只读的属性，不能用于只写的属性。这是因为属性声明中只能声明可读写的属性或只读的属性，不能声明只写的属性。

点操作符和消息表达式的对比如表 7-3 所示。

表 7-3 点操作符和访问方法的关系

操作	使用点操作符	使用消息表达式
赋值（setter）	obj.name = val;	[obj setName:val];
获取（getter）	val = obj.name;	val = [obj name];

7.3.2 复杂的点操作符的使用方法

下面对复杂的点操作符进行说明。

（1）连用点操作符

点操作符可以连用，如下所示。

```
n = obj.productList.length;
obj.contents.enabled = YES;
```

因为点操作符按照从左向右的顺序进行解释，所以上面的表达式可被解释为

```
n = [ [obj productList] length ];
[[obj contents] setEnabled: YES];
```

当一个对象的实例变量是另外一个对象时，可通过连用点操作符来访问对象的实例变量中的成员。如果连用表达式中有一个是 nil，则整个表达式的返回值就是 nil。

（2）连续赋值

下面的连续赋值的表达式会被如何解释呢？

```
n = 0;
k = obj.count = obj.depth = ++n
```

赋值时是从右向左解释，因此上面的表达式可被改为下面这样。

```
k = (obj.count = (obj.depth = ++n) )
```

内侧括弧中相当于执行了 [obj setDepth: ++n]，表达式的值就是 ++n 的值 1。外侧括弧中相当于执行了 [obj setCount: 1]，表达式的值也是 1。最后变量 n 和 k 还有 obj 的两个属性的值都为 1。

点操作符和 C 语言中的宏定义不同，不会对 n 的值重复加 1。如果 obj 是一个结构体的话，上面这段代码的执行结果和现在的结果则相同。

（3）对递增、递减和复合赋值运算符的解释

下面的这个表达式会被如何解释呢？

```
e = obj.depth++;
```

这个有点复杂，赋值表达式的右侧连续调用了 getter 和 setter 方法，相当于执行了 `[obj setDepth: [obj depth] + 1]`。最后为 e 赋值的是递增操作之前的 depth 的值。

复合赋值表达式也是一样，需要连续调用 getter 和 setter 方法。这种情况下，`obj.depth *= n;` 就相当于 `[obj setDepth: [obj depth] * n];`。而把 obj 换成结构体变量时，上面的这些表达式也都会得出同样的结果。

（4）self 使用点操作符

类的方法中可以通过对 self 应用点操作符来调用自己的访问方法。但要注意的是，不要在访问方法中使用 self，否则就会造成无限循环的递归，无法终止。

```
self.count = 12;
obj.depth = self.depth + 1
```

（5）super 使用点操作符

可以通过给 super 加点操作符来调用父类中定义的 setter 和 getter 方法。

```
- (void)setDepth:(int)val {
    super.depth = (val <= maxDepth) ? val : maxDepth;
}
```

（6）和构造体的成员混用

获取类属性的点操作符和访问结构体元素的点操作符可以混用，但有一些需要注意的事项。

例如，重设窗口大小的时候会用属性 minSize 来表示窗口的最小大小，这个属性就是一个结构体类型的变量（NSSize 类型，详情请参考附录 A），其中存储了窗口的长和宽。当需要获取窗口的最小宽度时，可以按照如下方式书写代码。

```
w = win.minSize.width;
```

上面这行代码等价于下面这种写法，但需要注意的是，width 前面的点操作符是访问结构体中的元素时使用的。

```
w = [win minSize].width;
```

但不能使用这种直观的写法来为 width 赋值，例如，下面这种赋值写法就是不允许的。`setMinSize:`只允许使用 NSSize 类型的结构体变量（sz）来对 minSize 进行赋值。

```
win.minSize.width = 320.0;  /* 不允许这种写法 */
```

如果要为 minSize 的 width 赋值，需要像下面这样写。

```
NSSize sz = win.minSize;
sz.width = 320.0;
win.minSize = sz;
```

虽然可以像 &cell.size 这样通过取地址符 & 来对结构体的成员取地址，但不能通过取地址符来对点操作符获得的属性取地址，这就和不能对函数的返回值取地址是一样的。

还有，当想给 obj 的实例变量 contents 发送消息时，你可能会像下面这样写。

```
[obj.contents retain]
```

但需要注意的是，实际上这行语句表示的是给 getter 方法的返回值发送了消息，并不一定会给 obj 的实例变量 contents 发送消息。

使用点操作符访问对象的实例变量和 C 语言中使用点操作符访问结构体的成员意义是不一样的。如第 4 章所述，访问对象的实例变量的最正统的方法是通过->操作符来访问。编译器在碰到点操作符的时候并没有直接访问实例变量而是调用了访问方法。

7.3.3 何时使用点操作符

没有参数的方法，无论其是不是和属性相关，都可以和 getter 方法一样通过点操作符来调用。例如，代码清单 7-9 中 NSString 的`UTF8String`方法就是通过点操作符调用的。但原则上还是应该只对属性声明中定义的属性应用点操作符。

在 C 语言中，使用点操作符访问结构体的成员基本上无任何额外负担。而同 C 语言不同，Objective-C 中使用点操作符则会带来调用访问方法的负担，因此，在某些对性能要求严格的情况下使用点操作符实在不能说是一个好的选择。

在 C++ 和 Java 等很多语言中，都可以通过点操作符调用对象的方法。但对 Objective-C 来说，使用点操作符的目的只是访问属性。

属性也会被用在同一个类的其他方法的实现中。例如，假设属性 hitPoint 被绑定到了实例变量 hitPoint 中，那么，同一个类的别的方法中是应该直接访问实例变量 hitPoint，还是应该通过 self.hitPoint 这种方法来访问 hitPoint 呢？

严格来说，使用依赖于实现的方式来访问实例变量是不允许的，所以应该避免直接访问实例变量。但属性对应的访问方法则一定要直接访问实例变量。无论选择哪种方法都是类内部的实现，对外都是不可见的。所以应该从执行效率、实现的难易度和是否有继承等方面综合考虑。

此外，在初始化方法中通过点操作符访问属性的时候也要注意。因为初始化方法执行的时候这个实例还没完成初始化，属性对应的访问方法有可能还没生成，所以访问时稍不注意就可能会带来风险。

第8章

类NSObject和运行时系统

使用 Objective-C 进行面向对象编程时，除了需要知道语言本身的语法和面向对象的知识外，还需要了解 Objective-C 的根类 NSObject 的信息。

本章将对根类 NSObject 的主要功能和 NSObject 与运行时系统之间的关系进行说明。

8.1 类 NSObject

8.1.1 根类的作用

作为一门动态编程语言，Objective-C 有很多动态的特性，因此，Objective-C 不仅需要编译环境，同时还需要一个**运行时系统**（runtime system）来执行编译好的代码。运行时系统扮演的角色类似于 Objective-C 的操作系统，它负责完成对象生成、释放时的内存管理、为发来的消息查找对应的处理方法等。

通常情况下，程序中无法直接使用运行时系统提供的功能。根类方法中提供了运行时系统的基本功能。继承了 NSObject 的所有类都可以自由地使用运行时系统的功能，也就是说，根类就相当于运行时系统的一个接口。

根类通过哪些方式提供了哪些功能对系统有很大的影响。因此，根类不同的系统之间是无法开发出通用的程序的。

Cocoa 是以 OPENSTEP 的核心 API 为基础发展起来的。OPENSTEP 的前身称为 NeXTstep。在 NeXTstep[①] 时代，根类是类 Object，而在 OPENSTEP 时代，根类则变为了 NSObject，同时类的设计方面也得到了大幅改进。

下面我们将对 NSObject 的主要功能进行说明，但不会对 NSObject 的所有方法逐一说明。另外，其中还会涉及一些目前还没有介绍到的 Objective-C 语言方面的功能，关于这些功能，我们会在后面的章节中进行详细说明。

8.1.2 类和实例

NSObject 只有一个实例变量，就是 Class 类型的变量 isa。isa 用于标识实例对象属于哪个类对象。因为 isa 决定着实例变量和类的关系，非常重要，所以子类不可修改 isa 的值。另外，也不能通过直接访问 isa 来查询实例变量到底属于哪个类，而要通过实例方法`class`[②] 来完成查询。

下面对类和实例变量的相关方法进行说明。NSObject 的方法与其说是为自己定义的，不如说是为其子类和所有的实例对象而定义的。

```
- (Class) class
```

返回消息接收者所属类的类对象。

```
+ (Class) class
```

返回类对象。

虽然可以使用类名作为消息的接收者来调用类方法，但当类对象是其他消息的参数，或者将类对象

① 也称为 NeXTSTEP 或 NEXTSTEP。

② 特别是第 20 章 Key-Value-Coding 的参考文档中提到的 isa 并不一定代表类。

赋值给变量的时候，需要通过这个类方法来获取类对象。

- (id) **self**

返回接收者自身。是一个无任何实际动作但很有用的方法。

- (BOOL) **isMemberOfClass:** (Class) aClass

判断消息接收者是不是参数aClass类的对象。

- (BOOL) **isKindOfClass:** (Class) aClass

判断消息接收者是否是参数aClass类或者aClass类的子类的实例。这个函数和isMemberOfClass:的区别在于当消息的接收者是aClass的子类的实例时也会返回YES。

+ (BOOL) **isSubclassOfClass:** (Class) aClass

判断消息接收者是不是参数aClass的子类或自身，如果是则返回YES。

- (Class) **superclass**

返回消息接收者所在类的父类的类对象。

+ (Class) **superclass**

返回消息接收类的父类的类对象。

8.1.3 实例对象的生成和释放

+ (id) **alloc**

生成消息接收类的实例对象。通常和init或init开头的方法连用，生成实例对象的同时需要对其进行初始化（见2.1节）。子类中不允许重写alloc方法。

- (void) **dealloc**

释放实例对象（见5.2节）。dealloc被作为release的结果调用。除了在子类中重写dealloc的情况之外，程序中不允许直接调用dealloc。

- (oneway void) **release**

将消息接收者的引用计数减1。引用计数变为0时，dealloc方法被调用，消息接收者被释放（详情请参考5.2节）。关于oneway关键字，请参考第19章。

- (id) **retain**

为消息的接收者的引用计数加1，同时返回消息接收者（详情请参考5.2节的内容）。

- (id) **autorelease**

把消息的接收者加入到自动释放池中，同时返回消息接收者（详情请参考5.2节的内容）。

- (NSUInteger) **retainCount**

返回消息接收者的引用计数，可在调试时使用这个方法。NSUInteger是无符号整数类型，详情请参考8.4节的内容。

- (void) **finalize**

垃圾收集器在释放接收者对象之前会执行该finalize方法。关于这个方法的定义，请参考6.1节的内容。

上面从dealloc到retainCount都是手动引用计数管理内存时使用的方法，使用ARC时不可用。finalize仅供垃圾回收有效时使用。

更多和复制相关的方法的详细内容，请参考第13章。

8.1.4 初始化

- (id) **init**

init可对alloc生成的实例对象进行初始化。子类中可以重写init或者定义新的以init开头的初始化函数，更多详细内容可参考3.2节。

+ (void) **initialize**

被用于类的初始化，也就是对类中共同使用的变量进行初始化设定等，详情请参考4.5节。这个方法会在类收到第一个消息之前被自动执行，不允许手动调用。

+ (id) **new**

new是alloc和init的组合。new方法返回的实例对象的所有者就是调用new方法的对象。但是，把alloc和init组合定义为new并没有什么优点。

根据类的实现的不同，new方法并不会每次都返回一个全新的实例对象。有时new方法会返回对象池中预先生成的对象，也有可能每次都返回同一个对象。

8.1.5 对象的比较

- (BOOL) **isEqual:** (id) anObject

消息的接收者如果和参数anObject相等就返回YES。

- (NSUInteger) **hash**

在把对象放入容器(详情请参考9.4节)等的时候，返回系统内部用的散列值[①]。

原则上来讲，具有相同id值也就是同一个指针指向的对象会被认为是相等的。而子类在这个基础上又进行了扩展，把拥有相同值认为是相等。我们可以根据需求对“相同值”进行定义，但一般都会让具备“相同值”的对象返回相同的散列值，为此就需要对散列方法进行重新定义。而反之则并不成立。也就是说，散列值相等的两个对象不一定相等。

另外，有的类中还自定义了compare:或者isEqualTo之类的方法。至于到底利用哪个方法，或者自定义类的时候是否需要定义比较用的方法，都需要根据目的和类的内容做具体分析。

① 快速检索系统中一般都有散列表，Objective-C的Foundation框架中也有用于计算散列的函数。数据存放的位置是由数据本身计算出来的散列值来决定的。即使内容相同的情况下，如果算出来的散列值不同，数据存放的位置也是不同的。计算散列值的方法多种多样，感兴趣的读者请参考算法教材。

8.1.6 对象的内容描述

+ (NSString *) **description**

返回一个NSString类型的字符串，表示消息接收者所属类的内容。通常都是这个类的类名。

- (NSString *) **description**

返回一个NSString类型的字符串，表示消息接收者的实例对象的内容。通常是类名加id值。子类中也可以重新定义description的返回值。例如，NSString的实例会返回字符串的内容，NSArray的实例会对数组中的每一个元素调用description，然后将调用结果用句号进行分割，并一起返回，详情请参考9.4节。

8.2 消息发送机制

8.2.1 选择器和 SEL 类型

至今为止我们把选择器（方法名）和消息关键字放在一起进行了说明。程序中的方法名（选择器）在编译后会被一个内部标识符所替代，这个内部标识符所对应的数据类型就是 SEL 类型。

Objective-C 为了能够在程序中操作编译后的选择器，定义了 @selector() 指令。通过使用 @selector() 指令，就可以直接引用编译后的选择器。使用方法如下所示。

```
@selector(mutableCopy)
@selector(compare:)
@selector(replaceObjectAtIndex:withObject:)
```

也可以声明 SEL 类型的变量。

选择器不同的情况下，编译转换后生成的 SEL 类型的值也一定不同，相同选择器所对应的 SEL 类型的值一定相同。Objective-C 的程序员不需要知道选择器对应的 SEL 类型的值到底是什么，具体的值是和处理器相关的。但是，如果 SEL 类型的变量无效的话，可设其为 NULL，或者也可以使用 (SEL)0 这种常见的表达方法。

可以使用 SEL 类型的变量来发送消息，为此，NSObject 中准备了如下方法。

- (id) **performSelector:** (SEL) aSelector

向消息的接收者发送aSelector代表的消息，返回这个消息执行的结果。

- (id) **performSelector:** (SEL) aSelector
withObject: (id) anObject

向消息的接收者发送aSelector代表的消息，消息的参数为anObject，返回这个消息执行的结果。

例如，下面两个消息表达式进行的处理是相同的。

```
[target description];
[target performSelector: @selector(description)];
```

下面的例子展示了如何根据条件动态决定执行哪个方法。

```
SEL method = (cond1) ? @selector(activate:) : @selector(hide:);
id obj = (cond2) ? myDocument : defaultDocument;
[target performSelector:method withObject:obj];
```

这种调用方法的方式很像 C 语言中函数指针（请参考后文中关于函数指针的 Column）的用法。使用函数指针也可以实现和上面的程序同样的功能。

函数指针是函数在内存中的地址。指针对应的函数是在编译的时候决定的，不能够执行指定之外的函数。SEL 类型就相当于方法名，根据消息接收者的不同（上例子中 target 的赋值），来动态执行不同的方法。

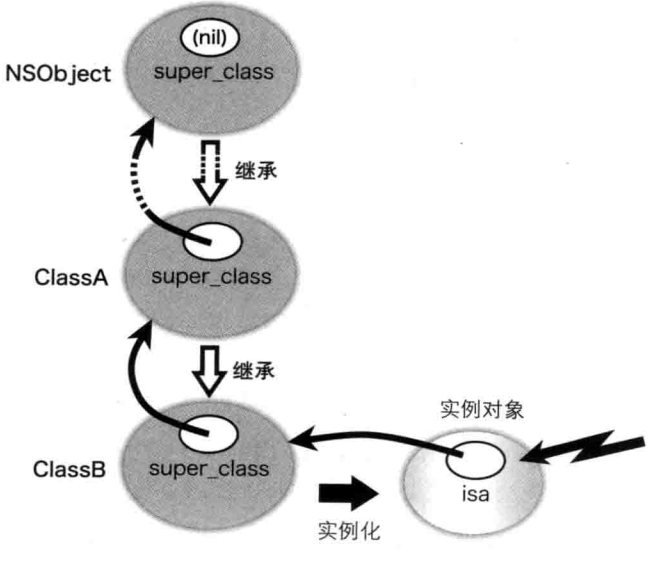
通过 SEL 类型来指定要执行的方法，这就是 Objective-C 消息发送的方式。也正是通过这种方法，才实现了 Objective-C 的动态性。

8.2.2 消息搜索

对象收到一个消息后会执行哪个方法是被动态决定的。

所有的实例变量都存在一个 Class 类型的 isa 变量，它就是类对象。当收到消息后，运行时系统会检查类内是否有和这个消息选择器相同的方法，如果有就执行对应的方法，如果没有就通过类对象中指向父类的指针来查找父类中是否有对应的方法（如图 8-1 所示）。如果一直找到根类都没有找到对应的方法，就会提示执行时错误。详情可参考 3.2 节。

图 8-1 消息探索和类对象



如果每次收到消息都需要查找相应的方法的话，消息发送过程的开销就会很大。针对这种情况，运行时系统内部会缓存一个散列表，表中记录着某个类拥有和什么样的选择器相对应的方法、方法被定义在何处等信息。这样一来，当下次再收到同样的消息时，直接利用上次缓存的信息即可，就不需要再重头进行搜索了。

NSObject 中定义了可以动态查询一个对象是否能够响应某个选择器的方法。

- (BOOL) **respondsToSelector:** (SEL) aSelector

  查询消息的接收者中是否有能够响应aSelector的方法，包括从父类继承来的方法。如果存在的话，返回YES。

+ (BOOL) **instancesRespondToSelector:** (SEL) aSelector

  查询消息的接收者所属的类中是否有能够响应aSelector的实例方法，包括从父类继承来的方法。如果存在的话，返回YES。

8.2.3 以函数的形式来调用方法

类中定义的方法通常是以函数的形式实现的，但通常在编程的时候并不会直接操作方法所对应的函数。

但如果想尽可能地让程序更快一点，或者需要按照C语言的惯例传递函数指针的时候，可以直接调用方法对应的函数，以节省发送消息的开销。另外，执行时动态加载方法的定义等时，也可以将方法作为函数调用。但要注意的是，如果以函数的形式来调用方法的话，将无法利用面向对象的动态绑定等功能。虽然消息发送同函数调用相比确实慢一点，但却有面向对象的动态绑定、多态等优点。同这些优点相比，速度上略微的损失是不值得一提的。而其实消息发送的速度也是非常快的。在下一节中，我们会用数字来比较消息发送的速度。

通过使用下面的方法，可以获得某个对象持有的方法的函数指针，这些方法都被定义在NSObject中。

- (IMP) **methodForSelector:** (SEL) aSelector

  搜索和指定选择器相对应的方法，并返回指向该方法实现的函数指针。实例对象和类对象都可以使用这个方法。对实例对象使用时，会返回实例方法对应的函数，对类对象使用时，会返回类对象对应的函数。

+ (IMP) **instanceMethodForSelector:** (SEL) aSelector

  搜索和指定选择器相对应的实例方法，并返回指向该实例方法实现的函数指针。

IMP是“implementation”的缩写，它是一个函数指针（关于函数指针请参考下面的Column），指向了方法实现代码的入口。

IMP的定义为

```
typedef id  (*IMP)(id, SEL, ...);
```

这个被指向的函数包括id（self指针）、调用的SEL（方法名），以及其他一些参数。

例如，有下面这样一个方法。

```
- (id)setBox:(id)obj1 title:(id)obj2;
```

foo是这个方法所属类的一个实例变量。获取指向setBox的函数指针，并通过该指针进行函数调用的过程如下所示。

```
IMP funcp;
funcp = [foo methodForSelector:@selector(setBox:title:)];
xyz = (*funcp)(foo, @selector(setBox:title:), param1, param2);
```

通过这个例子能够看出，调用方法对应的函数时，除了方法声明时的参数外，还需要把消息接收对象和消息的选择器作为参数。虽然没有明确声明，但方法内部也可以访问这两个参数，因此这两个参数也被叫作**隐含参数**（hidden arguments）。第一个参数消息的接收者实际上就是self，第二个参数选择器可以通过_cmd这个变量来访问。

因为没有明确指定IMP的方法参数的类型，所以编程的时候可以把一个实际的函数指针赋值给IMP类型的变量（需要通过cast进行类型转换）。

专栏：函数指针 COLUMN

C语言可以把函数名作为函数指针来处理。函数指针指向的是函数代码的首地址。函数指针可以赋值给变量或数组成员，也可以作为其他函数的参数。函数指针还可以作为结构体的变量，让结构体的处理像对象的处理一样。

函数指针的类型声明有一些麻烦。例如，假设有一个函数，其属性声明如下。

```
int funcA(double x, int *err);
```

声明一个指向该函数的函数指针p，如下所示。

```
int (*p)(double, int *);
```

如果需要大量声明这个类型的变量，可以使用typedef来声明一个类型，假设新类型的名称为t_func，具体方法如下所示。

```
typedef int (*t_func)(double, int *);
```

通过函数指针来调用函数一般有两种写法，如下面的（1）和（2）所示。这两种写法都能够完成函数的调用，但第二种写法的情况下，如果存在fptr函数，就有可能会发生冲突，所以一般我们都使用写法（1）来完成调用。

```
t_func fptr = funcA;
...
x = (*fptr)(1.4142, &error);    //(1)
x = fptr(1.4142, &error);       //(2)
```

8.2.4 对 self 进行赋值

上文中我们提到了 self 是方法的一个隐含参数，它代表的是收到消息的对象自身，因此，通过 self 可以给自己再次发送消息，self 也可以作为消息的参数或方法的返回值来使用。

此外，还可以对 self 进行赋值操作。初始化方法的定义中，用父类初始化方法的返回值对 self 进行赋值，如下例所示。

```
- (id)initWithMax:(int)a   /* 推荐使用这种写法 */
{
    if ((self = [super init]) == nil) {
        max = a;  // 从这里开始对子类进行初始化操作
    }
    return self;
}
```

在 OPENSTEP 时代，Objective-C 的初始化方法一般都采用下面这种写法。这种写法的前提是父类的初始化方法不会出错，但这里需要注意的是没有用父类初始化方法的返回值对 self 进行赋值。子类的初始化方法和父类的初始化方法都是对同一个对象进行操作的，所以不需要显式地对 self 进行赋值操作。

```
- (id)initWithMax:(int)a   /* 旧的写法 */
{
    [super init];
    max = a;
    return self;
}
```

需要注意的是这种写法也有可能出错。除了初始化失败之外，父类的初始化方法也有可能并**没有返回 self 而是返回了其他对象**。一个典型的例子是，由类簇（class cluster，详情请见第 11 章）构成的类在初始化方法中就没有返回 self。

所以，在定义初始化方法时，用父类初始化方法的返回值对 self 进行赋值并判断其不为 nil 是一种更安全的做法。另外，在用 ARC 的时候，如果初始化方法的返回值没被用到，编译时就会发生错误。

而如果用一个对象给 self 赋值的话会发生什么呢？self 代表消息的接收者，如果用一个对象给 self 赋值，那么这个对象就会变为消息的接收者继续运行下去。除了在初始化方法中之外，对 self 赋值都是一种非常有技巧性的操作，会让程序变得不好理解，因此不推荐使用。而使用 ARC 的时候，如果在初始化方法以外对 self 赋值，就会出现编译错误。

8.2.5 发送消息的速度

虽然我们可以肯定地说发送消息要比函数调用慢一些，但实际上会慢多少呢？让我们用一个简单的程序来测试一下。

首先来看看下面这个简单的程序。在这个程序中，我们会不断地给一个对象发送相同的消息。程序中的 LOOP 是宏定义。

代码清单 8-1 测试程序（1）：消息送信

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

unsigned long rnd = 201109;

@interface testObj : NSObject
- (int)testMethod;
@end

@implementation testObj
- (int)testMethod {
    rnd = rnd * 1103515245UL + 12345; // 计算随机数
    return (rnd & 1) ? 1 : -1;
}
@end

int main(void)
{
    id obj = [[testObj alloc] init];
    int v = [obj testMethod];
    for (int i = 0; i < LOOP; i++)
        for (int j = 0; j < 20000; j++)
            v += [obj testMethod];
    return (v == 0);
}
```

这里我们修改了 main() 函数，并生成了程序（2）~（4）。程序（2）中使用了 performSelector: 来进行消息送信。程序（3）中没有使用消息调用的方式，而是在程序的最开始获取了方法对应的函数指针，用函数调用的形式来调用方法。程序（4）中没用调用方法或函数，而是把方法中的操作直接在程序中展开了。

代码清单 8-2 程序（2）：使用 performSelector:

```
int main(void)
{
    id obj = [[testObj alloc] init];
    int v = [obj testMethod];
    for (int i = 0; i < LOOP; i++)
        for (int j = 0; j < 20000; j++)
            v += (int)[obj performSelector:@selector(testMethod)];
    return (v == 0);
}
```

代码清单 8-3 程序（3）：函数调用的形式

```
int main(void)
{
    int (*f)(id, SEL);
    id obj = [[testObj alloc] init];
    int v = [obj testMethod];
    f = (int (*)(id,SEL))[obj methodForSelector:@selector(testMethod)];
    for (int i = 0; i < LOOP; i++)
        for (int j = 0; j < 20000; j++)
            v += (*f)(obj, @selector(testMethod));
    return (v == 0);
}
```

代码清单 8-4 程序（4）：不使用方法或函数调用，直接展开

```
int main(void)
{
    id obj = [[testObj alloc] init];
    int v = [obj testMethod];
    for (int i = 0; i < LOOP; i++)
        for (int j = 0; j < 20000; j++) {
            rnd = rnd * 1103515245UL + 12345;
            v += (rnd & 1) ? 1 : -1;
        }
    return (v == 0);
}
```

下面，使用 time 命令（详情请参考 6.3 节）来测试每个程序执行的时间。表 8-1 中展示了每个程序执行 5 次的平均时间[①]。宏变量 LOOP 被设为了 8000。

表 8-1 测试程序的执行时间

	执行时间（单位：秒）	函数调用的时间（执行时间 –A）	比率（函数调用时间 /B）
（1）消息送信	1.128	0.622	2.046
（2）performSelector:	2.700	2.194	7.217
（3）函数调用	0.810	0.304 =B	1.000
（4）直接展开	0.508 = A	—	—

程序（1）~（3）的执行时间减去程序（4）的执行时间（A），就可以得到消息送信或函数调用所花费的时间。另外，如果我们把函数调用的时间（B）看作 1.0，那么就可以算出程序（1）和程序（2）的调用时间比率。

从这个试验中可以看出，消息送信所需要的时间是函数调用所需时间的 2 倍（不同的测试机得到的结果会存在一定的出入）。在本例中，因为一直都是发送相同的消息，所以可以利用缓存，考虑到这一点，

① 测试用机为 Mac mini（Mac OS X 10.7.1, 2.66GHZ Intel Core 2 Duo）。使用的编译器是 clang，编译选项为 -O2（大写字母 O 和阿拉伯数字 2）。clang 如果不使用优化选项的话会生成很多调试用的代码。另外，使用编译器 gcc 也能得到和优化后的 clang 相近的结果。

实际执行时所需要的时间可能会比这个更多一点。另外，直接发送消息比用 `performSelector:` 发送消息速度更快。

还有一点要注意的是，我们从上面的测试结果不能得出“Objective-C 的程序比 C 语言的程序慢 2 倍”这种结论。上面的测试仅仅是发送消息和函数调用开销方面的一个比较。在程序内部的数据处理方面，Objective-C 和 C 语言是使用了同样的方法实现的。所以，一般情况下，同样功能的程序用 Objective-C 实现会比用 C 语言实现更慢一下。

当然，和解释性语言相比的话，Objective-C 程序的速度要快很多。

8.2.6 类对象和根类

因为类对象也是一个对象，所以类对象可以作为根类 NSObject 的某个子类的对象来使用。下面这个语句看上去好像比较奇怪，但实际上它是正确的，会返回 YES。

```
[[NSString class] isKindOfClass:[NSObject class]]
```

这就说明了相当于类对象的类的对象是存在的。而类对象的类就被叫作**元类**（metaclass）。实例对象（instance object）所属的类是 class，类对象（class object）所属的类是 metaclass。

Objective-C 中的很多概念都来源于 Smalltalk，元类的概念就是其中之一。但现在的 Objective-C 中已经不存在元类的概念了，程序中也不能操作元类。用于表示对象的 id 类型和表示类的 Class 类型实际上都是指向结构的指针，被详细定义在 /usr/include/objc 下面的 objc.h 头文件中。通过查看 objc.h 中 id 和 Class 的定义，就会发现类和元类的关系如图 8-2 所示。Objective-C 2.0 更新了基本的数据结构，但没有改变类和元类的关系。

图 8-2 元类的概念

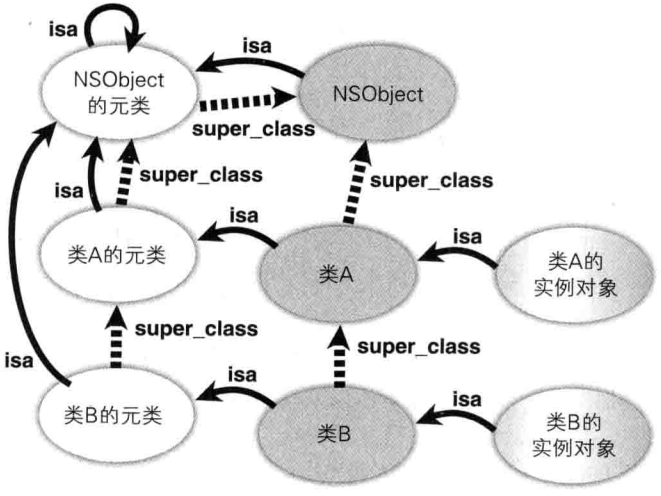


图 8-2 中，类 A 是 NSObject 的子类，类 B 是 A 的子类。类对象和实例对象中都存在一个成员变量 isa，它是一个 objc_class 类型的指针。

图中带有 isa 的实线表明了 isa 指向的对象，带有 super_class 的虚线则表明了父类的关系。

类对象中保存的是实例方法，元类对象中保存的是类方法。通过这样的定义能够统一实现实例方法和类方法的调用机制。

因为编程时不可以直接操作元类，所以并不需要完全了解图 8-2 中的细节。大家只需要记住任何一个类对象都是继承了根类的元类对象的一个实例即可。也就是说，**类对象可以执行根类对象的实例方法**。

例如，类对象可以执行 NSObject 的实例方法`performSelector:`和`respondsToSelector:`。当然前提是没有将这些方法作为类方法再次定义。

下面让我们总结一下。其中，（1）和（2）我们已经介绍过了。（3）比较不容易理解，请边看图 8-2 边思考。

（1）所有类的实例对象都可以执行根类的实例方法

- 如果在派生类中重新定义了实例方法，新定义的方法会被执行。

（2）所有类的类对象都可以执行根类的类方法

- 如果在派生类中重新定义了类方法，新定义的方法会被执行。

（3）所有类的类对象都可以执行根类的实例方法

- 即使在派生类中重新定义了实例方法，根类中的方法也会被执行。
- 如果在派生类中将实例方法作为类方法重新定义了的话，新定义的方法会被执行。

8.2.7 Target-action paradigm

通过使用 SEL 类型的变量，能够在运行时动态决定执行哪个方法。实际上，Application 框架就利用这种机制实现了 GUI 控件对象间的通信，让我们看看下面这个例子。

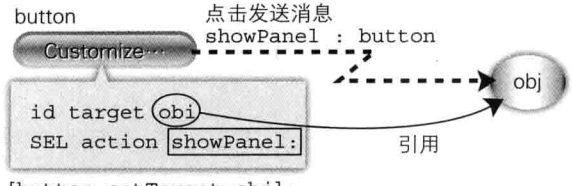
```
@interface myCell : NSObject
{
    SEL action;
    id  target;
    ...
}
- (void)setAction:(SEL)aSelector;
- (void)setTarget:(id)anObject;
- (void)performClick:(id)sender;
...
@end

@implementation myCell

- (void)setAction:(SEL)aSelector
```

```
{
    action = aSelector;
}

- (void)setTarget:(id)anObject
{
    target = anObject;
}

- (void)performClick:(id)sender
{
    (void)[target performSelector:action withObject:sender];
}
...
@end
```

这个类有 SEL 类型的实例变量 action 和 id 类型的实例变量 target。如果对这个类的实例变量发送消息 `performClick:`，action 表示的消息就会被发送给 target 对象。这时，消息的参数使用`performClick:`的参数。

Application 框架利用这种原理实现了 GUI 控件对象间的通信，叫作**目标 – 动作模式**（target-action paradigm）。

图 8-3 目标 - 动作模式



Application 框架的目标 – 动作模式在发送消息时使用了下面这种形式定义的方法，即只有一个 id 类型的参数，没有返回值。这种形式的方法叫作**动作方法**（action method）。

```
- (void) XXXXXX:(id)sender;
```

当用户操作了某个 GUI 控件，例如点击了一个按钮或者移动了滑动条时，action 指定的消息就会被发送给事先设定好的 target，消息的定义如上所示。消息的参数通常都是 GUI 控件的 id。这样一来，消息的接收者 target 就会知道到底哪个控件发送了什么样的消息。

通常使用方法`setTarget:`和`setAction:`来指定目标和动作。在基于引用计数的内存管理方式下，`setTarget:`不会对参数进行 retain 操作。使用 ARC 的时候推荐使用弱引用。

其他 GUI 系统（Windows 或 Linux 的 GUI 系统）中也经常会使用面向过程风格的方法，例如使用回调函数（函数指针）来操作 GUI 控件等。Cocoa 中的 GUI 控件和其接口都是用 Objective-C 实现的。在 NeXTstep 面世 20 多年后的今天，最新版的 Mac OS X 中仍然使用这种面向对象的概念，不得

不让人惊讶。

UIKit 是 iPhone 和 iPad 中用来建立和管理应用程序用户界面的框架。UIKit 和 Application 框架具有同样的概念，但具体细节方面也有很多不一样的地方。Application 框架中的动作方法只有一种格式，UIKit 框架中的动作方法则有以下三种格式。

```
(1) - (void)XXXXXX;
(2) - (void)XXXXXX:(id)sender;
(3) - (void)XXXXXX:(id)sender forEvent:(UIEvent *)event;
```

（2）的动作方法和 Application 框架中的动作方法的定义是一样的。（1）的动作方法没有参数。（3）的动作方法多了一个参数，表示与操作相关的事件信息。如果想了解更多实际的例子和事件的处理方法，请参考“UIControl Class Reference”等文档。

8.2.8 Xcode 中的动作方法和 Outlet 的写法

综合开发环境 Xcode 及其用于设计和测试 GUI 的工具 Interface Builder 中，为了连接对象，在类的接口部分中声明了一些宏，通过这些宏变量能够将代码连接到 nib。

首先，动作方法的声明如下所示，IBAction 是一个宏，被定义为 void。

```
- (IBAction) XXXXXX:(id)sender;
```

这里我们来介绍一下 outlet 的含义。outlet 可以被理解为插座，可以通过 outlet 从控件中取出信息，或将新的信息赋值给控件。Outlet 通常是一个类的实例变量。例如，一个指向 NSButton 类型的控件的实例变量的声明如下所示。

```
IBOutlet NSButton *theButton;
```

这里的 IBOutlet 也是一个宏，被定义为空。另外，还有下面这样一种定义方式。

```
IBOutletCollection(NSButton) NSArray *buttons;
```

这样声明之后，就可以将 Xcode 上面的多个控件（本例中为按钮）都连接到这个 Outlet 上。IBOutletCollection(NSButton) 也是一个宏定义，编译之后会被替换为空。也就是说，上面的语句定义了一个数组的实例变量。在多个控件需要统一处理的时候，这种定义方法会很方便。

属性声明也可以定义为 Outlet，如下所示。

```
@property(weak) IBOutlet NSButton *okButton;
```

Xcode 可以把这个属性声明看作是 Outlet，并用其来连接控件。另外，也可以为这个属性声明增加访问方法。

使用 ARC 进行开发的情况下，要注意避免形成对象之间的引用循环。所以，除了主要的对象之间的连接使用强引用之外，其余的对象之间进行连接时都推荐使用弱引用。属性声明时，建议加上 assign 或者 weak 选项。

8.3 Objective-C 和 Cocoa 环境

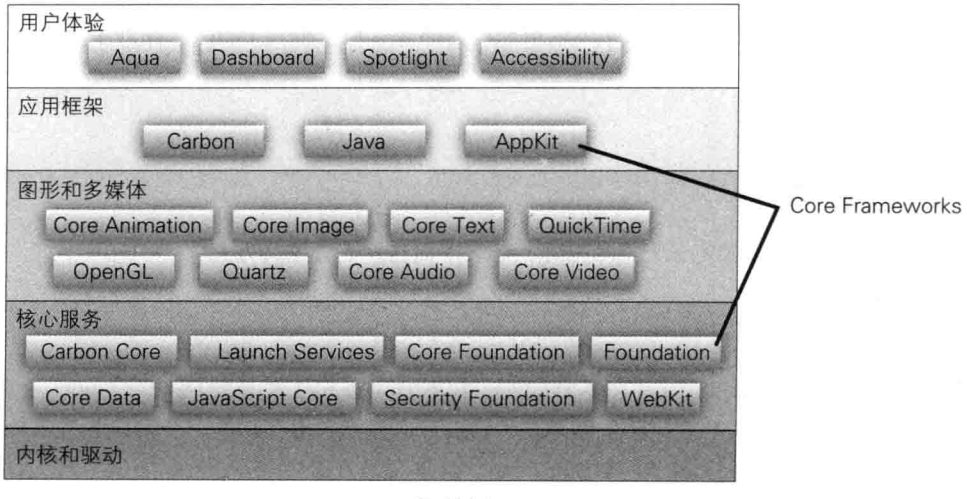
8.3.1 cocoa 环境和 Mac OS X

图 8-4 中展示了 Mac OS X 应用环境的概念。这张图来源于苹果公司文档“Cocoa Fundamentals Guide”，图中说明了 Mac OS X 中各功能组件所处的位置和彼此之间的关系，每个组件都依赖于下一层组件。

我们经常提到的 Cocoa 环境通常是指 AppKit 和 Foundation 这两个核心框架，但有时候也包含 Core Foundation 或 Core Data 等框架。

该文档中强调，对于下层来说，Cocoa 并不是一个简单的面向对象的接口。Cocoa 的前身 OPENSTEP 是跨平台的，能对应多种架构。而 Cocoa 是为了提供构建应用程序所必需的功能而设计的，所以才会利用下层的功能。

图 8-4 Mac OS X 的分层概念



资料来源：Apple Inc.，“Cocoa Fundamentals Guide”等

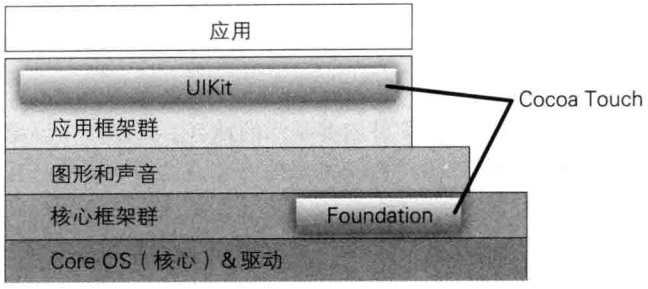
8.3.2 Cocoa Touch 和 iOS

iPhone 和 iPad 的操作系统 iOS 使用 Cocoa Touch 作为 GUI 环境。图 8-5 展示了 iOS X 的架构以

及和 Cocoa Touch 的关系（摘自苹果官方文档“Cocoa Fundamentals Guide”）。

由 Foundation 和 UIkit 框架组合而成的 GUI 环境称为 Cocoa Touch。

图 8-5 iOS 的架构和 Cocoa Touch



8.3.3 框架

在 Mac OS X 中，将开发和执行软件所必需的图形库、头文件和设定用的各种信息全部汇总在一起就构成了**框架**。值得一提的是，其中包括了应用程序执行时所必须的动态链接库。

框架是应用程序的骨架的意思。框架提供了程序运行的基本功能和 GUI 基础。在这些基本功能之上，通过添加独有的处理，就可以实现要实现的功能。

Mac OS X 中最重要的框架是 Foundation **框架**、Application **框架**（也称为 Appkit 框架或 Application Kit 框架）、Core Foundation **框架**和 System **框架**。Foundation 框架提供了包括 NSObject 在内的 Objective-C 的基本类库。Core Foundation 框架是一组 C 语言接口，它们为 iOS 应用程序提供了基本的数据管理和服务功能（详情请参照附录 B）。Application 框架包含了与 Cocoa 的 GUI 的基础——窗口环境相关的类。另外，System 框架包含了与 Cocoa 最底层的 Mach 核心和 Unix 相关的类库。因为系统框架通常都和程序执行相关，所以编译程序的时候不需要指定 -framework 选项。

还有一个 Cocoa **框架**，实际上它是由 Foundation 框架、Application 框架和 CoreData 框架组成的。像这样，将多个框架嵌套打包的技术称为 umbrella framework。不过该技术仅仅是苹果公司提供功能的一种方法，不建议普通开发者提供自己的 umbrella framework。

如图 8-5 所示，iOS 中的 Cocoa Touch 由 Foundation 框架和 UIKit **框架**构成。iOS 的 Foundation 框架和 Mac OS X 的 Foundation 框架有很大一部分是共用的，例如字符串或数组等的基本类。UIKit 是负责用户接口的框架，其中定义了基本的 GUI 控件和用于处理触摸屏幕之类的事件的类。另外，iOS 可以通过利用 Core Foundation 框架中的数据结构提供面向其他框架或设备的功能。关于 Core Foundation 框架的概要，请参考附录 B。

8.3.4 框架的构成和头文件

不同版本的 Mac OS X 的目录构成可能会有所不同，但系统提供的框架一直都在 /System/Library/

Frameworks 下面。应用程序用的框架有时会被放到 /Library/Frameworks 等目录中。应用程序专用的框架也会被放到应用程序的包中。

例如，Foundation 框架和 Application 框架的存放路径分别为

```
/System/Library/Frameworks/Foundation.framework
/System/Library/Frameworks/AppKit.framework
```

安装了 iOS 开发环境的 Mac 中，iOS 框架在开发环境相关文件内部很深的地方。根据开发环境版本和安装位置的不同，框架的安装路径也不尽相同，iOS 5.0 中框架的默认安装路径如下所示。真机和模拟器因为使用的 cpu 不同，所以会使用不同的框架。

```
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/
iPhoneOS5.1.sdk/System/Library/Frameworks
```

框架的每个目录中通常都包含了以下要素。

1. **类库**——框架同名文件
2. CodeResources——记载了文件散列值的 XML 文件
3. Headers——包含了头文件的目录
4. Resources——包含了不同国家语言用的文件等各类资源的目录
5. Versions——包含了框架的各种版本的目录

Mac OS X 中，一般情况下 1-4 都是指向 5 内部的符号链接。iOS 中，各种资源文件则没有被放到 Resources 目录中，而是被直接放到了框架的目录下面。

在至今为止的例子中，我们都是使用 `#import<Foundation/NSObject.h>` 这样的语句来加载框架的头文件。但实际上 NSObject.h 并不在 Foundation 目录下。加载头文件的时候只要使用了 `#import < 框架名 / 头文件名 >` 这种格式，编译器（更确切的说法是预处理）就会在对应的框架中查找到要加载的头文件，并加载。

一般来说，使用了 NSString 的程序需要在实现文件中加载 Foundation/NSString.h，使用了 Application 框架的 NSView 的程序需要加载 AppKit/NSView.h。但如果一个程序使用了各种各样的类，就需要加载很多头文件。

Foundation 框架中有 Foundation/Foundation.h 这样一个头文件。只引入了这个文件，就相当于引入了 Foundation 框架中所有类的头文件。使用 Foundation 框架的程序只要加载了这个头文件，就不需要再分别加载头文件了。同样，也存在 AppKit/AppKit.h、UIKit/UIKit.h 等头文件。

Mac OS X 中还有一个头文件 Cocoa/Cocoa.h。编写使用了 GUI 功能的应用程序时，只要加载了 Cocoa/Cocoa.h，就相当于加载了 Foundation 框架、Application 框架和 Core Data 框架的所有头文件，非常方便。本书中没有详细介绍 Core Data 框架的内容，有兴趣的读者请参见苹果公司的官方文档。

8.4 全新的运行时系统

8.4.1 对 64 位的对应和现代运行时系统

Mac OS X 从 10.7 Lion 开始支持 64 位系统，在此之前还更换了新的运行时系统。64 位的运行时系统，并不是仅仅在 32 位系统的基础上增加了 64 位的支持，还增加了其他的新功能。iOS 4.0 以后使用的运行时系统就是新版本的运行时系统。运行时系统分为早期版本（Legacy）和现代版本（Modern），苹果公司的官方文档中把 64 位的 Mac OS X 的运行时系统和 iOS 的运行时系统称为**现代运行时系统**，把 32 位 Mac OS X 的运行时系统被叫作**早期运行时系统**。

下面说明一下 64 位环境下编程时的注意事项和现代运行时系统的新功能。

8.4.2 数据模型

和很多 UNIX 系统一样，Mac OS X 使用 ILP32 和 LP64 作为编程的数据模型。ILP32 的含义是 int 类型、long 类型和指针类型都是 32 位。LP64 的含义是 long 类型和指针类型都是 64 位。16 位电脑的数据模型是 LP32，当前的 Windows 系统是 LLP64（long long 类型和指针类型都是 64 位）。表 8-2 中展示了数据模型和 C 语言数据类型之间的关系。ILP32 中没指定 long long 类型的位数，Mac OS X 中将其定义为了 64 位。

C 编译器可通过选项 -m32 或 -m64 来指定使用哪种数据模型生成代码。详情请参考 16.1 节中介绍的通用二进制（Universal Binary）的内容。

表 8-2 典型的数据模型和 C 语言数据类型的位数

	char	short	int	long	long long	pointer
LP32	8	16	16	32		32
ILP32	8	16	32	32	(64)	32
LP64	8	16	32	64	64	64
LLP64	8	16	32	32	64	64

8.4.3 64 位模型和整数类型

数据类型发生了变化的话，主要受影响的是调用 API 时的参数和返回值的类型。

ILP32 和 LP64 中的 int 类型都是 32 位，这种情况下，就算数据类型变更到了 64 位，因为当前程序使用的整数类型还是 32 位，所以也不能利用到 64 位的优点。为了解决这个问题，Cocoa 环境引入了 NSInteger 类型，NSInteger 在 32 位的数据模型下被定义为 int，在 64 位数据模型下被定义为 long。除此之外，Cocoa 中还定义了无符号的 NSUInteger 类型。Cocoa API 中参数为 int 类型的函数

或方法绝大多数都使用了 NSInteger 替代 int。

NSInteger 和 NSUInteger 都在头文件 NSObjCRuntime.h 中定义。宏 __LP64__ 在 64 位编译时为真。宏 TARGET_OS_IPHONE 在面向 iOS 真机编译时为真。宏 TARGET_IPHONE_SIMULATOR 表示面向模拟器编译。

```
#if __LP64__ || !TARGET_OS_IPHONE /* 省略了一部分代码 */
  typedef long NSInteger;
  typedef unsigned long NSUInteger;
#else
  typedef int NSInteger;
  typedef unsigned int NSUInteger;
#endif
```

使用 NSInteger 类型编程的时候有一个问题，就是如何在 printf 和 scanf 中指定格式化字符串（64 位系统中 NSInteger 是 long，格式化字符串应该指定“%ld”；32 位系统中 NSInteger 是 int 格式化字符串应该指定“%d”）。字符串类 NSString 中也有和 printf 一样的生成格式化字符串的方法。但真正的程序中很少会用到 scanf。可以通过下面这样的类型转换来解决 printf 的问题。

```
NSInteger n = ...;
printf("Input=%ld\n", (long)n);
```

如果因为程序中大量使用了 scanf 而需要根据不同的数据模型来实现不同版本的代码的话，可以用下面这种办法。这种方法来自苹果公司的文档，使用宏定义把两个版本的代码整合在了一起。图 8-6（a）时宏的定义需要在代码的最开始声明，图 8-6（b）是实际的使用。看起来好像有点麻烦，其实就是通过使用宏定义动态设置 scanf 和 printf 的参数。头文件 /usr/include/inttypes.h 中也定义了这样的宏。

图 8-6 使用宏来指定 NSInteger 的格式

```
#if __LP64__
# define FMTd  "ld"
#else
# define FMTd  "d"
#endif
```

（a）格式字符串的宏定义

```
NSInteger n;
scanf("%" FMTd, &n);
printf("Input=%" FMTd "\n", n);
```

（b）格式字符串中使用宏

8.4.4 Core Graphics 的浮点数类型

实数类型同数据模型无关，如 float 和 double 分别是 32 位和 64 位，不会发生改变。gcc 4.0 之后 long double 变为了 128 位，但这也和是否为 64 位数据模型无关。

在与 Mac OS X 和 iOS 的图形相关的 Core Graphics 框架（Quartz）中，窗口和 GUI 控件的位置、

大小等都使用了 32 位的 float 类型，但随着数据模型变为 64 位，原来用的 32 位的 float 类型也就被变为了 64 位的 double 类型。Core Graphics 框架中引入了 CGFloat 类型，据此就不用再考虑实际使用的是 float 类型还是 double 类型了。现在，在 Application 框架和 UIKit 框架中，当画图、图像、坐标等相关的接口中使用到实数的时候，都会使用 CGFloat 类型。

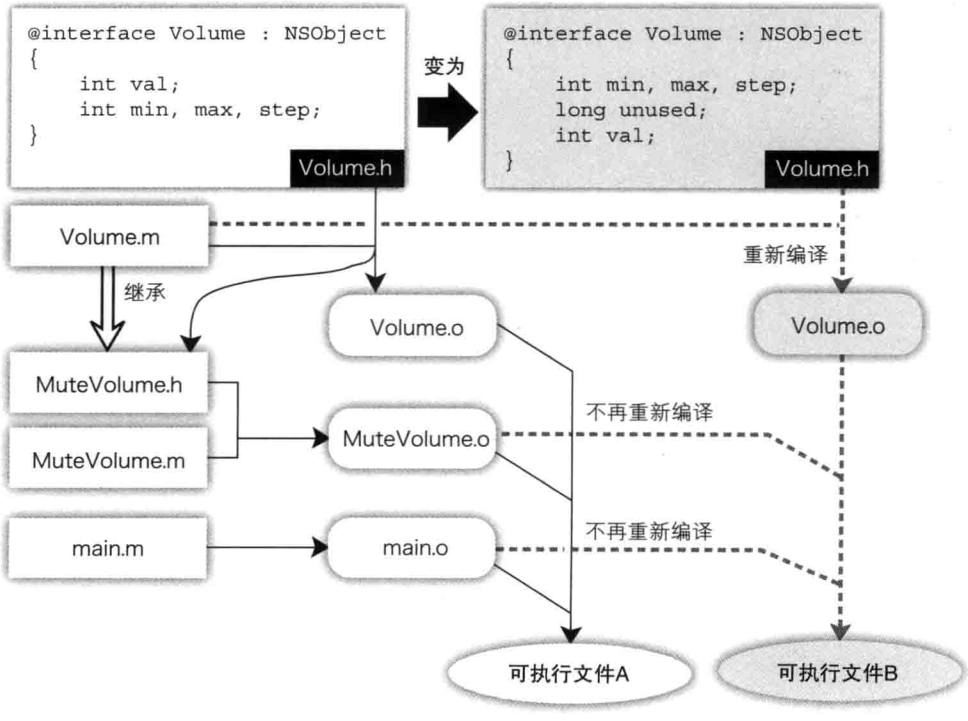
scanf 或 printf 操作 CGFloat 类型的数据时，也可以使用上面介绍的那种宏定义的方法。

8.4.5 健壮实例变量

和早期运行时系统相比，现代运行时系统的一个最显著的特征就是实例变量是**健壮的**（non-fragile）。下面我们用第三章中讲过的带有静音功能的音量类来说明一下。

让我们看一下图 8-7，编译源代码文件生成 Volume.o、MuteVolume.o 和 main.o，链接这些 .o 文件生成可执行文件 A。可执行文件 A 可以正常执行。接着，编辑 Volume.h，改变其中的实例变量的顺序并增加新的实例变量。这里重新编译生成 Volume.o，并链接 Volume.o 和原有的 MuteVolume.o 和 main.o 生成可执行文件 B。

图 8-7 改变实例变量的顺序



类 MuteVolume 继承自类 Volume，MuteVolume.o 中理应保存着变更之前的实例变量。实际上，在 32 位数据模型下编译链接生成的可执行文件 B，在早期的运行时系统中是无法正常执行的。而在

现代运行时系统中，可执行文件 B 和可执行文件 A 一样可以正常运行。

让我们考虑一下应用程序中定义了框架中类的子类时的情况。如果框架中的类发生了变化，应用程序就一定要重新编译才能够在早期的运行时系统中执行，这种问题称为**脆弱的二进制接口**。

现代运行时系统针对这个问题进行了改进，使实例变量发生变化的情况下，应用程序不重新编译也能够继续执行。这称为**健壮的实例变量**。但实例变量的改变仅限于图 8-7 中所示的变量顺序的变化或增加了新的实例变量这种程度，删除了实例变量或改变了实例变量的类型时还是需要重新编译才能够正常运行。

旧版的 Objective 中存在一个 @defs 关键字，被用于返回一个 Objective-C 类的 struct 结构，这个 struct 与原 Objective-C 类具有相同的内存布局。随着健壮实例变量的导入，Objective-C 2.0 中停止了使用这个关键字。

专栏：条件编译 COLUMN

ANSI C 中可以通过使用预处理指令来实现条件编译。当条件编译的条件变复杂时，嵌套的层数会加深，变得不好理解，这种情况下，可以通过使用条件表达式来回避这个问题。

另外，条件表达式中可以使用 defined 运算符，如果该运算符的参数宏已经定义了的话就返回真。通过使用这种方法可以让复杂的条件判定写起来更简单。下面就是一个例子，defined 定义的参数的括号可以省略。

```
const char *const version = "Ver. 3.2"
#if defined(__i386__)
    " for Intel"
#elif defined(__ppc__)
    " for PowerPC"
#endif
#if defined(__APPLE__) || defined(__NEXT__)
    " (powered by OPENSTEP)"
#endif
    ;
```

第9章

Foundation框架中常用的类

本章将首先讲述对象可变性的概念，然后再介绍 Foundation 框架中常用的字符串类、数据类、数组类和词典类等内容。Foundation 框架中其他类的介绍、函数和类型定义等内容请参考附录 A。

9.1 对象的可变性

9.1.1 可变对象和不可变对象

Objective-C 中的类分为**可变**（mutable）类和**不可变**（immutable）类。可变类的实例对象称为可变对象，指的是创建后能够改变其内容或状态的对象。不可变类的实例对象称为不可变对象，指的是创建后不可更改其内容的对象。对象是否可变的属性称为**可变性**（mutability）。

例如，字符串类就分为可变字符串类 NSString 和不可变字符串类 NSMutableString。表 9-1 中成对列出了 Foundation 框架中主要的可变类和不可变类。

表 9-1 Foundation 中常用的可变类和不可变类的组合

种类	不可变类	可变类
数组	NSArray	NSMutableArray
数据	NSData	NSMutableData
词典	NSDictionary	NSMutableDictionary
集合	NSSet	NSMutableSet
字符串	NSString	NSMutableString
带属性的字符串	NSAttributedString	NSMutableAttributedString
字符集合	NSCharacterSet	NSMutableCharacterSet
索引集合	NSIndexSet	NSMutableIndexSet

为什么要对数据类型区分可变类和不可变类呢？首先，这么做的原因是从实现复杂度方面考虑的。例如，如果要允许删除字符串中的某个字母，或者往数组中追加几千个对象的话，就必须在一开始就设计好这些操作所对应的数据结构和方法。与此相反，如果确定字符串或数组在程序中不会发生变化，那么就不用为它们设计复杂的数据结构，处理起来也会变得容易很多。

第二个原因是和对象处理的概念有关，让我们来看看下面这条语句。

```
[a add:b]
```

这条语句看上去像是 a 加 b，那么该如何解释这条语句的具体含义呢？根据对象类型的不同，这条语句可被解释为不同的意义。例如，a 和 b 是字符串时会有一种解释方法，a 和 b 是集合时又会有另外一种解释方法。根据 a、b 的类型的不同，a 的值有可能会变化，也有可能会返回新的计算结果。

根据对象可变性的不同（不可变的值对象或可变的属性对象），上面这条语句也会有不同的解释。例如，如果对象 a 和 b 是集合对象的话，这条语句的含义就既可以是 a 不变，返回一个新的集合——a 和 b 的并集；又可以是 a 可变，把集合 b 中的元素加到 a 中，并返回 a。前者的情况下 a 是不可变对象，后者的情况下 a 是可变对象。

不可变对象的概念在并行编程中很重要。因为可变对象的值可能会在不知不觉中被其他线程所

修改。但不可变对象就不用担心这方面的问题。详情请参考第 19 章并行编程的内容。

在本书中至今为止所列举的例子中，第 5 章中的分数类可被看为不变对象，第 4 章的三原色类可被看为可变对象。

下面是使用不可变类 NSString 编程的一个例子。假设tempDirectory方法会返回一个 NSString 的对象，是目标文件的目录。这里，我们将这个目录中的 filename 文件的扩展名更改为“bak”，如下所示。

```
NSString *str = [self tempDirectory];
NSString *work = [filename lastPathComponent];
        // 从完全路径中获取文件名
work = [work stringByDeletingPathExtension];
        // 删除文件的扩展名
work = [work stringByAppendingPathExtension:@"bak"];
        // 给文件加上扩展名“.bak”
str = [str stringByAppendingPathComponent:work];
        // 重新生成新的路径
```

例如，如果方法tempDirectory返回的路径是 /home/you/tmp，变量 filename 是 /usr/me/fine.doc 的话，上面的程序执行之后，str 的值就为 /home/you/tmp/fine.bak。

上面这段程序中，有多个地方用到了 NSString 对象，就好比数值计算的中间结果一样。

下面让我们看一个可变对象 NSMutableArray 的例子。方法nextEntry从文件逐行读入，每读入一行信息就生成一个类 MyInfo 的实例对象。到达文件末尾的时候返回 nil。到底生成了多少个类 MyInfo 的对象同具体文件相关，这里把生成的所有实例对象都放入数组对象 anArray 中。

```
MyInfo *entry;
NSMutableArray *anArray = [NSMutableArray array];  // 生成空的数组
while ((entry = [self nextEntry]) != nil) {
    [anArray addObject: entry];
        // 在数组对象的末尾追加 entry
}
```

使用方法addObject后，并不能生成新的对象，而是原来的数组对象 anArray 中会被添加新的元素。NSMutableArray 的实例对象 anArray 在初始时一个对象都没有，会随着对象的加入被自动扩展。

上面的例子也可以使用不可变数组 NSArray 来实现，但这种情况下，每当数组中的元素发生变化时，都需要新建一个数组对象并把元素重新复制进去，非常麻烦。通过这个例子可以看出，根据实际情况的不同来区分使用可变和不可变对象是十分重要的。

9.1.2 可变对象的生成

表 9-1 中列举了常用的可变类和与其对应的不可变类，其中可变类是不可变类的子类，例如类 NSMutableArray 就是类 NSArray 的子类，所以可变类的实例对象可直接作为不可变类的实例对象来使用。

与此相反，如果想把不可变类的实例对象作为可变类的实例对象来使用的话，该如何操作呢？可以使用mutableCopy方法为不可变对象创建一个可变的副本。这个方法定义在 NSObject 中，只要是成对出现的可变类和不可变类的对象，就都可以使用这个方法。更多详细内容请参考第 13 章。

- (id) **mutableCopy**

给不可变对象发送mutableCopy消息的话，会生成一个和不可变对象内容一致的可变对象。使用基于引用计数的内存管理方式时，新生成的实例对象的所有者就是mutableCopy消息的发送者（关于引用计数的更多内容请参考 5.2 节）。

9.2 字符串类 NSString

9.2.1 常量字符串

Objective-C 和 C 语言一样也在程序中提供了定义字符串对象的方法，即将想要表示的字符串用""括起来，并在开头加上 @，如下所示。

```
NSString *myname = @"T. Ogihara";
NSString *work = [@"Name: " stringByAppendingString: myname];
```

使用这种方式定义的字符串是**常量对象**（constant object），可被作为 NSString 的对象使用。常量字符串不仅仅可作为消息的参数，还可以作为消息的接收者。

@"" 表示一个长度为 0 的空字符串，但和 nil 有区别。另外，和 ANSI C 的字符串常量一样，编译器编译时会把用空格分割的字符串连在一起，如下所示。

```
#define  Manufacturer  @"Phantom Cookie, Inc."
#define  Year          @"2005"
NSString *note = @"Copyright " Year @" " Manufacturer;
```

Objective-C 的常量字符串除了支持 ASCII 编码的字符外，和 C 语言一样也使用 (\n) 表示换行，使用 (\t) 表示制表符。

NSString 还可以处理 Unicode。字符串常量也可以包含“\u + 四位 16 进制数”或“\U + 八位 16 进制数”这种 UTF-16 格式的通用字符（universal character）。

```
NSString *euro = @"Euro=\u20ac";          // 表示欧元的符号
NSString *clef = @"G-clef=\U0001d11e";    // G 高音谱号
```

关于对应多语言的应用程序的开发方法请参考 16.5 节的内容。源代码中的字符串常量要尽量避

免使用日语或其他特定的语言。

字符串常量从程序的执行开始到终止一直存在，调用release方法或者垃圾回收都不能释放字符串常量。

9.2.2 NSString

本节将说明 Coco 环境下的字符串类 **NSString** 的概要。NSString 代表不可变的字符串对象，一旦 NSString 被创建，我们就不能改变它。你可以对它执行各种各样的操作，例如用它生成新的字符串、查找字符串或者将它与其他字符串进行比较等，但是不能通过增加删除字符来改变它。NSString 和 NSMutableString 的实例称为**字符串对象**或者字符串。字符串对象内部使用 Unicode 编码。

可能有人会想：C 语言中已经有字符串了，为什么还要再定义一个字符串对象呢？首先，通过将字符串作为对象来处理，可以统一很多操作。例如，数组（NSArray）和词典类（NSDictionary）中都要求存放对象。NSString 的实例对象能够回答isKindOfClass:之类的查询，作为对象来处理的话生成和释放的逻辑也都相同。另外，NSString 中还定义了很多常用的方法，例如将字符串结合、取出部分子符串、把字符串作为路径来处理等。除此以外，可变字符串对象的情况下，编程的时候还可以不用关心字符串的长度。Cocoa 的 API 中涉及字符串的时候，都使用了 NSString 对象。

除了一些特殊操作需要使用到 C 语言风格的字符串外，绝大多数的工作都可以通过 NSString 来完成，因此，定义字符串类可以大大提高开发效率。

NSString 主要的方法都在 Foundation/NSString.h 中定义。由于 NSString 是以类簇的方式（class cluster，可以看作是“工厂类”，提供了很多方法接口，但是这些方法的实现是由具体的内部类来实现的）实现的，因此不能用通常的方法来为 NSString 定义子类。更多内容请参考第 11 章中抽象类和类簇的内容。

下面对 NSString 的主要方法进行说明，其他 NSString 的方法请参考苹果公司的官方文档。

在初始化方法的说明中，使用alloc和init的组合生成并返回实例的类方法（引用计数的内存管理方式下还包括自动释放）都被标记为了便利构造函数。更多详细内容请参考 5.2 节。

（1）操作 Unicode 编码的字符串

NSString 中的汉字都是用 Unicode 来表示的。Unicode 的 UTF-8 编码兼容 ASCII 的 7bit 编码（主要是英文字母和一些符号），字符串中只含有 ASCII 的 7bit 范围的编码的情况下，ASCII 的字符串可被当作 UTF-8 的 Unicode 字符串来处理。ASCII 的字符串不能够被当作 UTF-16 的字符串来处理。苹果电脑中的 Finder 使用 UTF-8 来表示日语文件名。

下面提到的 **unichar** 是两字节长的 char，代表 unicode 的一个字符，和 char 有所不同。

```
- (id) initWithUTF8String: (const char *) bytes
```

从以NULL结束的UTF-8编码的C字符串中复制信息，并初始化接收者。
便利构造器：stringWithUTF8String:

```
- (__strong const char *) UTF8String
```

返回编码为UTF、以NULL结尾的C语言字符串的指针。基于引用计数的内存管理模式下，返回的

字符串会在消息接收对象被释放的同时被释放掉，如果想在对象被释放之后也能使用，就需要额外复制一份。垃圾回收的内存管理模式下，因为返回的指针是强指针，所以不会被垃圾回收。

- (NSUInteger) **length**

返回字符串中Unicode编码的字符的个数。和C风格的字符串不同，不能用这个函数的返回值来计算需要的字节数和表示时需要的长度。

- (unichar) **characterAtIndex:** (NSUInteger) index

返回索引位置为i处的字符。编码为Unicode编码。

- (id) **initWithCharacters:** (const unichar *) characters
length: (NSUInteger) length

用characters中存储的length长的字符串来初始化并返回一个NSString对象，字符串的编码是Unicode编码。初始化时以length为准，characters中就算包括了'\0'也不能做为终止标记。
便利构造器：stringWithCharacters:length

- (void) **getCharacters:** (unichar *) buffer
range: (NSRange) aRange

NSRange是一个表示范围的结构体，其中包括数据的首指针和数据的长度。这个函数的作用就是将aRange所表示的字符串作为Unicode字符串写入缓冲区buffer中，末尾不自动添加NULL，缓冲区需要足够大。

(2)编码转换

C风格或字节类型的字符串和NSString之间可以相互转换。字符串编码定义为枚举类型NSStringEncoding，保存在NSString的头文件中，其中最常用的几个类型如下所示。更完整的定义请参见附录A。

NSASCIIStringEncoding	7bit的ASCII编码
NSUTF8StringEncoding	8bit的Unicode编码（UTF-8）
NSMacOSRomanStringEncoding	Mac OS用的编码
NSJapaneseEUCStringEncoding	8bit的日文EUC编码
NSShiftJISStringEncoding	8bit的日文Shift-JIS编码

- (id) **initWithCString:** (const char *) nullTerminatedCString
encoding: (NSStringEncoding) encoding

用C风格字符串初始化一个NSString对象，字符串nullTerminatedCString要求以NULL结尾，nullTerminatedCString的编码为encoding。
便利构造器：stringWithCString:encoding

- (__strong const char *) **cStringUsingEncoding:** (NSStringEncoding) encoding

返回消息接收对象的C风格字符串，编码由encoding指定。引用计数的内存管理模式下，返回的字符串会在消息接收对象被释放的时候同时被释放掉，如果想在对象被释放之后也能使用的话，就需要额外复制一份。垃圾回收的内存管理模式下，因为返回的指针是强指针，所以不会被回收（详情请参考6.2节的内容）。
如果字符串无法转换为指定编码，编码时会抛出异常。
使用方法getCString:maxLength:encoding:能够向准备好的内存空间写入C风格的字符串。

- (id) **initWithData:** (NSData *) data
  encoding: (NSStringEncoding) encoding

  用存储在data（data是NSData类型的对象，关于NSData的更多内容请参考9.3节）中的二进制数据来初始化NSString对象。data中二进制数据的编码是encoding，返回的NSString对象的编码是Unicode。
  同样，通过方法initWithBytes:length:encoding:能够用指定内存中的二进制数据来初始化一个NSString对象。

- (NSData *) **dataUsingEncoding:** (NSStringEncoding) encoding

  将接收消息的NSString对象的内容用encoding指定的方法编码，并将结果存储到一个NSData对象中并返回，如果编码转换失败则返回nil。
  使用方法lengthOfBytesUsingEncoding:能够返回NSString被编码之后所占的bytes数。

- (BOOL) **canBeConvertedToEncoding:** (NSStringEncoding) encoding

  测试接收消息的NSString对象能否转换为encoding编码。
  使用类方法availableStringEncodings能够返回当前环境下可用的编码方式。

- (NSString *) **stringByAddingPercentEscapesUsingEncoding:**
  (NSStringEncoding) encoding

  能够对一些特殊字符进行替换，主要被用于处理URL字符串，比如将空格替换成%20等。详细内容可参考9.7节。
  方法stringByReplacingPercentEscapesUsingEncoding:是上述方法的逆变换。这两个方法都定义在Foundation/NSURL.h中。

（3）生成指定格式的字符串

NSString 中有和 C 语言中的 printf() 一样功能的函数，能够生成指定格式的 NSString 对象。区别在于指定目标格式的格式化字符串本身也是一个 NSString 对象，指定格式时还可以使用 %@，所以要特别注意类型方面的处理。

%@ 对应的参数必须是一个对象。当输出 %@ 时（例如：NSLog(@"%@" , test)），实际上就是调用了 test 的 description 方法，返回的结果存储到 %@ 中。description方法是 NSObject 中定义的方法，任何对象都可以调用它。description返回的内容和类实现相关。

C 语言的 printf() 中，编译器支持参数类型的自动转换，例如，对表示实数的 %f 传入一个整数也是可以的。而 NSString 则不支持这种参数类型的自动转换，所以一定要保证格式化字符串中指定的类型和所传入的参数的类型一致。

- (id) **initWithFormat:** (NSString *) format, ...

  根据format中指定的格式串来生成一个字符串，使用这个字符串来初始化消息接收者。format中的参数用逗号分隔，格式字符串不能为空。
  便利构造器：stringWithFormat:

和 C 语言一样，Ojective-C 也支持可变参数的方法，如上所示，在参数的末尾加上逗号，然后再加上 ... 即可。

如果要自定义可变参数的方法，可参考 10.2 节的内容。

（4）NSString 的比较

下面对用于比较 NSString 的方法进行说明。

字符串比较时会返回一个 NSComparisonResult 类型的值。NSComparisonResult 是 enum 型数据，共有 3 个值，分别为 NSOrderedAscending（左侧小于右侧）、NSOrderedSame（两者相同）、NSOrderedDescending（右侧小于左侧）。这些方法还可以用在对多个 NSString 变量排序。更多详细内容请参考类 NSArray 的方法`sortedArrayUsingSelector:`的介绍。

- `(NSComparisonResult) compare: (NSString *) aString`

  比较消息的接收者和参数字符串aString，参数aString不可以为nil。
  如果想比较两个字符串的内容是否相同，除了`compare:`外还可以使用下面将介绍的方法`isEqualToString:`。

- `(NSComparisonResult) caseInsensitiveCompare: (NSString *) aString`

  `compare:`进行的是区分大小写的比较，`caseInsensitiveCompare:`进行的是不区分大小写的比较。
  除了使用`caseInsensitiveCompare:`外，也可以使用方法`compare:options:`来实现不区分大小写的比较，需要用或运算(|)来为option参数添加选项标记，不区分大小写比较的选项为NSCaseInsensitiveSearch:。

- `(NSComparisonResult) localizedStandardCompare: (NSString *) aString`

  按照Mac系统Finder的排序规则进行比较操作。通常对数组中的文件进行排序时，有可能希望文件的排序规则和Finder的排序规则一致，这时便可以使用方法`localizedStandardCompare:`。

- `(BOOL) isEqualToString: (NSString *) aString`

  比较消息的接收者和参数aString是否相等。

- `(BOOL) hasPrefix: (NSString *) aString`

  检查字符串是否以astring开头。
  可以使用`hasSuffix:`来判断消息接收者是否以参数字符串结尾。
  另外还可以使用方法`commonPrefixWithString:options:`来取出消息接收者和参数字符串开头部分相同的字符串。

（5）为字符串追加内容

- `(NSString *) stringByAppendingString: (NSString *) aString`

  在接收者字符串后面追加字符串aString，返回一个新的字符串。

- `(NSString *) stringByAppendingFormat: (NSString *) format, ...`

  在接收者字符串后面追加格式字符串，字符串的具体格式由format指定，然后返回一个新的字符串。

（6）截取字符串

截取指定的字符串并返回。使用结构体 NSRange 来表示要截取的字符串的开始位置和长度，更多关于 NSRange 的内容请参考附录 A。

- (NSString *) **substringToIndex:** (NSUInteger) anIndex

返回一个新的字符串，新字符串的范围是从接收者字符串的第一个字符开始到anIndex结束，anIndex不包含在内。

- (NSString *) **substringFromIndex:** (NSUInteger) anIndex

返回一个新的字符串，新字符串的范围是从anIndex开始一直到结尾，anIndex位置的字符也包含在内。

- (NSString *) **substringWithRange:** (NSRange) aRange

返回一个新的字符串，新字符串的开始位置和长度由aRange来指定。

（7）检索和置换

- (NSRange) **rangeOfString:** (NSString *) aString

在接收者字符串中查找aString，如果能找到，就将aString的位置和长度以NSRange的形式返回。如果没有找到，则返回一个位置为NSNotFound、长度为0的NSRange类型的对象。

还有一个方法rangeOfString:options:，这个方法带有选项参数。例如，通过指定选项NSCaseInsensitiveSearch，就可以进行不区分大小写的查找。

- (NSRange) **lineRangeForRange:** (NSRange) aRange

返回范围aRange所在行的范围。这里的行是用表9-2中的字符作为结尾标志的。

表 9-2 标识结尾的字符

符号	说明
\r 或 0x0d	CR、Mac OS 上用的换行符
\n 或 0x0a	LF、Unix 上用的换行符
\r\n	CRLF、Windows 用的换行符
U+2028	Unicode 的换行符
U+2029	Unicode 的分割符

- (NSString *) **stringByReplacingCharactersInRange:** (NSRange) range
  withString: (NSString *) replacement

将range范围内的内容替换为字符串replacement。

- (NSString *) **stringByReplacingOccurrencesOfString:**
  (NSString *) target
  withString: (NSString *) replacement

将字符串target替换为字符串replacement。

除了以上几种查找方法外，检索时还可以使用**正则表达式**。例如，查找以大写字母开头的5个字符以内的英文或数字字符串。正则表达式的写法请参考类NSRegularExpression的参考文档[①]。

可以使用选项NSRegularExpressionSearch来完成查找、替换等操作。例如，下面这个例子就是利用正则表达式来查找邮政编码（假设邮政编码的格式是：3个数字-4个数字，例如333-0852）。

① 本书中没有详细介绍正则表达式的使用方法。正则表达式的写法也可以通过Unix命令grep或脚本语言awk和perl来学习。虽然具体细节方面略有不同，但基本概念都是一样的。

```
[str rangeOfString:@"[0-9]{3}+-[0-9]{4}+"
           options:NSRegularExpressionSearch];
```

（8）大小写的处理

可以使用lowercaseString方法将字符串中所有的大写字母都转换为小写字母，与此相对，uppercaseString则被用于将所有的小写字母都转为大写字母。

除了这两个方法之外，方法capitalizedString能够将所有单词的首字母变为大写，其余字母变为小写。

（9）数值转换

方法doubleValue可把NSString类型的字符串转为double类型的数值。除此之外，方法floatValue、intValue、integerValue、boolValue分别被用来把NSString类型的字符串转为float、int、NSInteger和BOOL类型的数值。以上这些函数都会忽略字符串前面的空格。另外，当输入字符串的首字母是Y、y、T、t中的任何一个，或者是不以0开头的数字时，方法boolValue都会返回YES。例如，“001”、“Yeh!”时都会返回YES。

（10）路径的处理

文件的路径可用NSString来表示，NSString中提供有常用的处理文件路径的方法。这些方法的接口都定义在"Foundation/NSPathUtilities.h"中。

以文件路径名@"/tmp/image/cat.tiff"为例，tmp、image和cat.tiff称为构成路径的要素。本例中文件的扩展名是tiff。Unix中使用"/"作为路径的分隔符。也有方法可以处理用URL表示的文件路径，路径的分割符是"/"。

NSHomeDirectory()能够访回当前用户的主目录，更多内容请参考附录A。

- (NSString *) **lastPathComponent**

  提取文件路径中最后一个组成部分。上面的例子的情况下会返回"cat.tiff"。

- (NSString *) **stringByAppendingPathComponent:** (NSString *) aStr

  将aStr加到现有字符串的末尾并返回，根据需要会自动追加分割符。

- (NSString *) **stringByDeletingLastPathComponent**

  删除路径中最后一个组成部分，如果返回的结果不是根路径，那么最后的路径分割符也会被删除。文件路径@"/tmp/image/cat.tiff"的情况下，调用stringByDeletingLastPathComponent之后返回的结果是"/tmp/image"。

- (NSString *) **pathExtension**

  返回文件的扩展名。扩展名不包含"."，如果没有扩展名则返回空字符串。

- (NSString *) **stringByAppendingPathExtension:** (NSString *) aStr

  将"."和指定的扩展名添加到现有路径的最后一个组成部分上。

- (NSString *) **stringByDeletingPathExtension**

  删除文件的扩展名(包括".")。如果文件没有扩展名，则返回原来的字符串。

- (BOOL) **isAbsolutePath**

判断路径是不是一个绝对路径，如果是则返回YES。

+ (NSString *) **pathWithComponents:** (NSArray *) components

使用components中的元素来构建路径，结合的时候自动添加路径分割符"/"。想生成一个绝对路径的话，数组中的第一个元素使用@"/"。路径的最后想以路径分割符"/"结尾的话，数组的最后一个元素使用空字符串@""。

- (NSArray *) **pathComponents**

和pathWithComponents正好相反，把接收者作为路径名来解析，并将路径的各个组成部分放入数组中。输入的路经是绝对路径时，数组的第一个元素是@"/"。

- (NSString *) **stringByExpandingTildeInPath**

消息的接收者被看作路径名，如果路径的第一个字符是代字符(以~开头的，例如~/或~john/)，则返回用户主目录的路径字符串。如果不以代字符开头，则直接返回输入的字符串。`stringByAbbreviatingWithTildeInPath`和`stringByExpandingTildeInPath`的功能正好相反，会把标准格式的字符串转换为使用代字符的字符串。

- (__strong const char *) **fileSystemRepresentation**

返回路径的C风格字符串，使用当前系统的编码。获得的C风格字符串可被用于系统调用等。基于引用计数的内存管理中，返回的C语言风格的字符串会和消息的接收者对象一起被释放。垃圾回收的内存管理方式下，如果返回值为强引用，则不会被垃圾收集器回收。

使用方法`getFileSystemRepresentation:maxLength:`可以将返回的C风格字符串写入事先准备好的内存中。

(11)文件的输入和输出

可以从文件中读取字符串的内容，也可以将字符串的内容输出到文件中。另外，下面这些同文件相关的方法都带有一个error参数，并把发生错误时的错误信息写入到error中。error不能为NULL，否则就不会返回任何错误信息。更多关于NSError的信息请参考18.5节。

除了使用字符串外，还可以使用NSURL来描述文件的路径。更多关于NSURL的信息可参考9.7节的内容。

- (id) **initWithContentsOfFile:** (NSString *) path
encoding: (NSStringEncoding) enc
error: (NSError **) error

通过读取文件path中的内容来初始化一个NSString，文件的编码为enc。读取文件失败的时候会释放调用者，并在返回nil的同时将详细的错误信息设定到error。

便利构造器:`stringWithContentsOfFile:encoding:error:`

- (id) **initWithContentsOfFile:** (NSString *) path
usedEncoding: (NSStringEncoding *) enc
error: (NSError **) error

和上一个方法一样，通过读取文件path中的内容来初始化一个NSString。不同的地方在于这个函数能够自动判别文件的编码，并通过enc返回。文件的编码是通过文件的内容和文件的扩展属性(extended attribute)来判断的。更多关于扩展属性的内容请参考命令xattr。

便利构造器:stringWithContentsOfFile:usedEncoding:error:

