O'REILLY

涵盖 iOS 9、Xcode 7和Swift 2.0

HZ BOOKS
华章IT

# iOS编程基础

## Swift、Xcode和Cocoa入门指南

iOS 9 Programming Fundamentals with Swift

Matt Neuburg 著
张龙 译

机械工业出版社
China Machine Press

# iOS 编程基础：Swift、Xcode 和 Cocoa 入门指南

*Matt Neuburg* 著

张龙 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc. 授权机械工业出版社出版

机械工业出版社

**图书在版编目（CIP）数据**

iOS 编程基础：Swift、Xcode 和 Cocoa 入门指南 /（美）马特·诺伊贝格（Matt Neuburg）著；张龙译 . —北京：机械工业出版社，2017.1
（O'Reilly 精品图书系列）
书名原文：iOS 9 Programming Fundamentals with Swift
ISBN 978-7-111-55635-0

I. i… II. ①马… ②张… III. 移动终端－应用程序－程序设计 IV. TN929.53

中国版本图书馆 CIP 数据核字（2016）第 317494 号

北京市版权局著作权合同登记
图字：01-2015-2794 号

©2016 Matt Neuburg. All rights reserved.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Machine Press, 2017. Authorized translation of the English edition, 2016 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2016。

简体中文版由机械工业出版社出版 2017。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

**封底无防伪标均为盗版**
**本书法律顾问**
**北京大成律师事务所 韩光 / 邹晓东**

书　　名 / iOS 编程基础：Swift、Xcode 和 Cocoa 入门指南
书　　号 / ISBN 978-7-111-55635-0
责任编辑 / 陈佳媛
封面设计 / Karen Montgomery，张健
出版发行 / 机械工业出版社
地　　址 / 北京市西城区百万庄大街 22 号（邮政编码 100037）
印　　刷 / 三河市宏图印务有限公司
开　　本 / 178 毫米 ×233 毫米　16 开本　34.5 印张
版　　次 / 2017 年 4 月第 1 版　2017 年 4 月第 1 次印刷
定　　价 / 129.00 元（册）

**凡购本书，如有缺页、倒页、脱页，由本社发行部调换**
**客服热线：(010)88379426；88361066**
**购书热线：(010)68326294；88379649；68995259**
**投稿热线：(010)88379604**
**读者信箱：hzit@hzbook.com**

# O'Reilly Media, Inc. 介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

## 业界评论

“O'Reilly Radar 博客有口皆碑。”

——Wired

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

# 译者序

在2014年的WWDC大会上，苹果公司正式发布了Swift这门全新的编程语言。作为iOS与OS X平台上的老牌编程语言Objective-C的有益补充和替代者，Swift从发布伊始就激发了广大开发者的强烈兴趣。学习和尝试Swift编程语言的开发人员越来越多，这也促使Swift这门新语言在TIOBE编程语言排行榜上的排名一路攀升，成为一颗耀眼的编程语言新星，同时也是有史以来增长速度最快的语言。虽然Swift的初始版本存在着不少问题，但苹果公司仍在不遗余力地持续推动着这门语言的发展。作为iOS与OS X的开发者，我们欣喜地看到Swift语言不断增强的功能、不断增加的特性以及不断优化的性能。这些都是Swift能够迅速得到广大开发者青睐的重要因素。

值得一提的是，一年后苹果公司在WWDC 2015上正式宣布将Swift开源，并于同年年底发布了全新的网站https://swift.org。目前Swift开源代码托管在GitHub上，任何感兴趣的开发者都可以下载学习。Swift如此之快的发展速度一方面得益于苹果公司各项产品的推出，另一方面也是由于广大开发者的热烈追捧。作为一门年轻的编程语言，能在短短两年时间内就获得如此成功，这也是我们广大iOS开发者的一个福音。技术发展日新月异，只有跟上技术发展的步伐我们才能在未来立于不败之地。目前，国内外已经有不少公司将自己的iOS应用部分或全部由Objective-C迁移至Swift，很多新项目也已经开始使用Objective-C进行开发了。这都进一步证实了Swift未来巨大的发展潜力。

本书可谓是Swift编程语言的一部百科全书。在学习本书之前不需要读者具备任何Swift背景知识（当然，适当了解Objective-C将会有助于学习，但也并非必需），读者只需要打开本书，从第1章开始逐章阅读即可。全书采用了由浅入深、循序渐进的方式对Swift语言进行讲解，同时辅以大量可运行的代码示例帮助读者加深对理论知识的理解。毕竟，

无论学习何种知识与技术，基础永远是最为重要的；坚实的基础将会帮助你更好地掌握技术，并且也会对后续的学习产生积极的作用。

全书共分 13 章，每一章都单独讲解一个主题，目的在于帮助读者集中精力掌握好 Swift 每一个重要且关键的知识点。从 Swift 架构概览开始，接着介绍了函数、变量、对象类型与流程控制，这些都是 Swift 重要的基础知识；然后又介绍了 Xcode 项目的管理、nib、文档以及项目的生命周期；全书最后对 Cocoa 类、Cocoa 事件、内存管理与对象间通信等高级主题展开了详尽的介绍。此外，附录 A 对 C、Objective-C 与 Swift 之间的关系和调用方式进行了详尽的论述。学习完本书后，读者将会掌握 Swift 重要且关键的特性与知识点，完全可以着手通过 Swift 开发全新的 iOS 应用。

Swift 编程语言涉及的知识点与特性非常多，没有任何一本书能够穷尽 Swift 的每一项特性，本书也不例外。本书可以作为读者学习 Swift 编程语言的入门指引，学习完本书后可以通过苹果公司的 Swift 编程语言官方文档等在线资源进一步加深对该门语言的理解和认识，并通过实际动手来掌握 Swift 的每一项特性。可以这么说，通过阅读本书，读者将会具备 Swift 开发的一般知识与技能，辅以一定的实践操作，相信经过一段时间的锤炼，你就可以真正精通这门优秀的编程语言。

技术图书的翻译是一项异常艰苦的劳动，这里我要将深深的感激之情送给我的家人，感谢你们在生活中对我无微不至的关怀，使我能够专心于翻译工作；此外，我要将这本书送给我亲爱的孩子张梓轩小朋友，每当爸爸感到疲惫时，看到你就会立刻获得无尽的动力，你永远是爸爸的开心果，如果你未来有志成为一名程序员，爸爸愿意祝你一臂之力；最后，非常感谢机械工业出版社华章公司的缪杰老师，感谢你对我持续的帮助，每一次与你沟通都非常顺畅，虽未曾谋面，但已然是老友。

虽然译者已经在本书的翻译工作上倾注了大量的心力，不过囿于技术与英文水平，书中难免出现一些瑕疵。如果在阅读过程中发现了问题，请不吝赐教并发邮件至 zhanglong217@163.com，我会逐一检查每一项纰漏，以期重印时修订。

张龙

2016 年于北京

# 目录

前言 ..... 1

## 第一部分 语言

### 第1章 Swift架构纵览 ..... 11

1.1 基础 ..... 11
1.2 万物皆对象 ..... 12
1.3 对象类型的3种风格 ..... 14
1.4 变量 ..... 14
1.5 函数 ..... 15
1.6 Swift文件的结构 ..... 16
1.7 作用域与生命周期 ..... 19
1.8 对象成员 ..... 20
1.9 命名空间 ..... 20
1.10 模块 ..... 21
1.11 实例 ..... 22
1.12 为何使用实例 ..... 24
1.13 self ..... 26
1.14 隐私 ..... 27
1.15 设计 ..... 29
1.15.1 对象类型与API ..... 29
1.15.2 实例创建、作用域与生命周期 ..... 31
1.15.3 小结 ..... 32

## 第2章 函数 ..... 34

2.1 函数参数与返回值 ..... 34

2.1.1 Void返回类型与参数 ..... 37

2.1.2 函数签名 ..... 38

2.2 外部参数名 ..... 38

2.3 重载 ..... 41

2.4 默认参数值 ..... 42

2.5 可变参数 ..... 43

2.6 可忽略参数 ..... 44

2.7 可修改参数 ..... 44

2.8 函数中的函数 ..... 48

2.9 递归 ..... 49

2.10 将函数作为值 ..... 50

2.11 匿名函数 ..... 53

2.12 定义与调用 ..... 57

2.13 闭包 ..... 58

2.13.1 闭包是如何改善代码的 ..... 59

2.13.2 返回函数的函数 ..... 61

2.13.3 使用闭包设置捕获变量 ..... 64

2.13.4 使用闭包保存捕获的环境 ..... 64

2.14 柯里化函数 ..... 66

## 第3章 变量与简单类型 ..... 68

3.1 变量作用域与生命周期 ..... 68

3.2 变量声明 ..... 69

3.3 计算初始化器 ..... 72

3.4 计算变量 ..... 73

3.5 setter观察者 ..... 76

3.6 延迟初始化 ..... 77

3.7 内建简单类型 ..... 79

3.7.1 Bool ..... 80

3.7.2 数字 ..... 81

3.7.3 String ..... 88
3.7.4 Character ..... 93
3.7.5 Range ..... 96
3.7.6 元组 ..... 98
3.7.7 Optional ..... 102

第4章 对象类型 ..... 114

4.1 对象类型声明与特性 ..... 114
4.1.1 初始化器 ..... 115
4.1.2 属性 ..... 122
4.1.3 方法 ..... 125
4.1.4 下标 ..... 127
4.1.5 嵌套对象类型 ..... 128
4.1.6 实例引用 ..... 129
4.2 枚举 ..... 131
4.2.1 带有固定值的Case ..... 132
4.2.2 带有类型值的Case ..... 133
4.2.3 枚举初始化器 ..... 134
4.2.4 枚举属性 ..... 136
4.2.5 枚举方法 ..... 137
4.2.6 为何使用枚举 ..... 138
4.3 结构体 ..... 139
4.3.1 结构体初始化器、属性与方法 ..... 139
4.3.2 将结构体作为命名空间 ..... 140
4.4 类 ..... 141
4.4.1 值类型与引用类型 ..... 142
4.4.2 子类与父类 ..... 146
4.4.3 类初始化器 ..... 151
4.4.4 类析构器 ..... 159
4.4.5 类属性与方法 ..... 159
4.5 多态 ..... 161
4.6 类型转换 ..... 164

4.7 类型引用 ..... 168
4.8 协议 ..... 172
4.8.1 为何使用协议 ..... 174
4.8.2 协议类型测试与转换 ..... 176
4.8.3 声明协议 ..... 177
4.8.4 可选协议成员 ..... 178
4.8.5 类协议 ..... 180
4.8.6 隐式必备初始化器 ..... 180
4.8.7 字面值转换 ..... 182
4.9 泛型 ..... 183
4.9.1 泛型声明 ..... 185
4.9.2 类型约束 ..... 186
4.9.3 显式特化 ..... 189
4.9.4 关联类型链 ..... 190
4.9.5 附加约束 ..... 192
4.10 扩展 ..... 196
4.10.1 扩展对象类型 ..... 196
4.10.2 扩展协议 ..... 199
4.10.3 扩展泛型 ..... 202
4.11 保护类型 ..... 203
4.11.1 AnyObject ..... 203
4.11.2 AnyClass ..... 206
4.11.3 Any ..... 207
4.12 集合类型 ..... 208
4.12.1 Array ..... 208
4.12.2 Dictionary ..... 222
4.12.3 Set ..... 227

