移动开发经典丛书

Objective-C 开发经典教程

[美] James Dovey
Ash Furrow 著
冯宝隆 于鹏飞 译

清华大学出版社
北 京

James Dovey, Ash Furrow
Beginning Objective-C
EISBN：978-1-4302-4368-7

北京市版权局著作权合同登记号　图字：01-2013-5118

图书在版编目(CIP)数据

Objective-C 开发经典教程/(美) 达维(Dovey, J.)，(美) 弗罗(Furrow, A.) 著；冯宝隆，于鹏飞 译.
—北京：清华大学出版社，2014
(移动开发经典丛书)
书名原文：Beginning Objective-C

ISBN 978-7-302-34667-8

Ⅰ. ①O…　Ⅱ. ①达…　②弗…　③冯…　④于…　Ⅲ. ①C 语言—程序设计—教材　Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2013)第 291226 号

责任编辑：王　军　杨信明
装帧设计：牛艳敏
责任校对：邱晓玉
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：21.5　　**字　　数**：523 千字
版　　次：2014 年 1 月第 1 版　　**印　　次**：2014 年 1 月第 1 次印刷
印　　数：1~4000
定　　价：59.80 元

产品编号：

作 者 简 介

James Dovey 已经独立为 Macintosh(以及之后的 iOS)编写了 12 年软件。他出生于英国，在加拿大多伦多柯保公司(Kobo)工作，在那里一直担任公司 iOS 应用程序首席架构师，但最近他担任出版业和各种标准委员会的联系人，并且在办公室里扛着一个上面写着“实现 ePub 3”的大棒子(实际上这个大棒子看起来像是某种马里特木锤——读者可自行用 Google 搜索它)。作为一个黑客(这是个问题吗？请问我们能把它当作一个问题吗？)，他是很多开源项目的建立者，这些开源项目包括 AQGridView，该项目是原始的 iOS 表格视图控件；AQXMLParser，iPhone 最好的基于事件的 XML 解析器；以及原始的 Apple TV 第三方开发工具。他还开发了 Outpost，即最初的 iPhone Basecamp 客户端，还建立了基于 Apple TV 的数字信号系统。本书是他的第一本书，但他希望将来能出更多的书。

Ash Furrow 从 iOS 2 开始就在编写 iOS 应用程序。在完成他的学士学位时，他开发了用于地方选举的 iOS 应用程序并在新不伦瑞克大学教授 iOS 开发。他还开发了几个自己的应用程序(在 App Store 上销售)并发布开源项目。在 2011 年，他为了在 500px 工作移居到多伦多并开发了现在广为流行的 iOS 应用程序。

当前，Ash 是 500px iOS 组的首席开发者。他还喜欢发推特、写博客和摄影。

技术审校者简介

Felipe Laso Marsetti 是自学成才的 iOS 软件开发者，他当前作为系统工程师受雇于 Lextech Global Services。尽管过去使用过很多语言，但他认为没有比开发 iPhone 和 iPad 项目更让他高兴的事情了。Felipe 有超过两年的 iOS 专业经验。他喜欢在 http://iFe.li 上写博客，在 www.raywenderlich.com 上撰写 iOS 教程和文章，以及担任 Objective-C 和 iOS 相关书籍的技术审校者。Felipe 的推特账号为@Airjordan12345，Facebook 账号为其名字，App.net 账号为@iFeli。在他不工作或编程时，他喜欢阅读、学习新语言和技术、看体育比赛、烹饪或弹吉他和拉小提琴。

致　　谢

如果不是在 2009 年苹果世界大会上偶遇 Jeff LaMarche，这一切就不会发生，之后他在那一年的苹果全球开发者大会上把我介绍给了 Apress 的 Clay Andres。Apress 大家庭的作者和编辑给了我很大的帮助和鼓励，尤其是 Felipe Laso Marsetti，他在确保本书中有价值信息的正确性方面提供了无价的帮助，还有编辑 Katie Sullivan、Douglas Pundick 与 Steve Anglin，他们尤其应该得到奖励，因为在过去的一年里他们忍受了我道格拉斯·亚当斯式的应对最后期限的方法。

——James Dovey

我得到了很多帮助，包括在我撰写本书内容时和获得一个职位时得到的帮助。在该职位上我得到了足够的经验，从而能帮助编写这本书。没有人能只靠自己获得成功，每个人都会在他们的前进道路上得到帮助。实在有太多朋友、老师和导师需要感谢。我专门把我的想法讲给 Jason Brennan 和 Paddy O'Brien 听，把我的文章给他们看，他们在帮助我完善写作方面总是能提供无价的帮助，感谢他们的敏锐眼光。

在我撰写这本书期间，我的妻子给了我绝对的支持，她容忍我在深夜和周末继续工作，没有她本书就不能得以顺利付梓。

——Ash Furrow

目　录

第 1 章

Objective-C 入门

Objective-C 编程语言有一段很长的历史，但在过去的大多数时间里，Objective-C 只是一种一直被冷落在边缘地带的潜力级语言。iPhone 引入它之后，迅速给它带来了声誉(或者骂名)。2012 年 1 月，TIOBE 官方宣布 Objective-C 获得了 2011 年的 TIOBE 编程语言奖。这个奖是颁发给在之前 12 个月中使用量增长最快的语言的。对于 Objective-C 而言，2011 年它的使用量从 TIOBE 排行榜的第 8 位快速提升至第 5 位。你可以在图 1-1 中看到它快速、迅猛的上升趋势。

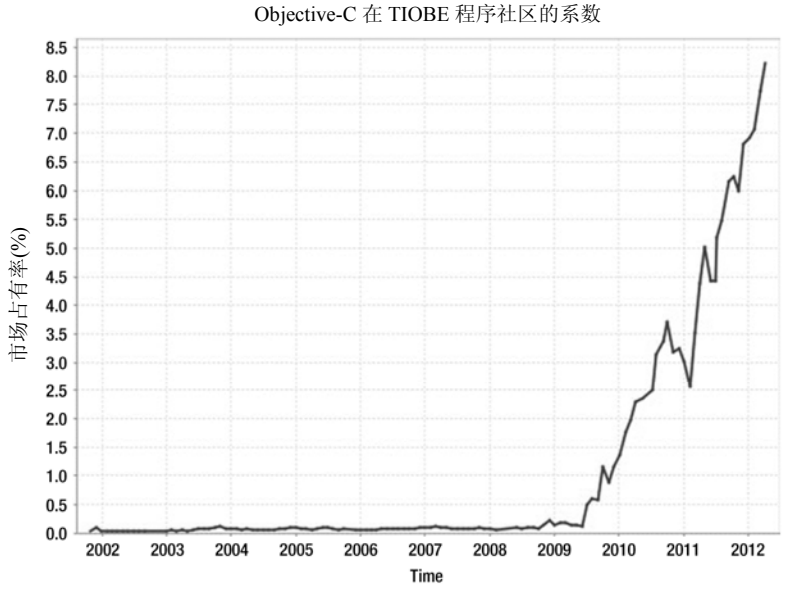

图 1-1　2002 年 1 月至 2012 年 1 月 Objective-C 的使用趋势

Objective-C 编程语言诞生于 20 世纪 80 年代初期，由 StepStone 公司的 Brad Cox 和 Tom Love 创建。它的设计目的是，使用 C 语言将 Smalltalk(由 Xerox PARC 公司于 20 世纪 70 年代创建)的面向对象编程思想带给全世界使用 C 编程语言实现的软件系统。1988 年，Steve

Jobs(对，就是那个大家都知道的 Steve Jobs)授权在 NeXT 操作系统中使用来自 StepStone 的 Objective-C 语言和运行时。NeXT 也在 GCC 中嵌入了 Objective-C 编译器，并且开发了 FoundationKit 和 ApplicationKit 框架，这些工作直接为 NeXTstep 操作系统的编程环境打下了基础。虽然 NeXT 电脑并没有席卷全球，但它使用 Objective-C 构建的开发环境已经在软件产业圈内得到了广泛的好评。在 20 世纪 90 年代中期，这个操作系统最终发展成为 OpenStep 标准，被 NeXT 公司和 Sun Microsystems 公司所使用。

1997 年，正在为下一代操作系统寻找一个坚实基础的苹果公司收购了 NeXT 公司。NeXTstep 操作系统从那时起成为 Max OS X 的基石，并于 2001 年早期发布了第一个 Max OS X 的商业版本。当旧的 Mac OS 系统可以兼容 AppKit 和 Foundation 库(从那时起就在市场上被称作 Cocoa)之后，这两个库构成了 OS X 上新的编程环境的核心。NeXT 公司的编程工具——Project Builder 和 Interface Builder——可以在 Mac OS X 上免费下载，但直至 2008 年，它才有了支持 iPhone SDK 的版本。从此，众多程序员开始积极为这个让人激动的新设备编写程序，Objective-C 也开始了它的腾飞之路。

在本章你将要学习如何使用 Xcode 编程环境创建一个简单的 Mac 应用程序，包括实现 UI 布局与用户交互。你将会接触到一些 Objective-C 语言本身的细节，如关键字、结构体、Objective-C 程序的格式，以及语言本身提供的一些功能。

1.1 Xcode

为 Mac 和 iPhone 编写程序主要使用苹果公司的免费工具集，该工具集主要扩展了 Xcode 集成开发环境(IDE)。过去，存放 OS X 副本的光盘会附带 Xcode，也可以通过前往 Apple Developer Connection 下载有效的 Xcode。但是近来 Xcode 主要通过 App Store 获取。打开你的 Mac 的 App Store 应用程序，在搜索栏中输入“Xcode”并且按下回车键，就可以找到它，如图 1-2 所示。

图 1-2　最新版本的 Xcode 可以在 Mac App Store 上免费下载

单击下载它，等待一段时间后就可以在 Applications 文件夹中找到可用的 Xcode 了。

与其他同名的 IDE 应用程序相比，Xcode 附带了很多东西。它包含了很多有用的调试和配置功能，而且可以选择下载 GCC 和 LLVM 编译套件的命令行版本。你将找到的现有工具中包含如下这些工具：

- Instruments：该工具可以为你的应用程序生成详细的运行时配置信息——这很可能是所有工具中对 Max 或 iOS 开发者最有用的工具。
- Dashcode：一个 HTML 和 JavaScript 编辑器，旨在帮助你轻松构建 Dashboard 组件和 Safari 插件。
- Quartz Composer：这个应用程序允许你使用无代码拼接组装技术创建复杂的图形转换、过滤器和动画。
- OpenGL Apps：为了使用 OpenGL(以及 iOS 上的 OpenGL ES)进行工作，该应用程序提供了完整的应用程序套件。在该套件中可以找到配置文件、性能监视器、着色器生成器，以及 OpenGL 驱动监视器。
- Network Link Conditioner：对于基于网络的软件工程师而言，一个梦想已经变为现实，这个方便的小工具可以让你模拟不同网络配置的主机。默认情况下，它模拟最常遇到的环境。你也可以设定带宽、丢包率、延迟和 DNS 延迟，创建自己的网络环境。想要测试你的 iOS 应用程序在 Wi-Fi 网络极边缘地带时会如何处理吗？使用这个小工具会使其变得完美且简单。