```
- (BOOL) writeToFile: (NSString *) path
          atomically: (BOOL) useAuxiliaryFile
            encoding: (NSStringEncoding) enc
               error: (NSError **) error
```

用于将字符串的内容写入到以path为路径的文件中，写入的时候使用enc指定的编码，写入成功则返回YES。

useAuxiliaryfile为YES的情况下，会首先新建一个临时文件，把字符串的内容写入到临时文件中。然后再在写入成功后把临时文件重命名为path指定的文件。通过采用这种方法，就算有同名文件存在，写入发生错误时也不会损坏原来的文件。useAuxiliaryfile为NO的情况下，则直接输入字符串的内容到path指定的文件中。如果写入文件失败，则会在返回NO的同时把出错的原因写入到error中，并返回给函数的调用者。

(12)其他

```
- (id) init
```

对接收者进行初始化，并返回一个空字符串。这个方法通常被用于NSMutableString的初始化。

便利构造器:string

```
- (id) initWithString: (NSString *) aString
```

返回一个字符串对象，其内容是aString的副本。

aString也可以是一个NSMutableString的实例对象，用这个方法可以用一个可变的字符串对象生成一个不可变的字符串对象。

便利构造器:stringWithString:

```
- (NSString *) description
```

这个方法是在NSObject中定义的，会返回表示消息接收者内容的字符串。

NSString的description方法会直接返回self。

```
- (id) propertyList
```

返回消息接收者的属性列表(property list)，更多关于属性列表的内容请参考第13.3节。属性列表是一种格式，用来存储串行化后的对象，由NSString、NSData、NSArray和NSDictionary构成。属性列表文件的扩展名为.plist，因此通常称其为plist文件。Plist文件通常被用于储存用户设置，也可以将其用于存储捆绑的信息。

```
- (NSArray *) componentsSeparatedByCharactersInSet:
    (NSCharacterSet *) sep
```

用参数sep指定的字符集合中的字符作为分隔符，对消息的接收者字符串进行分割，并返回分割后生成的字符串数组。关于字符集合对象，可参考类NSCharacterSet和NSMutableCharacterSet的参考文档。例如，使用空格或制表符(Tab)作为分隔符时，可以用如下语句生成分隔符数组。

```
chrs = [NSCharacterSet whitespaceCharacterSet];
```

9.2.3 NSMutableString

本节将对可变字符串NSMutableString进行说明。NSMutableString是NSString的子类，所以

NSMutableString 可以使用 NSString 中定义的所有方法。和 NSString 一样，NSMutableString 的接口也定义在文件 Foundation/NSString.h 中。

下面对 NSString 中未定义的方法进行说明，更详细的信息请参考苹果公司的官方文档。

（1）实例对象的生成和初始化

```
- (id) initWithCapacity: (NSUInteger) capacity
```

初始化一个NSMutableString类型的对象，capacity指明了要被初始化的NSMutableString对象的大小。NSMutableString的对象会随着字符串的变化而自动扩展内存，所以capacity不需要非常精密。除了这个方法之外，还可以使用NSString的init方法或NSString的类方法string:来生成一个空的NSMutableString对象。

便利构造器：stringWithCapacity:

（2）追加字符串

```
- (void) appendString: (NSString *) aString
```

在消息接收者的末尾追加aString。

```
- (void) appendFormat: (NSString *) format, ...
```

在消息接收者的末尾追加format格式的格式化字符串。

（3）插入，删除，置换

```
- (void) insertString: (NSString *) aString
              atIndex: (NSUInteger) loc
```

在消息接收者的atIndex位置插入字符串aString。

```
- (void) deleteCharactersInRange: (NSRange) range
```

结构体NSRange表示一个范围，其中包含了开始位置和长度（详细内容请参考附录A）。

这个方法的作用是从接收者中删除aRange指定范围内的字符串。

```
- (void) setString: (NSString *) aString
```

复制aString指定的字符串，并将其设置为消息接收者的内容。

```
- (void) replaceCharactersInRange: (NSRange) aRange
                       withString: (NSString *) aString
```

把aRange指定范围内的字符内容替换为aString指定的字符串。

```
- (NSUInteger) replaceOccurrencesOfString: (NSString *) target
                               withString: (NSString *) replacement
                                  options: (NSStringCompareOptions) opts
                                    range: (NSRange) searchRange
```

searchRange指定范围内如果存在字符串target，就将其替换为replacement。这个函数的返回值就是替换的次数。使用选项opts可以设置忽略大小写，或者使用正则表达式进行替换等。

9.3 NSData

9.3.1 NSData

NSData 是 Cocoa 下对二进制数据的一个**封装**（wrapper），能够把二进制数据当作对象来处理。

同 C 语言的数组相比，NSDate 的优点是可以进行更抽象化的操作，使内存管理更容易，同时也是 CocoaAPI 中操作二进制数据的标准。

NSData 是不可变的，所以实例对象一旦创建之后就不可改变其内容。而如果想改变数据的内容的话，就需要使用将在后面介绍的类 NSMutableData。NSData 和 NSMutableData 的实例对象有时也称为**数据对象**。

NSData 的接口文件定义在 Foundation/NSData.h 中。NSData 是以类簇的方式实现的，所以不能用通常的方法来为 NSData 定义子类，详情请参考第 11 章中有关抽象类和类簇的内容。

下面介绍了类 NSData 的几个主要方法，如果想了解全部方法，请参考苹果公司的文档。

（1）数据对象的生成和初始化

```
- (id) initWithBytes: (const void *) bytes
              length: (NSUInteger) length
```

复制以 bytes 开头、长度为 length 的数据，进行初始化使其成为数据对象的内容。
便利构造器：dataWithBytes:length:

```
- (id) initWithBytesNoCopy: (void *) bytes
                    length: (unsigned) length
              freeWhenDone: (BOOL) flag
```

将以 bytes 开头、长度为 length 的数据初始化为数据对象的内容。生成的 NSData 中保存的是指向数据的指针，并没有对数据进行复制操作。flag 为 YES 的时候，生成的 NSDate 对象是 bytes 的所有者，当 NSDate 对象被释放的时候也会同时释放 bytes，所以 bytes 必须是通过 malloc 在堆上分配的内存。当 flag 为 NO 的时候，bytes 不会被自动释放，释放 bytes 时要注意时机，不要在 NSData 对象还被使用的时候释放 bytes。
便利构造器：dataWithBytesNoCopy:length:freeWhenDone:

```
- (id) initWithData: (NSData *) aData
```

用指定的 NSData 对象 aData 来创建一个新的 NSData 对象。参数可以是 NSMutableData 对象，所以用这个方法可以为一个可变的 NSMutableData 对象生成一个不可变的 NSData 对象。
便利构造器：dataWithData:

```
+ (id) data
```

返回一个长度为 0 的临时 NSDate 对象。这个方法多被用于 NSMutableData 中（创建一个长度为 0 的 NSData 意义不大）。对应的初始化方法为 init。

（2）访问 NSData 中的数据

下面的说明中用到的 NSRange 包括开始指针和数据的长度，更多详细内容请参考附录 A。

- (NSUInteger) **length**

返回NSData对象中数据的长度。

- (const void *) **bytes**

返回NSData对象中数据的首指针。

- (void) **getBytes:** (void *) buffer
  length: (NSUInteger) length

复制NSData对象的数据到buffer中，复制时从NSData对象中数据的开头开始，副本的长度为length。如果想获得指定范围内的数据的话，可以使用方法getBytes:range:。

- (NSData *) **subdataWithRange:** (NSRange) range

用range指定范围内的data来生成一个新的NSData对象并返回。

- (NSRange) **rangeOfData:** (NSData *) dataToFind
  options: (NSDataSearchOptions) mask
  range: (NSRange) searchRange

在接收者中searchRange指定的范围内，如果能找到和dataToFind一样的数据，则返回数据的位置和长度。mask是搜索时用到的选项，使用mask可以从后向前查找。dataToFind不可以为nil。

(3)比较

- (BOOL) **isEqualToData:** (id) anObject

两个NSData的数据长度和内容一致时返回YES。

(4)文件输入和输出

可以从文件读入数据来初始化NSData对象，或者把NSData对象中的内容输出到文件。除了用NSString指定文件路径的函数外，还有使用NSURL来指定文件路径的函数，详情请参考9.7节。

- (NSString *) **description**

返回一个ASCII编码格式的字符串，采用的格式是NSData属性列表的格式，数据段输出的时候采用<>括住的16进制。

- (id) **initWithContentsOfFile:** (NSString *) path
  options: (NSUInteger) mask
  error: (NSError **) errorPtr

从参数path指定的文件读入二进制数据，用该数据初始化NSData对象。如果读文件失败，则释放调用者并返回nil，同时把错误信息写入指针errorPtr中(更多关于错误处理的内容请参考18.5节)。mask是一个选项信息，用于指定是否使用虚拟内存等(详情请参考文档)。
便利构造器：dataWithContentsOfFile:options:error:

- (id) **initWithContentsOfFile:** (NSString *) path

相当于将上面方法中的第二个参数和第三个参数分别指定为了0和NULL。
便利构造器：dataWithContentsOfFile:

- (BOOL) **writeToFile:** (NSString *) path
  atomically: (BOOL) flag

将接收者的二进制数据写入path指定的文件中。如果写入成功，则返回YES。flag选项为YES时会进行

安全写操作(请参考上节中NSString的方法writeToFile:atomically:encoding:error)。另外，也可以参考方法writeToFile:options:error:。

9.3.2 NSMutableData

本节将对可变的数据类 NSMutableData 进行说明。

NSMutableData 的实例对象在初始化之后可被修改，比如增加或删除数据等。

NSMutableData 会随着数据的变更自动管理内存，使用者不必关心 NSMutableData 对象的内存管理。

下面将对 NSMutableData 的几个主要方法进行说明。其有一点要注意的是，NSMutableData 是 NSData 的子类，所以 NSMutableData 可以使用 NSData 的全部方法。

NSMutableData 的接口和 NSData 一样都定义在 Foundation/NSData.h 中。

(1)数据对象的生成和初始化

```
- (id) initWithCapacity: (NSUIteger) capacity
```

生成一个容量为capacity字节的NSData对象。对象的空间会随着数据的增加自动扩展。除了这个方法外，还可以使用NSData的便利构造器data来创建一个长度为空的NSData对象。

便利构造器：dataWithCapacity:

```
- (id) initWithLength: (NSUInteger) capacity
```

初始化一个容量为capacity字节的NSData对象，同时将对象的数据都设为0。

便利构造器：dataWithLength:

(2)访问数据

```
- (void *) mutableBytes
```

返回NSdata对象中数据缓冲区的头指针。和NSData的btyes方法不同，mutableBytes返回的指针是可写的。数据缓冲区的长度为0时，返回NULL。

(3)追加数据

```
- (void) appendData: (NSData *) otherData
```

复制otherData中的数据，并将其追加到接收者的数据缓冲区之后。

```
- (void) appendBytes: (const void *) bytes
              length: (NSUInteger) length
```

在对象的末尾追加长度为length的bytes数据。

(4)更新数据

```
- (void) replaceBytesInRange: (NSRange )range
                   withBytes: (const void *) replacementBytes
                      length: (NSUInteger) replacementLength
```

把range指定范围内的数据替换为长度为replacementLength的数据replacementBytes。

range的长度length和要替换的数据的长度replacementLength不等的话，range后面的数据会相应地被前后移动，长度也会发生变化。

length为0的话，就相当于向原对象的头部插入replacementBytes。

- (void) **replaceBytesInRange:** (NSRange) range
  withBytes: (const void *) bytes

把接收者中range指定范围内的数据替换为bytes。相当于上面的方法replaceBytesInRange:withBytes:length:中，指定范围的长度和要替换的数据的长度一样时的情况。

- (void) **setData:** (NSData *) aData

设置NSMutableData类中的数据为aData所指向的内容。

- (void) **resetBytesInRange:** (NSRange) range

将range范围内的数据设为0。

(5)增长数据缓冲区的长度

- (void) **increaseLengthBy:** (NSUInteger) extraLength

为数据缓冲区增加长度extraLength，新增的区域都会被初始化为0。

- (void) **setLength:** (NSUInteger) length

重设数据缓冲区的长度为length，根据length的不同，数据缓冲区可增长或缩短。数据缓冲区增长的情况下，用0来填充新增的区域。

9.4 数组类

9.4.1 NSArray

本节将对 Cocoa 环境下的数组类 NSArray 进行说明。

数组是多个对象的有序集，可以通过对象在数组中的位置（即索引）来访问对象。和 C 语言一样，数组中元素的索引是从 0 开始的。数组中既可以存放不同类的对象也可以存放同一类的对象，但不能存放 nil，nil 被用于标志数组的结束。

NSArray 是不可变数组，一旦创建之后，就不能再添加、删除或修改其中的元素。如果想更改数组中的元素的话，就需要使用可变数组 NSMutableArray。NSArray 和 NSMutableArray 的实例称为**数组对象**，或简称为数组。

基于引用计数的内存管理模式下，会对每个加入到数组中的对象发送 retain 消息并进行保持。另外，当数组被释放的时候，就会对其中保存的全部对象发送 release 消息。本节后半部分会详述所有权的相关问题。

NSArray 的接口定义在 Foundation/NSArray.h 中，NSArray 是以类簇的方式实现的，所以不能用通常的方法来为 NSArray 定义子类，更多内容请参考第 11 章。

下面介绍一下 NSArray 的一些主要方法，全部方法的完整介绍请参考 Objective-C 的官方文档。

（1）数组对象的生成和初始化

+ (id) **array**

返回一个不包含任何元素的空的数组对象。可变数组NSMutableArray中经常会用到这个方法。这个方法对应的初始化方法为init。

+ (id) **arrayWithObject:** (id) anObject

生成并返回只包含anObject这一个元素的数组。

- (id) **initWithObjects:** (id) firstObj, ...

生成并返回数组，数组中的元素由参数指定。参数之间用逗号分割，并以nil结尾。
便利构造器：arrayWithObjects:

- (id) **initWithObjects:** (const id *) objects
count: (NSUInteger) count

参数objects是一个C语言风格的对象数组。这个方法会返回一个包含objects中前count个对象的数组。
便利构造器：arrayWithObjects:count:

- (id) **initWithArray:** (NSArray *) anArray

用已有数组anArray中的对象来初始化并返回一个新的数组。anArray也可以是可变数组NSMutableArray的对象，所以这个方法也可以被用来从一个可变数组生成一个不可变数组。
便利构造器：arrayWithArray:

- (id) **initWithArray:** (NSArray *) array
copyItems: (BOOL) flag

和上面的方法的功能一样，唯一的区别是flag为真的情况下，会用array中的每个元素的副本生成新的数组。关于如何生成一个对象的副本，请参考第13章的内容。

（2）访问数组中的元素

下面说明访问数组中的元素和查询数组中是否包含某个对象的方法。

在比较两个对象时，只要无特殊说明，两个对象相等指的就是调用方法isEqual:后返回YES。另外，结构体NSRange表明一个范围，包含缓冲区的开始位置和长度这两个参数。详情请参考附录A。

- (NSUInteger) **count**

返回数组中元素的个数。

- (NSUInteger) **indexOfObject:** (id) anObject

在数组中查询看是否有和anObject相等的元素，如果有就返回这个元素的索引，否者就返回NSNotFound（NSNotFound是一个宏，表示没有找到某个内容，详情请参考附录A）。
如果只想查询数组中是否包含某个元素的话，可以使用方法containsObject:。

- (id) **objectAtIndex:** (NSUInteger) index

返回index索引位置的元素。如果index超过数组的最大长度的话，就会触发异常NSRangeException的发生。

- (id) **lastObject**

返回数组中的最后一个元素。如果接收者为空的话，则返回nil。

- (void) **getObjects:** (id __unsafe_unretained []) aBuffer
  range: (NSRange) aRange

  将aRange指定范围内的对象复制到aBuffer指定的C语言缓冲区中。只复制指针，引用计数不发生变化，也就是说不会发生任何所有权的改变。使用ARC的时候，为了保持数组中的对象，会将其赋值给另一个强引用的变量。aBuffer需要足够大，以能够放下所有数据。

- (NSArray *) **subarrayWithRange:** (NSRange) range

  抽取原数组中的一部分(用range指定范围)来生成一个新的数组。

(3)比较

- (BOOL) **isEqualToArray:** (id) anObject

  比较两个数组是否一致。如果消息的接收者和anObject包含的元素个数相同，而且相同索引位置处的元素相等的话，则返回YES。

- (id) **firstObjectCommonWithArray:** (NSArray *) otherArray

  返回消息的接收者和otherArray这两个数组中第一个相同的元素。如果两个数组没有相同的元素，则返回nil。

(4)为数组增加新的对象

NSArray是不可变类型的对象，不可以直接为其增加对象。如果想为数组对象增加新的对象的话，可以返回一个新的数组，新数组由原来的数组和新增对象共同构成。

- (NSArray *) **arrayByAddingObject:** (id) anObject

  新生成并返回一个数组对象，新数组中的元素由消息接收者的元素和anObject共同构成，anObject加在新数组的末尾。

- (NSArray *) **arrayByAddingObjectsFromArray:** (NSArray *) anArray

  新生成并返回一个数组对象，新数组中的元素由消息接收者的元素和anArray中的元素共同构成，anArray中的元素加在原数组的末尾。

(5)排序

NSArray的排序方法会返回一个新的数组，新数组是对旧数组中的元素经过选择器排序后的数组。

- (NSArray *) **sortedArrayUsingSelector:** (SEL) comparator

  对数组中的对象逐个进行比较，并根据比较的结果生成一个新的数组。
  数组对象之间比较的时候使用选择器comparator指定的方法。选择器要求有一个输入参数，返回值是NSComparisonResult类型(参见附录A)。
  例如，如果要对一个NSString的NSArray进行排序的话，选择器可以使用NSString的方法compare:。

```
newArray = [anArray sortedArrayUsingSelector:@selector(compare:)];
```

- (NSArray *) **sortedArrayUsingFunction:**
  (NSInteger(*)(id, id, void *))comparator
  context: (void *)context

  这个方法和上面的方法一样，也是对NSArray中的元素进行排序，然后返回一个排好序的新的

NSArray。

区别在于对数组中的对象进行比较的时候使用的是函数comparator。函数comparator有3个参数，前两个分别是数组中的元素，第三个是比较时自定义的一些选项，例如可以设置为忽略大小写等。函数的返回值是NSComparisonResult类型。comparator是一个函数指针，关于函数指针的详细内容可参考第8章关于函数指针的Column。

一个排序用的函数的定义如下所示。

```
NSInteger myCmp(id arg1, id arg2, void *context);
```

(6)给数组中的元素发送消息

可以给数组中的每个对象发送消息，消息通过参数的消息选择器来指定。

发送消息的时候会从第一个元素开始直到最后一个元素结束。

元素在响应消息的同时有可能会发生波及作用，例如数组自身发生变化等，这种情况下就无法保证最后一个元素也能够成功响应消息。

- (void) **makeObjectsPerformSelector:** (SEL) aSelector

给数组中的每个对象发送消息，通过aSelector来指定消息。

- (void) **makeObjectsPerformSelector:** (SEL) aSelector
withObject: (id) anObj

给数组中的每个对象发送消息，通过aSelector来指定消息，消息可带有一个参数，通过anObj来指定。

(7)文件输入与输出

文件的输入与输出以属性列表的形式进行(见13.3节)。另外还存在通过NSURL指定文件的方法(见9.7节)。

- (NSString *) **description**

将数组对象中的内容以ASCII编码的属性列表格式的字符串返回。返回由给数组中每个元素发送description消息而得到的字符串，字符串之间用“,”分割，并用()括起来。

- (id) **initWithContentsOfFile:** (NSString *) aPath

从属性列表文件读取内容来初始化一个NSArray。如果读入失败，则释放消息接收者并返回nil。

便利构造器: arrayWithContentsOfFile

- (BOOL) **writeToFile:** (NSString *) path
atomically: (BOOL) flag

将数组中的内容以属性列表的格式写入aPath指定的文件中。正常写入时返回YES。flag为YES的情况下，执行安全写入。可以参考NSString的方法writeToFile:atomically:encoding:error。

- (NSString *) **componentsJoinedByString:** (NSString *) separator

返回一个临时字符串，字符串的内容为用“,”连接的数组中每个元素执行description的结果。

- (NSArray *) **pathsMatchingExtensions:** (NSArray *) filterTypes

这个方法被用于筛选带有特定扩展名的字符串，返回的是一个临时数组，数组中的每个元素的扩展名都属于filterTypes。逐个检查消息接收者数组中的每个元素，如果其扩展名在数组filterTypes中，就将其放入临时数组中。

9.4.2 NSMutableArray

本节将介绍可变数组类 NSMutableArray 的概要。NSMutableArray 是 NSArray 的子类，所以可以使用 NSArray 中定义的全部方法。NSMutableArray 的接口也定义在文件 Foundation/NSArray.h 中。

NSMutableDictionary 是可变的，所以可随意添加或删除其中的元素。随着添加或删除操作，数组中元素的位置会前后移动，如图 9-1 所示。和 C 语言的数组不同，NSMutableDictionary 中不存在空的位置。

图 9-1 为可变数组追加或删除元素

下面对 NSMutableArray 的主要方法进行说明，详细内容请参考苹果公司的官方文档。

（1）可变数组的初始化

```
- (id) initWithCapacity: (NSUInteger) numItems
```

创建并初始化一个长度为numItems的可变数组。虽然NSMutableArray会随着其中元素的增减自动管理内存，但也可以在初始化的时候指定数组的大小。也可以使用NSArray的array方法创建一个长度为0的可变数组。

便利构造器：`arrayWithCapacity:`

（2）向数组中追加和替换元素

基于引用计数的内存管理模式下，会给加入数组的对象发送`retain`消息，给从数组删除的对象发送`release`消息。

```
- (void) addObject: (id) anObject
```

添加元素anObject到数组的末尾，anObject不可以为nil。

```
- (void) addObjectsFromArray: (NSArray *) otherArray
```

将otherArray数组中的元素添加到数组的末尾。

```
- (void) insertObject: (id) anObject
                atIndex: (NSUInteger) index
```

添加anObject到index指定的位置，后面的元素顺次后移。index必须在0和数组范围之间。

```
- (void) replaceObjectAtIndex: (NSUInteger) index
                   withObject: (id) anObject
```

用anObject指定的对象代替index位置上的元素。若index超过数组范围，则返回NSRangeException。anObject不可为nil。

```
- (void) replaceObjectsInRange: (NSRange) aRange
          withObjectsFromArray: (NSArray *) otherArray
```

使用otherArray 数组中的元素替换当前数组中aRange范围内的元素。当前数组中aRange范围内的元

素会被删除掉。

- (void) **setArray:** (NSArray *) otherArray

用otherArray数组中的元素替换当前数组中的所有元素。当前数组中的所有元素都会被删除掉。

- (void) **exchangeObjectAtIndex:** (NSUInteger) idx1
  withObjectAtIndex: (NSUInteger) idx2

交换idx1和idx2位置上的元素。

(3)删除数组中的元素

下面介绍一下删除数组中的元素的方法，所有被删除的元素都会被发送release消息。

- (void) **removeAllObjects**

删除数组中的所有元素，将其置空。

- (void) **removeLastObject**

删除数组中的最后一个元素。

- (void) **removeObjectAtIndex:** (NSUInteger) index

删除数组中指定位置处的元素。

- (void) **removeObjectsInRange:** (NSRange) aRange

删除数组中指定范围内的元素。

- (void) **removeObject:** (id) anObject

删除数组中所有和anObject相等的元素。可以使用方法isEqual:来判断数组中的元素是否和anObject相等，返回为YES的时候代表两个元素相等。

- (void) **removeObjectsInArray:** (NSArray *) otherArray

从当前数组中删除otherArray数组中包含的所有元素。

(4)排序

- (void) **sortUsingSelector:** (SEL) comparator

对当前数组中的元素进行升序排序。
排序时使用指定的selector:comparator来进行元素之间的比较。更多详细内容请参考NSArray的sortedArrayUsingSelector:方法。

- (void) **sortUsingFunction:** (NSInteger (*) (id, id, void *))
  compare
  context: (void *) context

对当前数组中的元素进行升序排序。
排序时使用传入的comparator来进行元素之间的比较。更多详细内容请参考NSArray的sortedArrayUsingFunction:context:方法。

9.4.3 数组对象的所有权

基于引用计数的内存管理模式下，需要注意数组中的对象的所有权。

数组会给其中的所有对象发送retain消息。当数组被释放的时候，它会给数组中的所有对象发送release消息。当对象被从数组中删除时，它也会收到一条release消息。

了解了数组中对象的所有权的管理原则之后，让我们看一下下面这行错误代码，看看错在了哪里。

```
NSArray *arr = [[NSArray alloc] initWithObjects:
                    [[Card alloc] init], [[Player alloc] init], nil];
```

假设执行这段代码的对象为 obj，那么 obj 就是数组中的两个元素的所有者。同时数组 arr 也是这两个元素的所有者。当 arr 被释放的时候，虽然 arr 会给这两个元素发送 release 消息并释放所有权。但 obj 对这两个元素的所有权并没有被释放，因此就会发生内存泄漏。

在 ARC 的情况下，这行代码则没有问题。ARC 会在使用之前发送retain消息，使用之后发送release消息。但因为这两个对象是通过 init 初始化的，所以 ARC 不会发送retain消息，能够正确地释放掉这两个对象。手动管理内存的情况下，这段代码的正确书写方法如下。

```
NSArray *arr = [[NSArray alloc] initWithObjects:
                    [[[Card alloc] init] autorelease],
                    [[[Player alloc] init] autorelease], nil];
```

接下来让我们看一下想继续使用从数组中删除的元素时该如何进行引用计数的管理。只有在手动内存管理的时候才需要额外操作，ARC 的情况下不需要这些操作。

```
NSMutableArray *marr;
id obj;
    ...
/*ARC 的情况下 */
obj = [marr objectAtIndex: index]; // 因为是强引用，所以会被保持
[marr removeObjectAtIndex: index];

/* 手动内存管理的情况下，需要先 retain，再设为自动释放 */
obj = [[[marr objectAtIndex: index] retain] autorelease];
[marr removeObjectAtIndex: index];
```

数组、集合和词典对象这些可以包含多个对象的容器被总称为**集合**（collection）。

一个对象被放入集合时会收到retain消息，被从集合中删除的时候会收到release消息。

9.4.4 快速枚举

Objective-C 2.0 新提供了一个用于遍历容器类（数组、集合和词典等）的语法，叫作**快速枚举**（fast enumeration）。

下面就是利用快速枚举来遍历容器 group 的一个例子，遍历过程中 group 自身不会发生任何变化。

```
id obj;
for (obj in group) {
    printf("%s\n", [[obj description] UTF8String]);
}
```

本书中把这种语法称为 for...in **语法**，它的结构如下所示。其中“变量”必须是可以放入到容器中的类型。“集合”是一个容器类的对象。在第一次执行循环之前会判断“集合”是否合法和其中是否有元素。

语法	for...in 语法

```
for ( 变量 in 集合) {
      /*相应的处理*/
}
```

集合即可以是可变集合也可以是不可变集合。可变集合在循环的过程中也不允许被改变。如果集合发生了变化，就会抛出异常。

遍历容器中元素的顺序和容器的类型相关。如果是数组类型，就会从头开始遍历，而如果是集合或词典类型，遍历的顺序则和容器的内部实现相关。更多内容请参考 9.5 节。

这种语法也支持多重循环，下面就是利用二重循环来创建集合中两个不同元素的所有组合的一个例子。

```
id x, y;
for (x in group) {
    for (y in group)
        if (x != y) {
            if ([self checkCombination:x with:y])
                break;
        }
}
```

另外，变量 x 还可以在 for 的条件中定义，这种情况下 x 只在 for...in 语法块内有效。

```
for (id x in group) { ... }
```

9.4.5 枚举器 NSEnumerator

枚举器（enumerator）是一个用来遍历集合类（如 NSArray、NSSet、NSDictionary 等）中的元素对象的抽象类。Objective-C 2.0 中新增了更方便地遍历集合类的 for...in 语法。但除了遍历元素之外，枚举器还有其他用途，下面就让我们详细地看一下。

枚举器没有用来创建实例的公有接口，不能给枚举器类发送`alloc`消息，需要和集合类配合使用，返回一个用于遍历的实例对象。NSEnumerator 的接口定义在 Foundation/NSEnumerator.h 中。

NSEnumerator 中有两个抽象方法。

- (id) **nextObject**

  nextObject方法可以依次遍历每个集合元素，结束时返回nil。通过与while结合使用可遍历集合中所有的项。

- (NSArray *) **allObjects**

  allObjects方法可以返回集合中未被遍历的所有元素的数组。

不同的集合类返回枚举器的方法各不相同。例如，数组NSArray类返回枚举器的两个方法如下所示。

- (NSEnumerator *) **objectEnumerator**

  返回一个按照顺序进行遍历的枚举器。

- (NSEnumerator *) **reverseObjectEnumerator**

  返回一个按照逆序进行遍历的枚举器。

下面是一个使用枚举器来按照顺序遍历数组的例子。

```
NSArray *myarray;
id obj;
NSEnumerator *enumerator;
    ...
enumerator = [myarray objectEnumerator];
while ((obj = [enumerator nextObject]) != nil) {
    /* 处理 */
}
```

在使用枚举器遍历一个集合对象的同时，如果向该集合对象增加或删除对象，就可能会导致不可预期的结果，是很危险的。

基于引用计数的内存管理模式下，在通过枚举器遍历集合对象的时候，枚举器对象拥有这个集合对象的所有权。取出最后一个元素之后，它就会自动放弃对这个集合对象的所有权。同时，因为枚举器对象是一个临时对象，所以当其所在的自动释放池被释放的时候，该枚举器对象也会被释放掉。

9.4.6 快速枚举和枚举器

以上我们介绍了快速枚举和枚举器两种遍历集合对象的方法。快速枚举内部用C语言实现，以速度快而得名。

既然快速枚举有这么多优点，那是不是就不需要再使用枚举器了呢？答案是否定的。例如，在需要根据条件跳过某个元素等时，使用枚举器写代码就更方便。

另外，for...in语法还可以像下面这么使用，把上面介绍的“集合”换为“枚举器”。

语法	使用枚举器的 for...in 语法
`for ( 变量 in 枚举器 ) { /*处理*/ }`	

这个for...in语法通过利用从集合获得的枚举器来遍历这个集合的所有元素。枚举器可以有很多

种，例如数组可以有正序和逆序两种枚举器。下面就是一个利用逆序枚举器来遍历数组的例子。

```
enumerator = [myarray reverseObjectEnumerator];  // 获取一个逆序枚举器
for (obj in enumerator) {
    /* 处理 */
}
```

除了快速枚举和枚举器外，还可以通过下标来循环遍历集合中的所有元素。特别是如果想在遍历的过程中更改集合对象的话，就可以使用下标，因为快速枚举和枚举器都不允许更改操作。

例如，假设要遍历集合中的所有元素，并把满足某个条件的元素从集合中删除。这种情况下使用下标操作会更安全。但需要注意的是，遍历的时候要由后向前遍历。

```
NSMutableArray *myarray;
id obj;
NSInteger len, idx;
    ...
len = [myarray count];
for (idx = len - 1; idx >= 0; idx--) {
    obj = [myarray objectAtIndex: idx];
    if ([obj isGeek])
        [myarray removeObjectAtIndex: idx];
}
```

所有支持高速枚举的类都实现了 NSFastEnumeration 协议。枚举器类 NSEnumerator 也实现了这个协议，所以也支持 for...in 语法。更多关于协议的内容请参考第 12 章。高速列举协议和枚举器的接口都定义在 Foundation/NSEnumerator.h 中。

自定义的类也可以支持高速枚举，但实现起来比较麻烦。考虑到实际编程的时候数组、词典等集合类已经足够使用，本书中没有讲解高速枚举的实现，更多详细内容请参考苹果公司的官方文档。

9.4.7 集合类

Foundation 框架中提供了 **NSSet** 类，它是一组单值对象的集合。同 NSArray 不同，NSSet 是无序的，同一个对象只能保存一个。集合类也有两个，即不可变的 NSSet 类和可变的 NSMutableSet 类。**NSMutableSet** 是 NSSet 的子类，以类簇的形式存在。除此之外，类 NSMutableSet 还有一个子类 NSCountedSet。NSCountedSet 是一个可变的集合类，能够统计集合中对象的个数，这个类中可以存放多个相同值的对象。更多关于 NSCountedSet 的内容请参考苹果公司的官方文档。

集合类的接口定义在 Foundation/NSSet.h 中。

在基于引用计数的内存管理模式下，对元素对象所有权的处理和 NSArray 一样，即把对象放入集合的时候发送`retain`消息，把对象从集合中删除的时候发送`release`消息。

下面我们介绍一下 NSSet 的主要方法，更多详细内容请参考苹果公司的官方文档。

+ (id) **set**

返回一个临时的空的集合对象，对应的初始化方法是`init`。

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

  使用参数array中的元素来初始化生成一个集合。array中存在重复元素时，集合中只保存一个。除了这个函数之外，集合类还包含由nil结尾的对象列表创建的集合的初始化构造函数、参数是C语言风格的数组的初始化构造函数，以及以上各个函数相应的便利构造器。

- (NSUInteger) **count**

  返回集合对象中包含的元素个数。

- (NSArray *) **allObjects**

  将集合对象中所有的元素以数组的形式返回。

- (BOOL) **containsObject:** (id) anObject

  判断指定的anObject元素是否位于集合中，如果是则返回YES。

- (BOOL) **isEqualToSet:** (NSSet *) otherSet

  判断两个集合是否相等，相等时返回YES。

- (BOOL) **isSubsetOfSet:** (NSSet *) otherSet

  判断当前集合的对象是否全部位于集合otherSet中，如果是则返回YES。

- (BOOL) **intersectsSet:** (NSSet *) otherSet

  判断两个集合是否有共通的元素，如果有则返回YES。

下面说明 NSMutableSet 中的方法。

- (id) **initWithCapacity:** (NSUInteger) numItems

  初始化一个大小为numItems的集合。
  便利构造器：setWithCapacity:

- (void) **addObject:** (id) anObject

  向集合中追加元素anObject。如果结合中已经有这个元素，就什么也不会发生。
  同样，方法addObjectsFromArray:的意思是把anObject数组中的所有元素追加到集合中。

- (void) **removeObject:** (id) anObject

  从集合中删除元素anObject。

- (void) **unionSet:** (NSSet *) otherSet

  将集合otherSet中的元素加入到当前集合中，生成两个集合的并集。
  同样，方法minusSet:是从当前集合中删除同输入集合共通的元素。
  方法intersectSet:是生成两个集合的交集。

9.5 词典类

Cocoa Foundation 框架中提供了一种和 Java/C++ 中的 map 相类似的数据结构，叫作**词典**。词典也分为不可变词典 NSDictionary 和可变词典 NSMutableDictionary。

词典中的数据以键值对的形式保存，一个键值对称为 entry。键和值可以是任何对象，一般使用

字符串作为键。

传统的基于过程的编程语言中通常没有词典这样的数据结构。使用词典编程，可以大大提高效率。

让我们来看一个用词典表示乐曲信息的例子。表示乐曲信息的结构基本都是一样的，但也有个别乐曲有所不同，例如有的乐曲的作者是多个人，有的乐曲有参考信息等。

图 9-2（a）中，每个框包围的部分就是一个词典对象，其中每行都是一个 entry。一个乐曲的完整信息由多个词典对象构成，例如，可以通过 name 得到曲名，通过 lyrics 得到作词者，通过 music 得到作曲者，通过 note 得到备注。当然也有可能其中某一项是空的。

在这个例子中，key 和 value 都是字符串类型的对象，value 由多个字符串构成的时候可以用 () 把它们括起来，这种情况下词典对象的值就相当于一个数组对象。另外，除了字符串之外，词典对象的值还支持任意对象。例如，也可以把歌曲的声音或视频定义为一个对象放入词典对象中。

让我们再来看一个用词典对象表示大学教室的例子，如图 9-2（b）所示。和乐曲的例子一样，每个框包围的部分就是一个词典对象。关键字 capacity 用来表明教室可以装多少个人，mic 和 screen 用来表明教室是否配备了麦克和投影屏幕。数值类型的 value 既可以使用字符串也可以使用 NSNumber 来描述（更多关于 NSNumber 的内容请参考 9.6 节）。

▶ 图 9-2 词典对象的例子

关键字	值
name	"小岛的歌"
lyrics	"与田准一"
music	"芥川也寸志"
name	"故乡"
lyrics	"高野辰之"
music	"岗野贞一"
name	"月"
note	"文部省唱歌"
name	"东风"
music	"坂本龙一"
name	"手掛かり"
lyrics	("细野晴臣", "Peter Barakan")
music	("细野晴臣", "高橋幸宏")

（a）歌曲信息

关键字	值
room	"LR501"
capacity	180
mic	YES
projector	("PC", "DVD")
screen	YES
room	"LR401"
capacity	150
mic	YES
projector	"PC"
screen	YES
room	"LR402"
capacity	45
mic	NO
screen	NO
note	"带桌子的椅子"

（b）教室的信息

在面向过程的语言中，一般使用下标或成员名来获取数组或结构体的值。词典对象的 key 和 value 都可以是对象类型，所以可扩展性非常高，可以用在各种环境下。

词典的键必须是唯一的，也就是说，使用方法`isEqual:`来比较各个键时，必须各不相等。另外，nil 不能作为词典的键。

词典对象的值可以是除了 nil 外的任意对象，也可以是数组对象或词典对象。如果想保存数值或坐标对象的话，可以使用后面将介绍到的 NSNumber 和 NSValue 对象。另外，也可以使用 NSNull 来表明一个词典对象为空。

一个对象作为词典的 key 或者 value 时，词典中存放的是这个对象的一份副本。

基于引用计数的内存管理模式下，会给加入词典的 key 和 value 都发送一次`retain`，使它们的

引用计数器加 1。在词典对象被释放的时候，会给词典的所有 key 和 value 对象发送一次release消息，使计数器减 1。

更多关于集合对象所有权的内容，请参考 9.4 节。

9.5.1 NSDictionary

类 NSDictionary 是不可变的词典类，一旦创建之后就只能查询，不可再增加、删除或修改其中的内容。如果创建之后还想继续修改的话，请使用后面将要介绍到的类 NSMutableDictionary。

NSDictionary 的接口定义在 Foundation/NSDictionary.h 中。NSDictionary 是以类簇的方式实现的，所以无法用通常的方法为 NSDictionary 创建子类，更多详细内容请参考第 11 章。

词典有很多方法，下面对常用的方法进行说明，更多方法请参考苹果公司的官方文档。

(1)词典对象的生成和初始化

```
+ (id) dictionary
```

生成并返回一个空的词典对象。可变词典NSMutableDictionary经常使用这个方法返回一个空的词典，然后再通过init来初始化。

```
+ (id) dictionaryWithObject: (id) anObject
                     forKey: (id) aKey
```

返回一个词典对象，其中只包含一个关键字为aKey，值为anObject的键值对。

```
- (id) initWithObjects: (NSArray *) objects
               forKeys: (NSArray *) keys
```

从数组objects和keys中各取出一个元素作为value和key的键值对，返回包含该键值对的字典对象。数组objects和keys中必须包含相同数量的对象。

便利构造器：dictionaryWithObjects:forKeys:

```
- (id) initWithObjects: (const id []) objects
               forKeys: (const id []) keys
                 count: (NSUInteger) count
```

返回一个词典对象，由objects和keys数组中的元素作为键值对来初始化。词典对象中包含的元素个数由count来指定。

便利构造器：dictionaryWithObjects:forKeys:count:

```
- (id) initWithObjectsAndKeys: (id) object, (id) key, ...
```

使用指定的关键字和值来初始化词典对象，键值对以nil结束。

便利构造器：dictionaryWithObjectsAndKeys:

```
- (id) initWithDictionary: (NSDictionary *) otherDictionary
```

用一个已存在的词典对象来初始化另一个。

这个方法的参数也可以是一个可变词典对象NSMutableDictionary，这时能够为可变词典创建一个同样内容的不可变词典对象。

便利构造器为：dictionaryWithDictionary:

(2)访问词典对象

- (NSUInteger) **count**

返回词典对象中键值对的数量。

- (id) **objectForKey:** (id) aKey

返回指定aKey对应的值，如果aKey不存在就返回nil。

- (NSArray *) **allKeys**

返回一个数组，其中包含词典对象所有的关键字。如果词典对象为空，则返回一个空数组。方法allValues与此类似，返回一个包含词典对象所有值的数组。

- (NSEnumerator *) **keyEnumerator**

返回一个可访问词典中所有关键字的快速枚举器。与此类似，方法objectEnumerator返回一个可访问词典中所有值对象的快速枚举器。

- (NSArray *) **allKeysForObject:** (id) anObject

返回一个数组，数组中的元素为词典中值为anObject的所有关键字。如果输入的值在数组中不存在，则返回一个nil。使用方法isEqual:来进行比较。

(3)比较

- (BOOL) **isEqualToDictionary:** (id) anObject

比较两个字典，如果两个字典的键值对数和两个字典的每个关键字及其对应的值都相等，则返回YES。

(4)文件输入输出

可以通过文件来初始化词典对象或把词典对象中的内容输出到一个文件。文件的输入和输出都是通过属性列表完成的，更多内容请参考13.3节。输入输出用的文件名既可以使用字符串类型也可以使用NSURL类型(关于NSURL的更多内容请参考9.7节)。

- (NSString *) **description**

将词典对象的内容以ASCII编码的属性列表格式输出。
如果词典的关键字是字符串类型，就按照升序输出。如果不是字符串类型，则随机输出。

- (id) **initWithContentsOfFile:** (NSString *) path

从属性列表格式保存的文件来初始化词典对象。如果读取文件失败，则释放词典对象，返回nil。
便利构造器:dictionaryWithContentsOfFile:

- (BOOL) **writeToFile:** (NSString *) path
atomically: (BOOL) flag

把代表这个词典内容的属性列表输出到指定的文件。写入成功时返回YES。writeToFile:方法中有一个BOOL类型的参数flag，如果flag为YES，则代表安全写入。更多内容请参考NSString的writeToFile:atomically:encoding:error:方法。

9.5.2 NSMutableDicitionary

下面介绍可变词典类 NSMutableDictionary。

NSMutableDictionary 允许随意添加、删除或修改键值对。随着词典中元素的变更，NSMutableDictionary 会自动管理内存。

下面对 NSMutableDictionary 的常用方法进行说明，NSMutableDictionary 是 NSDictionary 的子类，所以 NSMutableDictionary 可以使用 NSDictionary 的全部方法。NSMutableDiction 的接口和 NSDictionary 一样都定义在 Foundation/NSDictionary.h 中。

（1）词典对象的生成和初始化

- (id) **initWithCapacity:** (NSUInteger) capacity

创建并初始化一个长度为capacity的可变词典。虽然NSMutableDiction会随着其中元素的增减自动管理内存，但也可以在初始化的时候指定词典对象的大小。

便利构造器：`dictionaryWithCapacity:`

（2）增加和删除键值对

向词典中追加元素的时候，词典中会保存一份键值的副本。

基于引用计数的内存管理模式下，会给追加的对象发送`retain`消息。当从词典中删除值对象的时候，则会给删除的键和值对象发送`release`消息。

向词典中追加键值对时，如果关键字已存在，则会用新值替换旧值。具体来说，就是给原有值发送`release`消息，同时给新值发送`retain`消息。

- (void) **setObject:** (id) anObject
  forKey: (id) aKey

向可变词典中添加元素，要追加的关键字和值都不能为nil。

- (void) **addEntriesFromDictionary:** (NSDictionary *) otherDic

将otherDic中的数据追加到当前词典中，如果两个词典的关键字相同，则以otherDic的值作为最终值。

- (void) **setDictionary:** (NSDictionary *) otherDic

用新的词典otherDic覆盖当前词典，当前词典中的所有数据都被删除。

- (void) **removeObjectForKey:** (id) aKey

删除关键字为aKey的键值对。

- (void) **removeObjectsForKeys:** (NSArray *) keyArray

删除键值为数组keyArray中元素的所有键值对。

- (void) **removeAllObjects**

删除词典中的所有键值对。

9.6 包裹类

Cocoa Foundation 框架的集合类（NSArray、NSDictionary 和 NSSet）中只可以放入对象，不能存储基本类型的数据。所以 Cocoa 提供了 **NSNumber** 类来包装 char、int、long 等基本类型的数据，使其能够被放入类似于 NSArray 或 NSDictionary 的集合中。像结构体、指针这些复杂的数据类型，NSNumber 没有办法把它们存储为对象，这时就可以使用 **NSValue**。NSValue 是 NSNumber 的父类，NSValue 可以把任意类型包装成对象。

除了 NSValue 和 NSNumber 之外，本节中我们还会介绍到 **NSNull**。NSNull 是为了把 nil 放入集合类中而定义的包裹类。

NSValue 和 NSNumber 是以类簇的形式实现的。例如，创建 NSNumber 对象时，实际上获得的可能是 NSInteger 类型的对象，所以无法用通常的办法为这两个类定义子类（请参考第 11 章中有关抽象类和类簇的内容）。

下面我们说明一下包裹类中最常用的一些方法，更多内容请参考苹果公司的官方文档。

9.6.1 NSNumber

NSNumber 的接口文件是 Foundation/NSValue.h。

（1）生成和初始化

下面的方法生成并初始化一个整数对象。注意 NSInteger 不是一个对象，而是基本数据类型的 typedef。它被 typedef 成 64 位的 long 或者 32 位的 int。

```
- (id) initWithInteger: (NSInteger) value
```

对应的类方法如下所示。

```
+ (NSNumber *) numberWithInteger: (NSInteger) value
```

表 9-3 中列出了为各种数据类型生成 NSNumber 对象的初始化实例方法和便利构造器。

表 9-3 NSNumber 的初始化和生成方法

数据类型	初始化方法	便利构造器
BOOL	-initWithBool:	+ numberWithBool:
char	-initWithChar:	+ numberWithChar:
double	-initWithDouble:	+ numberWithDouble:
float	-initWithFloat:	+ numberWithFloat:
int	-initWithInt:	+ numberWithInt:
NSInteger	-initWithInteger:	+ numberWithInteger:
long	-initWithLong:	+ numberWithLong:

（续）

数据类型	初始化方法	便利构造器
long long	-initWithLongLong:	+ numberWithLongLong:
short	-initWithShort:	+ numberWithShort:
unsigned char	-initWithUnsignedChar:	+ numberWithUnsignedChar:
unsigned int	-initWithUnsignedInt:	+ numberWithUnsignedInt:
NSUInteger	-initWithUnsignedInteger:	+ numberWithUnsignedInteger:
unsigned long	-initWithUnsignedLong:	+ numberWithUnsignedLong:
unsigned long long	-initWithUnsignedLongLong:	+ numberWithUnsignedLongLong:
unsigned short	-initWithUnsignedShort:	+ numberWithUnsignedShort:

（2）提取包裹类中的值

使用下面的方法可以提取出NSNumber实例中的NSInteger值。如果消息的接收者不是NSInteger的包裹类，会自动进行类型转换。

```
- (NSInteger) integerValue
```

表9-4列出了从包裹类中提取各种数据类型的方法。

表9-4 获取NSNumber中的值的方法

数据类型	获取值的方法
BOOL	-boolValue
char	-charValue
double	-doubleValue
float	-floatValue
int	-intValue
NSInteger	-integerValue
long	-longValue
long long	-longlongValue
short	-shortValue
unsigned char	-unsignedCharValue
unsigned int	-unsignedIntValue
NSUInteger	-unsignedIntegerValue
unsigned long	-unsignedLongValue
unsigned long long	-unsignedLongLongValue
unsigned short	-unsignedShortValue

（3）其他

```
- (BOOL) isEqualToNumber: (NSNumber *) aNumber
```

比较两个NSNumber对象是否相等，返回值是BOOL类型。

- (NSComparisonResult) **compare:** (NSNumber *) aNumber

  比较两个NSNumber对象的大小，返回值是一个NSComparisonResult。更多关于NSComparisonResult的内容请参考附录A。

- (NSString *) **stringValue**

  返回NSNumber的字符串表示形式。

9.6.2 NSValue

NSValue 的接口定义在 Foundation/NSValue.h 中，坐标等结构体定义在 Foundation/NSGeometry.h 中。NSValue 的装箱（封装）和拆箱（解封装）方法如表 9-5 所示。

表 9-5 NSValue 的装箱和拆箱方法

数据类型	生成临时对象的方法	从对象中获取值的方法
指针	+ valueWithPointer:	-pointerValue
NSPoint	+ valueWithPoint:	-pointValue
NSRect	+ valueWithRect:	-rectValue
NSSize	+ valueWithSize:	-sizeValue
NSRange	+ valueWithRange:	-rangeValue

对象之间的比较可以使用 isEqualToValue: 方法。

- (BOOL) **isEqualToValue:** (NSValue *) value

  比较两个NSValue对象是否相等，返回值是BOOL类型。

9.6.3 类型编码和 @encode()

Objective-C 是动态语言，很多时候都不会清楚地标明对象的实际类型。但在保存数据和网络通信的时候，有时会需要明示自己的类型。NSValue 和 NSNumber 可以包装很多种类型的数据，但其内部需要保存数据的实际类型。另外，在进程或线程间通信的时候，也需要知道要发送的数据的实际类型。

Objective-C 的数据类型甚至自定义类型，都可以使用 ASCII 编码的类型描述字符串来表示。@encode() 可以返回给定数据类型的类型描述字符串（C 风格字符串，char * 表示），例如，@encode(int) 返回 "i"，@encode(NSSize) 返回 "{_NSSize=ff}"。

表 9-6 中总结了常用的数据类型和其对应的类型描述符。

表 9-6 常用类型的编码

代码	数据类型	代码	数据类型	代码	数据类型
c	char	C	unsigned char	@	对象
s	short	S	unsigned short	#	类对象
i	int	I	unsigned int	:	选择器
l	long	L	unsigned long	[类型]	数组

（续）

代码	数据类型
q	long long
f	float
d	double

代码	数据类型
Q	unsigned long long
v	void
*	C 文字列 (char *)

代码	数据类型
{名字 = 类型 ...}	结构体
^类型	指向类型的指针
?	不知道的类型或元素

通过使用方法initWithBytes:objCType:，就可以把任意类型的结构体包装成 NSValue 类型的对象。方法getValue:可以取出包装好的结构体中的数值。

- (id) **initWithBytes:** (const void *) value
  objCType: (const char *) type

  参数type是指定的value的类型描述符。为了保证兼容性，不要通过直接输入C风格字符串来获取type，而一定要通过@encode来获取。

  便利构造器:valueWithBytes:objCType:

- (void) **getValue:** (void *) buffer

  从包装好的NSValue对象中复制数据到buffer中。

下面是一个为结构体创建 NSValue 实例的例子。

```
struct grid {
    int x, y;
    double weight;
};
struct grid  foo, bar;
id obj;
```

对结构体 foo 进行并封装返回一个对象 obj 的代码如下所示。

```
obj = [[NSValue alloc] initWithBytes: &foo
                            objCType: @encode(struct grid)];
```

把数据从 obj 中读入到变量 bar 中的代码如下所示。

```
[obj getValue:&bar]
```

NSValue 只能包装长度确定的数据，不能包装长度可变的数据或元素可变的数组。例如，NSValue 不能包装一个 C 风格的字符串。

另外，因为 NSValue 和 NSNumber 是以类簇的形式实现的，所以无法用通常的方法为其定义子类，详细内容请参考第 11 章中有关抽象类和类簇的内容。

9.6.4 NSNull

前面我们介绍过不能在数组和词典对象中放入 nil，因为在数组和词典对象中，nil 有着特殊的含

义。但有时我们又确实需要一个特殊的对象来表示空值。这种情况下，我们就可以使用 NSNull 这一用来表示空值的类。NSNull 的接口定义在 Foundation/NSNull.h 中。

NSNull 中定义的类方法如下所示。

```
+ (NSNull *) null
```

这个方法会返回 NSNull 的实例对象，是一个常量。因为返回的是一个常量，所以不能用 release 来释放。另外，NSNull 的description会返回字符串 <null>。

判断一个对象是否是 NSNull 的写法如下所示。

```
if ( obj == [NSNull null] ) ...
```

9.7 NSURL

9.7.1 关于 URL

本章将说明一下 Cocoa 环境中用来表示网络或本地文件位置的类 NSURL。

统一资源定位符（URL，Uniform Resource Locator）是因特网上标准的资源地址。URL 除了用在因特网上之外，也可以用于表示本机的资源。另外，除了 HTTP 协议之外，其他协议也可以使用 URL 来访问资源。

URL Scheme 是类似于 http://、ftp://、file:// 这样的东西，Mac OS X 和 iOS 中规定了四种可以使用 NSURL 来访问资源的形式（scheme）。

http ：超文本链接协议
https：超文本传输安全协议 http 的安全版本，是超文本传输协议和 SSL/TLS 的组合
ftp ：文件传输协议
file ：访问某台主机上的文件

访问某个资源的写法如下所示。

协议名 :// 主机名 / 主机内资源的路径

http 协议中，主机名部分可以包含认证用的用户名、密码和端口号。URL 可以带有参数，还可以在末尾加上 # 符号让用户浏览器在获得文件后导航。主机内资源的路径中的任何特殊字符（英文和数字以外的非 ASCII 字符）都需要用 %+16 进制的 URL 编码来表示。例如，下面是一个虚拟的 URL 例子，URL 的端口号为 8080，页面内偏移为 overview。%20 指的是空格。

http://www.apple.com:8080/iphone%20lib/index.html#overview

file 加上 URL 时，主机名既可以是其他主机也可以使用 localhost 来表示本机。使用 localhost 的情况下只能访问本机上的文件。例如，本机上有文件 /Users/me/cool.jpg，用 URL 表示的话就如下所示。

file://localhost/Users/me/cool.jpg

URL 中的路径和 Unix 等的文件路径一样，既可以使用相对路径也可以使用绝对路径。上面的 http 和 file 的例子用的都是绝对路径。让我们来看一个相对路径的例子，假设下面是一个用绝对路径表示的目录。

http://www.apple.com:8080/iphone%20lib/

以这个路径为基础的相对路径 ../mac/screen.jpg 所对应的 URL 如下所示。

http://www.apple.com:8080/mac/screen.jpg

用相对路径表示资源时，参照用的 URL 称为 baseURL。

9.7.2 NSURL 的概要

URL 中有很多特有的表达方法，所以操作 URL 的时候应该使用专用的 NSURL 类，而不要把 URL 当作字符串来手工解析。例如，假设要从 URL 中取出主机名，这一处理实现起来会非常麻烦。另外，如果可以把互联网和本机上的所有资源都通过 URL 来表示，那么资源的访问将会变得非常统一。

下面对 NSURL 的主要方法进行说明，更详细的内容可以参考苹果公司的官方文档。

以下说明中会把 %+16 进制数的表现形式称为 url 编码。NSURL 中的某些方法会调用 NSString 的`stringByAddingPercentEscapesUsingEncoding:`方法对字符串进行 url 编码。编码时使用 UTF-8。如果想多重编码的话，需要先对字符串执行这个编码。

在介绍 NSURL 中的方法之前，我们先来介绍一下方法名或参数名中包含的 string 和 path 的区别。string 这里指的是 URL 的字符串，将 string 作为返回值的情况下，一定要返回经过 url 编码的字符串。方法名中如果包含了 string，那么实际传入的参数也一定是经过编码的。另一方面，path 用于表示 URL 的路径，无论作为返回值还是作为参数都要返回未编码的字符串。

（1）NSURL 实例的生成和初始化

- (id) **initWithString:** (NSString *) URLString

  用表示URL的字符串URLString生成一个NSURL的对象，URLString必须是url编码之后的。如果传入的字符串解析失败，则返回nil。

  便利构造器：`URLWithString:`

- (id) **initWithString:** (NSString *) URLString
  relativeToURL: (NSURL *) baseURL

  使用base URL和相对路径URLString来生成一个NSURL对象。如果URLString想使用绝对路径，可以把baseURL设为nil。传入的字符串必须是经过了URL编码的字符串。如果传入的字符串解析失败，则会返回nil。

  便利构造器：`URLWithString:relativeToURL:`

- (id) **initFileURLWithPath:** (NSString *) path

  用字符串格式的文件路径生成一个用于文件的NSURL对象。path不需要URL编码。如果传入的是一个相对路径，则使用调用该函数的程序的根目录作为base URL。

  执行的时候会判断传入的path是不是一个目录，如果是一个目录的话就会自动地在其末尾加上/。如果确定path是一个目录，可以使用initFileURLWithPath:isDirectory:生成NSURL对象。

  便利构造器:initFileURLWithPath:

（2）NSURL 的构成要素

- (BOOL) **isFileURL**

  判断消息接收者对象是否是file开头的URL。

- (NSURL *) **baseURL**

  返回消息接收者对象的base URL。如果不存在的话，返回nil。

- (NSString *) **absoluteString**

  返回消息接收者对象的URL的字符串表示。返回的字符串是url编码之后的。

- (NSURL *) **absoluteURL**

  如果接收消息的对象是相对路径的URL，就生成并返回这个对象的绝对路径的URL。如果接收消息的对象本身就是绝对路径的URL，则返回自己。

- (NSString *) **relativePath**

  返回接收消息的对象中相对路径的部分。如果接收消息的对象是由绝对路径生成的，则返回整个绝对路径。返回的字符串是没有经过url编码的。

- (NSString *) **relativeString**

  返回URL字符串中相对路径的部分。如果接收消息的对象是由绝对路径生成的，则返回URL的绝对路径。返回的字符串是url编码之后的。

（3）路径操作的相关函数

所有与路径操作相关的方法的参数和返回值都是没经过 url 编码的。

- (NSString *) **path**

  返回消息接收者对象的路径字符串，也就是主机名之后的部分。

  方法lastPathComponent可以取出URL中的最后部分，方法pathExtension可以取出路径的扩展名。

- (NSURL *) **URLByAppendingPathComponent:**
  (NSString *) pathComponent

  在消息接收者对象URL的后面追加新的要素，返回一个新的URL对象。追加路径时别忘了追加相应的/。

  如果想追加扩展名的话可以使用方法URLByAppendingPathExtension:。

- (NSURL *) **URLByDeletingLastPathComponent**

  删除消息接收者对象URL的最后一个要素，返回一个新的URL对象。

  如果想删除扩展名的话可以使用方法URLByDeletingPathExtension。

（4）文件引用 URL

有两种表示文件位置的 NSURL 对象：一种是从文件路径字符串生成的 NSURL 对象，称为文件路径 URL；还有一种是通过文件系统的文件 ID 生成的 URL，这种 URL 叫作文件引用 URL。文件引用 URL 就算文件名被更改或者文件移动到别的地方，也能够继续索引文件。

```
- (NSURL *) fileReferenceURL
```

消息接收者对象如果是一个路径URL，返回其对应的文件引用URL对象。如果消息接收者对象不是一个本地文件，就返回nil。

与此相对，可以通过方法filePathURL来获取一个文件引用URL对象所对应的文件URL。

```
- (BOOL) isFileReferenceURL
```

如果消息接收者对象是一个文件引用URL的话就返回YES。

```
- (BOOL) checkResourceIsReachableAndReturnError:
    (NSError **) error
```

判断消息接收者对象所指向的文件是否存在，如果存在的话就返回YES。参数error返回具体的错误信息，如果不想返回信息，可把error设为NULL。关于类NSError的更多内容请参考18.5节。

（5）获取和变更文件属性

使用 NSURL 可以获取或变更文件、目录和文件系统的属性。一个有代表性的例子是通过 NSURL 可以获取文件的大小，修改时间和权限等属性。通过 Foundation 框架的 NSFileManager 也能够完成同样的功能，但还是多少有所不同。例如，通过 NSURl 可以改变 Finder 的颜色标签的颜色。

本书中只介绍了获取和变更属性值时需要用到的方法，访问属性值时需用到的字符串 key 请参考苹果公司的官方文档。

```
- (BOOL) getResourceValue: (out id *) value
                   forKey: (NSString *) key
                    error: (out NSError **) error
```

这个方法的功能是将key指定的属性值写入value中。获取成功的话就返回YES，失败的话就返回NO，失败的原因会写入error。value前面的out修饰符表示这个指针是用于返回传值的，更多内容请参考19.5节。

同时获得多个属性的值的话可以使用方法resourceValuesForKeys:error:。

```
- (BOOL) setResourceValue: (id) value
                   forKey: (NSString *) key
                    error: (NSError **) error
```

设定key的属性为value。设定成功的话就返回YES，失败的话就返回NO，失败的原因会写入error。同时设定多个属性值的话可以使用方法setResourceValues:error:。

9.7.3 使用 NSURL 来访问资源

本章中对 Foundation 框架中最常用的类进行了介绍，这些类中和文件输入输出相关的方法都是

通过字符串格式的文件路径来指定的。其实这些函数都还有一个用 NSURL 作为参数的版本。通过使用 NSURL，不仅可以指定本地的文件，还可以指定网络上的文件。

例如，许多类中都定义了下面这个方法。

```
- (id) initWithContentsOfFile: (NSString *) path ...
```

基本所有定义了这个方法的类中都还有一个具有类似功能的以 NSURL 作为参数的函数，如下所示。同时，很多类中也会定义相应的便利构造器。

```
- (id) initWithContentsOfURL: (NSURL *) aURL ...
```

原本 NSURL 是为了访问网络上的资源而设计的，随着 Cocoa API 版本的不断升级，原来很多文件输入输出的地方也开始使用 NSURL 了。这也说明 Mac OS X 特别是 iOS 已经把网络作为自己的一部分了。

第10章

范畴

本章将介绍范畴的概念。范畴不仅可以将类以模块为单位分散到多个不同类别或文件中实现，还提供了为现有类添加新方法的简便方式。

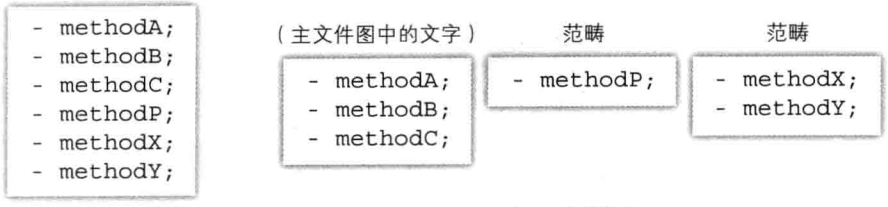
10.1 范畴

10.1.1 范畴

到目前为止我们都是在 @implementation 和 @end 之间实现类的方法，而对于有很多方法的超大的类，我们则可以把方法的实现分散到不同的模块中。

实现某个类的一部分方法的模块叫作**范畴**或**类别**（category）。一个类既可以不使用任何范畴，也可以由多个范畴构成。调用范畴中定义的方法和调用普通方式定义的方法一样。图 10-1（a）是目前为止使用的在一个文件中实现类的所有方法。与此相对，（b）是在多个范畴中实现类的方法。

图 10-1 范畴的概念



（a）没有使用范畴　　（b）使用范畴

范畴和类一样，都是在接口文件中声明，在类文件中实现。但范畴中不能声明实例变量，只能声明方法，声明的方法既可以是类方法也可以是实例方法。

范畴的语法如下所示。“类名”部分为范畴所属的类的名字或即将添加该范畴的类的名字。“类名”必须是已经存在的类，不能为一个不存在的类定义范畴。

语法　范畴的声明

```
@interface 类名( 范畴名 )
方法的声明;
   ...
@end
```

范畴名的命名规则和 C 语言变量的命名规则一样。在不使用类中已定义的范畴名的前提下，可为范畴随意命名，当然起一个和范畴内容相关的名字是最好的。范畴的实现部分的语法如下所示。

语法　范畴的实现

```
@implementation 类名 ( 范畴名 )
方法的定义;
   ...
@end
```

在实现部分中实现接口文件中定义的方法。实现部分除了不可以定义新的实例变量外，都和传统方式的实现文件一样。例如，方法的实现中可以自由调用别的方法、访问已定义的实例变量等。除此之外，也可以定义局部方法或 C 语言函数等。

10.1.2 范畴和文件的组织

下面说明一下范畴的接口部分和实现部分的文件组织方式。一个典型的例子就是包含主文件接口的头文件中也包含了其他所有接口，如图 10-2 所示。

▶ 图 10-2 只有一个接口文件时的范畴定义示例

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

@interface Card : NSObject
{
    ...
}
- (void)methodA;
- (void)methodB;
- (void)methodC;
@end

@interface Card (Display)
- (void)methodP;
@end

@interface Card (Sort)
- (void)methodX;
- (void)methodY;
@end
```

头文件Card.h

```
#import "Card.h"

@implementation Card
- (void)methodA { ... }
- (void)methodB { ... }
- (void)methodC { ... }
@end
```

文件 Card.m

```
#import "Card.h"

@implementation Card (Display)
- (void)methodP { ... }
@end
```

文件 Card+Display.m

```
#import "Card.h"

@implementation Card (Sort)
- (void)methodX { ... }
- (void)methodY { ... }
@end
```

文件 Card+Sort.m

另外一种定义范畴的方法就是将每个范畴部分都单独定义成一个头文件，如图 10-3 所示。每个实现文件都可以被单独编译。实现文件一般被命名为“类名 + 范畴名 .m”，更多内容请参考附录 C。

范畴的接口部分需要遵循以下几个原则。下面提到的引用既包括引用同一个文件中的内容也包括引用其他文件中的内容。

- **范畴的接口部分必须引用主文件的接口文件**
- **范畴的实现部分必须引用对应的接口文件**
- **使用范畴中的方法时必须引用这个方法所在的头文件**

需要注意的是，除了要调用范畴部分中定义的方法之外，主文件接口部分中不会引用各个范畴的接口和实现文件。

另外，也可以将多个范畴的实现部分写在同一个文件中，但这样就失去了定义范畴的意义了。

图 10-3 - 单独定义头文件时的范畴定义示例

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

@interface Card : NSObject
{
    ...
}
- (void)methodA;
- (void)methodB;
- (void)methodC;
@end
```

头文件 Card.h

```
#import "Card.h"

@implementation Card
- (void)methodA { ... }
- (void)methodB { ... }
- (void)methodC { ... }
@end
```

文件 Card.m

```
#import "Card.h"

@interface Card (Display)
- (void)methodP;
@end
```

头文件 Card+Display.h

```
#import "Card+Display.h"

@implementation Card (Display)
- (void)methodP { ... }
@end
```

文件Card+Display.m

```
#import "Card.h"

@interface Card (Sort)
- (void)methodX;
- (void)methodY;
@end
```

头文件 Card+Sort.h

```
#import "Card+Sort.h"

@implementation Card (Sort)
- (void)methodX { ... }
- (void)methodY { ... }
@end
```

文件Card+Sort.m

10.1.3 作为子模块的范畴

如果是有很多方法的规模很大的类，把所有实现部分都写在一个文件里就比较不方便。这种情况下可以通过范畴将类的实现部分以模块为单位分散到多个不同的文件中，也就是把范畴作为类的子模块来使用。

范畴本来是 Smalltalk 中的概念，被用于将多个方法按照相互关系、用途等特征分类，以便最快速地找到自己最想要的方法。

Objective-C 中也应该将关系紧密的方法作为一个范畴来分类。

把类按照范畴分类，和 C 语言编程中把某些函数保存在同一个文件中比较类似。把实现方法互相依赖或共用同一个局部变量的方法定义为一个范畴。通过这种方法，类中依赖性相对比较高的一部分就会被归纳出来，开发也会变得更加容易。

上面讲述的内容的前提都是类的规模非常大，这种情况下通过使用范畴可以提高开发效率，但实际上规模大的类并不是一个好的选择。在设计阶段，如果发现一个类的方法非常多，最好把这个类的功能分成几个类来实现。

与此相反，由过小的对象组成的系统也会有通信不好控制、功能不易划分等问题。所以，虽然并没有什么准则来指明一个类到底应该多大，但类的大小适当很重要。

10.1.4 方法的前向声明

第 3 章中我们曾经提到过这样一个问题，即未在接口部分中声明的局部方法，和未声明的 C 语言函数一样，其定义之前的代码是没法使用它们的。而通过使用范畴，我们就可以解决这个问题。

让我们来看一下下面这个例子，某个文件中定义了下面两个函数。

```
- (int)methodA {
     if (...)
        [self methodB: 0];
    ...
}

- (void)methodB:(int)arg {
    double v = [self methodA];
    ...
}
```

methodA和methodB:都是局部方法，这两个方法都未在接口中声明。所以编译器在编译方法 A 的时候，并不知道methodB:的参数和返回值的类型。就算交换methodA和methodB:的顺序也会有同样的问题。

这种情况下，通过把有问题的方法都加入到一个局部范畴中，并在实现文件的头部加上范畴的声明和定义，就可以解决这个问题。

```
@interface 类名 (Local)
- (int)methodA;
- (void)methodB:(int)arg;
@end

@implementation 类名 (Local)
- (int)methodA {
     if (...)
        [self methodB: 0];
    ...
}

- (void)methodB:(int)arg {
    double v = [self methodA];
    ...
}
@end
```

这里使用了 Local 作为范畴名，这个名字可以随意指定。这个范畴只在这个文件内被使用，是局部的。

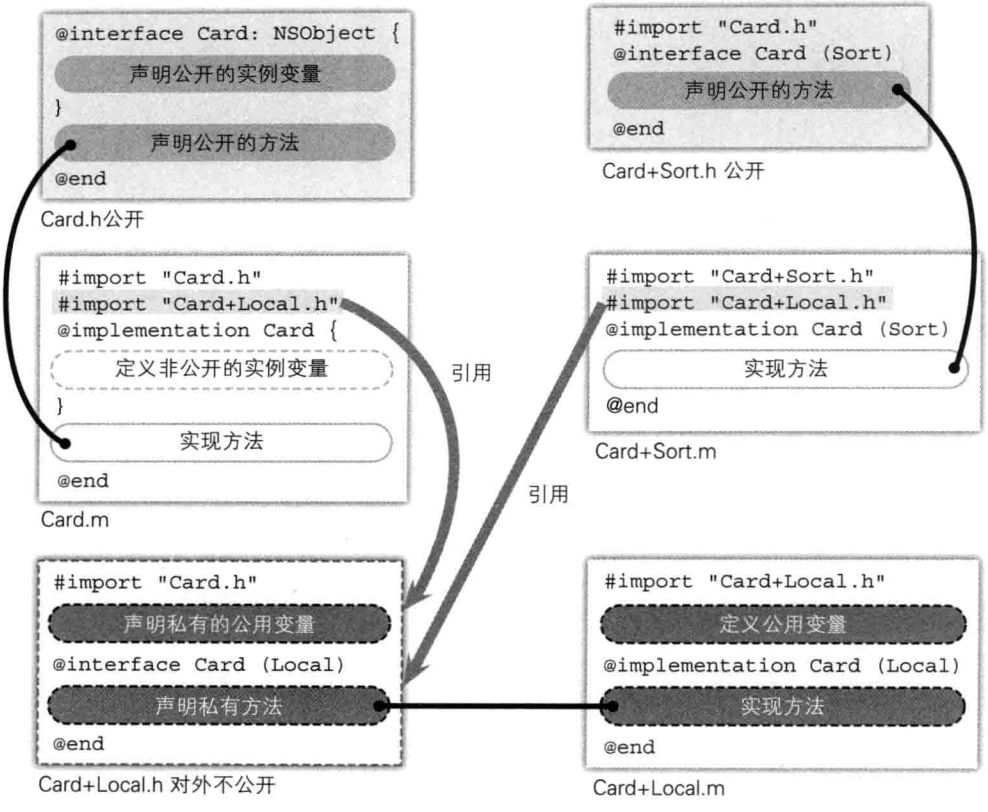
C 语言中可以通过在文件的最前面加上函数声明来解决类似的问题，从效果来讲是一样的。

10.1.5 私有方法

将类划分为多个范畴后，就必须要考虑该如何划分共有方法。

如图 10-4 所示，一种划分方法是创建一个对外不公开的接口文件，其中声明类的共有变量和方法。这样一来，方法就被放入到了某个范畴中。这些方法和变量的实现既可以被单独放入一个文件，也可以和别的范畴实现放在一起。本例中只有 Card.h 和 Card+Sort.h 的内容是对外公开的。Card+Local.h 中的内容是类的范畴之间共有的方法和变量。

图 10-4 非公开的头文件的概念



像这样，即使在多个文件中进行了共同的声明，也可以定义仅供类内文件使用而不对类外公开的方法和实例变量。本书中把这种方法叫作**私有方法**。

3.5 节中介绍了在类的实现文件中定义局部方法时因为子类不知道父类中所定义的局部方法而发生的方法覆盖的问题。私有方法也有同样的问题。为了避免这个问题，苹果公司推荐为私有方法名加上固定的前缀，详情请参考附录 C。

10.1.6 类扩展

由多个范畴组成的类就好像是给主类加上了各种选项，需要靠程序员来保证主类和各种范畴能够作为一个整体正常工作。

把类分为多个范畴来实现的情况下，主类和各个范畴都是独立的，每个范畴都不清楚其他的部分。有的范畴可能是执行前加载的，有的范畴可能是执行时动态加载的。

这种实现方法的可扩展性非常好，但编译器在链接时不会检查是否所有的范畴都被链接到了可执行文件中。如图 10-4 的例子所示，链接时就算忘记了 Card+Local.m 也不会报错。但如果可执行程序在执行的时候调用到了 Card+Local.m 中定义的函数，程序就会抛出异常，执行失败。

上一节中我们介绍了如何定义类内共享的私有方法，但这种方法有时候会忘记链接私有方法。

针对这种情况，**类扩展**[①]（class extension）的概念被引入了进来。

类扩展的声明和范畴比较相似，只是圆括号之间没有文本。例如，只有一个方法的类扩展的定义如下所示。

```
@interface Card ( )
- (BOOL)hasSameSuit:(Card *)obj;
@end
```

使用类扩展也可以增加实例变量，如下所示。