**第5章 流程控制与其他 ..... 232**

5.1 流程控制 ..... 232
5.1.1 分支 ..... 233
5.1.2 循环 ..... 245

5.1.3 跳转 ..... 250
5.2 运算符 ..... 261
5.3 隐私性 ..... 264
5.3.1 Private声明 ..... 265
5.3.2 Public声明 ..... 267
5.3.3 隐私性规则 ..... 267
5.4 内省 ..... 267
5.5 内存管理 ..... 269
5.5.1 弱引用 ..... 270
5.5.2 无主引用 ..... 272
5.5.3 匿名函数中的弱引用与无主引用 ..... 273
5.5.4 协议类型引用的内存管理 ..... 275

# 第二部分 IDE

## 第6章 Xcode项目剖析 ..... 279

6.1 新建项目 ..... 279
6.2 项目窗口 ..... 281
6.2.1 导航窗格 ..... 282
6.2.2 辅助窗格 ..... 288
6.2.3 编辑器 ..... 289
6.3 项目文件及其依赖 ..... 291
6.4 目标 ..... 293
6.4.1 构建阶段 ..... 294
6.4.2 构建设置 ..... 296
6.4.3 配置 ..... 297
6.4.4 方案与目标 ..... 298
6.5 从项目到运行应用 ..... 300
6.5.1 构建设置 ..... 303
6.5.2 属性列表设置 ..... 303
6.5.3 nib文件 ..... 304
6.5.4 其他资源 ..... 305

6.5.5 代码文件与应用启动过程 ..... 307
6.5.6 框架与SDK ..... 312
6.6 对项目内容进行重命名 ..... 314

第7章 nib管理 ..... 316
7.1 nib编辑器界面概览 ..... 317
7.1.1 文档大纲 ..... 318
7.1.2 画布 ..... 321
7.1.3 查看器与库 ..... 323
7.2 nib加载 ..... 324
7.2.1 何时加载nib ..... 325
7.2.2 手工加载nib ..... 326
7.3 连接 ..... 328
7.3.1 插座变量 ..... 328
7.3.2 nib拥有者 ..... 330
7.3.3 自动配置nib ..... 333
7.3.4 误配置的插座变量 ..... 333
7.3.5 删除插座变量 ..... 335
7.3.6 创建插座变量的其他方式 ..... 335
7.3.7 插座变量集合 ..... 338
7.3.8 动作连接 ..... 339
7.3.9 创建动作的其他方式 ..... 340
7.3.10 误配置的动作 ..... 342
7.3.11 nib之间的连接——不行！ ..... 342
7.4 nib实例的其他配置 ..... 343

第8章 文档 ..... 346
8.1 文档窗口 ..... 346
8.2 类文档页面 ..... 349
8.3 示例代码 ..... 351
8.4 快速帮助 ..... 352
8.5 符号 ..... 353

8.6 头文件 ..... 354
8.7 互联网资源 ..... 355

第9章 项目的生命周期 ..... 356

9.1 设备架构与条件代码 ..... 356
9.1.1 向后兼容 ..... 357
9.1.2 设备类型 ..... 359
9.2 版本控制 ..... 360
9.3 编辑与代码导航 ..... 362
9.3.1 自动补令 ..... 363
9.3.2 代码片段 ..... 364
9.3.3 Fix-it与实时语法检查 ..... 365
9.3.4 导航 ..... 366
9.3.5 查找 ..... 368
9.4 在模拟器中运行 ..... 369
9.5 调试 ..... 370
9.5.1 原始调试 ..... 370
9.5.2 Xcode调试器 ..... 372
9.6 测试 ..... 378
9.7 清理 ..... 383
9.8 在设备中运行 ..... 384
9.8.1 在没有开发者计划成员资格的情况下运行 ..... 386
9.8.2 获取开发者计划成员资格 ..... 387
9.8.3 获取证书 ..... 387
9.8.4 获取开发配置文件 ..... 389
9.8.5 运行应用 ..... 390
9.8.6 配置文件与设备管理 ..... 390
9.9 分析 ..... 391
9.9.1 仪表盘 ..... 391
9.9.2 Instruments ..... 392
9.10 本地化 ..... 394
9.10.1 本地化Info.plist ..... 396

9.10.2 本地化nib文件 ..... 398
9.10.3 本地化代码字符串 ..... 399
9.10.4 使用XML文件进行本地化 ..... 401
9.11 归档与发布 ..... 403
9.12 Ad Hoc发布 ..... 405
9.13 最后的准备 ..... 406
9.13.1 应用图标 ..... 407
9.13.2 其他图标 ..... 408
9.13.3 启动图片 ..... 408
9.13.4 屏幕截图与视频预览 ..... 409
9.13.5 属性列表设置 ..... 410
9.14 向App Store提交应用 ..... 411

# 第三部分 Cocoa

## 第10章 Cocoa类 ..... 417

10.1 子类化 ..... 417
10.2 类别与扩展 ..... 420
10.2.1 Swift如何使用扩展 ..... 420
10.2.2 你应该如何使用扩展 ..... 421
10.2.3 Cocoa如何使用类别 ..... 421
10.3 协议 ..... 423
10.3.1 非正式协议 ..... 425
10.3.2 可选方法 ..... 425
10.4 Foundation类精讲 ..... 427
10.4.1 常用的结构体与常量 ..... 427
10.4.2 NSString及相关类 ..... 428
10.4.3 NSDate及相关类 ..... 431
10.4.4 NSNumber ..... 433
10.4.5 NSValue ..... 434
10.4.6 NSData ..... 435
10.4.7 相等与比较 ..... 435

10.4.8 NSIndexSet ..... 437
10.4.9 NSArray与NSMutableArray ..... 438
10.4.10 NSDictionary与NSMutableDictionary ..... 440
10.4.11 NSSet及相关类 ..... 440
10.4.12 NSNull ..... 442
10.4.13 不变与可变 ..... 442
10.4.14 属性列表 ..... 443
10.5 访问器、属性与键值编码 ..... 443
10.5.1 Swift访问器 ..... 445
10.5.2 键值编码 ..... 446
10.5.3 键值编码的使用 ..... 447
10.5.4 KVC与插座变量 ..... 448
10.5.5 键路径 ..... 449
10.5.6 数组访问器 ..... 450
10.6 NSObject揭秘 ..... 450

**第11章 Cocoa事件 ..... 453**

11.1 为何使用事件 ..... 453
11.2 子类化 ..... 454
11.3 通知 ..... 455
11.3.1 接收通知 ..... 456
11.3.2 取消注册 ..... 458
11.3.3 发布通知 ..... 459
11.3.4 NSTimer ..... 460
11.4 委托 ..... 461
11.4.1 Cocoa委托 ..... 461
11.4.2 实现委托 ..... 463
11.5 数据源 ..... 465
11.6 动作 ..... 465
11.7 响应器链 ..... 468
11.7.1 推迟职责 ..... 469
11.7.2 Nil-Targeted动作 ..... 470

11.8 键值观测 ..... 471
11.9 事件泥潭 ..... 475
11.10 延迟执行 ..... 477

第12章 内存管理 ..... 480
12.1 Cocoa内存管理的原理 ..... 480
12.2 Cocoa内存管理的原则 ..... 481
12.3 ARC及其作用 ..... 482
12.4 Cocoa对象管理内存的方式 ..... 483
12.5 自动释放池 ..... 484
12.6 实例属性的内存管理 ..... 486
12.7 保持循环与弱引用 ..... 487
12.8 值得注意的内存管理情况 ..... 488
12.9 nib加载与内存管理 ..... 492
12.10 CFTypeRefs的内存管理 ..... 493
12.11 属性的内存管理策略 ..... 495
12.12 调试内存管理的错误 ..... 497

第13章 对象间通信 ..... 499
13.1 实例化可见性 ..... 500
13.2 关系可见性 ..... 502
13.3 全局可见性 ..... 503
13.4 通知与KVO ..... 504
13.5 模型－视图－控制器 ..... 505

附录A C、Objective-C与Swift ..... 507

# 前言

2014 年 6 月 2 日，苹果公司在 WWDC 大会最后宣布了一项令人震惊的公告：“我们开发了一门全新的编程语言。”开发者社区对此感到非常惊讶，因为他们已经习惯了 Objective-C，因此开始怀疑苹果公司是否有能力将既有资产迁移过来。不过，这一次开发者社区错了。

Swift 发布后，众多开发者立刻开始检视这门新语言：学习并批判它，决定是否该使用它。我的第一步就是将自己所有的 iOS 应用都转换为 Swift；这足以说服我自己，虽然 Swift 有各种各样的缺点，但它值得每一个 iOS 编程新兵去掌握；自此以后，我的书都会假设读者使用的是 Swift。

Swift 语言从一开始的设计上就具备如下主要特性：

*面向对象*

Swift 是一门现代化的、面向对象的语言。它是完全面向对象的：“一切皆对象。”

*清晰*

Swift 易于阅读和编写，其语法糖很少，隐藏的捷径也不多。其语法清晰、一致且明确。

*安全*

Swift 使用强类型，从而确保它知道（并且你也知道）在每一时刻每个对象引用都是什么类型的。

*小巧*

Swift 是一门小巧的语言，提供了一些基本的类型与功能，除此之外别无其他。其他功能需要由你的代码，或你所使用的代码库（如 Cocoa）来提供。

*内存管理*

Swift 会自动管理内存。你很少需要考虑内存管理问题。

*兼容于 Cocoa*

Cocoa API 是由 C 和 Objective-C 编写的。Swift 在设计时就明确保证可与大多数 Cocoa API 交互。

这些特性使得 Swift 成为学习 iOS 编程的一门优秀语言。

其他选择 Objective-C 依然存在，如果你喜欢还可以使用它。实际上，编写一个同时包含 Swift 代码与 Objective-C 代码的应用是很容易的；有时也需要这么做。不过，Objective-C 缺少 Swift 的一些优势。Objective-C 在 C 之上增加了面向对象特性。因此，它只是部分面向对象的；它同时拥有对象与标量数据类型，其对象需要对应于一种特殊的 C 数据类型（指针）。其语法掌握起来很困难；阅读与编写嵌套的方法调用会让人眼花，它还引入了一些黑科技，如隐式的 nil 测试。其类型检查可以而且经常关闭，这会导致程序员犯错，将消息发送给错误的对象类型并导致程序崩溃。Objective-C 使用了手工的内存管理；新引入的 ARC（自动引用计数）减轻了程序员的一些负担，并且极大地降低了程序员犯错的可能性，不过错误依旧有可能发生，内存管理最终还是要靠手工来完成。