以上这些只是我们最喜爱的工具中的一部分，但它决不是一个详尽的清单。你将在本书的后文中看到很多更加令人印象深刻的 Xcode 工具背后的技术。

1.2　创建你的第一个项目

第一次打开 Xcode 时，会看到 Xcode 的欢迎界面。下面的步骤能引导你创建一个新的项目。

(1) 单击名为 Create a new Xcode project 的按钮，然后选择要创建项目的类型。

(2) 在 Mac OS X 选项中，选择 Application，然后在主窗格内选择 Cocoa Application 图标。

(3) 单击 Next 按钮，会出现一些定义项目的选项，输入图 1-3 所示的细节信息，然后再次单击 Next 按钮并且选择项目存放的位置。

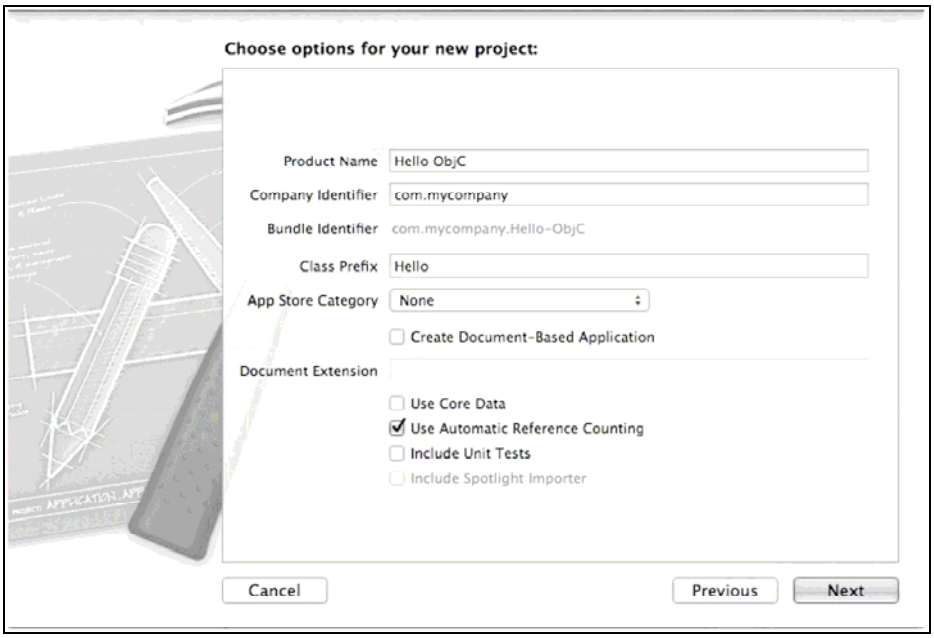

图 1-3　你的第一个项目的选项

浏览一下 Xcode 和这个新项目的大致布局。在窗口的左边可以看到导航栏，如图1-4 所示。在此可以浏览项目的源代码文件、资源、库和输出文件。这个导航栏也能让你浏览项目的类层次结构，在整个项目中执行查找和替换工作，以及浏览编译日志。

Xcode 窗口中间的窗格是编辑区。可以在此编写代码以及使用你的用户界面资源。

右边是实用工具窗格。上侧则会根据当前在编辑器窗格的焦点的内容而敏感地显示不同的选项卡。下侧是分隔板面板，可以从这里拖出用户界面元素、基于模板的新文件、代码片段以及媒体。也可以在这里添加自己的模板和代码片段。

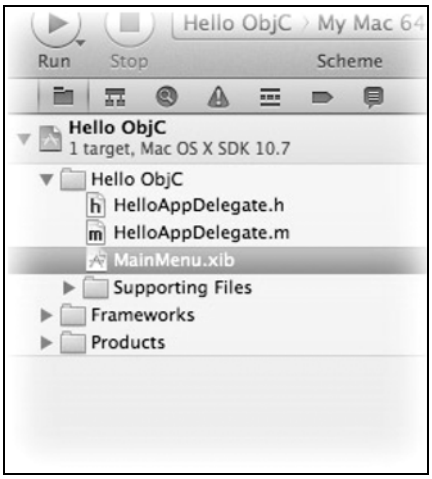

图 1-4　Xcode 导航栏窗格

1.2.1　应用程序模板

Cocoa Application 模板已经为你生成了很多信息。实际上，在这里你已经有了一个功能完善的应用程序。在导航栏选择浏览器选项卡(最左侧的选项)并且查看 Hello ObjC 文件夹中的内容，会看到一些主要的源文件和用户界面定义文件(一个.xib 文件)。这里还有一个 Supporting Files 文件夹，它包含了应用程序的 main.m 文件，该文件是整个应用程序自身的入口，同时作为前缀部分自动添加到项目中每个文件的开头。你还能看到 Hello ObjC-Info.plist 文件，它包含了应用程序的元数据；还有 InfoPlist.strings 文件，它保存了 plist 文件中的本地化版本的数据。通常不需要直接改动这些文件，因为 Info.plist 一般通过目标编辑器进行编辑，本书会在后面的章节中对此进行介绍。

在此你可能需要改动的一个文件是 Credits.rtf。这个文件的内容会显示在应用程序的 About 对话框中。而且因为它是一个.rtf 文件，所以你能随心所欲地设计它。这些内容将会显示在 About 对话框下可滚动的多行文本框中。

再向下是 Frameworks 文件夹。它包含了一个你的应用程序运行所基于的所有框架和动态库的列表。需要注意的是，这不是一个自动管理的列表：需要在使用框架(framework)和库时自己将它们添加到项目中。最后，Products 文件夹包含了一个指向编译过的应用程序的引用。现在它的名字呈红色显示，那是因为你还没有构建它。

单击 HelloAppDelegate.h，可以在编辑器窗格中看到它的内容。现在它看起来有点空，就像程序清单 1-1 显示的那样。这段代码声明了一个类的结构和接口，在这个例子中这个类的名称是 HelloAppDelegate。这段代码通知系统该类执行所有定义在名为 NSApplication-Delegate 的协议中的必要方法，并且该类有一个名为 window 的属性。在第 2 章中将介绍这个语法的细节部分，至于现在，你只需要相信它是按照我们的期望工作的。

程序清单 1-1　HelloAppDelegate.h

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

@interface HelloAppDelegate : NSObject <NSApplicationDelegate>
@property (assign) IBOutlet NSWindow *window;
@end
```

接下来介绍执行文件，参见程序清单 1-2。现在它的内容同样很简单。在定义 HelloApp-Delegate 类的实现的一些分隔符中，可以看到一个名为@synthesize 的指令，它看起来与你刚刚看到的 window 属性有关。事实正是如此，这个指令可以让 Objective-C 编译器生成获取(get)和设置(set)window 属性的方法，为你节省了自己编写这些方法所需的时间。它同时指定了用来存储该属性的实例成员变量的名称为_window。编译器也会为你创建该成员变量，这也为你节省了显式编写它的时间。

程序清单 1-2　HelloAppDelegate.m

```
#import "HelloAppDelegate.h"

@implementation HelloAppDelegate
@synthesize window = _window;
```

```
- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{
    // Insert code here to initialize your application
}

@end
```

1.2.2 界面生成器

如果选择了 MainMenu.xib 文件，编辑区会切换至界面生成器模式。之所以命名为“界面生成器”，是因为生成用户界面的任务直至最近才被改为一个名为“界面生成器”的单独应用域。你能看到如图 1-5 所示的界面生成器。

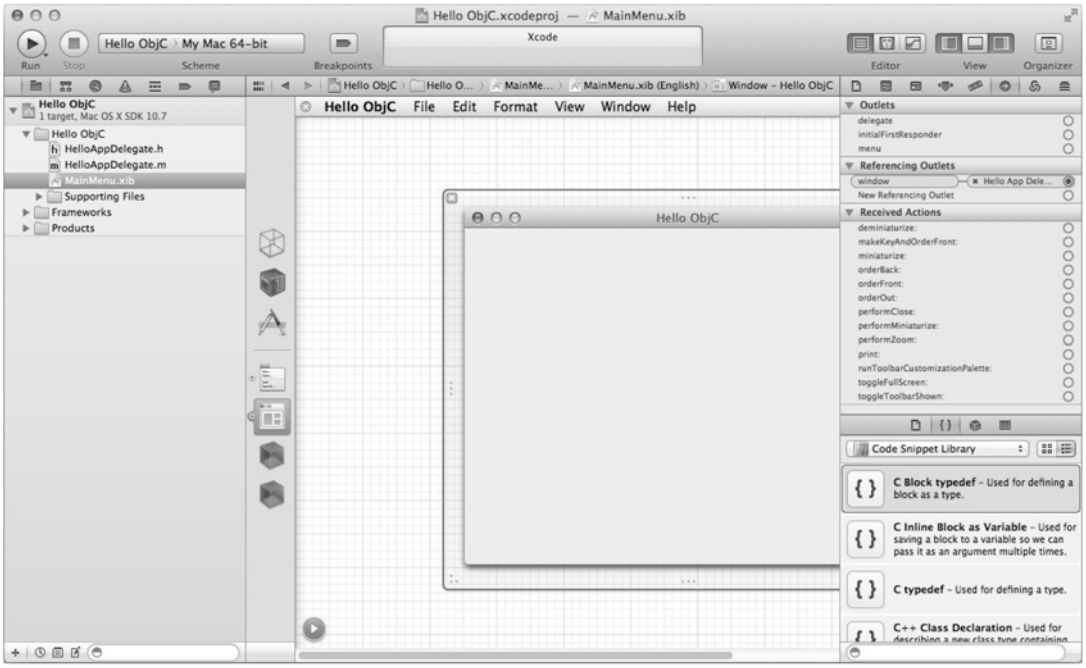

图 1-5　界面生成器

在图 1-5 中，可以看到应用程序的菜单。编译器的左下方是文档缩略图。这个界面文档中的所有对象都会在此列举。顶部是一些“推断(inferred)”对象，它们会出现在所有(或者是几乎所有)的.xib 文档中。分割线的下方是已经被显式添加到.nib 文件中的对象。列表中的第二项是应用程序的窗口。选择后可以让它出现在编辑器中。

现在你已经选中了它，Xcode 窗口右侧的实用工具窗格上方出现了很多选项卡。亲自点点这些选项卡看看会显示什么吧。将鼠标悬停在选项卡的小图标上，就会显示一个工具提示，告诉你这个选项卡的名称。

界面生成器具有很多强大的幕后智能特性，使你可以通过用户的输入和动态反馈构建优秀的应用程序。更重要的是，只需要给项目添加三行代码就能完成这件事情。

1.2.3　用户界面控件

首先，需要给用户提供一个可供输入的位置。可以从实用工具窗格下半部分的对象面板中取出控件。在图 1-6 中能看到你将用到的所有控件。

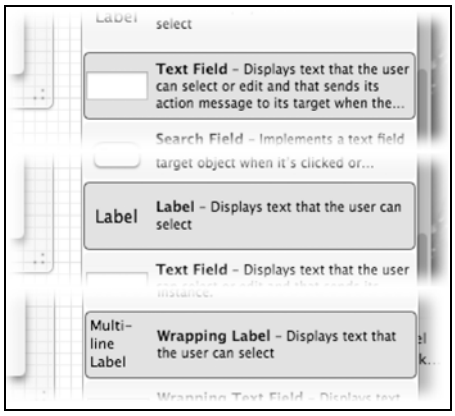

图 1-6　Text Field、Label 和 Multiline Label 控件

(1) 在实用工具窗格的下方，选择从右侧数的第二个选项卡(图标是个盒子)，切换到用户界面 Object Palette。

(2) 向下拉 Object Library 弹出式菜单，并且选择 Controls，将面板的内容限制为此时只显示标准控件(见图 1-6)。

(3) 需要找到的第一个控件是 Text Field。略微向下滚动就能找到它。

(4) 现在将该行从面板直接拖动到编辑器的窗口。你会发现这样做能使它变成一个真正的文本框。

(5) 将它向上拖动到窗口内容区域的右上角，会出现蓝色的引导线协助你定位。就把它放在那里，如图 1-7 所示。

图 1-7　放置 Text Field 控件

(6) 接下来，需要找到 Label 控件(可以认为它是一个没有特殊背景色的、不可编辑的 Text Field 控件)。

(7) 将 Label 拖动到界面左上角，但需要注意的是，在把它拖动到左上角的过程中，划过刚放置的 Text Field 控件的底部和其内部基线时也会出现蓝色引导线。后者就是你要找的地方。将 Label 放在该位置，如图 1-8 所示。

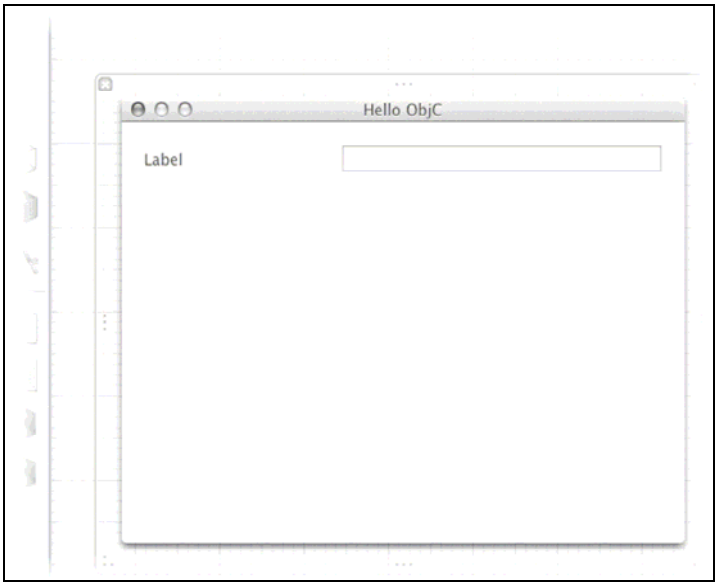

图 1-8　放置 Label 控件

(8) 双击 Label 可以编辑它的内容。输入“Your Name:”，然后按下回车键保存这个改动。现在同时按下⌘和=，让 Label 的大小变得跟它的文本大小一致。

(9) 选中 Text Field，并且将鼠标光标向 Text Field 的左侧边缘处移动，直到光标变为可以重定义尺寸的模式为止(一对向左和向右指向的箭头)。按下鼠标左键并且将 Text Field 的边缘向 Label 拖动，直至出现蓝色引导线为止。

(10) 最后，在对象库中找到 Wrapping Label 控件，并且将它拖动到窗口的中间，比 Text Field 略微偏下的位置。会出现更多的引导线帮助你将 Wrapping Lable 控件移动到窗口横向中心的位置并且正确地放置在 Text Field 的下方。我们建议你把它向下移动得远一些，使它周围可以有更多的空间。

(11) 将 Warpping Lable 的边缘拖动到窗口左侧和右侧最边缘的引导线位置，这可以使文本显示较多的内容。现在单击位于窗口下边缘中心的手柄，略微向上拖动来缩小窗口，这样这里就没有太多空闲的空间了。

1.2.4　界面绑定

如果在学习 Objective-C 之前使用过其他的语言，你可能会有这样的想法：通过挂钩(hooking)引用可变 UI 元素的变量来操控 UI。但在 Cocoa 中，这并不总是必要的。相反有一个被称为 Key-Value Coding(KVC)的系统，通过它可以观察指定对象中的指定值，该值通

过键进行引用。这个键可以是方法名或成员变量名——大多数时候是方法名。你之前看到的属性声明实际上生成了精确符合 KVC 要求的代码，即你将如何引用和存储变量值。

我们将在后面的章节对 KVC 做更全面深入的介绍，但现在需要使用一项基于它的技术：绑定。这个想法的实质是使用 KVC 将用户界面元素绑定到指定的值。这意味着当其中一个发生变化时，另一个也跟着变化。编辑 Text Field 会改变绑定到 Text Field 的值，反之亦然。很多 UI 元素的属性可以使用这种方式进行绑定，但现在你要先将重点放在最重要的地方：元素的值。

对于 Text Field 而言，元素的值是一个字符串。所以首先必须找个地方创建一个字符串属性，该属性将其与界面绑定。为了实现这一点，打开 HelloAppDelegate.h，在已有属性的下方输入一行新的内容(参见程序清单 1-3 中突出显示的代码行)。

程序清单 1-3　userName 属性