```
@interface Card ( ) {
    BOOL flag;
}
- (BOOL)hasSameSuit:(Card *)obj;
@end
```

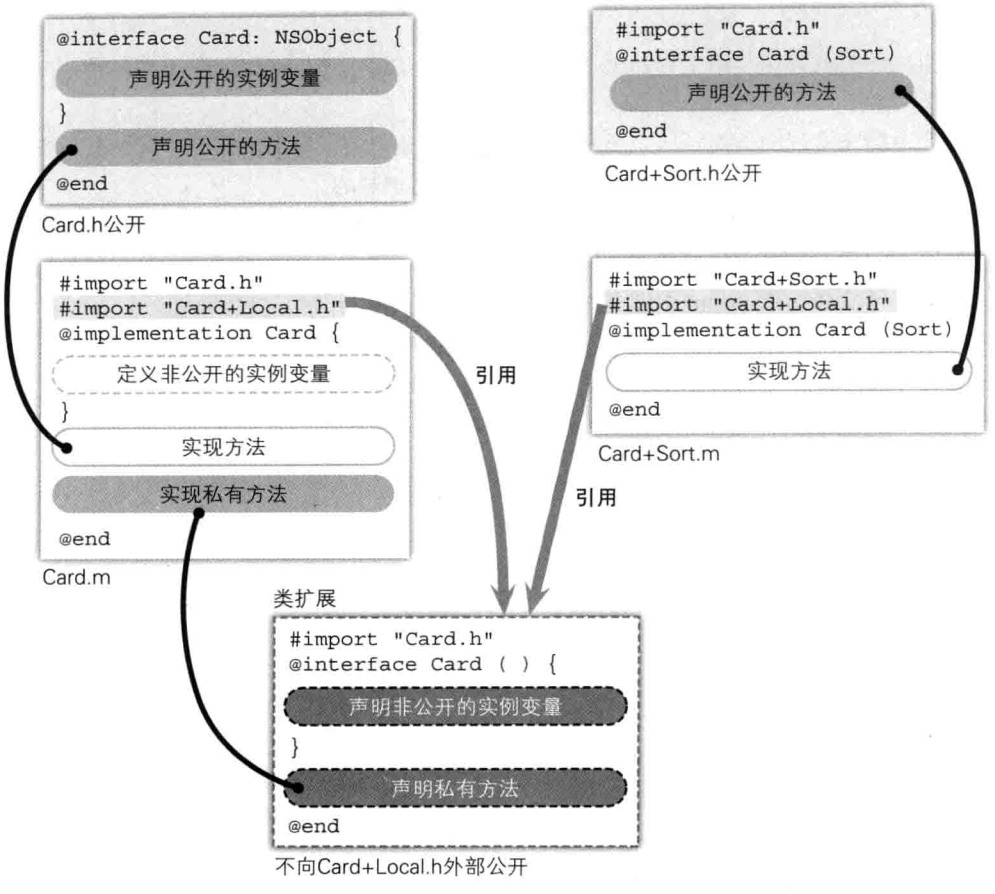
类扩展中声明的方法需要在类的实现文件中实现，如图 10-5 所示。不管是否引入了类扩展的定义，只要在类的实现部分中没实现对应的方法，就会出现编译错误。也就是说，通过将一定要实现的私有方法放入类扩展中，就可以防止忘记实现这个方法或漏掉方法的实现等问题。

但是，类扩展自身并不会让方法变为私有方法。如果想让方法变成私有方法，必须不能让包含了类扩展的头文件（图 10-5 中的 Card+Local.h）对外公开。

类扩展中声明的实例变量只能在引入（include）了类的主接口和扩展声明的范畴中使用。主类的实现部分也可以声明实例变量，但声明的实例变量只在该文件（实现文件）中有效。与此相对，类扩展中定义的实例变量可以在多个范畴中使用。

① “extension”通常被翻译为扩展、延长，这个单词也可以被用来表示建筑物的增建或接发等。

图 10-5 利用类扩展



10.1.7 范畴和属性声明

范畴的接口中可以包含属性声明。但要注意的是，**范畴的实现部分中不能包含 @synthesize**。范畴的接口中包含属性声明的情况下，实现部分中就需要手动定义访问方法。这是为了防止随意访问同一个类的不同文件中定义的实例变量。

类扩展中也可以包含属性声明。属性是通过在类的实现部分包含 @synthesize 或者属性方法来实现的。类扩展中也可以声明实例变量的属性。

范畴再加上类扩展可能会让你觉得混乱，让我们把上面介绍的内容用一张表总结一下。需要注意的是，类扩展只能引用类内的实例或方法，至于范畴的头文件是公开的还是只限类内使用，则是由具体的使用情况决定的。

表 10-1 主类、类扩展和范畴的使用方法

	实例变量	方法声明	@synthesize
主类 .h	○公开	公开	-
类扩展 .h	○类内	类内	-
主类 .m	○文件内	文件内	○
范畴 .h	×	公开 / 类内	-
范畴 .m	×	文件内	×

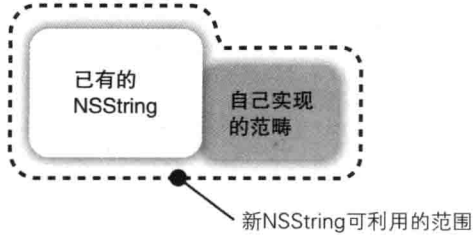
10.2 给现有类追加范畴

10.2.1 追加新的方法

无论是自己定义的类还是系统的类，利用范畴都可以为已有类追加新方法。

图 10-6 就是利用范畴给已有类追加新方法的示意图。上节中我们提到了范畴只可以追加方法不可以追加实例变量，但新追加的方法可以访问类中已有的实例变量和方法。通过这种方法，我们就可以为类增加新的功能。

图 10-6 通过范畴为类增加新的方法



我们一般通过继承的方式来为现有类增加新的功能。但有的时候使用子类并不方便，例如无法用通常的方法来为 NSString 创建子类（NSString 是以类簇的形式实现的）等。而范畴的好处就是可以为正在使用的类增加新的功能。

我们用给 NSString 增加方法的例子来说明一下。假设我们不使用范畴，也可以通过给 NSString 定义子类的方法来增加新功能。但 Foundation 众多类的参数或返回值写的都是 NSString，将所有的 NSString 都替换为新定义的子类显然不是一个好的方法。这种情况下，使用范畴来追加方法就是一个很好的选择了。

另外，通过范畴给现有类追加方法后，这个类的子类不需要做任何修改就能够使用新追加的方法。而通过继承定义子类的情况下则无法实现这个功能。

虽然使用范畴来追加方法很方便，但也要注意不要滥用。把看似方便的功能都加到已有类上不是一种好的编码方式。

10.2.2 追加方法的例子

字符串类 NSString 有方法stringByAppendingPathComponent:。如果接收消息的对象是一个目录的路径，该方法就会把参数的字符串加到目录路径的后面，并返回一个新的字符串（详情请参考 9.2 节）。例如，路径名是@"/tmp/image"的情况下，追加字符串@"thumb"后，返回值就为字符串@"/tmp/image/thumb"。这个方法只能接收一个参数，这里我们来尝试一下为 NSString 追加一个可同时追加多个参数的方法。

假设要追加的方法名为stringByAppendingPathComponents:（以 s 结尾代表可同时追加多个路径），参数是可变参数列表的形式，以 nil 结尾。可变参数方法的定义请参考下面的 Column。

NSString 是以类簇的形式实现的，下面的程序也可以作为给类簇增加新方法（详情请参考 11.3 节）的一个例子。

代码清单 10-1 NSString+PathComp.h

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

@interface NSString (PathComp)

- (NSString *)stringByAppendingPathComponents:(NSString *)str, ...
        NS_REQUIRES_NIL_TERMINATION;

@end
```

代码清单 10-2 NSString+PathComp.m

```
#import "NSString+PathComp.h"
#import <Foundation/NSPathUtilities.h>
#import <stdarg.h>
@implementation NSString (PathComp)

- (NSString *)stringByAppendingPathComponents:(NSString *)str, ...
{
    va_list varglist;
    NSString *work, *comp;

    if (str == nil)
        return self;
    work = [self stringByAppendingPathComponent:str];
    va_start(varglist, str);
    while ((comp = va_arg(varglist, NSString *)) != nil)
        work = [work stringByAppendingPathComponent:comp];
    va_end(varglist);
    return work;
}

@end
```

以上代码给 NSString 增加了新的范畴 PathComp。范畴的名字可以随便起，只要不和已有范畴重

名即可。

把以上两个文件和其他文件一起编译、链接之后，就可以像使用 NSString 中原本就有的方法一样使用stringByAppendingPathComponents:了。但是，要使用这个方法的模块必须要包含头文件 NSString+PathComp.h。

下面是为了测试新方法而写的一个简单的 main() 函数。

代码清单 10-3 测试程序

```
#import <Foundation/NSString.h>
#import <Foundation/NSPathUtilities.h>
#import <Foundation/NSAutoreleasePool.h>
#import "NSString+PathComp.h"
#import <stdio.h>

int main(void)    // 利用 ARC
{
    NSString *pict = @"Pictures";
    NSString *homedir, *s;
    @autoreleasepool {
       homedir = NSHomeDirectory();
       s = [homedir stringByAppendingPathComponent: pict];
       printf("%s\n", [s UTF8String]);
       s = [homedir stringByAppendingPathComponents: pict, @"tmp", nil];
       printf("%s\n", [s UTF8String]);
       s = [homedir stringByAppendingPathComponents:
            @"Desktop", pict, @"Wallpaper", nil];
       printf("%s\n", [s UTF8String]);
    }
    return 0;
}
```

专栏：可变参数的方法的定义 COLUMN

ANSI C 中可以定义 printf() 这样的可变参数的函数。Objective-C 中也同样可以定义可变参数的方法。可变参数有如下几个限制。

- 参数列表中不能只有可变参数
- 可变参数必须出现在参数列表的最后
- 可变参数的类型必须由程序来管理

也就是说，调用可变参数的方法的时候不能没有任何参数，要至少有一个变量。可变参数变量不能出现在参数列表的中间位置，只能出现在参数列表的最后。调用时参数列表中允许存在不同类型的参数，但是程序要保证类型的正确性。

定义可变参数的方法或函数时要引入头文件 stdarg.h。用 ... 来表示可变参数。获取可变参数前，需要定义一个 va_list 类型的变量。变量名可随意命名，下面这个例子中使用变量名 pvar。

从可变参数列表中取出每个参数的代码如下所示。

```
va_start(pvar, 可变参数前面一个变量的变量名);
   ...
f = va_arg(pvar, 类型名);  // 可变参数有多少个，就循环执行这条语句多少次
   ...
va_end(pvar);
```

va_start() 为访问可变参数进行准备[①]，va_start() 的第 2 个参数是可变参数前面紧挨着的一个变量，即 ... 之前的那个参数。

调用 va_arg() 获取可变参数的值，va_arg 执行的时候能够得到下一个变量值。va_arg() 的第二个参数是要获取的参数的类型，这个类型不一定都要一样。此处不能保证实际返回的参数类型和指定的参数类型一致，需要程序中做各种额外的容错处理。

获取所有的参数之后调用 va_end() 就可以关闭 pvar 指针。通常 va_start 和 va_end 都是成对出现的。

可变参数一般以 NULL 或 nil 结尾，但在实际使用的时候经常会出现忘记加 NULL 的情况。这个问题可以通过在函数或方法声明的末尾加上宏变量 NS_REQUIRES_NIL_TERMINATION 来解决。这样一来，如果在编译的时候发现忘了 NULL，就会提示警告。代码清单 10-1 声明方法时就加了宏变量 NS_REQUIRES_NIL_TERMINATION。

NS_REQUIRES_NIL_TERMINATION 自身定义在 Foundation/NSObjCRuntime.h 中，可以指定 gcc 风格的函数属性来替换它。

10.2.3 覆盖已有的方法

新定义的范畴中的方法如果和原有方法重名的话，新定义的方法就会覆盖老方法。通过使用这种方法，我们就可以在不使用继承的情况下实现方法的覆盖。

但如果不小心覆盖了原有的方法，也可能会引发不可预测的问题。例如，为 NSObject 定义了一个新的范畴，并在其中用自己定义的方法覆盖了原有的方法等。而一旦覆盖了比较重要的方法，很容易就会发生严重的问题[②]。另外，如果多个类别中定义了相同名字的方法，实际执行时就无法保证到底执行了哪个方法。

所以不建议利用范畴覆盖已有的方法。但是就算是覆盖了已有的方法，编译器或链接器也不会提示任何警告。管理方法名的时候一定要注意不要覆盖已有的方法。

① 使取数据的指针指向第一个可选参数。——译者注

② NSObject 中只声明了范畴，没有具体的实现，这也被叫作非正式协议，但这并不是覆盖了定义，更多内容请参考 12.3 节中有关非正式协议的内容。

10.3 关联引用

10.3.1 关联引用的概念

通过范畴可以为一个类追加新的方法但不能追加实例变量。但是，利用 Objective-C 语言的动态性，并借助运行时（runtime）的功能，就可以为已存在的实例对象增加实例变量，这个功能叫作**关联引用**（associative references）。将这个功能和范畴组合在一起使用，即使不创建子类，也能够对类进行动态扩展。

一般情况下，在类定义中，该类的所有实例都能够使用接口中声明的实例变量。

与此相对，关联引用指的是在运行时根据需要为某个对象添加关联。就算是同一个类的不同对象也有可能添加（或不添加）关联或添加不同种类和数量的关联。另外，已添加的关联也可以被删除。

10.3.2 添加和检索关联

下面我们来看一下添加关联和检索关联用的两个方法，这两个方法的定义在头文件 objc/runtime.h 中。

```
void objc_setAssociatedObject(id object, void *key, id value,
    objc_AssociationPolicy policy)
```

这个方法是为对象 object 添加以 key 指定的地址作为关键字、以 value 为值的关联引用，第四个参数 policy 指定关联引用的存储策略。

通过将 value 指定为 nil，就可以删除 key 的关联。

```
id objc_getAssociatedObject(id object, void *key)
```

返回 object 以 key 为关键字关联的对象。如果没有关联到任何对象，则返回 nil。

本章中把要增加关联的对象（想扩展的对象）称为所有者，把追加的对象称为引用对象。

例如，假设我们要为 obj 增加 r 和 s 两个关联引用。那么 obj 就是所有者，r 和 s 就是引用对象。考虑到可以为一个对象增加多个关联引用，所以要用 key 来区分。另外，必须使用确定的、不再改变的地址作为键值。例如，使用定义在实现文件中的静态局部变量的地址作为 key 就是一个不错的选择。关于策略 policy，我们将会在后面进行详细介绍。

```
static char rKey, sKey; /* 静态变量，这里只利用他们的地址作为 key*/
...
objc_setAssociatedObject(obj, &rKey, r, OBJC_ASSOCIATION_RETAIN);
objc_setAssociatedObject(obj, &sKey, s, OBJC_ASSOCIATION_RETAIN);
...
id x = objc_getAssociatedObject(obj, &rKey);
id y = objc_getAssociatedObject(obj, &sKey);
```

以上的操作就相当于将 x 和 y 分别赋值为了 r 和 s。使用地址作为 key 的原因是为了保证唯一性，上面例子中的变量 rKey 和 sKey 不会被用来存储什么，也不会对其进行赋值操作。

在上例中，函数调用时的第一个参数都用了 obj，其实这个位置可以使用任意对象。如果关联引用的目的是给一个对象增加实例变量的话，这个地方可以使用 self 代替 obj。使用 self 的情况下，因为运行时 self 表示的对象不同，所以就算是使用同一个 key 作为键值也无所谓。关于这点，后面会通过一个具体例子进行详细说明。

10.3.3 对象的存储方法

objc_setAssociatedObject() 的第四个参数 policy 是关联策略。关联策略表明了关联的对象是通过何种方式进行关联的，以及这种关联是原子的还是非原子的。policy 的值有以下几种可供选择，其中最常用的是 OBJC_ASSOCIATION_RETAIN。

截止到写作本书的时候，内存管理使用 ARC 的情况下，相当于弱指针的对象保存方式好像还没有出现。如果指定存储策略为 OBJC_ASSOCIATION_RETAIN，就是通过保持的方式进行关联。而 ARC 的情况下，指定 OBJC_ASSOCIATION_ASSIGN 则表示通过赋值的方式进行关联，可能会出现悬垂指针，编程的时候一定要注意。

OBJC_ASSOCIATION_ASSIGN

使用基于引用计数的内存管理时，不给关联对象发送retain消息，仅仅通过赋值进行关联。内存管理使用垃圾回收时，会把引用对象作为弱引用保存。

OBJC_ASSOCIATION_RETAIN_NONATOMIC

使用基于引用计数的内存管理时，会给关联对象发送retain消息并保持。如果同样的 key 已经关联到了其他对象，则会给其他对象发送release消息。释放关联对象的所有者时，会给所有的关联对象发送release消息。内存管理使用垃圾回收时，会以强引用的形式保存关联对象。

OBJC_ASSOCIATION_RETAIN

在对象的保存方面和 OBJC_ASSOCIATION_RETAIN_NONATOMIC 的功能一样，唯一有区别的是 OBJC_ASSOCIATION_RETAIN 是多线程安全的，支持排他性的关联操作。

OBJC_ASSOCIATION_COPY_NONATOMIC

在进行对象关联引用的时候会复制一份原对象，并用新复制的对象进行关联操作。

OBJC_ASSOCIATION_COPY

在对象的保存方面和 OBJC_ASSOCIATION_COPY_NONATOMIC 的功能一样，唯一区别的是 OBJC_ASSOCIATION_COPY 是多线程安全的，支持排他性的关联操作。objc_getAssociatedObject() 的操作和 OBJC_ASSOCIATION_RETAIN 一样。

要复制一个对象需要实现第 13 章中说明的方法。更多关于多线程的内容请参考第 19 章。

OBJC_ASSOCIATION_ASSIGN 之外的四个 policy（OBJC_ASSOCIATION_RETAIN_NONATOMIC、OBJC_ASSOCIATION_RETAIN、OBJC_ASSOCIATION_COPY_NONATOMIC 和 OBJC_

ASSOCIATION_COPY）是保留（retain）或复制（copy）和关联操作是原子的还是非原子的组合。这与属性声明的选项类似，更多详细内容请参考 7.2 节。

10.3.4 断开关联

Objective-C 中也提供了运行时断开关联的函数。

`void` **`objc_removeAssociatedObjects`**`(id object)`

断开object的所有关联。

这个函数会断开 object 对象的所有关联，有一定的危险性。例如，已有的代码可能已经使用了关联的对象，有的代码新添加的范畴也可能会使用到已关联的对象。所以不建议使用 objc_removeAssociatedObjects 一次性断开所有关联，推荐使用 objc_setAssociatedObject，传入 nil 作为其参数，来分别断开关联。

10.3.5 利用范畴的例子

下面我们来看一个使用范畴的例子。假设我们要给 NSArray 增加一个新的随机取元素的方法。这个方法取得的元素可以相同，但不可以连续取得相同的元素。

我们利用范畴为 NSArray 定义这样的一个方法，因为需要记忆前一次取得的元素，所以使用了关联引用。

代码清单 10-4 NSArray+Random.h

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

@interface NSArray (Random)

- (id)anyOne;

@end
```

代码清单 10-5 NSArray+Random.m

```
#import "NSArray+Random.h"
#import <objc/runtime.h>

@implementation NSArray (Random)

static char prevKey;  // 定义键使用的地址变了

static int random_value(void) { // 使用线性同余法（Linear congruential generators）
                                // 生成随机数
    static unsigned long rnd = 201008; // 随机数的种子，可随机设置
    rnd = rnd * 1103515245UL + 12345;
```

```
    return (int)((rnd >> 16) & 0x7fff);
}

- (id)anyOne
{
    id item;
    NSUInteger count = [self count];
    if (count == 0)
        return nil;
    if (count == 1)
        item = [self lastObject];
    else {
        id prev = objc_getAssociatedObject(self, &prevKey);
                                    // 初次使用关联引用时返回 nil
        NSUInteger index = random_value() % count;
        item = [self objectAtIndex:index];
        if (item == prev) {         // 如果和上回取得的值相同
            if (++index >= count) )// 索引加 1
                index = 0;
            item = [self objectAtIndex:index];
        }
    }
    objc_setAssociatedObject(self, &prevKey, item,
        OBJC_ASSOCIATION_RETAIN); // 存储最后返回的对象
    return item;
}

@end
```

代码清单 10-4 是范畴的头文件，其中只增加了一个方法。

代码清单 10-5 中首先定义了用作关联引用的键值的 static 变量，static 变量定义在范畴的实现文件中，其他范畴无法使用这个变量的地址。函数 random_value 中使用线性同余法生成了随机数。NSArray 以及它的子类都能够使用函数 random_value 和 anyOne。

anyOne方法首先会查看数组中有多少个元素，如果有两个以上的话就调用random_value方法随机生成索引，并用索引获得要返回的值。如果这次的值和上次返回的值相等，则索引值加 1。最后，把要返回的元素登记到关联引用，以便下次比较时使用。

因为关联到各个实例对象的是不同的元素（item），所以多个实例对象也可以使用这个方法。item 和实例变量一样可以被作为各实例变量自身的值来使用。另外，不使用这个方法的实例变量也不会生成关联引用。

代码清单 10-6 展示了使用以上范畴的 main 函数。这里同时声明了两个 NSArray 对象，其中一个是 NSArray 的子类 NSMutableArray，随着循环的进行元素的个数会增加。

代码清单 10-6 main.m

```
#import <Foundation/Foundation.h>
#import "NSArray+Random.h"

int main(void)    // 使用 ARC
{
    @autoreleasepool {
        id arr1 = [NSArray arrayWithObjects:
            @"A", @"B", @"C", @"D", @"E", @"F", @"G", @"H", nil];
        id arr2 = [NSMutableArray arrayWithObjects:
            @"01", @"02", @"03", @"04", nil];
        for (int i = 5; i < 20; i++) {
            @autoreleasepool {
                printf("%s %s\n", [[arr1 anyOne] UTF8String],
                                  [[arr2 anyOne] UTF8String]);
                [arr2 addObject:[NSString stringWithFormat:@"%02d", i]];
            }
        }
    }
    return 0;
}
```

下面是程序执行结果的前几行，可以看出，结果没有出现同一个元素连续出现的情况，NSMutableArray 也同预想的一样正常运行了。

```
D 02
G 04
D 01
H 03
E 07
H 05
A 08
B 07
```

通过综合使用关联引用和范畴，可以大大增强 Objective-C 编程的灵活性。但也要注意不能滥用范畴，滥用范畴会导致程序变得不好理解。因此，在使用范畴之前，要考虑是否有其他方法，比如创建子类等。另外，因为通过范畴扩展添加的实例变量并不是真正的实例变量，所以在对象复制和归档时要特别注意。

第11章

抽象类和类簇

抽象类自身不生成实例，它是为了给所有继承它的子类提供统一的接口而定义的。在本章中，我们将介绍隐藏了类具体实现的类簇这一概念。

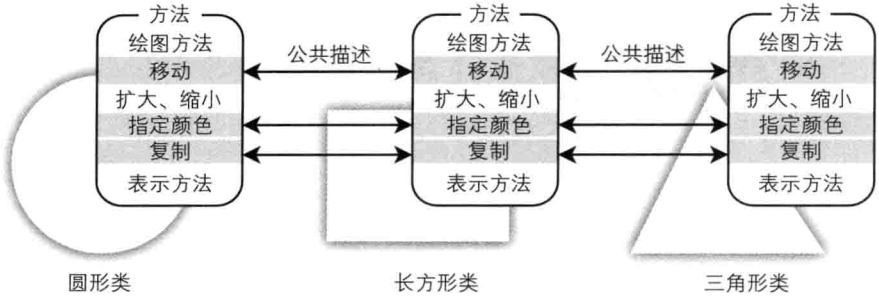
11.1 抽象类

11.1.1 什么是抽象类

我们都知道使用绘图工具就可以在窗口中绘制圆形、长方形、三角形等图形。虽然这些图形与窗口中的表示方法、鼠标绘图的方法有些不同，但操作鼠标时的动作（如移动、扩大、缩小等），颜色的指定、复制操作等都对应了同一个消息。

在这种情况下，每种图形移动相关的方法等都要分别定义，同样的工作我们不得不重复好多次（图 11-1）。

图 11-1 分别定义类的情况

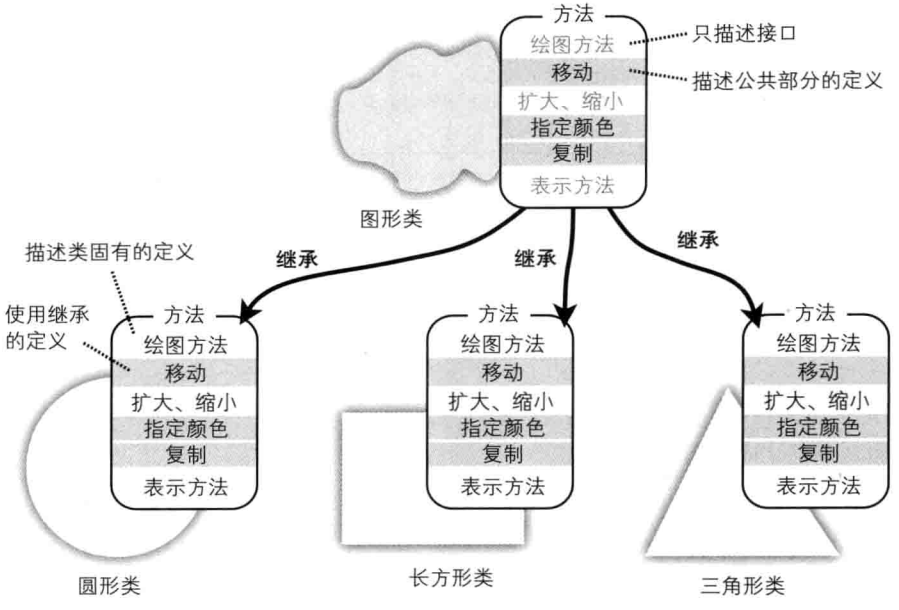


上图中圆为超类，而长方形是它的子类，显然这样的类层次结构让人感觉很不自然，而且也会给以后的使用带来很多问题。

按照一般思维，可能最容易被人接受的方法就是定义一个图形类，并将圆和长方形都定义为它的子类。移动、复制等图形的公共操作在图形类中描述，而具体的绘图方法则在子类中定义（图 11-2）。

对于超类图形来说，尽管每个子类只是实现方法上有所不同，但它们都需要分别声明。这样一来，只要是图形类的子类，就能实现对应超类图形的方法。因此，如果我们事先将对象的类型定义为超类图形，就可以使用静态类型 check 来书写代码。但是，超类图形自身并不能绘制具体的图形，所以不能生成对象实例。

图 11-2 使用抽象类定义的情况



如图，在定义子类时，在子类中只声明那些需要具体定义的方法，这样的类就是**抽象类**（abstract class），或者称为**虚类**（virtual class）

Objective-C 在语法上并没有什么特别的机制来区分抽象类和能够生成实例的普通类的定义。在 Objective-C 中，抽象类这一术语只停留在概念上。而大多数面向对象的语言都使用 virtual 或 abstract 作为关键字来指明抽象类。

在 Objective-C 中，即使是抽象类，只要能够使用`alloc`方法，也可以生成类实例。但这么做没有什么实际意义。例如，类 NSObject 可以生成实例，但几乎没有什么用处，只能为子类提供有用的公共方法。从这点看来，我们可以说类 NSObject 具有很强的抽象类特质。

11.1.2 抽象类的例子

例如，让我们试着在不使用 GUI 的条件下编写一个简单的图形类，据此来告诉大家抽象类到底为何物，以及到底该如何使用。

代码清单 11-1 类 Figure 的接口部分（Figure.h）

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

@class NSString;

@interface Figure : NSObject
@property(assign) NSPoint location;     // 设置图形的位置
```

```
- (void)setSize:(NSSize)newsize;          // 指定图形大小
- (double)area;                           // 计算图形面积
- (NSString *)figureName;                 // 返回表示图形名字的字符串
- (NSString *)stringOfSize;               // 返回表示图形大小的字符串
- (NSString *)description;                // 当前图形的位置、大小
                                          // 返回表示图形面积的字符串
@end
```

类 Figure 并没有规定具体的子类的实例是什么形状，但在结构体 NSPoint 类的属性 location 中保存了图形的位置坐标。类型 NSPoint 包含 CGFloat 类型的成员 x、y，表示画面上点的坐标（请参考附录 A）。

方法setSize:的参数为结构体 NSSize，类 NSSize 定义了 CGFloat 类型的成员 width 和 height，表示长方形的大小。

下面是类 Figure 的实现部分。

代码清单 11-2 类 Figure 的实现部分（Figure.m）

```
#import "Figure.h"
#import <Foundation/NSString.h>

@implementation Figure

@synthesize location;
- (void)setSize:(NSSize)newsize { /* virtual */ }
- (double)area { return 0.0; }  /* virtual */
- (NSString *)figureName { return nil; } /* virtual */
- (NSString *)stringOfSize { return nil; } /* virtual */

- (NSString *)description
{
    NSPoint loc = self.location;
    return [NSString stringWithFormat:
        @"%@: location=(%.2f, %.2f), %@, area=%.2f",
        [self figureName], loc.x, loc.y,
        [self stringOfSize], [self area]];
}

@end
```

首先，属性 location 由 @synthesize 来定义，子类中也使用该定义。需要注意的是，其他方法的定义实质上是没有意义的。

请大家注意最后定义的方法description。该方法会返回包含了图形名字、位置、大小、面积的字符串，在返回构造的字符串时，还会向 self 发送消息来获得相应的值。

方法内的 self 并不能被当作 Figure 类的直接调用接口，而是向某种具体的表示图形的子类的实例发送description消息时的接口。此时，self 表示的是那个图形类的接口，因此，通过使用那个子类中的定义，方法area就会计算出面积，方法stringOfSize也会返回表示大小的字符串。

不仅在抽象类的定义中，在面向对象的语言中，我们也经常使用超类来实现子类的定义。虽然

滥用的话会导致程序难于理解，但如果能熟练使用，对构筑灵活、易扩展的类层次关系还是非常有益的。

下面的代码清单表示的是类 Figure 的子类，即表示圆形的 Circle 和表示长方形的 Rectangle。

代码清单 11-3 类 Circle 的接口部分（Circle.h）

```
#import "Figure.h"

@interface Circle : Figure
{
    double  radius;
}
@end
```

代码清单 11-4 类 Circle 的实现部分（Circle.m）

```
#import "Circle.h"
#import <Foundation/NSString.h>
#import <math.h>

#define PI  3.14159

@implementation Circle

- (void)setSize:(NSSize)newsize
{
    double x = newsize.width;
    double y = newsize.height;
    radius = sqrt(x * x + y * y);
}

- (double)area {
    return radius * radius * PI;
}

- (NSString *)figureName {
    return @"Circle";
}

- (NSString *)stringOfSize {
    return [NSString stringWithFormat: @"radius=%.2f", radius];
}

@end
```

类 Circle 的大小用半径，也就是接口参数 radius 来表示，方法`setSize:`用来设定大小。该方法用长方形对角线长作为参数来确定半径，这和鼠标拖曳时以起点为圆心，将起点到终点的距离作为半径的方法类似。

代码清单 11-5 类 Rectangle 的接口部分（Rectangle.h）

```
#import "Figure.h"

@interface Rectangle : Figure
{
    NSSize size;
}
@end
```

代码清单 11-6 类 Rectangle 的实现部分（Rectangle.m）

```
#import "Rectangle.h"
#import <Foundation/NSString.h>

@implementation Rectangle

- (void)setSize:(NSSize)newsize { size = newsize; }

- (double)area { return size.width * size.height; }

- (NSString *)figureName {
    return (size.width == size.height) ? @"Square" : @"Rectangle";
}

- (NSString *)stringOfSize {
    return [NSString stringWithFormat:
        @"size=%.2f x %.2f", size.width, size.height];
}

@end
```

类 Rectangle 中使用宽和高两个参数来表示图形的大小。通过方法`figureName`查找图形名时，如果宽等于高，则返回 Squre（正方形）。

最后，我们来看一下测试该类的 main 函数。虽然有些罗嗦，但却是难点所在。使用 ARC 进行内存管理。测试流程的本体是函数 testLoop()。这里需要注意的是，变量 fit 的类型为（Figure *）。

代码清单 11-7 测试类 Figure 的 main 函数（main.m）

```
#import <Foundation/NSString.h>
#import "Figure.h"
#import "Circle.h"
#import "Rectangle.h"
#import <stdio.h>

BOOL testloop(void)
{
    Figure *fig = nil;
    double x, y, w, h;
    char buf[64], com;

    do {
```

```
        printf("Shape (C=Circle, R=Rectangle. Q=Quit) ? ");
        if (scanf("%s", buf) == 0 || (com = buf[0]) == 'Q' || com == 'q')
            return NO;
        switch (com) {
        case 'C': case 'c': /* Circle */
            fig = [[Circle alloc] init];
            break;
        case 'R': case 'r': /* Rectangle */
            fig = [[Rectangle alloc] init];
            break;
        }
    }while (fig == nil);

    printf("Location ? ");
    scanf("%lf %lf", &x, &y);
    fig.location = NSMakePoint(x, y);  // 生成 NSPoint 并设定
    printf("Size ? ");
    scanf("%lf %lf", &w, &h);
    [fig setSize: NSMakeSize(w, h)];   // 生成 NSSize 并设定
    printf("%s\n", [[fig description] UTF8String]);
    return YES;
}

int main(void)
{
    BOOL flag;
    do {
        @autoreleasepool {
            flag = testloop();
        }
    }while (flag);  // 循环直到输入 'Q' 为止
    return 0;
}
```

首先，从终端读入字符串，如果字符串的开头是 C 就生成圆形的实例，而如果开头是 R 则生成长方形的实例，并将其赋值给变量 fit。然后，输入图形位置和大小，这时图形的信息就会立刻被打印出来。从终端中读到以 Q 开头的字符串时，程序终止。

编译可按照如下方式进行。

```
clang -Wall -fobjc-arc -o fig Figure.m Circle.m Rectangle.m main.m
-framework Foundation
```

下面是执行的例子。

```
% ./fig
Shape (C=Circle, R=Rectangle. Q=Quit) ? c
Location ? 8.0 10.0
Size ? 4.0 4.0
Circle: location=(8.00, 10.00), radius=5.66, area=100.53
Shape (C=Circle, R=Rectangle. Q=Quit) ? r
Location ? 0 1.0
```

```
Size ? 4.8 5.0
Rectangle: location=(0.00, 1.00), size=4.80 x 5.00, area=24.00
Shape (C=Circle, R=Rectangle. Q=Quit) ? q
```

11.2 类簇

11.2.1 类簇的概念

类簇（class cluster）就是定义相同的接口并提供相同功能的一组类的集合。仅公开接口的抽象类也称为类簇的**公共类**（public class）。各个具体类的接口由公共类的接口抽象化，并被隐藏在簇的内部。这些具体类不能被直接使用，一般会被作为公共类的子类来实现，所以有时也称它们为私有子类。

实际编写代码时，公共类和普通类按照同样的方式使用，但是实际上被生成并存在在内存上的实例是隐藏在类簇中的某个类的实例。因为可以正确执行，所以程序几乎意识不到这点差异。

实现某个类的方法并不是一成不变的。适用于某种情况的最好的方式，也许在其他情况下就意味着高成本。类簇有一个机制，可以从多个已存在的类中挑选出最适合当前场景的类并且自动启用。具体来说就是，根据所使用的是初始化方法、便利构造器（convenience constructor）还是类名开头的临时对象生成方法（例如生成字符串的`stringWithUTF8String:`等），来决定如何实现[①]

字符串类 NSString 就是提供给用户使用的类簇。在程序运行的时候，会产生和公共类 NSString 同样的行为。另外，基于实现上的原因或运行效率方面的考虑，有时也会有更合适的其他类的实例来表示字符串。

例如，在 C 语言字符串和 NSString 字符串对象之间进行转换，这种操作在程序中是很常见的。这里假设有一个私有字符串类，名为 NSSimpleCString，它能够用最接近 C 语言字符串的形式来表示数据。这样一来，通过使用 NSSimpleCString，就可以将 C 语言字符串转换成 NSString 字符串。而当该字符串又必须转换为 C 语言字符串时，这种处理方式也可以保证快速方便地转换。再来关注一下表示文件路径的字符串，考虑到可能需要进行抽取文件扩展名、目录名等各种操作，为了能便利地操作组成路径的各要素和扩展名，有时也需要有一个合适的私有字串类。若能事先准备好几种这样的字串类，虽然它们只存储一种数据并时刻需要做各种变换，但应该能提高执行效率。

① `init...`等初始化方法是用来生成类的实例的，所以由初始化方法决定具体的类看起来很不可思议。但是，将初始化方法的返回值“替换”成不同于接收者的其他实例非常简单，而这也是类簇的实现机制。Foundation 的类的参考文档中也有“初始化方法有时会返回和接收者不同的对象”这样的描述，可见确实如此。但是，编程者不必担心这部分的实现细节。

▶ 表 11-1 Foundation 框架的主要类簇

种类	公共类
数组	NSArray, NSMutableArray
字符集合	NSCharacterSet, NSMutableCharacterSet
数据	NSData, NSMutableData
日期	NSDate
字典	NSDictionary, NSMutableDictionary
扫描器	NSScanner
集合	NSSet, NSMutableSet
字符串	NSString, NSMutableString
属性字符串	NSAttributedString, NSMutableAttributedString
数值型数据	NSValue, NSNumber
通知	NSNotification
管道	NSPipe

表 11-1 中列举了 Foundation 框架提供的一些重要的类簇。除了 NSString 之外，Foundation 框架中还有很多 NSArray、NSDictionary 等其他基本的类。虽然在参考文档中没有有关类簇的描述，但是像类簇（由初始化方法来产生的私有子类的对象）这样的类也是存在的。

11.2.2 测试程序

基于测试目的，我们编写了如下的程序。通过用多种方法产生 NSString 字符串，来查看一下实际运行时究竟使用了哪个类。

这仅仅是一个测试程序，实际程序中并不要求知道类簇中具体使用了哪个类，而且也不应该编写这种实现的程序。

▶ 代码清单 11-8 查看类簇中的类

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

static void printClass(NSString *obj)
{
    printf("Class=%s,\tMember=%s,\tKind=%s\n",
        [NSStringFromClass([obj class]) UTF8String],
        [obj isMemberOfClass:[NSString class]] ? "YES" : "NO",
        [obj isKindOfClass:[NSString class]] ? "YES" : "NO"
    );
}

int main(void)
{
    NSString *ss = @"static string";
    @autoreleasepool {
```

```
        printClass( ss );
        printClass( [ss stringByAppendingString:@"(^-^)"] );
        printClass( [NSString stringWithUTF8String:"(-_-)"] );
        printClass( NSHomeDirectory() );
    }
    return 0;
}
```

函数 printClass() 会打印输出如下信息：参数 NSString 字符串实例所属的类名、是否为 NSString 类的实例、是否为 NSString 子类的实例。在这里，函数 NSStringFromClass() 的作用是返回类名，函数 NSHomeDirectory() 的作用是返回执行程序的用户根目录（详情请参考附录 A）。

该程序在 Mac OS X 10.7 系统环境下执行后会产生下图所示的结果。

```
Class=__NSCFConstantString,   Member=NO,    Kind=YES
Class=__NSCFString,           Member=NO,    Kind=YES
Class=__NSCFString,           Member=NO,    Kind=YES
Class=NSPathStore2,           Member=NO,    Kind=YES
```

Mac OS X 系统版本不同的情况下会产生不同的结果，即使在程序看来是同样的 NSString，但实际上内部可能使用了不同的类。另外，还需要注意，方法`isMemberOfClass:`的执行结果并不是 NSString 的实例。

11.2.3 编程中的注意事项

Objective-C 中没有专门构成类簇的语法。一般情况下，公共类默认为抽象类，而具体的类则是作为公共类的私有子类来实现的。

使用类簇时，不用在意和普通类的差别，但要注意以下两点。

1. 查看实例所属的类时

对类簇来说，所有实例都是私有子类的实例，因此，从类簇的测试程序中我们可以看出，方法`isMemberOfClass:`即使是将公共类作为输入参数，也很难知道结果。

当实例所属类的处理策略被改变时，可以使用方法`isKindOfClass:`判断是否为子类实例，使用方法`respondsToSelector:`判断是否为特定方法，这些方法都十分有效[①]。

2. 生成子类时

很多情况下，公共类作为抽象类被实现的时候，各个方法是在私有子类中具体实现的。因此，即便生成了直接继承公共类的子类，也不能立即产生用户想要的功能。

在类簇中添加新功能时，请参考 11.3 节的介绍。

① Mac OS X 10.6 之前，包含了可变类和不变类的类簇 NSArray 和 NSMutableArray 共享实现了实体的私有类，所以很难分辨实例到底是可变的还是不变的。Mac OS X 10.7 中改善了这种情况，可以用方法 `respondsToSelector:` 来进行区分。

11.3 生成类簇的子类

类簇使多种类别实现抽象化，在公共类的外部只有类簇是可见的。虽然也可以使用类簇本身，但此时使用类簇中类别的子类时有些麻烦。

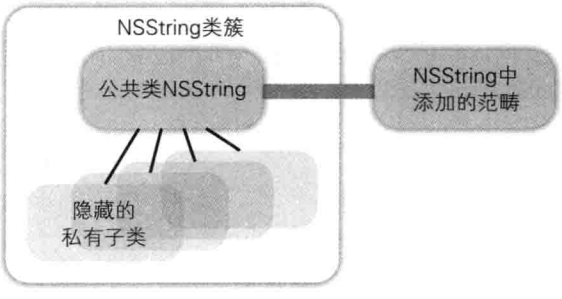
下面说明一下如何产生基于扩展或改变类簇功能的类。因为类簇目前是作为 Foundation 框架的基本类来实现的，所以一般情况下，没有必要生成子类。

11.3.1 使用范畴

虽然范畴不是用来生成子类的，但是在 10.2 节中提到过，添加新的范畴可以扩展公共类的功能，也可以实现像实例变量那样使用关联引用的功能。

图 11-3 是在 NSString 中添加新功能的概念图。因为公共类 NSString 中添加的范畴也会被类簇中隐藏的子类所继承，所以类簇中所有的类都可以使用新添加的功能。

▶ 图 11-3 使用范畴添加新功能



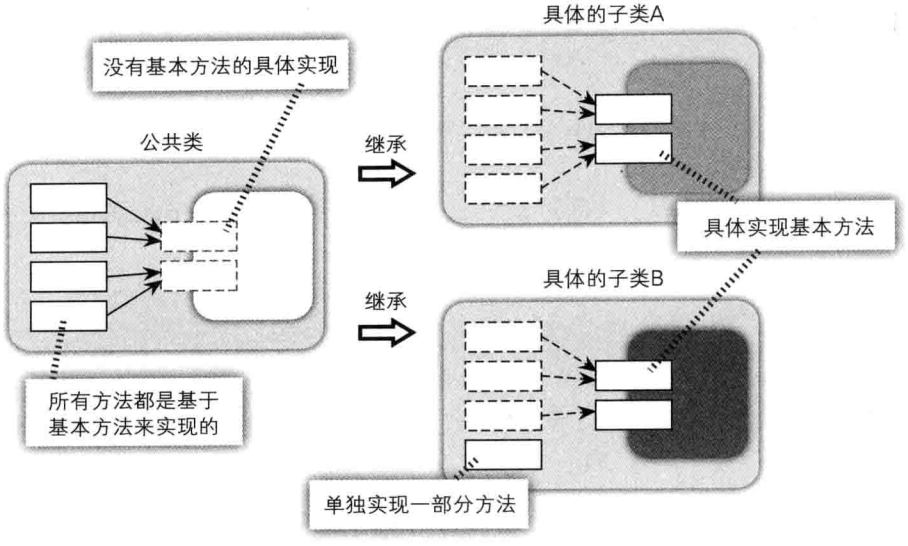
具体的例子请参考 10.2 节中的程序（代码清单 10-1~ 代码清单 10-3）。

11.3.2 基本方法的重定义

我们根据具体的数据结构和算法定义各种各样的类簇。类簇包含一小部分**基本方法**（primitive method），其他方法都是在基本方法的基础上实现的。基本方法在子类中实现，而其方法在公共类中实现。每个子类（私有类）中不同的实现细节都隐藏在基本方法中。也就是说，即使在类簇内部，也实现了过程抽象化和信息隐藏（图 11-4）。

因此，定义私有数据结构及对其访问的基本方法是为类簇生成新的子类的最好方式。

图 11-4 类簇的基本方法



在各个类簇中，哪些包含有基本方法，附录中都有详述。除基本方法之外，对那些希望独立实现的方法也做了相关的解释说明。

表 11-2 归纳了一些主要类簇的公共类的基本方法。例如，NSString 的基本方法是 `length` 和`characterAtIndex:`。NSString 的子类 NSMutableString 的基本方法是`replaceCharactersInRange:withString:`。

表 11-2 主要公共类的基本方法

公共类	基本方法
NSArray	`count` `objectAtIndex:`

NSMutableArray	`addObject:`
	`insertObject:atIndex:`
	`removeLastObject`
	`removeObjectAtIndex:`
	`replaceObjectAtIndex:withObject:`

NSData	`bytes` `length`
NSMutableData	`mutableBytes` `setLength:`
NSDate	`timeIntervalSinceReferenceDate`

NSDictionary	`count`
	`objectForKey:`
	`keyEnumerator`

NSMutableDictionary	`setObject:forKey:` `removeObjectForKey:`
NSString	`length` `characterAtIndex:`
NSMutableString	`replaceCharactersInRange:withString:`

下面具体说明类簇的子类的实现方法。

1. 确定私有数据结构

确定作为实例变量的数据结构，作为超类的类簇不能使用所有的数据结构。

2. 定义初始化方法

定义init…这样的初始化方法。不能继承和使用init之外的超类的初始化方法。只要没有私有数据结构，就可以使用init，所以没必要定义初始化方法。

3. 定义便利构造器

必要的话，以数据类型名作为前缀，定义生成临时对象的类方法。不能继承及使用超类的同样的方法。

4. 定义基本方法

定义自己的基本方法。

5. 定义其他方法

通过定义基本方法，公共类声明的方法可以暂且执行，但是利用生成数据结构的特征也许能够产生更加高效的方法，而且也可以重写这样的方法。如果已经在子类上单独扩展了功能，那么只要定义相应的方法就可以。

11.3.3 生成字符串的子类

下面尝试生成一个简单的示例程序。定义一个 NSString 的子类 BitPattern，它的取值对应整数范围 0~255，由表示 8 比特的 0、1 字符组成。为了获得更好的测试效果，我们只定义基本方法length和characterAtIndex:（代码清单 11-9~ 代码清单 11-11）。

代码清单 11-9 类 BitPattern 的接口部分（BitPattern.h）

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

@interface BitPattern : NSString
{
    unsigned char  value;
}

- (id)initWithChar:(char)val;     /* 指定初始化方法 */
- (NSUInteger)length;
- (unichar)characterAtIndex:(NSUInteger)index;
@end
```

代码清单 11-10 类 BitPattern 的实现部分（BitPattern.m）

```
#import "BitPattern.h"

@implementation BitPattern

- (id)initWithChar:(char)val
{
    if ((self = [super init]) != nil)
        value = val;
    return self;
}

- (NSUInteger)length
{
    return 8;
}

- (unichar)characterAtIndex:(NSUInteger)index
{
    return (value & (0x80 >> index)) ? '1' : '0';
}

@end
```

代码清单 11-11 类 BitPattern 的测试程序（main.m）

```
#import "BitPattern.h"
#import <Foundation/Foundation.h>
#import <stdio.h>

int main(int argc, char *argv[])  // 使用 ARC
{
    NSString *bits, *tmp;
    @autoreleasepool {
        bits = [[BitPattern alloc] initWithChar:argv[1][0]];
        printf("Bit Pattern = %s\n", [bits UTF8String]);
        tmp = [@"Bit Pattern = " stringByAppendingString: bits];
        printf("%s\n", [tmp UTF8String]);
    }
    return 0;
}
```

程序太简单了，你甚至都不敢相信这样的“字符串”真的能起作用。测试程序使用命令行参数的第一个文字作为 BitPattern 的初始化参数。main() 函数中，变量 bits 指定为 NSString* 类型，代入的参数为子类 BitPattern 的实例。在测试中，使用了两个字符串对象打印字符串的内容。程序的执行示例如下。

```
% ./a.out a
Bit Pattern = 01100001
Bit Pattern = 01100001
% ./a.out B
Bit Pattern = 01000010
Bit Pattern = 01000010
```

第12章

协议

协议就是声明方法的集合，它表示对象的行为。协议与具体实现无关，它是根据消息来获取对象的一套系统化的方法。如果能熟练掌握协议的使用，就能够定义出高灵活性、低耦合性的类。

12.1 协议的概念

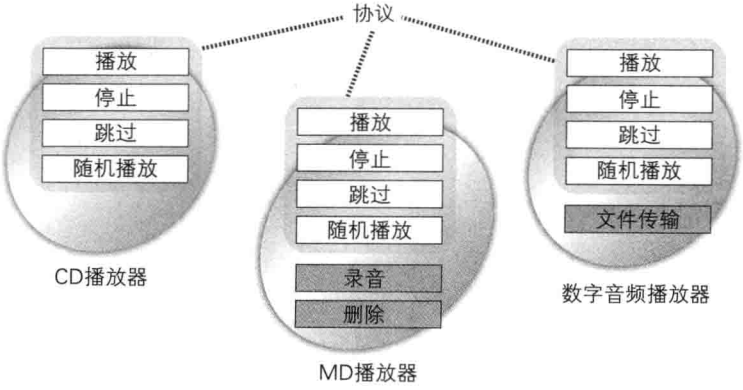
12.1.1 什么是协议

大多数情况下，对象的主要作用是表示所处理的消息的类型，而表示对象的作用和行为的方法的集合体就称为**协议**（protocol）。

协议这个称呼常用于表示互联网的通信协议。Objective-C 中的协议最初就是从各个对象之间的通信协议中抽象出来的一种语言称谓，而现在作为广义的概念来使用了。Java 中的接口这一概念也吸收了 Objective-C 中的协议概念。

这里我们拿现实生活中例子来说明协议的概念。我们都知道 CD 播放器、MD 播放器、数字音频播放器。这些产品都提供了播放、停止、跳过等公共功能。数字音频播放器虽然出现较晚，但由于它采用了同样的操作方法，所以即使是初次接触的用户也很容易上手。也就是说，CD 播放器、MD 播放器等的公共功能也适用于数字音频播放器（图 12-1）。

图 12-1 CD 播放器、MD 播放器、数字音频播放器和协议



播放、停止、跳过等公共功能就是我们所说的协议。只要与这些操作方法相对应，也就是说只要提供了公共协议，无论使用什么样的存储媒体，就都可以播放或停止播放音乐。而对协议来说，每个播放器也可以独立实现各自的功能。

12.1.2 对象的协议

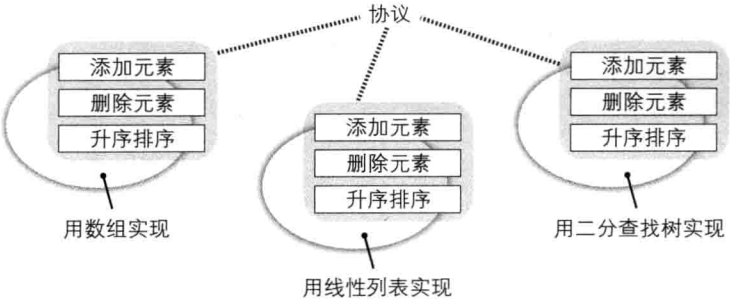
在对象模型化的软件世界里，不同的对象也可能包含相同的方法集合，但通常情况下，这些对象之间并不是继承关系。

将多个对象作为元素保存在队列中，并执行添加、删除、升序排列等操作，这就是我们所说的集合对象。集合对象只要有必要的功能即可，至于其内部是如何实现的（不考虑执行效率），这对使

用者来说则是非透明的。

在图 12-2 中，数组、线性表、二分查找树就是这样的集合对象。这些类之间并没有继承关系，但都有添加、删除、升序排序这样的操作。这些操作就是协议。因此，无论选择集合对象中的哪个类，都可以使用该协议来进行编程。

图 12-2 公共协议的不同实现



在继承中，超类的实现直接影响着子类。具体来说，定义的实例变量和方法都会被子类自动继承下来，但这样有时也有问题。例如，在之前描述的 3 个类的例子中，由于实现方法是完全不同的，因此无论是继承实例变量还是方法都没有意义。

Objective-C 中的协议仅仅是声明方法的集合体，实现方法则由各个类自行完成。因此，使用协议的各个类之间是否有继承关系都无关紧要，重要的是如何实现这些方法。

使用协议的情况下，如果类实现了该协议声明的所有方法，我们就说类**遵循**（conform）该协议。而它的子类实例因为继承关系也拥有了这些协议方法。当类适用于某个协议时，它的实例也适用于这个协议。

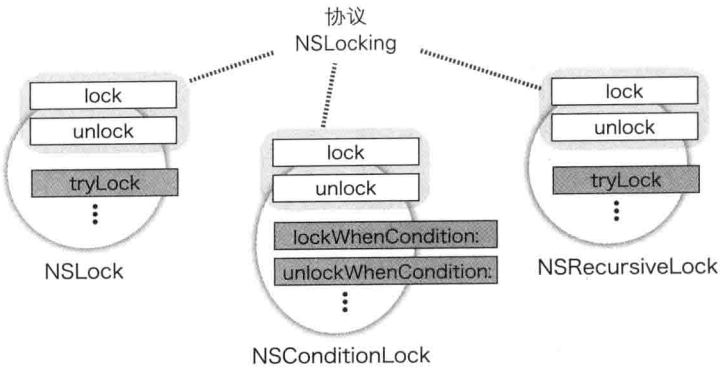
通过目前为止的说明我们已经了解到，当声明使用类时，因为子类实例也是父类的对象，所以子类实例也能执行父类的操作。协议也同样，使用协议名可以声明一种类型，该类型描述了适用于该协议的对象操作。此时，协议适用的类实例也就可以通过该类对象来执行操作了。

在上一章中，我们说明了抽象类的方法。而协议就可以被理解为只有声明而没有实现方法和实例变量等的抽象类。

下面我们来看一些具体的例子。

在 Foundation 框架中，像信号量这样发挥线程间互斥功能的类有 NSLock、NSConditionLock、NSRecursiveLock（详见第 19 章）。这些统称为锁对象，相互之间无继承关系，但它们都适用于协议 NSLocking。该协议仅由`lock`和`unlock`方法构成，类似于信号量的 P 操作和 V 操作（图 12-3）。

图 12-3 协议 NSLocking



在编程时，为了使访问公共变量的程序能够使用任意类型的锁，相对于使用特定的类型名，使用 NSLocking 协议适用的类集合对象来作为类型声明效果会更好。

12.2 Objective-C 中协议的声明

12.2.1 协议的声明

协议采用如下方式声明。

语法 协议的声明

```
@protocol 协议名
声明方法;
    ...
@end
```

协议名使用和 C 语言相同的语法规范。其命名习惯通常与类名相同，即首字母大写其他字母小写。此外，协议名也可以与已有的类同名。

方法的声明中也可以使用属性声明。

例如，上一节所列举的 NSLocking 协议就使用了如下定义。

```
@protocol NSLocking
- (void)lock;
- (void)unlock;
@end
```

协议通常被作为头文件书写，并在类的声明之前导入。

12.2.2 协议的采用

当声明的类中实现了某个协议的方法时，接口部分使用如下记述。和普通接口的书写不同的是，超类名字后要用 <> 将协议名括起来，这点需要注意。

语法 协议的采用

```
@interface 类名 : 超类名 <协议名>
{
    声明接口变量;
      ...
}
声明方法;
  ...
@end
```

这样声明时，协议中的方法就被作为了类声明的一部分。因此，在类的接口声明中，就无须再声明这些方法了。与接口中声明的方法一样，协议中的方法在实现文件中也必须要实现。但在超类中已实现的方法就不用再重新实现了。

像这样，类的接口声明指定了某个协议的情况，我们称为类**采用**（adopt）[①] 了该协议。采用协议的类及其子类也就成为了该协议适用的类，而子类也可以同时使用别的协议。

例如，采用协议 NSLocking 的类 NSLock 的接口如下所示（省略了一部分方法）。

```
@interface NSLock : NSObject <NSLocking> {
@private
    void *_priv;
}
- (BOOL)tryLock;
- (BOOL)lockBeforeDate:(NSDate *)limit;
@end
```

一个类中可以同时采用多个协议。这时在接口部分的 <> 括号内，将多个协议名用逗号分隔即可。例如，类 A 采用了协议 S 和协议 T，按照如下方式书写。

```
@interface A : NSObject <S, T>
  ...
@end
```

现在，即使多个协议中重复包含同一个方法的声明也没有任何问题。例如，协议 S 和协议 T 中都包含了方法copy:。只要在实现文件中实现了copy:方法，也就是实现了协议 S 中的copy:方法和协议 T 中的copy:方法。

但是，选择器相同而函数参数和返回值的类型不一样，即签名（见 4.3 节）不同的方法在协议中重复声明时就会产生问题。一个类内不能声明包含同一个选择器的另一个方法，也不能定义多个这

① adopt 一般解释为采用或采纳。此外也包含借用外来语、采取某种姿态或态度、特殊的说话方式或姿势等意思。因此，用它来表示协议的作用是再好不过的。

样的协议。

而当协议中的方法在某个范畴中实现时，就可以在该范畴中声明采用协议，指定方法如下。

语法 范畴中协议的采用

```
@interface 类名(范畴名)<协议名>
声明方法;
   ...
@end
```

通过此方法还可以在已知类中添加协议的方法实现。

12.2.3 协议的继承

在某个协议中，可以追加另一组方法来产生新的协议，这称为协议的继承。声明方法如下所示。

语法 协议的继承

```
@protocol 协议名1 <协议名2>
声明方法;
   ...
@end
```

这样声明的协议包含了继承的协议中的一组方法，以及新增的一组方法。而且协议还可以有多个继承源。增加多个继承源时在 <> 内将多个协议名用逗号分割即可。

12.2.4 指定协议的类型声明

我们可以声明某个对象适用于某个协议。例如，当我们想要声明变量 obj 适用于协议 S 时，就可以采用如下方式定义。

```
id <S> obj;
```

对象也可以是临时参数，下面的例子就表明临时参数 elem 适用于协议 mag。

```
- (void)addElement:(id <mag>) elem;
```

在声明指定协议的类型时，编译器会对类型进行静态检查。但需要注意的是，在运行时并不会对类型进行动态检查。

类是否适用于协议，与每个方法是否得到了实现无关，而是根据在接口文件中是否声明了采用协议来判断。也就是说，在类的实现部分，即便实现了某个协议的所有方法，只要在接口文件中没有声明采用协议，就不能认为这个类是协议适用的。

不仅是 id 类型，具体的类名和范畴的组合也可以被当成类型来使用，如下例所示。

```
- (void)setAlternativeView:(NSView <Clickable> *)aView;
```

在此例中，参数 aView 不仅是类 NSView 的实例，还使用了范畴或继承，同时还是协议 Clickable 适用的对象，此例中静态说明了这些特性。

如果一个对象使用了协议，那么在指定该对象的类时，类对象只要能适用于指定的协议就行，而不用管它是什么类的对象。这样就可以编写出不依赖于具体的类的实现的、高灵活性的代码。

像这样，代码中只关注协议和抽象类，而没有具体类名的对象称为**匿名对象**（anonymous object）。

系统框架提供的协议中，很多都继承了协议 NSObject。协议 NSObject 是基类 NSObject 的一部分接口，规定了对象的基本行为（参考附录 A）。如上所述，像 id < 协议 > 这样的类型定义虽然很常见，但如果协议继承了 NSObject，就能够向编译器告知对象的基本功能。

12.2.5 协议的前置声明

如果只在头文件的类型声明中使用协议名，可以指定前向引用。这与 4.3 节中说明的类的前置声明是一样的。但是，在定义文件中，为了让编译器能够检查类型，就必须引用协议的定义。

例如，声明 Clickable 这个名字为协议，可以采用如下方式。

```
@protocol Clickable;
```

12.2.6 协议适用性检查

在运行时可以动态地检查对象是否适用于某个协议，因此在程序中就有必要把协议当成数据来看待。

使用编译器命令符@protocol ( )后，就可以获得表示指定协议数据的指针。@protocol ( ) 参数中包含类型 (Protocol *)，可以代入变量。

```
+ (BOOL) conformsToProtocol: (Protocol *) aProtocol
```

aProtocol参数指定的协议和类适用时，返回YES。

```
- (BOOL) conformsToProtocol: (Protocol *) aProtocol
```

接收器类和参数aProtocol指定的协议适用时，返回YES。

例如，要检查对象 obj 是否适用于协议 NSLocking，可以用如下方式。

```
if ( [obj conformsToProtocol:@protocol(NSLocking)] ) ...
```

12.2.7 必选功能和可选功能

到目前为止，采用协议的类都必须实现协议中列举的所有方法。而 Objective-C 2.0 中规定了一项新的功能：协议列举的方法中，分为必须实现的方法和可选择实现的方法，也就是说**可以指定不用实现的方法**。

协议中声明的方法群虽然和协议关系紧密，但也存在一些可有可无的方法，这种情况下我们可以将这些方法指定为可选。这样一来，根据协议的记述，实现方法时就能统一接口。

在协议声明中，编译器命令符 @optional 和 @required 可用来设定其后出现的方法是可选的还是必选的。而 @optional 和 @required 命令符在声明中以什么样的顺序出现以及出现多少次都可以。如果声明中没有特殊指定，那么就默认为 @required，表示方法是必选的。

下面我们以闹钟协议为例进行说明。在闹钟协议中，当前时间的设定、闹铃的 ON / OFF 设定以及起床时刻的设定都是必选的；而贪睡功能（停下来几分钟后再响铃的功能）的 ON / OFF 设定、闹铃临时停止的设定则是可选的。下面的示例代码写得有点儿复杂，通常情况下是应该更简明一些的。

```
@protocol Alarm
- (void)setCurrentTime:(NSDate *)date;
@property (assign) BOOL alarm;

@optional
@property (assign) BOOL snooze;
- (void)pauseAlarm:(id)sender;

@required
- (void)setTimerAtHour:(int)h minute:(int)m;
@end
```

由于采用协议的类可以不实现可选方法，因此有时就需要动态地检查方法是否可用。在该例中，闹钟对象不一定具有贪睡功能。而如果有的话，由于方法的接口是明确的，因此检查该方法是否可用也很容易。

12.2.8 使用协议的程序示例

这里，我们首先来看一个协议 RealNumber，它的功能是取得实数值。

代码清单 12-1 协议 RealNumber（RealNumber.h）

```
@protocol RealNumber
- (double)realValue;
@end
```

这是一个很简单的例子。下面，我们在类 NSMutableArray 中添加范畴，保存协议的对象并添加排序操作。

代码清单 12-2 category RealArray 的接口部分（RealArray.h）

```
#import <Foundation/NSArray.h>
#import "RealNumber.h"

@interface NSMutableArray (RealArray)
- (void)addRealNumber:(id <RealNumber>)number;
- (void)sort;
@end
```

代码清单 12-3 category RealArray 的实现部分（RealArray.m）

```
#import "RealArray.h"

@implementation NSMutableArray (RealArray)

- (void)addRealNumber:(id <RealNumber>)number
{
    [self addObject: number];
}

static NSInterger compareReal(id <RealNumber> a, id <RealNumber> b, void *_)
{
    double v = [a realValue] - [b realValue];
    if (v > 0.0) return NSOrderedDescending;
    if (v < 0.0) return NSOrderedAscending;
    return NSOrderedSame;
}

- (void)sort
{
    [self sortUsingFunction:compareReal context:0];
}

@end
```

这里使用了 9.4 节中提到的`sortUsingFunction:context:`方法。比较各个元素的功能由局部函数 compareReal() 完成，函数返回值为 NSComparisonResult 枚举类型（参考附录 A）。

接下来定义使用协议的类 RealNumber。首先，使用 5.3 节中编写的分数计算器来尝试扩展这个分数类的功能，中间省略的部分和源代码是相同的。

代码清单 12-4 类 Fraction（协议适用版）的接口部分（Fraction.h）

```
#import <Foundation/NSObject.h>
#import "RealNumber.h"

@class NSString;

@interface Fraction : NSObject <RealNumber>
...（省略）...
@end
```

代码清单 12-5 类 Fraction（协议适用版）的实现部分（Fraction.m）

```
#import "Fraction.h"
#import <Foundation/NSString.h>
#import <stdlib.h>
#import <float.h>

@implementation Fraction
...（省略）...

- (double)realValue
{
    if (den == 0) return DBL_MAX;  // 最大实数值
    return (double)(sgn * num) / (double)den;
}
@end
```

字符串 NSString 也适用于协议 RealNumber，这里定义协议适用的范畴并扩展其功能。

代码清单 12-6 类 NSString 的范畴的接口部分（NSStringReal.h）

```
#import <Foundation/NSString.h>
#import "RealNumber.h"

@interface NSString (Real) <RealNumber>
@end
```

代码清单 12-7 类 NSString 的范畴的实现部分（NSString.m）

```
#import "NSStringReal.h"

@implementation NSString (Real)
- (double)realValue
{
    return [self doubleValue];
}
@end
```

同样，如果添加范畴，类 NSNumber 等也能够适用于协议 RealNumber。

最后，在 main() 函数中确认上述操作。

代码清单 12-8 用来测试协议 RealNumber 的（main.m）

```
#import <Foundation/Foundation.h>
#import <stdio.h>
#import "RealNumber.h"
#import "Fraction.h"
#import "NSStringReal.h"
#import "RealArray.h"

int main(void)
{
    @autoreleasepool {
```

```
        id array = [NSMutableArray array];
        [array addRealNumber:@"1.3"];
        [array addRealNumber:@"0.35"];
        [array addRealNumber:@"0.2"];
        [array addRealNumber:
            [Fraction fractionWithNumerator:1 denominator:3]];
        [array addRealNumber:
            [Fraction fractionWithNumerator:3 denominator:8]];
        [array addRealNumber:
            [Fraction fractionWithNumerator:3 denominator:2]];
        [array sort];
        printf("%s\n", [[array description] UTF8String]);
    }
    return 0;
}
```

程序按照如下方式编译。

```
clang -fobjc-arc Fraction.m NSStringReal.m RealArray.m main.m -framework Foundation
```

执行结果如下所示。

```
("0.2", 1/3, "0.35", 3/8, "1.3", 3/2)
```

专栏：类的多重继承 COLUMN

在 Objective-C 中定义类时必须指定一个超类，有的编程语言也可以指定多个超类。继承多个超类的方式称为**多重继承**（multiple inheritance），只继承一个超类的方式称为**单一继承**（single inheritance）。例如，C++ 中就可以多重继承。

应用多重继承时，多个超类的特征可以继承到一个子类中。例如，带有录像功能的电视、有数码照相机功能的手机等。

多重继承虽然使用方便，但是也容易产生问题。例如，生成栈功能、队列（等待队列）功能都可以方便使用的类时，即使继承了栈和队列，也只能分别使用栈和队列的功能，而不能同时拥有。此外，栈和队列都有一个同名的方法`nextObject`用来取出下一个元素，这时，多重继承的子类到底应该执行哪个`nextObject`方法呢？

除此之外，查找子类的方法、重复继承（重复包含相当于类的“祖先”的一个类）等都是非常麻烦的问题。因此也有调查称在实际的编程中多重继承并不经常使用。

Objective-C 没有类的多重继承功能。但通过对协议以及其他编程技术进行组合，也能实现同等功能。比起不经常使用且语法复杂的多重继承，协议的概念既灵活又有效。

实际上，后期产生的语言 Java 也没有加入多重继承功能，而是借用了 Objective-C 的协议的概念，只不过在 Java 中我们称之为接口。

12.3 非正式协议

12.3.1 什么是非正式协议

我们可以将一组方法声明为 NSObject 的范畴，这称为**非正式协议**（informal protocol），或者称为简化协议。为了和非正式协议相区别，有时也将上一章节之前讲述的协议称为正式协议（formal protocol）。虽然都称为协议，但是在功能上却是不同的。

非正式协议只是作为范畴进行声明，而并没有实现。实际上，范畴中声明的方法即使没有实现，也可以编译或执行，但是在发送消息时会出现运行时错误。

在类中，如果要使用一组非正式协议声明的方法，就需要在接口文件中重新声明这组方法（并不是必须的），并在实现文件中实现，但并不要求实现范畴中的所有方法。

由于非正式协议只是基类的范畴，因此，和使用正式协议时一样，既不能在编译时做类型检查，也不能在运行时检查协议是否适用。如果要检查非正式协议中的方法是否已实现，只能对每个方法调用`respondsToSelector:`。

下面我们来总结一下非正式协议的相关概念。

- **非正式协议被声明为 NSObject 类的范畴**
- **非正式协议中声明的方法不一定要实现**
- **编译时，不能检查类对非正式协议的适用性**
- **运行时，不能检查类对非正式协议的适用性。只能确认是否实现了每个方法**

12.3.2 非正式协议的用途

非正式协议有这么多问题，到底要如何使用它呢？在 Cocoa 中，从系统（框架）方调用用户编程方的对象来互发消息时，经常会使用非正式协议。

例如，Application 框架的类 NSColorPanel 中定义了非正式协议（AppKit / NSColorPanel.h）。

```
@interface NSObject (NSColorPanelResponderMethod)
- (void)changeColor:(id)sender;
@end
```

使用调色板改变颜色时，界面上选择的对象（响应者，见 15.4 节）如果实现了该方法，那么无论继承关系如何，都可以在接收到消息后执行改变颜色的处理。而如果没有实现该方法，则不会接收这个消息。

此外，iOS 的 UIKit 框架中定义了非正式协议 UIAccessibility（UIkit / UIAccessibility.h）。

```
@interface NSObject (UIAccessibility) // 一部省略
@property(nonatomic) BOOL isAccessibilityElement;
@property(nonatomic, copy) NSString *accessibilityLabel;
@property(nonatomic, copy) NSString *accessibilityHint;
@property(nonatomic, copy) NSString *accessibilityValue;
@property(nonatomic) UIAccessibilityTraits accessibilityTraits;
@property(nonatomic) CGRect accessibilityFrame;
@property(nonatomic, retain) NSString *accessibilityLanguage;
@end
```

可访问功能是指视觉、听觉有缺陷的用户也可以方便地使用设备的功能，例如功能 VoiceOver 就可以读出所触摸的按键等的种类。界面上设置的 UIKit 组件已经采用了上述非正式协议，而其他界面元素，只要实现了上述特性，也都能使用可访问功能。

由于非正式协议的方法是作为基类的方法定义的，因此，即使在程序中写明向任何一个实例发送消息，在编译时也不会出现警告。

例如 box 对象，即使我们不知道它是否实现了上述非正式协议 NSColorPanelResponderMethod 的方法，也可以按照如下方式使用方法`respondsToSelector:`来编程。值得注意的是，即使声明了 box 的类型，编译时 if 代码中也不会出现如下警告。

```
if ([box respondsToSelector: @selector(changeColor:)])
    [box changeColor:self];
```

非正式协议与实现范畴中的方法时的情况有所不同，它在 NSObject 中并没有添加什么实质内容。源文件中导入的范畴声明，实际上是方法的原型（prototyple）声明。

像上例这样，在并不需要整个方法群的情况下，一般都会使用非正式协议来声明。现在，Objective-C 2.0 中已经可以使用带可选标记的协议。而各部分代码中使用协议的编程风格也随处可见。

专栏：使用宏（macro）来区分系统版本的差异 COLUMN

如果留心观察框架中提供的类头文件，就会发现很多下面这样的预处理行。

```
#if MAC_OS_X_VERSION_MAX_ALLOWED >= MAC_OS_X_VERSION_10_2
```

宏 MAC_OS_X_VERSION_MAX_ALLOWED 表示当前使用的开发环境（SDK）的版本。同样，还有宏 MAC_OS_X_VERSION_MIN_REQUIRED，它用来说明使用该 SDK 开发的应用在运行时对系统版本的最低要求。

此外，还有像 MAC_OS_X_VERSION_10_2 这样表示系统版本的宏。通过使用这些宏，可以根据新旧等不同版本来编写合适的代码。但是，在旧版本的系统中不能定义新的宏。例如，在 Mac OS 10.2 环境中就没有定义宏 MAC_OS_X_VERSION_10_4。于是，为了使旧版本也可以进行编译，就必须按照如下方式，使 Mac OS X 10.1 对应 1010，10.2 对应 1020，10.3 对应 1030。

```
#if MAC_OS_X_VERSION_MAX_ALLOWED >= 1040
```

有时，在属性和方法等声明的末尾还会添加如下的宏。

（1）AVAILABLE_MAC_OS_X_VERSION_10_4_AND_LATER
（2）DEPRECATED_IN_MAC_OS_X_VERSION_10_4_AND_LATER
（3）NS_AVAILABLE (10_7, 5_0)
（4）NS_DEPRECATED(10_0, 10_6, 2_0, 4_0)
（5）NS_CLASS_AVAILABLE(10_7, 5_0)

（1）是 10.4 及其后版本使用的新功能。（2）与（1）相反，表示不推荐使用的旧功能。（3）与（1）相同，表示在 Mac OS X 10.7 之后的版本或者 iOS 5.0 之后的版本中可用。（4）更复杂，表示在 Mac OS X 10.0 及 iOS 2.0 之后的版本中可以使用，而分别在 10.6、4.0 中不推荐使用。这些与上述的宏 MAC_OS_X_VERSION_MAX_ALLOWED 及宏 MAC_OS_X_VERSION_MIN_REQUIRED 的定义相对应，可以用来替换指定函数属性的注释（annotation）。而如果使用了非法函数或非推荐的方法，在源代码的使用位置处就会产生编译错误或警告。

（5）表示类是否有效，对于添加给该宏的类，程序内采用如下方式判断其是否有效。

```
if ( [NewClass class] ) { /* 只有 NewClass 可以使用才能执行 */ }
```

关于这些宏，请参考 /usr/include/AvailabilityMacros.h 或 SDK Compatibility Guide 等文档。

第13章

对象的复制及存储

本章首先介绍复制对象的协议和方法。接着讲解能够将对象变换为可以保存在文件中或者发送到其他进程的格式的归档。最后将讲解 Cocoa 环境中的一种重要的数据类型——属性表 (property list)。

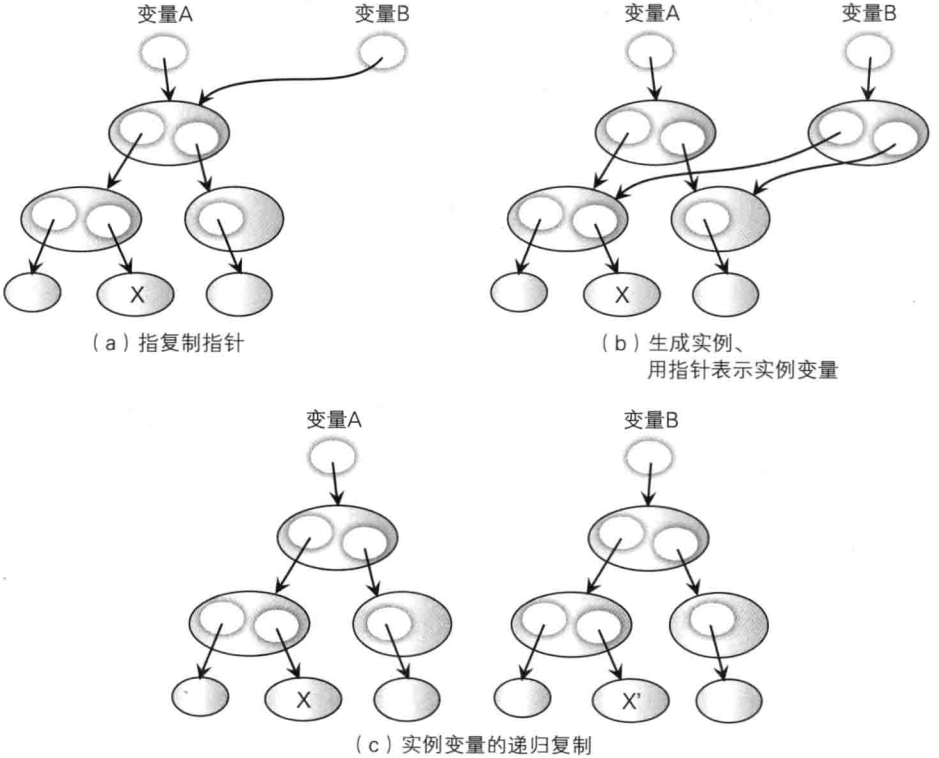
13.1 对象的复制

13.1.1 浅复制和深复制

用与一个实例对象相同的内容，生成一个新对象，这个过程一般称为复制。其中，只复制对象的指针称为**浅复制**（shallow copy），而复制具有新的内存空间的对象则称为**深复制**（deep copy）。浅复制和深复制有时也称为浅拷贝和深拷贝。

从图 13-1 可以看出，现在变量 A 指向了一个对象，而这个对象的实例变量又指向了另一个对象。把变量 A 复制到变量 B 时，首先，我们可以考虑只复制对象的指针这种方法（图 13-1（a））。其次，也可以将变量 A 指向的对象原样复制一份，并通过指针来共享这个对象和复制体中的实例变量（b）。最后，复制对象中的实例变量，还可以递归地进行对象复制（c）。

图 13-1 对象的指向



（a）是将对象代入到变量中。（b）是典型的浅复制，（c）是深复制。

用指针共享某一对象的时候，同时也会共享那个对象的操作结果。生成副本时，对源文件和副本的操作都是独立的。这一点通过（a）和（c）这两种情况下对 X 及 X' 这些对象进行变更操作就会

很清楚。如果想共享变更操作的结果，就应该选择（a）代入或（b）浅复制，而如果希望各自独立管理的话，那就用（c）深复制。

像这样，共享指针的方法和对象的复制方法差异确实很大。因此，在实际编程中，就需要我们根据目标需求来区分使用。例如，像只复制一部分实例变量，其他则通过指针来访问这样综合使用浅复制和深复制的方式有时也是比较合适的方法。

13.1.2 区域

Cocoa 中一直把动态分配的内存管理称为**区域**（zone）。新运行时系统（8.4 节）中不使用区域，而之后提到的`copyWithZone:`方法中的参数也只是格式类似。这里，我们先简单地介绍一下区域的概念。

动态分配内存的堆区域使用了地址空间中很大一片区域。一方面，从空间的局部性原理[①]和虚拟内存管理的角度来看，倾向于同时使用的有关联的数据群，如果能被有效配置在内存中距离较近的位置，就可以实现高效的内存访问。

于是，我们可以在堆内设定多个区块，使关联的数据和对象可从特定的内存区块中分配并保存。这个区块就叫作区域。

因为区域是为此目的服务的，所以它在运行效率上并没有多大贡献，现在较新的运行时环境中都没有用到它。但实例的生成和复制的机制中还保留着区域功能。

通常，生成实例对象要使用类方法`alloc`，而 NSObject 中也有一个类方法可以指定区域来生成实例。

```
+ (id) allocWithZone: (NSZone *) zone;
```