最近向 Objective-C 增加或修订的特性（ARC、合成与自动合成、改进的字面值数组与字典的语法、块等）让 Objective-C 变得更加简单和便捷，不过这些修复也使语言变得更加庞大，甚至会引起困惑。由于 Objective-C 必须要包含 C，因此其可扩展和修订的程度会受到限制。另一方面，Swift 则是个全新的开始。如果你梦想完全修订 Objective-C，从而创建一个更棒的 Objective-C，那么 Swift 可能就是你所需要的。它将一个先进、合理的前端置于你与 Cocoa Objective-C API 之间。

因此，Swift 就是本书通篇所使用的编程语言。不过，读者还需要对 Objective-C（包括 C）有所了解。Foundation 与 Cocoa API（这些内建的命令是你的代码一定会用到的，从而让 iOS 设备上的一切可以实现）依旧使用 C 与 Objective-C 编写。为了与它们进行交互，你需要知道这些语言需要什么。比如，为了在需要 NSArray 时可以传递一个 Swift 数组，你需要知道到底是什么对象可以作为 Objective-C NSArray 的元素。

因此，本书虽然不会讲解 Objective-C，但我会对其进行足够充分的介绍，从而使你在文档和互联网上遇到这类问题时能够知道解决方案，我还会时不时地展示一些 Objective-C 代码。本书第三部分关于 Cocoa 的介绍会帮助大家以Objective-C 的方式来思考——因为 Cocoa API 的结构与行为基本上是基于 Objective-C 的。本书最后的附录会详细介绍 Swift 与 Objective-C 之间的交互方式，同时还会介绍如何以 Swift 和 Objective-C 混合编程来编写应用。

# 本书范围

本书实际上是我的另一本书《Programming iOS 9》的配套参考书，该书以本书的结束作为起点。它们之间是互补的。我相信，这两本书的结构合理、内容通俗易懂。它们提供了开始编写 iOS 应用所需的完整基础知识；这样，在开始编写 iOS 应用时，你会对将要做的事情以及方向有着深刻的理解。如果编写 iOS 程序类似于用砖盖房子，那么本书将会介绍什么是砖以及如何使用它，而《Programming iOS 9》则会给你一些实际的砖并告诉你如何将其堆砌起来。

阅读完本书后，你将知道 Swift、Xcode 以及 Cocoa 框架的基础，接下来就可以直接开始阅读《Programming iOS 9》了。相反，《Programming iOS 9》假设你已经掌握了本书所介绍的内容；一开始它就会介绍视图与视图控制器，同时假设你已经掌握了语言本身和 Xcode IDE。如果开始阅读《Programming iOS 9》并且想知道书中一些没有讲解过的东西，如 Swift 语言基础、`UIApplicationMain` 函数、nib 加载机制、Cocoa 的委托与通知模式、保持循环等，那就不要尝试在该书中寻找答案了，我并没有在那本书中介绍这些内容，因为这里都介绍过了。

本书的 3 部分内容将会介绍 iOS 编程的基础知识：

- 第一部分从头开始介绍 Swift 语言。我没有假设你知道任何其他的编程语言。我讲解 Swift 的方式与其他人不同，如苹果公司的方式；我会采取系统的方式，逐步推进，不断深入。同时，我会讲解最本质的内容。Swift 并不是一门庞大的语言，不过却有一些微妙之处。你无须深入到全部内容当中，我也不会面面俱到地讲解。你可能永远都不会遇到一些问题，即便遇到了，那也说明你已经进入到了高级 Swift 的世界当中，而这已经超出了本书的讨论范围。比如，读者可能会惊奇地发现我在书中从来都没有提到过 Swift playground 和 REPL。本书的关注点在于实际的 iOS 编程，因此我对 Swift 的介绍将会关注在这些常见、实际的方面上；以我的经验来看，这些内容才是 iOS 编程当中用得最多的。
- 第二部分将会介绍 Xcode，这是我们进行 iOS 编程的地方。我将介绍何为 Xcode 项目，如何将其转换为应用，如何通过 Xcode 来查阅文档，如何编写、导航与调试代码，以及如何在设备上运行应用并提交到 App Store 等过程。该部分还有重要的一章用来介绍 nib 与 nib 编辑器（Interface Builder），包括插座变量与动作，以及 nib 加载机制；不过，诸如 nib 中的自动布局限制等专门的主题则不在本书的讨论范围之中。
- 第三部分将会介绍 Cocoa Touch 框架。在进行 iOS 编程时，你会使用到苹果公司提供的大量框架。这些框架共同构成了 Cocoa；为 iOS 编程提供 API 的 Cocoa 叫作 Cocoa Touch。你的代码最终将是关于如何与 Cocoa 进行通信的。Cocoa Touch 框架

提供了 iOS 应用所需的底层功能。不过要想使用框架，你需要按照框架的想法去做，将代码放到框架期望的位置处，实现框架要求你实现的功能。有趣的是，Cocoa 使用的是 Objective-C，你使用的是 Swift：你需要知道 Swift 代码如何与 Cocoa 的特性与行为进行交互。Cocoa 提供了重要的基础类，并添加了一些语言与架构上的设施，如类别、协议、委托、通知，以及关于内存管理的基本功能。该部分还会介绍键值编码与键值观测。

本书读者将会掌握任何优秀的 iOS 开发者所需的基础知识与技术；但本书并不会介绍如何编写一个有趣的 iOS 应用，书中会大量使用我自己编写的应用与实际的编程场景来阐述理论知识。接下来各位读者就可以阅读《Programming iOS 9》了。

# 版本

本书使用的是 Swift 2.0、iOS 9 与 Xcode 7。

总的来说，本书并不会对老版本的 iOS 与 Xcode 做过多介绍。我也不会有意在书中对老版本的软件进行讲解，毕竟这些内容在我之前的书中都有过介绍。不过，本书会针对向后兼容性给出一些建议（特别是在第 9 章）。

Xcode 7 所包含的 Swift 语言（Swift 2.0）相比于之前的版本 Swift 1.2 发生了很大的变化。如果之前使用过 Swift 1.2，那么你就会发现如果不做一些修改，代码是无法在 Swift 2.0 中编译通过的。与之类似，书中的代码使用 Swift 2.0 编写而成，它也完全无法与 Swift 1.2 保持兼容。你之前可能有过 Swift 1.2 的编程经验，随着我的不断讲解，你会发现不少重要的语言特性在 Swift 2.0 中都发生了变化。不过，我并不会介绍 Swift 1.2；如果想要了解（虽然我不知道你为什么要了解），那么请参考本书的前一版。

# 致谢

首先感谢 O'Reilly Media 的工作人员，正是他们才让一本书的写作过程充满了快乐：Rachel Roumeliotis、Sarah Schneider、Kristen Brown、Dan Fauxsmith 与 Adam Witwer。我也不会忘记编辑 Brian Jepson，虽然他并未参与本版的工作，但对我的影响却一直都在。

一直以来，一些优秀的软件对我起到了巨大的帮助作用，我在写作本书的过程中一直都心存谢意。这些软件主要有：

- git（*http://git-scm.com*）
- SourceTree（*http://www.sourcetreeapp.com*）
- TextMate（*http://macromates.com*）

- AsciiDoc（*http://www.methods.co.nz/asciidoc*）
- BBEdit（*http://barebones.com/products/bbedit/*）
- Snapz Pro X（*http://www.ambrosiasw.com*）
- GraphicConverter（*http://www.lemkesoft.com*）
- OmniGraffle（*http://www.omnigroup.com*）

我通过忠实的 Unicomp Model M 键盘（*http://pckeyboard.com*）完成了全书的输入与编辑工作，如果没有它我是不可能在如此长的时间内轻松敲下这么多字的。请通过 *http://matt.neuburg.usesthis.com* 了解我的工作环境。

# 《Programming iOS 4》前言

编程框架体现了一个人的品格，它是创建者对于目标与心智的反映。第一次使用 Cocoa Touch 时，我对其品格的评价是这样的："喔，创建它的人真是绝顶聪明啊！"一方面，内建的界面对象数量有意得到了限制；另一方面，一些对象的功能与灵活性（特别是 UITableView 等）要比其 OS X 的对应者更加强大。更为重要的是，苹果公司提供了一种聪明的方式（UIViewController）来帮助开发者创建整个界面并使用一个界面替换掉另一个，这些以一种可控、层次化的方式来实现，这样小小的 iPhone 就可以在一个应用中显示多个界面了，还不会让用户迷失或感到困惑。

iPhone 的流行（大量免费与便宜的应用起到了很大的帮助作用）以及随后 iPad 的流行让很多新的开发者看到为这些设备编写程序是值得的，虽然他们对 OS X 可能没有相同的感觉。苹果公司自己的年度 WWDC 开发者大会也反映出了这种趋势，其重心也由 OS X 逐渐向 iOS 倾斜。

不过，人们渴望编写 iOS 程序的想法也导致了这样一种趋势：还没有学会走就开始跑了。iOS 赋予了开发者强大的能力，心有多大舞台就有多大，不过这也是需要基础的。我常常看到一些 iOS 开发者提出的问题，虽然他们在编写着应用，但其对基础知识的理解非常肤浅。

这种情况促使我写作了这本书，本书旨在介绍 iOS 的基础知识。我喜欢 Cocoa，也一直希望能有机会写一本关于 Cocoa 的图书，不过 iOS 的流行却让我编写了一本关于 iOS 的图书。我尝试采取一种合乎逻辑的方式进行说明和讲解，介绍 iOS 编程所需的原则与元素。正如之前的图书一样，我希望你能完整阅读这本书（学习新东西肯定会不停翻书），并将其作为案头参考。

本书并不是要代替苹果公司自己的文档与示例项目。那些都是非常棒的资源，随着时间

的流逝将会变得越来越好。在准备本书写作的过程中我也将其作为参考资源。但是，我发现它们并不是按照顺序以一种合理的方式来完成一个功能的。在线文档会假设你的预备知识；它不能确保你会按照给定的方式来完成。此外，在线文档更适合作为参考而不是指南。完整的示例（无论注释有多么充分）都是难以跟着学习的；它可以作为演示，但却无法作为教学资源。

另外，图书的章节号和页码连续，内容的连贯性比较强；我可以在你学习视图控制器之前假定你已经知道视图了，因为第一部分位于第二部分之前。此外，我还会将自己的经验逐步分享给你。在全书中，你会发现我不断提及“常见的初学者错误”；除了一些其他人的错误，在大多数情况下，这些都是我曾经犯过的错误。我会告诉你一些陷阱，因为这些都是我曾经遇到过的，我相信你也一定会遇到。你还会看到我给出了不少示例，目的是解释一个大应用的一小部分内容。这并非用于讲解编程的一个已经完成的大程序，而是开发这个程序时的思考过程。我希望你在阅读本书时能够掌握这种思考过程。