```
@interface HelloAppDelegate : NSObject <NSApplicationDelegate>

@property (assign) IBOutlet NSWindow *window;
@property (copy) NSString * userName;

@end
```

这行代码声明 HelloAppDelegate 有一个名为 userName 的属性，并且它是一个字符串，同时也声明设置这个字符串时是复制了它而不是引用了它。如果你暂时还不理解这句话的意思，不要担心，很快你就能理解了。现在只需将它当成是正确的即可。

但这只声明了属性，实际执行它还需要进一步的操作。打开 HelloAppDelegate.m，输入程序清单 1-4 中突出显示的这行代码。

程序清单 1-4　合成 userName 属性

```
@implementation HelloAppDelegate

@synthesize window = _window;
@synthesize userName;

@end
```

现在你已经告诉编译器要执行合成操作。请注意：与 window 属性不同的地方是，你已选择不提供幕后属性的成员变量的名称。这通常意味着成员变量的名称与属性的名称相同。

接下来的步骤都是在界面生成器中操作的。再次打开 MainMenu.xib。

合成变量名称

有很多不同的方法可命名属性和相应的实例变量。每个开发者都有自己的喜好，但我们偏向于让编译器自己处理实例变量。

以下是两种最常见的方法：

- 没有指定名称：编译器会使用完全相同的名称创建变量[1]。
- 使用带下划线前缀的名称：这与苹果内部对实例变量命名的方式相匹配。我们与很多其他程序员都是这样做的，但是过去苹果已经建议过我们不要这样做，因为这样的命名方式会与苹果未来可能为类新增的变量名称产生潜在的冲突。

一些人认为应该在合成属性时明确提供变量名，但我们采取的是相反的方法，因为我们相信应该鼓励使用访问方法而不是直接访问基础变量。当非常深入地访问属性时，这就变得尤为重要：这个访问过程是同步的，而且已经对变量加锁，所以无法从辅助线程中读取一个正在被修改的变量，但是直接访问实例变量是无法获得保障的。

1. 绑定用户输入

(1) 选择 Text Field。

(2) 在实用工具窗格中，可以看到它的一些属性。在第 4 个选项卡下，能看到 Attributes 检查器(inspector)，可以在其中调整 Text Field 的一些属性，例如字体、颜色以及一些行为。第 5 个选项卡是 Size 检查器，可以在其中调整 Text Field 的大小和位置，以及当调整它所在视图(在本例中，就是那个窗口)的尺寸时，它所需要执行的操作。第 6 个选项卡是 Connections 检查器，你稍后会在本书中看到关于它的介绍。再后面是 Bindings 检查器，这是用来将 Text Field 的值与某个操作进行挂钩(hook)的地方。

(3) Bindings 检查器的最上方有一行可弹出的标题，标着“Value”。打开它可以看到很多选项。

(4) 最上方是 Bind to:弹出式菜单。可以在这里看到应用程序自身的引用、文件所有者(这是一个可以在运行时处理.xib 文件中接口定义的对象)、全局字体管理器和用户默认设置，以及应用程序的委托对象——Hello App Delegate。

委托

委托的概念并不是 Objective-C 特有的，但由于这个语言的动态特性，委托成为系统库使用的核心技术之一。委托对象是遵循一些预定义协议(一个它同意执行的方法的列表)的对象，其他对象可以通过这些协议要求它执行一些操作或者决定另一些对象的行为。例如，Text Field 委托能检查正在输入的文本并且通知 Text Field 拒绝特定的字符。

委托机制是一个非常强大的工具，这也是 Objective-C 很少倾向于使用子类(例如 Application 类)的原因。相反，委托对象是为解决重大事件而创建的，并且使 Application 实例独立，从而处理程序的运行。

(5) 在弹出的菜单中选择 Hello App Delegate。在 Model Key Path 框内输入“self.userName”。

1 译者注：在 Xcode 4.6 以后，你可以不使用@synthesize 语句，编译器会自动帮你对执行过@property 的属性生成带有@synthesize 的声明。即对于@property (copy) NSString * userName，如果我没有写@synthesize，相当于写入@synthesize userName=_userName。但是如果写过了@synthesize userName，就应该使用 userName 进行调用。

(6) 选中 Continuously Updates Value 复选框。输入结果应该如图 1-9 所示。

现在这个 Text Field 的值已经绑定至之前我们创建的属性。当用户在这个 Text Field 中输入内容时，这个属性的值也会相应地更新。

下一步要为这个值创建一些输出方式。

(1) 选中 Multiline Label，并且再一次打开相应的 Bindings 检查器。然而这次你不仅要在这里设置值，而且还要提供一种用 bound 值扩展的、类似于格式化字符串的模式。

(2) 打开 Display Pattern Value1 项，界面看起来与图 1-9 很相似，比之前多了 Display Pattern 值。这里默认填写的值是“%{value1}@”，这是一种将这里创建的 Value1 的值应用到 Label 的方法。你将在此像之前那样再次绑定同样的属性。

(3) 在弹出式菜单中选择 Hello App Delegate。在 Model Key Path 框内输入“self.user-Name”。

(4) 现在编辑 Display Pattern 框，它的值是“Hello, %{value1}@!”，这会使当给定值为“user”时，Label 将显示“Hello，user!”。应该在检查器中输入如图 1-10 所示的内容。

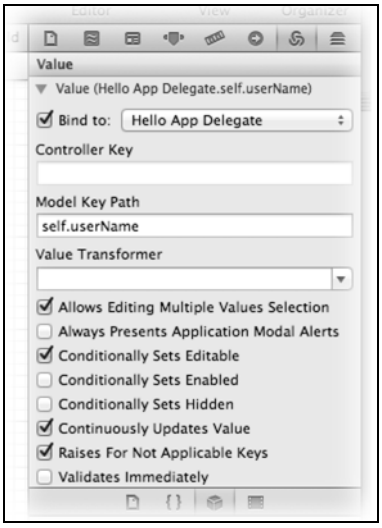

图 1-9　绑定文本字段的值

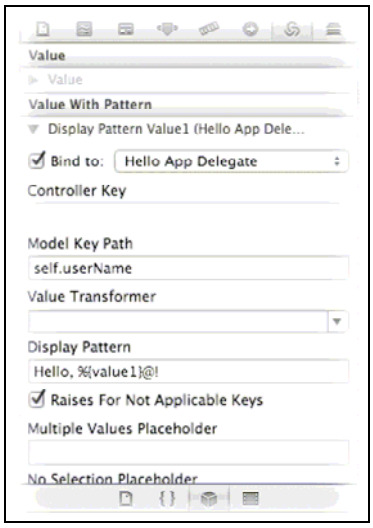

图 1-10　格式化输出字段

(5) 最后，切换到 Attributes 检查器(边栏上的第 4 个选项卡)，并且将字段的对齐方式改为居中(在 Alignment 处设置)，如图 1-11 所示。

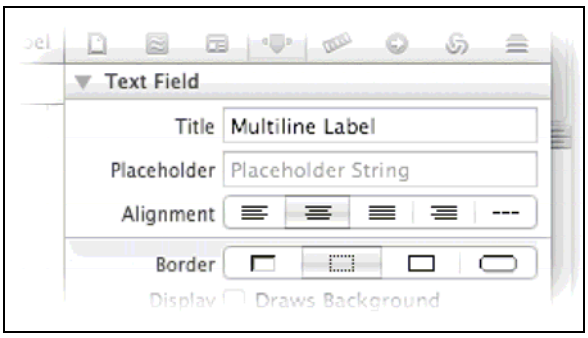

图 1-11　设定输出字段居中

1.2.5 运行应用程序

如果你知道现在已经几乎完成了整个应用程序，可能会很惊喜。现在可以单击 Xcode 工具栏中的 Run 按钮编译和运行它了。当在 Text Field 中输入内容时，下方 Label 的输出内容会动态更新。

但在第一次运行应用程序时看起来好像有点奇怪。Text Field 不包含任何内容，所以 Label 会显示“Hello,!”，而且这段字符看起来并不那么令人印象深刻。也许在程序启动时提供一个默认值会更好。实际上，预设当前用户的全名可能会有用。让我们来实现它。

再次打开 HelloAppDelegate.m。需要在一个空方法中填写代码，这个方法是 NSApplication 的委托协议的一部分。现在的代码如程序清单 1-5 所示。

程序清单 1-5 应用程序启动委托