NSZone 结构体是专门用于表达区域的数据结构。现代运行时中，因为参数 zone 会被忽略，所以一般设置为 NULL。但方法的功能和`alloc`一样。

13.1.3 复制方法的定义

NSObject 中有`copy`方法[②]，它能够通过复制接收器来生成新实例。但是，实际的复制操作并不是由`copy`来完成的，而是由实例方法`copyWithZone:`完成的。发送`copy`消息给实例对象后，指定参数为 NULL，这样就可以调用自身的`copyWithZone:`。该方法就是这样生成新的实例的。

鉴于这样的复制方式，为了使实例能够复制，光实现`copy`方法还不行，还需要定义方法`copyWithZone:`。方法`copyWithZone:`返回复制生成的新对象，如果执行失败则返回 nil 值。`copy`的返回值也是同样的。

---

① 程序代码和数据的访问，都趋于集中在内存上距离靠近的位置，这就是局部性原理，它构成了虚拟内存和缓存（cache）的基础。

② 请注意，与 Application 框架中使用的 copy: 方法不同。

由于方法copyWithZone:是在协议 NSCopying 中声明的，因此就要在类中实现采用该协议的方法。如果该方法适用于协议 NSCopying，那么在方法copy的声明属性指定为可选等情况下，编译器就会被告知可以进行复制。

```
@protocol NSCopying
- (id)copyWithZone:(NSZone *)zone;
@end
```

NSCopying 协议在头文件 Foundation / NSObject.h 中定义。但是 NSObject 自己并不采用该协议[①]。方法copy只使用 NSObject 进行简单的定义。

方法copyWithZone:的定义方法可总结如下。

（1）超类如果没有实现方法copyWithZone:，可使用alloc和init...来生成新实例，并将该实例变量谨慎地复制并代入。

（2）超类如果实现了方法copyWithZone:，可直接调用该方法来生成实例，再将子类中添加的实例变量根据需要复制并代入。

（3）实例变量中的共享对象没有必要复制。采用手动引用计数管理时需要retain。

（4）采用引用计数管理时，方法copyWithZone:及copy的调用者就是对象的所有者。因此该返回值不适用于autorelease。

（5）对常数对象的类而言，定义方法copyWithZone:不一定要生成新对象。采用手动引用计数管理方式的情况下，通过 self 适用retain来返回结果。使用 ARC 和垃圾回收器时，只返回 self。

此外，还有另一个函数 NSCopyObject()，它会将实例对象当作二进制序列完整地复制下来并生成另一个对象。但该函数**容易出错，比较危险**，所以并不推荐使用。特别在使用 ARC 的情况下绝对要禁止使用该函数。

13.1.4 复制方法的例子

接下来介绍几个复制方法的例子。

假设为了实现图像的管理而定义了 ImageCell 类。该名字中包含了字符串 image，界面中能显示的图像被保存在 NSImage 实例中，在实例变量中也有该名字。NSImage 是 Application 框架中的类。而且，表示该图像在窗口中的坐标的实例变量的类型为 NSPoint。NSPoint 内的成员 x 和 y，分别表示坐标位置（请参照附录 A）。

① 而类方法是有的，但并不适用于实例的复制。

```
@interface ImageCell : NSObject <NSCopying>
{
    NSMutableString *name;
    NSImage  *image;
    NSPoint  position;
}
...
@end
```

该类所表示的图像，就像扑克的图案一样，其大小是固定的。由于即使被复制，图片也会被共享，所以在复制 ImageCell 类实例时，就没有必要再复制 NSImage 的实例了。只要共享指针就可以了。大家查看一下附录就会发现，NSMutableString 紧挨着协议 NSCopying。这种情况下，可按如下方式定义copyWithZone:方法。

```
- (id)copyWithZone:(NSZone *)zone
{
    ImageCell *tmpcopy = [[[self class] allocWithZone:zone] init];
    /* 重要：使用 [self class] 而非类名 */
    if (tmpcopy) {
        tmpcopy->name = [name copyWithZone: zone];
        tmpcopy->image = [image retain]; /* 使用手动引用计数管理 */
        tmpcopy->position = position;
    }
    return tmpcopy;
}
```

这里即使不使用copyWithZone:的参数 zone 而赋值为 NULL 也没有关系。此外，还可以使用alloc和copy来代替allocWithZone:和copyWithZone:。

使用手动引用计数管理时，image 发送的retain消息必须被保存。虽然这里使用了->来直接赋值，但对属性赋值来说，如果有其他合适的初始化方法或访问器方法，也是应该考虑采用的。

接下来，让我们尝试一下继承 ImageCell 来定义新类。就像象棋中车这个棋子一样，这里在类 DirectedImageCell 的定义中增加了图像的方向属性。接口部分如下所示。enum direction 是列举方向的枚举类型。由于超类 ImageCell 适用于协议 NSCopying，因此 DirectedImageCell 中可以不指定协议。

```
@interface DirectedImageCell : ImageCell
{
    enum directions  direction;
}
...
@end
```

在 DirectedImageCell 中定义新的方法copyWithZone:。

```
- (id)copyWithZone:(NSZone *)zone
{
    DirectedImageCell *tmpcopy = [super copyWithZone: zone];
    if (tmpcopy)
        tmpcopy->direction = direction;
    return tmpcopy;
}
```

这里调用了超类的copyWithZone:方法，但结果返回的实例的类并不是 ImageCell 类，而是 DirectedImageCell 类，这点需要注意。定义超类方法copyWithZone:时不直接使用类名，而是将方法allocWithZone:适用于[self class]。当从子类 DirectedImageCell 中调用这个方法时，因为 self 为 DirectedImageCell 的实例，所以使用[self class]返回的类就是 DirectedImageCell 类。

那么，在定义类 ImageCell 的方法copyWithZone:时，将变量 tmpcopy 声明为类型 ImageCell* 会不会有问题呢？即使实际的对象是 DirectedImageCell 的实例，因为实现中包含的 ImageCell 初始化了，所以也没有问题。此外，数据类型如果表示为 id，->运算符不可以使用。

考虑到这个原因，不仅限于copyWithZone:方法，只要用到继承，我们都**不建议写方法内类自身固定的类名**。

13.1.5 实现可变复制

通过阅读第 9 章我们已经了解到，从常数对象生成可变对象可以使用MutableCopy方法。

```
- (id) mutableCopy
```

然而，该方法是使用 NSObject 简单生成的，实际上真正生成实例的方法是mutableCopyWithZone:。

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

该方法在只包含它的协议 NSMutableCopying 中声明，协议本身定义在头文件 Foundation/NSObject.h 中。NSObject 本身不采用该协议。在使用方法mutableCopy时，如果将 zone 参数指定为 NULL，则实际调用了mutableCopyWithZone:方法。这种结构与方法 copy 和 copyWithZone: 如出一辙。

如果自己需要定义一个包含了常数对象和可变对象的类，那么使用mutableCopyWithZone:方法会更加方便一些。

13.2 归档

13.2.1 对象的归档

有时我们会有这样的需求，即将程序中使用的多个对象及其属性值，以及它们的相互关系保存到文件中，或者发送给另外的进程。为了实现此功能，Foundation 框架中，可以把相互关联的多个对象归档为二进制文件，而且还能将对象的关系从二进制文件中还原出来。像这样，将对象打包成二进制文件就称为**归档**（archive）。

例如，将 Xcode 开发环境中构造的 GUI 对象的关系保存到名为 nib 的文件中，该文件中就包括了各对象的属性值及对象之间的关系。这样就可以在开发环境中再次打开该文件进行编辑，或者在应用运行时使其作为实际的对象运行。

此外，在制作包含复制和粘贴、拖曳和释放等功能的应用程序时，为了保存操作对象的信息，有时也会利用归档。之后讨论的 XML 和属性表，其中的一部分也会常使用包含归档数据的方法。

但是，并不是说归档适用于所有的数据保存。将对象归档保存时，保存方法、还原方法都必须要实现。由于归档与对象具体的结构紧密关联，因此，当类定义或对象间的关系改变的时候，归档的方法也必须随之改变。很多情况下，使用 XML 或属性表等高通用性的格式来保存数据，从程序的效率及稳定性上来说都会更好一些。

本章将简要说明 Foundation 框架的归档功能。关于归档的详细介绍请参照“归档及序列化编程指南”（Archives and Serializations Programming Guide）等官方文档。

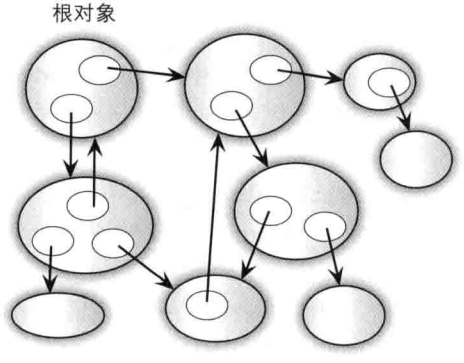
13.2.2 Foundation 框架的归档功能

将对象存储转换为二进制序列的过程称为归档、打包或编码，逆变换则称为**解档**（unarchive）、解码或对象还原。

Foundation 框架中的归档和系统架构是相互独立的。也就是说，PowerPC 和 Intel 都可以利用。在对象包含的实例变量值中，整数及实数这样的基本数据类型，以及指向其他对象的指针等都可以归档。普通指针虽然不能归档，但根据指向的数据类型有时则是可以归档的。

一个对象指向其他对象，这种关系称为对象图（5.5 节）。沿着对象的指向关系往下走，有时就会再次回到原来的对象，这样的情况称为闭环。Foundation 框架可以将对象图归档成书库，其中，作为出发点的对象就

图 13-2 对象图的例子



称为**根对象**。对象图中即使包含闭环也没有关系。

可以使用NSKeyedArchiver和NSKeyedUnarchiver完成对象的归档和解档操作，而它们都是抽象类NSCoder的子类。

所有可以归档的对象都必须要适用于协议 NSCoding。协议 NSCoding 在 Foundation/NSObject.h 中定义，NSObject 自身并不采用该协议。NSString、NSDictionary 等 Foundation 框架的主要类都适用协议 NSCoding。

协议 NSCoding 按照如下方式声明。

```
@protocol NSCoding

- (void)encodeWithCoder:(NSCoder *)aCoder;
- (id)initWithCoder:(NSCoder *)aDecoder;

@end
```

13.2.3 归档方法的定义

协议 NSCoding 中，函数encodeWithCoder:定义了归档自身的方法。下面我们就来看下这个方法的大概过程。参数中会传入 NSCoder（具体为 NSKeyedArchiver）的实例，该实例称为**归档器**。NSKeyedArchiver 的实例在 Foundation/NSKeyedArchiver.h 中声明。

```
- (void)encodeWithCoder:(NSCoder *)coder
{
    [super encodeWithCoder:coder];
    // 超类需要适用协议 NSCoding

    [coder encodeObject:对象 forKey : 关键词字串];
    // 或者使用 encodeConditionalObject : forKey
    ...
    [coder encodeDouble:实数变量 forKey : 关键词字串];
    // 有适合多种类型的方法
    ...
}
```

首先，如果超类适用于协议 NSCoding，那么 super 将调用 encodeWithCoder 对超类的实例变量进行归档。如果超类像 NSObject 一样不适用于协议 NSCoding，则不能调用。

接下来，类自身会对包含的实例变量归档。在类没有自己的实例变量且超类中定义了方法encodeWithCoder:的情况下，该方法就不需要再定义了。

通过使用 NSString 字符串作为键值，可以指定归档解档的内容。当某个类实例归档时，它的实例变量必须指定成不同的键值。在单个对象内部，如果超类使用了一个键值，那么子类中就不能使用该键值。键值只需在同一个类内区分出来即可，不同的类可使用相同的键值。

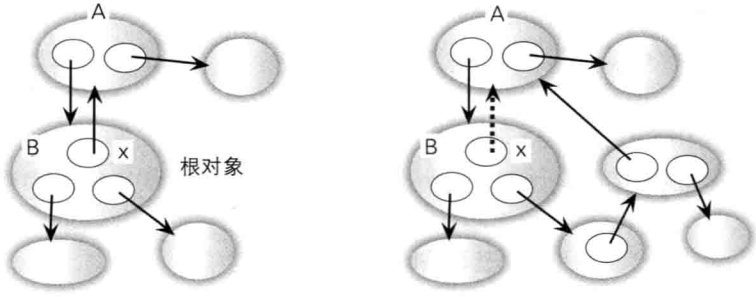
对象的归档使用方法encodeObject:forKey:。因为归档器要对第一个参数的对象调用方

法encodeWithCoder:进行归档，所以是递归调用。当对象图有闭环时，同一个对象会重复要求归档，而实际上已归档的对象是不用重复归档的。

C 的数据类型如整数实数等，都分别有相应的归档方法。数组、结构体等有时需要转换为二进制。

归档某个对象时并没有必要把对象指向的所有对象都归档。但图 13-3 的情况需要注意。

图 13-3 必须使用方法 encodeConditionalObject: 的情况示例



（a）不需要对象A　　（b）必须包含对象A的情况

图中，将对象 B 作为根对象进行归档操作时，去掉了实例变量 x 指向的对象 A，即不在归档中包含对象 A。但是，如图 13-3（b）所示，从对象 B 出发经由别的路径指向 A，结果对象 A 也包含在归档中的情况下，从变量 x 就找不到对象 A 了，这样在归档还原时就可能会出现问题。为解决这个问题，可以在别处使用encodeConditionalObject:forKey:方法，该方法可以确保在不归档的情况下不进行编码。

13.2.4 归档的方法定义

为了从归档中还原对象，需要在各类中定义初始化方法，即协议 NSCoding 的方法 initWithCoder:。下面将详细解释一下如何定义该方法。方法参数被传入 NSCoder（具体为 NSKeyedUnarchiver）实例变量，这个对象也称为解档器。NSKeyedUnarchiver 的接口在 Foundation / NSArchiver.h 中定义。

```
- (id)initWithCoder:(NSCoder *)coder
{
    self = [super initWithCoder:coder];
    // 超类不适用于协议 NSCoding 时
    // 建议使用 [super init]

    变量 = [[coder decodeObjectForKey : 键值 ] retain];
    ...
    变量 = [coder decodeDoubleForKey : 键值 ];
    ...
    return self;
}
```

只要在编码和解码中指定同样的键值，解码就可以按照任意顺序，即使有不能还原的变量也没有关系。

下面我们就将编码和解码方法一并列出。

```
- (void) encodeObject: (id) objv
                forKey: (NSString *) key
```

将字符串作为键值来对参数对象编码。

```
- (void) encodeConditionalObject: (id) objv
                          forKey: (NSString *) key
```

在对象图中需要参数objv时将字符串作为键值编码。

```
- (id) decodeObjectForKey: (NSString *) key
```

使用指定键值来还原编码对象。需要返回值对象时可以使用retain来保存（仅限于手动引用计数管理方式）。

编码和解码 C 数据类型的方法有很多，下面举一个整数编解码方法的例子。

```
- (void) encodeInt: (int) intv
            forKey: (NSString *) key
```

将字符串作为键值对整数参数归档。

```
- (int) decodeIntForKey: (NSString *) key
```

将键值为参数字符串的整数解码。

13.2.5 归档和解档的初始化方法

NSKeydArchiver 的实例，即归档器，可以将对象的编码结果写进数据对象中。NSKeyedArchiver 的初始化方法如下所示。

```
- (id) initForWritingWithMutableData: (NSMutableData *) data
```

将预先生成的NSMutableData实例作为参数，初始化NSkeyedArchiver实例。参数的数据对象被保存起来。这个数据对象最终生成归档。

生成归档器后，接下来就可以使用前述的encode...方法对对象或数据进行归档。编码根对象后，对象图全体就会被递归地进行归档。

所有归档都完成后，最后必须调用finishEncoding方法进行后处理。

```
- (void) finishEncoding
```

处理完成后，由于归档器初始化时传入的数据对象已经进行了归档，因此就可以将其保存到文件中，或者向其他进程发送消息。

解档器 NSKeyedUnarchiver 的初始化方法如下。

```
- (id) initForReadingWithData: (NSData *) data
```

为了将参数的数据对象从归档中还原，需要初始化接收器，参数data被保存在接收器中。

还原归档的对象图和还原一个对象是同样的，都使用方法 decodeObjectForKey：。

上面我们说明了归档和解档的实例生成方法，而其实还有一种更简单的方法也可以将归档结果写入到文件中，并进行读取和还原。

```
+ (BOOL) archiveRootObject: (id) rootObject
                    toFile: (NSString *) path
```

NSKeyedArchiver类中的方法，通过指定根对象将归档结果写入指定路径的文件中。函数处理成功时返回YES。

```
+ (id) unarchiveObjectWithFile: (NSString *) path
```

NSKeyedUnarchiver类中的方法。从指定路径文件中读入归档数据进行解档，返回根对象。方法处理失败时返回nil。

13.3 属性表

13.3.1 属性表概况

属性表（property list）是 Cocoa 环境中用来表示或保存各种信息的标准数据形式。它可以将数组或词典等对象的结构表现为字符串或二进制形式来保存在文件中，或者从中读取出对象的信息。

上一节提到的归档，其主要目的是在程序内部保存和还原对象。因此并不适合用来保存与具体的类关联较弱的信息。在保存及共享和程序的内部实现关系较弱的抽象数据时，建议使用属性表。

属性表有 ASCII 码、XML、二进制三种格式。在 ASCII 码格式的属性表中，字符串、数据、数组、字典（NSString、NSData、NSArray、NSDictionary）四个类的组合结构表现为文本格式。而在 XML 格式的属性表中，除字符串、数据、数组、字典之外，还可以表示包含数值和表示布尔值的 NSnumber，以及表示日期的 NSDate 的结构。而二进制格式则是将这样的结构保存成二进制文件，而不是文本。

字符串或数值等基本数据使用 NSString 和 NSNumber 来表示，表示二进制数据时使用 NSData。NSArray 和 NSDictionary 被用来生成对象结构。NSArray 和 NSDictionary 也可以嵌套使用。此外，这些类也可以是表示各种可变对象的子类。在 NSDictionary 的接口中保存整个属性表时，该字典对象就称为**根词典**。字典的键值必须为字符串。

将属性表保存成文件时，习惯使用文件扩展名“.plit”。可以使用 Xcode 来编辑这样的扩展名文件。

使用属性表来保存各个应用的环境变量值是用户默认的设置。关于这部分内容，请参考 16.4 节。

13.3.2 ASCII 码格式属性表

ASCII 码格式的属性表是从 OPENSTEP 中继承的，它在苹果的开发文档中被记为“Old-Style ASCII Property Lists”。有时也称为文本格式属性表。它的存在主要是为了兼容性，虽然可以从文件中读入，但却不能写出。其缺点是只能使用 ASCII 码字符，而不能使用 NSNumber 或 NSData，因此在实际程序中一般不使用。

但是，作为一种字符串格式，这种属性表可以很容易地取出，看起来也一目了然，同时还可以使用文本编辑器来编辑，因此在 Debug 或生成小的测试程序时也很有效。

通过将方法`description`适用于生成属性表结构的 NSArray 和 NSDictionary 实例，ASCII 码属性表就可以获得字符串格式。

反过来，从 ASCII 码属性表的字符串中还原对象结构时，发送方法`propertyList`到该字符串对象即可。

- (id) **propertyList**（NSString的实例方法）

从ASCII码属性表的字符串中还原并返回对应的对象结构。返回的结构为常量对象。

属性表和类型的对应关系如表 13-1 所示。

表 13-1 类型和属性表的字串对照

类	XML 格式的标签	ASCII 码格式的数据表示
NSString	<string>	“字符串”
NSNumber	<integer>、<real> <true/>、<false/>	无。表示为字符串
NSData	<data>	< 十六进制数 >
NSDate	<date>	无。表示为字符串
NSArray	<array>	(元素，元素，...)
NSDictionary	<dict>	{ 键 = 值 ; 键 = 值 ;...}

属性表内的字符串一般用“""”括起来，只有英文数字时引号可以省略。日语等要通过将 Unicode 编码到 ASCII 码范围内来表示。

二进制数据需要表示为十六进制并用“<>”括起来。

数组用“()”括起来，各元素间用逗号分隔。

字典用“{}”括起来，键通常使用字符串，并与其值用等号连接。各个键值对之间用分号分隔。

下面，我们来尝试使用 ASCII 码属性表来表示 9.5 节中图 9-2（b）教师设备的例子。整体使用一个包含 3 个字典对象的数组。如果含有布尔值就用数值来表示，实际上，数值也是被当作字符串来处理的。可以适当插入空格和换行。

```
(
    {capacity = 180; mic = 1;
        projector = (PC, DVD); room = LR501; screen = 1; },
    {capacity = 150; mic = 1;
        projector = PC; room = LR401; screen = 1; },
    {
        capacity = 45;
        mic = 0;
        note = "Chair with table top ";
        room = LR402;
        screen = 0;
    }
)
```

13.3.3 XML 格式属性表

NSArray 或 NSDictionary 的实例通过使用方法`writeToFile:atomically:`就可生成 XML 格式的属性表文件。反之，从文件中读出属性表并复原对象时，可使用方法 `initWithContentsOfFile:`（请参照第 9 章）。然而，对象构造只能为常量对象。

例如，可以使用 Unicode 表示的日语。

XML 格式的属性表中使用的标签如表 13-1 所示，为了表示字典的键，这里还用到了 <key> 标签。

与之前的例子相同，图 9-2（b）的教室设备一例中，这里用 XML 格式属性表来表示最初的字典对象的内容。可以看出，使用标签可表示布尔值或数值。

```
<dict>
        <key>capacity</key>
        <integer>180</integer>
        <key>mic</key>
        <true/>
        <key>projector</key>
        <array>
                <string>PC</string>
                <string>DVD</string>
        </array>
        <key>room</key>
        <string>LR501</string>
        <key>screen</key>
        <true/>
</dict>
```

13.3.4 属性表的变换和检查

转换或检查属性表格式时需要使用类 NSPropertyListSerialization。

该类实例在 Foundation/NSPropertyList.h 中定义。方法的参数类型为 NSPropertyListFormat，如下所示。

NSPropertyListOpenStepFormat	ASCII 格式
NSPropertyListXMLFormat_v1_0	XML 格式
NSPropertyListBinaryFormat_v1_0	二进制格式

```
+ (NSData *) dataWithPropertyList: (id) plist
                           format: (NSPropertyListFormat) format
                          options: (NSPropertyListWriteOptions) opt
                            error: (NSError **) error
```

参数plist是表示属性表的对象(字典、数组等)，将它转换为format指定的属性表格式(XML或者二进制格式)，并返回数据对象。参数opt指定为0[①]。参数error返回出错时的错误信息(见18.5节)。

```
+ (BOOL) propertyList: (id) plist
    isValidForFormat: (NSPropertyListFormat) format
```

检查参数plist的属性表是否符合format指定的转换格式。

```
+ (id) propertyListWithData: (NSData *) data
                    options: (NSPropertyListReadOptions) opt
                     format: (NSPropertyListFormat *) format
                      error: (NSError **) error
```

从存储着任意一种格式的属性表的数据对象中将对象结构复原出来。参数opt指定为0。
参数format为变量指针，用来返回参数data的属性表格式。如果Format为NULL则不返回结果。
参数error返回出错时的错误信息(见18.5节)。

关于属性表的详细介绍，请见参考文档“Property List Programming Guide”等。

① 官方文档记载着可以指定的常数值，但目前尚不可用于头文件中。

第14章

块对象

本章将介绍 Mac OS X 10.6 及 iOS 4.0 之后新导入的具有 C 语言功能的块对象。块对象将可执行代码及代码可访问的变量合成一个单元使用。

14.1 什么是块对象

14.1.1 C 编译器和 GCD

块对象（block Object）是在 Mac OS X 10.6（Snow Leopard）及 iOS 4.0 平台下可以使用的功能，它不是 Objective-C 而是 C 语言的功能实现。苹果公司的文档中将其称为块对象或 Block（复数为 Blocks），在其他编程语言中，它与**闭包**（closure）功能基本相同。为了和 C 语言的语法相区别，本书中称之为块对象。

如前所述，苹果公司正使用 LLVM 的 clang 作为 C 语言编译器来改进 C/Objective-C/C++ 系的编译处理。块对象也是这些语言功能的成果之一。

从 Mac OS X 10.6 及 iOS 4.0 起，为了使线程在多核上能更有效地运行，苹果公司新开发了**大・中央・调度**（GCD，Grand Central Dispath）这样的构造。GCD 广泛包含了运行时和框架改进等内容，块对象就是其中一个重要的成员。GCD 和多线程的相关内容在第 19 章中有更详细的介绍。

块对象是苹果公司提出的 C 语言的新功能，而不是标准功能[①]。但在 Mac OS X 及 iOS 中，块对象已经被普遍使用了。因此，面向这些平台的软件开发，离不开块对象的相关知识。

下面，我们将首先以 C 语言为前提，来介绍一下块对象的基本功能和使用方法。然后再在 14.3 节中，对 Objective-C 的程序中块对象的使用方法和注意事项进行详细说明。

14.1.2 块对象的定义

块指针或闭包是什么呢？下面我们将首先介绍一下其语法，然后再了解实际的使用方法。

块对象的参数列和主体部分的书写方法与普通函数相同。主体中如果有 return，就可以定义返回值块。

语法	块对象的定义
^(参数列) { 主体 }	

这里，从“^”开始到参数列、主体最后的大括号，这一段记述称为块对象的**块句法**（block literal）。实际上，块句法并不被用于在内存中分配的块对象，它只是编写代码时的一种表达用语。

块对象本身常用于代入到变量后评估，或被作为函数或方法的参数传入等。此时，变量或参数的类型声明和函数指针使用相同的书写方法。只是函数指针声明中使用“*”（参考第 8 章的专栏），而块对象使用“^”。

例如，可采用如下方式声明函数指针 f，实现传入一个 int 类型的参数，且无返回结果。

① 例如 Intel 公司等也发布了具有闭包功能的一系列处理，但语言格式却是不同的。

```
void (*f)(int);
```

同样，我们也能够生成传入一个 int 类型的参数，且无返回结果的块对象。假设将其代入变量 b，那么变量 b 即可采用如下方式声明。

```
void (^b)(int);
```

在变量声明的同时将块对象赋值为变量 b 的代码如下所示。块对象自身使用 printf 打印参数的整数值。

```
void (^b)(int) = ^(int i){ printf("%d\n", i); };
```

像函数的原型声明一样，这里也可以写临时参数名。

```
void (^b)(int i) = ...;
```

变量 b 可以像函数一样被调用执行，如代码清单 14-1 的例子所示。因为是 C 语言的程序，所以文件的后缀名为 .c。虽然编译器也不需要设定特殊的选项，但需要在 Mac OS X 10.6 及之后的版本中执行。程序执行后，会依次输出 5 和 20。

代码清单 14-1　将块对象代入变量的例子（block1.c）

```
#include <stdio.h>

int main(void)
{
    int k = 10;
    void (^b)(int) = ^(int i){ printf("%d\n", i); };

    b(5);
    b(k * 2);
    return 0;
}
```

可见，代码执行的结果，就是定义了像函数一样可以返回值的块对象。

在代码清单 14-2 的 main 函数中，首先，作成一个块对象返回输入参数的 2 倍，并用 printf 打印返回值。然后，将该块对象传递给函数 func。函数 func 会将其解释为参数 block，并将实际参数 10 传入该块对象，然后打印值。

main 函数还直接以写块句法的方式来表示传递给 func 函数的实参。该块对象实现的是累加 1 到参数整数之间的数值，然后返回累加值。由于块句法的主体可以采用和普通函数相同的书写方式，因此也可以使用局部变量。执行程序后，结果就会依次打印出 8、20、55。

代码清单 14-2 返回值的块对象例子（block2.c）

```
#include <stdio.h>

void func(int (^block)(int))  // block 为函数的临时参数
{
    int v = block(10);   // 打印传递的块对象参数。
    printf("%d\n", v);
}

int main(void)
{
    int (^b)(int) = ^(int x){ return x * 2; };   // 参数乘以 2
    printf("%d\n", b(4));
    func(b);              // 向函数传递块对象
    func(^(int a){        // 函数参数直接写为块句法
        int i, k = 0;
        for (i = a; i > 0; i--) k += i;
        return k;
    });
    return 0;
}
```

14.1.3 块对象和类型声明

同函数指针一样，为了简化类型的书写，我们也可以使用 typedef 简化声明。例如，有一个像代码清单 14-2 那样传入的参数为 int 型，且返回值为 int 型的块对象，如果将其类型声明为 myBlockType，可以采用如下方式。

```
typedef int (^myBlockType)(int);
```

使用该类型后，声明函数 func 就可以使用下面的方式。

```
void func(myBlockType block)
```

此外，定义包含 3 个该类型元素的数组 blocks 时，可使用如下方式，如①和②所示。

```
int (^blocks[3])(int);      ❶
myBlockType blocks[3];      ❷
```

如果块对象没有参数，参数列则可以设置为 (void)，此外也可以将参数列连同括号全部省略，或者只保留括号。如下面 3 个例子所示。

```
void (^foo)(void) = ^(void){ /* ... */ };      // (a)
void (^foo)(void) = ^{ /* ... */ };            // (b)
void (^foo)() = ^(){ /* ... */ };              // (c)
```

块句法和函数不同，它不指明返回值类型。编译器虽然会自己推测返回值的类型，但是有些时候为了使代码能够按照预期正确执行，指明 return 返回的类型也未尝不是一件好事。例如，下面的例子表面上看似没有什么问题，但由于 sizeof 操作符返回的类型为 unsigned long，因此，其中 1 个 return 就应该根据映射添加类型变换。

```
func( ^(int n) {
    if (n < sizeof(struct goods))
        return n;
    return sizeof(struct goods);
} );
```

此外，我们还能够通过使用 __BLOCKS__ 宏来检查当前系统环境下是否可使用块对象。根据编译的不同条件，即可区别可以使用块对象时的情况和不可以使用块对象时的情况。

14.1.4 块对象中的变量行为

从代码清单 14-1 和代码清单 14-2 中，我们了解到可以像使用函数指针那样使用块对象。

这里，让我们来关注一下在函数块内声明的自动变量在块句法中使用时的情况。我们说的自动变量其实是函数块内的局部变量，它们通常不用 static 关键字修饰。与此同时，我们也来看一下函数的形参。请见代码清单 14-3。

代码清单 14-3 包含自动变量的块对象例子（block3.c）

```
#include <stdio.h>

void myfunc(int m, void (^b)(void)) {
    printf("%d: ", m);      // 只打印数字
    b();                    // 运行块
}

int glob = 1000;            // 外部变量（全局静态变量）

int main(void)
{
    void (^block)(void);
    static int s = 20;      // 局部静态变量
    int a = 20;             // 自动变量

    block = ^{ printf("%d, %d, %d\n", glob, s, a); };  ❶
    myfunc(1, block);
    s = 0;
    a = 0;
    glob = 5000;
    myfunc(2, block);
    block = ^{ printf("%d, %d, %d\n", glob, s, a); };  ❷
    myfunc(3, block);
    return 0;
}
```

在代码 ❶ 中，变量 block 被赋值为块对象，请注意块对象中包含着外部变量 glob 和 main 函数内定义的局部变量 s 以及 a。之后，程序将块对象作为参数传给 myfunc 函数并调用。然后又在改变变量 glob、s、a 的值后再次调用函数 myfunc。在代码 ❷ 处，变量 block 被赋值为块对象，然后调用 myfunc 函数。执行程序可得到如下结果。

```
1: 1000, 20, 20
2: 5000, 0, 20
3: 5000, 0, 0
```

下面我们来看一下结果，第一行中各个变量的值应该没有任何问题，那么第二行是否有问题呢？可以发现，变量 glob 和 s 的值改变了，而变量 a 的值和第一行相比没有任何变化。第三行显示的是在代码 ❷ 处代入块对象后的变量值，此处变量 a 的值也改变了。如此看来，块对象好像只**在块句法中保存自动变量的值**。

块对象就是把可以执行的代码和代码中可访问的变量“封装”起来，使得之后可以做进一步处理的包。而闭包这个称呼本身就是把变量等执行环境封装起来的意思。我们把闭包引用、读取外部变量称为**捕获**（capture）。

下面我们来总结一下之前的介绍。

1. 块句法主体中，除块句法内部的局部变量和形参外，还包括块句法当前位置处可以访问的变量。这些变量中有外部变量，还有包含块句法的代码块内可以访问的局部变量。
2. 从块对象内部可以直接访问外部变量及静态变量（static 变量），也可以直接改变变量的值。
3. 在包含块句法的代码块内可访问的局部变量中，书写块句法时自动变量（栈内变量）的值会被保存起来，然后再被访问。
   （a）所以，即使变量最初的值发生了变化，块对象在使用时也不会知道。
   （b）变量的值可以被读取但是不能被改变。
   （c）自动变量为数组的时候，会发生编译错误。

下面举例说明一下为什么不能改变自动变量的值。假设代码清单 14-3 的 main 函数内有如下块句法。

```
block = ^{
    glob += 10;     // 没问题
    s += 10;        // 没问题
    a += 10;        // 产生编译错误
};
```

也就是说，因为改变了自动变量 a 的值所以产生了编译错误，错误信息如下所示。这里，不能赋值的变量不是 main 函数内的自动变量 a，而是块内捕获的变量（稍后介绍 __block）。

```
block3.c:22:11: error: variable is not assignable (missing __block type specifier)
```

看起来好像有点儿复杂，简言之就是，在块对象中虽然可以使用可访问的变量，**但自动变量的**

话只能读取复制值。换言之，自动变量在运行时就相当于 const 修饰的变量。

14.1.5 排序函数和块对象

这里我们来看一个 Mac OS X 10.6 及 iOS 4.0 的 C 库（BSD API）中所引入的使用块对象的函数，并体会一下使用块对象的好处。

C 语言的标准库中有一个 qsort 函数可以按照升序（从小到大的方向）排列数组中的数据。因为排序算法使用的是快排，所以便使用了 qsort 这个名字。头文件 stdlib.h 中有一个函数采用了如下方式声明。

```
void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *));
```

数组的头指针由参数 base 指定，数组的元素数、每个元素的大小分别用 nel 和 width 表示。第四个参数是函数指针。该函数的指针参数指向两个元素，并在其之间进行比较。第一个元素大时（大元素排序在后）返回正整数，小时返回负整数，两个相等时返回 0。

例如，下面这个结构体表示了商品的价格、库存和商品名称。

```
struct goods {
    int price;
    int stock;
    char *item;
};
```

该结构体的数组 table 中包含 PRODUCTS 个数据，这里我们调用 qsort 函数将这些数据按照价格从小到大的顺序来排序，如下所示。

```
qsort(table, PRODUCTS, sizeof(struct goods), compf);
```

函数 compf 的定义如下。

```
static int compf(const void *p1, const void *p2)
{
    const struct goods *e1, *e2;
    e1 = p1;
    e2 = p2;
    return ( e1->price - e2->price );
}
```

下面，介绍一个使用块对象而非函数指针来进行排序的函数 qsort_b。原型声明在头文件 stdlib.h 中。这时，函数最后一个参数就不再是函数指针，而是换成了块对象。

```
void qsort_b(void *base, size_t nel, size_t width,
             int (^compar)(const void *, const void *));
```

使用 qsort_b 函数也能像上面那样实现排序数组 table 的数据的功能，方法如下。毫无疑问，块对象也可以代入变量。

```
qsort_b(table, PRODUCTS, sizeof(struct goods),
    ^(const void *p1, const void *p2) {
        const struct goods *e1, *e2;
        e1 = p1;
        e2 = p2;
        return ( e1->price - e2->price );
    });
```

这样，我们就知道了使用块对象可以实现和函数指针相同的功能。既然如此，我们为什么还要专门使用块对象呢？

大家不妨考虑一下，在上例中，如果不是按价格而是按商品库存数量来排序的话该怎么做呢？或不进行升序而进行降序（大的在前小的在后）排序的话呢？

代码清单 14-4 示范了块对象的使用方法。因为块对象可以捕获变量 sort_stock 和 descending 的值，所以可根据设定值按库存数量进行排序，或者进行降序排序。此外，当想用其他排序方法时，只要修改一下包含 qsort_b 函数调用的代码块附近的部分就可以完成。

代码清单 14-4 使用代码块来灵活排序的例子

```
int sort_stock = /*  0：价格，1：按库存数量排序  */ ;
int descending = /*  0：升序，1：降序排序  */ ;
...;
qsort_b(table, PRODUCTS, sizeof(struct goods),
    ^(const void *p1, const void *p2) {
        const struct goods *e1, *e2;
        int d;
        e1 = p1;
        e2 = p2;
        if (sort_stock)
            d = e1->stock - e2->stock;
        else
            d = e1->price - e2->price;
        if (descending) d = -d;
        return d;
    });
```

向排序函数传递的函数指针或块对象会告诉函数如何进行排序，而与之最相关的信息就是排序函数调用前后的代码。不仅限于排序算法，其他 API 也是如此。通过在想要调用函数的地方使用块对象，就可以利用调用处的上下文信息。

使用函数指针时，需要写不同的函数来对应各种不同的功能，此外，为了给函数传递需要的附加信息，往往还要使用多余的参数和外部变量，而这些都违背了编写代码要尽可能独立易懂的原则。例如，Mac OS X 的 C 库函数中很早就有了函数 qsort_r。该函数与 qsort 函数的不同之处是增加了参数 thunk，另外，函数指针的第一个参数 void * 也是新增的。

```
void qsort_r(void *base, size_t nel, size_t width, void *thunk,
             int (*compar)(void *, const void *, const void *));
```

参数 thunk 中可以传递任意指针，而该指针会被传递给函数 compar 的第一个参数。因此，在将某些信息以一个结构体的形式传递时，就可以控制排序行为。但是，这就要求我们在信息传递方法上多费些心思，结构体和函数必须分开定义。否则扩展性和易读性就会变差。

不仅是 C 库的函数，Objective-C 中也有像这样通过传递指针来传递附加信息的方法。但是，通过灵活使用块对象，我们就可以将这样的函数或方法实现为易读、灵活的书写方式。

14.2 块对象的构成

14.2.1 块对象的实例和生命周期

在之前的例子中，块对象都是在函数等代码块的内部，而其实块对象也可以写在函数的外部。例如代码清单 14-5 中的 ❶ 处，代入到变量 g 中的块对象就是一个很好的示例。

这段代码虽然编译时没有任何问题，但执行后就可能会在注释 ❺ 处显示异常值，或者发生运行时错误。而如果不明白块对象的实质到底是什么，我们就不能理解出现这一问题的原因。

代码清单 14-5 块对象和栈之间的关系示例（block5.c）

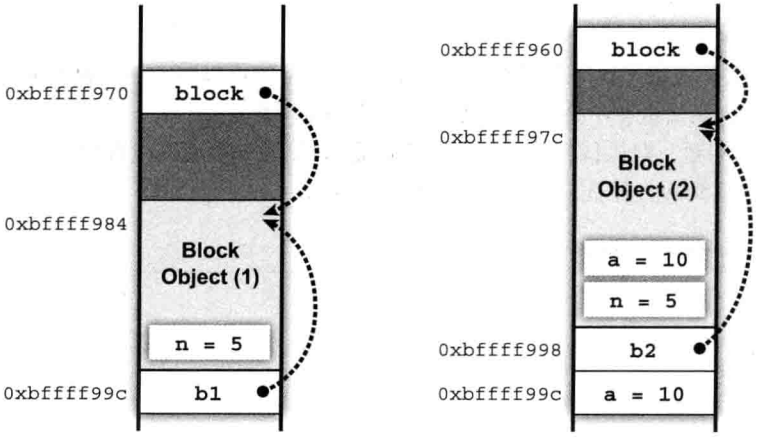
```
#include <stdio.h>

void pr(int (^block)(void)) {   // 参数为块对象
    printf("%d\n", block());    // 执行块对象后打印结果
}

int (^g)(void) = ^{ return 100; };                          ❶

void func1(int n) {
    int (^b1)(void) = ^{ return n; };                       ❷
    pr(b1);
    g = b1;                                                 ❸
}

void func2(int n) {
    int a = 10;
    int (^b2)(void) = ^{ return n * a; };                   ❹
    pr(b2);
}

int main(void)
{
    pr(g);
    func1(5);
    func2(5);
    pr(g);                                                  ❺
    return 0;
}
```

编译块句法时，会生成存放必要信息的内存地址（实体为结构体）和函数。变量中代入的以及向函数传入的实参，实际上就是这片内存区域的指针。如代码清单 14-5 中的 ❶ 所示，在函数外部的块句法被编译后，块对象的内存区域就同外部变量一样被配置在了静态数据区中。

另一方面，❷ 和 ❹ 处函数中的块句法有着不同的行为。执行包含块句法的函数时，和自动变量相同，块对象的内存区域会在栈上得到分配。因此，这些块对象的生命周期也和自动变量相同，只在函数执行期间存在。

图 14-1 表示的是在执行代码清单 14-5 的代码时栈的情况（地址因编译方法或系统环境而异）。（a）是函数 func1 调用函数 pr 的情况。（b）是函数 func2 调用函数 pr 的情况。可以看出，这些函数内的块对象和自动变量 b1、b2 以及 a 并列位于栈上。变量 b1 和 b2 是指向块对象首地址的指针。位于栈顶部的块 block 是函数 pr 的形式参数，也指向相同的地址。

图 14-1 栈内的块对象



（a）函数func1正在调用函数pr　　（b）函数func2正在调用函数pr

块对象将要保存的自动变量的信息复制到了内存区域中。图 14-1 的（a）中保存了形参 n 的值，（b）中保存了变量 a 的值，内存区域的大小是不同的。该内存区域中也包含了评估块对象时所执行的函数指针等的信息。

即使反复执行块句法处的代码，也不会每次都为块对象动态分配新的内存区域。但是，被复制到内存区域中的自动变量的值每次都会更新。另一方面，含有块句法的函数在递归调用时，同自动变量相同，块对象就会在栈上保存多个内存区域。

下面我们来总结一下。

1. 块句法写在函数外面时，只在静态数据区分配一片内存区域给块对象。这片区域在程序执行期会一直存在。
2. 块句法写在函数内时，和自动变量一样，块对象的内存区域会在执行包含块对象的函数时被保存在栈上。该区域的生命周期就是在函数运行期间。

此外，在现在的实现中，当函数内的块句法不包含自动变量时，就没必要复制值，所以块对象

的内存区域也会被设置在静态数据区。但因为实现方法可能改变，所以应该尽量避免编写具有这种依赖关系的程序。

14.2.2 应该避免的编码模式

下面，让我们重新看一下代码清单 14-5，想一下发生执行时错误的原因。❸ 处将变量 b1 的块对象赋值了给外部变量 g。但是变量 b1 中保存的是指向栈的指针，也就是图 14-1（a）中 0xbfff984 地址表示的位置。函数 func1 执行完后，该块对象的生命周期也随之结束。如图 14-1（b）所示，由于函数 func2 的调用会使得栈上被写入其他信息，因此 ❺ 处执行时就会发生错误。像这样，栈上生成的块对象在生命周期外是不能被使用的。

如果能明白块对象的构成，那么也就能很容易看懂下面的代码清单 14-6 的复制操作。我们可能会认为，在这个复制操作中，返回值为 i 的各个块对象被保存在数组 blocks 的第 i 个元素中。但是，由于保存块对象实体的内存区域只有一块，因此在 for 循环中数组中保存的指针都是一样的。这样一来，数组的每个元素都保存着返回值为 9 的块对象。

代码清单 14-6　证明块对象只有一个实体的示例（bloc6.c）

```
void func(void) {
    int i;
    int (^blocks[10])(void);
    for (i = 0; i < 10; i++)
        blocks[i] = ^{ return i; }; // 将块对象代入数组的各个元素中
    for (i = 0; i < 10; i++)
        pr(blocks[i]);              // 打印块对象的输出结果
}
```

如果想为数组的各个元素代入不同的块对象，就必须要进行下一节中所说的复制。但是，使用 ARC 时操作是不同的（之后介绍）。

14.2.3 块对象的复制

如前所述，函数内的块对象和自动变量相同，生命周期只在函数执行期间。但是，在函数或方法的参数中代入块对象也是很常见的方式。在这种情况下，使用与函数调用关系或栈状态无关的块对象是非常便利的。

有一个函数可以复制块对象到新的堆区域。通过使用该功能，即使是函数内的块对象也能独立于栈被持续使用。此外，还有一个函数可以释放不需要的块对象。

Block_copy(block)

参数为栈上的块对象时，返回堆上复制的块对象。否则（参数为静态数据区或为堆上的块对象）则不进行复制而直接将参数返回，但会增加参数的块对象的引用计数。

Block_release(block)

减少参数的块对象的引用计数，减到0时释放块对象的内存区域。

在使用这些函数时，源文件中需要添加头文件 Block.h，该文件在 /usr/include 目录中。

如前所述，堆上分配的块对象使用引用计数来管理。即使在使用垃圾回收的情况下，也必须成对调用 Block_copy 和 Block_release 函数[①]。

代码清单 14-5 的例子中，如果将 ❸ 处的代入改成如下方式，在运行程序时就可以不用在意块对象的生命周期。

```
g = Block_copy( b1 );
```

代码清单 14-6 中数组的代入也可以按照如下方式书写。

```
for (i = 0; i < 10; i++)
    blocks[i] = Block_copy( ^{ return i; } );
```

14.2.4 指定特殊变量 __block

块对象可以包含其访问的自动变量的副本，但是只能直接读取值。另外，若要在多个块对象之间共享可以读写的值，就只能利用外部变量或静态变量。

多个块对象间可以共享值，然而，这种共享需要指定函数块内的局部变量。这样的变量用 __block 修饰符来指定。这个修饰符不能和 static 或 auto 或 register 同时指定。

下面就让我们通过一个程序示例来看看修饰符 __block 到底发挥了什么样的作用。代码清单 14-7 中，函数 func 内写了两个块句法，两个都读取、更改 __block 修饰的变量 sh。如果是一般的自动变量，无论哪个块中，变量 sh 都保持 0 值不会更改。所以就要在函数中生成块对象的副本并代入外部变量 g。

代码清单 14-7 __block 修饰符的效果示例 (block7.c)

```
#include <stdio.h>
#include <Block.h>

void (^g)(void) = NULL;     // 块对象使用的外部变量
int c = 0;                  // 每次执行块时变量都加 1

void func(int n) {
    __block int sh = 0;
    void (^b1)(void) = ^{
        sh += 1;
        printf("%d: b1, n=%d, sh=%d\n", ++c, n, sh); };
    void (^b2)(void) = ^{
        sh += 20;
```

① 可参考头文件 Block.h 内的注释。

```
        printf("%d: b2, n=%d, sh=%d\n", ++c, n, sh); };

    b1();                   // [1], [5]
    b2();                   // [2], [6]
    g = Block_copy(b1);
    sh += n * 1000;
    n = 999;
    b2();                   // [3], [7]
}

int main(void)
{
    void (^myblock)(void); // 块对象使用的变量
    func(1);
    myblock = g;
    myblock();              // [4]
    func(2);
    myblock();              // [8]
    Block_release(g);
    Block_release(myblock);
    return 0;
}
```

程序执行结果如下。开头的行编号对应着代码执行位置，这些在代码的注释中都有说明。形式参数 n 的值即使被修改，在执行结果中也显示不出来，而变量 sh 修改后的结果却会被共享。

最有趣的是执行结果的最后一行。这里被执行的块对象是在第一次调用函数时生成的副本。可以看出，执行块对象后，第一次调用中使用的变量 sh 值一直保留了下来。

```
1: b1, n=1, sh=1
2: b2, n=1, sh=21
3: b2, n=1, sh=1041
4: b1, n=1, sh=1042
5: b1, n=2, sh=1
6: b2, n=2, sh=21
7: b2, n=2, sh=2041
8: b1, n=1, sh=1043
```

从这里例子中能看出，通过使用 __block 修饰的变量，因块对象的处理而改变的值就可以被共享并读取。这种关系在同一个块句法执行时生成的块对象之间有效。

修饰符 __block 修饰的变量（简称为 __block 变量）有如下功能。

1. 函数内块句法引用的 __block 变量是块对象可以读取的变量。同一个变量作用域内有多个块对象访问时，他们之间可以共享 __block 变量值。
2. __block 变量不是静态变量，它在块句法每次执行块句法时获取变量的内存区域。也就是说，同一个块（变量作用域）内的块对象及它们间共享的 __block 变量是在执行时动态生成的。
3. 访问 __block 变量的块对象在被复制后，新生成的块对象也能共享 __block 变量的值。
4. 多个块对象访问同一个 __block 变量时，只要有一个块对象存在着，__block 变量就会随之存在。如果访问 __block 变量的块对象都不存在了，__block 对象也会随之消失。

因为可能会涉及实现，所以这里省略了对 __block 变量的行为的说明。但有一点需要注意的是，随着块对象的复制，__block 变量的内存位置有时会发生变化。而且，**大家不要写使用指针来引用__block变量的代码**，因为那样可能得不到预期的结果。

14.3 Objective-C 和块对象

14.3.1 方法定义和块对象

随着块对象的引入，Cocoa API 中添加了使用块对象的各种方法。

首先，我们来看一下块对象作为方法参数传递时参数类型的指定方法。现在，假如有一个块对象，它有两个整数类型的参数并返回 BOOL 类型的结果。将块对象代入变量 block 时，使用了如下方式。

```
BOOL (^block)(int, int) = ^(int index, int length){ ...; };
```

使用该块对象作为参数的方法setBlock:声明如下。外侧的圆括号包围的部分是参数类型，block 是形式参数名。

```
- (void) setBlock: (BOOL (^)(int, int)) block;
```

类型部分中也可以写形式参数名。这样记述虽然略显冗长，但可以使各个参数的作用更容易理解。

```
- (void) setBlock:(BOOL (^)(int index, int length))block;
```

方法返回的块对象在传递时也是同样的。

```
- (BOOL (^)(int, int)) currentBlock;
```

在 Objective-C 中使用块对象时，在块句法内也可以写消息等 Objective-C 的语法元素。

14.3.2 作为 Objective-C 对象的块对象

令人惊奇的是，Objective-C 程序在编译运行时，**块对象会成为 Objective–C 的对象来执行操作。**

块对象可以像继承了 NSObject 的类实例那样运行，一般的接口对象执行的操作虽不完全但也能适用于块对象，也可以代入 id 类型的变量。

此外，我们也可以使用 copy 方法生成实例对象的副本。这和 Block_copy 的使用相同。使用手动计数管理方式时，可以使用retain、release及autorelease方法，同样也可以将块对象保存

在数组等集合中。然而，需要注意的是，目前retainCount方法返回的引用计数结果是不正确的。

14.3.3 ARC 和块对象

使用 ARC 和不使用 ARC 时，块对象的操作是有区别的，这点需要注意。

如之前所述，栈上保存的块对象在包含块句法的函数执行结束后也可以继续使用，但必须复制块对象才行。同样，在 ARC 中需要保存块对象时，**编译器会自动插入 copy 操作。**

具体地说，就是在被代入强访问变量以及被作为 return 的返回值返回的时候。这些情况下，程序不需要显式地执行块对象的副本。

代码清单 14-6 的数组代入的例子中也没有完整地书写copy方法，但是也可以按照预期执行。也就是说，数组 blocks 是强访问变量，在块代入之前就已经生成了 copy 执行代码。

但是，作为方法参数传入的块对象是不会自动执行 copy 的。而且当块对象作为声明属性的值时，属性选项一般会指定 copy。例如，Application 框架的 NSTask 类就包含如下块对象的属性声明。

```
@property (copy) void (^terminationHandler)(NSTask *);
```

然而，程序中即使有copy方法，由于编译器会判断所有者的操作而进行相应的处理，因此这点儿也不用在意。

此外，**请不要使用 Block_copy 和 Block_release。**因为已经定义了 (void *) 型参数指针，所以 ARC 不能推测所有者。

还有一点是 __block 变量的行为不同。不使用 ARC 时，__block 变量只代入值，示情况可能会悬空指针。ARC 中因为有 __strong 修饰符修饰 __block 变量，使其作为强访问变量来使用，因此就不会成为悬空指针。

14.3.4 对象内变量的行为

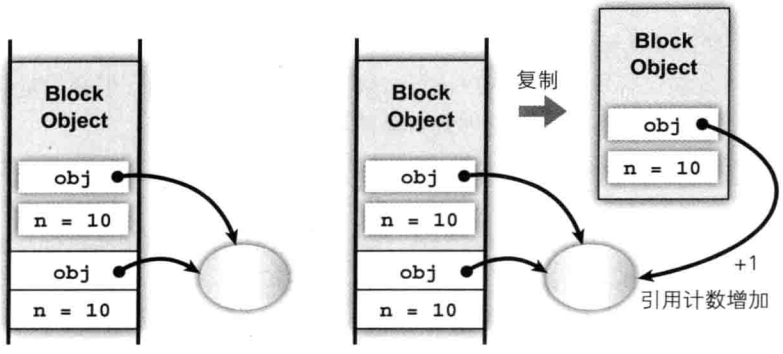
之前已经介绍了块句法中有外部变量或自动变量时这些变量的行为。这里，我们将详细介绍块句法内使用对象时的行为，特别是引用计数的处理。

首先让我们来看下下面这段代码。

```
void (^cp)(void);    // 可以保存块的静态变量

- (void)someMethod {
    id obj = ... ;  // 引用任意实例对象
    int n = 10;
    void (^block)(void) = ^{ [obj calc: n]; };
    ...
    cp = [block copy];
}
```

图 14-2（a）中，块对象在栈上生成，自动变量 obj 和 n 可以在块内使用。变量 obj 引用任意实例对象时，块对象内使用的变量 obj 也会访问同一个对象。这时，实例变量的引用计数不发生改变。

图 14-2 块对象的副本和对象引用



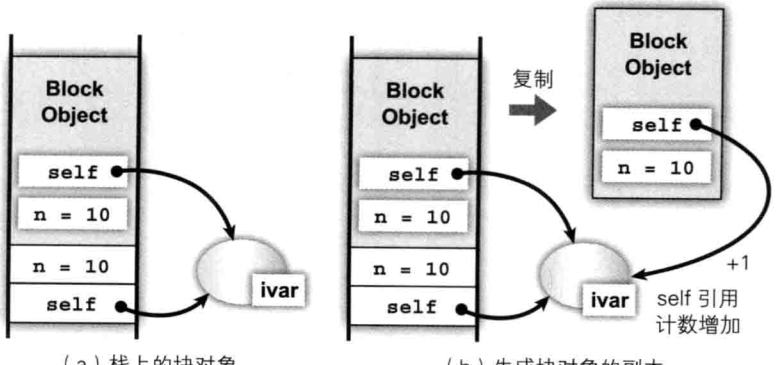
（a）栈上的块对象　　（b）生成块对象的副本

下面，假设块对象自身被复制，并在堆区域中生成了新的块对象（图 14-2（b））。这里，实例对象的引用计数加 1。由于方法执行结束后自动变量 obj 也会消失，因此，引用计数加 1，就使得块对象成为了所有者。实例对象不是被复制而是被共享的，不只从块对象，从哪都可以发送消息。

需要注意的是在某个类方法内的块句法中被书写了同一类的实例变量这种情况。例如下面这个例子，假设 ivar 为包含方法someMethod的类实例变量。

```
void (^cp)(void);

- (void)someMethod {
    int n = 10;
    void (^block)(void) = ^{ [ivar calc: n]; };
    ...
    cp = [block copy];
}
```

图 14-3 块对象的副本和实例变量的引用



（a）栈上的块对象　　（b）生成块对象的副本

这种情况下，当块对象被复制时，self 的引用计数将加 1，而非 ivar。如图 14-3 所示，方法的引用参数（见 8.2 节）self 在堆上分配。在上例中，self 好像没有出现在块句法中，我们可以按下面的方式理解。

```
^{ [self->ivar calc: n]; };
```

块句法内的实例变量为整数或实数时也是一样的，self 的引用计数也会增加。也就是说，当与 self 对等的对象不存在时，所有的实例变量都将不能访问。

块对象和对象的关系可总结如下。

1. 方法定义内的块句法中存在实例变量时，可以直接访问实例变量，也可以改变其值。
2. 方法定义内的块句法中存在实例变量时，如果在栈上生成块对象的副本，retain就会被发送给 self 而非实例变量，引用计数器的值也会加 1。实例变量的类型不一定非得是对象。
3. 块句法内存在非实例变量的对象时，如果在栈上生成某个块对象的副本，包含的对象就会接收到retain，引用计数器的值也会增加。
4. 已经复制后，堆区域中某个块对象即使收到copy方法，结果也只是块对象自身的引用计数器加 1。包含的对象的引用计数器的值不变。
5. 复制的块对象在被释放时，也会向包含的对象发送release。

14.3.5 集合类中添加的方法

Mac OS X 及 iOS 的 API 特别是 Foundation 框架中，增加了很多使用块对象的类和方法。由于篇幅有限，这里举一些集合类中添加的典型方法作为例子进行介绍。

第一个例子是再排序。NSArray 类中有一个通过使用块对象比较元素来排序的方法。这个方法会将排序结果保存到新的数组中返回。NSMutableArray 类中也有排序方法，但它排列的是数组元素自身。

```
- (NSArray *) sortedArrayUsingComparator: (NSComparator) cmptr;
```

NSComparator 是块对象的类型，使用 typedef 定义，如下所示。NSComparisonResult 为比较结果的枚举类型（参考附录 A）。基本上采取与函数 qsort_b 同样的方式代入块对象。

```
typedef NSComparisonResult (^NSComparator)(id obj1, id obj2);
```

下面的方法实现的是从数组中返回满足块对象指定条件的最初的元素索引。如果没有满足条件的元素，则返回 NSNotFound 值。

```
- (NSUInteger) indexOfObjectPassingTest:
    (BOOL (^)(id obj, NSUInteger idx, BOOL *stop)) predicate
```

从数组第一个元素开始，作为方法参数的块对象会被顺序传入元素的对象和索引。传递给块对象的元素如果满足条件就返回 YES，否则就返回 NO。第三个参数 stop 为输出用的指针参数，当返

回 YES 时，可以打断条件检查的递归处理。

例如，该方法的参数中可以传入下面的块对象。该块对象会搜索大于 10 个字符长度的字符串，在遇到符合变量 terminator 内容的字符串时，处理结束。

```
^(id obj, NSUInteger idx, BOOL *stop) {
    if ([obj isEqualToString: terminator])
        *stop = YES;
    return (BOOL)([obj length] > 10);
}
```

下面的方法会从数组的第一个元素开始按照顺序将元素对象和索引赋值给块对象。因此，块对象将会对元素依次进行处理。块对象的参数和上一个方法相同。

```
- (void)enumerateObjectsUsingBlock:
    (void (^)(id obj, NSUInteger idx, BOOL *stop))block
```

最后举一个字典类 NSDictionary 中添加方法的例子。该方法会依次取出字典中存储的入口，并将键值和元素对象传入块对象。使用 BOOL 类型指针来中断处理的方法和之前所示的方法同样。

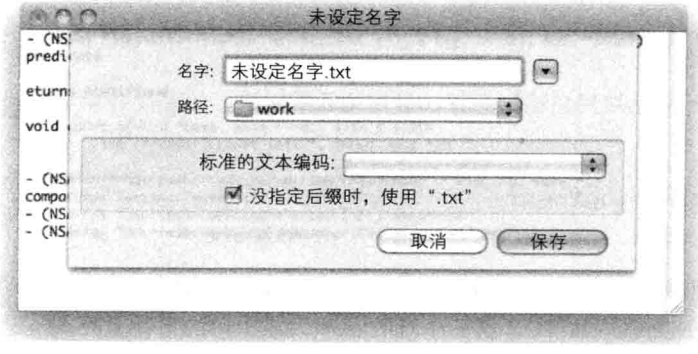
```
- (void)enumerateKeysAndObjectsUsingBlock:
    (void (^)(id key, id obj, BOOL *stop))block
```

14.3.6 在窗体中使用块对象

在 Mac OS X 中打开应用窗口进行操作时，执行“保存”操作后，窗口上会显示出窗体，我们可以指定保存地点、保存文件名以及各种参数（图 14-4）。这里跳出的窗体就是 Application 框架的 NSSavePanel 类的实例。

从窗体出现到点击关闭按钮期间，应用可以执行其他和跳出窗体完全无关的操作。为此，在执行关闭操作来暂时停止显示弹出窗体时，就要执行其他方法进行善后处理。但是，关闭跳出窗体时应该进行的操作和用到的信息，与显示跳出窗体的方法是关系最为紧密的。

图 14-4 保存时显示的跳出窗体



这里，我们会预先在显示窗体时将关闭窗体时执行的善后处理以块对象的形式传入。只要在块句法中书写了善后处理代码，就可以在自动复制、保存或调用时使用处理用到的变量值和对象。

下面是在指定窗口中显示窗体时调用的 NSSavePanel 类的方法。

```
- (void)beginSheetModalForWindow:(NSWindow *)window
      completionHandler:(void (^)(NSInteger result))handler
```

通过调用这个方法来执行保存处理的save:方法可按照如下方式书写。按下窗体的按键时执行的处理也可以书写在同一个位置。

```
- (void)save:(id)sender
{
    ...
    NSSavePanel *svpanel = [NSSavePanel savePanel];
    [svpanel beginSheetModalForWindow:window completionHandler:
            ^(NSInteger result) {
                if (result == NSFileHandlingPanelOKButton) {
                    /* 按下“保存”按键时的处理 */
                }else {
                    /* 按下“取消”按键时的处理 */
                }
            }];
}
```

然而，上述情况下，因为块对象是在栈上生成的，所以 save: 方法执行结束时，块对象也会随之消失。那么，块对象是否应该先复制再传入呢?

调查发现，NSSavePanel 在接收块对象后会复制保存一份。所以，在方法 save: 中就没有必要再复制块对象了。在延迟（或异步）使用所接收的块对象的方法中，其实现与该方法应该是相同的。但遗憾的是，这样的信息在类文档中不一定会写明。

相反，如果自定义了将参数块对象随后执行这样的方法，则需复制块对象，并在不需要时尽快释放。

然而，使用 ARC 书写代码时，就不需要考虑这一点了。代入变量的块对象会自动复制一份并保存。

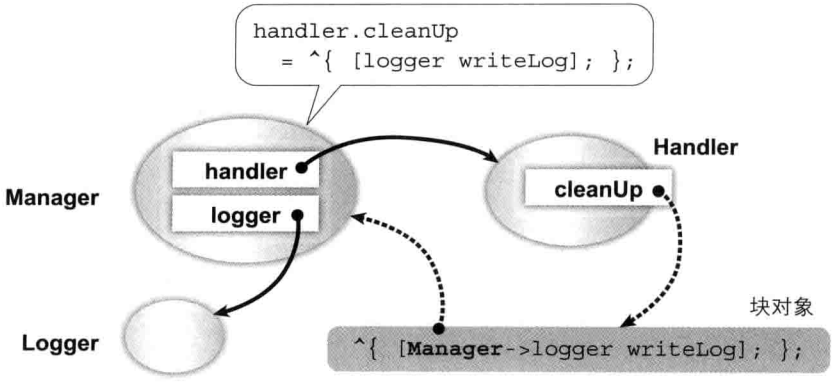
14.3.7 ARC 中使用块对象时的注意事项

使用 ARC 开发软件时需要注意不要写死循环代码。之前也讨论过，使用块对象时，相关对象可能会被自动保存，这时也许就会产生死循环。先看一个简单的例子。

图 14-5 中，对象 Handler 为专门执行某个处理的对象，对象 Logger 是打印系统的操作历史的对象，而对象 Manager 则持有这些对象。假设在对象 Handler 处理结束时块对象已经被赋值给了属性 cleanUP，那么在随后的善后处理中就对其进行调用。而实际的应用也是由执行整体控制的对象，以及分担各个处理作业的对象协调构成的。

现在，假设对象 Manager 将图对象代入到了 Handler 属性中。如之前所述，块句法中有实例变量时，self 而非实例变量的引用计数会自增。这时，块对象内包含了 Manager，形成了 Manager → Handler →块对象→ Manager 的循环包含。这些对象不会被释放，一直保存在内存中。

图 14-5 块对象的循环包含示例



这样的情况很容易在不知不觉中就发生了。clang 编译器只能检测出典型的包含循环，所以总会有漏网的情况。

这种情况下可以用代入临时变量的方法来解决。

在图 14-5 中，对象 Manager 方法内使用了如下方式设定块对象。

```
handler.cleanUp = ^{ [logger writeLog]; };
```

接着，使用弱访问的自动变量 weakLog，如下所示：

```
__weak Logger *weakLog = logger;
handler.cleanUp = ^{ [weakLog writeLog]; };
```

采用这种方式后，从块对象中就不用引用对象 Manager 了。因为不能强访问原先的对象 Logger，所以只要存在该块对象就会释放 Logger 对象。

这个例子比较简单，而一般我们遇到的处理都很复杂。在执行块对象期间，可能就会因副作用而释放弱访问对象。为此，我们可以将 self 自身作为弱访问对象捕获，并在块对象内使用强访问来保存对象。由于和 self 相当的对象在释放时可能会因操作符->而发生执行时错误，所以需要进行条件判断。

```
__weak Manager *weakSelf = self;
handler.cleanUp = ^{
    Manager *strongSelf = weakSelf;  // 确保执行时不释放
    if (strongSelf) {
        [strongSelf->logger writeLog];
    }
};
```

第15章

消息发送模式

本章将介绍应用执行所必须的运行回路。

接着介绍在消息发送时常用到的委托、通知、反应链等设计模式。此外，本章还会详细说明消息转发，以及撤销结构的概况。

15.1 应用和运行回路

15.1.1 运行回路

无论是有 GUI（图形用户界面）的 Mac OS X 应用，还是 iphone、ipad 的应用，都是通过用户选择菜单、点击（或拖曳）界面或操作键盘等而作出反应的。这些程序和命令行程序相比有很大的不同。

这些应用从操作系统中接收鼠标点击等事件的消息，并将其转到相应的例行程序来处理，如此反复，这样的过程被称为**运行回路**（run loop）或**事件循环**（event loop）。

图 15-1 应用的消息处理流程

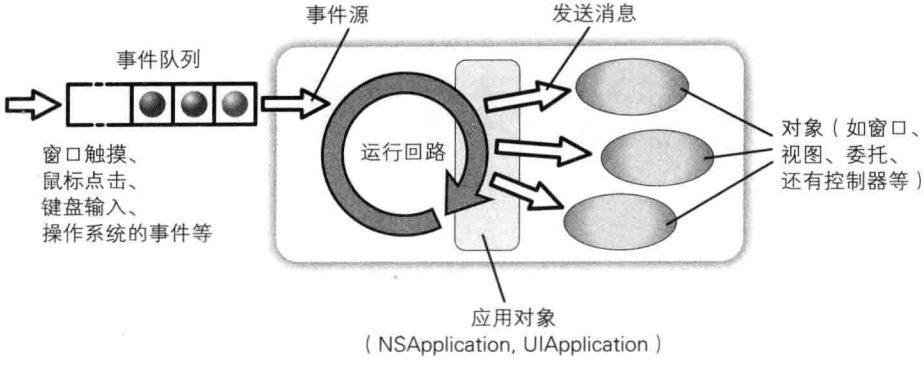


图 15-1 概括说明了应用和运行回路的处理流程。运行回路从操作系统（更精确地说是窗口服务器）中接收事件，并根据事件种类和状态来调用相应的例行程序，同时也会忽视那些没必要处理的事件。这些事件的到来非常随机，当应用程序正在处理某个事件消息时，新接收到的消息就不能立即被处理，而必须被放到等待队列中，等到空闲的时候才能被处理。没有消息到来的时候，应用会进入休眠等待的状态。

从应用程序开发者的角度来说，某个事件触发时，总有一个 Objective-C 的对象接收到该事件消息。例如：在创建 GUI 应用时，每个 GUI 组件都有对应的动作消息（action message），当按钮、菜单等组件被点击时，这些行为消息就会被发送到相应的组件对象中处理（见 8.2 节）。在视图上进行拖曳操作时也一样，视图方法会获取描述事件的参数并传给视图对象，这样视图对象就能成功地获得拖曳操作的坐标等信息。

因此，开发者只要集中精力实现消息事件的处理方法就可以了。这样一来，编程就变得非常简单，只要实现用户的各种操作请求即可。也就是说需要实现并封装一些方法以应对用户发出的命令和操作。而命令行程序从开始执行到结束基本上都是流水线式的处理，可见两者的编码方式明显不同。

因为 Cocoa 应用本身就有 GUI 功能，所以一旦开始运行，就一定会产生一个运行回路，它也称为主运行回路。同时，应用的事件处理或资源的管理功能需要有一个对象来完成，所以 Mac OS X 中提供了 NSApplication 类的实例，iOS 中提供了 UIApplication 类的实例，该实例会根据操作系统发

送的事件选择对应的处理对象，并发送相应的消息。关于这些类的情况在下一章节中还会接触到。

回路运行后，当有新的事件消息到来而别的处理还没有结束的情况下，应用就可以把该事件放入等待队列中并在之后按顺序执行。这种性质非常适合多线程的异步执行。多线程应用的情况下，在应用的主运行回路之外，每个线程都会启动自己的运行回路，并将其用于线程间的通信等。应用使用 NSRunLoop 类来访问运行回路，多线程的相关内容请参考第 19 章。

在使用引用计数方式的情况下，主运行回路在启动事件处理方法时会生成一个自动释放池，并在方法终止的时候释放它。而垃圾回收的情况下，一个事件处理完后就会启动一个垃圾回收器。应用中执行的方法基本上是**自己管理自动释放池，不需要担心什么时候进行垃圾回收**。当然，如果某个方法要生成很多临时对象，最好在合适的地方释放这些内存。而主线程之外执行的方法则要自己生成自动释放池，这一点请注意。

15.1.2 定时器对象

现在介绍一下能够在特定的时间或者按照一定的时间间隔来发送指定消息的组件。如果能灵活使用该组件，就能够轻松地实现一些必须使用并行处理才能够实现的操作。

在 Foundation framework 中，NSTimer 类能够让指定的消息在一定的时间间隔后执行。而**定时器对象**（或定时器）即实现了 NSTimer 类的实例。

定时器的使用必须要有运行回路。在运行回路上注册定时器后，到达规定的时间时，运行回路就会调用注册的方法来处理。

这里只简单说明一下 NSTimer 的主要方法。详细介绍请参考相关文档。此外，NSTimer 的接口在 Foundation/NSTimer.h 文件中有详细描述。

```
+ (NSTimer *) scheduledTimerWithTimeInterval: (NSTimeInterval) sec
                target: (id) target
              selector: (SEL) aSelector
              userInfo: (id) userInfo
               repeats: (BOOL) repeats
```

创建NSTimer对象，并在运行回路上注册之后返回。该对象已实现了运行回路。
该方法自身的调用很快就会结束，sec变量指定的秒数过去后，定时器被触发，参数aSelector指定的消息被发送给target参数的对象。NSTimerInterval的类型是实数，目前由typedef定义为double类型。aSelector指定把定时器对象当作唯一参数的选择器。定时器对象本身就是消息发送时的参数。如果要附加信息，通过参数userInfo指定包含该信息的对象即可。无附加信息时，则将userInfo设为nil。参数repeats为YES的时候，同一个消息会按照指定的间隔被反复发送。为NO时，消息只被发送一次。

```
- (id) userInfo
```

返回作为附加信息的定时器对象。主要被用于获得定时器信息。

```
- (NSTimeInterval) timeInterval
```

返回定时器对象中设定的时间间隔。

- (void) **invalidate**

将定时器对象设置为无效，使运行回路释放定时器消息。

15.1.3 消息的延迟执行

下面介绍一下经过一段时间后再执行消息的方法。该方法由 NSObject 定义，所有对象都可以使用。同样，延迟执行也要有运行回路。

- (void) **performSelector:** (SEL) aSelector
withObject: (id) anArgument
afterDelay: (NSTimeInterval) delay

该消息被发送后，至少要经过delay秒，aSelector指定的消息才会被真正发送给anArgument参数指定的接收者。

+ (void) **cancelPreviousPerformRequestsWithTarget:** (id) aTarget
selector: (SEL) aSelector
withObject: (id) anArgument

如果有使用performSelector:withObject:afterDelay:这个三个实例方法注册的请求，则取消。在该方法中，指定的选择器与参数对象必须一致。而在方法cacelPreviousPerformRequestsWithTarget:中，只要目标对象一致就可以取消。

和使用定时器对象的情况相比，虽然该方法不能定义重复执行的方式，但是它可以指定发送给目标对象的参数，具有高灵活性。

显然，延迟执行可以实现在设置的时间间隔后执行某种操作。而除此之外，在使消息处理和 GUI 操作并发执行时，以及在消息没有按照预期顺序执行时，通过延迟执行都能有效地解决这些问题。

例如，我们都知道 GUI 组件的下拉菜单，当用户选择菜单项的时候，只有在相应的处理结束后菜单才会关闭。根据下拉菜单的选择的不同，在某些情况下，当窗体大小和形状发生变化时，下拉菜单也会保持着被打开的状态并停留在原来的位置上。为了避开这样的情况，只需在选择菜单项时使用延迟执行，并预定调用方法的时机即可。

15.2 委托

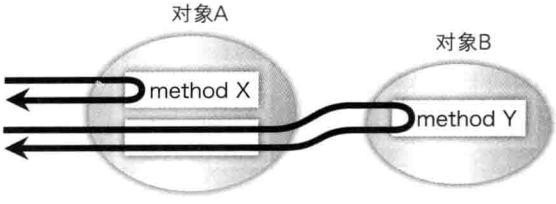
15.2.1 委托的概念

当对象需要根据用途改变或增加新功能时，为了执行新添加的处理，就需要引用一个特殊的类似于“被咨询者”的对象。这个对象就称为**委托**（delegate）。委托可在运行时动态分配。

委托是对象之间分担功能并协同处理时的一个典型的设计模式。在面向对象中，委托一般可以

被解释为“某对象接收到不能处理的消息时让其他对象代为处理的一种方式”。在图 15-2 中，对象 A 可以处理消息 X，但是不能处理消息 Y。这里，对象 A 委托对象 B 来处理消息 Y，但表面上看却是 A 处理了两个消息。这就是典型的委托。

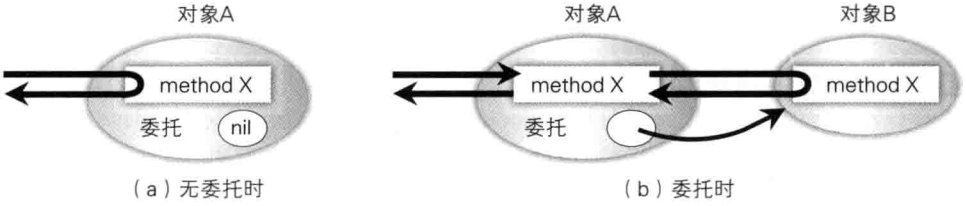
图 15-2 一般的面向对象中的委托概念



15.2.2 Cocoa 环境中的委托

在 Cocoa 环境中，与其说委托是处理自身不能处理的消息，还不如说它是应用为了添加必要的处理而“增设”的对象。图 15-3（a）是非委托的情况，这种情况下会执行默认处理（也可能无法处理）。（b）为委托的情况，这种情况下会与委托对象协同处理或完全由委托对象处理。

图 15-3 Cocoa 环境中委托的概念



（a）无委托时　　（b）委托时

委托有很高的可复用性，在作为组件使用的类中十分有效。通过使用委托，可以在不损害原类别的独立性的同时，给软件增加独立的功能。

举例说明如下。

Application 框架的 NSWingdow 类是普通显示界面的高可复用类。NSWindow 实例可以指定委托对象进行某些处理。窗体显示、主窗体显示、按下关闭键、尺寸设定等消息发生时，委托对象都会收到通知，以及关于是否能处理这些消息的咨询。

例如，按下窗体的关闭按钮时，按照默认操作该窗体会立即被关闭。但是，有时候也会希望在某个操作结束之前不要关闭窗体。按下窗体的关闭按键后，将询问是否可以关闭的消息发送给委托对象。委托对象就可以根据当时的情况，拒绝关闭或弹出用户确认窗体。

再举一个关于 iPhone 等内置加速度传感器的例子。

加速度传感器可以使用 UIKit 框架的 UIAccelerometer 类。加速度传感器是单体类，也就是只能有一个实例的类型。那么应用中如何感知 iPhone 的晃动及倾斜呢？

实际上是创建单个对象，将其注册为 UIAccelerometer 实例的委托。这样，检测到的加速度值作

为消息每隔一定时间就传送给委托对象，不需要再询问 UIAccelerometer。应用既可以一直获得加速度值信息，也可以在传感器发生较大变化时获得。这其中的差异在委托侧的程序中实现。

委托可为已有类增加新功能，是一种灵活性较高的获取并利用信息的方式。而且，一个对象并不是只能有一个委托，某些情况下一个对象也可以拥有多个委托。

同样的事件，如果使用继承的话会怎样呢？虽然在某种程度上使用继承也可以实现，但在运行时无法灵活地分配、切换委托，而且一个对象也不能同时拥有多个对象的委托。因此，建议只将默认操作之外的功能在其他类中实现，这样就能写出高独立性的软件。

毋庸置疑，现实中也存在一些建议使用继承的情况。虽然继承和委托必须要区分使用，但委托有很多继承没有的优点。熟练使用委托可以在很多情况下达到事半功倍的效果。

15.2.3 委托的设置和协议

在 Cocoa 环境中，委托常见的实现例子是名为 delegate 的 id 类型的实例变量。设置委托、返回委托对象的方法使用如下函数。

- (void) **setDelegate:** (id *<XxxxxDelegate>*)anObject

将参数对象设置为委托。一般不需要保留(retain)参数对象。

- (id *<XxxxxDelegate>*) **delegate**

返回接收的委托对象。

一直以来，委托的类型只使用 id 类型，Mac OS X 10.6 以后，开始显式调用上述协议方法。协议名就是包含委托的类名加上“Delegate”。例如，NSWindow 类的委托的协议就是“NSWindowDelegate”这个名字。大部分情况下，协议在包含委托的类的头文件中声明。而引用在类和协议中多采用不同的声明方式。