# 本书约定

本书中使用以下排版约定：

*斜体（Italic）*

表示新术语、URL、电子邮件地址、文件名和文件扩展名。

等宽字体（Constant width）

表示代码示例，以及插穿在文中的代码，包括：变量或函数名、数据库、数据类型、环境变量、语句，以及关键字。

**等宽粗体（Constant width bold）**

表示新术语、URL、电子邮件地址、文件名和文件扩展名。

*等宽斜体（Constant width italic）*

表示新术语、URL、电子邮件地址、文件名和文件扩展名。

这个元素表示提示或建议。

这个元素表示一般注解。

这个元素表示警告。

# 如何使用示例代码

本书在这里帮助你完成你的工作。总的来讲，你可以在你的程序和文档中使用本书中的代码。你不需要联系我们以征得许可，除非你正在复制代码中的重要部分。比如，使用书中的多段代码写一个程序并不需要获得许可。

若将O'Reilly公司出版的书中的例子制成光盘来销售或发行则需要获得许可。在回答问题时，引用本书和列举书中的例子代码并不需要许可。把本书中的代码作为你产品文档的重要部分时需要获得许可。

我们希望但并不要求你在引用本书内容时说明引文的文献出处。引用通常包括题目、作者、出版社和ISBN号。例如，《iOS 9 Programming Fundamentals with Swift》，Matt Neuburg (O'Reilly)。Copyright 2016 Matt Neuburg，978-1-491-93677-1。”

如果你感觉你对代码示例的使用超出合理使用以及上述许可范围，请通过*permissions@oreilly.com*联系我们。

# Safari® 图书在线

Safari图书在线（*www.safaribooksonline.com*）是一个按需数字图书馆，它采用图书和视频两种形式发布专业级的内容，作者都是来自技术和商业领域的世界顶尖专家。

技术专家、软件开发者、网站设计者和商业及创新专家都使用Safari图书在线作为他们研究、解决问题、学习和职业资格培训的首要资源。

Safari 图书在线为各种组织、政府机构和个人提供丰富的产品和定价程序。订购者可在一个全文可检索数据库中浏览数以千计的图书、培训视频和预出版手稿。它们来自O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等的众多出版社。关于 Safari 图书在线的更多信息，请在线访问我们。

# 如何联系我们

美国：

O'Reilly Media，Inc.
1005 Gravenstein Highway North
Sebastopol，CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

我们为本书提供了网页，该网页上面列出了勘误表、范例和任何其他附加的信息。您可以访问如下网页获得：

*http://oreil.ly/HP-Drupal*

要询问技术问题或对本书提出建议，请发送电子邮件至：

*bookquestions@oreilly.com*

要获得更多关于我们的书籍、会议、资源中心和O'Reilly 网络的信息，请参见我们的网站：

*http://www.oreilly.com.cn*
*http://www.oreilly.com*

# 第一部分

# 语言

本部分将会从头开始介绍 Swift 这门语言，整个介绍是非常严密且有序的。通过本部分的介绍，你将熟悉并适应 Swift，从而能够进行实际的编程工作。

- 第 1 章从概念与实践上介绍 Swift 程序的结构。你将学习到 Swift 代码文件的组织方式，以及面向对象的 Swift 语言最重要的底层概念：变量与函数、作用域与命名空间，以及对象类型与实例。
- 第 2 章将会介绍 Swift 函数。我们首先会从函数的声明与调用方式基础开始；接下来介绍参数——外部参数名、默认参数与可变参数。然后将会深入介绍 Swift 函数的功能，同时还会介绍函数中的函数、作为一等值的函数、匿名函数、作为闭包的函数，以及柯里化函数。
- 第 3 章首先会介绍 Swift 变量——变量的作用域与生命周期、如何声明与初始化变量，以及一些重要的 Swift 特性，如计算变量与 setter 观察者等。然后会介绍一些重要的内建 Swift 类型，包括布尔、数字、字符串、范围、元组与 Optional。
- 第 4 章将会介绍 Swift 对象类型——类、结构体与枚举。本章将会介绍这 3 种对象类型的工作方式，如何声明、实例化与使用它们。接下来会介绍多态与类型转换、协议、泛型及扩展。本章最后将会介绍 Swift 的保护类型（如 AnyObject）与集合类型（Array、Dictionary 与 Set，还包括 Swift 2.0 新引入的用于表示位掩码的选项集合）。
- 第 5 章内容比较庞杂。我们首先会介绍用于分支、循环与跳转的 Swift 流程控制结构，包括 Swift 2.0 的一个新特性——错误处理。接下来将会介绍如何创建自己的 Swift 运算符。本章最后将会介绍 Swift 访问控制（私有性）、内省机制（反射）与内存管理。

第 1 章

# Swift 架构纵览

首先对 Swift 语言的构建方式有个总体认识并了解基于 Swift 的 iOS 程序的样子是很有用的。本章将会介绍 Swift 语言的整体架构与本质特性，后续章节将会对细节进行详尽的介绍。

## 1.1 基础

一个完整的 Swift 命令是一条语句。一个 Swift 文本文件包含了多行文本。换行符是有意义的。一个程序的典型布局就是一行一条语句：

```
print("hello")
print("world")
```

（print 命令会在 Xcode 控制台提供即时反馈。）

可以将多条语句放到一行，不过这就需要在语句间加上分号：

```
print("hello"); print("world")
```

可以将分号放到语句的末尾，也可以在一行上单独放置一个分号，不过没人这么做（除了习惯原因之外，因为 C 和 Objective-C 要求使用分号）：

```
print("hello");
print("world");
```

与之相反，单条语句可以放到多行，这样做可以防止一行中出现过长的语句。不过在这样做的时候要注意语句的位置，以免对 Swift 造成困扰。比如，左圆括号后面就是个不错的位置：

```
print(
    "world")
```

一行中双斜线后面的内容会被当作注释（即所谓的 C++ 风格的注释）：

```
print("world") // this is a comment, so Swift ignores it
```

还可以将注释放到 /*...*/ 中，就像 C 一样。与 C 不同，Swift 风格的注释是可以嵌套的。

Swift 中的很多构建块都会将花括号用作分隔符：

```
class Dog {
    func bark() {
        print("woof")
    }
}
```

根据约定，花括号中的内容由换行符开始，并且通过缩进增强可读性，如上述代码所示。Xcode 会帮助你应用该约定，不过实际情况却是 Swift 并不在意这些，像下面这样的布局也是合法的（有时也更加便捷）：

```
class Dog { func bark() { print("woof") }}
```

Swift 是一门编译型语言。这意味着代码必须要先构建（通过编译器，由文本转换为计算机可以理解的某种底层形式），然后再执行并根据指令完成任务。Swift 编译器非常严格；在编写程序时，你经常会构建并运行，不过你会发现第一次甚至都无法构建成功，原因就在于编译器会识别出一些错误，如果想让代码运行，你就需要修复这些问题。有时候，编译器会给出一些警告；这时代码可以运行，不过一般情况下，你应该有所警戒并修复编译器报出的警告。编译器的严格性是 Swift 最强大的优势之一，可以在代码开始运行前提供最大程度的审计正确性。

Swift 编译器的错误与警告消息涵盖范围非常广，从洞察性极强到一般性提示再到完全误导人。很多时候，你知道某行代码有问题，不过 Swift 编译器却不会清晰地告诉你什么地方出错了，甚至连是哪行都不会告诉你。对于这些情况，我的建议是将可能有问题的代码行放到简单的代码块中，直到发现问题所在位置。虽然提示消息有时起不到帮助作用，不过请保持与编译器的亲密接触吧。请记住，虽然编译器有时无法准确地进行描述，但它知道的一定比你多。

## 1.2 万物皆对象

在 Swift 中，万物皆对象。这与各种现代化面向对象语言是一致的，不过这表示什么意思呢？这取决于你所理解的对象，那“万物”又是什么意思呢？

首先来定义一下对象，大概来说，对象指的是你可以向其发送消息的某个实体。一般来说，消息指的是一种命令指令。比如，你可以向一只狗发送命令：吼叫！坐下！在这种情况下，这些短语就是消息，而狗则是你向其发送消息的对象。

在Swift中，消息发送语法采用的是点符号。首先是对象，然后是一个点，接下来是消息（有些消息后会跟圆括号，不过现在请不用管它，消息发送的完整语法是接下来将会详细介绍的一个主题）。如下是有效的Swift语法：

```
fido.bark()
rover.sit()
```

万物皆对象的想法表明即便是“原生”的语言实体都可以接收消息。比如，1。它是个数字字面值，除此之外别无其他。如果你曾经使用过其他编程语言，那么在Swift中像下面这样做就不会觉得有什么奇怪的了：

```
let sum = 1 + 2
```

不过，让人奇怪的是1后面可以跟着一个点和一条消息。这在Swift中是合法且有意义的（现在可以不用管其含义）：

```
let x = 1.successor()
```

还可以更进一步。回到之前那个1+2代码示例上来。实际上这是一种语法技巧，是表示并隐藏实际情况的一种便捷方式。就好像1实际上是个对象，+是一条消息；不过这条消息使用了特殊的语法（运算符语法）。在Swift中，每个名词都是一个对象，每个动词都是一条消息。

也许判别Swift中的某个实体是不是对象的根本标准在于你能否修改它。在Swift中，对象类型是可以扩展的，这意味着你可以定义该类型下自己的消息。比如，正常情况下你无法向数字发送sayHello消息。不过你可以修改数字类型使得希望达成：

```
extension Int {
    func sayHello() {
        print("Hello, I'm \(self)")
    }
}
1.sayHello() // outputs: "Hello, I'm 1"
```

这样，情况就发生了变化。

在Swift中，1是个对象。在其他一些语言（如Objective-C）中显然不是这样的；1是个原生或标量内建数据类型。区别在于，当我们说万物皆对象时，一方面指的是对象类型，另一方面指的是标量类型。Swift中是不存在标量的；所有类型最终都是对象类型。这就是“万物皆对象”的真正含义。

## 1.3 对象类型的 3 种风格

如果了解 Objective-C 或是其他一些面向对象语言，你可能好奇于 Swift 中的对象 1 是个什么概念。在很多语言（如 Objective-C）中，对象指的是一个类或一个类的实例。Swift 拥有类与实例，你可以向其发送消息；不过在 Swift 中，1 既不是类也不是实例：它是个结构体（struct）。Swift 还有另外一种可以接收消息的实体，叫作枚举。

因此，Swift 拥有 3 种对象类型：类、结构体与枚举。我喜欢称它们为对象类型的 3 种风格。后续内容将会介绍它们之间的差别。不过它们都是确定的对象类型，彼此之间的相似性要远远高于差异性。现在，只需知道这 3 种风格的存在即可。