```
@synthesize userName;

- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{
    // Insert code here to initialize your application
}

@end
```

这个方法是当应用程序已经启动完毕并且准备好显示窗口以及处理用户输入时，由 NSApplication 对它的委托调用的。这也是为编写的任何应用程序设置初始状态的位置。在本例中，需要使用 C 函数 NSFullUserName()获得用户名并且将其分配给 userName 属性。分配和指定属性使用类似于结构的语法，以区别于常规的方法调用。编译器在编译项目时会使用真正的 Objective-C 方法调用进行替换。输入程序清单 1-6 中的粗体代码。

程序清单 1-6 获取用户名

```
@synthesize userName;

- (void)applicationDidFinishLaunching:(NSNotification *)aNotification
{
    // Fetch the current user's name as a default value
    self.userName = NSFullUserName();
}

@end
```

现在再次运行应用程序，看一下你的修改结果(作者的结果如图 1-12 所示)。你的名字会预填在 Text Field 中。

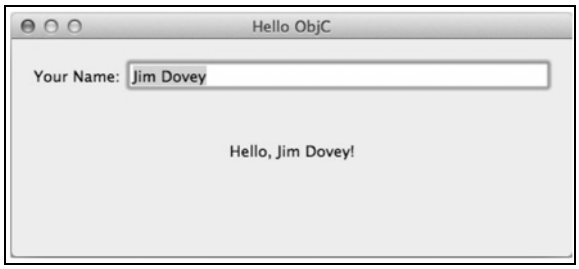

图 1-12　已经完成的应用程序

这是你本次使用 Xcode 的最后一步。我们会在本书的后面再次对这个过程中的某些部分进行介绍，但从现在开始我们的焦点要从之前介绍的工具集回到语言本身。另外，其他书籍也会讲解你需要知道的关于 Xcode 的所有内容，也可以在互联网上找到更多的相关信息。

1.3　语言基础

本节将简要回顾一下 C 编程语言的核心。我们不会面面俱到地介绍 C 语言，也不会讲解如何成为一名伟大的 C 程序员，但是会讲解足够的基础知识使你能够理解本书的剩余内容。你也将了解到如何将 Objective-C 的关键字添加到 C 语言中，尤其是如何确定你要找的到底是纯 C 语言代码还是 Objective-C 代码。

在下面的定义中，方括号“[]”内的文本表示可选元素，尖括号“<>”内的文本表示必需的元素。

1.3.1　类型和变量

C 语言(Objective-C 也一样)是一种静态类型的命令式语言。这意味着每一个变量在能够被使用之前必须声明它的类型。变量的值可以改变(这也是为什么我们称它们为“变量”的原因)，但是它们的类型不会变化。

```
double d = 4.789203;
int x = 78;
d = d + 10.0;
x = x + 5;
x = x + 8.3; // invalid-- a type error!
```

C 语言定义了一些内置的标量类型，但是其中有一部分类型占用内存的大小取决于当前使用的系统。因为程序在 OS X 和 iOS 上运行，所以只有一种要牢记的有区别的类型，即 long int 类型。

- int：32 位有符号整型。
- short int(或者只写 short)：16 位有符号整型。
- char：8 位有符号整型，一般使用单引号表示一个 ASCII 字符(例如，char c = 'a';)。

- long int(或者只写 long)：在 32 位系统(如 iOS)和更低位数的系统上，这是 32 位有符号类型(与 int 相同)。在 64 位 OS X 上，这是 64 位有符号类型。
- long long int(或者 long long)：64 位有符号整型。
- float：32 位单精度浮点值。
- double：64 位双精度浮点值。
- long double：128 位双精度浮点值。
- void：没有指定类型。只能用作函数的返回值(这个函数什么都不返回)或者指针引用的数据类型。

上面提到的所有整型类型默认都是有符号的，这意味着其中有一位是符号位。在类型的前面添加 unsigned 关键字(如 unsigned int)能够声明符号位。这能使类型中的所有位都用来表示精度，但是也只能表示正值。

C 语言也提供了结构体和数组的概念。结构体是由按特定序列排列的小类型所组成的更大类型。

```
struct telephone
{
   unsigned int area_code;
   unsigned int number;
   unsigned int extension;
};
```

数组是指按顺序存储的同一类型的相邻的块。

```
int n[10];
struct telephone directory[100];
```

1.3.2 指针

C 语言中的指针是一个很强大的概念。一个指针变量是一个引用类型而不是一个值。复制值时，会创建一个新的变量并且将其设置为输入的值。复制引用时，相关值保持不变，只是把它的内存地址信息进行了传递。一个 C 语言指针实质上是一个指定了内存大小的变量，它指向另一个变量的内存地址。所以记住指针变量的大小取决于目标 CPU 很重要。iOS 设备使用的是 32 位的 ARM CPU，所以指针大小是 32 位。另一方面，现在的 OS X 计算机使用的是 64 位的 CPU，所以指针大小是 64 位。&(与符号)运算符获取的是变量或其他符号的内存地址，而*(星号)运算符获取的是指针指向的值。

```
int i = 100;  // a regular integer variable
int *p = &i;  // a pointer to i
i = 42;       // change the value of i
int x = *p;   // x is assigned 42
*p = 200;     // i is assigned 200
```

在 C 语言中，字符串只是一个指向 char 元素的数组的指针。字符串声明的简写方式是在双引号内包含多个字符。

```
char * name = "Ernest Hemingway";
```

值得注意的是，数组实际上也是使用了简单指针语法的指针，而且这两种类型可以互换。当声明 int a[10]时，实际上它的类型是 int *。此外，数组下标访问(例如，a[5] = 7)实际上使用了 void *类型的指针，这只意味着它是“指向某些类型的指针”。

1.3.3　函数和声明

函数声明的方式与变量类似。首先会看到它们的类型(或者返回值类型，即函数计算后的返回变量的类型)，然后是名称。参数写在名称之后，括在括号中。紧随其后的是组成函数自身的语句，括在花括号中(参见稍后的 1.3.4 节“作用域”)。

```
int mean_average(int a, int b)
{
    return( (a + b) / 2 );
}

int x = mean_average(12, 4);
```

函数必须在使用前进行声明。这意味着它们必须在给定文件的使用位置之前进行声明。因为在 C 语言中定义函数也有声明的效果，所以前面的示例能够运行。但如果一个程序被分为多个文件，常见的做法是给每一个实现文件(在 C 语言中是.c 文件，在 Objective-C 中是.m 文件)都声明一个头文件(如.h 文件)来包含这些声明。当显式声明函数或变量时，需要省略它的内容并用一个分号来替代。

```
int mean_average(int a, int b);
int x;
```

这些语句简单地声明了变量或函数的存在、它们的格式以及名称，之后编译器就可以引用它们。然后可以使用#include(或者 Objective-C 的#import)指令将头文件导入到实现文件中。

```
#include "something.h"

int r = mean_average(12, 14);
```

被声明项的作用域默认情况下是开放的，这意味着你能在任何能看到它的声明的地方引用它。

1.3.4　作用域

C 语言中的作用域是指符号(例如变量或者函数)的可见性。有两种确定作用域的方法。第一种是文件作用域。任何在文件中输入的内容，如果没有其他关于作用域的限制条件，都是文件级的作用域。第二种方法是使用大括号：“{”和“}”。任何放在一对大括号中的变量或函数的作用域都仅仅是在这对大括号之中。在 C 语言中，这些方法被用来设定函数或循环的作用域，以及用来包括某些特定类型的内容，例如结构体或 Objective-C 中类的实

例变量。

在 C 语言中，关键字 extern 和 static 是唯一内置的作用域限定符。

- extern 指定符号有外部可见性。提供这个符号是为了在运行时从当前编译单元(当前文件及其包含的头文件)访问外部的内容。默认是这样的：漏写这个限定符会导致产生一个外部可见的符号。
- static 指定符号是私有的。即它的可见性仅局限于当前作用域内。这个符号对指向编译后的单个编译单元的链接器是不可见的。

这意味着，在文件的根作用域使用 static 符号进行声明，那么它将对这个文件中的一切都可见。在函数或其他语法作用域内(如一个循环、一条 if/else 语句，或者类似的语法)使用 static 来限定变量，那么这个变量只在那个有限的作用域内可见。static 关键字还规定，变量存储在由编译器预定义的内存地址中，通常由输出的二进制文件直接映射而来。因此，一个内存变量仅会在内存中出现一次，即便是在一个函数内声明它，也不会在堆栈中再创建一份新的副本。但在函数中声明变量时，这个变量的作用域将会被限制在这个函数内。

1.3.5 条件

C 语言中所说的条件主要是指使用 if 和 else 关键字。

```
if(value == 1)
{
   // do something
}
else
{
   // error!
}
```

需要注意的是，当且仅当在条件判断结果中需要执行一行命令时，才能省略大括号。而不是必须一直使用大括号，但是这样做往往能使代码更简洁，且更具可读性。

C 语言还提供了 switch 语句来处理较大的条件组。在 switch 块中，case 语句可以匹配值和执行相应的操作，break 语句可以跳出 switch 块，在没有 case 语句能够匹配待查值时执行 default 语句的操作。

```
switch(value)
{
    // 花括号可选
    case 1:
    {
       result = "yes";
       break;
    }

    case 0:
       result = "no";
       break;
    default:
```

```
        result = "maybe";
        break;
}
```

在 case 语句中，除非要在这个 case 块中声明一个新的变量，否则几乎可以完全省略大括号。为了便于阅读，常见的做法是将 case 块中的内容缩进，即使省略了大括号也如此。

请注意本例中一直都使用了 break 关键字。因为没有它，程序会继续执行下一 case 语句中的内容，就好像下一 case 语句也已经匹配一样，我们称之为“向下连通(fall-through)”。思考下面的例子：