委托也可以作为属性类声明。例如，UIKit 的 UIPickerView 类中有如下方式声明。

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

在此例和上述一般的设置方法中，由于不保留委托对象，因此解除委托功能时就需要显式设置为其他值或者 nil，否则就有成为空指针的危险。

使用 ARC 时，可以使用弱引用来定义委托。

```
@property(weak) id <xxxDelegate> delegate;
```

Foudation 框架、Application 框架、UIKit 框架中很多类都有委托功能。在实际编程时，除了要关注各类中可以调用的方法之外，还需要注意可以用委托实现什么样的功能。

15.2.4 使用委托的程序

将自定义类的实例作为某些对象的委托来调用时要怎么做呢？例如，至于是否关闭窗体，NSWindow 会向委托发送 windowShouldClose：消息来询问。委托向消息发送者返回 YES 时窗体关闭，返回 NO 时则操作取消。

如果要将某个对象作为委托使用，就需要在该类的接口部分中声明使用委托的协议。定义实现委托方法的范畴就是一个好方法。而且，一个对象也可以作为多个类的委托进行操作。此时将使用与之相对应的多个协议。

声明委托的协议时，基本上所有方法中都要指定 @optional 选项。持有委托的对象会在运行时检查该委托所能处理的消息。委托中没实现的消息不会被发送。所以，委托对象只需实现相应的处理方法即可，而不需要实现协议中记录的所有方法。也就是说，在之前的窗体例子中，只要实现windowShouldClose:：方法就可以了。

此外，由于某些情况下向委托发送消息是以存在运行回路为前提的，因此我们就需要注意生成命令行程序时的情况。例如，在 Application 框架中，NSSpeechSynthesizer 类可以根据声音合成来播放文本。虽然通过命令行程序也可以执行播放功能，但向委托发送终止播放的消息时，如果没有运行回路，该消息则无法执行。

一方面，自定义类虽然也可以有委托功能，但是实现起来并不简单。实现时需要使用respondsToSelector:：等来检查委托是否可以处理某个消息。

使用委托的程序例子，可参看第 17 章。

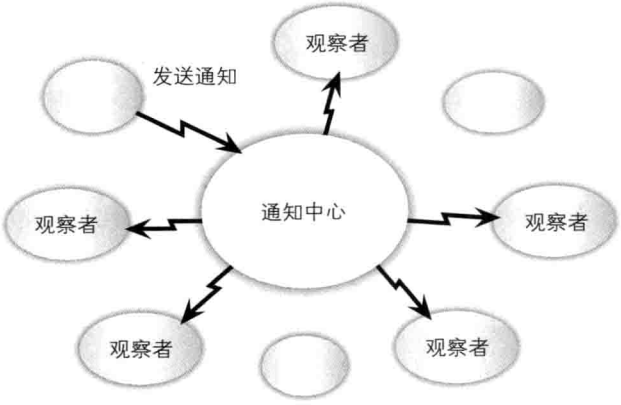
15.3 通知

15.3.1 通知和通知中心的概念

像警车鸣响警报、拉面摊儿吹起喇叭一样，现实中经常需要将发生的事情广泛通知给相关人员。同样，在面向对象的程序中，有时也需要将发生的事件通知给多个对象。例如，某个窗体代替别的窗体显示在前面时，为了执行这一行为，其他的窗体、面板或菜单的显示都要发生变化。这时，程序就需要将窗体在前面显示这一事件向关联的多个对象发出通知。

在 Foundation 框架中，这种消息发送方式就称为**通知**（notification），它的概念如图 15-4 所示。

图 15-4 通知中心的概念



程序内提供了**通知中心**（notification center）这个对象。期望取得通知的对象预先向通知中心注册期望取得的通知。通知有多种类型，分别以通知名来标记。也可以自定义新的通知名。

某对象向通知中心发送消息发送请求，这称为**发送**（post）通知。于是，只要是注册过该通知的对象，都会获得通知中心推送的消息。

消息发送目标，也就是在通知中心注册的对象，称为**观察者**（observer）。对象在通知中心将自己注册为观察者时，会指定接收什么名字的通知以及什么选择器的消息。因为通知的接收方法是在观察者一方定义的，所以很容易实现。而且也可以指定只接收特定对象发送（post）的通知。观察者很多或者完全没有都没有关系。

无论什么对象都可以自由使用通知的发送机制，而且不需要在通知中心注册。至于观察者对象是什么类的实例或者定义了什么样的方法，发送通知的对象都不需要知道。两者之间必须的共享的仅仅是通知的名字。

这样的机制既不会降低发送通知的对象和接收通知的观察者之间的独立性，又可以实现向多个对象发送消息。

区别于发送者和接收者的一对一通信，某个对象向特定的多个对象发送消息，这样的通信形式称为**多播**（multicast）。如果说一对一通信是电话或信件，那么多播就可以被比作是演讲会或邮件杂志。这里所说的通知的功能也可以被认为是多播的一种。此外，向非特定的多个对象发送消息称为**广播**（broadcast）。这和无线电播放或街头散发的传单一样，不需要选择特定的接收对象，有时多播也称为广播。

使用通知的程序例子，在第 17 章中有详细介绍。

15.3.2 通知对象

向通知中心发送消息时，必要的信息会在 NSNotification 类实例中集中后发送给通知中心。这个对象就被称为**通知对象**，或者简称为通知。观察者从通知中心取得消息时也会取得附带的通知对象。

通知对象的接口在头文件 Foundation/NSNotification.h 中定义。

通知对象包含以下信息。

名字（name）

用来识别通知的短文本。向 NSNotification 接口发送如下消息就可以取出名字。

```
- (NSString *) name
```

对象（object）

和通知一起发送的附带信息的对象。多为发送通知的对象，也可以为 nil。用如下消息取出。

```
- (id) object
```

用户字典（userInfo）

为了传递和通知相关的各种信息而使用的 NSDictionary 字典。通过下面的消息取出，有时为 nil。

```
- (NSDictionary *) userInfo
```

因为向通知中心发送通知时会自动创建通知对象，所以没有必要自己创建。创建通知对象，可以使用如下任意一种 NSNotification 的类方法。

```
+ (id) notificationWithName: (NSString *) aName
                     object: (id)anObject
                   userInfo: (NSDictionary *) userInfo
```

指定通知名、对象和用户字典，生成临时通知对象。

```
+ (id) notificationWithName: (NSString *) aName
                     object: (id) anObject
```

指定通知名和对象生成临时通知对象。用户信息字典为nil。

对象通常指定为发送通知的对象（即 self），也可以设定为其他对象或 nil。用户信息字典一般指定包含了通知相关信息的 NSDictionary 接口，也可以指定为 nil。

通知名方面，可以自己决定新的名字，也可以预先在系统中决定。这些都在相关的类文档中有记载。

15.3.3 通知中心

通知中心使用 NSNotificationCenter 的类接口实现。通知中心的接口在头文件 Foundation/NSNotification.h 中定义。

（1）默认的通知中心

各进程中都会预先准备一个通知中心，一般不需要自己创建。这就是默认的通知中心，可以使用如下类方法取得。

```
+ (id) defaultCenter
```

(2) 通知的发送

可以使用如下任意一种方法来发送通知。

```
- (void) postNotification: (NSNotification *) notification
```

发送通知对象给接收该通知的通知中心。参数notification不能为nil。

```
- (void) postNotificationName: (NSString *) notificationName
                       object: (id)anObject
                     userInfo: (NSDictionary *) userInfo
```

指定通知名、对象和用户字典生成通知，然后发送给接收者的通知中心。
同样的方法postNotificationName: object:是将用户信息字典指定为nil后发送。

(3) 观察者注册

观察者对象要从通知中心获得消息，必须实现如下方法。观察者注册时，该方法的选择器也会同时注册。

```
- (void) XXXXXX: (NSNotification *) notification
```

观察者注册使用如下通知中心的方法。

```
- (void) addObserver: (id) anObserver
            selector: (SEL) aSelector
                name: (NSString *) notificationName
              object: (id) anObject
```

对象anObserver在通知中心上注册为观察者。参数aSelector为包含anObserver的方法的选择器。
在参数notificationName中指定期望发送消息的通知名。指定为nil时不会设定通知名相关的条件。
只接收特定对象发送的通知时，参数anObject中需指定该对象。如果指定nil则表示没有设定发送源。
两者都指定为nil时，则可以接收所有关于发送消息的通知。

发送通知时，参数 aSelector 中指定的消息将被发送给观察者。参数为通知对象。

仅指定特定的多个通知名的通知时，可以将各个通知名分别在通知中心注册，或者指定通知名为 nil 使其发送有关所有通知名的消息，并在接收后只处理那些必要的通知。

(4) 删除观察者的注册

作为观察者注册的对象也可以取消注册。这需要使用如下任意方法。

```
- (void) removeObserver: (id) anObserver
```

删除参数anObserver为观察者的所有设定。

```
- (void) removeObserver: (id) anObserver
                   name: (NSString *) notificationName
                 object: (id) anObject
```

删除对指定的观察者、通知名和发送源的监视设定。参数为nil时则不会被作为条件使用。

例如，从默认的通知中心删除对象 obj 为观察者的全部设定，如下所示。

```
[[NSNotificationCenter defaultCenter] removeObserver:obj];
```

从默认的通知中心删除指定对象 pobj 为发送源的所有设置，可采用如下方式。

```
[[NSNotificationCenter defaultCenter]
        removeObserver:nil name:nil object:pobj];
```

这里需要格外注意对象的内存管理。

首先，使用引用计数管理的情况下，通知中心在注册观察者时，并不保留（retain）观察者及发送源对象。所以，在释放这些对象之前，要确实从通知中心删除相关设置。如果不这样的话，指向释放对象的指针将成为空指针。

使用垃圾回收机制时，观察者和发送源对象会使用弱引用在通知中心注册。所以，无论这些对象在通知中心是否注册，都有可能被回收释放。释放的指针也同时被清零，因此无需显式删除观察者的注册。

15.3.4 通知队列

调用发送通知的方法时，所有相关的观察者都会被有顺地发送通知消息，这些处理终止后，发送方法也会终止。但这样的操作有时会产生问题，所以便引入了**通知队列**（notification queue）。通知队列是将通知对象临时存储在等待队列中，然后按照先入先出的原则（FIFO，First In First Out）向通知中心发送消息。通知队列是通过 NSNotificationQueue 类来实现的。

这里解释一下通知队列的操作概要。详细介绍请参考相关文档。

通知队列提供了两种常见的通知操作。

（1）异步发送

将通知追加到队列，然后在运行回路完成当前处理或者运行回路中的输入都被处理完成后，再发送通知。通过使用该方法，就可以直接终止队列中追加的方法并进行下一项处理。这样一来，通知的发送就会在一连串处理之后再执行。这种方式也称为异步（asychronous）发送。一般的处理方式是同步（synchronous）发送。

（2）合并相同的通知

通知队列中有相同的通知时，将他们合并为一个，以此来删除多余的通知。例如，窗体内再次进行绘图操作的通知被从多处发送时，并不需要全都处理。重复的通知只处理一次会更加高效。通知名相同、通知源相同，或者两者都相同时，通知就可以被合并。

专栏：通知名或异常名的定义 COLUMN

自定义使用通知名或异常名等时，一般都不直接使用常数字符串，而是用全局变量或宏名来定义。虽然看起来没什么用，但却是十分重要的。

例如，假设某程序中有如下部分，它不能正确运行。而如果追究原因的话就非常麻烦，而且最终也注定是无用功（最后缺少“i”）。但如果是变量名或宏名，那么即使是拼写错误也会被编译器指出。

```
[[NSNotificationCenter defaultCenter]
      addObserver: self selector: @selector (showProgress:)
      name: @"anonymousProcessPriority" object:nil];
...
[[NSNotificationCenter defaultCenter]
      postNotificationName: @"anonymousProcessPriority"
      object:self];
```

此外，定义全局变量时，既可以节约内存，也可以快速进行字符串比较。

15.4 反应链

15.4.1 反应链概述

反应链（responder chain）是具有层次结构的 GUI 组件间自动发送消息的一种方式。

从菜单中选择“复制”“取消”等时，或者通过鼠标、按键输入时，根据位于最前端的窗体的不同，以及所选择的窗体 GUI 组件的不同，处理也不相同。虽然基本上是最前选择的 GUI 组件进行处理，而如果该组件不能处理消息，那么就可能由包含该组件的上层组件处理。此外，包含组件的窗体或委托或许也可以处理。

处理消息的候补组件对象如算珠般串联在一起，将消息传递给这之间的某个对象时，在发现可以处理该消息的对象之前，会顺次向后面的对象“转交”该消息。这样的构成就称为反应链。

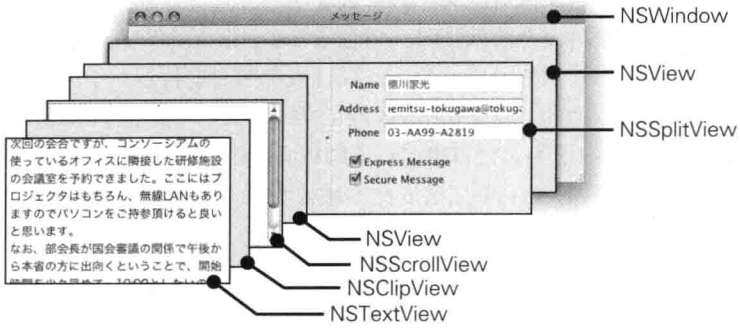
在 Mac OS X 中，按键、滑块、文本域等组件以及窗体、面板等对象都是 Application 框架的 NSResponder 类的子类。同样，在 iOS 中是 UIKit 框架的 UIResponder 类的子类。这些接口构成了反应链。当存在包含某组件 A 的组件 B 时，反应链通常按照 A—>B 的方向传递消息。在 Xcode 中创建 GUI 时会同时生成反应链，可以使用程序进行连接，也可以在运行中动态变更。而且元素（回应）即使不是 NSResponder（或者 UIResponder）子类也没有关系。

在图 15-5 的窗体 GUI 组件层次中，文本输入部分和信息显示部分分别位于左右两侧，通过中央的间隔可以调整大小。文本输入部分包含在带有滚动功能的组件中，如果也包含配置用的组件的话，实际上就组成了图示的关系。窗体之外是 NSView 的子类，NSView 同时也是 NSResponder 的子类。

从图最前端的 NSTextView 接口到窗体，按此顺序就构成了像算珠一样的反应链。而如果指定了窗体的委托，那么就需要确认委托能够处理哪些消息。

各窗体中，最初被发送消息的对象称为**第一反应者**（first responder）。第一反应者基本上都是光标最近选择的对象。

图 15-5 GUI 组件的层次和反应链



15.4.2 应用中的反应链

在 Mac OS X 平台下，应用中的窗体分为处理应用的主要操作的窗体和提供辅助功能的窗体。前者称为**应用窗口**，特别是读取文件信息进行显示或编辑的窗体称为**文档窗口**。其他窗体则包括打开面板、字体面板，以及表示各种信息的窗口等（图 15-6）。

在应用窗口或文档窗口中，显示在最前端的窗口称为**主窗口**。在包含面板等在内的其他窗口中，最前面的窗口则称为**关键窗口**。主窗口也可以是关键窗口。

图 15-6 窗口和面板的种类

反应链内的搜索，首先从关键窗口的第一反应者开始。关键窗口中无法应对找不到时，如果存在其他主窗口，主窗口的反应链也会按同样方式搜索。

有时主窗口也不能处理时，消息最终会被传给 NSApplication（管理应用全体的类）的接口。这样一来，当不能处理的消息就会被忽视，或者响起警告音。

在 iOS 平台下，表示绘图区域的类 UIView 是其他 GUI 组件的超类，同时也是表示反应者的 UIResponder 的子类。接口由窗口（UIWindow 类）、UIView 类及其子类 GUI 组件有层次地重叠构成。UIKit 会自动将任意一个按键或文本域作为第一反应者。第一反应者不能处理的消息，将沿着重叠的 GUI 组件，被有序地向窗口甚至引用程序（UIApplication 类）传递。

第 17 章的示例程序中，也存在一些处理消息的对象由反应链构成来动态决定的地方。

在 Cocoa 环境中，生成包含 GUI 的应用必须掌握反应链相关的知识。详细内容请参考“Cocoa Event-Handling Guide”“Event-Handling Guide for iOS”等参考文档。

15.5 消息转送

15.5.1 消息转送的构成

将消息发送给没有实现该消息方法的对象时，通常会出现运行时错误。但是，我们可以将不能被处理的消息转送给其他对象，这样的功能是可以实现的。

下面将详细说明转送的结构。

首先，将某消息发送到相应的接收者。这里，如果接收者没有实现应对该消息的方法，运行时系统就会发送如下消息给接收者。

```
- (void) forwardInvocation: (NSInvocation *) anInvocation
```

这个方法在 NSObject 中定义，所有的对象都可以处理。在 NSObject 中，这个方法可调用如下方法。

```
- (void) doesNotRecognizeSelector: (SEL) aSelector
```

生成错误消息，表示异常 NSInvalidArgumentException 发生，无法处理参数选择器对应的消息。

也就是说，只要接收者重定义了 `forwardInvocation:` 方法，当被发送不能处理的消息时，就可以将其转送给其他对象，或者自己执行错误处理。

15.5.2 消息转送需要的信息

正如上面所说的那样，方法 `forwardInvocation:` 的参数会被传入 NSInvocation 对象，而利用该对象包含的信息，就可以实现消息转送。NSInvocation 的对象中保存了目标、选择器、参数等

消息发送所必需的全部元素。

下面是 NSInvocation 类的主要方法。NSInvocation 的接口在头文件 Foundation/NSInvocation.h 中声明。

- (SEL) **selector**

  返回设定的选择器。

- (id) **target**

  返回设定的目标。

- (void) **invokeWithTarget:** (id) anObject

  向目标参数对象发送表示接收者的消息。将消息的结果返回给源发送者。

也可以自定义 NSInvocation 接口，或者重定义选择器、参数、返回值，这里不再赘述。

15.5.3 消息转送的定义

使用上述的 NSInvocation 信息，forwardInvocation:就可以被重定义，从而实现消息转送。但是，可以转送的仅仅是参数个数固定的方法。参数为不确定的列表的方法则不能转送。

例如，调查不能处理的消息是否可以转送给对象 fellow，当不能转送时则交由超类处理，可采用如下方式定义。

```
- (void)forwardInvocation:(NSInvocation *)anInvocation
{
    SEL sel = [anInvocation selector];
    if ([fellow respondsToSelector:sel])
        [anInvocation invokeWithTarget:fellow];
    else
        [super forwardInvocation:anInvocation];
}
```

再有，为了使运行时系统能够使用转送目的地的对象信息生成 NSInvocation 实例，必须重新定义返回的**方法签名**（method signature）对象。

为了表示方法签名，定义了 NSMethodSignature 类。它的对象保存着方法的参数和返回值，但在消息转送或通信之外的情况下，程序中是不处理的。

返回方法签名的方法在methodSignatureForSelector:和 NSObject 中定义。例如，转送目的地的对象如果是 fellow，就可以按如下方式重新定义。

```
- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector
{
    if ([super respondsToSelector:aSelector])
        return [super methodSignatureForSelector:aSelector];
    return [fellow methodSignatureForSelector:aSelector];
}
```

然而，使用转送方法处理的消息不能被respondsToSelector:等调用。当需要知道该消息是否为目标对象可处理的消息时，必须重定义respondsToSelector:等方法。

15.5.4 禁止使用消息

如前所述，方法doesNotRecognizedSelector:表示消息不能处理时所产生的异常。

利用此特性，在使用某个方法时就可以定义运行时产生的错误。一个典型的例子就是，当用超类定义的方法在子类中不能使用时，可以写下该方法及时禁止其使用。

例如，想禁止使用方法setSize:时，可采用如下方式定义。这里 _cmd 为方法的隐藏参数，表示方法的选择器（见 8.2 节）。所以，写下@selector(setSize:)也是一样的。

```
- (void)setSize:(NSSize *)size
{
    [self doesNotRecognizeSelector:_cmd];
}
```

15.5.5 程序示例

这里定义一个行为类似于 NSString 子类的 ReversibleString 类。该类包含reversedString方法，返回反转后的实例变量的字符串。自身不能处理的其他消息则被转送给实例变量的字符串对象。而且使用 ARC 来管理内存。

代码清单 15-1 类 ReversibleString 的定义及测试程序

```
#import <Foundation/NSString.h>
#import <Foundation/NSMethodSignature.h>
#import <Foundation/NSInvocation.h>
#import <stdio.h>
#import <stdlib.h>

@interface ReversibleString: NSObject
{
    NSString *content;
}
- (id)initWithString:(NSString *)string;
- (id)reversedString;
@end


@implementation ReversibleString

- (id)initWithString:(NSString *)string
{
    if ((self = [super init]) != nil)
        content = string;
```

```
    return self;
}

- (id)reversedString
{
    unichar *buffer;  // 将 Unicode 字符串反转
    int length, i, j, tmp;
    id reversed;

    if ((length = [content length]) <= 0)
        return @"";
    buffer = malloc(sizeof(unichar) * length);
    [content getCharacters:buffer range:NSMakeRange(0, length)];
    for (i = 0, j= length-1; i < j; i++, j--)
        tmp = buffer[i], buffer[i] = buffer[j], buffer[j] = tmp;
    reversed = [NSString stringWithCharacters:buffer length:length];
    free(buffer);
    return reversed;
}

- (void)forwardInvocation:(NSInvocation *)anInvocation
{
    SEL sel = [anInvocation selector];
    if ([content respondsToSelector:sel])  //content 可以对应时转送
        [anInvocation invokeWithTarget:content];
    else
        [super forwardInvocation:anInvocation];
}

- (BOOL)respondsToSelector:(SEL)aSelector
{
    if ([super respondsToSelector:aSelector])
        return YES;
    if ([self methodForSelector:aSelector] != (IMP)NULL)
        return YES;
    if ([content respondsToSelector:aSelector])
        return YES;
    return NO;
}

- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector
{
    if ([super respondsToSelector:aSelector])
        return [super methodSignatureForSelector:aSelector];
    return [content methodSignatureForSelector:aSelector];
}
@end

int main(void)
{
    char buf[100];
    id s, a, b, c, d, e;
    @autoreleasepool {
        scanf("%s", buf);
        s = [NSString stringWithUTF8String:buf];
```

```
        a = [[ReversibleString alloc] initWithString:s];
        b = [[ReversibleString alloc] initWithString:@"Reverse?"];
        printf("%s\n", [a UTF8String]);
        c = [[a reversedString] stringByAppendingString: b];
        printf("%s\n", [c UTF8String]);
        d = [[ReversibleString alloc] initWithString:c];
        e = [b stringByAppendingString: [d reversedString]];
        printf("%s\n", [e UTF8String]);
    }
    return 0;
}
```

类 ReversibleString 包含 content 实例变量，并向它转送消息。为了转送消息，方法 `forwardInvocation:` 和`methodSignatureForSelector:`都被重新定义了。

执行结果如下所示。该例子表示的是从文件中读入“烧毁竹林”字符串（UTF-8）。可见，除自定义的方法之外，还能够应对 NSString 方法 `UTF8String` 和 `stringByAppendingString:`。

```
烧毁竹林
烧毁竹林Reverse?
Reverse?esreveR烧毁竹林
```

然而，类 ReversibleString 也没有实现 NSString 所有的接口方法。例如，NSString 的初始化器就没有实现。由于在实例初始化完成前，实例变量 content 还没有准备好，因此是不能进行消息转送的。

15.6 撤销构造

15.6.1 撤销构造的概念

Cocoa 环境中，为了能够**撤销**（undo）或**重复**（redo）之前的操作，准备了专门的类 `NSUndoManager`。这里将简要说明使用 NSUndoManager 的撤销结构。

取消之前的操作，可以通过记录操作前的状态来还原，或者执行与当前操作效果相反的操作。NSUndoManager 根据后者实现了撤销。

NSUndoManager 也是 Foundation 框架的类之一。接口在 Foundation/NSUndoManager.h 中声明。将类方法 `alloc` 和初始化器 `init` 相组合就可以创建实例。下面将 NSUndoManager 的实例称为**撤销管理器**。

在包含多个文档窗口的应用中，每个窗口都有一个撤销管理器。有的应用也可以只限定一个。然而，在使用 Application 框架中 NSDocument 类来管理文档窗口的应用中，可以从 NSDocument 实例取得撤销管理器。

应用中运行某种操作时，撤销管理器会同时记录包含逆作用的消息（或消息群），这就是撤销的

基本结构。撤销管理器将这些消息有序地记录在撤销栈中，当有撤销请求时，就取出最新的记录内容执行。

为了说明，下面举一个最简单的例子。

```
- (void)increment:(int)n
{
    撤销管理器中记录 [self decrement: n];
    value += n;
}

- (void)decrement:(int)n
{
    撤销管理器中记录 [self increment: n];
    value -= n;
}
```

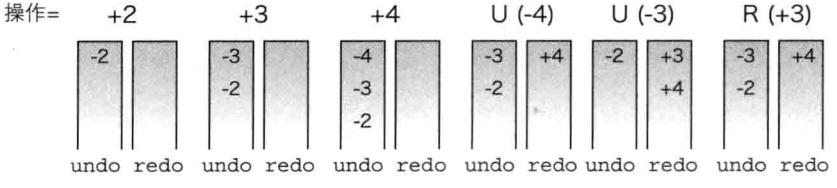
由于这两个方法都包含逆操作，因此执行该方法后就能取消已完成的方法，这样也就实现了撤销操作。但是撤销的方法在执行时，方法内也会执行“撤销管理器的记录”。

实际上，在执行撤销时，**一旦“撤销管理器的记录”被执行，NSUndoManager 就会将该消息记录在重复用的栈上**。重复和撤销是反义关系，在重复中执行的“撤销管理器的记录”保存在撤销栈中。所以，按上述方法实现时，就实现了撤销和重复操作。

例如，假设方法increment:的调用（用数字标记）和撤销、重复的要求（用 U 和 R 标记）按如下顺序产生。此时栈状态如图 15-7 所示。

+2　+3　+4　U　U　R

图 15-7　注册撤销管理器和栈的变化



15.6.2　在撤销管理器中记录操作

在撤销管理器中记录操作有两个方法。其中之一是使用如下方法记录撤销时本应执行消息的接收器、选择器及参数对象。与行为方法相类似的方法也会被记录。

```
- (void) registerUndoWithTarget: (id) target
                        selector: (SEL) aSelector
                          object: (id) anObject
```

另一个就是使用如下方法。该方法的返回值为撤销管理器自身。

```
- (id) prepareWithInvocationTarget: (id) target
```

例如，上例中“记录 [self decrement: n]”的部分可以按如下方式书写。但是，这样一来 decrement： 消息就会在这里被处理完成。而上述方法的返回值原本可以处理 decrement： 吗？实际上这里有些技巧。

```
[[undoManager prepareWithInvocationTarget:self] decrement: n];
```

该方法中，撤销时的接收者在接收消息的同时，为了对下面收到的消息（此例中为 decrement： 消息）进行特别处理，还会进行些准备工作。具体来说就是，**接收应该在撤销时发送来的消息，并将其作为NSInvocation对象完整保存起来**。15.5 节中为了转送消息而使用了 NSInvocation，而这里使用 NSInvocation 则是为了保存。行为方法不限制消息形式。

撤销管理器在请求撤销、重复时使用如下方法。

```
- (void) undo;
- (void) redo;
```

在窗口中包含撤销管理器时，常见的方法是通过窗口委托来管理撤销管理器。此外，就如 Application 框架的类 NSTextView 那样，有的则各自包含撤销管理器。所以，从菜单等发出撤销 / 重复请求时，该消息就会被通过反应链传递。第 17 章的例题中有简单的撤销功能的实现，请参考。

撤销管理器中还有其他方法，如清除栈、将多个方法调用合并为一个撤消操作来记录等，关于这些，本书中不再一一介绍。详细信息请参考“Undo Architecture”等参考文档。

第16章

应用的构造

本章将介绍 Cocoa 应用的构造、资源的访问以及多语言方法等话题。

Foundation 框架中提供了支持这些功能的类。

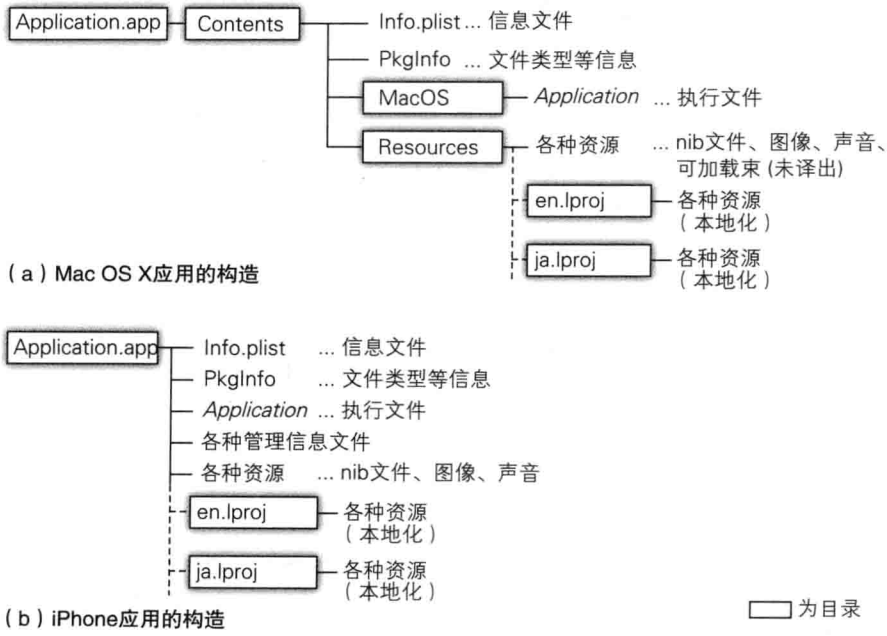
16.1 应用束

16.1.1 应用束的构造

Cocoa 应用会将执行文件或必要的资源格式保存在一个委托结构中，这称为**应用束**（application bundle）、应用包装（wrapper）或应用包。资源就是应用执行时所必须的图像、声音、文本、设置信息等文件。像这样，将几个文件或目录以确定形式保存到目录结构中，然后再作为一个集合进行处理，在 Cocoa 中是很常见的方法。这样的结构一般称为**束**。

图 16-1 显示了应用束的基本构造。通常，在使用 Xcode 进行开发时，应用束会自动产生，程序没有必要逐一指定目录中的结构。此外，图中的概况在将来可能会发生变化，但实际中的差异会被使用束的 NSBundle 类所吸收，因此写代码时不需要太在意这一点。

图 16-1 应用束的结构



（a）Mac OS X应用的构造

（b）iPhone应用的构造

通过上图可以看出，Mac OS X 中产生了多个子目录，而 iOS 则并不明确区分存放执行文件及各种资源。二者都有 Info.plist 文件，该文件为属性列表（见 13.3 节）格式，记载着应用相关的重要信息。此外 PkgInfo 文件中还记录了与 Mac OS 文件类型及创建器类似的信息。

16.1.2 nib 文件和各语言资源

使用 Xcode（Interface Builder）生成的 GUI 定义被写到 **nib 文件**中。nib文件的后缀为“nib”，也就是 Next Interface Builder，是 NeXTstep 的由来。在 nib 文件中，应用的菜单和窗口组件的配置等信息会被归档化，并在执行时被动态读入。根据用途或目的的不同，包含多个 nib 文件的应用也很常见。

nib 文件的后缀名为“nib”或“xib”，xib 文件为 XML 格式文件，两种后缀名的文件内容相同，在构建应用时，会作为带有后缀“nib”的资源文件被保存在应用束中。

从 Xcode4.2 起，iOS 应用中开始可以使用 storyboard 文件。因为 Storyboard 是以场景移动为中心来构造 GUI 的，所以比较适合边显示边处理这样的应用的架构。在各场景内部，对象之间的关系和 nib 文件相同。在应用被构建的同时，Storyboard 内部也会变为 nib 文件资源。

在使用 nib 文件、消息、用于说明的文本及图片等资源文件时，用户可能会需要根据语言来切换资源。另一方面，图片图像及效果声音文件等与资源也被期待着可以在各语言间共享。

与语言无关的共享资源在 Mac OS X 系统中被放置在 Resources 目录下，在 iOS 系统下则被放置在应用束目录下。根据所选择的语言种类来切换的资源被放置于“语言名 .lproj”目录下。语言名采用 ISO 语言编码标准，使用 en 表示英文，用 jp 表示日文（见下文的 Column）。后缀名“lproj”来自于 language project，并从 NeXTstep 时期开始使用。

应用在运行时，会在运行环境中查找选择的（或是高优先度的）语言子目录，并使用其中的资源。NSBundle 类执行资源检索，因此程序就不需要记录指定语言的编码。

使应用能够对应特定语言的过程，称之为**本地化**（localize）。在 Cocoa 应用中，如果能在编程时使之对应多种语言，那么之后只需在应用束中追加特定语言的资源就能简单地实现本地化。

专栏：指定语言和地区 COLUMN

存储各语言资源的目录名，基本上使用 ISO 639-1 标准中指定语言的两个字母，有时也会根据需要将地区代码和本地 ID 组合起来使用。下面列举几个重要语言的指定。

在英语中，如果想要区分英国方言和美国方言，可指定为 en-GB、en-US。详情请参考“Internationalization Programming Topics”及 ISO 标准。

旧规定	新规定
English	en
French	fr
German	de
Spanish	es
Italian	it

旧规定	新规定
Japanese	ja
Chinese	zh
zh_TW（台湾）	zh-Hant（繁体）
zh_CN（中国）	zh-Hans（简体）
Korean	ko

16.1.3 信息文件的主要内容

应用束内的信息文件 Info.plist 以属性列表的形式将应用执行所必需的信息记录起来。当使用 Xcode 创建工程时，会使用“工程名 -Info.plist”这样的文件名来进行文件的编辑。构建应用时，应用束中就会基于该文件生成 Info.plist。

使用应用束时，信息文件内关键词 CFBundleIdentifier 指定的字符串称为**应用标识符**（application identifier），该标识符常被用来在系统中寻找相应的应用程序。为了不产生重名，推荐使用 Java 的包定义方式（参考附录 C）。例如 com.apple.iPhoto。

主键及其含义如表 15-1~15-4 所示。键的前缀文字 LS 是“Launch Services”的意思。键的含义和相关设定请参考“Information Property List Key Reference”等。

属性列表中实际的键值和 Xcode 中表示的字符串有时是不同的。例如，之前所述的键 CFBundleIdentifier 可能就会被表示为“Bundle indentifier”（下面称为 Xcode 名）。为表示键值原本的意思，在 Xcode 中，可以右击并选择上下文菜单中的“Show Raw Keys/Values”[①]。

▶ 表 16-1 名字设定

Xcode 名 / 键值	说明	Mac/iOS
Bundle identifier CFBundleIdentifier	应用识别名（大小写英文字母、“.”和“-”）	Mac / iOS
Bundle display name CFBundleDisplayName	本地化应用名时使用（参考 P.422 的 Column）	Mac / iOS
Bundle name CFBundleName	应用名小于 16 个字符	Mac / iOS

▶ 表 16-2 资源

Xcode 名 / 键值	说明	Mac/iOS
Executable file CFBundleExecutable	执行文件名	Mac / iOS
Principal class NSPrincipalClass	可加载束的主要类	Mac
Main nib file base name NSMainNibFile	指定主 nib 文件名（参考 P.410“启动应用”）	Mac / iOS
Icon file CFBundleIconFile	应用的图标图像文件名（参考表 16–4 的 CFBundleIconFiles）	Mac / iOS
Help Book directory name CFBundleHelpBookFolder	保存帮助的文件名（.lproj 下）	Mac
Help Book identifier CFBundleHelpBookName	帮助文件名	Mac

① 因开发环境不同而有所差异。

表 16-3 其他属性

Xcode 名 / 键值	说明	Mac/iOS
Document types CFBundleDocumentTypes	定义应用读写的文件后缀或图标图像	Mac / iOS
Bundle OS Type code CFBundlePackageType	Mac OS 的文件类型。应用为“APPL”，可加载束为“BNDL”	Mac / iOS
Bundle creator OS Type code CFBundleSignature	Mac OS 的创建器编码。不特别指定时为“????”即可	Mac / iOS
Scriptable NSAppleScriptEnabled	使用 AppleScript 来查看可否行动	Mac
Services NSServices	记述 Cocoa 平台提供的各种服务	Mac
Minimum system version LSMinimumSystemVersion	执行应用的最低 OS 版本要求	Mac
Get Info string CFBundleGetInfoString	将版权相关信息（除了版本信息）保存在文本中（不推荐）	Mac
Copyright (human-readable) NSHumanReadableCopyright	将版权相关信息（除了版本信息）保存在文本中（不推荐）	Mac
Bundle versions string, short CFBundleShortVersionString	表示应用版本号（例如 3.1.2）	Mac / iOS
InfoDictionary version CFBundleInfoDictionaryVersion	Xcode 自动生成属性列表的版本号	Mac / iOS
Bundle version CFBundleVersion	包含构建号的版本号	Mac / iOS
Localization native developme nt region CFBundleDevelopmentRegion	开发者所在地域的语言（或开发者原本使用的开发语言）	Mac / iOS
Application Category LSApplicationCategoryType	应用的种类（Business、Game、Education 等），登陆 App Store 时需要	Mac

图 16-4 iOS 固有属性

Xcode 名 / 键值	说明
Icon files CFBundleIconFiles	应用中各种大小的图标文件数组
Icon files（iOS 5） CFBundleIcons	应用中保存各种图标信息的字典
Application requires iPhone environment LSRequiresIPhoneOS	应用必须在 iOS 环境中运行

（续）

Xcode 名 / 键值	说明
Minimum system version MinimumSystemVersion	执行应用的最低 OS 版本要求（Xcode 生成）
Application does not run in background UIApplicationExitsOnSuspend	表示应用是否在后台执行
Required background modes UIBackgroundModes	表示应用必须继续在后台执行
Targeted device family UIDeviceFamily	应用的目标设备（Xcode 生成）
Application supports iTunes file sharing UIFileSharingEnabled	表示应用是否可通过 iTunes 和电脑共享文件（16.3 节）
Initial interface orientation UIInterfaceOrientation	表示最初的用户接口是横向还是纵向
Launch image UILaunchImageFile	指定启动时表示的图像文件名
Icon already includes gloss effects UIPrerenderedIcon	表示图标图像是否已经点亮
Required device capabilities UIRequiredDeviceCapabilities	列举摄像头或 GPS 等执行时必需的功能
Application uses Wi-Fi UIRequiresPersistentWiFi	表示应用是否使用 WiFi
Status bar is initially hidden UIStatusBarHidden	表示应用启动时状态栏是否显示
Status bar style UIStatusBarStyle	应用启动时显示状态栏
Status bar style UISupportedInterfaceOrientations	列举应用支持的接口

16.1.4 通过 NSBundle 访问资源

NSBundle 是为各种束提供接口的类，可以从指定的束中搜索有 GUI 定义的 nib 文件、图像、声音、加载代码等。下面展示了几个主要方法。详细情况请参考相关文档。NSBundle 的接口在头文件 Foundation/NSBundle.h 中记载。

```
+ (NSBundle *) mainBundle
```

程序所包含的应用束也称为**主束**（main bundle）。该方法返回与主束的路径相对应的对象。不能识别应用束时返回nil。

```
+ (NSBundle *) bundleWithPath: (NSString *) path
```

返回path指定路径的束对象，没有path可访问的束时返回nil。此外也有使用NSURL指定位置的方

法BundleWithURL:。

- (NSString *) **bundleIdentifier**

返回信息文件(lifo.plist)中指定的应用程序名。

- (NSDictionary *) **infoDictionary**

将信息文件(info.plist)的内容作为字典对象返回。

- (id) **objectForInfoDictionaryKey:** (NSString *) key

在信息文件(info.plist)中，返回以参数key为键值的对象。可能的话将其本地化后返回。

NSBundle 实例表示具体的束，如应用束。该实例可以寻找束内的资源。

- (NSString *) **pathForResource:** (NSString *) name
ofType: (NSString *) extension

在接收者的束内，返回name指定的名字和extentsion指定的有后缀的资源路径名。文件没有后缀时，将extension指定为nil或@""。找不到文件时返回nil。同样，也有使用NSURL返回位置的方法URLForResource:withExtension:。

上面的方法pathForResource:ofType:首先会在束内按顺序查找根据运行环境选定的语言子目录，如果找不到就在 Resources 下面（iOS 时在应用束下）寻找。这和使用 NSBundle 的查找资源的方法是一样的。

例如，可按如下方式读取应用束内的 SoCool.jpg 图像，并将其作为 NSImage（操作图像的类）的实例使用。图像文件在 Resouces 下时也可以被特定语言本地化。程序方面不用担心这一点。

```
NSImage *coolImage = nil;
NSBundle *bundle = [NSBundle mainBundle];
NSString *imgPath = [bundle pathForResource:@"SoCool" ofType:@"jpg"];
if (imgPath)
    coolImage = [[NSImage alloc] initWithContentsOfFile:imgPath];
```

获取保存资源的路径可以采用resourcePath方法，但是不推荐使用如下方式获取资源路径。当不能本地化时，也不能使用后述的指定 iOS 资源的方法。

```
rpath = [[NSBundle mainBundle] resourcePath];
imgPath = [rpath stringByAppendingPathComponent:@"SoCool.jpg"];
```

从束中获取资源路径或 URL 的方法不只上述一种。

此外，在 Mac OS X 平台中，可以利用pathForImageResource:方法获得图像路径。它是 Application 框架中被作为范畴添加到 NSBundle 中的方法，在 AppKit/NSImage.h 中声明。其他 nib 文件或声音文件的获取方法请参考“NSBundle Additions Refrerence”。

16.1.5 iOS 中资源的访问

如前所述，iOS 中也可以使用 NSBundle 方法来访问资源。但是，因为 iOS 有特殊的后缀，所以这里先介绍一下其概要。

在 iOS 中，只要开发一个应用包，iPhone 和 iPad、iPod touch 等就都可以使用。这时，为了适应所使用的设备，nib 文件的内容和图像的大小就必须要改变，但是人们又不想为各种设备分别进行编码工作。于是就可以使用这样一种指定方法，即根据所使用的设备来自动选择资源或参数。

（1）按设备准备任意资源

在 iOS 中，当束的检索请求被指定为“文件名 . 后缀”这样的文件时，就会搜索如下形式的文件。

文件名 ~ 设备 . 后缀

设备部分可以指定以下几种形式。

iphone：表示 iPhone 或 iPod touch
ipad　：表示 ipad

例如，如果束内有“Above~iphone.nib”和“Above~ipad.nib”，那么，iPhone 在运行中要加载“Above.nib”时，实际上就加载了“Above~iphone.nib”。当然，如果只有“Above.nib”，那也只能加载它了。该记法在图像、声音等其他资源中也可以使用。

（2）准备高像素图片

可以添加高像素的图片。iPhone 4 的像素是以往的 2 倍，使用 iPhone 4 读入图像文件时，如果有高像素文件，那么能够支持自动读入的话就会十分方便。为此，高像素图片可使用如下文件名设置。

文件名 @2x. 后缀名
文件名 @2x~ 设备 . 后缀名（兼容上述设备指定时）

UIKit 框架中使用类 UIImage 来处理图像。该类可以使用如下方法检索并加载主束内的图片文件，生成包含图片的实例。

```
+ (UIImage *) imageNamed: (NSString *) name
```

在 iPhone 4 执行应用的过程中，当给该方法指定了“beyond.png”文件名时，如果主束中存在 beyond@2x.png 文件，它们就会被自动读取。

（3）使用信息文件的键值字符串指定设备

为了和设备相匹配，可以详细设置信息文件中读入的信息。

例如，信息文件的键值 UIInterfaceOrientation 会在应用启动时指定接口适用的类型（表 16-4）。信息文件中记录的内容在各个设备上通常是共享的，如果只想在 iPad 中改变设置，可使用如下字符串作为键值。

```
UIInterfaceOrientation~ipad
```

像这样，在一般的键值字符串后加上后缀“~设备”，即可表示仅此设备适用。设备部分可以指定如下值。

iphone：表示 iPhone
ipod ：表示 iPod touch
ipad ：表示 iPad

关于 iOS 资源的详细内容，请参考“iOS Application Programming Guide”“iPad Programming Guide”“Resource Programming Guide”等参考文档。

16.1.6 通用二进制

通用二进制（universal binary）是苹果公司的专门术语，是为了能在多个操作系统不同的 CPU 上运行而生成的执行文件形式。在 PowerPC 的 Mac 以及 Intel 的 Mac 中可以运行 Mac OS X，iOS 平台下，也希望能适配各种处理器版本不同的 iPhone 手机 。原理很简单，只需将不同 CPU 所对应的机器指令保存在一个文件中，然后在执行时选择合适的来运行即可。同样，32 位及 64 位所对应的机器指令也可以包含在通用二进制文件中。该技术在拥有 4 个不同的运行平台的 OPENSTEP（见 1.1 节的 Column）中已经得到了实现，称为 MAB（Multi-Architecture Binary）或胖二进制（fat binary）。

此外，**通用应用**这个术语指的是仅用一个应用包就可以在多个不同的环境下运行的应用。为此，束内保存了面向不同环境的资源和设定信息。在很多情况下，执行文件就是通用二进制。

应用通常由 Xcode 构建，但也可以由命令行生成通用二进制。下面将简单说明一下其方法。

首先，生成某个架构专用的执行文件时，编译器使用 -arch 选项。例如，编译 pri.c 源文件来生成面向 Intel 32 位机器的执行文件时，可按如下方式。编译器使用什么样的机器都没有关系（“%”为命令提示符）。

```
% clang -arch i386 pri.c
```

通常会生成 a.out 执行文件，这里，通过为 lipo 命令指定 -info 选项，就可以查看它包含什么样的运行指令。如下所示。

```
% lipo -info a.out
Non-fat file: a.out is architecture: i386
```

生成同时面向 32 位及 64 位指令的执行文件时，重复指定 -arch 选项即可。而如果指定 8.4 节中介绍的 -m32 或 -m64 选项，就只能生成其中一方的代码指令。

```
% cc -arch i386 -arch x86_64 pri.c
% lipo -info a.out
Architectures in the fat file: a.out are: i386 x86_64
```

基本方法就是在编译时面向各架构分别指定编译选项。所以，源程序中如果有编译警告，指定的架构数就可能会反复显示。

在 Mac OS X 10.6 之前的版本中可以生成面向 PowerPC 的指令代码（架构指定为 ppc），而在 10.7 之后就不行了。因为 32 位代码指令中不可以使用现代运行库，使用了 ARC 等新功能的源程序不能编译，所以，在 Mac OS X 10.7 中，通用二进制的优势并没有显露出来。而且指定编译器架构 i386 或 x86_64 时，也可以分别利用 __i386__ 或 __x86_64__ 这样的宏定义。

iOS 平台的机器使用 ARM 架构的 CPU。因为编译时指定的架构是 armv6 或 armv7，所以在生成 iPhone 和 iPad 上执行的通用应用时，要同时指定 armv6 和 armv7（因为有兼容性，所以单指定 armv6 时也可以运行）。除此之外还有宏定义 __arm__。

上述查看执行文件时使用的命令 lipo，原本是被用来整理面向各个架构的执行代码并生成通用二进制的，相反也可以被用来抽取其中的一部分代码。在生成通用二进制的情况下，编译时可以只指定必要的 -arch 选项，此外也可以先生成面向各种架构的执行文件，然后再使用 lipo 命令对其加以整理。详情请参考在线手册。

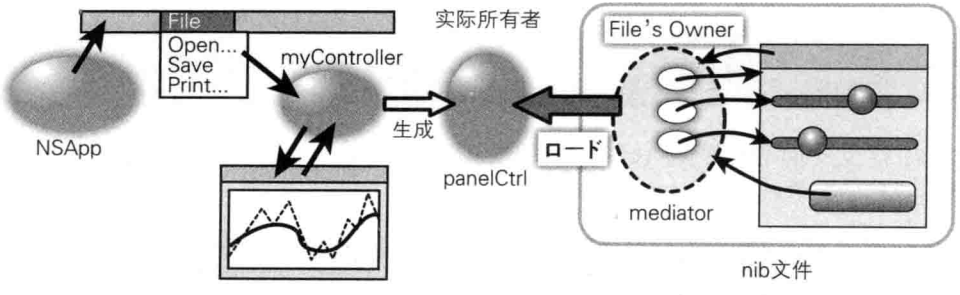
16.2 加载 nib 文件

16.2.1 nib 文件实例化

应用在执行时打开多个窗口，或者打开新面板时，有时会从保存 GUI 组件间关联及配置的 nib 文件中读取信息。下面将简要说明一下这个处理过程。因为 nib 文件的生成方法不在本书的研究范围内，所以在此不做说明。

请看图 16-2，假设图右侧的 nib 文件部分是设定选项参数的面板。这个对象或实例并不是在应用启动时进行实例化，而是在必要时才进行。

图 16-2 nib 文件的加载和所有者



nib 文件定义了 GUI 组件间的通信消息，其中必须有一个称为所有者的对象。图中名为 mediator 的对象就承担着该角色。所有者就是连接 nib 文件内的对象群和外部世界的桥梁。应用方面的咨询及

nib 文件内对象的访问基本上都是通过所有者来传达的。

图中，myController 对象会生成 panelCtrl，同时显示 nib 文件的加载。

加载 nib 文件后，其中定义的对象会被实例化。但是所有者不会被实例化。所有者使用的对象在加载 nib 文件前就存在。像这样，所有者可以根据与 nib 文件外对象的关系来设定，也可以确定 nib 文件中的对象，两者可以同时存在。

加载 nib 文件将对象实例化与在 13.2 节中说明的解档基本相同。nib 文件是对象图的归档，所有者的作用就相当于根对象。但是，nib 文件中也可以生成无指向的指针对象。这在内存管理上有时会出现问题。这一点在下面的说明也会提到。

16.2.2 在 Mac OS X 中加载 nib 文件

在 Mac OS X 中加载 nib 文件通常使用如下方法。

和其他资源一样，如果有面向语言的目录就从中查找。这是 NSBundle 的类方法，在 Application 框架中被定义为范畴。接口在 Application/NSNibLoading.h 中声明。

```
+ (BOOL) loadNibNamed: (NSString *) aNibName
                owner: (id) owner
```

在束内查找字符串指定的nib文件(不需要后缀名)，并将所有者对象owner实例化。如果失败就返回NO。

参数 owner 指定的对象必须与 nib 文件内设定的类相同。如图 16-2，将 myController 生成的对象 panelCtrl 指定为参数 owner 来加载 nib 文件。这样一来，nib 文件内的对象就会被实例化，panelCtrl 就开始作为 nib 文件内的 mediator 而发挥作用。

这里，首先详细说明内存的管理。

在使用引用计数的管理方式管理内存时，nib 文件内的所有（除了所有者）对象都会被引用计数置为 1 并被实例化。接着，构造对象间的引用关系，被其他对象持有的对象则通过autorelease来释放所有者关系。如果 nib 文件中存在没有被引用的对象，那么最终便只有该对象不会被释放而保留着，所以就需要使用其他方法来释放内存。

使用垃圾回收时，所有者强引用的对象因为可以访问所以没有任何问题。而如果 nib 文件中存在没有被引用的对象，那么该对象在实例化后就有被释放的危险。

16.2.3 在 iOS 中加载 nib 文件

iOS 中 NSBundle 使用下面的方法。和其他资源同样，如果存在面向所使用的语言的目录就从中查找。该方法用 UIKit 框架的范畴定义，接口在 UIKit/UINibLoading.h 中声明。

```
- (NSArray *) loadNibNamed: (NSString *) name
                     owner: (id) owner
                   options: (NSDictionary *) options
```

在束内查找字符串命名的nib文件(没有后缀名),并将所有者对象owner实例化。参数options中可以指定选项,而且一般都传入nil。在nib文件内,没有被其他对象包含的对象(顶级对象)将用数组保存并返回。

在 iOS 中,将 nib 文件中的所有对象(所有者除外)的引用计数都置为 1 并进行实例化。接着,构筑对象间的引用关系,被其他对象持有的对象则通过autorelease来释放所有者关系。使用键值编码的setValue:forKey:方法(第 20 章)来设定对象出口。需要明确出口所使用的实例变量的引用、存储方式时,请准备设置或声明属性。

nib 文件中如果存在没有被引用的对象,则将其保存在返回数组中。无论使用什么方法,我们都要确保内存释放。

16.2.4 nib 文件内的包含循环

使用 ARC 管理内存时,nib 文件内的对象之间容易形成包含循环,这点一定要注意避免。

基本上,nib 文件就是一个树形结构,其中,所有者是根节点,并和其他组件和对象相互关联。首先,所有者引用顶层对象(不能成为其他对象组件的对象)时使用强引用。窗口上配置的组件有层次关系,保持着树形构造。另一方面,程序新增的自定义对象,可以将这些对象作为出口自由引用。因此,**自定义对象包含的出口原则上应该通过弱引用才能使用**。

框架提供的类中,有些没有对应弱引用。由于这样的类会使用 __unsafe_unretained 修饰的实例变量或 assign 属性的特性,因此一定要注意不要有空指针。不管是否是弱引用,在释放对象前,都需要将引用其他对象的实例变量赋值为 nil,这点请注意。

16.2.5 nib 文件内对象的初始化

nib 文件内包含的对象如果实现了如下方法,在实例化后该方法就会运行。

- (void) **awakeFromNib**

从nib文件中读出,在完成实例化、出口及访问的连接后调用该方法。

该方法在 Application 及 UIKit 框架中被声明为非正式协议。头文件分别为 NSNibLoading.h 和 UINibLoading.h。因为是非正式协议,所以任何类都可以实现该方法并记录初始化流程。

16.2.6 启动应用

应用最先读入的 nib 文件中,保存着菜单等运行应用所需要的重要信息。这称为主 nib 文件,信息文件(表 16-2)中也有文件名。该文件在应用启动后运行回路(见 15.1 节)启动前被读入。

加载主 nib 文件以及启动运行回路可通过 Mac OS X 中的 NSApplication 及 iOS 中的 UIApplication 类来完成。NSApplication 类是 Application 框架的类,接口位于 AppKit/NSApplication.h

中。UIApplication 类是 UIKit 框架中的类，接口在 UIKit/UIApplication.h 中记述。

该实例在应用启动后仅被生成一个，除运行回路之外，还负责管理应用的各种资源，如菜单、窗口、应用的隐藏或终止等。

为了访问这个唯一的实例，NSApplication 类与 UIApplication 类都使用sharedApplication类方法。该方法是 15.2 节中也曾描述过的典型的单体设计模式。

```
+ (NSApplication *) sharedApplication （NSApplication 类）
+ (UIApplication *) sharedApplication （UIApplication 类）
```

返回应用类的唯一一个实例对象。

使用NSApplication时，如果实例不存在就生成实例，并执行必要的初始设置。

为了生成 Mac OS X 应用，在 Xcode 中选择“Cocoa 应用”后，如下 main 函数就会自动生成（Xcode4.2 版本）。

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

int main(int argc, char *argv[])
{
    return NSApplicationMain(argc,  (const char **) argv);
}
```

函数 NSApplicationMain() 为 Application 框架提供的函数，内容如下所示。这里，NSApp 为表示 NSApplication 实例的全局变量。

```
int NSApplicationMain(int argc, char *argv[]) {
    [NSApplication sharedApplication];                                    ❶
    if ( [NSBundle loadNibNamed:主nib名 owner:NSApp] )                    ❷
        [NSApp run];                                                      ❸
    exit(0);
}
```

在开始执行后，main 函数开始运行，❶ 生成 NSApplication 实例，❷ 加载主 nib 文件，❸ 启动运行回路。

另一方面，在 Xcode 中，选择面向 iPhone 或 iPad 的应用对象后，就会自动生成如下 main 函数。

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

int main(int argc, char *argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil,
                        NSStringFromClass([AppDelegate class]));
    }
}
```

函数 UIApplicationMain() 是 UIKit 框架提供的函数。第三个参数需要使用字符串指定 UIApplication 或其子类名。如果为 nil 则选择 UIApplication。第四个参数需要指定被作为 UIApplication 的委托（参考 15.2 节）使用的类名。AppDelegate 是同时生成的委托类。函数 UIApplicationMain() 的内部虽然被隐藏了，但基本上和 NSApplicationMain() 同样。

此外，应用中所固有的各种初始化设置，不需要放在 main 函数中，而是应该写到 NSApplication 和 UIApplication 的委托对象内。应用启动后，由于通知启动完成的消息会被送给委托对象，因此应在委托内部书写。而应用切换时或应用终止前想要执行的操作也可以写入。详情请参考 NSApplicationDelegate 协议或 UIApplicationDelegate 协议的参考文档。

16.3 iOS 的文件保存场所

16.3.1 主要目录及功能

在 iOS 中，为了安全起见，应用间的文件交换等会受到限制。为此，与个人电脑环境不同，文件保存的场所也被局限在了各应用所分配的特定场所中。该场所在应用安装时被决定，称为应用的 home 目录。应用程序包及生成的文件被保存在 home 目录下，删除应用时会将其一并删除。

应用的 home 目录下设置着主要目录。各目录具有不同的功能，文件能够长期保存的场所、与 iTunes 同步的场所，以及备份场所的路径名都是固定的。

在这些文件夹中创建的文件，必须由应用程序来管理。不需要的文件如果不被删除，设备的内存容量就会变得紧迫。

- **home/ 应用名 .app/**

  应用束本身存放在这里。因为有署名所以不能改变。

- **home/Documents/**

  应用生成并保存文件的场所。与 iTunes 连接时启动备份。而且，将信息文件的键值 UIFileSharingEnabled 设定为 YES 后，就可以通过 iTunes 使文件和电脑同步。

- **home/Library/Preferences/**

  应用设定（16.4 节中介绍的用户默认）被写入。与 iTunes 连接时被备份。

- **home/Library/Caches/**

  能够将应用临时使用的信息作为文件保存在这里。在下一次启动时加载可以利用的操作历史或操作过程等信息。但是，在设备还原等时可能会随之消失。iTunes 不会备份。

- **home/tmp/**

  能够将应用临时使用的信息作为文件保存在这里。在应用不运行期间可能会被系统释放。

16.3.2 获取目录路径

在获取上述目录中的 Documents 和 Cashes 的路径时，推荐使用 Foundation 框架的函数 NSSearchPathForDirectoriesInDomains()。函数和使用的参数在 Foundation/NSPathUtilities.h 中声明。

```
NSArray * NSSearchPathForDirectoriesInDomains(
    NSSearchPathDirectory  directory,
    NSSearchPathDomainMask domainMask,
    BOOL                   expandTilde
)
```

该函数本来是 Mac OS X 专用的，形参参数的组合在 iOS 中可能没有意义。而且，由于返回数组中只保存有一个路径，所以请使用`objectAtIndex:`方法将首部的元素取出使用。

第一个形参中指定目录的种类。为了获得 Documents 和 Caches 的路径，请分别指定参数 NSDocumentDirectory 和 NSCachesDirectory。无论哪种情况，都请指定第二个形参为 NSUserDomainMask，第三个形参为 YES。

而且，tmp 目录的路径可使用下面的函数获取。

```
NSString *NSTemporaryDirectory(void)
```

然而，可以使用这些函数取得的目录路径并不一定存在，必要时需要由程序创建。为了查找路径是否存在或创建路径，可以使用 Foundation 框架的 NSFileManager 类。使用类方法 `defaultManager` 获得唯一的实例，并使该实例负责指定路径相关的操作，这里不再详细讨论，。

16.4 用户默认

16.4.1 保存设定值

易用的应用会保存曾经的设定值、配置窗体等，并在下次执行时直接使用之前的设定。这样的设定值包括环境设定窗体中的设置，以及运行中自动保存的用户选择的内容等。

在 Cocoa 应用中，为了保存这些设定而使用了一种通用结构——**用户默认**（user defaults），或者称为默认数据库。类 NSUserDefaults 提供了访问用户默认所需的接口。

用户默认会访问与字典对象相同的字符串键值这样的信息。Mac OS X 中，被记录的信息被保存在 ~/Library/Preferences/ 目录下的属性文件中。保存用户默认的文件名为应用名加后缀名“plist”。

iOS 中，用户默认被保存在应用的根目录下的 Library/Preferences/ 目录中。

iOS 中可以通过应用来改变设定值，也可以从系统工具“设定”中构建可设定的应用。为此，应用需要包含 Settings 窗体。详情请参考“iOS Application Programming Guide”等。

16.4.2 默认域

用户默认可以被分为多个组考虑，这些组称为域或者默认域（defaults domain）。在域中，有的将内容保存在文件中并在之后继续使用，而有的则仅在应用启动时存在。下面将详细说明。

（1）应用域

使用属性列表管理的应用固有的设定值的集合。该设定值的集合称为应用域，域名中使用应用标识名（见 16.1 节）。

（2）全局域

使用者使用账户设定的各个应用共享的设定值。例如，所使用的语言、是否附加文件名后缀，以及当文字大于多少时将其平滑表示等，都涉及多方面的因素。域名为 NSGlobalDomain。

该域即使在 iOS 中也有效，可以获得设定的语言或键盘信息。

（3）形参域

Mac OS X 的 Cocoa 应用可以从终端命令行启动。例如，虽然文本编辑器在 /Applications/TextEdit.app 的路径下，但用命令直接指定应用束内的执行文件也可以启动。

```
% /Applications/TextEdit.app/Contents/MacOS/TextEdit
```

此时，应用域或全局域中包含的设定值可以作为参数来指定。例如，全局域中名为 AppleAntiAliasingThreshold 的键值表示将大于某一数值的文字平滑显示。在启动时可按照如下方式使用（因为太长所以显示为两行，实际上是一行）。

```
% /Applications/TextEdit.app/Contents/MacOS/TextEdit
      -AppleAntiAliasingThreshold 15
```

于是，在该文本编辑器中，15 点以下的文字都不会被平滑表示。这是因为全局域的设定值是通过参数指定来临时传入的，所以通常在重新启动时就会恢复原值。

像这样，参数指定的设定值组就称为参数域（argument domain）。域名用 NSArgumentDomain 表示，且不能被保存在文件中。iOS 中不使用该域。

（4）语言域

指定使用语言时，会生成该语言名（例如 ja）为域名的临时域（不保存文件）。

（5）登录域

用户尚未设定应用的各种设定值时使用的默认值集合。域名为 NSRegistrationDomain，且不被保存在文件中。

当应用使用 NSUserDefaults 类从键值中查找对应的设定值时，会按如下顺序查找上述 5 个域，并使用最先找到的值。

1. 参数域

2. 应用域（保存在文件中）
3. 全局域（保存在文件中）
4. 语言域
5. 登录域

如果各种设定值都需要默认值，可以将键和值的集合以字典方式组织，并在应用初始化时在登录域中登录。因为登录域查找在最后执行，因此，当其他域都不能登录时，就只能使用默认值。

16.4.3 查找用户默认的工具

Mac OS X 平台下，用户默认除了在各个应用中使用外，还可以通过直接编辑属性列表，或使用命令行工具 defaults 来进行访问。只在终端上输入 defaults 就可以看到使用方法。也可以使用 man 命令查看帮助手册。

通过下面的方式可以全部显示某个应用域的内容。域名为应用的标识名。这里指定“-globalDomain”或 NSGlobalDomain，就能显示全局域的内容。

```
% defaults read 域名
```

想要查看某个键的值时可通过如下方式。

```
% defaults read 域名 键
```

不使用 read 而使用 write 或 delete 时，虽然可以写入或修改，但如果胡乱修改用户默认，就会导致应用无法正常运行等问题，所以一定要注意。而如果发生了这样的状况，就要删除应用域对应的文件并重新启动应用。

16.4.4 NSUserDefaults 概要

下面将简要说明一下为了访问用户默认而提供的 NSUserDefaults 类。类接口在 Foundation/NSUserDefaults.h 中记述。全局域、参数域、登录域的域名字符串分别为 NSGlobalDomain、NSArgumentDomain、NSRegistrationDomain 变量。

（1）取得实例对象

NSUserDefaults 只会生成一个实例并使用。可以使用下面的类方法来取得该实例对象。

```
+ (NSUserDefaults *) standardUserDefaults
```

最初调用时将实例初始化后返回。一般情况下，自第2次调用起，会返回同一个实例。

（2）取出键对应的值

要取出键对应的值，可在 NSUserDefaults 实例中使用如下方法。如前所述，键值是通过在多个域中查找而得到的。

- (id) **objectForKey:** (NSString *) defaultName

取出键值对象。
返回值为属性列表对象，即字符串、数据、日期、数组或字典对象。没有键值时返回nil。

- (NSString *) **stringForKey:** (NSString *) defaultName

将键值作为字符串对象取出。取出的对象不是字符串或没有键值时返回nil。
同样，取出数据对象、数组对象、字典对象、URL时，可以分别使用下面的方法。
dataForKey:、arrayForKey:、dictionaryForKey:、URLForkey:

- (NSArray *) **stringArrayForKey:** (NSString *) defaultName

取出包含字符串对象的数组对象。取得的结果不是字符串对象以及不存在键值时返回nil。

- (NSInteger) **integerForKey:** (NSString *) defaultName

取出键对应的对象，并返回其整数值。不存在值时返回0。
同样，取出对象后返回实数值、双精度值、布尔值的方法分别是floatForKey:、doubleForKey:以及boolForKey:。

(3)指定键并设置或删除值

在应用域中设置键值。

- (void) **setObject:** (id) value
forKey: (NSString *) defaultName

在应用域中设置键对象。value如果不是属性列表可以处理的对象，则不能保证该处理的正确性。

- (void) **setURL:** (NSURL *) url
forKey: (NSString *) defaultName

在应用域中设置键对应的NSURL实例。

- (void) **setInteger:** (NSInteger) value
forKey: (NSString *) defaultName

在应用域中设置参数的整数值。
同样，设定实数值、双精度值、布尔值的方法分别是setFloat:forKey:、setDouble:forKey:以及setBool:forKey:。

- (void) **removeObjectForKey:** (NSString *) defaultName

从应用域中删除键对应的入口。

(4)设定默认值

- (void) **registerDefaults:** (NSDictionary *) dictionary

在登录域中添加保存默认值的字典对象。指定的键值没有在应用域等其他域中登录的情况下，将登陆域信息作为默认值使用。

(5)取出域内容

- (NSDictionary *) **dictionaryRepresentation**

将所有的域内容作为字典对象取出。

(6)在文件中显示设定内容

- (BOOL) **synchronize**

  将域内容写入到对应的属性列表文件中。因为该方法通常为自动调用，所以除了在应用终止时需要在文件中显示调用外，其他情况下都不用调用。出于某种原因不能写入文件时，返回NO。

16.5 应用的本地化

16.5.1 消息的本地化

使用束结构可以将程序显示的消息与各种语言相对应。Objective-C 的源程序中，没有必要输入日语字符串。

为此，首先需要生成 Localizable.strings 文件，并将其保存到各语言对应的目录（语言 .lprog）中。文件名也可以为别的名字，不特别指定时使用 Localizable.strings。

Localizable.strings 文件是将键字符串及键值字符串用等号连接成一对，并在结尾设置分号。无论是键还是值，一般都用“”括起来。也可以省略值，不过这时也要同时删除等号。值即使被省略，键本身也仍然会被当成入口值。此外，也可以用 /* … */ 的形式来书写命令。

键字符串一般使用 ASCII 码范围内的字符来保存消息（一般为英语）。值字符串中记录着使用了目标语言（日语等）字符集的消息。文件一定要使用 UTF-16 编码。仅使用 ASCII 码范围内的字符就可以记录全部时（英语），保持 ASCII 编码方式即可。

下面来看一个 Localizable.strings 文件的简单用例。

面向英语的 Localizable.strings 文件

```
/* File does not exist */
"File does not exist";
/* Do you want ? */
"Do you want ?" = "Do you want to do it, Really ?";
/* OK */
"OK";
```

面向日语的 Localizable.strings 文件（UTF16 编码）

```
/* File does not exist */
"File does not exist" = "文件不存在";
/* Do you want ? */
"Do you want ?" = "当真可以执行么?";
/* OK */
"OK";
```

应用本身的束（主束）的 NSBundle 对象，通过使用下面的消息，可以取得键值字符串。

```
- (NSString *) localizedStringForKey: (NSString *) key
                               value: (NSString *) value
                               table: (NSString *) tableName
```

在束内查找参数tableName指定的名字及含有后缀名“.strings”的文件，返回参数key指定的键值字符串。当找不到对应的键时，参数value指定返回值。
tableName为nil或@""时，使用Localizable.strings。

为了更方便地获得本地化的字符串，定义了使用上述方法的宏（头文件 Foundation/NSBundle.h）。从主束中取出字符串时，使用该宏十分便利。

NSLocalizedString(key, comment)

在主束内查找文件Localizable.strings，返回参数key指定的键值字符串。Objective-C方式下不使用comment。字符串对应的注释可以自由书写。

NSLocalizedStringFromTable(key, tbl, comment)

在主束内查找参数tbl指定的文件，返回参数key指定的键值字符串。

上述 Localizable.strings 的例子中，“File does not exist” 对应的本地化字符串可以通过如下方式获得。

```
NSString *msg = NSLocalizedString(@"File does not exist", NO_FILE);
```

而且，这些在源文件中写的宏，可以使用命令 genstrings 生成 Localizable.strings。在命令行中输入 genstrings 且不加参数时，将显示帮助信息。

16.5.2 本地化指针

使用日语与英语不同，显示消息不能用日语编码。即使只构建日语应用时也需要有 Localizable.strings。该文件以及定义了 GUI 的 nib 文件、帮助等，包含日语的文件全都应该被保存在 ja.lproj 目录下。

将英语、法语等多语言的应用本地化为日语是比较容易的。基本上只需生成各国语言子目录中的 Localizable.strings、nib 文件及各种资源的日语版，并追加 ja.lproj 就可以了。

如果仅仅是英语应用，则不需要担心本地化。包括英语在内，面向各语言的子目录和 Localizable.strings 基本上都不是必须的。但是，这样的应用被认为没有被国际化（internationalize）。相反，将这样的应用在事后进行国际化或本地化是非常困难的。因为消息是写在源程序里面的，所以必须要使用 NSLocalizedString() 等来修改。

虽然与本地化相关的详细设定和目录结构的生成可交给 Xcode 来完成，但我们需要了解本地化设定和实际生成的应用结构之间是如何对应的。

专栏：本地化应用名 COLUMN

Mac 的 Finder 及 iPhone、iPad 的本地界面中显示的应用名可以设为各国语言。例如，这里我们来看一下如何将“SpaSearch.app”改为“温泉搜索（临时版）.app”。

首先，信息文件（Info.plist）内应用名使用了应用束的名字。此时，需要书写 CFBundleDisplayName 和 CFBundleName 键（参考表 16-1）。

```
<key>CFBundleName</key>
<string>SpaSearch</string>
<key>CFBundleDisplayName</key>
<string>SpaSearch</string>
```

然后，面向各语言的子目录中生成 InfoPlist.strings 文件。该文件形式上与 Localizable.strings 相同，并用 Unicode 编码。子目录 Ja.lproj（或 ja.lrog）下的 InfoPlist.strings 编辑如下。

```
CFBundleDisplayName = "温泉搜索（临时版）";
CFBundleName = "温泉搜索";
```

Finder 的表示使用 CFBundleDisplayName 字符串。CFBundleName 被用于应用启动时菜单栏的表示，需指定短于 16 个字符（日语指定为 8 个字符）的字符串。

16.5.3 本地化

本地化（locale）就是编辑文本时的各种习惯或使用单位等的集合体。例如，表示日期的方式、作为小数点使用的字符（句号还是逗号）等，根据国家和语言的不同是存在差异的。为了生成在任何语言环境下都能通用的软件，就需要事先准备好这些字体，然后再在软件操作时选择适合的显示。

为了表示本地化，Cocoa 环境中提供了 NSLocale 类，当创建可以应对各种语言或地域习惯的应用时，就可以使用该类。用户根据使用环境而选择的本地化信息称为当前本地化，应用使用当前本地化信息进行操作。

例如，NSString 中有`localizedStringWithFormat:`类方法，它与`stringWithFormat:`同样可以指定字符串格式。假设使用该方法进行如下记述。

```
id date = [NSDate date]; // 返回表示当前时间的对象
id str = [NSString stringWithFormat:@"Date=%@", date];
printf("%s\n", [str UTF8String]);
str = [NSString localizedStringWithFormat:@"Date=%@", date];
printf("%s\n", [str UTF8String]);
```

在日语环境下执行时，结果显示如下。

```
Date=2010-09-10 03:05:50 +0900
Date=2010年9月10日星期五 3点05分50秒 JST
```

同样的方法有如下初始化方式。

```
- (id) initWithFormat: (NSString *) format
                locale: (NSDictionary *) dictionary, ...
```

参数 dictionary 可按如下方式指定当前本地信息。

```
id default = [NSUserDefaults standardUserDefaults];
id dic = [default dictionaryRepresentation];
str = [[NSString alloc] initWithFormat:@"Date=%@" locale:dic, date];
```

用户默认信息保存在字典变量 dic 中，其中包含了语言相关的各种设定。这其实就是本地化的功能。此时，变量 str 中存储的字符串与使用localizedStringWithFormat:时是同样的。NSArray、NSDictionary 等包含descriptionWithLocale:方法。该方法的参数也可以用同样的方式指定。

本地化相关的详细信息，请见参考文档“Locals Programming Guide”等。

专栏：消息内的语序 COLUMN

用日语表示英语消息时，因为语序不同所以会费些功夫。此时，可以利用 printf() 函数或 NSString 方法等的格式可以指定的特殊书写方式。

通常，格式字符串内的格式符（%s 等）可以与后面的参数依次对应。通过在格式符“%”后指定“数字”+“$”，就可以将数字指定顺序的参数，在该位置处发生变换。例如，下面的 printf 就会输出“Good: 100”。

```
printf("%2$s: %1$d\n", 100, "Good");
```

使用面向日语的 Localizable.strings 文件按照如下方式书写时，就可以将该字符串作为格式字符串使用，并使用日语的语序（像“文件 0012.txt 正在处理”那样）来表示。虽然不是万能的，但是灵活性很高。

```
"%@ file %04d.txt" = "文件 %2$04d.txt %1$@";
```

16.6 模块的动态加载

此章节将介绍程序的模块在执行中是如何实现加载的，**该功能无法在iOS中使用**。仅在 Mac OS X 中可以使用。

16.6.1 可加载束

程序内使用的类或范畴中，将当前不使用或可能不使用的模块保存在执行文件之外的其他文件中，并在必要时动态加载，这样就可以加速程序的启动，或者生成限定了功能的程序版本。

而且，通过事先决定要加载的模块形式，并生成有别于程序的模块，就可以为程序添加各种各样的功能，这就是**插件**（plug-in）。

在执行文件外创建并动态加载的模块通常会以**可加载束**（loadable bundle）的格式被保存在文件中。可加载束有与应用相同的目录结构。一个可加载束中保存着多个类和范畴的代码。可加载束可以作为资源保存在应用束内，也可以保存在外部。然而，加载方和被加载方仅有一个时都不能使用垃圾回收。如果使用引用计数管理方式，可以采用手动方式和与 ARC 混合的方式（见 6.1 节）。

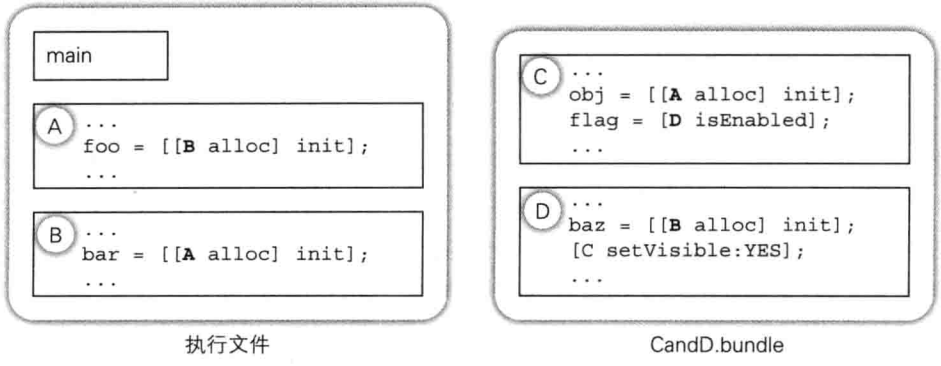
比起手动创建，可加载束一般采用开发工具 Xcode 来创建。不管哪种方式，都需要说明使用可加载束的应用的概况。

16.6.2 使用可加载束的程序

假设存在一个由类 A、B、C、D 以及主函数组成的应用。将主函数和类 A、B 链接到执行文件中，将类 C 和类 D 当作 1 个可加载束（图 16-3）。

在执行文件一方，类 A、B 可以自由使用，但是使用类 C、D 的代码不能写入到程序内。具体来说，代码中不能有显示类名的类对象。但是，用类 C、D 定义的消息可以自由发送。这是因为，在引用类对象时，如果该类代码是必要的，那么消息选择器所对应的具体方法的确定就被推迟到了执行时进行。