（如果了解 Objective-C，那么你会惊讶于 Swift 中的结构体与枚举竟然都是对象类型，不过它们并非对象。特别地，Swift 中的结构体要比 Objective-C 的结构体更加重要，使用更为广泛。Swift 与 Objective-C 对待结构体和枚举的不同方式在 Cocoa 中显得尤为重要。）

## 1.4 变量

变量指的是对象的名字。从技术角度来说，它指向一个对象；它是一个对象引用。从非技术角度来看，你可以将其看作存放对象的一个盒子。对象可能会发生变化，或是盒子中的对象被其他对象所替换，但名字却不会发生变化。

在 Swift 中，不存在没有名字的变量，所有变量都必须要声明。如果需要为某个东西起个名字，那么你要说“我在创建一个名字”。可以通过两个关键字实现这一点：`let` 或是 `var`。在 Swift 中，声明通常会伴随着初始化一起——使用等号为变量赋值，并作为声明的一部分。下面这些都是变量声明（与初始化）：

```
let one = 1
var two = 2
```

如果名字存在，那么你就可以使用它了。比如，我们可以将 `two` 中的值修改为 `one` 中的：

```
let one = 1
var two = 2
two = one
```

上面最后一行代码使用了前两行所声明的名字 `one` 与 `two`：等号右侧的名字 `one` 仅仅用于引用盒子中的值（即 1）；不过，等号左侧的名字 `two` 则用于替换掉盒子中的值。这种语句（变量名位于等号左侧）叫作赋值，等号叫作赋值运算符。等号并不是相等性断言，这与数学公式中的等号不同；它是一个命令，表示“获取右侧的值，然后使用它替换掉左侧的值”。

变量的这两种声明方式是不同的，通过 let 声明的名字是不能替换掉其对象的。通过 let 声明的变量是个常量，其值只能被赋予一次并且不再变化。如下代码是无法编译通过的：

```
let one = 1
var two = 2
one = two // compile error
```

可以通过 var 声明一个名字来实现最大的灵活性，不过如果知道永远不会改变变量的初始值，那么最好使用 let，这样 Swift 在处理时效率会更高；事实上，如果本可以使用 let，但你却使用了 var，那么 Swift 编译器就会提示你，并且可以帮你修改。

变量也是有类型的，其类型是在变量声明时创建的，而且永远不会改变。比如，如下代码是无法编译通过的：

```
var two = 2
two = "hello" // compile error
```

一旦声明 two 并将其初始化为 2，那么它就是一个数字了（确切地说是一个 Int），而且一直都将如此。你可以将其替换为 1，因为 1 也是个 Int，但不能将其值替换为“hello”，因为 "hello" 是个字符串（确切地说是一个 String），而 String 并非 Int。

变量有自己的生命——更准确地说是有自己的生命周期。只要变量存在，那么它就会一直保存其值。这样，变量不仅是一种便于命名的手段，还是一种保存值的方式。稍后将会对此做详细介绍。

根据约定，如 String 或 Int（或 Dog、Cat）等类型名要以大写字母开头；变量名则以小写字母开头，请不要违背该约定。如果违背了，那么你的代码虽然还是可以编译通过并正常运行，但其他人却不太容易理解。

## 1.5 函数

如 fido.bark() 或 one = two 这样的可执行代码不能随意放置。一般来说，这样的代码必须要位于函数体中。函数由一系列代码构成，并且可以运行。一般来说，函数有一个名字，这个名字是通过函数声明得到的。函数声明语法的细节将会在后面进行详细介绍，先来看一个示例：

```
func go() {
    let one = 1
    var two = 2
```

```
    two = one
}
```

上述代码描述了要做的一系列事情——声明 one、声明 two，将 one 值赋给 two——并且给这一系列代码赋予一个名字 go；不过该代码序列并不会执行。只有在调用函数时，该代码序列才会执行。我们可以在其他地方这样执行：

```
go()
```

这会向 go 函数发出一个命令，这样 go 函数才会真正运行起来。重申一次，命令本身是可执行代码，因此它不能位于自身当中。它可以位于不同的函数体中：

```
func doGo() {
    go()
}
```

请等一下！这么做有点奇怪。上面是一个函数声明；要想运行该函数，你需要调用 doGo，它才是可执行代码。这看起来像是无穷无尽的循环一样；似乎代码永远都不会运行。如果所有代码都必须位于一个函数中，那么谁来让函数运行呢？初始动力一定来自于其他地方。

幸好，在实际情况下，这个问题并不会出现。记住，你的最终目标是编写 iOS 应用。因此，应用会运行在 iOS 设备（或是模拟器）中，由运行时调用，而运行时已经知道该调用哪些函数了。首先编写一些特殊函数，这些函数会由运行时本身来调用。这样，应用就可以启动了，并且可以将函数放到运行时在某些时刻会调用的地方——比如，当应用启动时，或是当用户轻拍应用界面上的按钮时。

Swift 还有一个特殊的规则，那就是名为 *main.swift* 的文件可以在顶层包含可执行代码，这些代码位于任何函数体的外部，当程序运行时真正执行的其实就是这些代码。你可以通过 *main.swift* 文件来构建应用，不过一般来说没必要这么做。

# 1.6 Swift 文件的结构

Swift 程序可以包含一个或多个文件。在 Swift 中，文件是一个有意义的单元，它通过一些明确的规则来确定 Swift 代码的结构（假设不在 *main.swift* 文件中）。只有某些内容可以位于 Swift 文件的顶层部分，主要有如下几个部分

*模块 import 语句*

模块是比文件更高层的单元。一个模块可以包含多个文件，在 Swift 中，模块中的文

件能够自动看到彼此；但如果没有 import 语句，那么一个模块是看不到其他模块的。比如，想想如何在 iOS 程序中使用 Cocoa：文件的第 1 行会使用 import UIKit。

*变量声明*

声明在文件顶层的变量叫作全局变量：只要程序还在运行，它就会一直存在。

*函数声明*

声明在文件顶层的函数叫作全局函数：所有代码都能看到并调用它，无需向任何对象发送消息。

*对象类型声明*

指的是类、结构体或是枚举的声明。

比如，下面是一个合法的 Swift 文件，包含（只是为了说明）一个 import 语句、一个变量声明、一个函数声明、一个类声明、一个结构体声明，以及一个枚举声明。

```
import UIKit
var one = 1
func changeOne() {
}
class Manny {
}
struct Moe {
}
enum Jack {
}
```

这个示例本身并没有什么意义，不过请记住，我们的目标是探求语言的组成部分与文件的结构，该示例仅仅是为了演示。

此外，示例中每一部分的花括号中还可以加入变量声明、函数声明与对象类型声明。事实上，任何结构化的花括号中都可以包含这些声明。比如，关键字 if（Swift 流程控制的一部分，第 5 章将会介绍）后面会跟着结构化的花括号，它们可以包含变量声明、函数声明与对象类型声明。如下代码虽然毫无意义，但却是合法的：

```
func silly() {
    if true {
        class Cat {}
        var one = 1
        one = one + 1
    }
}
```

你会发现我并没有说可执行代码可以位于文件的顶部，因为这是不行的。只有函数体可以包含可执行代码。函数自身可以包含任意深度的可执行代码；在上述代码中，one =

one + 1这一行可执行代码是合法的，因为它位于if结构中，而该if结构又位于函数体中。但 one = one + 1 这一行不能位于文件顶层，也不能位于 Cat 声明的花括号中。

示例 1-1 是一个合法的 Swift 文件，其中概要地展示了其结构的各种可能性。（请忽略枚举声明中关于 Jack 的 name 变量声明；枚举中的顶层变量有一些特殊规则，稍后将会介绍。）

示例 1-1：合法的 Swift 文件的概要结构

```
import UIKit
var one = 1
func changeOne() {
    let two = 2
    func sayTwo() {
        print(two)
    }
    class Klass {}
    struct Struct {}
    enum Enum {}
    one = two
}
class Manny {
    let name = "manny"
    func sayName() {
        print(name)
    }
    class Klass {}
    struct Struct {}
    enum Enum {}
}
struct Moe {
    let name = "moe"
    func sayName() {
        print(name)
    }
    class Klass {}
    struct Struct {}
    enum Enum {}
}
enum Jack {
    var name : String {
        return "jack"
    }
    func sayName() {
        print(name)
    }
    class Klass {}
    struct Struct {}
    enum Enum {}
}
```

显然，可以一直递归下去：类声明中可以包含类声明，里面的类声明中还可以包含类声明，以此类推。不过这么做毫无意义。

# 1.7 作用域与生命周期

在 Swift 程序中，一切事物都有作用域。这指的是它们会被其他事物看到的能力。一个事物可以嵌套在其他事物中，形成一个嵌套的层次结构。规则是一个事物可以看到与自己相同层次或是更高层次的事物。层次有：

- 模块是一个作用域。
- 文件是一个作用域。
- 对象声明是一个作用域。
- 花括号是一个作用域。

在声明某个事物时，它实际上是在该层级的某个层次上进行的声明。它在层级中的位置（即作用域）决定了是否能被其他事物看到。

再来看看示例 1-1。在 Manny 的声明中是个 name 变量声明和一个 sayName 函数声明；sayName 花括号中的代码可以看到更高层次中花括号之外的内容，因此它可以看到 name 变量。与之类似，changeOne 函数体中的代码可以看到文件顶层所声明的 one 变量；实际上，该文件中的一切事物都可以看到文件顶层所声明的 one 变量。

作用域是共享信息的一种非常重要的手段。声明在 Manny 中的两个不同函数都会看到在 Manny 顶层所声明的 name 变量。Jack 中的代码与 Moe 中的代码都可以看到声明在文件顶层的 one。

事物还有生命周期，这与其作用域是相关的。一个事物的生命周期与其外部作用域的生命周期是一致的。因此，在示例 1-1 中，变量 one 的生命周期就与文件一样，只要程序处于运行状态，one 就是有效的。它是全局且持久的。不过，声明在 Manny 顶层的变量 name 只有在 Manny 存在时才存在（稍后将会对此做出说明）。声明在更深层次中的事物的生命周期会更短；比如，看看下面这段代码：

```
func silly() {
    if true {
        class Cat {}
        var one = 1
        one = one + 1
    }
}
```

在上述代码中，类 Cat 与变量 one 只在代码执行路径通过 if 结构这一短暂的时间内才会存在。当调用函数 silly 时，执行路径就会进入 if 结构中，Cat 会被声明并进入存活状态；接下来，one 被声明并进入存活状态；然后代码行 one = one + 1 会被执行；接下来作用域结束，Cat 与 one 都会消失殆尽。

# 1.8 对象成员

在 3 种对象类型（类、结构体与枚举）中，声明在顶层的事物具有特殊的名字，这在很大程度上是出于历史原因。下面以 Manny 类作为示例：

```
class Manny {
    let name = "manny"
    func sayName() {
        print(name)
    }
}
```