```
switch(value)
{
    case 0:
        result = "no";
    case 1:
        result = "yes";
    default:
        result = "maybe";
}
```

在这段代码中，result 的值总会被设置成“maybe”。如果 value 是 0，那么 result 首先会被设置成“no”，然后被设置为“yes”，然后再次被设置成“maybe”。但是如果在每一个 case 语句的最后都调用 break 语句，程序就会在第一个匹配的 case 调用之后跳出 switch 块。

1.3.6　循环

C 语言中的循环是通过使用 for、while 语句和 do 关键字来实现的。for 循环可以用来初始化、检测，以及执行循环内部的下一个指令，这样循环事件可以与这些循环语句保持独立，从而显得更加简洁。这种模式常用于遍历数组的内容。

```
for(int i = 0; i < 10; i++)
{
     x = x + array[i];
}
```

while 循环会同时使用 while 和 do 关键字。它有两种形式，第一种是在每一次循环迭代开始时检查条件语句，另一种则在循环结束时检查条件语句。两者的区别是后者至少会执行一遍代码，因此更适合处理代码的重试部分。

```
while( has_more_data() )
{
    append(data, read());
}
process_data(data);

int sent_data = 0;
do
{
```

```
        sent_data = try_to_send(data);
    } while(sent_data == 0);
```

1.3.7 Objective-C 的附加功能

Objective-C 只是在 C 语言的基础上添加了一些小改动，而且几乎所有的关键字都使用@符号开头。Objective-C 字符串字面量(NSString 类的实例)是通过在常规 C 字符串之前添加@符号来声明的，例如@"A string value"。同样，可以使用类似的格式创建基于 NSNumber 的数字，例如@42 或@812.90731 会生成支持整型或浮点型的 NSNumber 实例。

Objective-C 的所有新关键字都以@开头，这是为了同标准 C 代码进行区分，因为在 C 语言中使用@符号开头的字符不是合法的符号名。例如，Objective-C 中使用@try/@catch/@finally 关键字来处理异常，使用@class、@interface 和@implementation 关键字来定义类。Objective-C 与普通 C 语言相比仅剩最后一点区别，即在 Objective-C 调用方法时需要使用方括号括起来，例如[someObject doSomething]。因为没有纯 C 指令是使用方括号开头的，所以 Objective-C 编译器可以简单地区分和识别 Objective-C 的方法调用。

在第 2 章介绍 Objective-C 语言核心时，你将会看到有关这些内容更多、更完整的介绍。

1.4 小结

本章介绍了如何使用 Xcode 创建一个简单的 Mac 应用程序，也介绍了作为 Mac 或 iOS 开发者，你将花费大量时间去使用的主要工具。本章还介绍了如何使用界面生成器，例如，使用绑定的方式几乎不用编写代码即可创建可交互的应用程序。另外，也介绍了 Objective-C 语言本身的基本构建块。

第 2 章将介绍面向对象编程的概念，以及了解如何将它们应用到 Objective-C 自身当中，从而加深你对 Objective-C 世界的了解。

第 2 章

面向对象编程

面向对象编程已不是一个新概念，但它很可能是目前这个星球上使用最广泛的编程范式。它的历史可以追溯到 20 世纪 50 年代末期和 60 年代初期，那时候人们首次在 Simula 67 语言中开始使用类和实例的概念。20 世纪 70 年代，Xerox PARC 公司扩展了这个概念，创建了 Smalltalk 语言。Smalltalk 语言系统使用面向对象的概念描述了无处不用的对象和消息的概念。Smalltalk 语言中的一切都是对象，甚至连类似于“62”这样的标量也是对象。20 世纪 80 年代，Objective-C 诞生了，它将 Smalltalk 语言面向对象的思想(和一些语法)与命令式编程的 C 语言进行了合并。

本章将为你讲解面向对象编程的基本知识，也将介绍 Objective-C 语言实现 OOP(Object-Oriented Programming，面向对象编程)的方法。到本章的末尾，你应该已经熟悉在本书后面章节中所必须理解的所有相关术语和概念，也已经理解了使用 Objective-C 语言编写的文本。

2.1 对象：类和实例

在迭代编程语言中，指令序列被拆分成一些方法，从而提供一种封装这些指令的形式。程序员进一步把功能抽象划分至不同的文件或库，同时通过其他方法组能调用的头文件来提供接口。面向对象编程更进了一步，它提供了一种使用相关的方法封装数据的手段，并且首次提出类语言结构来定义这个封装。

提出数据封装概念背后的原因相当简单：由于非 OOP 编程没有一种确切的封装数据的方法，可以显式地在不同函数中传递数据，因此经常把其中的一些数据置为全局作用域，从而使它几乎可在任何地方都能被访问到。这直接导致非 OOP 编程的程序尺寸增大，并且需要程序员自己尝试预防对这些共享资源的并发访问，如果不这样做，程序就很可能存在有重大影响的 bug。

2.1.1 封装

面向对象编程把数据和操作数据的方法一起放在对象中。数据本身通常是程序的其他部分不可访问的，并且要使用它们自身指定的方法进行保存。对象本身也能被创建和复制，这使程序很简单地就能在飞速运行中的任何时候都可以有很多组相似的数据，并且降低使用可全局访问的变量导致 bug 的可能性。

OOP 编程中的每一个对象通常被设计成代表一个单独的概念。例如，一个对象可能代表一个窗口、一个标签、指向某个指针的接口，或者一个网络数据流。这些概念中的每一个都会被定义为类——一种资源类型在字面上的分类(从语感的角度出发)。类描述了对象的结构和含义，指定了它包含的数据以及它提供的与这些数据交互的方法。当程序想要使用这些对象时，就会实例化一个给定类的对象，这意味着这个类获得了从内存中分配的空间，并且它的内部状态已经被初始化。此时，这个产生的对象就可以在它自己的权限内作为一个独立的实体进行调用了，并且独立于任何一个由同一个对象的类创建的实例。

在 OOP 编程中，对象的数据通常可以认为是内置在对象中的。外部访问者只能通过调用对象提供的指定方法来修改对象的状态，这样做有几个关键的目的。

- 保密性：确保私有数据被隐藏在没有其他对象能访问到的地方，而且也没有其他对象能访问它。
- 安全性：确保任何连接的变量和常见的状态能适当地保持同步。
- 简易性：通过聚合所有的数据和方法，执行严格的模块化方法，从而提高了程序的可维护性。

2.1.2 继承

面向对象编程的另一个重要组成部分是继承的概念。这个思路是指一个类可能是另一个类的特化。例如，圆形、矩形和立方体都属于 Shape 中的一种类型。因此你可能会编写一个 Shape 类，Circle、Rectangle 和 Cube 都会成为 Shape 类的子类。可以在图 2-1 中看到这个思路的分解情况。

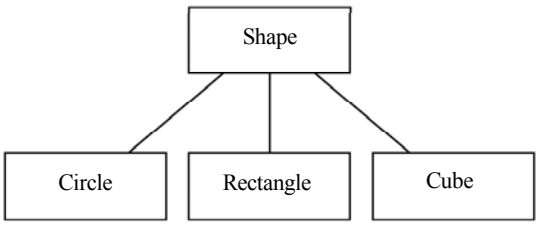

图 2-1　类的层次结构示例：Shape 类包含了 Circle、Rectangle 和 Cube 子类

通过继承 Shape 类，这三个新类在各种情况下仍会做出与 Shape 类相同的处理。Circle、Rectangle 或 Cube 适合于任何可以使用 Shape 对象的地方。

子类可以调整由 Shape 类提供的方法和接口的行为。例如，Shape 可能含有用来指示其包含的棱和面的个数的属性，而三个子类中这些属性的值可能不同，但它们都是一样的属性。因此，唯一需要做的就是声明 Shape 类有 numberOfFaces 属性，Circle 这样的子类

由于其本身也是 Shape，因此会继承这个属性，这样它们都有了相同的属性和方法。方法和属性的实现可能不同(Circle 有一个棱，而 Cube 有 12 个棱)，但它们的有效性相同。这意味着我们可以编写处理 Shape 对象的代码，而这些代码也会自动用于处理 Circle、Rectangle 和 Cube 对象。

有用的术语

- 类：对象的类型。类描述了构成对象的方法和数据。
- 父类/子类：给定类在类层次结构中的父级和子级。
- 实例：程序运行过程中的对象。特定类的对象是有别于其他个体的独立个体。
- 成员：对象的方法或属性。“数据成员”是指与特定实例关联的数据。“成员函数”或“成员方法”是指基于实例进行操作的方法。
- 实例变量/实例方法：与“成员”同意，是个体对象的一部分数据或方法。
- 接口：用于外部使用的、仅对该对象生效的方法和属性列表。
- 实现：对于对象的函数成员和数据成员的定义。
- 覆盖：实现父类已有的方法，可以在类的实例中取代父类中相应的实现。

子类范式的另一个关键部分是：你的新类不仅能继承其超类的数据和方法，也能通过再次实现某个方法来替换它。在类似于 Java 或 C++的语言中，只有使用特定关键字(Java 或 C++语言使用的是 virtual 关键字)标注的方法能被子类覆盖。但在 Objective-C 中可以覆盖任何方法。

例如，Circle 类可以实现 Draw 方法，但它的子类 SkewedCirle 可以重新实现 Draw 方法以实现不同的操作。通过这种方法，子类可以覆盖其本身继承的行为。这是面向对象编程最强大的方面之一：通过继承已有类并且仅覆盖其小部分特定的行为，就能新建完整、复杂的类。

很多流行的面向对象语言都遵循 Simula 模式，它们会在编译期完成很多工作。结果，调用对象内的方法生成的代码仅适用于特定的实现。Shape 类调用的 Draw 方法与 Label 类调用的 Draw 方法并不通用，除非这个方法是通过同一个超类声明的。

相比之下，Objective-C 语言尽可能多地将处理过程从编译期和链接期移至运行时。它也在运行时动态推断对象的类，并且在运行时判定对象是否实现了某个给定的方法。这意味着方法(甚至包括实例变量)不需要在编译期和链接期进行明确的设置，而是能在运行时需要进行访问时再进行判定。无论何时进行处理，任何对象都能实现 Draw 方法，并且调用方法也与其他对象的 Draw 方法一样。

Objective-C 语言的这种核心动态机制被称为消息传递。你将在本章的剩余部分学习所有相关知识。

2.2　Objective-C 中的对象

同其他语言一样，在 Objective-C 中对象使用专门的操作访问数据。在 Objective-C 中把这些数据称为实例变量，其他语言环境下你可能已经听说过它们，简称 ivars 或成员变量，

但是所有这些术语都是指一个相同的概念。默认情况下，Objective-C 对象的实例变量对其他对象而言是不可用的，至于如何设定，我们会在讨论作用域时再介绍。

Objective-C 提供了一种单独的数据类型：id(与“did”押韵)。它指与它所属的类无关的任何对象类型，并且可以指代实例和类自身。它实际上作为指向基础 C 类型的指针实现，这种类型是所有 Objective-C 对象的基础。参见程序清单 2-1。

程序清单 2-1　Objective-C 中 id 的定义