图 16-3 可加载束和类的关系



在类 C、D 方面，不仅可以使用类 C、D，也可以使用类 A、B 的代码。因为即使加载的代码中包含未解决符号（类名或函数名），但只要在加载时能解决也就可以执行。

简便起见，假设类 C、D 的可加载束被保存为应用束的源文件。使用 CandD.bundle 作为其名字。在执行文件方加载该可加载束来取得类 C 的类对象时，可以先取得 CandD.bundle 对应的束对象，然

后对该对象适用classNamed：方法。

```
- (Class) classNamed: (NSString *) className
```

返回一个类对象，它包含参数className指定的类名。束的执行代码尚未加载时，该方法会加载这个代码。不能加载指定的类时，返回Nil。

加载某个类代码时，它的可加载束内包含的其他代码也会同时被加载到内存中。所以，仅加载类C，内存中就会存在类D的代码。即使类C在内部使用了类D，也没有必要显式加载类D。

当知道所加载的模块内的类名时，可以使用下面的函数指定。

```
Class NSClassFromString (NSString *aClassName);
```

参数aClassName指定了类名，返回持有该类名的类对象。类尚未加载时返回Nil。

类或范畴被动态加载时，如果类中定义了方法load，那么就可以自动执行。据此，加载后就可以执行任何初始化设置，这就是NSObject声明的方法。由于在某个类的load方法启动前，该超类的load方法就已经被调用，所以只需要写和该类相关的动作代码。

```
+ (void) load
```

下面，举例说明从执行文件方面加载可加载束的代码。可以看出，除字符串外，都没有使用类名C。

```
static Class classC = Nil;

if (classC == Nil) {
    NSBundle *bundle, *module;
    NSString *path;
    bundle = [NSBundle mainBundle];
    path = [bundle pathForResource:@"CandD" ofType:@"bundle"];
    module = [NSBundle bundleWithPath: path];
    if (module == nil) /* ERROR */
        return nil;
    classC = [module classNamed:@"C"];
}
return [[[classC alloc] init] autorelease];
```

可加载束内有时会包含图像、文本及本地化的文件等。为了使用这些文件，被动态加载的类需要知道自己被哪个束所包含。为此，可以使用如下的类方法。

```
+ (NSBundle *) bundleForClass: (Class) aClass
```

参数aClass使用被动态加载的类来返回包含该类的可加载束所对应的束。

16.6.3 插件概述

插件就是在程序执行过程中被动态追加的提供附加功能的模块。例如，为了能够读入不能被标准功能处理的数据文件，就可以通过插件来实现。此时，假设存在可读入新格式文件的类，并准备了使用这个类来处理文件的结构。那么只要能对应各种格式的文件开发出多种插件，就可以在不改

变程序本身的情况下增加新的功能。

在 Objective-C 编程中，插件也可以作为可加载束来实现。与程序中预先准备的代码不同，程序执行时并不一定需要插件。但是，一个程序可以同时加载使用多个插件。插件应该被定义为独立性高的模块，这样，无论是谁，只要遵从此特性，就都能生成新插件。

插件与构成程序的可加载束不同，不能提前知道代码中包含的类名。为此，插件要使用如下方法加载代码。

- (Class) **principalClass**

返回可加载束的主要类(principal class)的类对象。必要时加载代码，不能加载时返回 Nil。

主要类是插件中包含的一个类，在可加载束的信息文件 (Info.plist) 中为键 NSPrincipalClass (表 16-2) 的值，也是链接代码时最先出现的类。由于使用该类的插件已经开始执行，所以必须要决定哪个类是主要类。

Mac OS X 的屏幕储存器在 /System/Library/Screen Savers/ 或 ~/Library/Screen Savers/ 目录中，并被保存成后缀名为“saver”的文件，这实际上就是可加载束。Mac OS X 中有 ScreenSaver 框架，将其中定义的类 ScreenSaverView 的子类作为主要类生成可加载束后，就可以作为屏幕储存器进行工作。这就是典型的插件例子。

专栏：沙盒 (App Sandbox)

COLUMN

App Store 中出售、发布的 Mac OS X 应用为了强化其安全性，必须使用 App SandBox 来实现 (2012 年 3 月开始)。沙盒就是只能在预先设定好的权限条件下运行的应用的执行环境。

在开发使用 App Sandbox 的应用时，会在 Xcode 中设置它应该具有什么资格 (entitlement)。例如，是否可以读写文件、是否可以访问网络、是否可以使用系统摄像头等。由于这些被放入了代码署名中，因此即使受到恶意影响，也不能执行预先设定之外的动作。而且，与 iOS 相同，应用可以读写文件的场所也受到了限定。

更多详细内容请参考“App Sandbox Design Guide”等文档。

第17章

实例：简单图像视图

本章将通过使用 Cocoa GUI 的程序示例来介绍 GUI 组件和自定义对象间的协作关系。这也是应用束和可加载束、委托、通知、撤销、本地化等目前为止介绍的内容的实例。

17.1 Application 框架和 Interface Builder

下面介绍一个使用 Application 框架的 GUI 的程序。在 Cocoa 环境中创建 GUI 时，一般使用 Xcode 和一部分接口构建工具 Interface Builder，而这里我们将介绍的是如何从源文件中创建。

Interface Builder 被用来生成 GUI 外观及初始设置、添加 GUI 组件（窗体或按钮、滑块等）和程序生成实例对象间的关系等。所以，在使用 Interface Builder 时，编程者只需要集中精力进行控制 GUI 对象的编程就行了。

然而，Interface Builder 虽然使用方便，但也有人质疑说编程学习者很难理解 Interface Builder 实际上到底是做什么的。因此，这里我们首先会将所有的设定都使用文件记录，之后再说明使用 Interface Builder 后哪些部分可以不用记录。

Application 框架中的各个类以及 Interface Builder 的说明不是我们介绍的重点，因此这里不做详细解释。不过，Interface Builder 需要根据很多指定的选项和设定的链接来执行，这一点还希望大家能够理解。

但是，由于程序使用 ARC 类进行内存管理，并使用了对象的弱引用，因此在 Mac OS X 10.6 及以前的开发环境中是不能编译的。

17.2 程序概况

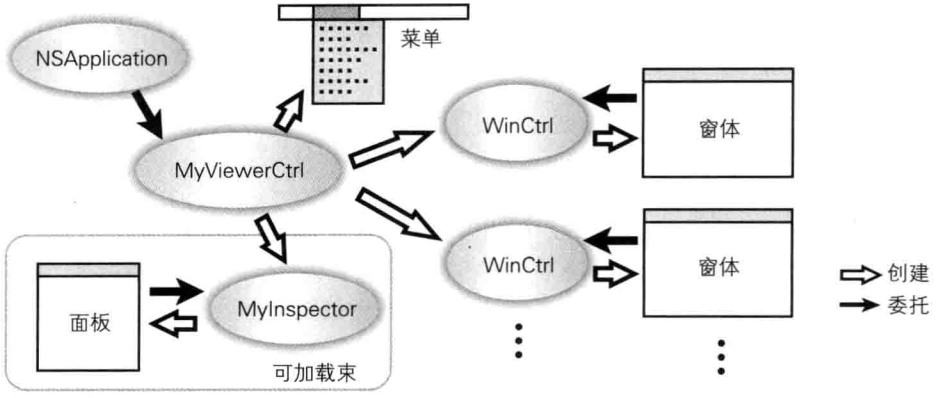
现在尝试创建简单的显示图像用的应用，命名为 MyViewer。MyViewer 将打开窗中指定的图像文件显示在新窗体里。可同时打开多个图像。也可以有简单的检查面板，并将图像缩小到 1/2 大小。

下面公开这些代码，省略其中一些冗长繁琐的部分。

17.2.1 对象间的关系

图 17-1 中显示了构成应用的主要对象间的关系。白色空箭头指向的目标对象是由源对象创建的，黑箭头指向的目标对象是源对象的委托。

图 17-1 主要对象间的关系



NSApplication 类接口就是代表应用自身的对象，管理着运行回路。该对象的委托中使用了 MyViewerCtrl 类的实例。该对象不仅处理发送给委托的消息或通知，还会处理菜单发来的消息，承担了应用的核心角色。

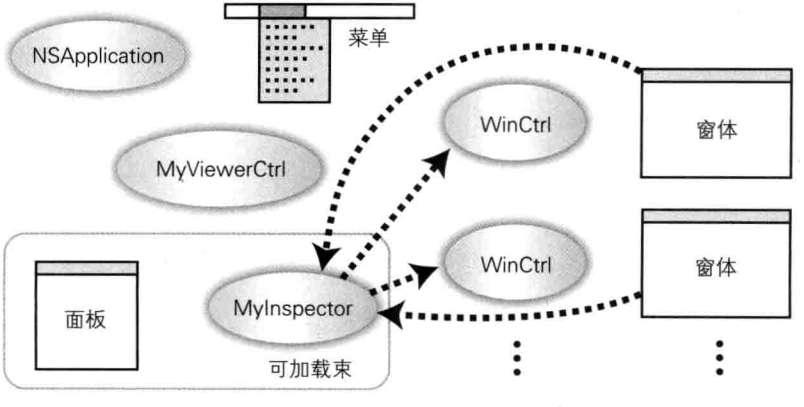
每次需要显示新图像时，MyViewerCtrl 都会创建 WinCtrl 类的实例，并生成多个接口。这些接口负责创建显示用的窗口和显示图像等具体工作，而且也是显示用的窗体的委托。

17.2.2 通知

应用中也包含了简易的检查面板。检查面板中显示当前主窗体（最顶层表示的窗体）中正在显示的图像的名字和尺寸。创建并管理检查面板可通过类 MyInspector 实例。该实例只能有一个，而且也是显示面板的委托。

在主窗体改变时，该应用会使用通知功能将这一消息传递给检查面板（图 17-2）。图中箭头显示了通知的结果，即信息来自哪个对象以及被哪个对象所使用。各窗体成为主窗体时或被关闭时，都会发送相应的通知。MyInspector 类接口从这些通知中了解到主窗口的切换，进而更改显示的面板。

图 17-2 通过通知传送信息



检查面板中有两个按键，其中一个表示将当前主窗体中显示的图像缩小到原图像的 1/2，另一个表示将现在显示的所有图像都缩小到原图像的 1/2。为此，MyInspector 类的实例会自己发送通知。WinCtrl 类的实例接收到该通知后，会将各自管理的图像缩小。

17.2.3 撤销和重做

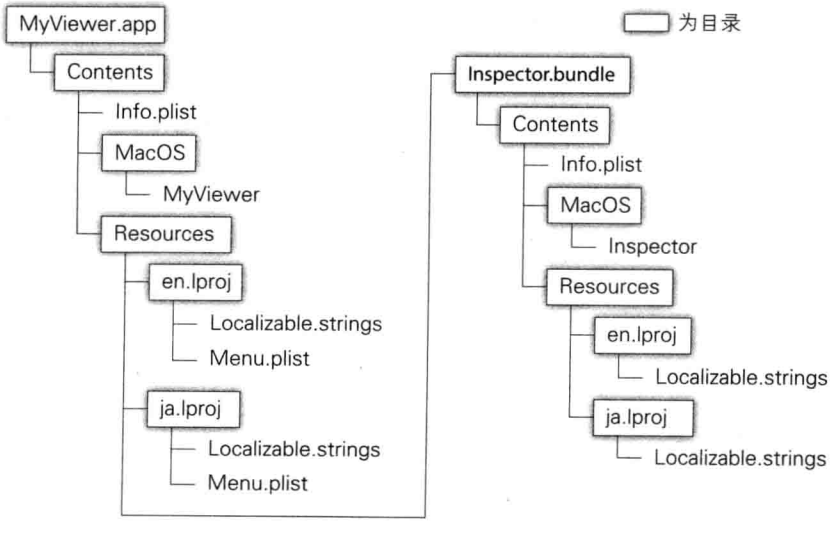
点击菜单上的“取消”或者使用命令键与“Z”的组合就可以实现撤销。撤销被应用于反应链的快速反应。管理各个窗体的 WinCtrl 类的实例都各自拥有撤销管理器，最顶层的窗体接收撤销消息并进行处理。取消操作仅包含图像缩小和文本编辑，同样也可以进行重做操作。

17.2.4 可加载束和本地化

管理检查面板的 MyInspector 类构成可加载束（图 17-2）。检查面板在首次使用时会被动态加载。而且该应用中还进行了本地化，即便是日语也可以表示消息。

图 17-3 为应用包的结构。可加载束在应用束内，因为它自己持有本地化用的文件，所以结构有些复杂。

图 17-3 应用包的内部结构



17.2.5 用户默认

菜单中有“自动缩小”项，选中时会将界面中不能容纳的大图像缩小显示。该选项的有无在用户默认中设定。而且，检查面板的位置和大小也保存在用户默认中。

17.3 编程介绍

17.3.1 main 函数和 MyViewerCtrl 类

下面开始介绍 MyViewer 的源程序。其中也会简要说明 Application 框架类，详细介绍请参考各类的参考文档。

首先是 main 函数。处理内容在列表内的注释中说明。一般在使用 Xcode 创建应用时，会自动创建 main 函数，所以不需要自己来写。也可以使用 Interface Builder 来设定菜单和委托。

▶ 代码清单 17-1 MyViewer 的 main 函数 (main.m)

```
#import <Cocoa/Cocoa.h>
#import "MyViewerCtrl.h"

int main(int argc, const char *argv[]) {
    @autoreleasepool {
        MyViewerCtrl *controller = [MyViewerCtrl sharedController];
        NSApplication *app = [NSApplication sharedApplication];
        [app setDelegate:controller];                    // 设置委托
        [app run];                                       // 启动运行回路
    }
    return 0;
}
```

下面介绍 MyViewerCtrl 类。因为该类将成为 NSApplication 的委托，所以它采用了 NSApplicationDelegate 协议。

MyViewerCtrl 类实例只需一个即可，并通过单体 (singleton) (见 5.2 节) 模式定义。应用启动后，该实例接收到委托消息后便执行创建菜单等操作。使用 Menu.plist 文件中用 ASCII 码属性列表记录的信息来创建菜单，但因为过程太繁琐所以这里省略相关说明。使用 Interface Builder 时，菜单和 GUI 组件的创建由 nib 文件进行，因此不需要执行程序那样的处理。

实例变量 autoResizeItem 表示是否选择自动缩小的菜单项。为了避开包含循环，该变量使用弱引用。

应用启动后就会处理菜单对应的点击操作。使用方法`openFile:`开启打开面板并选择文件，进而生成 WinCtrl 实例。这里使用了块对象。还有一点需要注意的是，选择的文件应作为 NSURL 实例传入。生成的 WinCtrl 实例被保存在 MyViewerCtrl 内的数组中，窗口关闭时就会被从数组中删除。如果不这样做，就不能保留 WinCtrl 实例，该实例在窗体显示后就会被直接释放。保存到数组及从数组中删除这些操作都由 WinCtrl 实例自动进行。

在方法`activateInspector:`中读入可加载束并显示检查面板。

终止时要确保已将用户默认信息写入到文件。

代码清单 17-2 MyViewerCtrl 接口部分（MyViewerCtrl.h）

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

@interface MyViewerCtrl: NSObject <NSApplicationDelegate>
+ (id)sharedController;                  // 返回共享实例
- (void)setupMainMenu;                   // 准备菜单
- (void)openFile:(id)sender;             // 打开文件
- (void)addWinCtrl:(id)obj;              // 添加 WinCtrl 类实例
- (void)removeWinCtrl:(id)obj;           // 删除 WinCtrl 类实例
- (void)activateInspector:(id)sender;    // 打开简介
- (void)toggleAutoResize:(id)sender;     // 切换是否自动缩小
@end
```

代码清单 17-3 MyViewerCtrl 实现部分（MyViewerCtrl.m）

```
#import "MyViewerCtrl.h"
#import "WinCtrl.h"
#import "MyInspector.h"

static NSString *AutoResizeKey = @"AutoResize";
    // 将是否自动缩小写入用户默认时的键
static id sharedCntr = nil;     // 保存唯一一个共享实例

@implementation MyViewerCtrl
{   // 因为是依赖实现的实例变量，所以将其设为私有
    __weak id autoResizeItem;  // 指向自动缩小的菜单项
    NSMutableArray *ctrlPool;  // 保存 WinCtrl 的实例
}

+ (id)sharedController {  // 单体模式
    if (sharedCntr == nil)
        sharedCntr = [[[self class] alloc] init];
    return sharedCntr;
}

/* 中略（构建菜单的本地方法）*/

- (void)setupMainMenu  { /* 此处由于太过复杂故省略 */ }
/* 在本地化的资源 Menu.plist 内用属性列表格式书写菜单内容。
   该方法会读入该信息，然后构建菜单
   点击菜单时的消息接收者中，“打开文件”“简介”“自动缩小”
   三个被设置为 self，其他被设置为快速反应 */

- (void)openFile:(id)sender  // 显示打开面板
{
    NSOpenPanel *oPanel = [NSOpenPanel openPanel];  // 打开面板
    [oPanel setAllowsMultipleSelection:YES];  // 可以选择多个文件
    [oPanel setAllowedFileTypes:[NSImage imageFileTypes]];
    // 显示打开面板。面板中只显示可以显示的图像文件
    [oPanel beginWithCompletionHandler: ^(NSInteger result) {
        if (result == NSFileHandlingPanelOKButton) {
            for (NSURL *aFile in [oPanel URLs])
                (void)[[WinCtrl alloc] initWithURL:aFile];
        }
```

```
    }];             // 使用打开面板按下 OK 键后的处理
}                   // 分别为多个 URL 生成 WinCtrl 实例

// 添加或删除 WinCtrl 实例
- (void)addWinCtrl:(id)obj { [ctrlPool addObject:obj]; }
- (void)removeWinCtrl:(id)obj { [ctrlPool removeObject:obj]; }

- (void)activateInspector:(id)sender     // 显示简介
{
    static Class inspectorClass = Nil;  // 指向加载的类
    NSBundle *bundle;
    NSString *path;

    if (inspectorClass == Nil) { // 尚未被加载时
        path = [[NSBundle mainBundle]
                    pathForResource:@"Inspector" ofType:@"bundle"];
        // 从主束内获得可加载束
        if ((bundle = [NSBundle bundleWithPath: path]) == nil)
            return; // ERROR!
        inspectorClass = [bundle classNamed:@"MyInspector"];
    }
    [[inspectorClass sharedInstance] activate];
}

- (void)toggleAutoResize:(id)sender  // 切换菜单的复选项
{
    BOOL newState = ([sender state] != NSOnState); // 否定现在的状态
    // 由 WinCtrl 类来管理是否自动缩小
    [WinCtrl setAutoResize: newState];
    [sender setState:(newState ? NSOnState : NSOffState)];
    // 写入用户默认
    [[NSUserDefaults standardUserDefaults]
        setBool:newState forKey:AutoResizeKey];
}

// 应用启动后 NSApplication 发送的委托消息
- (void)applicationWillFinishLaunching:(NSNotification *)aNotification
{
    BOOL flag;
    [self setupMainMenu];  // 生成并设定主菜单
    ctrlPool = [NSMutableArray array];  // 预留空数组
    // 从用户默认中读取自动缩小的设定值，并在 WinCtrl 类和菜单中设定
    flag = [[NSUserDefaults standardUserDefaults]
                                boolForKey:AutoResizeKey];
    [WinCtrl setAutoResize: flag];
    [autoResizeItem setState:(flag ? NSOnState : NSOffState)];
}

// 应用终止时 NSApplication 发送的委托消息
- (void)applicationWillTerminate:(NSNotification *)aNotification {
    [[NSUserDefaults standardUserDefaults] synchronize];
}

@end
```

17.3.2 类 WinCtr

WinCtrl 类实例生成显示图片的 NSImage 实例，并在初始化时打开图像文件。接着，打开窗体来显示该图片，其自身也是该窗体的委托。为了更简单地显示图片，这里使用了 NSImageView 类。在显示窗体的同时，也将自身保存在 MyViewerCtrl 的实例持有的数组中。

使用 Interface Builder 时，没用必要像调用`windowSetUp:`方法那样自己写代码来完成窗体的创建和设定。

该类也会使用`shrink:`方法将显示图像缩小一半。而实际进行处理的方法 `setScaleFactor:` 则会将返回到当前缩小率的操作写在撤销管理器中。据此可以实现撤销、重做缩小操作。缩小处理执行到最后，会发送图像大小变化的通知。反之，如果成为检查面板发送的通知的观察者，就可以实现将所有的窗体图像同时缩小一半的操作。这些功能中虽然可以使用自定义通知，但正如在 P.382 的 Column 中所述的那样，推荐将通知名等作为全局变量和宏名进行定义和使用。

WinCtrl 类实例就是窗体显示的委托。所以要采用协议 NSWindowDelegate。按下窗体关闭钮时，首先会发送询问消息`windowShouldClose:`。如果图像没有缩小显示则返回 YES，然后窗体被关闭。此时，由于`windowWillClose:`消息也会被发送，因此这里还需要释放自己的内存。

而图像缩小显示时，`windowShouldClose:` 方法则会显示确认窗体（图 17-4）并返回 NO。按下按键后，将调用显示页时指定的方法 `alertDidEnd:returnCode:contextInfo:`。如果按下 OK 键，就会取消窗体委托并关闭窗体，然后释放自身内存。

为了能够执行窗体撤销操作，需要实现 `windowWillReturnUndoManager:` 方法。

▶ 图 17-4 确认页

▶ 代码清单 17-4 WinCtrl 接口部分（WinCtrl.h）

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

extern NSString *ShrinkAllNotification;       // 自定义通知名：缩小全部
extern NSString *SizeDidChangeNotification;   // 自定义通知名：改变大小

@interface WinCtrl : NSObject <NSWindowDelegate>
{
    NSString *filename;           // 文件路径名
    NSWindow *window;             // 窗口
}
+ (void)initialize;
+ (BOOL)autoResize;
+ (void)setAutoResize:(BOOL)flag;
- (id)initWithURL:(NSURL *)aFile;
- (NSString *)attributesOfImage;
- (void)shrink:(id)sender;
@end
```

代码清单 17-5 WinCtrl 实现部分 (WinCtrl.m)

```
#import "WinCtrl.h"
#import "MyViewerCtrl.h"

#define  ImageSizeMIN   32       // 缩小时的最小尺寸
#define  TitleBarHeight 22       // 标题栏宽度

/*extern*/
NSString *ShrinkAllNotification = @"ShrinkAllNotification";
NSString *SizeDidChangeNotification = @"SizeDidChangeNotification";

static BOOL autoResize = NO;
static NSSize screenSize = { 1028.0, 768.0 };

@implementation WinCtrl {  // 实现为私有实例变量
    id docImage;                  // 图像对象
    NSSize originalSize;          // 图像的原始尺寸
    NSUndoManager *undoManager;   // 撤销管理器
    double mag;                   // 当前显示倍率
}

+ (void)initialize
{
    static BOOL nomore = NO;
    if (nomore == NO) {
        NSScreen *screen = [[NSScreen screens] objectAtIndex:0];
        screenSize = [screen visibleFrame].size;
        // 屏幕可以使用的尺寸
        nomore = YES;
    }
}

+ (BOOL)autoResize { return autoResize; } // 是否自动缩小
+ (void)setAutoResize:(BOOL)flag {
    autoResize = flag;
}

/* Local Method */
- (void)windowSetUp
{
    static int wincount = 0;                // 设置窗体显示位置
    float sft = (wincount++ & 07) * 20.0;   // 稍微挪动时使用
    id imageview;
    NSRect winrect, imgrect, contrect;
    float x, y;
    NSUInteger wstyle = (NSTitledWindowMask | NSClosableWindowMask
            | NSMiniaturizableWindowMask); // 窗体不可以改变尺寸

    contrect.size.width = (int)(originalSize.width * mag);
    contrect.size.height = (int)(originalSize.height * mag);
    contrect.origin = NSZeroPoint;      // 窗体内容的位置和大小
    winrect = [NSWindow frameRectForContentRect:contrect
                  styleMask:wstyle];    // 计算窗体的位置和大小
    x = 100.0 + sft;                    // 调整以置于屏幕中
```

```
    y = 150.0 + sft;
    if (x + winrect.size.width > screenSize.width)
        x = screenSize.width - winrect.size.width;
    if (y + winrect.size.height > screenSize.height)
        y = screenSize.height - winrect.size.height;
    contrect.origin = NSMakePoint(x, y);
    window = [[NSWindow alloc] initWithContentRect:contrect
        styleMask:wstyle backing:NSBackingStoreBuffered defer:YES];
    [window setReleasedWhenClosed:NO]; // 关闭时不自动释放
    imgrect.size = originalSize;
    imgrect.origin = NSZeroPoint;
    imageview = [[NSImageView alloc] initWithFrame:imgrect];
    [imageview setImage:docImage];
    [imageview setEditable:NO];                 // 不能拖曳图像
    [imageview setImageScaling:YES];            // 可以改变大小
    [imageview setFrameSize: contrect.size];    // 设定尺寸
    [window setContentView:imageview];          // 贴在窗口上
    [window makeFirstResponder:imageview];      // 成为快速反应者
}

/* Local Method: 接收通知的方法 */
- (void)shrinkAll:(NSNotification *)notification {
    [self shrink:[notification object]];
}

- (id)initWithURL:(NSURL *)aFile
{
    if ((self = [super init]) == nil)
        return nil;
    docImage = [[NSImage alloc] initWithContentsOfURL:aFile];
    if (docImage == nil)  // 如果不能从 URL 生成图像则初始化失败
        return nil;

    undoManager = [[NSUndoManager alloc] init];
    filename = [aFile path];
    originalSize = [docImage size];
    mag = 1.0;
    if (autoResize) {// 设为自动缩小时
        double wr, hr, ratio;
        wr = screenSize.width / originalSize.width;
        hr = (screenSize.height - TitleBarHeight) / originalSize.height;
        ratio = (wr < hr) ? wr : hr;
        if (ratio < 1.0)  // 需要缩小图像时
            mag = ratio;
    }
    [self windowSetUp];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(shrinkAll:) name:ShrinkAllNotification
        object:nil];            // 指定观察者
    [window setDelegate:self]; // 设定委托
    [window setTitleWithRepresentedFilename:filename];  // 设定标题
    [window makeKeyAndOrderFront:self];         // 成为主窗体
    [window setDocumentEdited:(mag < 1.0)];     // 在关闭按键中加入符号
    [[MyViewerCtrl sharedController] addWinCtrl: self];
    // 在 MyViewerCtrl 实例数组中记录并保持
```

```
    return self;
}

- (NSString *)attributesOfImage     // 生成简介中用于显示的字符串
{
    static NSString *fnamestr, *sizestr, *magstr;
    NSSize sz;

    if (fnamestr == nil) {  // 获得本地化使用的字符串
        fnamestr = NSLocalizedString(@"Filename", "filename");
        sizestr = NSLocalizedString(@"Size", "Size");
        magstr = NSLocalizedString(@"Magnification", "Magnification");
    }
    sz = [docImage size];
    return [NSString stringWithFormat:
        @"%@: %@\n%@: %d x %d\n%@: %.1lf%%",
        fnamestr, [filename lastPathComponent],
        sizestr, (int)sz.width, (int)sz.height, magstr, mag*100.0];
}

/* Local Method: 设为指定的缩小率 */
- (void)setScaleFactor:(double)factor
{
    id view = [window contentView];      // NSImageView 实例
    NSRect rect = [window frame];        // 得到窗体大小
    NSSize prev = rect.size;
    NSSize sz = [view frame].size;       // 得到图像大小
    int wdif = prev.width - sz.width;    // 图像和窗体的尺寸差
    int hdif = prev.height - sz.height;
    static NSString *shrinkName = nil;  // 取消操作的名称

    if (shrinkName == nil)
        shrinkName = NSLocalizedString(@"Shrink", "");
    // 在撤销管理器中记录当前缩小率的设置操作
    [[undoManager prepareWithInvocationTarget:self]
                                    setScaleFactor:mag];
    [undoManager setActionName: shrinkName];

    mag = factor;
    sz.width = (int)(originalSize.width * mag);
    sz.height = (int)(originalSize.height * mag);
    [view setFrameSize:sz];                  // 将图像设为新的尺寸
    rect.size.width = sz.width + wdif;
    rect.size.height = sz.height + hdif;
    rect.origin.x += (int)(prev.width - rect.size.width) / 2;
    rect.origin.y += (int)(prev.height - rect.size.height) / 2;
    [window setFrame:rect display:YES];      // 也将窗体变为新的大小
    [window setDocumentEdited:(mag < 1.0)]; // 在关闭窗口中设置符号
    [[NSNotificationCenter defaultCenter]    // 发送尺寸变化的通知
        postNotificationName:SizeDidChangeNotification object:window];
}

- (void)shrink:(id)sender     // 图像缩小一半
{
    NSSize sz = [[window contentView] frame].size;
```

```
    if (sz.width < ImageSizeMIN || sz.height < ImageSizeMIN)
        return;   // 太小时不缩小
    [self setScaleFactor:(mag * 0.5)];
}

/* Local Method: 框体关闭前的善后处理 */
- (void)removeWindowClosing:(BOOL)flag {
    [window setDelegate:nil]; // 解除 window 的委托
    [[NSNotificationCenter defaultCenter] removeObserver:self];
    // 解除所有通知的观察者设定
    if (flag)
        [window close];  // 关闭窗体
    window = nil;        // 在自己释放前删除引用
    [[MyViewerCtrl sharedController] removeWinCtrl: self];
    // 将自己从 MyViewerCtrl 数组中删除、释放
}

/* Local Method: 页的按钮被按下时被调用 */
- (void)alertDidEnd:(NSAlert *)alert
    returnCode:(NSInteger)returnCode contextInfo:(void *)contextInfo
{
    if (returnCode == NSAlertDefaultReturn)
        [self removeWindowClosing: YES];
}

/* 委托的方法 */
- (BOOL)windowShouldClose:(id)sender
{
    static NSString *warnstr, *closestr, *okstr, *cancelstr;
    NSAlert *alert;

    if (![window isDocumentEdited])
        return YES;        // 关闭键中没有符号时直接关闭
    if (warnstr == nil) { // 取得本地化使用的字符串
        warnstr = NSLocalizedString(@"File %@ is edited.", "Edited");
        closestr = NSLocalizedString(@"Close the window?", "Close?");
        okstr = NSLocalizedString(@"OK", "OK");
        cancelstr = NSLocalizedString(@"Cancel", "Cancel");
    }
    alert = [NSAlert alertWithMessageText:closestr
        defaultButton:okstr alternateButton:cancelstr otherButton:nil
        informativeTextWithFormat:warnstr,
                [filename lastPathComponent]];
    [alert beginSheetModalForWindow:window modalDelegate:self
        didEndSelector:@selector(alertDidEnd:returnCode:contextInfo:)
        contextInfo:nil];  // 在窗口中显示页
    return NO;             // 不关闭窗口
}

/* 委托的方法 */
- (void)windowWillClose:(NSNotification *)aNotification {
    [self removeWindowClosing: NO];
}

- (NSUndoManager *)windowWillReturnUndoManager:(NSWindow *)win
```

```
{   // 委托管理撤销管理器时，实现该消息
    return undoManager;
}
@end
```

17.3.3 类 MyInspector

下面将介绍控制检查面板的类 MyInspector。该类成为可加载束后，本地化文件必须单独包含，这点请一定要注意。

类方法 sharedInstance 在创建实例时只会创建一个。

检查面板上按键和文本的设定由本地方法 panelSetting 来完成。该面板可以改变尺寸，也可以被设置成即使关闭后也不会释放的方式。这里用 setFrameAutosaveName: 方法设定了面板的名字，而据此该面板的位置和尺寸也会被自动记录到用户默认信息中。使用 Interface Builder 时不需要进行这些设定。

面板左侧的“缩小”按键，虽然目标设定为了 nil，但在 GUI 环境中仍会向快速反应者发送消息。缩小和撤销、重做都可以从菜单操作，但这些也都会向快速反应者发送消息（为了避免繁琐，这里不详细介绍）。使用 Interface Builder 时，通过将快速反应者的图标作为目标与组件相连接，即可表示向快速反应者发送消息。

使用该程序，可以编辑显示用的文本域。因为文本域从最初就实现了撤销，因此利用这一点可以确认反应链的动作。检查面板为关键窗口时，执行撤销操作后，文本域的编辑结果而非显示图像的窗体就会被撤销。

图像显示用的窗体为主窗体时或者被关闭时，MyInspector 实例就会成为图像尺寸改变时发送的通知的观察者。接收这些通知时，主窗体可能会切换。

MyInspector 实例还是检查面板的委托。能够处理面板关闭时以及面板为关键窗体时的消息。这里使用的 isClosed 变量指明检查面板是否被显示。当不显示时，就不需要更新。

通过 showInfo: 局部方法可以显示更新。主窗体切换时、图像缩小后，以及检查面板显示后，都将调用该方法。该方法的参数为窗体。窗体的委托是 WinCtrl 实例，所以从该对象可以获得显示用的信息。

代码清单 17-6 MyInspector 接口部分（MyInspector.h）

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

@interface MyInspector: NSObject <NSWindowDelegate>
{
    NSPanel             *panel;
    __weak NSTextField *textfield;    // 文本显示区域
}                      // textfield 为 panel 中的组件，所以使用弱引用

+ (id)sharedInstance;
- (id)init;
- (void)activate;
- (void)showMain:(NSNotification *)aNotification;
- (void)windowClosed:(NSNotification *)aNotification;
- (void)shrinkAll:(id)sender;
@end
```

代码清单 17-7 WinCtrl 实现部分 (WinCtrl.m)

```
#import "MyInspector.h"
#import "WinCtrl.h"

#define  PanelWidthDef    250      // 面板默认大小
#define  PanelHeightDef   200
#define  BtnWidth         64       // 按键大小
#define  BtnHeight        32
#define  Margin           10       // 面板边缘的空白

@implementation MyInspector {
    BOOL        isClosed;          // 面板已被关闭了吗?
}

static id sharedInstance = nil;    // 保存唯一一个生成实例

+ (id)sharedInstance {             // 单体模式
    if (sharedInstance == nil)
        sharedInstance = [[self alloc] init];
    return sharedInstance;
}

/* Local Method */
- (void)panelSetting     // 面板一直使用同样的东西
{                        // 因此仅在实例初始化时设定一次
    NSButton *button[2];
    NSBundle *bundle;
    NSString *localstr;
    NSTextField *text;
    NSRect rect, btrect;
    float wid, hgt;
    int i;

    // 获取包含自己的类的可加载束
    bundle = [NSBundle bundleForClass:[self class]];
    rect = NSMakeRect(300, 300, PanelWidthDef, PanelHeightDef);
    panel = [[NSPanel alloc] initWithContentRect:rect
        styleMask: (NSTitledWindowMask | NSClosableWindowMask |
                    NSResizableWindowMask)    // 可以改变大小
        backing:NSBackingStoreBuffered
        defer:YES];
    [panel setReleasedWhenClosed:NO];  // 关闭时不释放
    [panel setMinSize:NSMakeSize(PanelWidthDef, PanelHeightDef)];
        // 设置改变的最小尺寸
    [panel setFrameAutosaveName:@"Inspector Panel"];
        // 设定将面板的位置和大小保存到用户记录时的名字
        // 已经用该名记录时，修改位置和大小
    localstr = [bundle localizedStringForKey:@"Inspector"
        value:@"" table:nil];          // 本地化使用的字符串
    [panel setTitle: localstr];
    [panel setDelegate: self];         // 指定自己为委托
    isClosed = YES;

    btrect = NSMakeRect(Margin, Margin, BtnWidth, BtnHeight);
```

```
    for (i = 0; i < 2; i++) {           // 设定按键的位置和大小
        button[i] = [[NSButton alloc] initWithFrame:btrect];
        [[panel contentView] addSubview:button[i]];
        [button[i] setAutoresizesSubviews:YES];
        [button[i] setAutoresizingMask:
                (NSViewMaxXMargin | NSViewMaxYMargin)];
        btrect.origin.x += Margin + BtnWidth;
    }
    localstr = [bundle localizedStringForKey:@"Shrink"
        value:@"" table:nil];
    [button[0] setTitle: localstr];     // 设置标题
    [button[0] setAction:@selector(shrink:)];  // 被按下时的行为
    [button[0] setTarget:nil];          // 被发送给快速反应者
    localstr = [bundle localizedStringForKey:@"All"
        value:@"" table:nil];
    [button[1] setTitle: localstr];
    [button[1] setAction:@selector(shrinkAll:)];
    [button[1] setTarget:self];         // 将实例自己作为目标

    rect = [panel frame];
    wid = rect.size.width - Margin*2;
    hgt = rect.size.height - (Margin + BtnHeight) * 2;
    textfield = text = [[NSTextField alloc] initWithFrame:
        NSMakeRect(Margin, BtnHeight*2 - Margin, wid, hgt)];
        // 设定文本显示部分的位置和大小
        // textfield 为弱引用，因此需要另外的强引用变量
    [text setSelectable:YES];  // 可以编辑、选择文本；也可以撤销
    [text setEditable:YES];
    [text setBezeled:YES];      // 有边框
    [[panel contentView] addSubview: text];  // 被保存为子视图
    [text setAutoresizesSubviews:YES];
    [text setAutoresizingMask:
        (NSViewWidthSizable | NSViewHeightSizable)];
}

- (id)init
{
    id center;

    if ((self = [super init]) != nil) {
        [self panelSetting];
        center = [NSNotificationCenter defaultCenter];
        [center addObserver:self selector:@selector(showMain:)
                name:NSWindowDidBecomeMainNotification object:nil];
        [center addObserver:self selector:@selector(showMain:)
                name:SizeDidChangeNotification object:nil];
        [center addObserver:self selector:@selector(windowClosed:)
                name:NSWindowWillCloseNotification object:nil];
        // 成为窗体发送的通知的观察者
    }
    return self;
}

- (void)dealloc {
    [[NSNotificationCenter defaultCenter] removeObserver:self];
```

```
    [panel setDelegate: nil];
}

- (void)activate
{
    [panel setFloatingPanel:YES];          // 总是在最前面显示的面板
    [panel makeKeyAndOrderFront: self];    // 显示在前面
}

/* Local Method */
- (void)showInfo:(id)obj  // obj 为主窗口
{
    NSString *log = @"";  // 空字符串时不显示
    if ([obj isKindOfClass:[NSWindow class]]) {
        id ctrl = [obj delegate];  // 委托理应为 WinCtrl
        if ([ctrl respondsToSelector:@selector(attributesOfImage)])
            log = [ctrl attributesOfImage];
    }
    [textfield setStringValue: log];
}

- (void)showMain:(NSNotification *)aNotification
{
    id obj;
    if (!isClosed  // 如果面板已被打开，且发送通知的一方为主窗口
    && (obj = [aNotification object]) == [NSApp mainWindow])
        [self showInfo:obj];
}

- (void)windowClosed:(NSNotification *)aNotification
{
    if (!isClosed && [aNotification object] == [NSApp mainWindow])
        [textfield setStringValue:@""];
        // 主窗口被关闭时不显示
}

- (void)shrinkAll:(id)sender  // 发送通知
{
    [[NSNotificationCenter defaultCenter]
            postNotificationName:ShrinkAllNotification object:self];
}

/* Delegate Message ... 显示面板 */
- (void)windowDidBecomeKey:(NSNotification *)aNotification
{
    isClosed = NO;
    [self showInfo:[NSApp mainWindow]];  // 再次显示信息
}

/* Delegate Message ...  面板被关闭 */
- (BOOL)windowShouldClose:(id)sender
{
    isClosed = YES;
    return YES;
}

@end
```

17.4 应用束的组织

17.4.1 创建编译和设置文件

如果要将主执行文件编译，可做如下操作。必须指定 Foundation 框架、Application 框架或者 Cocoa 框架（本应为一行，这里因为空间问题显示为了两行）。

```
clang -Wall -fobjc-arc -o MyViewer MyViewerCtrl.m WinCtrl.m
      main.m -framework Foundation -framework AppKit
```

接下来，编译可加载束的执行代码（因为空间关系这里显示为两行）。选项 -bundle 生成束。选项 -bundle_loader 表示下一个参数 MyViewer 是加载该束的程序。在链接处理时，如果发现未定义名字，就要检查是否已在加载方定义。

```
clang -fobjc-arc -bundle -bundle_loader MyViewer -o MyInspector
      MyInspector.m -framework Foundation -framework AppKit
```

应用的信息文件如代码清单 17-8 所示。应用名用键 CFBundleIndentifier 指定。应用执行后，为了保存用户默认信息，~/Library/Preferences 中会生成以该名字命名的属性列表文件。

代码清单 17-8 MyViewer 的信息文件 (Info.plist)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
     <key>CFBundleExecutable</key>
     <string>MyViewer</string>
     <key>CFBundleIdentifier</key>
     <string>jp.objCBook.MyViewer</string>
     <key>CFBundlePackageType</key>
     <string>APPL</string>
     <key>CFBundleSignature</key>
     <string>????</string>
</dict>
</plist>
```

可加载束的信息文件如代码清单 17-9 所示。

代码清单 17-9 可加载束的信息文件（Info.plist）

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
     <key>CFBundleExecutable</key>
     <string>Inspector</string>
     <key>NSPrincipalClass</key>
     <string>MyInspector</string>
     <key>CFBundleIdentifier</key>
     <string>jp.objCBook.MyViewer</string>
     <key>CFBundlePackageType</key>
     <string>BNDL</string>
     <key>CFBundleSignature</key>
     <string>????</string>
</dict>
</plist>
```

应用束用于日语的 Localizable.string 如代码清单 17-10 所示。文字编码为 UTF-16。用于英语的文件由于没有等号及右边的项，因此此处做了省略。

代码清单 17-10 MyViewer 用于日语的字符串文件（Localizable.strings）

```
"Filename" = "文件名";
"Size" = "尺寸";
"Magnification" = "放大率";
"Shrink" = "缩小";
"File %@ is edited." = "文件 %@ 被编辑。";
"Close the window?" = "是否关闭窗体";
"OK";
"Cancel" = "取消";
```

可加载束用于日语的 Localizable.strings 如代码清单 17-11 所示。

代码清单 17-11 可加载束用于日语的字符串文件（Localizable.strings）

```
"Inspector" = "简介";
"Shrink" = "缩小";
"All" = "所有";
```

这些都必须整理成图 17-3 的应用包的格式。app 后缀的目录不方便 finder 操作，因此从命令行执行创建目录的操作会更方便。从网络下载源码，然后输入 make 命令即可完成构建。

完成应用束后，双击启动程序。

17.4.2 程序运行例子

程序运行界面的例子如图 17-5、17-6 所示。因为设置了快捷键，所以命令键和“o”都可以显示

打开面板。

在图 17-5 中尝试打开多个图像。然后，按下检查面板的“所有”按键，将所有图像都缩小 1/2（图 17-6）。菜单的“撤销”适用于各种图像，也可以还原图像大小。

图 17-5 程序运行示例（1）

图 17-6 程序执行示例（2）

17.4.3 GUI 定义文件和程序

使用 Interface Builder 定义 GUI 后，定义内容会被保存在 nib 文件中。每个 nib 文件中会定义如

下事项。

- 配置窗体和面板的内部组件，改变大小时指定尺寸变化
- 指定与各组件外观、动作相关的属性
- 配置图像和声音文件
- 按下按键时发送的消息及其目标
- 将其他对象设定为某个对象的出口
- 非 GUI 组件类的实例需相互连接
- 设置反应链
- 设置菜单
- 作为 nib 文件范围内的对象所有者的对象
- 设置工具条（鼠标位置处显示的说明）
- 设置 Cocoa 绑定

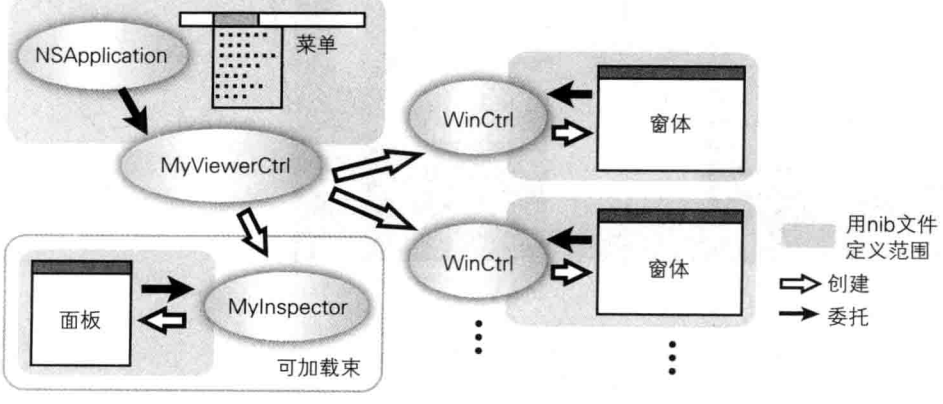
其中，Cocoa 绑定就是利用第 20 章所述的键值编码，来大幅减少对象间因耦合而产生的代码的技术。

Interface Builder 并不是单纯的 GUI 组件配置工具。基于目标 - 动作模式（参考 8.2 节）的概念，当操作某个 GUI 组件时，可以设定向什么对象发送什么消息。这些对象可以配置包括自定义类在内的任意类实例。这样构成的对象只要是 GUI 组件，就可以在 Interface Builder 内实际操作，而且也不需要编译。

此外，一个对象的实例变量或声明属性中也可以连接别的对象。Interface Builder 将此“连接口”称为出口。在面向对象的程序中，设置由对象构成的拓扑结构非常麻烦。而利用 Interface Builder 就可以用非常简单的方式构建对象间的连接。但是，使用 ARC 时要注意不要产生死循环（见 16.2 节）。

本章的图像视图如果利用 Interface Builder 来创建的话，就可以省略一大部分程序代码。对象间关系（图 17-1）中可以用 Interface Builder 设置的部分如图 17-7 所示。

图 17-7 可以使用 nib 文件设定的部分



从程序方面来看，只要在运行中指定并读入 nib 文件，用 Interface Builder 设定的对象结构就会

被动态构成，之后直接利用就行了。例如，在 nib 文件中定义 WinCtrl 实例和窗体的关系时，只需读入 nib 文件，就可以生成与新的 WinCtrl 实例相对应的窗体了（见 16.2 节）。

这次生成的源程序中，使用类 MyViewerCtrl 的方法 setupMainMenu、类 WinCtrl 的方法 windowSetup 及类 MyInspector 的方法 panelSetting 定义的内容基本上都不需要。此外，也不需要把菜单的内容当作属性列表记录的 Menu.plist 文件。

专栏：Objective-C 调试器的功能 COLUMN

Mac OS X 的 Objective-C 编译器因为基于 gcc，所以可以使用调试器 gdb（GNU Debugger）。在 Xcode 中可以使用的调试器也是 gdb。除了在控制台输入命令之外，在利用断点查找等情况下，如果已知命令也会非常方便。编译时，gcc 和 clang 都指定选项 -g 来生成调试信息。

只要了解几个命令就能有效利用 gdb。因此，即使不知道使用方法，也要尝试去用一下。gdb 使用方法可以在书中或与 Unix 相关的文章、网站等处查找。

Mac OS X 的 gdb 中增加了面向 Objective-C 的功能。下面将简要说明。

（1）表示对象

新增的命令 po（print-object）可被用来表示对象的相关信息。该命令的参数对象可使用 description 方法，并显示返回的字符串。

为了获得变量或公式的值，通常使用命令 p（print），但即使对象可使用 p 命令，结果也只能表示 id 值，也就是指针值。而如果在使用 * 运算符后再使用 p 命令，调用该对象中的数据时就可以采用与表示结构体同样的方式。

（2）使用消息格式

p 命令的参数不只是变量值，也可以是书写的格式或函数调用等，这里还可以为消息格式。据此，就可以方便地向某个对象发送消息或查看状态等。

消息格式的返回值为对象时，可以用 po 命令来表示，而如果返回值为整数或实数，则必须通过映射指定类型后才能表示。因为实际执行的方法是动态变化的，所以返回值类型与调试器无任何关联。例如下面的式子。

```
p (int)[val initValue]
p (NSRect)[view frame]
```

（3）设置断点

使用 b（break）命令来设定断点，参数中可以指定源码内的行数或函数名。这里还可以新指定消息选择器。

Objective-C 语言中，很多情况下不同的类都定义了同样的名字。此时可以显示出候补的一览列表，并从中选择目标名字。所有的同名方法都可以指定断点。例如，发生例外时使用 `raise` 方法（参考第 18 章），但如果指定该 raise 为断点，那么，当异常发生时，调试器就可以十分方便地查找原因。

指定特定的类方法时，可按如下方式。

```
break -[MyClass initWithName:]
break +[FirstClass new]
```

前面的 + 号为类方法，– 号为指定的实例方法，括号左侧是类名，右侧为方法选择器。

第18章

异常和错误

在程序执行的过程中，如果发生了某些操作错误、异常等，有时就要中断正常执行来进行处理。本章将说明系统为处理这样的错误而提供的异常处理结构。而且，在 Cocoa 环境中，为了统一表示错误的种类和表现消息等，可以使用 NSError 类。有关该类的使用方法也会在本章中详细说明。

18.1 异常

18.1.1 异常处理的概念

异常（exception）就是指必须中断程序的正常执行来进行处理的特殊状态。

Foundation 框架有很多类，当指定的方法不能正常处理时就会产生异常。例如，NSMutableArray 类的方法`addObject:`会将参数对象添加到接收器的数组中，当参数为 nil 时就会产生表示输入参数不正确的异常 NSInvalidArgumentException。而且，使用方法`objectAtIndex:`访问的索引位置如果超过了元素个数，表示范围错误的异常 NSRangeException 就会发生。像这样，除了程序错误引起的异常外，文件、通信信道的输入输出终止也能引起异常。

在最近的编程语言中，编码时往往采取将异常发生时的处理和程序原本应该执行的处理分开进行的结构。这就是**异常处理机制**（exception handling mechanism）。当异常状况产生时，程序会自动脱离出普通的函数和方法的调用，转而执行异常处理过程。

因此，提供功能的程序方面就可以把精力集中在原本应该执行的处理上，来编写相应的代码。而功能的使用方也是如此，将异常处理代码和普通执行流程分开写更好，这样就可以清晰地记述繁琐的错误处理了。

18.1.2 Objective-C 中的异常处理

Objective-C 的 Mac OS X10.3 中增加了异常处理机制，当异常发生时就会进入专门的处理流程进行处理。但是，**利用异常的程序或通过引发异常来控制动作的编程类型并不被推荐**，这点也被明确写在了苹果公司的各种文档中。也就是说，因设计错误或程序错误而产生的执行时错误，在开发时就应该被去除。

这里我们来看一下使用 Objective-C 的异常处理机制的目的。首先，在软件开发时，特别是在测试等的时候，使用该机制可以把握意外的发生并对调试起到一定的辅助作用。基于这个目的的情况下，完成后的软件不会包含任何异常。

再有，当应用中发生异常时，为了使该异常的影响不会波及到整个应用，而且在必要的善后处理后就可以快速终止程序的执行，也会使用异常处理机制。同样，当某个库或框架内发生异常时，该机制也会被用来在其内部捕捉并处理异常，以防止波及到外部。

总之，Objective-C 的异常的使用范围并不广泛，不是说感到“不能正常运行了，尝试修正一下吧”就来使用异常，异常是一种在非常情况下退出流程的方式，使用异常是为了“趁事态不严重时尽可能地妥善处理，然后再终止程序”，大家一定要认识到这一点。在 Java 等程序中，读入文件不存在时也会使用异常来处理，但在 Objective-C 中，这样的编程风格是不被推荐的。

有关异常处理的详细内容请参考“Exception Programming Topics”等文档。

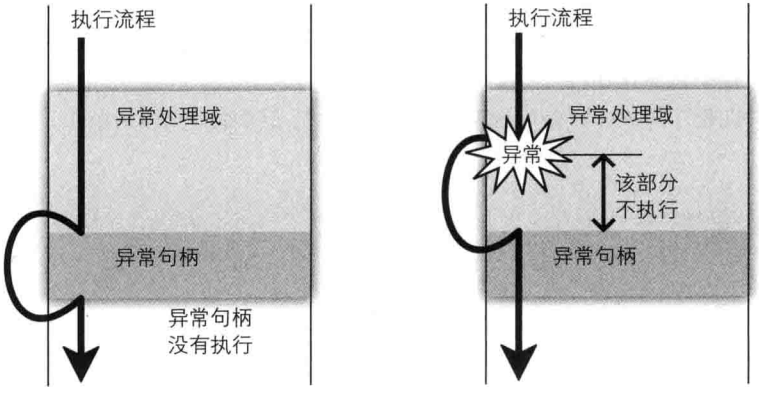
18.2 异常处理机制概述

18.2.1 异常句柄和异常处理域

异常发生时，为执行异常处理而准备的流程称为**异常句柄**（exception handler）。该异常句柄处理的对象的代码部分称为**异常处理域**（exception handling domain）。

Fig18-1（a）是异常没有发生时的情况，此时异常句柄不被执行。（b）是在异常处理域执行的过程中又有其他异常发生时的情况，此时正常的执行流程会被中断，异常句柄的处理被立刻执行。异常句柄首先会查看该异常的内容，如果可以处理则执行相应的处理，然后继续执行程序。不能处理的异常一般就成为运行时错误，并导致程序终止。用异常句柄执行异常处理，有时也称为用异常句柄捕获（catch）异常。

图 18-1 异常处理的概念



（a）异常没有发生时　　（b）异常发生时

18.2.2 异常表示类 NSException

在 Cocoa 环境中，可以用类 NSException 来表示异常。接口在 Foundation/NSException.h 中声明。

通过生成 NSException 实例并发送 raise 消息，就可以启动异常处理。这也称为“产生异常”或“抛出异常”。NSException 实例称为**异常对象**。

- (void) **raise**

向异常对象抛出异常。

或许有人不太不习惯这种说法。实际上，像“没有内存可分配”、“参数不正确”这样必须要处理的问题就是异常状况，因为向 NSException 类实例发送raise消息时该状况已经发生了。

但是，在异常处理机制方面看来，只要没有开始异常处理，就不是异常。也就是说，向NSException 实例发送消息raise这一时刻和异常发生被同等看待了。而且，有时也会将指向NSException 的实例称为异常，这也是同样的想法。

异常对象包含表 18-1 的信息。

表 18-1 NSException 持有的信息

信息	解释
名称	为了识别异常而使用的短字符串。判断是否为应该使用异常句柄处理的对象时使用。使用下面的方法取出 `- (NSString *) name`
原因	用文字描述异常原因的长字符串。用如下方法取出 `- (NSString *) reason`
用户字典	传递与异常相关的各种信息时使用的 NSDictionary 字典。字典中的信息因异常而异。用如下方法取出 `- (NSDictionary *) userInfo`

异常的名字为全局字符串变量，被预先定义在各种类的头文件中。调查发生的异常时，将捕捉的异常对象名和这些异常名比较。而且，像什么样的情况下发生什么异常之类的信息，以及用户字典中传入的信息，在各类的参考文档中都有详述。

用户也可以定义新的异常，并在程序进入非正常状况时抛出异常（之后会详细介绍其方法）。

18.2.3 异常处理机制的语法

异常处理机制采用如下语法[①]。

语法 异常处理

```
@try {
        /* 异常处理域 */
        /* 在此记述正常处理 */
        ...
}
@catch (NSException *exception) {
        /* 异常句柄 */
        /* 记述异常处理过程。例如显示错误消息等 */
        ...
}
@finally {
        /* 后处理 */
        /* 在此记述无论异常发送与否最后都会执行的代码 */
        ...
}
```

@try 必须写在 @catch 和 @final 的前面。

① 显然为 Java 的语言风格。

@catch 块可以写多个。@catch 块也可以不写，但此时必须有 @finally 块。按上述方式书写 @catch 后括号内的内容时，可通过变量 exception 访问与发生的异常相对应的 NSException 实例。在这部分中，可以像函数的形式参数那样指定任意变量名。这里指定的变量名是只在 @catch 块内有效的局部变量名。

@finally 块不需要时也可以不写。无论异常发生与否、@catch 块捕捉到异常与否，该块内的代码都一定会执行。有关该块的详细介绍在后面的章节中还会提到。

18.2.4 简单的异常处理的示例程序

下面是一个简单的程序例子，它会处理发生的异常。

代码清单 18-1 example.m

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

int main(int argc, char *argv[])                    // 使用 ARC
{
    @autoreleasepool {
        id array = [NSMutableArray array];          // 空数组

        @try {
            if (argc == 1)
                array = [array objectAtIndex:1];    // NSRangeException
            else
                [array addObject:nil];  // NSInvalidArgumentException
            NSLog(@"success\n");
        }
        @catch (NSException *ex) {
            NSString *name = [ex name];
            NSLog(@"Name: %@\n", name);
            NSLog(@"Reason: %@\n", [ex reason]);
            if ([name isEqualToString:NSRangeException])
                NSLog(@"Exception was caught successfully.\n");
            else
                [ex raise];     // 再次抛出异常
        }

        NSLog(@"main exit");    // 行尾没有换行符也没有关系
    }
    return 0;
}
```

该程序根据命令行中是否存在参数而产生不同的异常。生成空数组，取出第一个不存在的元素时将产生异常 NSRangeException，追加新的元素 nil 时将产生异常 NSInvalidArgumentException。因为 @catch 块中捕获的异常可以访问 ex 变量，所以首先显示异常的名字和原因。接着，当发生的异常为 NSRangeException 时就显示能够捕获异常这一消息，而当其他异常发生时，则向 ex 发送 raise 消息来再次抛出异常。这样一来，NSRangeException 以外的异常通常就会由运行时系统来处理。

下面实际运行一下。

首先，不给参数传递任何变量时的执行例子如下所示（省略了每一行头部的附加信息（见下文的 Column））。通过“main exit”可以看出，异常被捕获，后续部分正在被执行。

```
% ./exsimple
Name: NSRangeException
Reason: *** -[__NSArrayM objectAtIndex:]: index (1) beyond bounds (0)
Exception was caught successfully.
main exit
```

接着是传递了参数时的情况。可以看出，异常虽然被捕捉了，但之后却显示了和普通的运行时错误发生时几乎相同的消息，然后程序异常就终止了。

```
% ./exsimple X
Name: NSInvalidArgumentException
Reason: *** -[__NSArrayM insertObject:atIndex:]: object cannot be nil
*** Terminating app due to uncaught exception 'NSInvalidArgumentException',
reason: '*** -[__NSArrayM insertObject:atIndex:]: object cannot be nil'
（以下省略）
```

这是因为运行时系统的**未捕获异常句柄**（uncaught exception handler）流程捕获并处理了异常。异常处理域外部发生异常时，或程序异常句柄中异常处理不能结束时，未捕获的异常句柄就会向终端、控制台或日志文件中打印消息然后终止程序。

但是，Cocoa 应用的主线程内发生异常时，NSApplication（或 UIApplication）实例将捕捉未处理的异常，所以未捕获异常句柄并不会直接终止应用。此时的日志可以从控制台中得到确认。然而，即使应用能捕捉到，异常的产生也仍可能会给应用带来若干影响，导致应用之后的处理不能正确执行，这点需要注意。

专栏：日志输出函数 NSLog()　COLUMN

代码清单 18-1 中使用了 NSLog() 函数。该函数使用与 printf() 相同的格式字符串及相应个数的参数，向 stderr（标准错误输出）输出错误。与带有 NSString 类的格式的方法相同，格式通过 NSString 字符串指定，也可以使用“%@”。输出的各行的头部有时间、程序名等信息。行尾没有回车时将自动添加回车。而且，从 Cocoa 应用中用 NSLog 输出的字符串，也是控制台输出的日志。

18.3 异常的发生和传播

18.3.1 异常的传播

异常处理的语法通常可以和其他控制语句组合，也可以嵌入到到其他语句中使用。而且，异常处理语法块内调用的方法或函数有时也包含异常处理语法。也就是说，伴随着程序的执行，某个异常处理域中可能还会有其他异常处理在执行，异常句柄中也可能再次发生其他异常。

只使用 @catch 块不能结束某个异常处理时，为了委托外部异常句柄来处理，有时就需要在 @catch 块内故意抛出异常。像这样，为保证异常的正常处理，被调用方向调用方有序传递异常的过程，称为异常**传播**（propagation）。任何异常句柄，当不能被合适处理或在异常处理域外部发生异常时，最终都会由未捕获异常句柄来处理。

18.3.2 自己触发异常

如前所述，向异常对象发送raise消息就可以抛出异常。但是，实际上使用下面的类 NSException 的方法会更加容易。

```
+ (void) raise: (NSString *) name
        format: (NSString *) format,...
```

将异常名及原因保存成信息并创建临时实例，直接产生异常。
原因由 printf() 样式的格式字符串，以及用逗号分割的任意个参数构成。用户字典为 nil。

该方法及其他产生异常的方法的相关介绍，请参考 NSException 的参考文档。

也可以自定义异常名并产生新型的异常。此时，必须起一个可以和其他异常相区别的名字。异常名的添加方法请参考附录 C。

18.3.3 用 @throw 语法产生异常

产生异常一般使用 @throw 语法。语法格式如下所示，object 部分中可以指定任何对象。将 @throw 方法应用于 NSException 实例，同发送raise消息是一样的。

语法	产生异常
`@throw object;`	

即便 @catch 块内没有参数也可以使用 @throw。此时，@catch 块捕捉的异常对象就被视为参数，同传入参数的效果一样。使用该写法，能够更加方便地记述捕捉到的异常的传播处理。

将 NSException 以外的普通对象指定为 @throw 参数也可以产生异常，在捕获这样的异常时，@

catch 语法中必须要指明该对象的类型。如果对象类型不明确，那么使用 id 类型后，结果就将捕捉包含 NSException 在内的所有异常。

@catch 块可以被书写多个，但是必须要注意它们的顺序。这是因为，根据书写顺序能够得知用 @throw 产生异常时使用的对象和哪个 @catch 块参数一致。例如下面的例子，MyClassA 是 MyClassB 的超类，object 是其中一个类的实例，这时异常一定会被 ❶ 的 @catch 块捕捉到，而 ❷ 不会捕捉到。此外，如果将 ❸ 的 @catch 块写在 ❶ 和 ❷ 的上方，则异常只能被 ❸ 的块捕捉到。

```
@try {
    ...; @throw object;
}
@catch (MyClassA *ex) {                                    ❶
    ...
}
@catch (MyClassB *ex) {                                    ❷
    ...
}
@catch (id ex) {                                           ❸
    ...
}
```

18.3.4 @catch 的特殊语法

@catch 参数除了上面介绍的书写方法之外，也可以采用下面的方式书写。即括号内不省略，而是写三个点号。

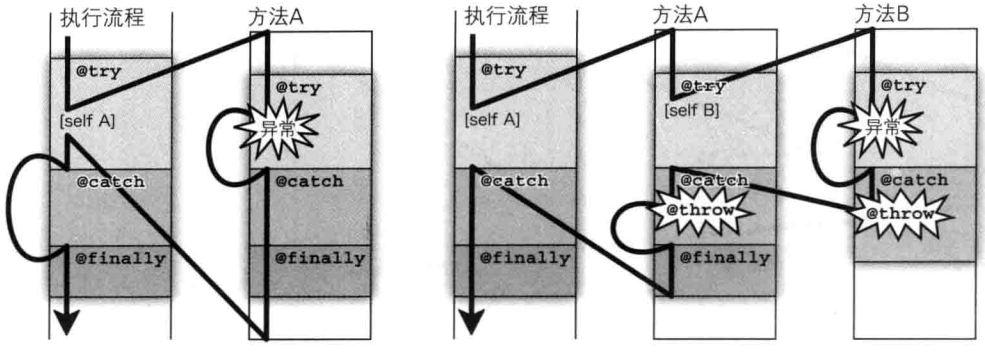
```
@catch ( ... ) {
    /* 异常句柄 */
}
```

在需要捕捉异常而不需要知道异常内容的情况下，经常会使用该写法。如果存在其他的 @catch 块，则必须将其书写为最后的块。块内 @throw 不指定参数时，就能传播异常。

18.3.5 异常传播和 @finally

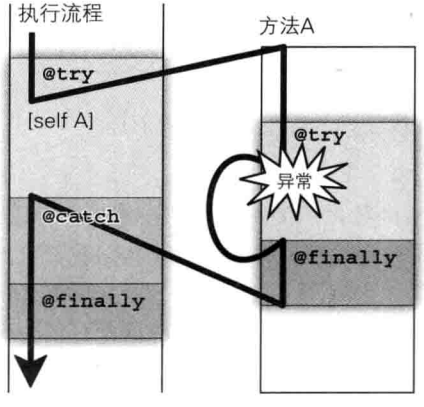
@catch 块内发生异常时（或传播），或者不存在可以捕捉发生的异常的 @catch 块时，通常就会直接将处理移交给外部的异常句柄来执行。但如果有 @finally 块，那么在向外侧异常句柄移交处理前，@finally 块的内容将先被执行。在没发生异常时，@finally 块会在 @try 块之后执行，而当异常不向外侧传播时，@finally 块则必须紧接着 @catch 块执行。所以，这里就要书写实例释放或文件关闭等必须要执行的后处理（图 18-2）。如果不捕捉异常而只想执行后处理，也可以不写 @catch，只写 @try 和 @finally。此时，异常发生后 @finally 块就会被立刻执行（图 18-3）。

图 18-2 异常的传播和 @finally 块



（a）异常不传播时　　（b）异常传播时

图 18-3 只有 @try 和 @finally 时



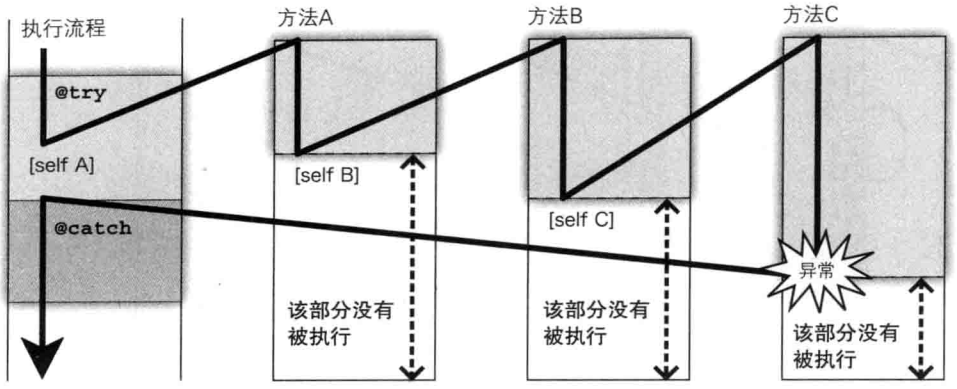
18.3.6 异常处理程序的注意点

（1）递归方法调用内发生异常

N 重递归方法调用（或函数调用）中发生异常时，处理有时就会跳过内侧的方法而转移到外侧的异常句柄中进行（图 18-4）。此时，在被跳过的方法中，有些处理就可能会被非正常终止。例如，为了释放对象 / 对象不被释放掉而进行的处理可能尚未结束，或者文件或通信相关的后处理没有得到执行等等。

所以，当程序使用异常处理时，异常发生源、异常句柄及可能因异常而导致处理被中断的方法调用都需要我们小心应对。在图 18-4 中，无论是方法 C 还是方法 A 和方法 B，如果处理被中断时必须进行后处理，那么就要在 @try 中加入方法调用，并在 @catch 或 @finally 中写后处理代码。

图 18-4 因为发生异常而导致处理被中断的例子



(2) 引用计数管理和异常

使用引用计数的内存管理方式时，如果因为异常的发生而将处理交给了异常句柄，那么就不能保证这期间使用的对象能被正确释放。手动的计数管理方式和 ARC 也都一样。使用 ARC 时，还存在弱访问的变量不能清零的问题。

因此，我们可以这么说，在 Objective-C 中，写代码时不能以使用异常处理机制为前提。

(3) 关于 C 语言中的全局跳转函数

C 语言中有两个函数 setjmp() 和 longjmp()，它们都可以从函数调用的深层递归中返回。但是，由于这里的异常处理机制在实现时使用了这些功能，所以异常处理机制流程和这些跳转函数不可以同时使用。

Objective-C 语言中组合使用 C 语言的函数时，要确保 setjmp() 和 longjmp() 的组合只在 C 语言的流程内结束，这点请注意。

18.4 断言

18.4.1 断言是什么

出于调试的目的，有时需要查看程序必须满足的制约条件是否被破坏了。例如，在某个方法执行前，变量 x 的值必须为零、此类条件的组合不应该发生等等。

像这样，在程序中书写程序必须满足的条件，当条件破坏时就触发异常的结构称为**断言**（assertion）。

断言使用宏来书写，宏定义在头文件 Foundation/NSException.h 中。

断言宏的参数中写着必须为真的条件表达式。例如，变量 x 的值必须比变量 y 大时，可按如下方式表述（关于宏的内容会在后边介绍）。

```
NSAssert(x > y, @"Illegal values x(%d) & y(%d)", x, y);
```

条件为假时，异常 NSInternalInconsistencyException 就会被触发。实际上，每个线程中都会自动生成 NSAssertionHandler 类实例，该对象虽然会触发异常，但基本上不需要对该类直接操作。

编译时定义了NS_BLOCK_ASSERTIONS宏时，断言不会被写入代码。而因为没有写入，所以就会被当作空字符串处理。于是，在编译完成后的程序时，就要在编译器的选项中加入-DNS_BLOCK_ASSERTIONS。选项 -D 的作用是可以从命令行定义宏。这样使用时，就不需要特意在文件中书写#define了。

18.4.2 断言宏

断言宏可以在 Objective-C 方法内以及 C 函数内使用。断言条件为假时，如果是在函数内，那么就用函数名来表示；而如果是在方法内，那么就用方法名来表示。

首先，只在条件为真时使用的宏有两个。

NSParameterAssert(*condition*)　　… 在方法内使用

NSCParameterAssert(*condition*)　　… 在函数内使用

此外还有宏可以在条件为假时传递触发的异常，以及生成用来说明异常的字符串。它们和 printf() 一样，取得格式字符串和对应的参数。使用目前为止的 C 语言规范尚不可以定义参数个数不确定的宏，而在新的规范中是可以的。

在方法内使用

NSAssert(*condition, NSString *description [, arg, ...]*)

在函数内使用

NSCAssert(*condition, NSString *description [, arg, ...]*)

专栏：包含可变个数的参数的宏 COLUMN

在新的 C 语言规范中，可以书写包含可变个数的参数的宏。

例如，下面是一个很常见的例子。宏的形参部分中写着“…”，宏的展开部分中可以用 __VA_ARGS__ 来访问该处。然而，在这个例子中，因为参数 fmt 被“>>>”连接，所以必须要指定确定个数的字符串。

```
#include <stdio.h>

#define Debug(fmt, ...)  fprintf(stderr, ">>>" fmt, __VA_ARGS__)

int main(void)
{
    Debug("%d\n", 1);
    Debug("%d, %d, %d, %d\n", 1, 2, 3, 4);
    return 0;
}
```

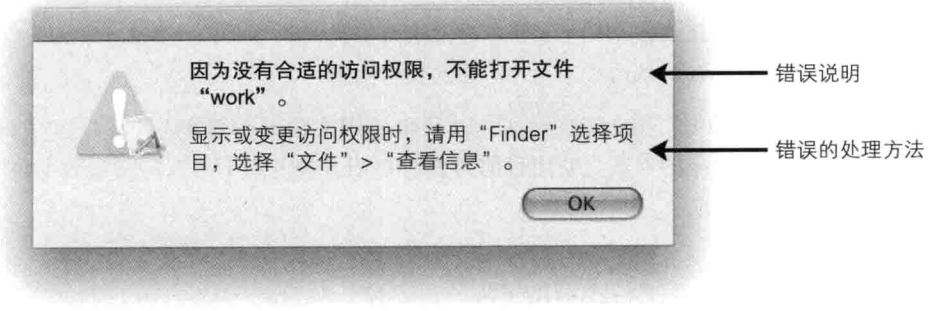
18.5 错误处理

18.5.1 错误处理结构的目的

在一般的流程型程序中，因为不能读入指定文件或输入数据有误等问题不能按照正常流程运行时，利用手写或函数返回值来表示错误发生是常见的方法。但是，单纯靠错误代码和消息，是很难确定错误的原因以及如何处理的。

在 Cocoa 环境下，为了能够统一表示错误的种类和消息，可以使用类 NSError。简单地说，NSError 持有显示图 18-5 那样的报警面板时所需的相关信息。

图 18-5 基于错误显示的报警面板



类 NSError 实例在 Foundation/NSError.h 中声明。然而在下面的描述中，NSError 实例有时又称为错误对象。

18.5.2 表示错误的类 NSError 的使用方法

NSError 实例包含表 18-2 所示的信息。

表 18-2 NSError 包含的信息

信息	解释
域	导致错误的技术因素，例如，显示 Coacoa、Carbon、Unix 等中谁的 API 相关的字符串。用下面的方法取出 `- (NSString *)domain`
代码	表示错误类型的整数值。需要注意的是代码的意义可根据域而发生变化。用下面的方法取出 `- (NSInteger)code`
用户字典	为得到有关错误的各种信息而使用的字典对象。字典中保存的信息随错误而不同。用下面的方法取出 `- (NSDictionary *)userInfo`

因为这些信息的具体内容因域和错误种类而异，因此这里不再详细说明，但书写确切应对错误的函数时，有必要详细调查一下各个信息。必要的信息在相关类别、API 的参考文档及头文件中都有说明。

错误对象可作为函数或方法返回值返回给调用方，在 Coacoa API 中，大多数情况下都是用可能发生错误的消息的末尾参数来获得。

例如在 NSString 中，使用如下方法可使指定的文件内容作为字符串返回（第二个参数 enc 指定字符串编码）。

```
+ (id) stringWithContentsOfFile: (NSString *) path
                       encoding: (NSStringEncoding) enc
                          error: (NSError **) error
```

第三个参数 error 为指针，指向 NSError 的实例指针变量，也就是指针的指针。此处的使用方法非常简单。如下所示，准备好保存实例的变量，并使用 & 运算符指向该地址即可。发生指定的文件不存在等错误时，代入错误对象给变量 err。如果没有发生错误则不代入对象。因此，为了明确表示错误没有发生，应该养成预先为变量 err 代入 nil 的操作习惯。

然而，使用 ARC 管理内存时，方法内的自动变量地址必须将（NSError **）类型的参数指定为 nil（见 5.6 节）。

```
NSString *path, *contents;
NSError *err;

...
err = nil;
contents = [NSString stringWithContentsOfFile: path
                encoding: NSShiftJISStringEncoding
                error: &err];
if (contents == nil) {
    NSAlert *alert = [NSAlert alertWithError: err];
    (void)[alert runModal];  // 显示警告窗口
    break;
...
}
```

这里使用了 Application 框架的 NSAlert 类，将错误对象作为参数，即可方便地显示相应的警告窗体。也可以像第 17 章的示例程序那样，将其作为表单显示在窗体中。但正如简单图像视图的例子那样，NSApplication 应用必须要考虑到运行回路。

而且，如果应用被本地化为日语，那么警告面板中显示的消息就多为日语。实际上图 18-5 中的面板的日语是系统显示的。因为要使用到系统预先保存的日语文字，因此如果没有日语资源则使用英语。

方法内部发生错误时，可以选择直接向调用方返回错误。像框架方法那样使用最后的参数返回错误时，方法如下所示。这在使用 ARC 时也同样适用。

```
- (NSString *)contentsOf:(NSString *)name error:(NSError **)error
{
    ...
    path = [directory stringByAppendingPathComponent: name];
    contents = [[NSString alloc] initWithContentsOfURL: path
                    encoding: NSShiftJISStringEncoding
                    error: error];  // 直接使用传入的指针
    if (contents == nil)
        return nil;
    ...
}
```

18.5.3 获取错误对象的信息

为了获得表示错误内容的消息字符串，可向 NSError 实例发送下面的消息。没有必要直接访问用户字典或域等。

- (NSString *) **localizedDescription**

返回说明错误的字符串。字符串被本地化，不会返回 nil。用户字典中如果存在 NSLocalizedDescriptionKey 键的对象，则返回该对象。

- (NSString *) **localizedFailureReason**

显示错误产生的原因，返回本地化的短字符串。有时返回 nil。用户字典中如果存在 NSLocalized FailureReasonErrorKey 键的对象，则返回该对象。

- (NSString *) **localizedRecoverySuggestion**

显示错误的处理方法或建议，返回本地化的短字符串。有时返回 nil。用户字典中如果存在 NSLocalized RecoverySuggestionErrorKey 键的对象，则返回该对象。

以图 18-5 的警告面板显示时的情况为例，错误对象在调用上述 3 个方法时依次返回如下对象。

localizedDescription

…字符串“因为没有合适的访问权限，不能打开文件‘work’。”

localizedFailureReason

…字符串“没有合适的访问权限”

localizedRecoverySuggestion

…字符串“显示或变更访问权时，请用‘Finder’选择文件，选择‘文件’>‘查看信息’。”

18.5.4 生成自定义错误对象

与异常相同，自己也可以生成 NSError 实例。可以使用下面的简易常数类。

+ (id) **errorWithDomain:** (NSString *) domain

```
code: (NSInteger) code
userInfo: (NSDictionary *) dict
```

指定域名、错误代码和用户字典后创建错误对象。用户字典为nil也没有关系，域名必须要指定。

典型的错误域名如下所示（表 18-3）。

表 18-3 典型的错误域名

域名	解释	头文件
NSCocoaErrorDomain	与 Cocoa 环境相关的错误	Foundation/FoundationErrors.h
		AppKit/AppKitErrors.h
		CoreData/CoreDataErrors.h
NSPOSIXErrorDomain	与 Unix 环境相关的错误	sys/errno.h
NSMachErrorDomain	与 Mach 核相关的错误	mach/kern_return.h

返回新创建的错误对象时，如果错误原因和上述任一个域相同，就可以直接使用域和错误代码。在 Mac OS X 环境下，用户字典即使为 nil，只要指定了域名和错误代码，多数情况下警告面板中也都会显示日语的错误消息。