在上述代码中：

- name 是声明在对象声明顶层中的变量，因此叫作该对象的属性。
- sayName 是声明在对象声明顶层中的函数，因此叫作对象的方法。

声明在对象声明顶层的事物（属性、方法以及声明在该层次上的任何对象）共同构成了该对象的成员。成员具有特殊的意义，因为它们定义了你可以向该对象所发送的消息！

# 1.9 命名空间

命名空间指的是程序中的具名区域。命名空间具有这样一个属性：如果事先不穿越区域名这一屏障，那么命名空间外的事物是无法访问到命名空间内的事物的。这是一个好想法，因为通过命名空间，我们可以在不同地方使用相同的名字而不会出现冲突。显然，命名空间与作用域是紧密关联的两个概念。

命名空间有助于解释清楚在一个对象顶层声明另一个对象的意义，比如：

```
class Manny {
    class Klass {}
}
```

通过这种方式来声明 Klass 会使得 Klass 成为一个嵌套类型，并且很好地将其“隐藏”到 Manny 中。Manny 就是个命名空间！ Manny 中的代码可以直接看到 Klass，不过 Manny 外的代码则看不到。需要显式指定命名空间才能穿过命名空间所代表的屏障。要想做到这一点，必须先使用 Manny 的名字，后跟一个点，然后是术语 Klass。简而言之，需要写成 Manny.Klass。

命名空间本身并不会提供安全或隐私；只是提供了便捷的手段而已。因此，在示例 1-1 中，我给 Manny 一个 Klass 类，也给 Moe 一个 Klass 类。不过它们之间并不会出现冲突，因

为它们位于不同的命名空间中，如果必要，我可以通过 Manny.Klass 与 Moe.Klass 来区分它们。

毫无疑问，显式使用命名空间的语法依旧是消息发送点符号语法，事实上，它们是一回事。

实际上，你可以通过消息发送进入本无法进入的作用域。Moe 中的代码不能自动看到 Manny 中声明的 Klass，不过可以采取一个额外的步骤来实现这个目标，即通过 Manny.Klass。之所以可以这么做是因为它能看到 Manny（因为 Manny 声明的层级可以被 Moe 中的代码看到）。

## 1.10 模块

顶层命名空间是模块。在默认情况下，应用本身就是个模块，因此也是个命名空间；大概来说，命名空间的名字就是应用的名字。比如，假如应用叫作 MyApp，那么如果在文件顶层声明一个类 Manny，那么该类的真实名字就是 MyApp.Manny。但通常情况下是不需要这个真实名字的，因为代码已经在相同的命名空间中了，可以直接看到名字 Manny。

框架也是模块，因此它们也是命名空间。比如，Cocoa 的 Foundation 框架（NSString 位于其中）就是个模块。在编写 iOS 程序时，你会 import Foundation（还有可能 import UIKit，它本身又会导入 Foundation），这样就可以直接使用 NSString 而不必写成 Foundation.NSString 了。不过你可以写成 Foundation.NSString，如果在自己的模块中声明了一个不同的 NSString，那么为了区分它们，你就只能写成 Foundation.NSString 了。你还可以创建自己的框架，当然了，它们也都是模块。

如示例 1-1 所示，文件层级之外的是文件所导入的库或模块。代码总是会隐式导入 Swift 本身。还可以显式导入，方式就是以 import Swift 作为文件的开始；但没必要这么做，不过这么做也没什么弊端。

这个事实是非常重要的，因为它解决了一个大谜团：如 print 来自于哪里，为何可以在任何对象的任何消息之外使用它们？事实上，print 是在 Swift.h 头文件的顶层声明的一个函数——你的文件可以看到它，因为它导入了 Swift。它就是个普通的顶层函数，与其他函数一样。你可以写成 Swift.print("hello")，但没必要，因为并不会出现冲突。

你可以查看 *Swift.h* 文件并阅读和研究它，这么做很有帮助。要想做到这一点，请按住 Command 键并单击代码中的 print。此外，还可以显式 import Swift 并按住 Command 键，然后单击 Swift 来查看其头文件！你看不到任何可执行的 Swift 代码，不过却可以查看到所有可用的 Swift 声明，包括如 print 之类的顶层函数、+ 之类

的运算符以及内建类型的声明，如 Int 和 String（查找 `struct Int`、`struct String` 等）。

# 1.11 实例

对象类型（类、结构体与枚举）都有一个共同的重要特性：它们可以实例化。事实上，在声明一个对象类型时，你只不过是定义了一个类型而已。实例化类型则是创建该类型的一个实例。

比如，我可以声明一个 Dog 类并为其添加一个方法：

```
class Dog {
    func bark() {
        print("woof")
    }
}
```

但程序中实际上并没有任何 Dog 对象。我只不过是描述了 Dog 类型。要想得到一个实际的 Dog，我需要创建一个。

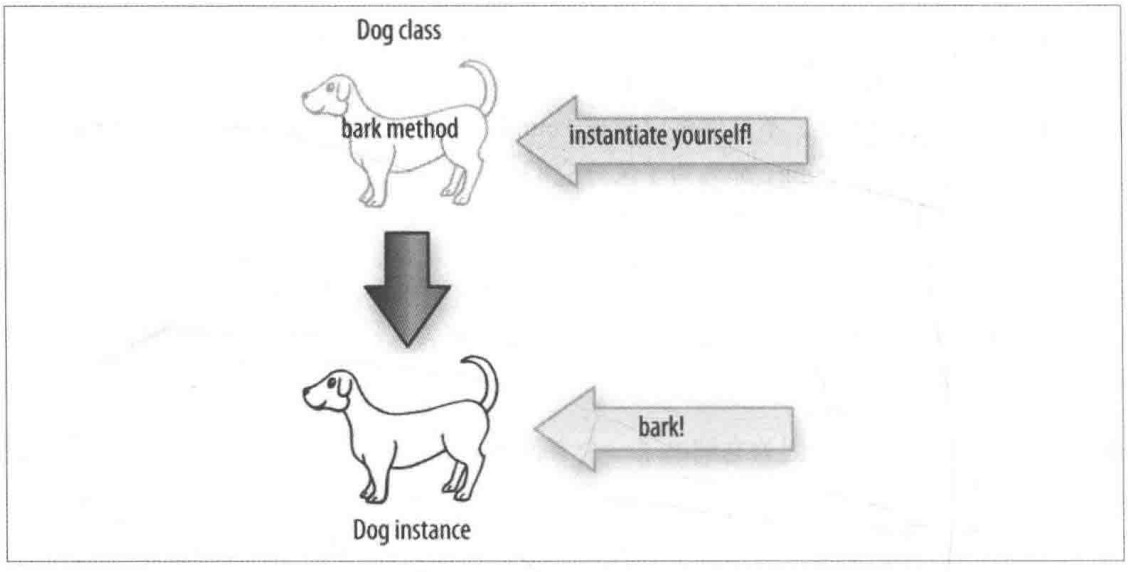


**图 1-1：创建实例并调用实例方法**

创建一个类型为 Dog 类的实际的 Dog 对象的过程就是实例化 Dog 的过程。结果就是一个全新的对象——一个 Dog 实例。

在 Swift 中，实例可以通过将对象类型名作为函数名并调用该函数来创建。这里使用了

圆括号。在将圆括号附加到对象类型名时，你实际上向该对象类型发送了一条非常特殊的消息：实例化自身！

现在来创建一个 Dog 实例：

```
let fido = Dog()
```

上述代码蕴涵着很多事情！我做了两件事：实例化了 Dog，得到一个 Dog 实例；还将该 Dog 实例放到了名为 fido 的盒子中——我声明了一个变量，然后通过将新的 Dog 实例赋给它来初始化该变量。现在 fido 是一个 Dog 实例。（此外，由于使用了 let，因此 fido 将总是指向这个 Dog 实例。我可以使用 var，但即便如此，将 fido 初始化为一个 Dog 实例依然表示 fido 将只能指向某个 Dog 实例）。

既然有了一个 Dog 实例，我可以向其发送实例消息。你觉得会是什么呢？它们是 Dog 的属性与方法！比如：

```
let fido = Dog()
fido.bark()
```

上述代码是合法的。不仅如此，它还是有效的：它会在控制台中输出 "woof"。我创建了一个 Dog，并且让其吼叫（如图 1-1 所示）！

这里有一些重要的事情要说明一下。在默认情况下，属性与方法是实例属性与实例方法。你不能将其作为消息发送给对象类型本身；只能将这些消息发送给实例。正如下面的代码所示，这么做是不合法的，无法编译通过：

```
Dog.bark() // compile error
```

可以声明一个函数 bark，使得 Dog.bark() 调用变成合法调用，不过这是另外一种函数（类函数或是静态函数）。如果声明了这样的函数就可以这么调用了。

属性也一样。为了说明问题，我们为 Dog 增加一个 name 属性。到目前为止，每一个 Dog 都有一个名字，这是通过为变量 name 赋值而得到的。不过该名字并不属于 Dog 对象本身。name 属性如下所示：

```
class Dog {
    var name = ""
}
```

这样就可以设置 Dog 的 name 了，不过需要是 Dog 的实例：

```
let fido = Dog()
fido.name = "Fido"
```

还可以声明属性 name，使得 Dog.name 这种写法变成合法的，不过这是另外一种属性——类属性或是静态属性——如果这么声明了，那就可以这样使用。

## 1.12 为何使用实例

虽然没有实例这个事物，不过一个对象类型本身就是个对象。之所以这么说是因为我们可以向对象类型发送消息：可以将对象类型看作命名空间，并显式进入该命名空间中（如 Manny.Klass）。此外，既然存在类成员与静态成员，我们可以直接调用类、结构体或枚举类型的方法，还可以引用类、结构体或枚举类型的属性。既然如此，实例还有存在的必要吗？

答案与实例属性的本质有关。实例属性的值的定义与特定的实例有关，这正是实例真正的用武之地。

再来看看 Dog 类。我为它添加了一个 name 属性和一个 bark 方法；请记住，它们分别是实例属性与实例方法：

```
class Dog {
    var name = ""
    func bark() {
        print("woof")
    }
}
```

Dog 实例刚创建出来时 name 是空的（一个空字符串）。不过其 name 属性是个 var，因此一旦创建了 Dog 实例，我们就可以为其 name 赋予一个新的 String 值：

```
let dog1 = Dog()
dog1.name = "Fido"
```

还可以获取 Dog 实例的 name：

```
let dog1 = Dog()
dog1.name = "Fido"
print(dog1.name) // "Fido"
```

重要之处在于我们可以创建多个 Dog 实例，两个不同的 Dog 实例可以拥有不同的 name 属性值（如图 1-2 所示）：

```
let dog1 = Dog()
dog1.name = "Fido"
let dog2 = Dog()
dog2.name = "Rover"
print(dog1.name) // "Fido"
print(dog2.name) // "Rover"
```