```
typedef struct objc_class * Class;
struct objc_object {
    Class isa;
};
typedef struct objc_object * id;
```

Objective-C 中对象方法的默认返回类型(如果没有明确指定类型，那么会由编译器设定)总是 id，而对于普通的 C 语言结构而言默认返回 int 类型。另外，Objective-C 对象有明确的“零”值，定义为 nil。任何返回对象实例的方法都能返回 nil 作为零值或失败值。

消息传递与动态机制

Objective-C 中方法的概念是使用消息传递(message passing)系统实现的。这实际是指编译器不会生成直接在内存中调用对象方法的代码。它给对象传递消息，对象自己决定要调用的实际编译函数。本质上编译器只记录传来的消息的名称(在 Objective-C 中称其为选择器)，并且生成一个将该选择器传递给对象的函数调用。如果可以，该对象会响应它。消息传递的概念是 Smalltalk 语言的核心原则之一。有时 Smalltalk 和 Objective-C 都被称为面向消息的语言。今后，当你读到“消息”一词时，你通常可以认为这个术语跟“方法”是可以互换的。

值得注意的是，Objective-C 中的类也是对象的一种类型。编译器在编译期为每一个类明确地创建一个对象。按照约定，类名以大写字母开头，例如 MyObject；而实例名以小写字母开头，例如 myObject。类对象也有它们自己的类型——Class，以及属于它们的零值——Nil。注意这些名字也遵守前面提到的大小写约定。

因此，如果你生成了一个名为 String 的类的实例，就可以给它发送消息。也可以直接给 String 类发送消息，而不是给某个特定的实例。在 Objective-C 中，对类使用“工厂方法模式”返回该类初始化后的实例是很常见的。从运行时库的角度来看，向对象实例发送消息与向类发送消息是没有区别的。对于作为程序员的你而言，唯一的区别就是每一个类自身只存在一次，并且可以通过它的类名进行访问(在本例中是 String)，而不是通过输入某个变量进行访问。

Objective-C 程序通常符合一些特定的编程范式，以协助封装的数据和逻辑。所有这些都将在本书后面的章节中更详细地探讨，但现在只简单介绍一下你可能会遇到的，或者今后编写 Objective-C 代码时可能会用到的模式。

- 委托模式：委托模式是指一个逻辑单元(通常是一个对象)可以把一部分关于它行为的判定转移给另一个对象，这个对象被称为代理。委托的设计就是为了实现一些响应该判定的方法。
- 观察模式：这种模式允许任何有兴趣的对象从其他任何系统接收和响应状态更新。特别是 Objective-C 中的动态绑定允许这种情况以一种完全分离的方式发生：它允许处理离散事件，而不需要明确伪造观察者和事件源之间的连接。
- MVC(Model-View-Controller，模型-视图-控制器)模式：Objective-C 的 UI 框架(Mac 的 AppKit 和 iOS 的 UIKit)广泛使用这种模式。它分离了数据管理(模型)与数据呈现(视图)。这两者之间是控制器，它的作用是搭建两者之间沟通的桥梁。应用程序的逻辑就写在这里。这种模式有助于防止数据的呈现、代码和模块之间的相互依赖，允许对这些代码更大限度地重用。
- 代理模式：Objective-C 中消息和类型在运行时的动态处理使其允许使用代理对象作为中介。这些代理将消息向内转发至一个调用者也无法访问的、“真实的”对象。实现代理可以为现有对象添加额外的接口，如实现一种共享资源间的同步形式，或者为一个存在于其他系统中的对象提供引用(你会在本书后面学习 XPC 时看到这点)。例如，系统 API 可能为实现某个受限接口的内部对象返回一个代理。

如你所见，这些模式特别适合带有动态运行时的语言。在其他语言中常见的模式不经常出现在 Objective-C 中，如 C++标准模板库中经常使用的迭代器模式。例如，当语言中“任何对象”的概念生效时，以及当使用消息传递实现方法调用时，很少使用迭代器。对大多数迭代、排序和枚举操作而言，了解对象类型是非常必要的。

2.3　编写 Objective-C 代码

本节介绍 Objective-C 对象的使用和语法。这些内容分为几个部分，我们按下述顺序依次进行介绍：

- 对象的分配和初始化
- 给对象传递消息
- 内存管理的相关规则

然后，你将学习如何在以下方面定义自己的对象类：

- 声明类的接口
- 显式地定义实例数据
- 通过属性隐式地定义实例数据
- 声明类方法和实例方法
- 最后，实现以上所有内容

但在开始前需要知道一个简单而重要的知识点：Objective-C 中所有的对象都是指针类型。换句话说，你永远不要单独使用 String，而是应该使用 String *。所有的 Objective-C 对象都是在堆中分配内存的，而不是在栈中进行分配(也并非全部，这里也有你稍后会遇到的例外)。如果你声明一个静态分配的对象，而非一个指针，编译器将会为你提供有用的提

示，参见程序清单 2-2。

程序清单 2-2　Objective-C 对象都是指针类型

```
String aString;   // Semantic Issue: "Interface type cannot be statically allocated."
String *aString; // Correct.
```

2.3.1　内存分配和初始化

在 Mac 和 iOS 设备上运行的 Objective-C 中，所有对象[1]都是从 NSObject 类[2]向下继承。这个根类实现基本的分配和初始化方法，会被其他所有的 Objective-C 对象继承。这些方法中包含+alloc 和-init 方法，合并使用这两个方法可以创建给定类的新实例。

正值、负值，还是无关值？

你可能已经注意到上面提到的方法名，并且想要知道那些加号和减号起什么作用。它们跟“引用计数”(你可能暂时还不清楚这个概念，但你可能已经听说过很多关于由它带来的可怕后果的故事)有关吗？这些是编译器某种行为的暗示？或者可能只是返回值的标识？

答案(相对)很简单。定义类时，你不仅要能创建类的实例执行的方法，还要能定义类自身[3]执行的方法。实例方法由前置连字符标识，类方法由前置加号标识。如此，类似于-doSomething 这样的消息会被发送给对象实例，例如[nameString doSomething]；而+doSomething 会被发送给类，例如[String doSomething]。

第一个方法+alloc 被发送给 class 对象，使之在堆中分配该类的新实例。这在 Objective-C 中是类对象很常用的方法。但是分配内存只是全部处理的一部分。+alloc 方法会分配内存并且将其填充为 0，并且返回一个新分配的实例。之后这个实例需要使用-init 方法初始化。类会在这里设置它的实例变量、属性和相关状态。

-init 方法是最基本的初始化方法。当你创建自己的对象时包含它通常会很有用处，并且大多数(如果不是全部的话)系统对象会在调用时适当地初始化自身。但是多数类使用传参的方式实现更多特定的初始化方法。这些方法都以“init”开头以把它们标记成初始化方法，但也可以在“init”后面采取除此之外的其他形式，因此-initWithString:、-initWithOrigin:andSize:等都是有效的初始化方法。

+alloc 和-init 都返回 id 类型，并且通常连在一起调用。+alloc 返回后立即传递-init 消息，之后把调用的结果作为变量保存起来，否则不要使用+alloc 的结果。如果它不能执行这个函数，那么函数可能返回 nil。但对-init 而言，该方法也会释放+alloc 分配的内存。程序清单 2-3 中的代码介绍了如何使用 alloc/init 方法新建一个对象，并且可以让我们很好地了解消息传递的语法。

---

1 差不多是所有对象，稍后会遇到极个别的例外。

2 如果你正在使用不带主要框架的“纯”Objective-C，会有运行时头(runtime header)定义的等价 Object 类(但该类出现得更早一些，也没有真正维护)。

3 熟悉 C++或 Java 的读者应该知道这些方法是指静态成员方法。

程序清单 2-3　分配和初始化新的实例

```
MyObject * obj1 = [[MyObject alloc] init];
NamedObject * obj2 = [[NamedObject alloc] initWithName: "aName"];
if(obj2 == nil)
{
   // error allocating or initializing NamedObject instance
}
```

2.3.2　发送消息

如程序清单 2-3 所示，Objective-C 的消息发送操作被写在方括号内。消息的目标(接收者)放在左边，穿插着参数的消息体放在右边。这类似于 C++、Ruby 或 Java 这样的面向对象语言中使用的系统，只是由于方法名称中内置的参数位置的不同产生了一些细微的区别。图 2-2 展示了如何从一个例子映射到另一个例子，Objective-C 的消息语法写在上面，C 语言风格的方法调用写在下面。

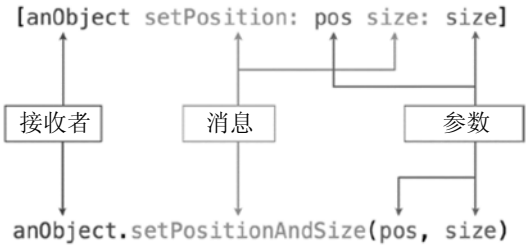

图 2-2　消息(上侧)与方法调用(下侧)

与 C 语言风格的函数名称不同，Objective-C 中的消息名称会被穿插于其中的参数所分隔，每个参数前面都有一个冒号字符。有一点很重要需要注意：Objective-C 消息名称中的每一个部分都是可选的，并且可以改变它们的顺序。一些语言实现了被称为“命名参数”或“关键字参数”的、看起来相似的概念，这些概念支持运行时重新排序或省略的特定参数，并且一些参数可能也已经预定义了默认值。这些特点都不适用于 Objective-C 消息。在需要使用到很长的参数列表的地方，可以简单地把参数在方法名称中的位置当作一种有助于程序员的句法。思考程序清单 2-4，这里会使用 Objective-C 和 C++调用简单的函数。

程序清单 2-4　方法调用对比

```
[myObject drawLineFromX: 100 Y: 120 toX: 140 Y: 120 weight: 2 color: redColor];
myObject.drawLine(100, 120, 140, 120, 2, redColor);
```

在这个 Objective-C 例子中，可以立即理解每一个参数的意义。但阅读 C++的例子时，如果不阅读或记录方法的参数文档，那么参数的意义就不是那么明了了。

但是 Objective-C 消息的这种不平常的语法并不妨碍你使用其他语言的任何东西。函数或其他消息的返回值可以直接内联放置，像参数和接收者一样。[[MyObject alloc] init]中的后者负责将内存排序，前者则可以嵌入任何类型的返回值，如程序清单 2-5 所示。注意缩进只是出于安排布局的目的。与 C 语言一样，Objective-C 会高效地将所有空白当作空格处理。

程序清单 2-5　深层嵌套的消息