自定义程序可以创建自己专用的新域名。此时，错误代码和用户字典的内容也需要自己来管理。错误消息及警告面板中显示的按钮标题等信息被一并保存在用户字典中传递。建议将字典中保存的字符串本地化。

用户字典可以自由地登录入口，但为了特定目的，这里有多个键被预先定义为了常数。如表 18-4 所示。值为显示消息的字符串时，基本上都已经被本地化了。

表 18-4 错误对象在字典中的预定键

键	值
NSLocalizedDescriptionKey	说明错误内容的字符串
NSLocalizedFailureReasonErrorKey	说明错误原因的短字符串
NSLocalizedRecoverySuggestionErrorKey	显示错误处理方法的字符串
NSLocalizedRecoveryOpionsErrorKey	显示恢复方法的按键的标题字符串数组
NSRecoveryAttempterErrorKey	执行恢复处理的对象
NSUnderlyingErrorKey	该错误对象的源错误对象
NSFilePathErrorKey	错误相关的文件路径字符串
NSStringEncodingErrorKey	包含编码号的 NSNumber 对象
NSURLErrorKey	错误相关的 NSURL 对象

18.6 错误反应链

通过将 NSError 结合应用框架来使用，就可以在 GUI 环境中灵活地进行错误处理。核心模块是 Mac OS X 10.4 后引入的**错误反应链**（error-responder chain），该模块与 15.4 节中介绍的反应链类似。

下面将简述什么是错误反应链，该功能只在 Mac OS X 上有效。详情请参考“Error Handling Programming Guide”等文档。

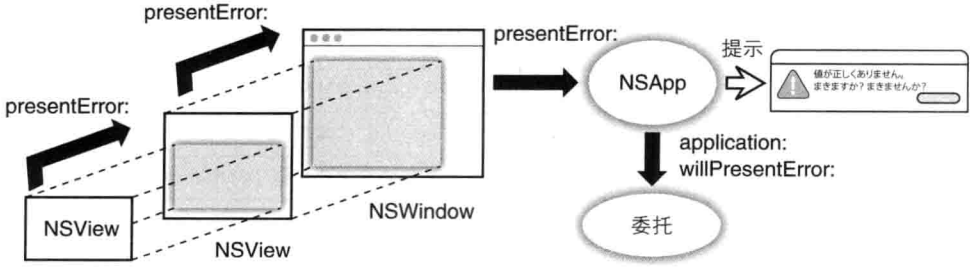
18.6.1 错误反应链的结构

当与某组件操作相关的错误发生时，有必要将错误内容发送给用户，或者根据用户的指示对错误进行处理（恢复）。但是，如果根据错误发生场所来分别进行处理的话，从应用整体来看就会有处理不统一的危险。于是，可以将错误处理规则与 GUI 组件的层次结构相互对应，这样就形成了错误反应链。

正如在介绍反应链时所说的那样，GUI 组件大部分是 Application 框架的 NSResponder 类的子类，它们在窗体中保持着层次构造。错误反应链和反应链同样，是位于底层的组件指向上层组件的对象（反应器）的连接。

当与某个组件相关的错误发生时，会将错误对象作为参数将消息`presentError:`发送给该组件。于是错误对象就会从该组件开始被转发给上层的组件。最终由窗体发送给 NSApplication 实例，错误提示和恢复被执行（图 18-6）。

图 18-6 错误反应链的概念



在错误传送中使用了方法`presentError:`，它是 NSResponder 实例的方法，将错误发送给错误反应链指向的下一个对象。如果错误恢复成功则返回 YES。

```
- (BOOL) presentError: (NSError *) anError
```

处理文档的窗口可使用如下方法。

```
- (void) presentError: (NSError *) error
```

```
modalForWindow: (NSWindow *) aWindow
      delegate: (id) delegate
didPresentSelector: (SEL) didPresentSelector
   contextInfo: (void *) contextInfo
```

使用该方法时，错误通过指定的窗体表单显示。这里省略使用该方法时的详细情形，下面只简单说明一下presentError：方法。

接收presentError：方法的反应器使用 self 的错误对象作为参数来调用下面的方法，并将返回的错误对象发送给上层的反应器。

```
- (NSError *) willPresentError: (NSError *) anError
```

NSResponder 的实现中将原样返回该参数。但是，NSResponder 子类中需要重写该方法。子类中可以访问错误对象参数包含的信息，然后将包含更加详细的错误消息以及恢复所需要的信息的新错误对象返回。然而，willPresentError：方法需要在子类中由用户来定制，而 presentError：方法不能重写。

同样，Application 实例的委托也可以定义下面的方法。

```
- (NSError *) application: (NSApplication *) application
        willPresentError: (NSError *) error
```

18.6.2 错误对象的更改和恢复

针对发生的错误，有时也可以根据用户的指示来执行恢复。例如，读入的数据形式不正确时，是继续读取还是使用代替数据或终止处理程序，这些都需要根据用户的选择来进行。

此时，可以在相关的组件类中写入willPresentError：方法的实现代码，或者在 NSApp 委托中定义application:willPresentError:，并返回包含恢复所需信息的错误对象。

因为发生错误传递的错误对象以及用户字典是常数对象，所以上述方法必须要返回一个新创建的错误对象。而且，原错误对象使用新错误对象用户字典中 NSUnderlyingErrorKey 键并保存它（表 18-4）。

首先，为了让用户选择如何进行恢复，必须按照选择分支的个数来传入按键标题字符串。具体来说，就是像下面的错误对象方法返回字符串数组那样，将数组记录在用户字典中。

```
- (NSArray *) localizedRecoveryOptions
```

将警告面板的按键标题所使用的本地化字符串放入数组中返回。用户字典中如果存在NSLocalizedRecoveryOpionsErrorKey键的对象则返回。

包含多个字符串时，第一个字符串在面板的最右侧（默认），然后是右边第二个，按照此顺序排列。也可能会返回 nil，此时警告面板中只显示“OK”按键。

当然，为了使用户选择合适的按钮，需要向用户展示确切的说明消息。如上一章所述，或许也需要预先将 NSLocalizedDescriptionKey 或 NSLocalizedRecoverySuggestionErrorKey 键的消息字符串在用户字典中重新记录。

接着，在取得按键的选择结果后，指定对象来执行恢复处理。该对象就称为**恢复尝试对象**（recovery attempter），为了能作为下面的错误对象方法的返回值返回，该对象在用户字典中记录。

- (id) **recoveryAttempter**

  返回错误的恢复尝试对象。用户字典中如果存在NSRecoveryAttempterErrorKey键的对象则返回。如果没有入口则返回nil。

恢复尝试对象可以是任意类的实例，但为了进行恢复处理，需要实现下面所示的 NSErrorRecovery Attempting 非正式协议的其中任意一个方法。该非正式协议在 Foundation/NSError.h 中声明。

```
@interface NSObject(NSErrorRecoveryAttempting)
- (void)attemptRecoveryFromError:(NSError *)error
      optionIndex:(NSUInteger)recoveryOptionIndex
      delegate:(id)delegate
      didRecoverSelector:(SEL)didRecoverSelector
      contextInfo:(void *)contextInfo;
- (BOOL)attemptRecoveryFromError:(NSError *)error
      optionIndex:(NSUInteger)recoveryOptionIndex;
@end
```

前者主要被用于窗体页的应用，所以此处不做介绍。后者的情况下，表示错误对象和警告面板中用户选择的按键位置的索引会被作为参数传入。该索引与错误对象的用户字典中传入的字符串数组的索引一致，所以可以访问该值并进行相应的处理。恢复成功时返回 YES。该返回值也是最初调用的方法presentError:的返回值。

例如，某新错误对象可以选择继续、其他文件、取消这三个选项，返回该对象的方法定义如下所示。因为恢复尝试对象指定了 self，所以能够应对选择结果的处理方法必须要在同一个类中定义。

```
- (NSError *)willPresentError:(NSError *)err
{
    NSMutableDictionary *dic;
    NSError *nwerr;
    dic = [NSMutableDictionary dictionaryWithDictionary:
              [err userInfo]];
        // 将原错误对象的用户字典复制为可变的字典对象。
    [dic setObject:[NSArray arrayWithObjects:
          NSLocalizedString(@"Continue", "Continue"),
          NSLocalizedString(@"Try Other", "Try Other"),
          NSLocalizedString(@"Cancel", "Cancel"), nil]
        forKey:NSLocalizedRecoveryOptionsErrorKey];
        // 使其能显示 3 个按键。并进行本地化。
    [dic setObject:self forKey:NSRecoveryAttempterErrorKey];
        // 指定 self 为恢复尝试对象
    [dic setObject:err forKey:NSUnderlyingErrorKey];
        // 将原错误对象保存在用户字典中返回
    nwerr = [NSError errorWithDomain:[err domain] code:[err code]
        userInfo:dic];  // 生成新错误对象
    return nwerr;
}
```

也可以在 NSApp 委托中实现同样的方法，此时必须要确保通过访问错误代码和用户字典，可以恰当处理各种类型的错误对象。

专栏：单元测试 COLUMN

目前，各种软件开发环境中都提供了可以方便地进行单元测试（单体测试）的机制。

单元测试就是在类或方法等单元中确认生成的程序是否正确执行的一种测试。在多数开发环境中，通过在软件开发的同时进行这样的单元测试，就可以在修正程序或添加功能等时机自动执行该测试。因此，该测试不仅能保证程序的质量，对软件的改善（重构）也有不错的效果。

即使是 Objective-C，使用 Xcode 的程序开发中也可以实施单元测试。这里主要借助了开源的成果，具体来说就是从 Xcode 构建 OCUnit 单元测试并执行。本书中不详细介绍 Xcode 中的实现方法，只简单地说明一下概要。

单元测试将 SenTestingKit 框架类 SenTestCase 的子类定义为工程目标。从子类方法中调用测试操作，并通过使用 SenTestKit 框架定义的断言来确认是否能够得到期待的结果。如果结果与预想的不同，断言中就会发送错误报告。

关于单元测试的具体实现方法，可以参考“iOS App Development Wordflow Guide（iOS 开发向导）”等文档。

第19章

并行编程

本章将说明使用线程进行并行处理的方法。首先会接触一些基本的线程及锁的相关概念，然后再说明能够高效进行并行化功能的类 NSOperation 的使用方法。最后介绍可以在 Mac OS X 中使用的连接的概况。

19.1 多线程

19.1.1 线程的基本概念

线程（thread）是进程（process）[①] 内假想的持有 CPU 使用权的执行单位。一般情况下，一个进程只有一个线程，但也可以创建多个线程并在进程中并行执行。应用在执行某一处理的同时，还可以接收 GUI 的输入。

使用多线程的程序称为**多线程**（multithread）运行。从程序开始执行时就运行的线程称为**主线程**，除此之外，之后生成的线程称为次线程（secondary thread）或子线程（subthread）。

创建线程时，创建方的线程为父线程，被创建方的线程为子线程。父线程和子线程并行执行各自的处理，但父线程可以等到子线程执行终止后与其会合（join）。而另一方面，在线程被创建后，也可以切断父子关系指定它们不进行会合。该操作称为**分离**（detach）。这里所说的 NSThread 就是在分离状态下创建线程。

由于被创建的线程共享进程的地址空间，所以能够自由访问进程的空间变量。多线程访问的变量称为**共享变量**（shared variable）。共享变量大多为全局变量或静态变量，但因为地址空间是共享的，所以理论上所有内存区域都可以称为共享变量。

如果多线程胡乱访问共享变量，那么就不能保证变量值的正确性。所以有时就需要按照一定的规则使多线程可以协调动作。此时就必须执行线程间**互斥**（或者排他控制，mutual exclusion）（见 19.2 节）。

各个线程都分配有栈且独立进行管理。基本上不能访问其他线程的栈内的变量（自动变量）。通过遵守这样的编程方式，就可以自由访问方法或函数的自动变量，而且不用担心互斥。

使用引用计数管理方式时，为了使对象之间解耦合，子线程方需要创建与父线程不同的自动释放池来管理。使用垃圾回收时不需要这样。

19.1.2 线程安全

多个线程同时操作某个实例时，如果没有得到错误结果或实例没有包含不正确的状态，那么该类就称为**线程安全**（thread-safe）。结果不能保证时，则称为非线程安全或线程不安全（thread-unsafe）。

一般情况下，常数对象是线程安全的，变量对象不是线程安全的。常数对象可以在线程间安全地传递，但对变量对象共享时，需要恰当地执行互斥或同步切换。

需要注意的是 C 语言的函数。就现状来看，BSD 函数的大部分，例如 printf() 等，都不是线程安全的。

---

① 任务（task）这一名称也被用来表示与进程同样的概念，在苹果公司的文档“Multithreading Programming Topics”中，可以包含多线程的程序的执行单元称为进程，而任务则被用来抽象地表示应该进行的作业。

19.1.3 注意点

在某些情况下，使用多线程可以使处理高速化、实现易于使用的接口、使实现更简单等。但并不是说使用多线程后就一定会得到这些优点。

要想使多线程程序不出错且高效执行，并行编程的知识必不可少。线程间的任务分配和信息交换、共享资源的互斥、与 GUI 的交互及动画显示等，在使用时都要特别小心。

一般情况下，自己实现多线程程序是很困难的，而且也容易埋下高隐患。稍有差错或设计失误，多线程便不能发挥效果，甚至还会导致未知原因的释放或异常终止。使用 19.3 节中介绍的 NSOperation，虽然可以较容易地实现多线程程序，但是也必须掌握线程动作、互斥等相关知识。不能适应这些的读者建议去参考一下并行编程的相关书籍。

而且，很多多线程中遇见的问题都可以通过 NSTimer 类或延迟消息发送（参考 15.1 节）来解决。大家也不妨尝试一下用这些方法来解决相关问题。

19.1.4 使用 NSThread 创建线程

Foundation 框架中提供了 NSThread 类来创建并控制线程。该类的接口在 Foundation/NSThread.h 中声明。

创建新线程需要执行下面的类方法。

```
+ (void) detachNewThreadSelector: (SEL) aSelector
                        toTarget: (id) aTarget
                      withObject: (id) anArgument
```

对对象 aTarget 调用方法创建新线程并执行。选择器 aSelector 必须是仅获取一个 id 类型参数且返回值为 void 的执行方法（参考 8.2 节）。

指定的方法执行结束后，线程也随之终止。线程从最初就被执行了分离，所以终止时没有和父线程会合。当主线程终止时，包含子线程的程序也全部随之终止。

使用引用计数管理（手动及 ARC）时，有时需要执行的方法自身来管理自动释放池。此外，参数 aTarget 和 anArgument 中指定的对象也与线程同时存在，即在创建线程时被保存，在线程终止时被释放。

使用下述的 NSApplication 类中的方法也能创建线程。该方法使用上面的方法，而且在使用引用计数管理时还会创建线程的自动释放池。

```
+ (void) detachDrawingThread: (SEL) selector
                    toTarget: (id) target
                  withObject: (id) argument
```

创建新线程并执行的方法除了上述方法还有很多，本书中不再一一介绍。其他方法请参考 NSThread、NSObject 的参考文档。

程序可以调用 NSThread 类方法来确认是否是多线程运行。

```
+ (BOOL) isMultiThreaded
```

多个线程并行执行时或者只有主线程在执行时，只要在此之前已经创建了线程，则返回 YES。

19.1.5 当前线程

一个线程称自身为**当前线程**（current thread），区别于其他线程。

子线程将创建时指定的方法执行完后也会随之终止，但也可以中途终止。为此，可以使用当前线程（线程自身）来执行下一个 NSThread 类方法。但是，使用引用计数管理时，终止前一定要释放自动释放池。

```
+ (void) exit
```

使用下述方法获得表示线程的 NSThread 实例。

```
+ (NSThread *) currentThread
```

获得表示当前线程的 NSThread 实例。

```
+ (NSThread *) mainThread
```

获得表示主线程的 NSThread 实例。查看当前线程是否为主线程时，可以使用类方法 isMainThread。

每个线程都可以持有一个该线程固有的 NSMutableDictionary 类型的字典。向 NSThread 实例发送下面的消息类就可以取得字典。

```
- (NSMutableDictionary *) threadDictionary
```

可以使当前线程仅被中断几秒。为此，可在当前线程中执行下面的类方法。参数为实数。

```
+ (void) sleepForTimeInterval: (NSTimeInterval) ti
```

也可以使线程在某一时刻前中断，这时可采用下面的类方法。参数是表示日期的类 NSDate 实例。

```
+ (void) sleepUntilDate:(NSDate *) aDate
```

如果要使线程到某个条件成立前一直保持休眠状态，则要使用下一章节介绍的锁。

19.1.6 GUI 应用和线程

在使用 GUI 的应用中，事件处理和绘图等大部分处理中线程都发挥了重要作用。也可以在子线程中创建窗体，或分担部分绘图功能，但要注意避免竞争或内存泄漏。详情请参考相关文档。

GUI 应用中有较容易的方法来使用线程，即将 GUI 相关的时间处理或绘图集中在主线程中进行。使用下面的方法，就可以从子线程依赖主线程中的方法处理。该方法为 NSOjbect 的范畴，在头文件 Foundation/NSThread.h 中声明。

```
- (void) performSelectorOnMainThread: (SEL) aSelector
                          withObject: (id) arg
                       waitUntilDone: (BOOL) wait
```

选择器aSelector和参数arg中指定的方法的执行依赖于主线程。wait为YES时，当前线程会一直等待至执行结束。主线程中必须有事件循环(运行回路)。

19.2 互斥

19.2.1 需要互斥的例子

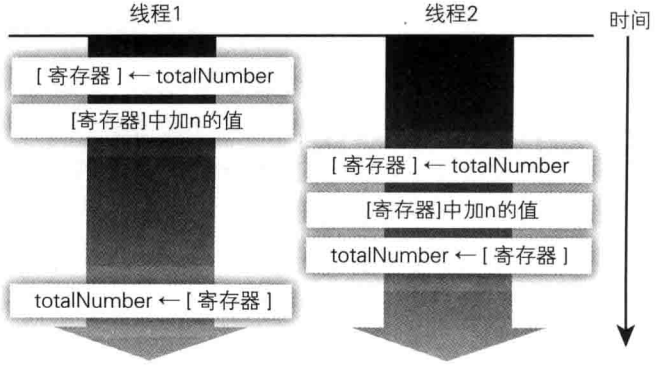
在多线程环境中，无论哪个函数或方法都可以在多线程中同时执行。但是，在使用共享变量时，或者在执行文件输出或绘图等的情况下，多线程同时执行就可能得到奇怪的结果。

例如，使用整数全局变量 totalNumber 来累加所处理的数据的个数。为了执行下面的加法计算，在多线程环境中执行该方法会得到什么结果呢?

```
- (void)addNumber:(NSIngeger)n
{
    totalNumber += n;
}
```

在 OS 功能支持下，线程在运行的过程中会时而得到 CPU 的执行权，时而被挂起执行权，2 个方法的执行情况如图 19-1 中所示。在该图中，线程 1 将新计算的值保存在寄存器时挂起 CPU 执行权，同时线程 2 开始执行方法。即使 CPU 的执行权被挂起，寄存器的值也仍然可以被保存，所以各线程都能正常处理。但是，由于线程 2 写入的值消失了，因此整体上看，这偏离了我们期待的结果。原因是值的读取、更新、写入操作被多线程同时执行了。

图 19-1 没有互斥的例子



在图 19-1 的例子中，我们将同时只可以由一个线程占有并执行的代码部分称为**临界区**（critical section），或称为危险区。互斥的目的就是限制可以在临界区执行的线程。

19.2.2 锁

为了使多个线程间可以相互排斥地使用全局变量等共享资源，可以使用NSLock类。该类的实例也就是可以调整多线程行为的**信号量**（semaphore）或者**互斥型信号量**（mutual exclusion semaphore）。Cocoa 环境中也称为**锁**（lock）。

锁具有每次只允许单个线程获得并使用的性质。获得锁称为“加锁”，释放锁称为“解锁”。

锁和普通的实例一样，使用类方法alloc和初始化器init来创建并初始化。但是，锁应该在程序开始在多线程执行前创建。

```
NSLock *countLock = [[NSLock alloc] init];
```

获得锁的方法和释放（unlock）锁的方法都在协议 NSLocking 中定义。

- (void) **lock**

  如果锁正被使用，则线程进入休眠状态。
  如果锁没有被使用，则将锁的状态变为正被使用，线程继续执行。

- (void) **unlock**

  将锁置为没有在被使用，此时如果有等待该锁资源的正在休眠的线程，则将其唤醒。

在上例中，使用锁后会产生如下效果。但需要预先创建 NSLock 的实例 aLock。在该代码中，从某线程执行 ❶ 取得锁到该线程执行 ❷ 释放锁期间，其他线程在执行 ❶ 时将进入休眠状态，不能执行临界区代码。锁被释放后，在执行 ❶ 时休眠的线程中选择一个线程，该线程在取得锁后进入临界区执行。

```
- (void)addNumber:(NSIngeger)n
{
    [aLock lock];  ——————————————————————————————————————— ❶
    totalNumber += n;    // 临界区
    [aLock unlock];  ——————————————————————————————————————— ❷
}
```

某个锁被lock后，必须执行一次unlock。而且lock和unlock必须在同一个线程执行[①]。

下面来看另外一个使用锁的例子。考虑一下全局变量值自增时返回其结果的方法。多线程执行时，全局变量 theCount 若想正确地自增，就需要使用锁 countLock 来管理。

可以采用如下定义。

① lock 和 unlock 必须在同一个线程中执行，因为 NSLock 是基于 POSIX 线程实现的。

```
- (int)inc
{
    [countLock lock];
    ++theCount;
    [countLock unlock];
    return theCount;
}
```

咋一看好像没问题，但从释放锁到返回值期间，其他线程可能会修改变量值。如下所示。

```
- (int)inc
{
    int tmp;

    [countLock lock];
    tmp = ++theCount;
    [countLock unlock];
    return tmp;
}
```

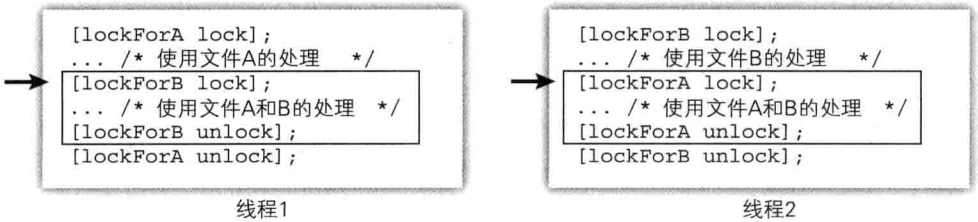
各线程持有独立的栈，自动变量 tmp 可以在整个线程中局部利用，而且无需担心被其他线程访问。如果按照这样的方式执行，就可以得到临界区之后的正确值。

19.2.3 死锁

线程和锁的关系必须在设计之初就经过仔细的考虑。如果错误地使用锁，不但不能按照预期执行互斥，还可能使多个线程陷入到不能执行的状态，即**死锁**（deadlock）状态。

死锁就是多线程（或进程）永远在等待一个不可能实现的条件而无法继续执行，如图 19-2 所示。

图 19-2 陷入死锁的典型例子



线程 1 占有文件 A 并正在进行处理，途中又需要占有文件 B。而另一方面，线程 2 占有着文件 B，途中又需要占有文件 A。大家不妨设想一下，如果线程 1 和线程 2 同时执行到了图中的箭头位置会怎么样呢？线程 1 为了处理文件 B 想要获得锁 lockForB，但是它已经被线程 2 获得。同样，线程 2 想要获得的锁 lockForA 也被线程 1 占有着。这种情况下，线程 1 和线程 2 就会同时进入休眠状态，而且双方都不能跳出该状态。

像这样，当多个线程互相等待资源的释放时，就非常容易出现死锁现象。有时是多个线程相干

预，有时则是一个线程因为自己需要获得锁而进入休眠状态。此外，由于多数情况下各个线程本身并没有错误处理，而且死锁又随时可能发生，因此追究原因就非常困难，也不能排除导致程序 bug 的可能。

19.2.4 尝试获得锁

NSLock 类不仅能获得锁和释放锁，还有检查是否能获得锁的功能。利用这些功能，就可以在不能获得锁时进行其他处理。

- (BOOL) **tryLock**

用接收器尝试获得某个锁，如果可以取得该锁则返回YES。不能获得时，与lock处理不同，线程没有进入休眠状态，而是直接返回NO并继续执行。

该方法十分便利，但要确保只能在可以获得锁时才执行 unlock，创建程序时必须注意这一点。

19.2.5 条件锁

类 NSConditionLock 称为**条件锁**（condition lock）。该锁持有整数值，根据该值可以获得锁或者等待。

- (id) **initWithCondition:** (NSInteger) condition

NSConditionLock实例初始化，设置参数condition指定的值。
NSCondtionLock的指定初始化器。

- (NSInteger) **condition**

此时返回锁中设定的值。

- (void) **lockWhenCondition:** (NSInteger) condition

如果锁正在被使用，则线程进入休眠状态。
锁不在被使用时，如果锁值和参数condition的值一致，则将锁状态修改为正在被使用，然后继续执行，如果不一致，则线程进入休眠状态。

- (void) **unlockWithCondition:** (NSInteger) condition

在锁中设置参数condition指定的值。将锁设置为不在被使用，此时如果有等待获得该锁且处于休眠状态的线程，则将其唤醒。

- (BOOL) **tryLockWhenCondition:** (NSInteger) condition

尚未使用锁且锁值与参数condition相同时，获得锁并返回YES。不能获得锁时也不进入休眠状态，而是返回NO，线程继续执行。

使用方法lock、unlock或tryLock都可以获得锁和释放锁，而且无需关心锁的值。

然而，由于 NSConditionLock 实例可以持有的状态为整数型，所以事先用枚举常数或宏定义就可以了。如果只使用 0 或 1，不仅不容易理解，也可能造成错误。

19.2.6 NSRecursiveLock

某线程获得锁后，到该线程释放锁期间，想要获得该锁的线程就会进入休眠。使用类 NSLock 的锁时，如果已经获得锁的线程在没有释放它的情况下还想再次获得该锁，该线程也会进入休眠状态。但是，由于没有从休眠状态唤醒的线程，所以这就是死锁。下面是一个简单的例子，这段代码不会执行。

```
[aLock lock];
[aLock lock];        // 这里发生死锁
[aLock unlock];
[aLock unlock];
```

解决这种情况可以使用NSRecursiveLock类的锁，拥有锁的线程即使多次获得同一个锁也不会进入死锁。但是，其他线程当然也不能获得该锁。获得次数和释放次数一致时，锁就会被释放。

NSRecursiveLock 类的锁使用起来十分方便，但排除被重复加锁的情况，用 NSLock 来重新记述的话，性能则会更好。

19.2.7 @synchronized

程序内的块可以指定为不被多线程同时使用。为此可以使用 @synchronized 编译符，如下所示。

语法 线程的锁

```
@synchronized(obj)
{
    // 记述想要排斥地执行的内容
}
```

通过使用该段代码，运行时系统就可以创建排斥地执行该代码块的锁（mutex）。参数 obj 通常指定为该互斥锁要保护的对象。obj 自己不需要是锁对象。

线程如果要执行该代码块，首先会尝试获得锁，如果能获得锁则可以执行块内代码。块执行结束时一并释放锁。使用 break 或 return 从块内跳出到块外时也被视作块执行终止。而且，在块内发生异常时，运行时系统会捕捉异常并释放块。

@synchronized 的参数对象决定对应的块。所以，**同一个对象参数的@synchronized块如果有多个，则不可以同时执行**。

根据参数的选择方法的不同，@synchronized 会在并行执行的受限对象和可以执行的普通对象之间动态切换。下面展示 @synchronized 参数的使用示例。

(a) 是指定只能单独存在的对象时的情景。同一个对象在其他地方也作为 @synchronized 的参数使用时，所有这些块不能同时执行。(b) 也是一样，因为限制了参数的使用范围，互斥对象显然只能是该方法内的块。

(c) 是各个实例互斥的例子。一个实例一次只能执行一个线程，同一类别的其他实例则多个线程

可以同时存在。(d) 在参数对象可能在多个地方更改的情况下有效，但以同样方式使用该对象的所有场所中都需要按照该方式书写，否则就没有任何意义。

而且，也可以按照 (e) 的方式书写。此外还可以指定类对象，或者使用消息选择器（隐藏参数的 _cmd ）来指定方法等。不过一般情况下，为互斥的对象使用专门的锁对象是比较可靠的方法。

```
static id g = ...;

- (void)doSomething:(id)arg
{
    static id loc = ...;
    @synchronized(g) { ... }                    // (a)
    @synchronized(loc) { ... }                  // (b)
    @synchronized(self) { ... }                 // (c)
    @synchronized(arg) { ... }                  // (d)
    @synchronized([self localLock]) { ... }  // (e)
}
```

@synchronized 与上述 NSRecursiveLock 类的锁一样，可以递归调用。例如，下述这种简单的例子就不会死锁。

```
@synchronized(obj) {
    @synchronized(obj) {
        ...
    }
}
```

使用 @synchronized 块时，加锁和解锁必须成对进行，因此可以防止加锁后忘记解锁这种问题的发生。和普通的锁相比，复杂的并行算法的书写会较为复杂，但多数情况下都会使互斥更容易理解。

19.3 操作对象和并行处理

19.3.1 新的并行处理程序

Mac OS X 10.6 及 iOS 4.0 后，导入了可以使系统全体线程更高效运行，并且使并行处理应用更易开发的架构，称为**大中央调度**（GCD，Grand Central Dispatch）。

目前为止，为了提高应用的运行速度和 GUI 的易用性，程序多采用了多线程。但是，这就需要与线程的创建、同步、共享资源的互斥等相关的知识、观察力、经验及技术，而且代码的调试和功能扩展也存在诸多困难。

在使用 GCD 的程序中，可以将想要提高处理速度的部分分解为非同步执行的多个任务，并将这些任务放入等待队列（queue）中。虽然必须要指定任务间的依赖关系（前后关系），但却不需要关心

线程的创建和调度。然后，CPU 内核数以及其他应用的要求都会被动态地判断，由系统决定最佳的并行程度和高效的调度方式并执行。

GCD 的核心是用 C 语言写的系统服务，应用将应该执行的各种任务封装成块对象写入到队列中。通过对比各任务启动线程的代价，该操作可以非常轻量地运行。

为了直接使用 GCD 的功能，必须要使用 C 语言函数，然而，在 Objective-C 中，通过用 NSOperation 类来表示任务，并将其追加到 NSOperationQueue 类队列中，即可实现并行处理。下面，简单说明这些类的使用方法。详细内容请参考“Concurrency Programming Guide”“Grand Central Dispatch（GCD）Reference”等文档。

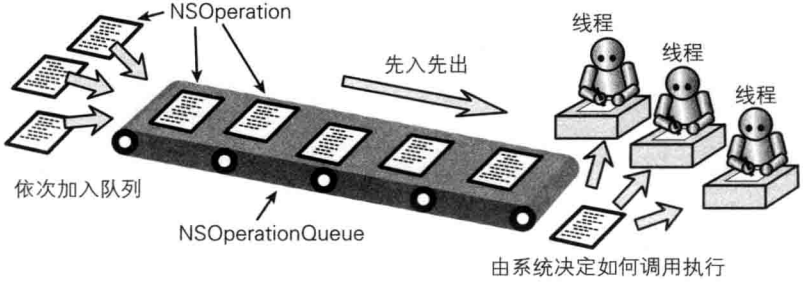
在下面的说明中，我们将需要执行的任务聚合在一起，称为抽象**任务**（task）。

19.3.2 使用 NSOperation 的处理概述

和将 NSArray 实例称为数组对象一样，下面的 NSOperation 实例也可以称为**操作对象**（operation object），或简称为**操作**。

操作对象可以将需要执行的代码和相关数据集成并封装。操作对象通常不是被直接执行，而是在等待队列中被按顺序调用。线程也不是一直被创建，而是保持一定的数量，并将操作交给空闲线程处理（图 19-3）。这种线程管理方式称为线程池。在 GCD 中，线程数由硬件或当前系统的整体负荷等条件来决定。至于各个操作在哪个线程执行以及采取怎样的调度方式等，编程人员则不用关心。

图 19-3 操作对象和队列的概念



操作不仅仅可以按照等待队列中的顺序被调用，也可以像“任务 A 完成后调用任务 B”这样指定调用的先后关系。此时，任务 B 依赖于任务 A，这种关系称为**依赖关系**（dependency）。而且，还可以设定操作的**优先权**（priority）。

为操作提供了等待队列功能的是 **NSOperationQueue** 类。以下将该类的实例称为**操作队列**（operation queue），或简称为**队列**。等待队列是一种先入先出的数据结构，也称为 FIFO（First In, First Out）。操作一旦被加入到队列，至于如何执行，就要任凭系统来调度。但是，加入到队列的操作在执行之前也可以取消。

下面我们来看一下如何将 NSOperation 类加入队列并执行任务。不过，在此之前，这里先介绍一下其他操作。操作对象也可以直接调用`start`方法来执行任务。但是为了能并行执行任务，标准做

法是在start内创建线程。本书中不再介绍按照这种方式使用操作对象的方法。关于操作的定义方法等详细内容，请参考 NSOperation 类的参考文档或“Concurrency Programming Guide”。

19.3.3 NSOperation 和 NSOperationQueue 的简单用法

NSOperation 提供了处理任务的功能，它本身又是抽象类。所以就需要先定义子类，然后再在子类中定义必要的处理。但是，由于操作对象可以被多个线程同时访问，因此子类方法必须是线程安全的。NSOperation 实例在头文件 Foundation/NSOperation.h 中定义。

NSOperation 子类必须将要执行的任务写入到下面的main方法中。由于在 NSOperation 中定义的方法不发挥任何作用，所以并没有必要调用 super。

```
- (void) main
```

NSOperation 子类至少需要包含 main 方法，此方法应按照如下方式定义。利用垃圾回收机制时不需要自动释放池，只需启动垃圾回收器即可。另外，@catch 后的括号内并不表示省略的意思，而是就应该写成“...”（见 18.3 节）。

```
- (void)main {
    @try {
        @autoreleasepool {
            // 在这里书写任务中需要进行的处理
        }
    }
    @catch(...) {
        // 再次抛出异常，不能使用 @throw 等向外部传播
    }
}
```

NSOperation 子类可以根据需要定义初始化器，由于 NSOperation 类指定的初始化器为init，因此子类的初始化器一定会调用init。

NSOperationQueue 的使用方法也很简单。创建实例后，将操作对象加入到队列中即可。NSOperationQueue 接口也在头文件 NSOperation.h 中定义。初始化器使用init。

可以使用下面的方法将操作对象添加到队列。

```
- (void) addOperation: (NSOperation *) operation
```

使用引用计数方式（手动或 ARC）进行内存管理时，程序会一直保持着添加到队列中的操作对象。使用垃圾回收时，因为队列需要访问操作对象，所以直到执行终止操作对象才会被释放。

创建的队列即使被多个线程同时操作也不会出任何问题，因此并不需要为互斥加锁，或者用 @synchronized 块来保护。

一个程序中可以创建多个队列。但是，一个操作对象一次只能加入到一个队列中。而且，已执行完或正在执行的操作对象都不可以加入到队列中。否则就会引发异常。

操作对象一旦加入到队列，就不能从队列中消除。如果不想执行该任务，则可以取消该处理。

稍后会说明取消的方法。

19.3.4 等待至聚合任务终止

虽然无法知道任务按照什么样的顺序执行，但在队列中添加的操作全部完成之前的这段时间，通过使用 NSOperationQueue 的下述方法，就可以一直锁住当前线程（使之等待）。这样一来，在聚合任务终止后就可以简单地进行下一个任务。

- (void) **waitUntilAllOperationsAreFinished**

到接收器队列中添加的操作全部执行完为止，一直加锁调用该函数的线程。队列为空时，没有要执行的任务，该方法立即返回。

调用该方法的线程在被锁期间，接收器队列会继续运行。其他线程也可以添加操作。

- (void) **addOperations:** (NSArray *) ops
  waitUntilFinished: (BOOL) wait

将数组 ops 中的操作对象添加到接收器队列中。参数 wait 为 YES 时，到指定操作执行完为止，调用该方法的线程一直处于被锁状态。

使用引用计数管理方式时，添加的操作对象被保存在队列中，而数组 ops 则不被保存。

19.3.5 使用操作对象的简单例子

代码清单 19-1 中展示了一个简单的程序例子（使用 ARC）。该程序中定义了操作对象 MyOperation 类。任务内容是等待一个随机时间将任务编号添加到 MyList 数组中。线程执行各任务时，会分别等待不同的时间，因此 MyList 中的任务编号理应不会按顺序排列。main 函数会创建 Tasks 个操作对象并将其按顺序添加到队列中。

队列保存操作对象，直到任务终止后操作对象才被释放。在dealloc方法中，尝试使用 NSLog 来输出信息。

而且，可以使用waitUntilAllOperationsAreFinished方法来查看是否所有线程都终止了，但此时操作对象会被同时释放，所以很难知道任务终止的顺序。于是，这里设置为了等待 2.0 秒（在某些环境下还需要等待更长时间，否则所有任务或许就不能都结束）。

代码清单 19-1 NSOperation 的使用示例（opr1.m）

```
#import <Foundation/Foundation.h>  // 使用 ARC
#import <stdio.h>
#import <stdlib.h>
#import <unistd.h>

NSMutableArray *MyList;

@interface MyOperation : NSOperation
{
    int number;                // 各任务号
```

```
    NSTimeInterval interval;  // 时间间隔
}
- (id)initWithNum:(int)sn;
- (void)main;
@end

@implementation MyOperation

- (id)initWithNum:(int)sn
{
    if ((self = [super init]) != nil) {
        number = sn;
        interval = (double)(random() & 0x7f) / 256.0;
    }                          // 生成少于0.5秒的随机值
    return self;
}

- (void)dealloc { // 释放时打印信息
    NSLog(@"Release: %d", number);
}

- (void)main {
    @try {
        @autoreleasepool {
            NSNumber *obj = [NSNumber numberWithInt:number];
            [NSThread sleepForTimeInterval: interval]; // 等待一会儿
            @synchronized(MyList) {
                [MyList addObject:obj];
            }
        }
    }
    @catch(...) { /* 只捕捉异常，不做任何操作 */ }
}

@end

int main(void)
{
    const int Tasks = 10;
    srandom((unsigned)getpid());  // 随机数种子

    @autoreleasepool {
        int i;
        NSOperationQueue *queue = [[NSOperationQueue alloc] init];
        MyList = [[NSMutableArray alloc] init];

        for (i = 0; i < Tasks; i++) {  // 创建操作并将其添加到队列中
            NSOperation *opr = [[MyOperation alloc] initWithNum:i];
            [queue addOperation:opr];
        }
        [NSThread sleepForTimeInterval: 2.0];  // 等待终止
        for (id obj in MyList)
            printf(" %d", [obj intValue]);
        printf("\n");
    }

    return 0;
}
```

该程序每次执行时结果都不一样，例如，可以得到下面的结果（包括 NSLog() 打印的开始部分）。

```
2011-10-06 12:15:12.087 oprl[27521:2603] Release: 5
2011-10-06 12:15:12.122 oprl[27521:2603] Release: 1
2011-10-06 12:15:12.156 oprl[27521:1c03] Release: 4
2011-10-06 12:15:12.185 oprl[27521:2603] Release: 6
2011-10-06 12:15:12.205 oprl[27521:1c03] Release: 7
2011-10-06 12:15:12.245 oprl[27521:2603] Release: 2
2011-10-06 12:15:12.281 oprl[27521:1d03] Release: 0
2011-10-06 12:15:12.295 oprl[27521:2603] Release: 8
2011-10-06 12:15:12.334 oprl[27521:1f03] Release: 3
2011-10-06 12:15:12.377 oprl[27521:1d03] Release: 9
 5 1 4 6 7 2 0 8 3 9
```

最后一行为 main 函数的输出结果。可以看出，基本上是按照任务在数组中加入号的顺序来释放的操作对象。另外，在 NSLog 的输出中，括号内的数字为进程 id 和线程号，可见，不仅有多个线程被创建，有些线程还承担着多个任务的处理。

19.3.6 NSInvocationOperation 的使用方法

在 Cocoa 框架中，NSOperation 的子类包括 NSInvocationOperation 和 NSBlockOperation。使用这些类时，即使不定义子类也能创建操作对象。

这里先简单介绍一下 NSInvocationOperation。

NSInvocationOperation 使用如下初始化器，返回向目标对象发送消息的任务操作对象。当任务内容已经被作为方法定义时，该方法有效。

```
- (id) initWithTarget: (id) target
              selector: (SEL) sel
                object: (id) arg
```

初始化实例，使参数选择器sel向target发送消息。可以指定包含一个参数的选择器，并指定参数arg。选择器没有包含参数时，向arg传入nil。

使用引用计数管理方式时，target和arg会通过调用初始化器被保存，并在接收器的操作被释放时被release。而且，该操作会在创建自动释放池后执行任务。

19.3.7 NSBlockOperation 的使用方法

NSBlockOperation 使用块对象构造任务。使用下面的类方法可以创建临时实例。

```
+ (id) blockOperationWithBlock: (void (^)(void)) block
```

返回以参数中指定的块对象为任务的临时操作对象。参数块对象被生成复制，并在操作对象内保存。使用引用计数管理方式时，该操作会在创建自动释放池后执行任务。

使用该方法，可以将代码清单 19-1 的程序修正为代码清单 19-2 的程序，这里只显示了对应的循

环部分。如果任务很简单，使用 NSBlockOperation 和块对象就可以紧凑地书写。另外，虽然在循环中使用了块对象，但由于方法blockOperationWithBlock:会生成参数的复制，因此包含各种不同值的块对象就会被生成。

代码清单 19-2 NSBlockOperaion 的使用示例（opr2.m 的一部分）

```
for (i = 0; i < Tasks; i++) {
    NSBlockOperation *opr;
    NSTimeInterval interval = (double)(random() & 0x7f) / 256.0;
    opr = [NSBlockOperation blockOperationWithBlock: ^{
        NSNumber *obj = [NSNumber numberWithInt:i];
        [NSThread sleepForTimeInterval: interval];
        @synchronized(MyList) {
            [MyList addObject:obj];
        }
    }];
    [queue addOperation:opr];
}
[queue waitUntilAllOperationsAreFinished]; // 等待终止
```

由于没有包含类似于代码清单 19-1 的dealloc方法的部分，因此，执行结果如下所示，只有一行输出。

```
4 8 1 7 3 9 0 2 5 6
```

然而，通过使用 NSOperationQueue 的addOperationWithBlock:方法，就可以在一次调用中完成复制块对象、创建操作对象、并将其追加到队列中这一系列操作。

19.3.8 NSBlockOperation 中添加多个块对象

NSBlockOperation 中包含数组对象，里面保存着多个块对象，而且这些块对象可以被当作任务执行。当有多个块对象时（如果可能的话），这些块对象就会被并行执行。下面是与数组相关的方法。

- (void) **addExecutionBlock:** (void (^)(void)) block

  将参数块追加到接收器数组中。

- (NSArray *) **executionBlocks**

  返回接收器包含的块对象数组。

NSBlockOperation 是 NSOperation 的子类，可以使用init初始化。初始化后，块对象数组中没有任何元素。

19.3.9 设置任务间的依赖

当存在多个任务时，有时我们会需要指定一定的顺序关系，在执行某个任务后继续执行另一个

任务。比如，当其他任务需要使用某个任务的处理结果时，或绘图处理的前景和背景有关联时，都需要这样的顺序关系。当然，如果所有任务的处理顺序已经确定，那也就没有并行处理的必要了。在多任务中，某些任务有依赖关系，而其他任务可以并行处理，这也是有效的。

对某个操作对象 A 来说，可以设定比 A 更靠前执行的操作对象 B。因此，可以表示为任务 A 依赖于任务 B。在队列中加入的多个操作对象中，可以执行的操作被一个接一个地处理，而有依赖关系的情况下，则在前者处理结束后再执行后面的操作。

然而，如果有多个队列，也可以将有依赖关系的操作分别加入多个队列中。

NSOperation 中定义的设置依赖关系及解除关系的方法如下。

- (void) **addDependency:** (NSOperation *) operation

  将接收器操作依赖的，也就是比接收器先执行的操作指定为参数。

- (void) **removeDependency:** (NSOperation *) operation

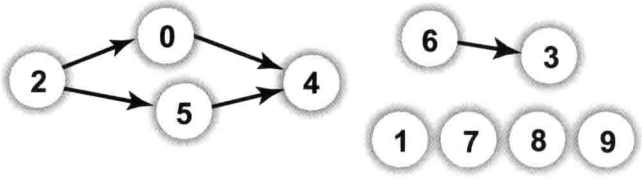
  从接收器中删除注册的依赖操作。

- (NSArray *) **dependencies**

  创建新数组，接收器保存依赖操作并返回。数组中也包括已经终止的操作。不包含依赖操作时，返回空数组。

修改代码清单 19-1 的程序，将 10 个操作设定为如图 19-4 所示的依赖关系。在该图中，例如，操作 2 必须先于操作 0 执行，这些都使用箭头表示了出来。操作 4 必须在操作 0 和操作 5 都执行完毕后才可以执行。操作 1、7、8、9 之间没有依赖关系。

图 19-4 操作间的依赖关系



该依赖关系中，箭头不能形成闭环。也就是说，可以沿箭头指向的方向返回到原点的依赖关系是被禁止的。闭环关系中的所有操作都不能执行，因此处理也不能终止。

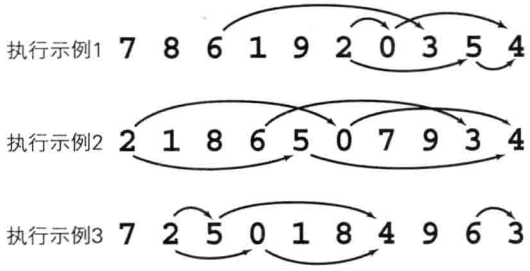
代码清单 19-3 展示了代码清单 19-1 中发生改变的部分代码。

代码清单 19-3 设置操作中的依赖关系并执行的例子（opr4.m 一部分）

```
NSOperation *oprs[Tasks];
for (i = 0; i < Tasks; i++)
    oprs[i] = [[MyOperation alloc] initWithNum:i];
[oprs[0] addDependency:oprs[2]]; // 操作 0 在操作 2 之后
[oprs[5] addDependency:oprs[2]];
[oprs[4] addDependency:oprs[0]];
[oprs[4] addDependency:oprs[5]];
[oprs[3] addDependency:oprs[6]];
for (i = 0; i < Tasks; i++)
    [queue addOperation:oprs[i]];
[NSThread sleepForTimeInterval: 3.0];
```

图 19-5 中显示了执行结果。因为使用了随机数，所以每次的执行结果都不一样。这里提取了 3 次执行结果，设置了依赖关系的操作的执行顺序如箭头所示。可以看出，有依赖关系的两个操作间还有其他操作被执行，但是无论如何，所指定的执行顺序都没有变。

图 19-5 程序的执行结果



19.3.10 任务的优先级设置

在多个任务中，既有需要比其他任务优先执行的任务，也有非优先执行的任务，很多时候我们都会需要进行这样的设定。此时可以使用 NSOperation 提供的优先级的设定方法。

但是，并不能保证高优先级的任务一定会先于低优先级的任务执行。

- (void) **setQueuePriority:** (NSOperationQueuePriority) priority

为接收器的操作指定优先级。

- (NSOperationQueuePriority) **queuePriority**

返回接收器的优先级。

这里，表示优先级的 NSOperationQueuePriority 为整数类型，使用按如下方式定义的常数。即使指定其他整数值，也会被修改为最接近的常数，因此是无效的。

```
enum {
    NSOperationQueuePriorityVeryLow  = -8,    // 最低优先级
    NSOperationQueuePriorityLow      = -4,
    NSOperationQueuePriorityNormal   = 0,     // 一般优先级
    NSOperationQueuePriorityHigh     = 4,
    NSOperationQueuePriorityVeryHigh = 8      // 最高优先级
};
typedef NSInteger NSOperationQueuePriority;
```

19.3.11 设定最大并行任务数

操作队列可以同时启动多个任务的情况下，可以设定最多可以并行执行的任务数。NSOperationQueue 提供了下面的方法。

- (void) **setMaxConcurrentOperationCount:** (NSInteger) count

设定可以并行执行的最大操作数。

- (NSInteger) **maxConcurrentOperationCount**

返回可以并行执行的最大操作数。

一般情况下，任务的并行执行可以根据系统状况判断。选择该状态时，指定常数值 NSOperation QueueDefaultMaxConcurrentOperationCount。该值为确定值（实际值为 -1）。

即使指定了最小值为 1，也不一定会按照加入到队列时的顺序来处理任务。而且，指定较大数字也不会使执行更高速。

19.3.12 终止任务

下面向大家介绍一下如何取消队列中的操作对象的任务。NSOperation 有如下方法。

- (void) **cancel**

使下面的方法isCancelled返回YES。

- (BOOL) **isCancelled**

接收上面的cancel方法后返回YES。否则则返回NO。任务正在执行时或执行终止后也可以返回YES。

而且，NSOperationQueue 类中也有如下方法。

- (void) **cancelAllOperations**

此时向队列中的所有操作对象发送 cancel 消息。

操作对象即使接收cancel，也并不代表插入或异常这样的特殊处理会被执行。

isCancelled在任务执行前接收cancel并返回 YES 时，操作对象将被解除执行。

在任务执行的过程中接收到cancel时，为了转移到中断处理，任务内容必须按如下方式编程实现。具体来说，程序在main方法的开头以及在处理途中的各个重要地方检查自己的isCancelled值，如果返回 YES 则进行中断处理。

19.3.13 设置队列调度为中断状态

也可以临时停止某个操作队列的执行，来停止执行队列中的操作。停止的队列也可以再恢复。中断和解除的方法在类 NSOperationQueue 中定义。

- (void) **setSuspended:** (BOOL) suspend

将接收器的队列变为中断状态或者解除队列的行为。参数指定为YES时是中断状态。进入中断状态后，停止从新的队列中取出操作并执行。

即使为中断状态，也可以向队列中添加操作。而且即使处于中断状态，也不会停止正在执行的任务。

- (BOOL) **isSuspended**

接收器队列如果处于中断状态则返回YES,否则则返回NO。

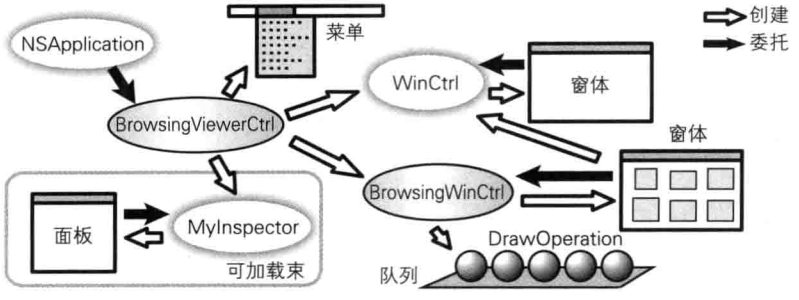
19.4 并行处理的示例程序

19.4.1 程序概要

这里将介绍在程序中使用操作对象的例子。

我们在第 17 章中创建了简单的图像视图，其中增加了一览显示指定目录中包含的图像文件的功能。双击一览显示的小图像，就可以显示出普通大小的图像。原来的图像视图的部分保持不变。这里，将新应用的名字定为 BrowsingViewer，使用 ARC 来管理内存。

图 19-6 新增对象的关系



新对象的关系如图 19-6 所示。这里没有像以往那样放置 MyViewerCtrl 的实例，而是放置了它的子类 BrowsingViewerCtrl。该对象可以指定希望一览显示的目录，并创建 BrowsingWinCtrl 实例。BrowsingWinCtrl 实例会显示一个窗体，来显示图像文件一览。为显示图像，操作对象使用 DrawOperation 类的实例。其他方面，main 函数或信息文件等本地化资源中，有些部分必须改变。

下面仅仅展示了其中的部分源码，这是因为有关窗体和图像显示的部分大都涉及尺寸和坐标计算，和本例的并行处理没有直接关系。

19.4.2 类 BrowsingViewerCtrl

代码清单 19-4 是类 BrowsingViewerCtrl 的实例，该类也是 MyViewerCtrl 的子类，因此只是新增了方法`openDirectory:`。在指定目录时，该方法会显示打开面板。

代码清单 19-4 BrowsingViewerCtrl 的接口部分（BrowsingViewerCtrl.h）

```
#import "MyViewerCtrl.h"

@interface BrowsingViewerCtrl : MyViewerCtrl
- (void)openDirectory:(id)sender;
@end
```

代码清单 19-5 为类 BrowsingViewerCtrl 的实现部分。显示打开面板之前的内容都比较简单，记述关闭面板后的处理的块文法却很长。

在块中，使用类 NSFileManager 将表示指定目录内的文件的 URL 对象保存在数组 files 中。由于数组 types 中加入了系统可处理的图像文件的后缀名，因此，利用这一点，就可以只将数组 files 内包含图像文件后缀名的 URL 放入到可变数组 images 中。最后，创建类 BrowsingWinCtrl 实例并传入数组。BrowsingWinCtrl 是一览显示图像的窗体管理类。

代码清单 19-5 BrowsingViewerCtrl 的实现部分（BrowsingViewerCtrl.m）

```
#import "BrowsingViewerCtrl.h"
#import <Cocoa/Cocoa.h>
#import "BrowsingWinCtrl.h"

@implementation BrowsingViewerCtrl

- (void)openDirectory:(id)sender
{
    NSOpenPanel *oPanel = [NSOpenPanel openPanel];  // 打开面板
    [oPanel setAllowsMultipleSelection:NO];  // 不能选择多个文件
    [oPanel setCanChooseDirectories:YES];    // 可以选择目录
    [oPanel beginWithCompletionHandler: ^(NSInteger result) {
        NSArray *files, *types;
        if (result != NSFileHandlingPanelOKButton)
            return;
        files = [[NSFileManager defaultManager]
            contentsOfDirectoryAtURL:[oPanel URL]
            includingPropertiesForKeys:NULL
            options:NSDirectoryEnumerationSkipsHiddenFiles
            error:NULL]; // 将选中目录中的文件的 URL 放入数组中
        types = [NSImage imageFileTypes]; // 图像文件后缀名
        NSMutableArray *images = [NSMutableArray array];
        for (NSURL *url in files) {
            if ([types containsObject:[url pathExtension]])
                [images addObject:url];
        } // 只将包含图像文件后缀名的 URL 保存在数组 images 中
        if ([images count] > 0)
            (void)[[BrowsingWinCtrl alloc] initWithURLs:images];
    }];
}

@end
```

19.4.3 类 BrowsingWinCtrl

类 BrowsingWinCtrl 的接口部分如代码清单 19-6 所示。只增加初始化器，并指定窗体委托的协议。

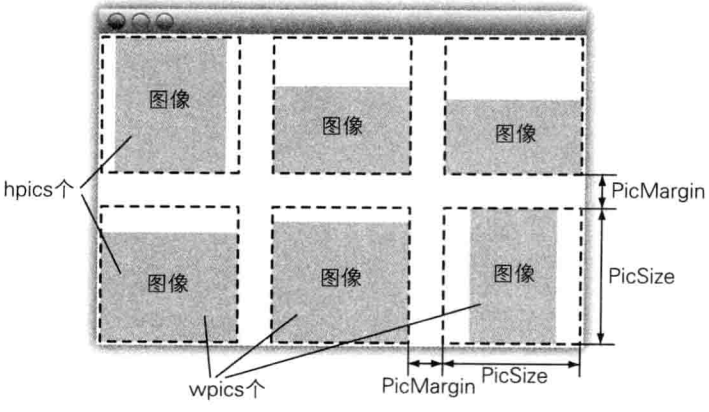
代码清单 19-6 BrowsingWinCtrl 的接口部分（BrowsingWinCtrl.h）

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

@interface BrowsingWinCtrl : NSObject <NSWindowDelegate>
- (id)initWithURLs:(NSArray *)content;
@end
```

代码清单 19-7 是实现部分。所有实例变量都在这里。由于图像的显示使用 NSOperation，因此这里包含了队列实例变量。配置图像的视图使用 docview 来访问，但由于是 window 内部包含的组件，因此使用弱引用。

类 BrowsingWinCtrl 在所管理的窗口中配置图像的情形如图 19-7 所示。图像缩小、横竖大小都需要在 PicSize 范围内。图像仅以 PicMargin 为间隔排列。PicSize、PicMargin 是在 BrowsingWinCtrl.m 中定义的宏。hpics 为窗体中显示的行数，wpics 为每行的图像数，图像较多的情况下可以使用滚动条来查看全体图像。此时全体行数为 rows。从应该显示的图像个数，到横向纵向各排列多少个图像，都由方法`fixDimensions:`决定。

图 19-7 用于一览显示的窗体



方法`showImages`创建用于显示各图像的操作对象，然后将其加入到队列中。操作对象被定义为类 DrawOperation，初始化时传入文件 URL、显示区域（视图）、显示位置的坐标。

方法`initWithURLs:`为初始化器，参数为保存图像文件 URL 的数组。根据该数组的元素个数决定窗体的尺寸，并设定绘图用的视图后显示窗体。最后使用操作对象显示图像。

类定义中的一大部分都被用在了计算窗体尺寸和图像的配置方法。

▶ 代码清单 19-7 BrowsingWinCtrl 的实现部分（BrowsingWinCtrl.m）

```
#import "BrowsingWinCtrl.h"
#import "BrowsingViewerCtrl.h"
#import "DrawOperation.h"

#define PicSize         200
#define PicMargin       6
#define PreviewSize     (PicSize + PicMargin)
#define ScreenMargin    48

@implementation BrowsingWinCtrl {
    NSArray    *imageURLs;          // 文件 URL
    NSWindow   *window;             // 用于一览显示的窗体
    __weak NSView     *docview;     // 图像配置视图
    NSOperationQueue *queue;        // 操作队列
    int        wpics, hpics, rows;  // 横向和纵向的图像个数
}

/* Local Method: 决定横向和纵向的图像个数 */
- (void)fixDimensions:(int)count
{
    /* 省略详细内容。窗体中纵向的图像数为 hpics，横向的图像数为 wpics。
       使用滚动条时，全体行数为 rows */
}

/* Local Method: 创建操作对象来显示图像 */
- (void)showImages
{
    int i, count;
    int x, y;
    NSPoint loc;          // 各个图像左下角的坐标
    CGFloat hgt;          // 表示域的纵向长度
    DrawOperation *op;

    count = [imageURLs count];           // 图像文件的个数
    hgt = [docview frame].size.height;   // 表示域的纵向长度
    for (i = 0; i < count; i++) {
        y = i / wpics;
        x = i - (y * wpics);
        loc.x = x * PreviewSize;
        loc.y = hgt - y * PreviewSize - PicSize;  // 左下角为原点
        op = [[DrawOperation alloc]
                initWithURL:[imageURLs objectAtIndex:i]
                parentView:docview at:loc];
        [queue addOperation:op];
    }
}

- (id)initWithURLs:(NSArray *)content
{
    NSScrollView *scview;
    NSClipView *clip;
    NSString *pathname;
    BOOL withScroller;
```

```
    NSRect contrect, viewrect;
    NSUInteger wstyle = (NSTitledWindowMask |
        NSClosableWindowMask | NSMiniaturizableWindowMask);

    if ((self = [super init]) == nil)
        return nil;
    [DrawOperation setPictureSize:PicSize];     // 设置图像大小
    queue = [[NSOperationQueue alloc] init];    // 初始化队列
    [queue setMaxConcurrentOperationCount:10]; // 设置最大数
    imageURLs = content;                        // 保存 URL 的数组
    [self fixDimensions:[imageURLs count]];     // 确定纵向的图像数
    /*
        省略。确定窗体和绘图区域的大小，初始化设定。
    */
    [window makeKeyAndOrderFront:self];         // 显示窗体
    [[BrowsingViewerCtrl sharedController] addWinCtrl:self];
    [self showImages]; // 使用操作对象显示图像
    return self;
}

/* Delegate Messages */
- (BOOL)windowShouldClose:(id)sender
{
    [window setDelegate:nil];      // 解除委托
    window = nil;                  // 放弃附属关系
    [queue cancelAllOperations];   // 停止操作队列
    [[BrowsingViewerCtrl sharedController] removeWinCtrl:self];
    return YES;
}

@end
```

19.4.4 类 DrawOperation

代码清单 19-8、19-9 中的类 DrawOperation 为操作对象，用于将给定 URL 的图像缩小，并将其绘制在视图的指定位置。该动作可以并行执行。

视图可以进行有层次地重叠配置，某个视图中包含的其他视图称为子视图。在该程序中，并不是直接在大的视图中绘图，而是在其他小的子视图中绘图。为此，代码清单 19-9 中定义了只在该文件内使用的类 ClickableImageView。该类继承了 NSImageView，拥有显示图像的功能。而且可以获得鼠标点击事件，双击时则创建第 17 章中的 WinCtrl 类实例，将图像显示为普通大小。

在类 DrawOperation 的方法main中，从 URL 创建图像，将其设定为 ClickableImageView 的实例，并在指定位置配置子视图。如果图像尺寸太大，则缩小图片，使其不超过绘图范围，这里不再详细说明。

代码清单 19-8 DrawOperation 的接口部分（DrawOperation.h）

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

@interface DrawOperation : NSOperation
{
    NSURL          *file;    // 图像文件的 URL
    __weak NSView  *view;    // 没必要保存绘图用的视图
    NSPoint  location;       // 绘图区域左下角
}
+ (void)setPictureSize:(CGFloat)value;
- (id)initWithURL:(NSURL *)imgurl
    parentView:(NSView *)parent at:(NSPoint)loc;
- (void)main;
@end
```

代码清单 19-9 DrawOperation 的实现部分（DrawOperation.m）

```
#import <Cocoa/Cocoa.h>
#import "DrawOperation.h"
#import "WinCtrl.h"

@interface ClickableImageView : NSImageView  // 定义局部类
@property (retain) NSURL *url;
@end

@implementation ClickableImageView
@synthesize url;

- (void)mouseDown:(NSEvent *)theEvent
{
    if ([theEvent clickCount] >= 2)  // 双击显示图像
        (void)[[WinCtrl alloc] initWithURL:url];
}
@end //ClickableImageView 的定义到此为止

static CGFloat pictureSize = 200.0;

@implementation DrawOperation // DrawOperation 类主体的定义

+ (void)setPictureSize:(CGFloat)value {
    pictureSize = value;
}

- (id)initWithURL:(NSURL *)imgurl
    parentView:(NSView *)parent at:(NSPoint)loc
{
    if ((self = [super init]) != nil) {
        file = imgurl;  // 保存 URL
        view = parent;  // 弱访问
        location = loc;
    }
    return self;
}
```

```
/* Local Method */
- (NSImage *)imageResized:(NSURL *)url
{ /* 省略。从 URL 中读取图像，如果图片大于 pictureSize，则创建缩小版图片 */
}

- (void)main
{
    NSImage *img;
    ClickableImageView *imgview;
    NSRect frame;
    NSSize sz;

    @try {
        @autoreleasepool {
            if (view == nil || [self isCancelled])
                return; // 被取消时程序终止
            if ((img = [self imageResized:file]) == nil)
                return;
            sz = [img size];
            frame.size = sz;
            location.x += (pictureSize - sz.width) / 2.0;
            frame.origin = location;
            imgview = [[ClickableImageView alloc] initWithFrame:frame];
            [imgview setImage:img];      // 创建小视图来设定图像
            imgview.url = file;
            [view addSubview: imgview]; // 子视图
            [imgview setNeedsDisplay:YES];
        }
    }
    @catch(...) { }
}

@end
```

19.4.5 其他改变

代码清单 19-10 中展示了新的 main 函数。这里只是将第 17 章的程序中使用 MyViewerCtrl 类的部分置换为了 BrowsingViewerCtrl 类。

代码清单 19-10 新的 main 函数 (main.m)

```
#import <Cocoa/Cocoa.h>
#import "BrowsingViewerCtrl.h"

int main(int argc, const char *argv[]) {
    @autoreleasepool {
        BrowsingViewerCtrl *cn = [BrowsingViewerCtrl sharedController];
        NSApplication *app = [NSApplication sharedApplication];
        [app setDelegate: cn];
        [app run];
    }
    return 0;
}
```

因为改变了应用名，所以 Info.plist 的应用识别名、束名、执行文件名都需要改变。此外，在描述菜单信息的 Menu.plist 文件中添加“打开目录”菜单项。然后，将新加的源码进行编译并添加到链接文件，这样就可以执行了。

图 19-8 一览显示的例子

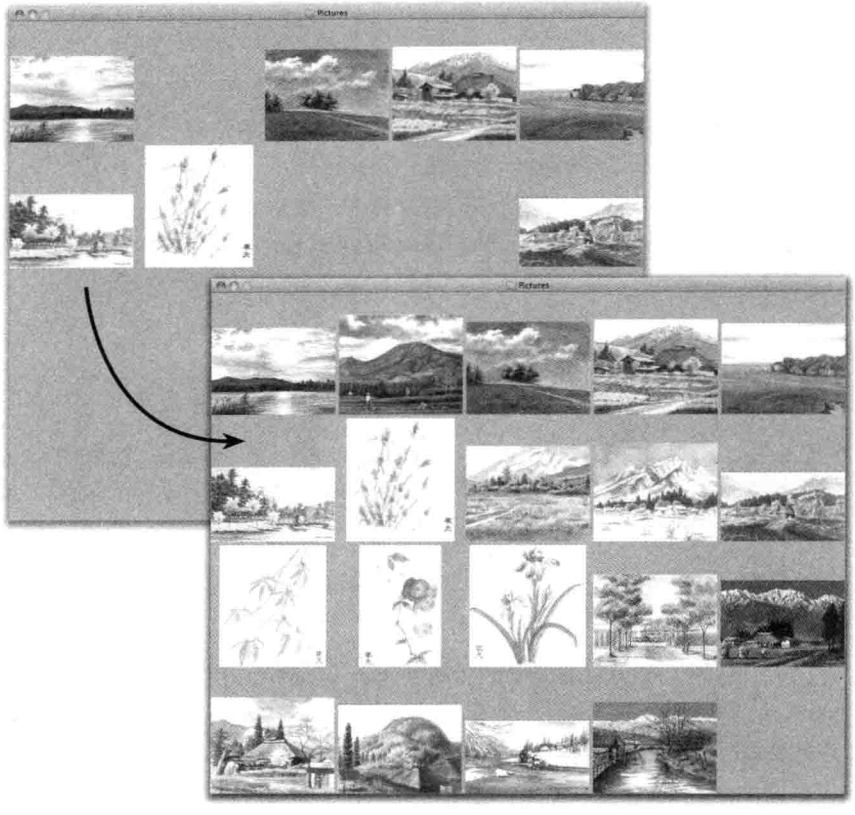


图 19-8 为执行示例。显示一览时，先是依次显示可以显示的图像，然后形成最终的一览界面。双击小图标显示的图像，即可将对应的图像显示为普通大小。

19.5 使用连接的通信

在 Mac OS X 的 Foundation 框架中，为了使不同的线程或进程可以双工通信，提供了类NSConnection。NSConnection 对象除了被作为线程间线程安全的通信线路使用外，也提供了创建应用间所使用的**分布式对象**（distributed object）的方法。

而 iOS 中则不提供该方法。

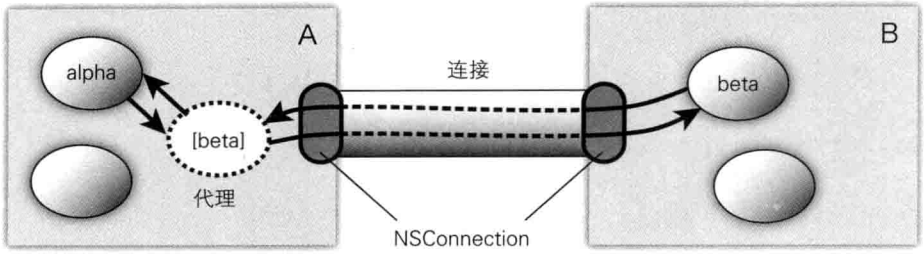
根据苹果公司的最新文档，当创建新应用时，如果希望使用线程来提高效率，可使用上一节讲述的操作对象或 GCD。在 Mac OS X 上的应用间进行通信时，或者在任务明确的线程间通信时，该方法非常有用。

19.5.1 连接

NSConnection 对象在进行通信的线程或进程间创建，并成对使用。在进程中设置的 NSConnection 对象和在线程间设置的 NSConnection 对象的创建方法是不同的，但它们在通信中的概念是相通的。由 NSConnection 对象创建的通信线路称为**连接**（connect）。

图 19-9 中，A 和 B 表示不同的进程地址空间或不同线程管理的对象群。假设要从 A 中包含的对象 alpha 向 B 中包含的对象 beta 发送消息。这时，可以在 A 和 B 间用 NSConnection 对象设置连接，并在 A 方创建对象 beta 的行为“代理人”“beta”对象。

图 19-9 NSConnection 对象的连接概念



对象 alpha 要发送消息给 beta 时，会首先将消息发送给代理人“beta”。于是，该消息就沿着连接被传送到 B 侧，由对象 beta 接收。执行方法后，该消息会再次经由连接被传送到 A 侧，并由代理人“beta”返回执行结果到 alpha。就好像是对象 alpha 和代理人“beta”之间在进行消息互换。

这里需要注意的是，对象 alpha 和“beta”在 A 的进程（或线程）中执行，对象 beta 在 B 的进程（或线程）中执行。而即使进程不同，也可以通过发送消息来互换信息。而且，线程间通信时，因为不同线程管理的对象是不可以由其他线程直接操作的，所以就可以使用线程安全的方法进行通信。

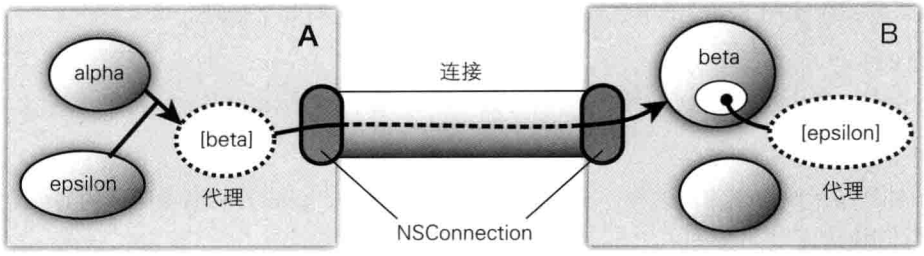
NSConnection 对象创建的连接包含消息等待队列的功能。当接收者正在进行其他作业而不能接收消息时，被发送的消息就会在等待队列中排队等待处理。

19.5.2 代理

在连接的另一侧实现实体对象代理人功能的对象称为**代理**（proxy）。上例中，“beta”为对象 beta 的代理。代理并不是实体对象的副本，而是经由连接发送消息时行为与实体对象相同的对象。

将对象作为消息参数或结果经由连接传递时，并不是向另一侧发送给实体对象，而是创建新的代理，并传递给它。

图 19-10 由消息传入的对象和代理



例如，图 19-10 中，为了设定 B 侧对象 beta 的实例变量 gamma，假定我们有方法`setGamma:`。A 侧对象 alpha 指定 A 侧对象 epsilon 为`setGamma:`方法的参数并发送消息。这时，B 侧将新创建 epsilon 的代理“epsilon”，并将其作为`setGamma:`方法的参数传入。之后，在 B 侧向“epsilon”发送消息时，消息就会沿着连接被发送给 A 侧的 epsilon。

因此，一旦设定了连接，就可以经由代理进行简单的通信。但是，使用连接接收或发送消息时，首先必须确保有对象接收连接传递的消息。在上例中，对象 beta 及其代理“beta”如果不存在，A 和 B 之间就不能进行通信，也不能设定新的代理。

所以，NSConnection 对象在设定连接时，必须要指定经由该连接接收消息的对象。该对象就称为**根对象**（root object）。设定根对象后，就可以自动生成所对应的代理。

在 Foundation 框架中，代理用 NSProxy 类表示。而实际通信中则经常使用该类的子类 NSDistantObject。

NSProxy 是 Foundation 框架中的又一个根类，且不是 NSObject 的子类。因为它不会自己处理所接收的大部分消息，而是传送给实体对象，因此不需要继承 NSObject。

查看某个对象是否是代理，可以调用下面的方法。

```
- (BOOL) isProxy
```

该方法在 NSObject 协议中声明。NSObject 协议包含了对象的基本行为，NSProxy 也遵守这些行为。

19.5.3 方法的指针参数

使用连接收发消息时，如果在参数或返回值中使用了整数、实数等基本类型或结构体，那么就和普通的方法调用一样，使用副本传递。

如之前所述，传递对象时会向对方传递该对象的代理。而无论使用对象类型的 id 类型，还是使用指定类的静态类型，都同样适用于这一点。

当参数数据类型是对象之外的指针时，需要注意一点。例如，我们来考虑一下下面的方法。

```
- (void)beat:(struct techno *)sound
```

一般情况下，在发送消息时，参数 sound 指向的结构体指针会被发送。但在其他进程中传递指针则没有任何意义。参数中使用了指针的消息通过连接被发送时，会进行如下处理。

（1）发送消息时

运行时系统首先会确保接收方的地址空间，复制参数指针指向的内容，并将其指针传给接收器。

（2）处理终止时

使用指针返回处理结果给发送方。这样一来，当消息处理终止时，接收方的内存区域内容就会被复制到发送方指针指向的位置并返回。

指针不仅可以指向单一的变量，也可以表示数组的开头。但是，程序上无法做任何区别。即使表示数组开头的指针以参数方式传入，因为只复制第一个元素，所以也要注意。

参数中使用了指针的情况下，因为要在传值时和接收时执行两次复制操作，所以消息发送的成本会很大。因此编程人员就要明确目标。此外，为了减少复制步骤，Objective-C 中新增了修饰符 in、out 以及 inout，如表 19-1 所示。

表 19-1 指针参数对应的类型修饰符

修饰符	说明
in	参数只用来向接收器传入信息。
out	参数只用来向发送器返回信息。
inout	参数用来传入或接收信息。

例如，如果只向接收器传递信息，就可以将上述方法按如下方式书写。这样，在处理终止时，就不需要复制值并返回了。

```
- (void)beat:(in struct techno *)sound
```

参数中没有指定 in、out、inout 时，同指定 inout 时的操作一样。但是，const 修饰的参数同附加 in 的参数也是一样的。指针之外的参数也可以使用 in，而 out 和 inout 则对指针之外的参数没有意义。

这些修饰符，在没使用连接的一般消息通信中也可以指定。例如，经由参数来获取错误对象（见 18.5 节）时，参数类型可以指定为 out。

19.5.4 对象的副本传递

使用连接的消息发送中，当参数或返回值中指定了对象时，并不是直接传递该对象，而是创建它的代理并传入。

但是，也可以不创建代理，而是在接收参数或返回值侧的地址空间中创建对象的副本，然后传递该副本。当对象不需要在发送者和接收者之间共享，且使用副本的代价比使用代理小时，该方法的效果比较好。这时，需要指定类型修饰符 bycopy。

例如，将参数对象复制并传入，如下所示。

```
- (void)setCurrentData:(bycopy id)obj
```

然后，返回值对象被复制并接收，如下所示。

```
- (bycopy id)currentData
```

接收对象的副本的一方也可能是其他进程。此时，该程序必须加载被复制传入的对象的类模型。

此外，与 bycope 相反，可以使用类型修饰符 byref 来明确指明接收代理。但是，由于传递代理是默认的行为，因此几乎没有使用 byref 的情况。

19.5.5 异步通信

一般的消息发送，与函数调用一样，方法处理终止后控制权就会返回给发送方。经由连接发送消息也是一样。只是，不需要返回值时，接收侧不需要等待方法完成就可继续进行下面的处理。这是最有效的方式。

这样的消息称为**异步消息**（asynchronous message）。与此相对，发送方需要一直等待处理终止这样的消息发送方式称为**同步消息**（synchronous message）。

为了指定消息的传递方式为异步，可以使用 oneway 类型修饰符，如下所示。

```
- (oneway void)startNextJob
```

与 oneway 组合使用的只有 void。

19.5.6 设置协议

向代理发送消息时，虽然和实体对象具有相似的行为，但实际上，截至到消息发送为止，我们并不知道相应的方法。向代理发送消息时，运行时系统首先需要询问实体对象，以确定该消息的参数要如何编码发送，然后再实际进行消息的发送。所以每发送一次消息，都要进行两次信息交互。

为避免多余的操作，可以事先将实体对象有什么样的方法告诉给代理。为此，实体对象会将对应的方法集汇总在协议中，并使用如下方法设置代理。这是 NSDistantObject 类的方法。

```
- (void) setProtocolForProxy: (Protocol *) aProtocol
```

该设定执行后，就不再需要询问协议中包含的方法以及参数类型，所以提升了通信效率。关于类型 Protocol，请参考 12.2 节。

19.5.7 运行回路的开始

正如我们在 15.1 节中介绍的那样，NSConnection 等通信线路的输入，由运行回路进行处理。在使用 NSConnection 连接的消息通信中，各种对象都可以发送异步消息。为了能够监视消息以立刻启动相应的处理，各个线程中都必须要启动运行回路。

Cocoa 环境中，各线程自动提供了 NSRunLoop 对象。自己不需要创建 NSRunLoop 的实例。各线程的 NSRunLoop 对象如下所示，可以使用 NSRunLoop 类方法获得。

```
+ (NSRunLoop *) currentRunLoop
```