注意，Dog 实例的 name 属性与 Dog 实例所赋予的变量名之间没有任何关系。该变量只不过是一个盒子而已。你可以将一个实例从一个盒子传递给另一个。不过实例本身会维护自己的内在统一性：

```
let dog1 = Dog()
dog1.name = "Fido"
var dog2 = Dog()
dog2.name = "Rover"
print(dog1.name) // "Fido"
print(dog2.name) // "Rover"
dog2 = dog1
print(dog2.name) // "Fido"
```

上述代码并不会改变 Rover 的 name；它改变的是 dog2 盒子中的狗，将 Rover 替换成了 Fido。

基于对象编程的威力现在开始显现出来了。有一个 Dog 对象类型，它定义了什么是 Dog。我们对 Dog 的声明表示任何一个 Dog 实例都有一个 name 属性和一个 bark 方法。不过每个 Dog 实例都有自己的 name 属性值。它们是不同的实例，分别维护着自己的内部状态。因此，相同对象类型的多个实例的行为都是类似的：Fido 与 Rover 都可以吼叫，如果向它们发送 bark 消息它们就会这么做，但它们是不同的实例，可以有不同的属性值：Fido 的 name 是 "Fido"，而 Rover 的 name 是 "Rover"。

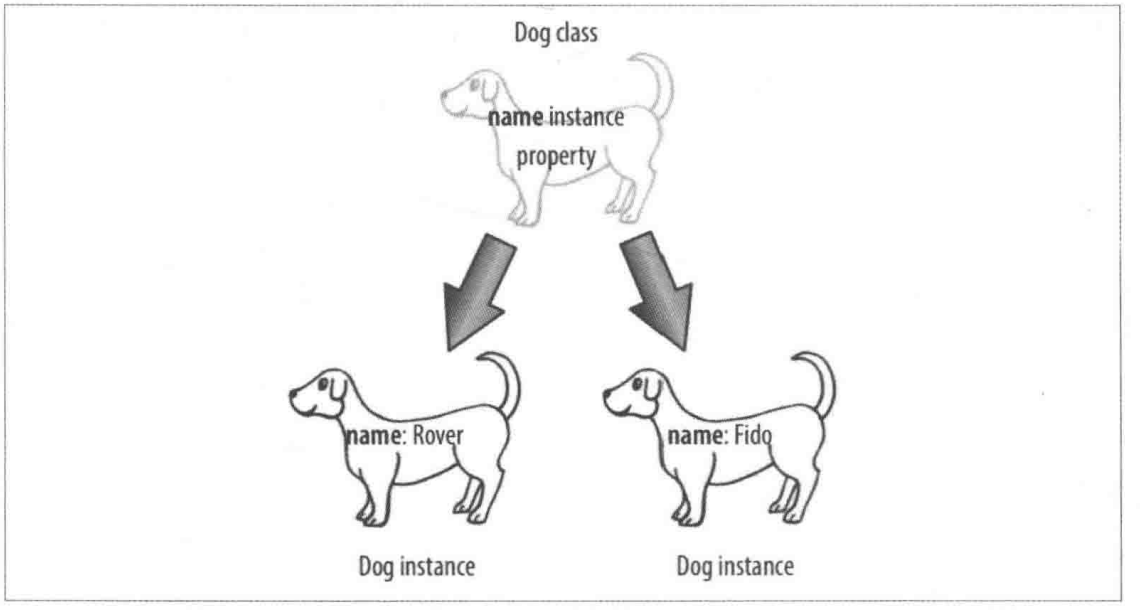


图 1-2：具有不同属性值的两只狗

（对于数字 1 与 2 来说同样如此，不过事实却并不是那么显而易见。Int 是一个 value 属性。

1 是一个 Int，其值是 1，2 表示值为 2 的 Int。不过，这一事实在实际开发中却没那么有趣，因为你显然不会改变 1 的值！）

实例是其类型的实例方法的反映，但这并非全部；它还是实例属性的集合。对象类型负责实例所拥有的属性，但对这些属性的值并不是必需的。当程序运行时，值可以发生变化，并且只会应用到特定的实例上。实例是特定属性值的集合。

实例不仅要负责值，还要负责属性的生命周期。假设我们创建了一个 Dog 实例，并为其 name 属性赋予了值 "Fido"。那么只要我们没有使用其他值替换掉其 name 值并且该实例处在存活状态，那么该 Dog 实例就会一直持有字符串 "Fido"。

简而言之，实例既包含代码又包含数据。代码来自于实例的类型，并且与该类型的所有其他实例共享，不过数据只属于该实例本身。只要实例存在，其数据就一直存在。在任意时刻，实例都有一个状态——自身属性值的完整集合。实例是维护状态的一个设备，它是数据的一个存储箱。

## 1.13 self

实例是一个对象，对象则是消息的接收者。因此，实例需要通过一种方式才能将消息发送给自己。这是通过神奇的单词 self 实现的。该单词可以用在需要恰当类型的实例的情况下。

比如，假设我想要在一个属性中记录下 Dog 吼叫时所喊出的内容，即 "woof"。接下来，在 bark 的实现中，我需要使用该属性，可以像下面这样做：

```
class Dog {
    var name = ""
    var whatADogSays = "woof"
    func bark() {
        print(self.whatADogSays)
    }
}
```

与之类似，假设我想要编写一个实例方法 speak，表示 bark 的同义词。该 speak 实现只需调用自己的 bark 方法即可。可以像下面这样做：

```
class Dog {
    var name = ""
    var whatADogSays = "woof"
    func bark() {
        print(self.whatADogSays)
    }
    func speak() {
```

```
            self.bark()
        }
    }
```

注意该示例中 self 只出现在实例方法中。当一个实例的代码使用 self 时，表示引用该实例。如果表达式 self.name 出现在 Dog 的实例方法代码中，它表示该 Dog 实例的名字，即此时此刻运行该代码的实例。

实际上 self 是完全可选的，你可以省略它，结果完全一样：

```
class Dog {
    var name = ""
    var whatADogSays = "woof"
    func bark() {
        print(whatADogSays)
    }
    func speak() {
        bark()
    }
}
```

原因在于如果省略消息接收者，那么你所发送的消息就会发送给 self，编译器会在底层将 self 作为消息接收者。不过，我从来不会这么做（除非写错了）。作为一种风格，我喜欢显式在代码中使用 self。我觉得省略 self 的代码的可读性与可理解性都会变得很差。在某些情况下，self 是不能省略的，因此我倾向于在可能的情况下都使用 self。

## 1.14 隐私

我之前曾说过，命名空间本身并非访问内部名字的不可逾越的屏障。不过如果你愿意，它还是可以作为屏障的。比如，并非存储在实例中的所有数据都需要修改，甚至要对其他实例可见。并非每一个实例方法都可以被其他实例调用。任何优秀的基于对象的编程语言都需要通过一种方式确保其对象成员的隐私性——如果不希望其他对象看到这些成员，那么可以通过这种方式做到这一点。

比如：

```
class Dog {
    var name = ""
    var whatADogSays = "woof"
    func bark() {
        print(self.whatADogSays)
    }
    func speak() {
        print(self.whatADogSays)
    }
}
```

对于上述代码来说，其他对象可以修改属性 whatADogSays。由于该属性由 bark 与 speak 使用，因此最后可能出现的结果就是，我们让一只 Dog 吼叫，但它却发出“猫叫声”。这显然不是我们想要的：

```
let dog1 = Dog()
dog1.whatADogSays = "meow"
dog1.bark() // meow
```

你可能会说：真够笨的了，为何要使用 var 声明 whatADogSays 呢？使用 let 声明不就行了，让它成为常量。现在就没人能够修改它了：

```
class Dog {
    var name = ""
    let whatADogSays = "woof"
    func bark() {
        print(self.whatADogSays)
    }
    func speak() {
        print(self.whatADogSays)
    }
}
```

这个答案还不错，但并不是最好的答案。它有两个问题。假设我希望 Dog 实例本身能够修改 self.whatADogSays，那么 whatADogSays 就必须得是 var 了；否则，即便实例本身也无法修改它。此外，假设我不希望其他对象知道这只 Dog 吼的是什么，除非调用 bark 或是 speak 才可以。即便使用 let 声明，其他对象依然还是可以读取到 whatADogSays 的值。我可不想这样。

为了解决这个问题，Swift 提供了关键字 private。稍后将会介绍这个关键字的全部含义，不过现在只要知道它可以解决问题就行了：

```
class Dog {
    var name = ""
    private var whatADogSays = "woof"
    func bark() {
        print(self.whatADogSays)
    }
    func speak() {
        print(self.whatADogSays)
    }
}
```

现在，name 是个公有属性，但 whatADogSays 却是个私有属性：其他对象是看不到它的。一个 Dog 实例可以使用 self.whatADogSays，但引用 Dog 实例的不同对象（如 dog1）就无法使用 dog1.whatADogSays。

这里要说明的重要的一点是，对象成员默认是公有的，如果需要访问隐私信息，那就需

要请求。类声明定义了一个命名空间；该命名空间要求其他对象通过额外的点符号来引用命名空间内部的事物，不过其他对象依然可以引用命名空间内部的事物；命名空间本身并不会对可见性形成屏障，而 private 关键字则形成了这道屏障。

# 1.15 设计

现在你已经知道了何为对象，何为实例。不过，程序需要什么对象类型呢，这些对象类型应该包含哪些方法与属性呢，何时以及如何实例化它们呢，该如何使用这些实例呢？遗憾的是，我无法告诉你这些，这是一门艺术，基于对象编程的艺术。我能告诉你的是，在设计与实现基于对象的程序时，你的首要关注点应该是什么，程序正是通过这个过程不断发展起来的。

基于对象的程序设计必须要基于对对象本质的深刻理解之上。你设计出的对象类型需要能够封装好正确的功能（方法）以及正确的数据（属性）。接下来，在实例化这些对象类型时，你要确保实例拥有正确的生命周期，恰到好处地公开给其他对象，并且要能实现对象之间的通信。

## 1.15.1 对象类型与 API

程序文件只会包含极少的顶层函数与变量。对象类型的方法与属性（特别是实例方法与实例属性）实现了大多数动作。对象类型为每个具体实例赋予了专门的能力。它们还有助于更有意义地组织程序代码，并使其更易维护。

可以用两个短语来总结对象的本质：功能封装与状态维护（我最早是在 *REALbasic: The Definitive Guide* 一书中给出的这个总结）。

*功能封装*

每个对象都有自己的职责，并且在自身与外界对象（从某种意义上说是程序员）之间架起一道不透明的墙，外界只能通过这道墙访问方法与动作，而这是通过向其发送相应的消息实现的。在背后，这些动作的实现细节是隐藏起来的，其他对象无须知晓。