```
[myView addSubview:
    [[LabelView alloc] initWithTitle:
        [[String alloc] initWithCString: "title"]]];
```

现在，你可能感觉你被困在某个非常像电影“盗梦空间”的地方，在重复调用嵌套的过程中翻滚。这实际上是相当贴切的比喻：电影中嵌套梦境的方式与前面展示的嵌套方法完全相同：在第一次调用中，你进入另一次调用，之后再次进入另一次调用，等等。然后，最后一次调用完成后，其返回值反馈到前者，前者继续向上反馈它的返回值，直到一切都获得解决。

在这样的情况下，Objective-C 的括号内语法实际上使事情变得更简单。每一个左方括号标识一个新的方法调用的开启。当右方括号出现，方括号中声明的结果会作为周围声明的一部分而出现。这在前面的例子中有以下步骤：

(1) 通过查看[myView addSubview:，看到了左方括号。下潜一个级别。

(2) 在另一个左方括号的后面又跟了一个环绕着[LabelView alloc]的左方括号。因此该方法会被求值。

(3) 返回值后面紧跟着 initWithTitle:，以及下方另一个级别的更多左方括号。

(4) 在此你又看到了相同的结构。对[String alloc]求值，并将返回值与 initWithCString:"title" 匹配。

(5) 现在找到了一个右方括号，对 initWithCString:call 求值，返回上一个级别。

(6) 现在找到了另一个右方括号，该返回值会发送给-initWithTitle:。

(7) 对于最后一个右方括号，前面几步的返回值被应用至-addSubview:，之后再次上调一个级别跳出整个表达式。

2.3.3　内存管理

我们已经了解了创建和使用对象的基础，让我们现在学习一下内存管理。像你之前看到的一样，所有的 Objective-C 对象都是在堆中申请的，因此如果你不想让内存消耗殆尽，你就需要在某些地方把这些对象释放掉。

早在遥远而黯淡的 20 世纪 80 年代至 20 世纪 90 年代初期，Objective-C 使用的是跟 C 语言非常相似的内存管理模块：如果你申请了内存，你还要记得释放它。而且你还需要小心你得到的对象，如果有方法是返回的对象，那么还需要使用 free 语句释放它 [4]。

这里提出了一直沿用至今的引用计数方法的思路：每一个对象记录了所有持有它的引用的次数。如果已经接收了一个对象并且你想要保留它，你需要对这个对象执行 retain 语句，从而增加它的引用计数。使用完成后，你需要对它执行 release 语句，从而减少它的引用计数。一旦计数器的数值变为 0，这意味着没有其他对象在引用这个对象，它就会自动调用 dealloc 方法以释放自己。

4 例如，请参阅 1991 年使用 Objective-C 编写的早期 Web 浏览器的源码，网址为 www.w3.org/History/1991-WWW-NeXT/Implementation/。

另外，对象可以设置为“在未来某个(期待的)时刻”使用 autorelease pool 释放。这种想法是指：系统会在某些时刻(通常是开始线程或等待响应输入事件时)创建 autorelease pool 并将其推送至堆栈中。之后对任何对象设置 autorelease 方法都可以把它标记在 autorelease pool 中。当 autorelease pool 被释放时，它会简单地给所有它标记的对象发送 release 方法。这样任何不再使用(例如没有明确调用过 retain 语句)的对象都会被释放。

不久前(具体地说，直到发布 OS X 10.7 Lion 和 iOS 5 之前)，这项工作是由 Objective-C 类(主要指 NSObject 和 NSAutoreleasePool)的 Foundation 框架负责执行的。但是现在，它已经成为这种语言本身运行时的一部分，执行它的代码也被合并至运行时代码。这使运行时和编译器可以对对象的生命周期做出更多指导性的判定，也使运行时可以大量优化对引用计数和 autorelease pool 的处理。

现在，你将看到三种 Objective-C 使用的不同类型的内存管理技术。

- 手动引用计数：这是指由程序员调用-retain、-release 和-autorelease 方法来管理单个对象的引用计数。这已经成为 iOS 5.0 之前以及历代 OS X 中使用的标准模式。
- 垃圾回收(Garbage Collection，GC)：伴随着 OS X 10.4 的到来，Objective-C 获得了一种与其他许多当代编程语言相似的、自动管理内存的垃圾回收器。这使引用计数方式变得不那么必要，它也能应用到系统中使用 C 语言代码做出的任何内存申请上。它也提供了一个非常有用的“弱引用归零”系统。通过它，指向对象的弱引用能一直保留，直至引用计数归零才失效。一旦对象被删除，所有指向它的弱引用也都会被置为 0。但是，垃圾回收机制需要大量的资源，它太消耗资源，从而不能在 iPhone 或 iPad 平台上部署。也因如此，在 OS X 上对垃圾回收机制的使用也被正式弃用，取而代之的是列表中的下一项。
- 自动引用计数(Automatic Reference Counting，ARC)：在 OS X 10.7 Lion 和 iOS 5.0 中介绍过(也能在认可的 OS X 10.6 和 iOS 4 中执行)的 ARC 技术把引用计数机制植入语言运行时，并且加强了 LLVM 编译器对它的支持(想了解 LLVM 和其他 OS X 上的编译器的更多信息，请参阅附文“编译”)。这个机制使编译器可以准确地判定对象在哪里保留、释放，以及自动释放，这意味着这项工作是由编译器完成的，而非程序员。ARC 也包含了通过 OS X 的垃圾回收器生效的弱引用归零系统。ARC 和 GC 之间的区别在于：GC 会让对象积聚起来并且在一个特定的间隔之后释放，ARC 只是简单地替你插入相应的 retain/release 调用。因此没有内存积聚，也没有高系统开销的收集阶段。事实上，编译器会严谨地优化整个 retain/release 循环。

在这本书中，我们将使用 ARC 守则，因为对那些习惯于自动内存管理语言的人而言，这种方式更加平易近人，而且他们更习惯使用这样的代码。另外，使用 ARC 可以允许我们使用弱引用归零机制，这使得大量的代码更稳定且无差错，尤其是当我们进入这本书后面的章节时。

但也不要以为你可以完全摆脱困难。你可能不再需要实际动手输入 retain 和 release，但它还在起作用，也仍然可能遇到问题，例如可怕的 retain 循环。在第 3 章，我们会看到这里究竟发生了什么，以及如何调试可能出现的问题。

编　　译

Objective-C 是一种编译语言，这意味着你的源代码被转换成了高度优化过的机器码。执行这项任务的软件是编译器。

从历史来看，Xcode 和它的前身——项目构造器使用的都是 GNU GCC 编译器(GNU Compiler Collection)。多年以来，苹果为这个编译器做了很多调整和贡献。但是在 2007 年，一种新型的、现代化的编译系统被创建出来，被称为 LLVM(Low Level Virtual Machine)。它最初是作为苹果公司图形库的一部分，因为它使源代码被编译成高效的字节码格式，这种格式可以非常迅速地运行之前为 CPU 或 GPU 编译的机器代码。苹果公司用这项技术使它们的图形可以运行在任何 GPU 或 CPU 之上，而不需要为每个目标做特定的手动调整。

LLVM 还有其他的优势，其中最主要的或许是：它的中段编译格式保留了很多关于源代码(这些源代码是它衍生的来源)的信息，远多于 GCC 和其他编译器。例如，这使它可以跟踪变量何时被使用和弃置，然后在它们的基础上做出判定。这个特点使 ARC 得以运行。从本质上讲，这种编译器能定位某个变量不再被使用的确切时机，并且确定它的内存特性，从而自动插入合适的内存管理方法。

基于 20 世纪 70 年代编译器设计基础的 LLVM 已经被证明是非常强大的，而且跟 GCC 相比更易于扩展。因此，苹果公司已经把它的编译工作移至全心全意支持 LLVM 项目上，甚至雇用该项目的领导者和主要贡献者做全职工作。基于已经奠定的基础，苹果公司正在把越来越多的编译器支持的特性植入 Objective-C 语言，如 ARC 以及 OS X 10.8 和 iOS 6.0 中引入的新对象的文字声明。

2.3.4　类接口

创建新对象类的首项任务是定义它的接口。这是使用@interface ... @end 块完成的。实例变量可以在紧跟@interface 关键字的大括号中进行定义(如程序清单 2-6 所示)，或者它们可以保持“不可见”并且作为类实现的一部分而不仅仅是声明，参见下面的例子。

程序清单 2-6　类接口

```
@interface MyObject
{
   int var1;
}

...

@end

...
```

然而，如果在应用程序中真正使用这个类，编译器会发出适量的警告。这与在类的层次结构中使用根类有关。Objective-C 中很多常见的方法都是 NSObject 类实现的，从历史上看(在 OS X 10.7 和 iOS 5.0 之前)，这包括整个引用计数的内存管理系统，现在它包含了很多类似比较、散列等默认功能。因此，最好永远不要创建没有超类的类，NSObject 是所

有情况下类层次结构中的根类。

指定超类的方法是把它放在类名后的冒号之后，如程序清单 2-7 所示。

程序清单 2-7　NSObject 的一个子类

```
@interface MyObject : NSObject
{
   int var1;
}

...

@end
```

面向对象语言中的实例变量通常保持着对除了对象自己定义的方法以外的其他任何调用的密封性，Objective-C 也不例外。Objective-C 提供了一些作用域来指定访问单个变量的级别。这些作用域都使用单独的关键字进行指定，任何关键字后面的变量都有相同的作用域。这些关键字与其他流行的面向对象编程语言中的关键字很相似。

- @public：后面的实例变量可以被所有对象访问。
- @protected：后面的变量只能由该类和该类的子类访问。如果不进行指定，这将是默认的作用域。
- @private：后面的变量只能由该类进行访问，子类不能继承访问。
- @package：特殊情况下，实例变量在相同的可执行映像(例如框架或插件)中是可以访问的，就像@public 作用域一样，但在这些映像之外会被认为是@private 作用域。这主要由苹果公司用于紧耦合框架类，这些框架的实例变量访问是共享的，但不应该被导出到框架外。

2.3.5　方法

Objective-C 中的方法声明跟你所知的其他语言区别较大。括号中首先出现的项只能出现类型(返回值类型和每个参数的类型)。参数本身也以一种与消息传递相似的方式穿插在方法名称中(参见图 2-3)。一个 Objective-C 方法声明的结构如图 2-3 所示。

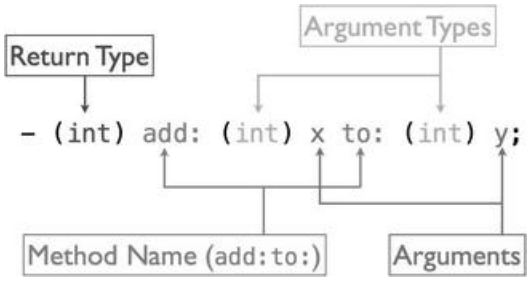

图 2-3　分解一个 Objective-C 方法声明

值得注意的是：每个参数前面的冒号也是方法名称的一部分。图 2-3 中编译器记录(运行时使用这个名称调用方法)的方法名称是 add:to:，而不是 addto。你也可能会注意到，与

提供函数重载的语言(如 C++)不同，这个方法名称不太在意返回值类型和参数类型。这意味着方法可以用代码中最明显最直接的方式调用，相比之下，C++编译器则生成繁琐的方法名称。

图 2-3 中没有显式调用的一项是处于方法开头的连字符。这与你之前学习内存分配和初始化时遇到的连字符相同。

定义方法时，在行首输入连字符表示这是一个实例方法，输入加号则表示这是一个类方法，否则会与消息传递过程中使用的布局相同。其他需要附加说明的是，返回值类型和参数类型需要括起来，分别写在方法和参数名称的前面，如程序清单 2-8 所示。

程序清单 2-8　一个同时有类方法和实例方法的类

```
@interface MyObject : NSObject
{
    int var1;
}
// a class factory method, equivalent to [[MyObject alloc] initWithValue: value]
+ (id)objectWithValue: (int)value;

// an initializer which takes an initial value
// this is the designated initializer
- (id)initWithValue: (int)value;

// you can also call -init, which will use a starting value of zero

@end
```

注意，不需要显式声明这个对象支持+alloc 或-init 方法，这些方法是从 NSObject 超类继承而来的。MyObject 可以重定义这些方法，但实际上 NSObject 已经声明了它们，所以不需要再重复这项工作。

有一个新术语隐藏在程序清单评论的最后一行中——你看到了吗？指定的初始化方法必须保证执行对象“真正的”初始化方法。换句话说，所有其他的初始化方法最终都会调用这个方法。与为了覆盖所有基本功能而重写超类的所有初始化方法相比，这使将另一个对象设置为子类的过程更简单(也可以说更安全)，你可以仅实现特定的初始化方法，并且心里清楚地知道现在其他对象调用的是重写的方法。

2.3.6　属性

除了显式定义数据成员和操作的方法，Objective-C 还提供了属性的概念。属性是指在对象外能够通过调用特定的 getter 和 setter 接口进行访问和修改的一块数据。在 Objective-C 中，这提供了读取或设置属性过程中复杂的相互作用的定义，所有的属性会被隐藏在非常简单的基于结构的语法中。这样做是为了区分使用主要用来调整数据的方法，通过这些方法能实现复杂的计算或功能。它也允许程序员卸下一些创建 getter 和 setter 方法的工作，编译器自动合成这些访问方法。编译器甚至能合成用于存储属性值的成员变量，使类接口更

简洁。

属性能在@interface 块中使用@property 关键字声明，与方法声明位置相同。简单来讲，属性声明看起来非常类似于变量声明，参见程序清单 2-9。

程序清单 2-9　一个简单的属性声明