向该对象发送run方法后，运行回路被启动，如下所示。

```
[[NSRunLoop currentRunLoop] run];
```

这是一个无限循环，不会停下来。想要中断处理时，一般会中断启动该回路的线程。关于运行回路的详细内容，请参照 NSRunLoop 参考或“Threading Programming Guide”等文档。

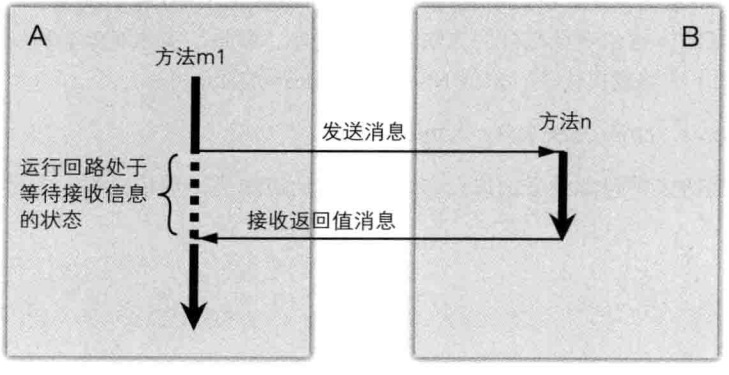
19.5.8 收发消息时的处理

如上所述，在经由连接进行消息收发的线程（或进程）中，运行回路监视着消息的接收。而且，将消息发送给其他线程后，运行回路也会监视该结果被对方返回的过程。

在不执行任何方法，只是等待从外部发送来的消息时，被接收到消息的方法会被立刻执行。那么，在其他线程中经由连接发送消息并等待返回结果时又是怎样的呢？实际上，此时如果有接收到的消息，这些消息也将被执行。

假设从线程 A 管理的对象经由连接向另一个线程 B 管理的对象发送了消息。此时，如果没有其他需要处理的消息，线程 A 的运行回路就会进入消息等待状态，而如果从线程 B 返回了结果，则再次调用方法来处理（图 19-11）。

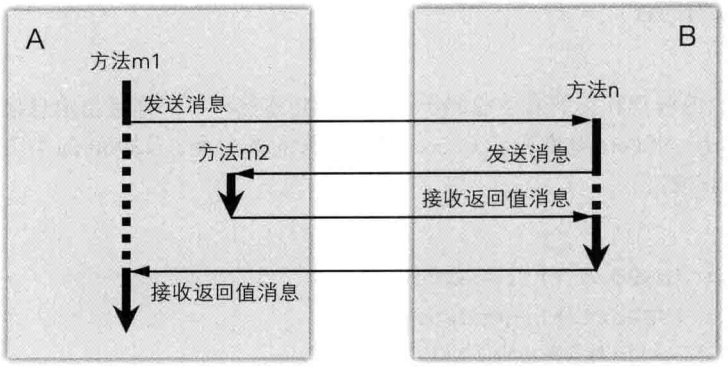
▶ **图 19-11 消息发送・接收的概念图**



线程 B 侧的对象为了处理消息，会向消息发送方线程 A 侧的对象发出询问消息，这是一般的处理流程。此时，假设线程 A 的运行回路直到目前正在执行的方法终止才会处理其他消息，那么，当线程 B 侧发出询问时，两者都将引起死锁并被迫停止。

为了避开这样的事故，即使是等待消息返回期间也要允许其他方法执行（图 19-12）。

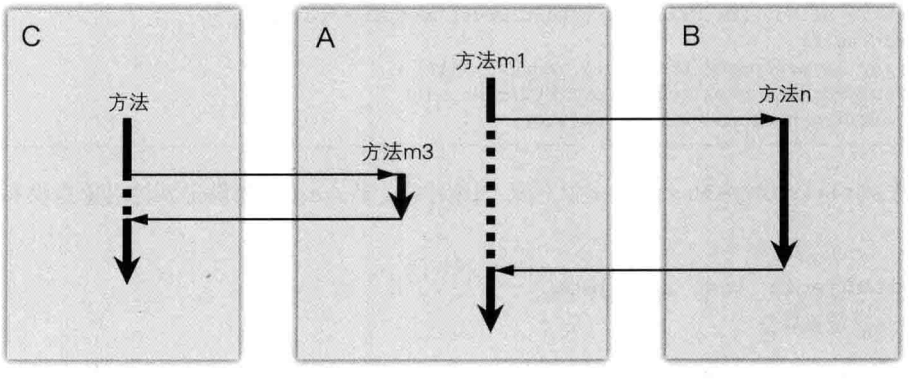
图 19-12 在等待消息返回期间处理其他消息



也可以设置为该处理不能执行[①]，但这时很容易会陷入死锁状态，所以要格外注意。

但是，这种在等待消息返回期间允许其他方法执行的分布式对象的行为，在某些情况下可能会产生不可预期的行为。

图 19-13 消息调用未按预期顺序发生时



如图 19-13，在方法 m1 执行期间，线程 A 向线程 B 发送消息，进入到等待回复的状态。在这期间，假设线程 A 又从线程 C 接收到消息并启动方法 m3 来处理。伴随着方法 m3 的处理，实例变量等数值会发生变化，而这就会影响到方法 m1 的继续执行。

这样一来，就可能发生不按预期顺序执行方法的情况，因此，从多个线程发送异步消息的对象需要密切关注实例变量的修改等。然而，不能靠使用锁来防止这种突然插入的方法。因为执行接收

① 详情请参考 NSConnection 的文档。

的消息的都是同一个线程。

19.5.9 线程间连接

下面说明同一个进程内不同线程间设定 NSConnection 连接的方法。设定方法是固定的，大多数情况下按照这里介绍的方法操作都没有问题。

父线程侧

处理情况如图 19-14 所示，在当前线程和接下来生成的子线程间创建连接。一个连接由消息接收、消息发送两个 NSPort 对象构成。NSPort 类为表示 Cocoa 通信线路的抽象类。NSPort 的类方法port为返回 NSPort 实例的便捷构造器。

连接使用如下方法来初始化。

```
- (id) initWithReceivePort: (NSPort *) receivePort
                  sendPort: (NSPort *) sendPort
```

使用接收消息、发送消息使用的NSPort对象来初始化NSConnection。
便捷构造器：connectionWithReceivePort: sendPort:

图 19-14 父线程侧的处理

```
NSArray *portArray;
NSConnection *conn;
NSPort *port1 = [NSPort port];
NSPort *port2 = [NSPort port];
conn = [[NSConnection alloc] initWithReceivePort:port1 sendPort:port2];
[conn setRootObject:self];
portArray = [NSArray arrayWithObjects:port2, port1, nil];
[NSThread detachNewThreadSelector:@selector(XXXWithPorts:)
        toTarget:XXXTarget withObject:portArray];
```

下面，使用方法setRootObject：来设定根对象。图例中设定了 self，而设定其他对象也没有关系。

```
- (void) setRootObject: (id) anObject
```

连接的对方侧设定根对象。

最后，将收发消息使用的 NSPort 对象互换并保存在数组对象中，然后启动子线程，以执行以此为参数的方法。图中，对象 XXXTarget 执行了方法XXXWithPorts：。

然而，在父线程侧必须要启动运行回路。如果父线程本身是使用了 NSApplication 的主线程，那么运行回路就已经被启动了。其他程序或子线程的情况下，则必须要显式启动 NSRunLoop。

子线程侧

子线程侧的处理情况如图 19-15 所示。

如图所示，使用引用计数管理方式时，首先设定线程使用的自动释放池。接着使用父线程启动

时传入的参数 NSPort 对象来创建 NSConnection 对象。这就是与父线程侧成对使用的连接。收发消息的端口与父线程侧是相反的，这点需要注意。

▶ 图 19-15 子线程侧的处理

```
- (void)XXXWithPorts:(NSArray *)portArray
{
    @autoreleasepool {
        id *actor, *parent;
        NSConnection *conn = [NSConnection
            connectionWithReceivePort:[portArray objectAtIndex:0]
            sendPort:[portArray objectAtIndex:1]];

        parent = [conn rootProxy];                          // (a)
        [parent retain];  // MRC
        [parent setProtocolForProxy:@protocol(XXX)];        // (b)
        actor = [[XXXClass alloc] init];                    // (c)
        [parent setXXXActor: actor];
        [actor release];  // MRC: 手动引用计数管理

        [[NSRunLoop currentRunLoop] run];
    }
    [NSThread exit];
}
```

(a) 中，使用方法rootProxy取得连接的根对象代理。之后，基本上就是对代理执行经由连接的消息发送。图中 parent 为局部变量，也可以是实例变量等等。

```
- (NSDistantObject *) rootProxy
```

(b) 中，设定该代理的协议。这虽然不是必须的，但是为了提高效率应该这么做。

(c) 中，创建新实例，然后将以该实例为参数的消息经连接发送出去，并在连接的另一侧设定自己的代理。依据该操作，对方也能够向这一侧发送消息。如果没有必要，也可以不进行 (c) 的操作。此外，也可以不向对方传递新的实例，而是传递已有对象，但由于线程内生成的局部实例不能被其他线程访问，因此我们比较推荐这样的做法。然而，虽然图例中全都是实例方法，但如果可以创建实例，那么类方法也是可以的。

设定完成后，启动运行回路，这样就可以处理经连接发送的消息了。

图 19-15 中，向根对象侧发送了方法setXXXActor:，如下面的例子所示。参数 obj 中接收的对象为图 19-15(c) 中创建的实例代理。设定协议并持有这些代理，以便之后在通信时使用。使用 ARC 或垃圾回收时，必须保存在实例变量或外部变量中。

```
- (void)setXXXActor:(id)obj
{
    [obj setProtocolForProxy:@protocol(XXXActor)];
    actor = [obj retain];  // 手动引用计数管理时必须保存
}
```

19.5.10 进程间连接

下面介绍一下在不同进程间设定 NSConnection 连接的方法。首先在主机内设定一个包含唯一服务名的连接。称为**有名 NSConnection**。其他进程通过使用服务名也可以创建和该 NSConnection 的连接。使用有名 NSConnection 后，也可以和其他主机运行的进程进行通信，本书中省略对这部分的讲解。下面我们来看看同一个主机内进程间如何通信。

（1）设置有名 NSConnection

图 19-16 展示了设置有名 NSConnection 的例子。在线程内创建新的 NSConnection 实例，并将其设定为有名 NSConnection。然而有一点需要注意的是，进程内必须启动运行回路。

图 19-16 设定有名 NSConnection 的例子

```
NSConnection *conn = [[NSConnection alloc] init];
[conn setRootObject:anXXXserver];
if ([conn registerName:@"TheXXXServer"] == NO) {
    /* 错误处理 */
}
```

处理从外部经由连接发送来的消息时需要用到对象（图例中 anXXXserver），而为了设定有名 NSConnection，就要将该对象设定为连接的根对象并决定连接名，且在**端口名字服务器**中登记。这也称为声明对象 anXXXserver。

端口名字服务器管理着为其他进程提供连接的有名 NSConnection。这里，通过有名 NSConnection 的登记，其他进程就可以根据名字取得连接。在端口名字服务器中登记时使用 NSConnection 的方法`registerName:`。

```
- (BOOL) registerName: (NSString *) name
```

使用默认的端口名字服务器，并使用指定的名字将接收器在本地主机上登记。如果登记成功则返回 YES。如果名字登记失败，则可能是因为其他连接已经使用了该名字。
将参数指定为nil，即可取消登记。

端口名字服务器实际上是 NSPortNameServer 及其子类实例，在同一个主机内的进程间通信时，由默认的名字服务器处理即可，因此不必直接访问该类。

连接中只能指定一个名字。所以当想要提供多个名字的 NSConnection 时，必须要新创建 NSConnection 实例来设定。

（2）确立连接

为了指定名字并获得连接，NSConnection 可以使用下面的类方法。

```
+ (id) connectionWithRegisteredName: (NSString *) name
                               host: (NSString *) hostName
```

使用默认的端口名字服务器，查找并返回指定主机中使用指定名字登记的NSConnection。主机名指定为@""或nil时，检索本地主机内登记的名字。如果找不到则返回nil。

对 NSConnection 实例适用方法rootProxy，取得提供的对象的代理。必要时还需要设定所使用的协议。该步骤与图 19-15 的子线程侧的处理是同样的，大家一看便知。

但是，由于多数情况下即使取得了 NSConnection 也不会使用，因此可以采用从连接名直接得到代理的类方法。

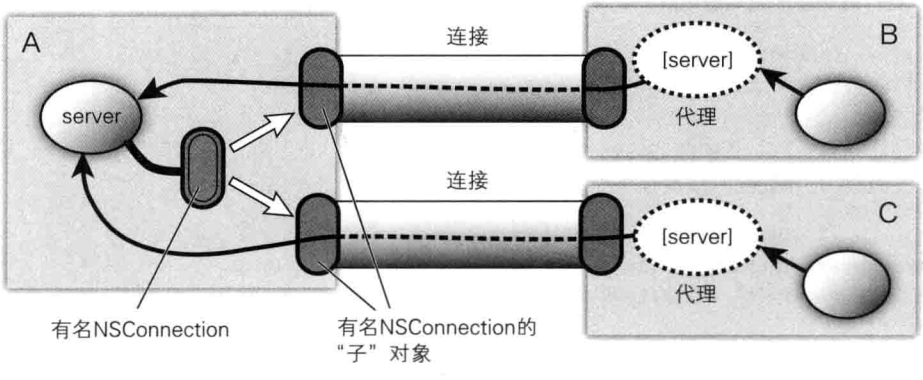
```
+ (NSDistantObject *)
    rootProxyForConnectionWithRegisteredName: (NSString *) name
                                        host: (NSString *) hostName
```

例如，可以像下面这样使用。

```
id theProxy = [NSConnection
        rootProxyForConnectionWithRegisteredName:@"TheXXXServer"
        host:@""];
[theProxy retain];  // 手动内存管理方式时
[theProxy setProtocolForProxy: @protocol(XXXservices)];
```

有名 NSConnection 可以构成多个进程间的连接。需要连接时，登记名字使用的 NSConnection 的“子”NSConnection 会被创建，该对象与各进程侧的 NSConnection 形成一对，这样就形成了连接（图 19-17）。

图 19-17 有名 NSConnection



19.5.11 进程间连接的例子

进程间设定连接是很简单的，如下所示。

这里，我们尝试为第 17 章中介绍的图像视图中，追加了服务器功能。首先，代码清单 19-11 所实现的仅仅是使用服务器端添加的方法接收并显示表示图像文件的 URL。返回值为错误消息，由宏 SERVER 定义使用的服务器名。

代码清单 19-12 为在类 MyViewerCtrl 中增加的范畴。可以在第 17 章的源码部分或前面的改良版中追加。采用代码清单 19-11 的协议。

代码清单 19-11 通信中使用的协议的声明（ShowImage.h）

```
#define SERVER  @"ShowImage"  // 服务器名

@class NSString, NSURL;

@protocol ShowImage
- (NSString *)showImage:(NSURL *)aFile;
@end
```

代码清单 19-12 进行服务器设置的范畴的声明（MyViewerCtrl+Connection.h）

```
#import "MyViewerCtrl.h"
#import "ShowImage.h"

@interface MyViewerCtrl (Connection) <ShowImage>
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification;
@end
```

代码清单 19-13 是新增的源码的主体，很简单。类 MyViewerCtrl 的实例为应用的委托，在启动完成后调用的方法applicationDidFinishLaunching:中设定服务器。因为存在应用自身的运行回路，所以没有必要创建各线程独立的运行回路。指定根对象为 self。

因为方法showImage:会被发送该 self，所以需要事先在同一个文件内定义。读取文件时会创建类 WinCtrl 的实例并显示图像，不能读取时则返回简单的错误消息。

代码清单 19-13 进行服务器的设定的范畴（MyViewerCtrl+Connection.m）

```
import "MyViewerCtrl+Connection.h"
#import <Cocoa/Cocoa.h>
#import "WinCtrl.h"

@implementation MyViewerCtrl (Connection)

- (NSString *)showImage:(NSURL *)aFile
{
    if ([[NSFileManager defaultManager]      // 能够读取文件时
            isReadableFileAtPath:[aFile path]]) {
        (void)[[WinCtrl alloc] initWithURL:aFile];
        return nil;
    }
    return @"Can't read";                    // 不能读取时
}

NSConnection *conn = nil;

/* Delegate Method */
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{   // 应用启动终止时被调用
    conn = [[NSConnection alloc] init];
    [conn setRootObject:self];
    if ([conn registerName:SERVER] == NO) {
        NSLog(@"Can't establish %@ server.\n", SERVER);
    }
}

@end
```

代码清单 19-14 是客户端的程序。这次是简单地从命令行执行的程序。命令行参数需要指定为图像文件的绝对路径。

启动该程序时，首先要启动将上述追加的源码编译后的图像视图。接着，从终端启动该程序。程序指定本地机器内的服务器后，创建和该服务器的连接，该根对象在 theProxy 中取得。不能创建连接时则显示错误。向代理发送的消息被传递给图像视图的对象，然后图像被显示出来。不能读取文件时，向客户端返回错误消息。

该程序和服务器只进行一次通信就终止了，而当程序之间相互通信时，就必须使用线程提供运行回路和自动释放池。

代码清单 19-14 客户端程序（client.m）

```
#import <Cocoa/Cocoa.h>
#import "ShowImage.h"

int main(int argc, const char **argv)
{
    if (argc <= 1)
        return 1; // Error
    @autoreleasepool {
        id theProxy = [NSConnection
            rootProxyForConnectionWithRegisteredName:SERVER host:@""];
        if (theProxy) {
            NSString *s, *r;
            [theProxy setProtocolForProxy: @protocol(ShowImage)];
            s = [NSString stringWithUTF8String:argv[1]];
            r = [theProxy showImage: [NSURL fileURLWithPath:s]];
            if (r) fprintf(stderr, "Message: %s\n", [r UTF8String]);
        }else
            fprintf(stderr, "Error: Can't connect\n");
    }
    return 0;
}
```

第20章

键值编码

所谓键值编码，并不是访问器方法的启动和实例变量的访问这种直接的方式，而是使用表示属性（property）的字符串来间接访问对象属性值的一种结构。如果能够很好地利用该结构，就能创建易读且量少的代码，而且对动态连接对象也很有效。本章将介绍键值编码的概况和 Cocoa 绑定的基础。

20.1 键值编码概况

20.1.1 什么是键值编码

对象状态大多通过实例变量的值来表示，为了访问、改变该值，声明属性或提供访问器方法是一般做法。另外，虽然不推荐，但通过将可视属性设置为 @public 来访问实例变量的方法也是可以的。

与此相对，**键值编码**（key-value coding）[①] 是指，将表示对象包含的信息的字符串作为键值使用，来间接访问该信息的方式。键值编码提供了非常强大的功能，基本上，**只要存在访问器方法、声明属性或实例变量，就可以将其名字指定为字符串来访问**。本章中，可以访问、设定的对象状态的值称为**属性**（property）[②]。

之所以说键值编码的访问是间接的，是因为以下两点。

1. 也可以在运行中确定作为键的字符串

访问器或实例变量在程序内书写时只能访问该程序的属性。例如，假设在程序内有`setName:`，那么在这个地方就只能访问一个属性。与此相对，由于键可以保存在字符串变量中，因此，根据是“name”字符串还是“age”字符串，可以访问不同的属性。

2. 使用者无法知道实际访问属性的方法

如下所述，通过键访问属性的方法有很多种，可以根据每个类的情况选择合适的方法。而且，不管各类中是怎么实现的，每个属性都可以用同样的方法访问，这是它的重要性质。

键值编码提供了非常强大的抽象化功能。

在为了便于创建或改变 GUI 而引入的 Cocoa **绑定技术**中，以及 AppleScript 可操作的应用开发中，键值编码都是一个重要的因素。由于键值编码以字符串为媒介，因此还可以应用在与 GUI 组件的结合或脚本控制等方面。在本章的后半部分，我们将介绍有关键值观察和 Cocoa 绑定的基础知识。

然而，虽然键值编码、键值观察技术在 iOS 中也可以使用，但现阶段 Cocoa 绑定还不可以用在 iOS 中。

20.1.2 键值编码的基本处理

键值编码必需的方法在非正式协议 NSKeyValueCoding 中声明（头文件 Foundation/NSKeyValueCoding.h）。这些默认在 NSObject 中实现。

首先，我们来看看下面两个方法。

---

① 虽然感觉“键值”很难表达“key-value”的意图，但“键值”是苹果公司的日语文档中的用语。此外，也可以简称为 KVC。

② 关于“属性”指向的对象，请参考第 7 章的讨论。

- (id) **valueForKey:** (NSString *) key

返回表示属性的键字符串所对应的值。如果不能取得值，则将引起接收器调用方法valueForUndefinedKey:。

- (void) **setValue:** (id) value
forKey: (NSString *) key

将键字符串key所对应的属性的值设置为value。不能设定属性值时，将引起接收器调用方法setValue:ForUndefinedKey:。

不可以访问或设定值时将调用一些方法，这一点稍后再进行说明。这里先来看下实际的程序运行。

代码清单20-1中，Person类包含字符串类型name、email以及整数型age的实例变量，访问器只有setName:和emai。这里，我们来尝试使用setValue:forKey:和valueForKey:。

代码清单20-1 键值编码的简单示例（kvcsimple.m）

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

@interface Person : NSObject  // 使用ARC
{
    NSString *name;
    NSString *email;
    int       age;
}
- (void)setName:(NSString *)aName;
- (NSString *)email;
@end

@implementation Person

- (void)setName:(NSString *)aName
{
    NSLog(@"Access: setName:");
    name = aName;
}

- (NSString *)email
{
    NSLog(@"Access: email:");
    return email;
}

@end

int main(void)
{
    static NSString *keys[] = { @"name", @"email", @"age", nil };
    @autoreleasepool {
        id obj = [[Person alloc] init];
        [obj setValue:@"Taro" forKey:@"name"];
        [obj setValue:@"taro@ryugu-jo" forKey:@"email"];
        [obj setValue:[NSNumber numberWithInt:16] forKey:@"age"];
```

```
        for (int i = 0; keys[i]; i++)
            NSLog(@"%@: %@", keys[i], [obj valueForKey:keys[i]]);
    }
    return 0;
}
```

程序执行结果如下（本章的执行示例中，去掉了 NSLog() 的输出的开头部分）。可以看出，有访问器的属性会使用该访问器，没有访问器的属性也可以设定值和访问。而且，变量 obj 也可以为 id 类型，这点需要注意。以上就是键值编码的基本操作。

```
% ./kvcsimple
Access: setName:
name: Taro
Access: email:
email: taro@ryugu-jo
age: 16
```

20.2 访问属性

20.2.1 键值编码的方法的行为

首先，我们我们来看看`valueForKey:`的具体行为。下面的讲述将围绕着键字符串 name 进行。以下划线开始的名字不能用于方法名或实例变量名，关于这一点，请参考附录 C。此外，也不要使用以 get 开始的名字。

1. 接收器中如果有`name`访问器（或`getName`、`isName`、`_name`、`_getName`）则使用它。
2. 没有访问器时，使用接收器的类方法`accessInstanceVariablesDirectly`来查询。返回 YES 时，如果存在实例变量 name（或 _name、isName、_isName 等）则返回其值。
3. 既没有访问器也没有实例变量时，将引起接收器调用方法`setValue:forUndefinedKey:`。
4. 应该返回的值如果不是对象，则返回被适当的对象包装的值（参考下面）。

但是，如果接收器包含与带索引的访问器模式（见 20.3 节）一致的方法，则将返回有数组对象行为的代理（proxy）对象。

决定可否访问实例变量的类方法以及取值失败时调用的方法如下所示。

+ (BOOL) **accessInstanceVariablesDirectly**

通常定义为返回YES，可以在子类中改变。该类方法返回YES时，使用键值编码可以访问该类的实例变量。返回NO时不可以访问。只要该方法返回YES，实例变量的可视属性即使有@private修饰，如果也可以访问。

- (id) **valueForUndefinedKey:** (NSString *) key

不能取得键字符串对应的值时，从方法valueForKey:中调用该方法，默认情况下，该方法的执行会触发异常NSUndefinedKeyException。不过，通过在子类中修改定义，就可以返回其他对象。

下面介绍一下setValue:forKey:，依然使用键字符串 name。

1. 接收器中如果有setName:访问器（或_setName:）则使用它。set 后面的键的第一个字母必须大写。也就是说setname:是不可识别的。
2. 没有访问器时，使用接收器的类方法accessInstanceVariablesDirectly来询问。返回 YES 时，如果存在实例变量 name（或 _name、isName、_isName 等）则设定其值。使用引用计数管理方式时，实例变量如果为对象，则旧值会被自动释放，新值被保存并代入。
3. 既没有访问器也没有实例变量时，将引起接收器调用方法setValue:forUndefinedKey:。
4. 如果应该设定的值不是对象，则将变换到适合值（参考下面）。

如果设定值失败，则调用下面的方法。

- (void) **setValue:** (id)value
forUndefinedKey: (NSString *)key

不能设置键字符串key对应的属性值时，从方法setValue:forKey中调用该方法。默认情况下，该方法的执行会触发异常NSUndefinedKeyException。不过，通过在子类中修改定义，就可以返回其他对象。

关于属性和访问器方法名的对应在属性声明（第 7 章）中也有类似说明。但是请注意，键值编码比访问器名以及实例变量名更具灵活性。

由于键值编码所接收的对象都是 id 类型，因此，在该部分中，编译时不会进行仔细的类型检查。所以一定注意不要传入与属性不符的对象。

使用键值编码的程序如何执行，不是由静态解析源码语法的结果决定的，而是使用程序运行时包含的信息动态决定的。与实例变量的可视属性不同，有方法决定是否可以访问实例变量这一点，会使人感觉有损一致性。总之，键值编码这一强大功能就像是一把双刃剑，也伴随着危险，因此不可以滥用。

20.2.2 属性值的自动转换

在下面的讨论中，我们将属性中的单纯的数值，也就是整数或实数、布尔值这样的数据称为**标量**（scalar）值。将标量值、结构体、字符串或 NSNumber 等常数对象称为**属性**（attribute）。

能够使用键值编码操作的属性中，不仅有对象，也包括标量值、结构体等。上节中说明的方法valueForKey:在返回值为标量值或结构体时，会返回将其自动包装的对象。另一方面，为了给setValue:forKey:传入值，也需要使用适当的对象来包装。代码清单 20-1 所示的访问 int 类型的实例变量 age 就是这样一种情况。

传入的值如果为单纯的数值，也就是整、实数或布尔型，NSNumber 类就会被用于包装。例如，

BOOL 类型的情况下，会根据类方法numberWithBool:创建临时对象。为了从该对象中取得 BOOL 值，就要适用方法boolValue。关于其他单纯类型的数值，请参考 9.6 节。

如果是结构体，则采用 NSValue 类的实例来包装。在 Cocoa 中，可以使用 4 种标准的结构体 NSPoint、NSRange、NSSize、NSRect 来实现自动传入。关于获取生成方法和值的访问器，请参考表 9-5。而如果是其他的结构体或指针，传递时就需要自定义访问器方法。

属性值如果为对象，则可以将 nil 作为值传递。另一方面，使用setValue:forKey来设置标量型属性为 nil 时，setNilValueForKey: 方法将被发送给接收器。

- (void) **setNilValueForKey:** (NSString *) key

在键字符串key所对应的标量属性中设定nil时，会从方法setValue:forKey:中调用该方法。默认情况下，执行该方法将产生NSInvalidArgumentException异常，不过，通过在子类中修改定义，也可能产生其他行为。

20.2.3 字典对象和键值编码

字典类 NSDictionary 和 NSMutableDictionary 包含了协议 NSKeyValueCoding 的方法，使用它们可以进行键值编码。

NSDictionary 中定义了下面的方法。

- (id) **valueForKey:** (NSString *) key

键字符串开头不是“@”时，将调用方法objectForKey:。开头如果为“@”，则将去除开头字符后剩余的字符串作为键，调用超类的方法alueForKey:。

NSMutableDictionary 中定义了以下方法。

- (void) **setValue:** (id) value
  forKey: (NSString *) key

一般会调用方法setObject:forKey:，参数value为nil时，则调用方法removeObjectForKey:删除键对应的对象。

20.2.4 根据键路径进行访问

属性为对象时，该对象还可能持有属性。例如，假设存在类 WorkingGroup，它的实例变量 leader 与代码清单 20-1 中的 Person 具有相同的类。

```
@interface Person : NSObject
{
    NSString *name;
    NSString *email;
    int       age;
}
@end
```

```
@interface WorkingGroup : NSObject
{
    Person          *leader;    // 领导
    NSMutableArray  *members;   // 成员：可变的数组对象
}
@end
```

此时，调用方法alueForKey:从 WorkingGroup 的实例 aGroup 中取出领导名时，可以按照如下方式书写。

```
id aPerson = [aGroup valueForKey:@"leader"];
id name = [aPerson valueForKey:@"name"];
```

但是，这样的写法太繁琐。在键值编码中，使用某个键访问获得某个属性对象后，如果希望再用别的键来访问该对象，可采用如下方法。

```
id name = [aGroup valueForKeyPath:@"leader.name"];
```

像这样，用“.”号连接键表示的字符串称为**键路径**（key path）。只要能找到对象，点和键多长都没有关系。例如，假设我们想知道某台电脑的所有者所隶属的组的系统管理者的 email 地址，这时，就可以用 @"owner.group.admin.email" 这样的表达（如果对象和属性的关系是如此连接的）。

该书写方式和声明属性的点运算符的使用方法很类似。但是，声明属性的点是运算符，而这里的键路径则是一个字符串。

使用键路径访问属性的方法如下所示。

- (id) **valueForKeyPath:** (NSString *) keyPath

  以点切分键路径，并使用第一个键向接收器发送valueForKey:方法。然后，再使用键路径的下一个键，向得到的对象发送ValueForKey:方法，如此反复操作，返回最后获得的对象。

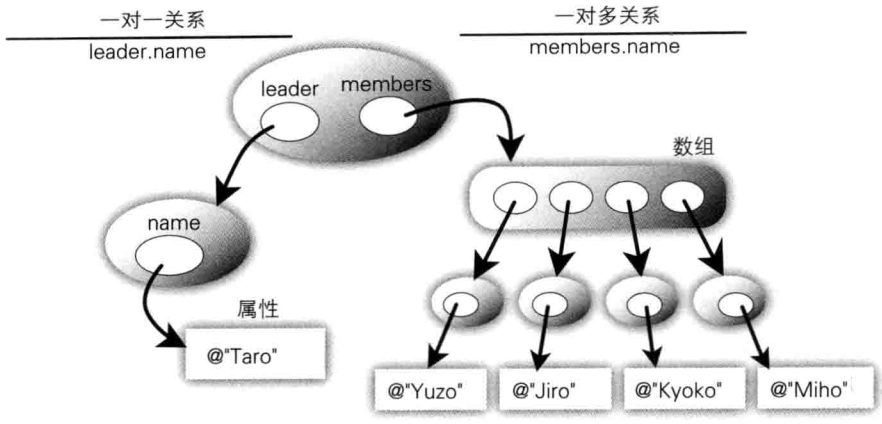
- (void) **setValue:** (id) value
  forKeyPath: (NSString *) keyPath

  与valueForKeyPath:方法同样取出对象，这里只对路径中的最后一个键调用setValue:forKey:方法，并设定属性值为value。

20.2.5 一对一关系和一对多关系

用键字符串或键路径指定属性时存在两种情况，一种是取得的值仅限一个，一种是可以获得多个值的集合。例如，在前例中，类 WorkingGroup 的属性 leader 是 Person 类型的对象，如果组确定，那么就确定只有一个领导。另一方面，members 是数组，包含多个对象。@"members.name" 键路径中对应着与人数相同个数的人名（图 20-1）。

▶ 图 20-1 一对一关系和一对多关系



像这样，使用键（或键路径）访问时，我们将对象确定为一个的属性称为指定**一对一关系**（to-one relationship）的属性，将属性值为数组或集合的属性称为指定**一对多关系**（to-many relationship）的属性。

然而，字典对象虽是集合的一种，但由于以字符串作为键的入口和属性拥有同等功能，因此，如果键对应的值是一个对象，那么也是一对一关系。

关于一对多关系属性的访问、更改，需要留意以下几点。

1. 使用集合元素对象持有的键访问一对多关系属性时，键对应的属性被作为数组或集合返回。
2. 使用集合元素对象持有的键设定一对多关系属性时，各元素对象键对应的属性全都被更改。

下面以上述的 Person 和 WorkingGroup 为例进行说明。代码清单 20-2 仅展示了 main 函数部分。类 Person 中有指定名字和年龄的初始化器，WorkingGroup 中有指定领导的初始化器。假设要使用键路径更改领导的名字以及成员的年龄，该如何操作呢？

▶ 代码清单 20-2 使用键路径访问的测试（relation.m 的 main 函数）

```
int main(int argc, char **argv)     // 使用 ARC
{
    @autoreleasepool {
        WorkingGroup *group;
        id chief = [[Person alloc] initWithName:@"Taro" age:30];
        id staff1 = [[Person alloc] initWithName:@"Yuzo" age:26];
        id staff2 = [[Person alloc] initWithName:@"Miho" age:24];
        group = [[WorkingGroup alloc] initWithLeader:chief];
        [group addMember: staff1];
        [group addMember: staff2];

        NSLog(@"1: %@", [group valueForKeyPath:@"leader.name"]);
        NSLog(@"2: %@", [group valueForKeyPath:@"members.name"]);
        NSLog(@"3: %@", [group valueForKeyPath:@"members.age"]);
        [group setValue:@"Jiro" forKeyPath:@"leader.name"];
        [group setValue:[NSNumber numberWithInt:10]
             forKeyPath:@"members.age"];
```

```
        NSLog(@"4: %@", [group valueForKeyPath:@"leader.name"]);
        NSLog(@"5: %@", [group valueForKeyPath:@"members.age"]);
    }
    return 0;
}
```

执行结果如下所示。为方便查看，这里省略了 NSLog 的开头，并将多行输出汇总在了一行。字符串或数字用（）括起来的部分是数组对象的输出（ASCII 型属性列表）。

可以看出，属性表示单一对象时，访问和变更的只能是该对象。另一方面，表示键路径的属性为数组时，访问得到的结果仍然是数组。而且，改变值时，所有对应的值都会被改变。

```
% ./relation
1: Taro
2: ( Yuzo, Miho )
3: ( 26, 24 )
4: Jiro
5: ( 10, 10 )
```

数组元素又是别的数组的属性的情况下，使用键路径时得到的对象有可能是嵌套数组（成为子数组）。

20.2.6 数组对象和键值编码

数组类 NSArray 和 NSMutableArray 以及集合类 NSSet 和 NSMutableSet 都包含协议 NSKeyValueCoding 的方法，也都有键值编码。之前的程序示例中也有集合方法的执行。

- (id) **valueForKey:** (NSString *) key

以key为参数，对集合的各元素调用方法valueForKey:后返回数组(NSSet时返回集合)。对各成员适用方法valueForKey:，返回nil时，则包含NSNull实例。

- (void) **setValue:** (id) value
  forKey: (NSString *) key

对集合各元素调用方法setValue:forKey:。需要注意的是，即使集合对象自身不可以改变，也能调用该方法。

20.3 一对多关系的访问

20.3.1 带索引的访问器模式

包含一对多关系的属性也可以像其他属性一样访问，但访问用户自定义类中包含的各元素对象的方法却有些不同。从键值编码方面来说，如果该属性也能像数组对象那样处理，使用起来就简单多了。

即使是非数组对象，如果有某个模式的访问器，也可以进行像数组一样的键值编码操作。该访问器模式称为**带索引的访问器模式**（indexed accessor pattern）。

这里具体讲一下下面两个方法的实现。下划线部分中会输入键字符串。

```
- (NSUInterger)countOf _ _ _ ;
- (id)objectIn _ _ _ AtIndex:(NSUInterger)index;
```

请大家回想一下类簇。数组类为类簇，原始方法是count和objectAtIndex。只要定义了这两个方法就可以定义数组行为的对象。上述两个方法也是同样，用键值编码访问时也可定义数组行为。

下面来看下实际的程序运行。代码清单 20-3 中有类 Team。C 的数组 members[] 的开头存入了若干个对象，变量 count 记录着对象个数。根据上述带索引的访问器模式，我们定义了两个方法countOfFellows和objectInFellowsAtIndex:。

代码清单 20-3　带索引的访问器模式的使用示例（kvcindex.m）

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

#define MAXMEMBER  8

@interface Team : NSObject     // 使用 ARC
{
    id   members[MAXMEMBER];
    int  count;                // 初始值为 0
}
- (NSUInteger)countOfFellows;
- (id)objectInFellowsAtIndex:(NSUInteger)index;
@end

@implementation Team
- (void)addMember:(id)someone {
    if (count < MAXMEMBER)
        members[count++] = someone;
}

- (NSUInteger)countOfFellows {
    return count;
}
```

```
- (id)objectInFellowsAtIndex:(NSUInteger)index {
    return (index < count) ? members[index] : nil;
}
@end

int main(void)
{
    @autoreleasepool {
        id obj;
        id aTeam = [[Team alloc] init];
        [aTeam addMember:@"Hiroshi"];
        [aTeam addMember:@"Mika"];
        obj = [aTeam valueForKey:@"fellows"];
        NSLog(@"obj=%@", NSStringFromClass([obj class])); // 类名
        NSLog(@"Fellows: %@", obj);
    }
    return 0;
}
```

请注意下 main 函数内的注释部分。这里用 @"fellows" 键访问 Team 类实例，但该类中既没有 fellows 实例变量也没有访问器。执行结果如下所示。

```
% ./kvcindex
obj=NSKeyValueArray
Fellows: (Hiroshi, Mika)
```

可以看出，用 @"fellows" 键访问后，结果返回了数组对象。人名用 ( ) 括起来是数组对象的属性列表的书写方法。NSKeyValueArray 是系统内部类，提供了数组的接口这样一种代理。

为了提高运行效率，除上述两个方法之外，还可以实现下面的方法。这也是和数组类簇中的`getObjects:range:`相同的方法。

```
- (void)get _ _ _ :(id _ _ unsafe _ unretained [])aBuffer
        range:(NSRange)aRange;
```

20.3.2 一对多关系的可变访问

实现之前说明的带索引的访问器模式后，使接收器对象内看起来包含了常数数组，而除此之外，也可以包含可变数组。该可变数组不仅包含属性的元素对象，添加或删除数组元素后，也会联动追加或删除属性元素。

首先来看一下获得像这样的可变数组对象的方法。

- (NSMutableArray *) **mutableArrayValueForKey:** (NSString *) key

返回相当于用键字符串指定的一对多关系的属性的可变数组。操作被返回的数组与操作属性同时进行。

```
- (NSMutableArray *) mutableArrayValueForKeyPath:
    (NSString *) keyPath
```

接收器属性在键路径中指定，其他与上述方法相同。

用该方法操作属性时，除了之前提到的访问常数数组需要添加两个方法之外，还需要实现用来插入和删除的方法。下划线的部分加入了键字符串（通常为复数形式）。使用这些方法并通过键值编码访问时，内部的代理就会作为数组进行操作了。

```
- (void)insertObject:(id)obj in _ _ _ AtIndex:(NSUInteger)index;
- (void)removeObjectFrom _ _ _ AtIndex:(NSUInterger)index;
```

不仅是上述两个方法，通过实现下面的方法也可以实现属性的可变访问。

```
- (void)set _ _ _ :(id)anArray;
```

在实现该方法时，通过使用被作为参数传入的数组元素对象，就可以置换一对多关系的全部属性内容。那么怎么实现可变数组操作呢？从接收器中取出常数数组对象，对其进行元素更新操作后，通过该方法就可以完全置换属性。但是，和实现了插入和删除的两个方法时相比，效率要低一些。

在添加了插入和删除的方法的基础上，如果能实现下面的方法，则将能显著改善运行效果。

```
- (void)replaceObjectIn _ _ _ AtIndex:(NSUInteger)index
                                withObject:(id)obj;
```

如果上述方法都没有实现，那么当存在与键字符串同名的实例变量且该变量又是可变数组对象时，方法mutableArrayValueForKey:将直接返回该值。

上面介绍了将一对多关系的属性作为可变数组进行操作的方法。同样，也有将其作为可变集合的操作方法，而关于这一点，本书不做讨论。

20.4 KVC 标准

20.4.1 验证属性值

通过使用键值编码，可以在属性名中自动选择访问器或者直接修改实例变量的值。然而在某些情况下，如果预期之外的对象被设定为了的属性值，那么就可能出现问题。

因此，在为某属性代入对象前，为了验证该对象是否有误，可以使用相应的方法来验证。但是该验证方法不能自动调用[①]，因此，在访问属性前，必须自行调用该方法。

① 使用 Cocoa 绑定时，可以设定自动验证。

验证某键字符串的属性值的方法可按如下形式定义。下划线中写入键字符串。参数 ioValue 为需要验证的对象指针。参数 outError 被用来当验证结果中存在问题时返回出错信息。

```
- (BOOL)validate___:(inout id *)ioValue error:(out NSError **)outError;
```

要验证的对象没有问题时，方法返回 YES，两个参数值不变。

对象有问题，但是能将对象值修正为有效值时，方法会创建新的对象，并取代原对象将新对象代入 ioValue。参数 outError 不变，返回值为 YES。

对象有问题且不能修正时，则创建错误对象并将其代入参数 outError。方法返回 NO。

设定属性值的访问器方法（`set___:`）不能调用验证方法。

例如，大家不妨考虑一下邮政号码字符串为属性时的情况。如果属性名为 postalCode，则验证方法为`validatePostalCode:error:`。赋值“数字 3 位 - 数字 4 位”这种形式的字符串时返回 YES，否则返回 NO。数字如果是全角，则将其改成半角后返回。

运行时键会被动态地赋值给对象的情况下，不能在代码中使用上述方法名。此时，可以使用下面的两个方法。

```
- (BOOL) validateValue: (inout id *) ioValue
                forKey: (NSString *) key
                 error: (out NSError **) outError
```

使用指定键寻找 validate___:error: 的验证方法并调用，如果不存在这样的验证方法则返回 YES。

```
- (BOOL) validateValue: (inout id *) ioValue
            forKeyPath: (NSString *) keyPath
                 error: (out NSError **) outError
```

与上述方法的不同之处在于键路径的指定。需要注意的是，实际被调用的验证方法并不是该方法的接收器，而是与最后的键元素相对应的属性的验证方法。

使用引用计数方式管理内存的情况下，需要格外注意验证方法的调用。验证的对象、错误对象都由指针来访问时，它们的值可能会被修改。此时，即使原对象没有被显式地释放，也不能再进行访问。此外，传递那些拥有所有权的对象时也需要小心。

20.4.2 键值编码的准则

如果可以使用键值编码来访问某个属性，则称该属性是键值编码的准则，或称为 **KVC 准则**（compliance）[①]。反之，如果知道某属性为 KVC 准则，那么就可以编写使用键值编码的程序。KVC 准则和协议适用的概念不同，它不是以类为单位，而是讨论以各个属性为单位是不是准则的问题。

要使某属性为 KVC 准则，就必须实现能使用`valueForKey:`方法的访问器。当属性可变时，

① 该 compliances 与企业发生丑闻时经常使用的“遵守法律”一词同义。正确遵守药物的使用量及用法也是 compliance。总之就是严格遵守规则的意思。

还需要与方法setValue:forKey:相应的访问器。下面列举一些具体的条件。

Property 为属性（标量值或单纯型的对象）或一对一关系时，要想成为 KVC 准则，就需要满足如下条件。属性名为“name”。

1. (a) 实现了name或isName访问器方法。或者
   (b) 包含 name（或 _name）实例变量。
2. 可变属性时，还需要实现setName:方法。需要执行键值验证时，要实现验证方法（validateName:error:）。但是，setName:方法中不能调用验证方法。

属性为一对多关系时，要想成为 KVC 准则，需满足如下条件。属性名为“names”。

1. (a) 实现了返回数组的names方法。或者
   (b) 持有包含 names（或 names）数组对象的实例变量。或者
   (c) 实现了带索引的访问器模式的方法countOfNames以及objectInNamesAtIndex:。
2. 当一对多关系的属性可变时
   (a) 持有返回可变数组对象的names方法。或者
   (b) 实现了带索引的访问器模式的方法
   insertObject:inNamesAtIndex:以及removeObjectFromNamesAtIndex:。

当然，在实现带索引的访问器模式的方法时，为改善执行效率，也可以添加其他方法来实现。

20.5 键值观察

20.5.1 键值观察的基础

键值观察（key-value observing），即某个对象的属性改变时通知其他对象的机制。有时也记作 KVO。为了在 GUI 组件和程序之间调整值，Coacoa 绑定会将键值编码和键值观察组合起来使用。因此，要想理解 Cocoa 绑定，首先就必须理解键值观察的概念。

对被观察对象来说，键值观察就是注册想要监视的属性的键路径和观察者。当属性改变时，观察者会接收到消息。虽然和 NSNotification 通知相类似，但键值观察中不存在相当于通知中心的对象，所以不可以指定消息的选择器。

键值观察功能可用 NSObject 来实现，可以监视属性、一对一关系、一对多关系等各种形式的属性。

仅仅在使用键值编码准来则访问访问器或实例变量的情况下，才可以监视属性的变化。在方法内直接改变实例变量值时，就不能监视了。具体的 KVC 准则有以下三点。

1. 随访问器方法而改变。

2. 使用setValue:forKey:和键进行改变。此时也可能不经由访问器。
3. 使用setValue:forKeyPath:和键路径进行改变。此时也可能不经由访问器。不仅仅是最终的监视对象的属性，当路径中的属性发生变化时，也会被通知（参考下面的例子）。

NSObject 中提供了键值观察所必需的方法，头文件 Foundation/NSKeyValueObserving.h 中将其定义为了非正式协议的形式。

首先，需要使用下面的方法注册键值观察。

```
- (void) addObserver: (NSObject *) anObserver
          forKeyPath: (NSString *) keyPath
             options: (NSKeyValueObservingOptions) options
             context: (void *) context
```

从接收器的角度来看，监视键路径keyPath中的某个属性，要在接收器中注册。观察者为对象anObserver。属性变化时发送的通知消息中，包含着显示变化内容的字典数据、参数context中指定的任意指针（或对象）。options中指定字典数据中包含什么样的值。值可取下面的常数或它们的位或运算。

NSKeyValueObservingOptionNew … 提供属性改变后的值。
NSKeyValueObservingOptionOld … 提供属性改变前的值。

使用该方法后，当指定监视的属性被改变时，下面的消息将被发送给观察者。所有的观察者都要实现如下方法。

```
- (void) observeValueForKeyPath: (NSString *) keyPath
                       ofObject: (id) object
                         change: (NSDictionary *) change
                        context: (void *) context
```

从参数object的角度来看，当键路径keyPath的属性发生变化时会发送通知。字典change中保存着改变的相关信息。参数context中返回注册观察者时指定的值。

被作为通知消息的参数传入的字典 change 中保存着入口的键字符串及其内容，如下所示。

表 20-1 字典中保存的入口

键字符串	内容
NSKeyValueChangeKindKey	表示属性改变的种类。该入口值为包含整数的NSNumber实例，该整数为NSKeyValueChangeSetting时，表示保存着和属性不同的值。其他值为下面三个，使用一对多关系的可变数组来表示元素对象的改变 NSKeyValueChangeInsertion 插入 NSKeyValueChangeRemoval 删除 NSKeyValueChangeReplacement 置换
NSKeyValueChangeNewKey	观察者被注册时，如果用选项指定了NSKeyValueObservingOptionNew，则属性更新后的信息会被保存。所保存的对象因改变的类型（上述NSKeyValueChangeKindKey）而异。属性中保存有新对象时，该新对象为入口值。对可变数组进行了插入、置换时，包含新元素的数组（NSArray）为入口值

（续）

键字符串	内容
NSKeyValueChangeOldKey	观察者被注册时，如果用选项指定了 NSKeyValueObservingOptionOld，则属性更新前的信息会被保存。所保存的对象因改变类型（上述 NSKeyValueChangeKindKey）而异。属性中保存有新对象时，改变前的旧对象为该入口值。对可变数组进行了删除、置换时，包含空元素的数组（NSArray）为入口值
NSKeyValueChangeIndexesKey	动态数组中进行了插入、删除、置换时，表示执行了这些操作的索引集合——NSIndexSet 类的实例为入口值

如果要停止已经注册的监视，可使用下面的方法。

```
- (void) removeObserver: (NSObject *) anObserver
             forKeyPath: (NSString *) keyPath
```

观察者为anObserver，键路径将删除keyPath内容注册的监视这一消息传达给接收器。

使用引用计数管理方式时需要注意一些事项。注册属性监视时，不需要持有观察者及监视对象的属性。还有，如果在不删除注册信息的情况下将关联对象释放，那么随着属性的变更，就可能会发生访问已释放了的对象的危险。

20.5.2 示例程序

下面让我们通过一个简单的程序来看看键值观察到底是怎么回事。这仅仅是一个建模游戏登场人物关系的程序，没有其他意义。

类 Creature 中包含属性字符串 name 和整数 hitPoint。hitPoint 为声明属性，还有一个与其无关的`sufferDamage:`方法。为变成键值观察的观察者，这里还定义了接收消息的方法。方法`description`用来简单地表示对象值。继承该 Creature 的类是 Person 和 Dragon，Dragon 中只添加了属性 master（饲养者），其他与 Creature 相同。

创建两个 Person 实例及一个 Dragon 实例。（A）位置处的设定是，当 dra 的 master 的 hitPoint 被改变时可以通知给 dra 自身。（B）的设定是，当 p1 的 hitPoint 被改变时可以通知 p2。

代码清单 20-4 键值观察的例子（kvosample.m）

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

@interface Creature : NSObject  // 使用 ARC
{
    NSString *name;
    int      hitPoint;  // 命中点
}
- (id)initWithName:(NSString *)str hitPoint:(int)n;
@property int hitPoint;
- (void)sufferDamage:(int)val;
- (NSString *)description;
- (void)observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object change:(NSDictionary *)change
```

```
    context:(void *)context; // 接收通知的方法
@end

@implementation Creature
- (id)initWithName:(NSString *)str hitPoint:(int)n {
    if ((self = [super init]) != nil)
        name = str, hitPoint = n;
    return self;
}

@synthesize hitPoint;

- (void)sufferDamage:(int)val {  // 受伤后命中点降低
    hitPoint = (hitPoint > val) ? (hitPoint - val) : 0;
}

- (NSString *)description {
    return [NSString stringWithFormat:@"<<%@, %@, HP=%d>>",
                NSStringFromClass([self class]), name, hitPoint];
}

- (void)observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object change:(NSDictionary *)change
    context:(void *)context
{   // 接收键值观察的通知
    printf("%s", [[NSString stringWithFormat:
            @"--- Received by %@ ---\n"          // 使用字符串常量连接
            @"Object=%@, Path=%@\n"
            @"Change=%@\n", self, object, keyPath, change]
            UTF8String]);
}
@end

@interface Person : Creature
@end

@implementation Person
@end

@interface Dragon : Creature
@property(weak) Person *master;  // 饲养人
@end

@implementation Dragon
@synthesize master;
@end

int main(void)
{
    int opt = NSKeyValueObservingOptionOld
            | NSKeyValueObservingOptionNew; // 接收改变前后的值
    @autoreleasepool {
        Person *p1, *p2;
        Dragon *dra;
        p1 = [[Person alloc] initWithName:@"Taro" hitPoint:500];
```

```
        p2 = [[Person alloc] initWithName:@"Jiro" hitPoint:140];
        dra = [[Dragon alloc] initWithName:@"Choco" hitPoint:400];
        dra.master = p1;
        [dra addObserver:dra forKeyPath:@"master.hitPoint"
                options:opt context:NULL];  ------------------------------ (A)
        [p1 addObserver:p2 forKeyPath:@"hitPoint"
                options:opt context:NULL];  ------------------------------ (B)

        [p1 setHitPoint: 800];  ---------------------------------------- ❶
        [p1 sufferDamage: 200];  --------------------------------------- ❷
        dra.master = p2;  ---------------------------------------------- ❸
        p1.hitPoint -= 100;  ------------------------------------------- ❹
        p2.hitPoint += 200;  ------------------------------------------- ❺
        [dra removeObserver:dra forKeyPath:@"master.hitPoint"];  ------- ❻
        p2.hitPoint -= 300;  ------------------------------------------- ❼
        [p1 removeObserver:p2 forKeyPath:@"hitPoint"];
    }
    return 0;
}
```

执行结果如下所示。首先，❶ 处改变了 p1 的 hitPoint，并确实向 p2 和 dra 发送了通知消息。

另一方面，❷ 中虽然调用了更改 hitPoint 的消息`sufferDamage:`，但却没有收到相应的通知。像这样，使用非 KVC 准则的消息改变属性后，就成为监视外的对象了。

❸ 中改变了 dra 的 master。这时将接收到 master.hitPoint 被改变的消息。所以，即使不是指定的键路径的属性，在产生改变时也会被通知。❹、❺ 中使用声明属性改变了 p1、p2 的 hitPoint，这也被监视到了。可见，虽然（A）的设定中使用了 master.name 键路径，但即便不使用相同的键路径，也能进行监视。

在 ❻ 中删除以 dra 为观察者注册的监视后，❼ 中 p2 的 hitPoint 即使被改变，也无法接收到通知消息。

```
--- Received by <<Person, Jiro, HP=140>> ---                    【对应❶】
Object=<<Person, Taro, HP=800>>, Path=hitPoint
Change={ kind = 1; new = 800; old = 500; }
--- Received by <<Dragon, Choco, HP=400>> ---                    【对应❶】
Object=<<Dragon, Choco, HP=400>>, Path=master.hitPoint
Change={ kind = 1; new = 800; old = 500; }
--- Received by <<Dragon, Choco, HP=400>> ---                    【对应❸】
Object=<<Dragon, Choco, HP=400>>, Path=master.hitPoint
Change={ kind = 1; new = 140; old = 600; }
--- Received by <<Person, Jiro, HP=140>> ---                     【对应❹】
Object=<<Person, Taro, HP=500>>, Path=hitPoint
Change={ kind = 1; new = 500; old = 600; }
--- Received by <<Dragon, Choco, HP=400>> ---                    【对应❺】
Object=<<Dragon, Choco, HP=400>>, Path=master.hitPoint
Change={ kind = 1; new = 340; old = 140; }
```

20.5.3 一对多关系的属性监视

属性为一对多关系时，监视方法与前述方法相同。但是，有一点必须要注意。那就是，针对一对多关系为数组类型时使用方法mutableArrayValueForKey:获得的对象，以及一对多关系为集合类型时使用方法mutableSetValueForKey:获得的对象，如果不修改值是不能进行监视的（方法分别为~ForKeyPath:时也一样）。

例如，假设某对象aParty的实例变量members包含NSMutableSet实例，使用同名访问器members传入该对象。此时执行下面两行代码。

```
id a = [aParty members];
id b = [aParty mutableSetValueForKey:@"members"];
```

a和b这两个对象是不同的。实际上，执行这个程序后就会发现，a和b分别是__NSCFSet类和NSKeyValueSlowMutableSet类的实例（在Mac OS X 10.7.1中执行）。因为a是集合，所以即使有元素操作也不会收到变更通知。而b中的元素改变则是被监视的。

也可以指定数组内的特定元素进行监视，本书中对此不做讨论。

20.5.4 依赖键的登记

某属性值随着同一对象的其他属性的改变而改变是常有的事情。通过事先将这样的依赖关系在类中注册，那么即便属性值间接地发生了改变，也会发送通知消息。为此需使用下面的类方法。

```
+ (void) setKeys: (NSArray *) keys
    triggerChangeNotificationsForDependentKey:
                (NSString *) dependentKey
```

数组keys中可以保存多个键。注册依赖关系，使当这些键中任意一个键对应的属性发生改变时，都会自动引起与键dependentKey的属性变化时一样的行为（被监视时发送通知）。

例如，Person类中有image属性，该属性保存有weapon（武器）、armor（铠甲）、helmet（头盔）以及用来表示人物的图片。假设根据所持有的武器、铠甲、头盔的不同，图片也不一样。那么，当改变持有的物品时，就需要重绘图片。因此，与其分别监视武器、铠甲、头盔，不如设定他们之间的依赖关系，这样只监视图片就可以了，程序也会变简单。

20.6 Cocoa 绑定概述

20.6.1 目标 - 行为 - 模式的弱点

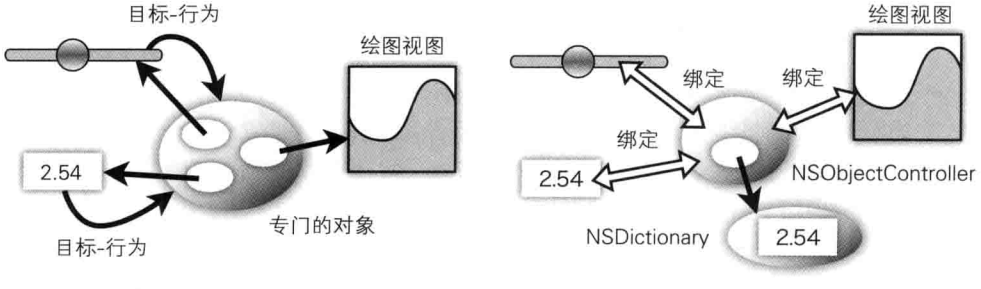
在面向对象程序设计中，应尽可能地去除特定的类与类之间的关系，定义低耦合的类。也就是说，某个类改变时，最好不会影响其他的类，否则就不是我们期望的编程。但是，就算完美地定义了每个类，它们间如果不能联动也就不能实现功能。一个典型的例子就是，窗体及窗体上面的按钮、菜单等 GUI 组件也是对象，它们间如果不连接，程序就不能运行。

这样看来，除了各个类或 GUI 组件原本就应该有的功能之外，还需要为它们之间的联动补充必要的代码。而这样的代码就像是把元素粘在一起的胶水，因此称为**胶水代码**（glue code）。胶水代码既可以被写成专门的类，也可以渗透在各个关联的类中。

Mac OS X 从第一个版本 NeXTstep 开始，就有了将 GUI 组件与使用组件的对象结合在一起的开发工具 Interface Builder。“胶”的部分不需要特意编写代码，使用 Interface Builder 的 GUI 环境就可以简单地实现，因此能大幅提高编程效率。Interface Builder 中 GUI 组件被操作时，会预先指定向哪个对象发送什么消息，这里，Objective-C 灵活的消息发送机制发挥着非常大的作用。以上就是我们说明过的目标 – 行为模式。

但是，“从操作组件向目标发送消息”这样的方式在很多情况下都是无能为力的。特别是当值改变时，为了使多个对象联动，必须书写专门的代码。图 20-2（a）希望实现的是，绘图用的参数可从滑块或文本域输入中获得，并根据值的变化来改变各种显示。这样的情况很常见，但只用 Interface Builder 连接解决不了这样的问题，还有必要使用专门的对象。程序自身虽然简单，但这样的组合多了的话，就要写大量相似的代码。

图 20-2 组件连接方法的异同



（a）目标-行为　　（b）Cocoa绑定

20.6.2 什么是 Cocoa 绑定

从 Mac OS X 10.3 起开始引入的 **Cocoa 绑定**（Cocoa binding）是指，使用键值编码和键值观察的

组合，**在多个对象间共享属性值的变化的机制**（在 iOS 中不可以使用）。

下面以图 20-2（b）为例来说明了它的概要。

再该例中，滑块和文本域、及绘图视图共享某个属性。滑块或文本域不仅可以改变属性值，还可以将属性值的改变通知给各个对象。这样的属性（的集合）称为**模型**，管理模型的对象称为**控制器**。根据用途的不同，可以使用字典对象或数组对象作为模型。假设这里以字典对象为模型，并共享它的一个条目。以字典为模型时，可以使用已有的 NSObject Controller 类实例作为控制器。

对控制器指定模型属性为键路径，将对象设定为该属性改变时可以获得相应的通知消息。同时，该对象必须定义方法来接收通知消息。使用这个机制将对象和控制器关联起来，就称为**绑定**。

Interface Builder 除了能根据目标 - 行为机制进行组件间的连接外，还可以连接为 Cocoa 绑定而设定的 GUI 组件。滑块或文本域等大家所熟知的 GUI 组件中也已经实现了 Cocoa 绑定的方法。NSObjectController 等控制器实例在 Interface Builder 中也可作为组件来使用。因此，图 20-2（b）中只需要考虑绘图视图自身的制作方法、及视图与控制器绑定的方法即可。

除了之前提的 NSObjectController 这样的控制器外，其他还有很多控制器类。虽然也可以自己制作需要的控制器，但一般情况下，根据希望通过绑定来共享的属性的性质或目的，从预先准备好的类中选择就可以了。例如，操作数组或集合时选择 NSArrayController 类，连接用户默认（见 16.4 节）与 GUI 组件时选择 NSUserDefaultsController 类等等。

另外，将属性值传给绑定目标时，还可以变换值的单位或者反转真假值。详情请参考类 NSValueTransFormer 的参考文档等。

20.6.3 Cocoa 绑定所需的方法

使用 Cocoa 绑定，将某对象绑定到控制器属性时，该对象必须实现下面的方法。该方法用头文件 AppKit/NSKeyValueBinding.h 中的非正式协议 NSKeyValueBindingCreation 来声明。

```
- (void) bind: (NSString *) binding
            toObject: (id) observable
        withKeyPath: (NSString *) keyPath
             options: (NSDictionary *) options
```

因为参数 binding 为表示和其他实例属性绑定的属性的字符串，所以称为**绑定名**。参数 observable 是控制器，由参数 keyPath 中控制器可见的键路径来指定控制器属性。options 中指定记录选项条目的字典对象，不做特别指定时传入 nil。

在对象侧，至少还需要定义一个属性改变时接收通知消息的方法。

```
- (void) observeValueForKeyPath: (NSString *) keyPath
                        ofObject: (id) object
                          change: (NSDictionary *) change
                         context: (void *) context
```

这也是讲解键值观察时说明的方法（非 NSKeyValueBindingCreation 协议方法）。也就是说，绑定时必须要设定键值观察。参数 keyPath 中会被传入上述指定的键路径，参数 object 中则会被传入控制器。通过上述两个方法的组合，Cocoa 绑定即可发挥功效。

下面详细了解一下绑定名。根据对象的不同，有的可能会与多个控制器绑定，有的则是在同一个控制器间持有多个不同目标的绑定。此时，该字符串的作用就是向接收器传递哪个属性为绑定对象。所以，该字符串不一定要与属性名或键路径一致。此外，滑块等 Interface Builder 的 GUI 组件使用了“value”这一绑定名来表示设定值。而最大・最小值、可否利用、字体等则使用共享绑定名，在 Interface Builder 中可以容易地确认。

也可以定义非正式协议 NSKeyValueBindingCreation 的其他方法。例如，解除设定过的绑定时，通过定义下面的方法就可以解除键值观察。

```
- (void) unbind: (NSString *) binding
```

将类作为组件在 Interface Builder 中注册时，需要定义其他方法。关于这一点，本书中不做说明。

20.6.4 例题：绘制二次函数图的软件

鉴于篇幅问题，本书中没有对 Cocoa 绑定进行详细说明，因此，这里我们选取一个简单的例子，通过展示之前介绍的方法定义，来从整体上把握 Cocoa 绑定。

例子是一个绘制二次函数图的软件，也就是在 xy 平面上绘制抛物线。通过滑块改变下面式子中的系数 a、b、c 的值时，该软件可以自动修改绘图。系数值还可以在文本域中设定（图 20-3）

$$y = ax^2 + bx + c$$

而且，在该软件中，拖动图框时系数也会随之变化。上下移动图时系数 c 的值会跟着变化，而左右拖动时会怎么变化呢？

图 20-4 显示了接口的概况。有 3 组滑块和文本域，对应着 3 个参数。创建一个 NSObjectController 实例，以可变字典对象为模型。字典对象可以通过“content”键来引用。字典对象的 3 个系数值分别用“a”、“b”、“c”这 3 个键来保存。这样一来，参数 a 的值就可以通过“content.a”键路径从控制器中引用。然后将它和滑块、文本域绑定。这些组件与设定值对应的绑定名为“value”。图中只描述了系数 a，其他两个系数也是同样的。

图 20-3 例子执行时的画面

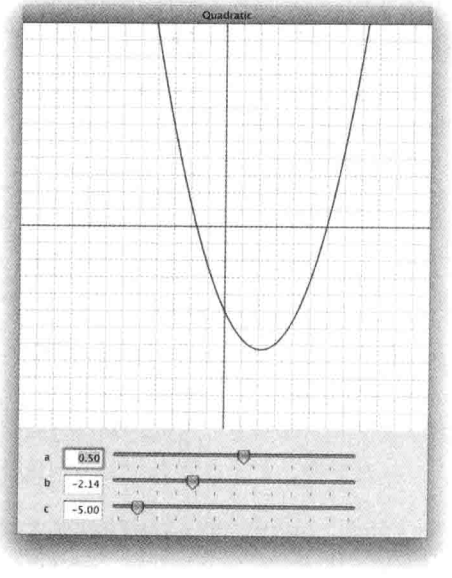
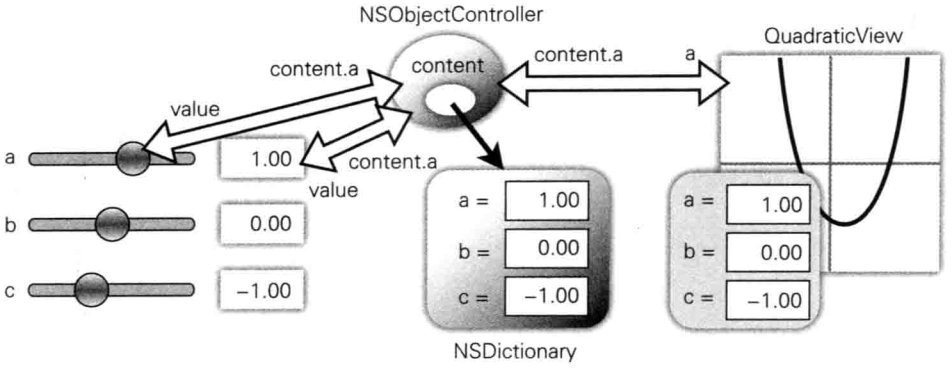


图 20-4 二次函数绘图软件的绑定（仅系数 a）



绘图通过被作为 Application 框架的 NSView 子类（自定义视图）定义的 QuadraticView 类来实现（Quadratic Function，二次函数）。QuadraticView 中可以使用绑定名“a”、“b”、“c”。而且还有和窗体中的组件层次进行绑定的 FunctionCtrl 类。需要自定义的类仅此两个。

关于窗体组件的配置和应用簇的创建，我们在第 17 章的例子中已经做过介绍，因此不再赘述。这里只展示与 Cocoa 绑定相关的代码。

首先介绍 FunctionCtrl，它是 NSApplication 实例的委托，应用启动后只进行菜单和窗体的设定。这些设定大部分都可通过 Interface Builder 完成。也可以使用在窗体中配置组件的方法来绑定组件。代码清单 20-5 展示了其中的一部分，只是重复调用绑定方法而已。而且，为了将创建的自定义视图和模型在 Interface Builder 中绑定起来，自定义视图作为 Interface Builder 的托盘注册，稍有些麻烦。

代码清单 20-5 组件间绑定（FunctionCtrl.m 的一部分）

```
- (void)windowSetUp {            // 使用 ARC
    ...
    static NSString *title[] = { @"a", @"b", @"c" }; /* 系数的名称 */
    static double initVal[] = { 1.0, 0.0, -1.0 };    /* 系数的初始值 */
    NSTextField *tx[3];
    NSSlider *sl[3];
    QuadraticView *qv;
    NSString *str;
    NSMutableDictionary *dic;
    dic = [[NSMutableDictionary alloc] initWithObjectsAndKeys:
        [NSNumber numberWithDouble:initVal[0]], @"a",
        [NSNumber numberWithDouble:initVal[1]], @"b",
        [NSNumber numberWithDouble:initVal[2]], @"c", nil];
    objCtrl = [[NSObjectController alloc] initWithContent:dic];
    /* 设定控制器 */

    for (i = 0; i < 3; i++) {
        ...
        tx[i] = [[NSTextField alloc] initWithFrame:r];
        ... /* 设定文本域 */
        sl[i] = [[NSSlider alloc] initWithFrame:r];
        ... /* 设定滑块 */
        str = [NSString stringWithFormat:@"content.%@", title[i]];
```

```
        /* 为每个系数创建 "content.a" 这样的键路径 */
        [tx[i] bind:@"value" toObject:objCtrl /* 第 i 个文本域 */
            withKeyPath:str options:nil];     /* 绑定到控制器 */
        [sl[i] bind:@"value" toObject:objCtrl /* 将第 i 个滑块 */
            withKeyPath:str options:nil];     /* 绑定到控制器 */
        [qv bind:title[i] toObject:objCtrl    /* 将视图的第 i 个系数 */
            withKeyPath:str options:nil];     /* 绑定到控制器 */
    }
    ...
}
```

20.6.5 自定义视图的方法定义

我们来看下 QuadraticView 是怎么定义的。接口如代码清单 20-6 所示。注释中带“省略”的方法与绑定无关，因此代码清单 20-7 中也不会包含。虽然定义了专门用于绘图的方法的范畴，但是本书中不做详细介绍。请下载源码查看。

代码清单 20-6 QuadraticView 的接口部分 (QuadraticView.h)

```
#import <Foundation/Foundation.h>
#import <AppKit/NSView.h>
#import <AppKit/NSKeyValueBinding.h>
#define XY_MAX      12                  /* xy 平面的大小 */
#define COEFFS      3                   /* 系数个数 */

@class NSImage;

@interface QuadraticView : NSView
{
    NSMutableDictionary *coefficients;  /* 保存系数的字典 */
    NSImage *cache;                     /* 绘图用的缓存 */
    NSMutableArray *gridArray;          /* 用于绘制网格和 xy 轴 */
}
- (id)initWithFrame:(NSRect)rect;      /* 省略 */
@property(copy) NSMutableDictionary *coefficients;
- (void)drawRect:(NSRect)aRect;        /* 省略 */
- (void)moveGraph:(NSPoint)d;          /* 为了移动图形而进行系数计算 */
- (void)mouseDown:(NSEvent *)event;    /* 监视鼠标拖拽 */

/* Key-Value Binding */
- (void)bind:(NSString *)binding toObject:(id)obsvObj
    withKeyPath:(NSString *)keyPath
    options:(NSDictionary *)options;
- (void)observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object change:(NSDictionary *)change
    context:(void *)context;
@end
```

代码清单 20-7 中展示了 QuadraticView 的实现部分。但省略了和绑定无关的部分。

首先，实例变量数组 bindObj 和数组 bindKey 被用来在该 QuadraticView 实例被绑定时保存绑定对象和键路径。

声明属性 coefficients 为可变字典对象，包含 3 个系数的值。当该属性被修改时，图也随之改变。除了采用这种实现之外，例如将系数保存在 double 型的数组中也是可以的。

下面首先说明一下方法`bind:toObject:withKeyPath:options:`。该方法会检查被作为绑定名传递的第一个参数是否为数组 BindKeys 的元素，如果不是就交由超类来处理。如果是则说明指定了某个系数，这种情况下就要对第二个参数对象 obsvObj 和第三个参数键路径 keyPath 设定键值观察，并指定 self 为观察者。此时，虽然会将 BindKeys 数组对应的元素指针作为`context:`的参数传入，但由于使用了 ARC，为了传入 void * 类型，必须使用 __bridge 修饰符来转换。被指定的对象 obsvObj 和键路径分别被保存在数组中。

然后我们再来看方法`observeValueForKeyPath:ofObject:change:context:`。在上述设定键值观察后属性发生了变化时，该方法会被调用。使用该类时，必须查看 3 个系数中哪个属性发生了变化。这里，由于第 4 个参数 context 中会返回之前传入的数组 BindKeys 的元素指针，因此只要查看一下就能知道是哪个系数发生了变化。但是这里也必须使用 __bridge 做转换。知道了对应的系数后，改变字典对象 coefficients。之所以将变量 coefficient 自身作为参数来调用设置器`setCoefficients:`，一是为了绘图，二是为了使属性 coefficients 被监视时也能够对应。

像这样，希望从控制器传入模型值的变化时，可以使用方法`bind:toObject:withKeyPath:options:`设定模型属性的键值观察。由于模型改变时方法`observeValueForKeyPath:ofObject:change:context:`就会被调用，因此知道哪个属性发生了变化，就能将其反映到自身取值的变化上。

下面是从视图侧改变模型值的情况。

方法`moveGraph:`会根据参数结构体表示的移动量更新系数 b、c，同时移动图。而系数 a 不变。这个方法`moveGraph:`从`mouseDown:`中调用。方法`mouseDown:`为 NSView 类的方法，用来跟踪鼠标被点击后的行为。

方法`moveGraph:`在字典对象 coefficients 改变后调用设置器`setCoefficients:`重新进行绘图。最后，因为需要改变模型值，所以这里使用了数组 bindKey 的键路径来设定数组 bindObj 元素的值。这两个数组的值，也就是方法`bind:toObject:withKeyPath:options:`被调用时设定的控制器和模型的属性的键路径。

代码清单 20-7 QuadraticView 的实现部分（QuadraticView.m 的一部分）

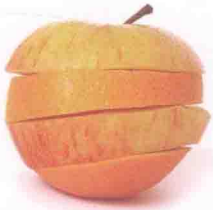
```
#import "QuadraticView.h"           // 使用 ARC
#import "QuadraticView+Draw.h"
#import <Cocoa/Cocoa.h>

static NSString *BindKeys[] = { @"a", @"b", @"c" };

@implementation QuadraticView {
    __weak id bindObj[COEFFS];  /* 保存绑定到各系数的对象 */
    id bindKey[COEFFS];         /* 各对象的属性的键路径 */
}
```

```
@dynamic coefficients;
- (NSMutableDictionary *)coefficients { return coefficients; }
- (void)setCoefficients:(NSMutableDictionary *)val
{
    if (coefficients != val)
        coefficients = [val mutableCopy];
    [self redrawGraph];
    [self display];
}

- (void)moveGraph:(NSPoint)d  /* 移动图 */
{
    double a = [[coefficients valueForKey:@"a"] doubleValue];
    double b = [[coefficients valueForKey:@"b"] doubleValue];
    double c = [[coefficients valueForKey:@"c"] doubleValue];
    // new b := b - 2ax,  new c := c + ax^2 - bx + y
    double w = a * d.x;
    id bval = [NSNumber numberWithDouble:(b - 2.0 * w)];
    id cval = [NSNumber numberWithDouble:(c + (w - b) * d.x + d.y)];
    [coefficients setObject:bval forKey:@"b"];
    [coefficients setObject:cval forKey:@"c"];
    [self setCoefficients: coefficients]; // KVC & redraw
    /* 改变模型值 */
    [bindObj[1] setValue:bval forKeyPath:bindKey[1]]; /* b */
    [bindObj[2] setValue:cval forKeyPath:bindKey[2]]; /* c */
}

- (void)mouseDown:(NSEvent *)event
{
    /* 捕捉鼠标拖拽并移动图 */
    NSPoint p, prev, delta;
    const NSUInteger DRAG_MASK =
        (NSLeftMouseUpMask|NSLeftMouseDraggedMask);
    double scale = (XY_MAX * 2.0) / [self frame].size.width;

    p = [event locationInWindow];
    prev = [self convertPoint:p fromView:nil]; /* View based point */
    for ( ; ; ) {
        event = [[self window] nextEventMatchingMask:DRAG_MASK];
        if ([event type] == NSLeftMouseUp)
            break;
        p = [event locationInWindow];
        p = [self convertPoint:p fromView:nil];
        delta.x = scale * (p.x - prev.x);
        delta.y = scale * (p.y - prev.y);
        [self moveGraph: delta];
        prev = p;
    }
}

- (void)bind:(NSString *)binding toObject:(id)obsvObj
   withKeyPath:(NSString *)keyPath options:(NSDictionary *)options
{
    int i;
    for (i = 0; i < COEFFS; i++)
```

```
        if ([binding isEqualToString: BindKeys[i]]) break;
    if (i < COEFFS) {
        [obsvObj addObserver:self forKeyPath:keyPath
            options:NSKeyValueObservingOptionNew
            context:(__bridge void *)BindKeys[i]];

        bindKey[i] = keyPath;
        bindObj[i] = obsvObj;
    }else
        [super bind:binding toObject:obsvObj
            withKeyPath:keyPath options:options];
}

- (void)observeValueForKeyPath:(NSString *)keyPath
   ofObject:(id)object change:(NSDictionary *)change
   context:(void *)context
{
    int i;
    for (i = 0; i < COEFFS; i++)
        if (context == (__bridge void *)BindKeys[i])
            break;

    if (i < COEFFS) {
        [coefficients setObject:[object valueForKeyPath:keyPath]
                         forKey:BindKeys[i]];
        [self setCoefficients: coefficients]; // KVC & redraw
    }else
        [super observeValueForKeyPath:keyPath
            ofObject:object change:change context:context];
}

@end
```

像这样，从自定义视图改变模型值时，将最初绑定时获得的模型属性，用键值编码更新就可以了。而控制器中绑定的其他组件也会收到改变的通知，这样就实现了更新值共享。

通过对该软件加以改良，就可以很容易地根据系数值计算出二次方程的根并将其显示出来。创建根据系数计算、显示根的对象，然后再和模型绑定就行了。



Objective-C 编程全解

Objective-C首选教程，6年长销第3次改版

- **全面深入**
  从内存管理到并行编程，完美涵盖Objective-C的方方面面
- **符合东方人思维**
  日本资深开发者操刀撰写，更易理解消化
- **讲解细致**
  理论结合实例，代码支持最新Mac OS X/iOS系统

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/程序设计

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-37719-7

ISBN 978-7-115-37719-7

定价：79.00元