```
@interface MyObject : NSObject

@property id anotherObject;

@end
```

也可以对 Objective-C 中的属性使用一些限定符，用以说明如何使用这些属性。它们被放置在@property 关键字后面的括号里，包含如下这些限定符：

- 访问(readonly、readwrite)：表示该属性是可读写的还是只读的。默认是 readwrite。对于单个属性只能指定为其中的一项。
- 线程安全(atomic、nonatomic)：通过指定 atomic 关键字(默认项)，这个属性的所有合成访问器将会为保证线程安全而被锁定和合成。nonatomic 关键字不能这样做而且也更常用在 iOS 设备上。大量的锁定操作会降低在 iOS 设备上运行时的性能。
- 存储(assign、retain、copy、strong、weak)：标准变量类型默认使用 assign 存储类型，而对象默认使用 retain 来增加值的引用计数(当值被改变或释放时也只是简单地调用 release 来释放它)。如果一个对象支持它，可以使用 copy 命令表明这个对象应该被批量复制，而不是简单地执行 retain(用于可变值)。strong 和 weak 是 ARC 新增的限定符。前者是对于对象的强引用(会被保留)，而后者是不会被保留的归零引用。如果值被释放，这个属性的值会自动置为 nil。对于单个属性，只能指定其中的一项。
- 方法(getter=、setter=)：这些操作允许对属性设置自定义消息选择器。默认情况下，一个名为 myProperty 的属性会有一个名为 myProperty 的 getter 方法和一个名为 SetMyProperty:的 setter 方法。这最常用在 Boolean 类型的属性上：hidden 属性可以使用 isHidden 和 setHidden:作为它的方法。

可以在程序清单 2-10 中看到更多常用的属性声明。

程序清单 2-10　更多常用的属性声明

```
@interface MyObject : NSObject

@property (nonatomic, assign, getter = isHidden) BOOL hidden;
@property (weak) id myParent;
@property (nonatomic, copy) String * title;

@end
```

因为属性声明附带了很多信息，所以编译器可以生成关于它们的所有内容，包括访问方法和用以支持这些属性的实例变量。因此，如果要使类的数据成员可供访问，常见的方法(也是符合习惯的做法)是只使用属性语法声明它们，然后让编译器处理实现细节。

2.3.7 协议

不少类的接口还有另外一个组成部分，那就是协议。Objective-C 中的协议在概念上与 Java 或.NET 的接口相似，但也存在一些区别。

尽管使用@protocol 关键字替换了@interface 关键字，声明协议仍然与声明一个新的类使用相似的方式。协议没有超类(或者超协议)，但可以声明它遵循另一个协议，实际上这是一回事儿。对协议和类的声明而言，声明遵循某个协议的语法是一致的：任何协议的名称都放置在类/协议名称和任何超类名称之前的尖括号中(参见程序清单 2-11)。

程序清单 2-11 使用协议

```
@protocol MyProtocol<AnotherProtocol>
...
@end

@interface MyObject : NSObject<MyProtocol>
...
@end
```

协议声明可以包含除了实例变量以外能包含类接口的任何东西。协议可以使用与声明类本身相同的语法来指定属性、类和实例方法。唯一的附加项是一对类似于用来为实例变量设定访问作用域的关键字：@optional 和@required。@required 关键字声明遵循这个协议的类必须执行@required 关键字后面声明的项。如果没有指定@required 和@optional，默认选项是@required。相反，@optional 关键字声明其后的项在正确遵守这个协议的过程中并不是必要的。可选的协议方法(亦称非正式协议)常用来定义委托的接口，委托对象选择只调用其中一至两个方法，不需要执行很多潜在的委托方法。使用这种方式编写协议，允许编译器在灵活实现某个对象的同时确保指向该对象的委托符合所必需的协议。

正式协议(一切都是必需的)用来声明和划定类可能实现的特定功能，例如包含对对象执行完整复制的能力，以及把它序列化为已知方式或相似活动的能力。如果对某个属性添加 copy 存储性质，复制工作将会使用 NSCoding 协议描述的方法执行。作为协议，任何对象都能选择实现它描述的方法，而不关心它的类层次结构。Objective-C 的消息传递机制的动态特性意味着可以使用与任何其他对象同样的方法调用一个对象的 copy 方法。协议的声明和执行提供了一种运行时和编译期的方法来决定给定对象(甚至可能不知道它的类)是否实现了一个已知的功能集。

2.3.8 实现

最后，我们来看看对象代码的实现。对象的类和实例方法的定义位于@implementation...@end 块中。

另外，也能使用与前面提到的@interface 块相似的方法在这个块中声明对象的实例变量。但请注意，所有的变量必须一起放置在相同的位置。不能将一些变量置于@interface 块中并且将另一些变量置于@implementation 中。程序清单 2-12 展示了这种行为方式。

程序清单 2-12　一个简单的对象实现

```
@implementation MyObject
{
    int aValue;
}

@synthesize value = aValue;

+(id)objectWithValue: (int)value
{
   return([[self alloc] initWithValue: value]);
}

-(id)initWithValue: (int)value
{
    self = [super init];
    if(self == nil)
        return(nil);

    aValue = value;

    return(self);
}

- (id)init
{
    // call through to the designated initializer
    return([self initWithValue: 0]);
}

@end
```

可以看到类方法的定义和两个初始化方法，还有一些新的关键字。

首先是@synthesize 声明，这是类接口中@property 声明的匹配项。这个关键字的后面是由逗号分隔的属性名称列表，或是属性名-变量名配对的列表。对于前一种情况，编译器将属性匹配成相同名称的实例变量。如果已经指定了某个实例变量，它会使用该实例变量，否则它在运行时创建一个实例变量。对于后一种情况，已经声明了实例变量，并且@synthesize 声明简单地告诉编译器使用这个变量存储属性的值。它们的类型必须完全相同，否则编译器会发出警告。程序清单 2-12 中设定了 value 属性将 aValue 实例变量用作存储变量。

多数程序员偏向于使用@synthesize myVar = _myVar 或相似的方法显式地为他们的属性设置名称。我们建议只有在会暴露内部计算变量时使用这种方法。当属性只是作为一些主要由外部使用的值的容器时，我们偏向于仅使用@synthesize myVar 让编译器来处理它。这提醒我们要记住，通过迫使自己使用属性访问器的语法来使用属性。

再往下是一个实现工厂函数的类方法。这里引入了另一个新的关键字 self。在 Objective-C 方法的执行过程中，self 用来指代接收消息的对象。它的用法与 C++或 C#中该关键字的功能相似，也同样类似于 Ruby 中名为 self 的关键字的用法。它的类型永远是指向正确的类

实例(或类对象)的指针——编译器掌握足够的信息提供这些内容——它也能像其他对象一样接收消息。由于这是一个类方法，因此这里的 self 指的是 MyObject 类自身。这意味着任何发送给它的消息也必须是一个类方法，这里调用的+alloc 就是这样一个方法。这个方法会返回一个新分配的实例，该实例之后会发送-initWithValue:消息以将其初始化，之后返回初始化的结果。

下一个方法实现对象的指定初始化方法。如前所述，通过这个初始化方法汇聚了所有其他的初始化例程。请注意，“指定的初始化方法”的概念是指由程序员而非编译器构建的某些方法。它是 API 的一部分，它明确通知子类，如果实现了这个初始化方法，那么无论超类在运行时调用的初始化方法到底是什么，都将保证它的执行。

初始化方法内是另一个新的关键字：super。像 self 一样，这个关键字只在 Objective-C 方法定义内生效，且是特殊的对象接收者。除了任何消息的处理都是由当前类的超类执行以外，它的调用方法与 self 的一样。因此，通过在这里使用[super init]，实际上调用了 NSObject 类中实现的-init 方法，而非这个类内部提供的方法。这种机制允许子类顺延其超类的实现行为，从而允许超类初始化实例变量和其他继承而来的状态，就像这个例子中的用法一样。

当前的 self 指什么？

你会发现 self 的值是超类初始化方法的结果。这是非常重要的一步：重要到如果你忽略它，编译器会弹出明显的警告。这背后的原因在于，对象的初始化可以做任何事情，甚至包括返回并非从-init 消息获取不同实例。这在一些单态(singleton)设计模式(本书后面会介绍)的实现中完成，但实际上是一个特别有用的 Objective-C 垃圾收集子系统的具体示例。

为了提升收集器的性能，一些对象的分配是分批进行的。例如，很多字符串经常被创建和释放。所以运行时会为一些字符串对象一次性分配足够的内存，并且当一个字符串可被收集时，它会被放入一个近期使用过的字符串池中。之后，当下次有字符串需要分配内存时，会耗费很小的系统开销，只是从池中取出一片现有的被弃用的内存，而非重新分配一片内存。字符串可能很复杂并难以驾驭。它们是可变的吗？它们使用什么编码呢？字符串可以小到它的字符能够内置在一个数组中吗？或者它能长到必须分配独立的内存空间来存储所有的字符吗？因此，字符串类的+alloc 方法可能返回了这样一片内存，它适合于存储一段使用内联存储的简短字符串。

这听起来很牵强，但它却是经常发生的一类事情。实际上，在 Objective-C 的 Foundation 框架中有很多不同的字符串类，出现在程序员面前的所有字符串类都是一个类：NSString。创建 NSString 对象可能会返回众多私有 NSString 子类中的一个子类的分配结果，其中任何子类可能执行不同的可选方式来存储其数据。总之，像数组一样，字符串的使用非常频繁，并且会从很多非常细致的优化中获益。

现在，既然-init 方法有可能返回 nil(你也许认为这种情况不太可能发生，但你永远无法确定)，就应该检查该方法的返回值。如果它返回 nil，就立刻这样做，因为任何失败(因为返回了 nil)的初始化方法必须释放自身。最近，这项任务由编译器强制执行，这是 ARC 带来的好处之一，所以实在没有理由不把它处理干净。调用[super init]返回 nil 之后尝试访

问任何成员变量都会导致无效的内存访问，你会访问到何处？不知道，反正不是正确有效的地方。

超类的初始化成功后，可以初始化自身的实例变量和/或属性，最后返回 self。

程序清单 2-12 中的最后一个方法展示了这种基本的、无参数的初始化方法如何通过指定的初始化方法进行调用，并且给它简单地传递一个默认值。这种模式确保了 MyObject 的任何子类都可以通过重写那个初始化方法拦截所有的初始化事件。

2.4　小结

本章是漫长而沉重的理论引导。你已经看到了面向对象程序的基本构建模块以及 Objective-C 语言如何提供和执行它们。现在你了解了类和实例、变量和属性，也能声明和执行协议。

学习完这样漫长的一章之后，可能你会这样问自己：“那么我现在能做什么了？”，答案是：你在本章中已经理解的信息将成为理解本书剩余部分的基石。没有了使用广泛、灵活且功能强大的框架，Objective-C 就变得一无所用。在第 3 章，将开始使用对象和类来完成你自己的应用程序。之后，你将学习 Objective-C 语言的特性，并探索一些该语言提供的、更加复杂且有用的工具。