*状态维护*

每个实例都有自己所维护的一套数据。通常来说，这些数据是私有的，因此是被封装的；其他对象不知道这些数据是什么，也不清楚其形式是怎样的。对于外界来说，探求对象所维护的私有数据的唯一方式就是通过公有方法或是属性。

比如，假设有一个对象，它实现了栈的功能——也许是 Stack 类的实例。栈是一种数据结构，以 LIFO（后进先出）的顺序维护着一组数据，它只会响应两个消息：push 与

pop。push 表示将给定数据添加到集合中，pop 表示从集合中删除最近刚加进去的数据。栈像是一摞盘子：盘子会一个接着一个地放到栈上面或从栈上移除，因此只有将后面添加的盘子都移除后才能移除第一个添加的盘子（如图 1-3 所示）。

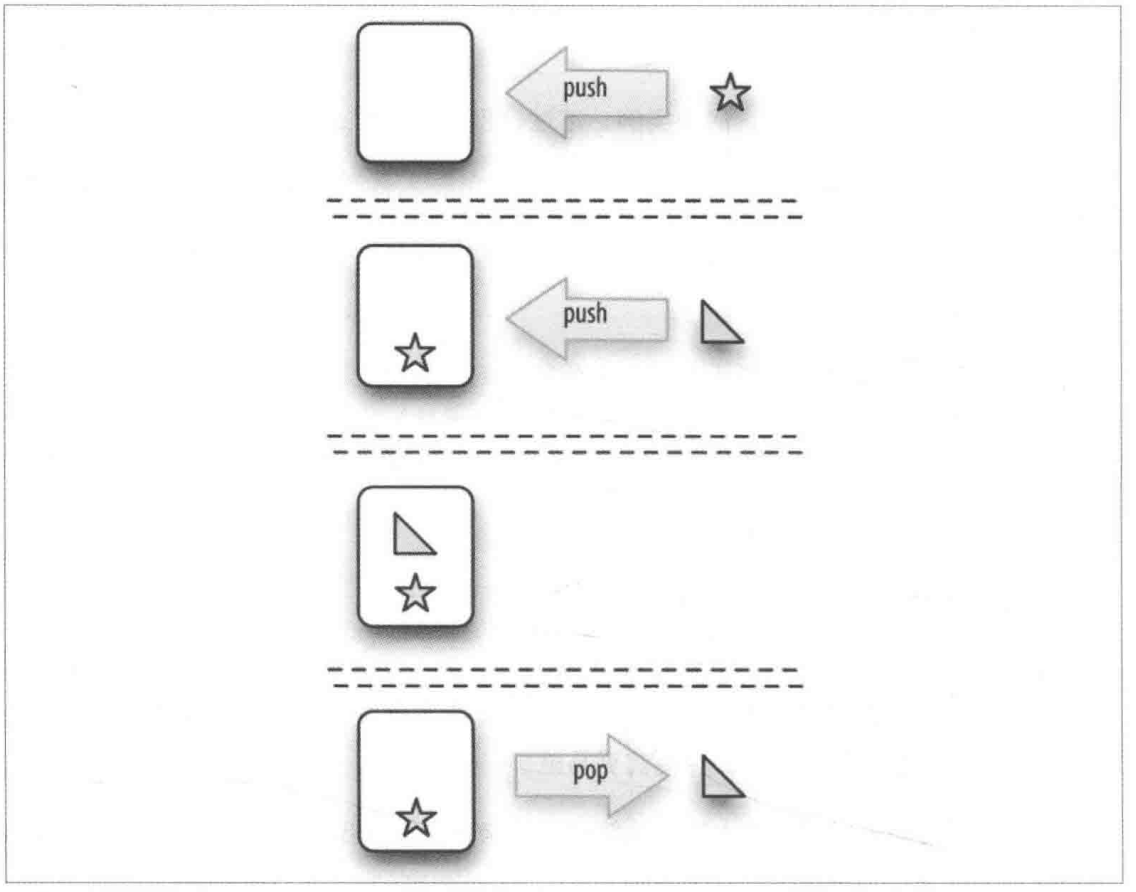


图 1-3：栈

栈对象很好地说明了功能的封装，因为外部对栈的实现方式一无所知。它可能是个数组，也可能是个链表，还有可能是其他实现。不过客户端对象（向栈对象发送 push 或 pop 消息的对象）对此却并不在意，只要栈对象坚持其行为要像一个栈这样的契约即可。这对于程序员来说也非常棒，在开发程序时，它们可以将一个实现替换为另外的实现，同时又不会破坏程序的机制。恰恰相反，栈对象不知道，也不关心到底是谁向其发送 push 或 pop 消息以及为何发送。它只不过以可靠的方式完成自己的工作而已。

栈对象也很好地说明了状态维护，因为它不仅仅是栈数据的网关，它就是栈数据本身。其他对象可以访问到栈的数据，但只能通过访问栈对象本身的方式才行，栈对象也只允许通过这种方式来访问。栈数据位于栈对象内部；其他对象是看不到的。其他对象能做的只是 push 或 pop。如果某个对象位于栈对象的顶部，那么无论哪个对象向其发送 pop

消息，栈对象都会收到这个消息并将顶部对象返回。如果没有对象向这个栈对象发送 pop 消息，那么栈顶部的对象就会一直待在那里。

每个对象类型可以接收的消息总数（即 API，应用编程接口）类似于你可以让这个对象类型所做的事项列表。对象类型将你的代码划分开来；其 API 构成了各部分之间通信的基础。

在实际情况下，当编写 iOS 程序时，你所使用的大多数对象类型都不是你自己的，而是苹果公司提供的。Swift 本身自带了一些颇具价值的对象类型，如 String 和 Int；你可能还会使用 `import UIKit`，它包含了为数众多的对象类型，这些对象类型会涌入你的程序中。你无须创建这些对象类型，只要学习如何使用它们就可以了，你可以查阅公开的 API，即文档。苹果公司自己的 Cocoa 文档包含了大量页面，每一页都列出并介绍了一种对象类型的属性与方法。比如，要想了解可以向 NSString 实例发送什么消息，你可以先研究一下 NSString 类的文档。其页面包含属性与方法的长长的列表，告诉你 NSString 对象能做什么。文档上并不会列出关于 NSString 的一切，不过大部分内容都可以在上面找到。

在编写代码前，苹果公司已经替你做了很多思考与规划。因此，你会大量使用苹果公司提供的对象类型。你也可以创建全新的对象类型，但相对于使用现有的对象类型来说，自己创建的比例不是很大。

## 1.15.2 实例创建、作用域与生命周期

Swift 中重要的实体基本上就是实例了。对象类型定义了实例的种类以及每一种实例的行为方式。不过这些类型的具体实例都是持有状态的“实体”，这也是程序最为关注的内容，实例方法与属性就是可以发送给实例的消息。因此，程序能够发挥作用是离不开实例的帮助的。

不过在默认情况下，实例是不存在的！回头看看示例 1-1，我们定义了一些对象类型，但却并没有创建实例。如果运行程序，那么对象类型从一开始就会存在，但仅仅只是存在而已。我们实现了一种可能性，能够创建一些可能存在的对象的类型，但在这种情况下，其实什么都不会发生。

实例并不会自动产生。你需要对类型进行实例化才能得到实例。因此，程序的很多动作都会包含实例化类型。当然，你希望保留这些实例，因此还会将每个新创建的实例赋给一个变量，让这个变量来持有它、为其命名，并保持其生命周期。实例的生命周期取决于引用它的变量的生命周期。根据引用实例的变量的作用域，一个实例可能会对其他实例可见。

基于对象编程的艺术的要点就在于此，赋予实例足够的生命周期并让它对其他实例可见。你常常会将实例放到特定的盒子中（将其赋给某个变量、在某个作用域中声明），这多亏了变量生命周期与作用域规则，如果需要，实例会留存足够长的时间供程序使用，其他代码也可以获得它的引用并与之通信。

规划好如何创建实例、确定好实例的生命周期以及实例之间的通信这件事让人望而生畏。幸好，在实际情况下，当编写 iOS 程序时，Cocoa 框架本身会再一次为你提供初始框架。

比如，你知道对于 iOS 应用来说，你需要一个应用委托类型和一个视图控制器类型；实际上，在创建 iOS 应用项目时，Xcode 会帮你完成这些事情。此外，当应用启动时，运行时会帮你实例化这些对象类型，并且构建好它们之间的关系。运行时会创建出应用委托实例，并且让其在应用的生命周期中保持存活状态；它会创建一个窗口实例，并将其赋给应用委托的一个属性；它还会创建一个视图控制器实例，并将其赋给窗口的一个属性。最后，视图控制器实例有一个视图，它会自动出现在窗口中。

这样，你无须做任何事情就拥有了在应用生命周期中一直存在的一些对象，包括构成可视化界面的基础对象。重要的是，你已经拥有了定义良好的全局变量，可以引用所有这些对象。这意味着，无须编写任何代码，你就已经可以访问某些重要对象了，并且有了一个初始位置用于放置生命周期较长的其他对象，以及应用所需的其他可视化界面元素。

## 1.15.3 小结

在构建基于对象的程序来执行特定的任务时，我们要理解对象的本质。它们是类型与实例。类型指的是一组方法，用于说明该类型的所有实例可以做什么（功能封装）。相同类型的实例只有属性值是不同的（状态维护）。我们要做好规划。对象是一种组织好的工具，一组盒子，用于封装完成特定任务的代码。它们还是概念工具。程序员要能以离散对象的思维进行思考，他们要能将程序的目标与行为分解为离散的任务，每个任务都对应一个恰当的对象。

与此同时，对象并不是孤立的。对象之间可以合作，这叫作通信，方式则是发送消息。通信途径是多种多样的。对此做出妥善的安排（即架构），从而实现对象之间的协作、有序的关系是基于对象编程最具挑战性的一个方面。在 iOS 编程中，你可以得到 Cocoa 框架的帮助，它提供了初始的对象类型集合以及基础的架构框架。

使用基于对象的编程能够很好地让程序实现你的需求，同时又会保持程序的整洁性和可维护性；你的能力会随着经验的增加而增强。最后，你可能想要进一步阅读关于高效规划及基于对象编程架构方面的读物。我推荐两本经典、值得收藏的好书。Martin Fowler 所写的 *Refactoring*（Addison-Wesley, 1999）介绍了如何根据哪些方法应该属于哪些类而

重新调整代码的原则（如何战胜恐惧以实现这一点）。 Erich Gamma、Richard Helm、Ralph Johnson 及 John Vlissides（又叫作“四人组”）所写的 *Design Patterns*（本书中文版《设计模式：可复用面向对象软件的基础》已由机械工业出版社引进出版，书号：978-7-111-07575-2。）是架构基于对象程序的宝典，书中列举了如何根据正确的原则以及对象之间的关系来安排对象的所有方法（Addison-Wesley, 1994）。

# 第 2 章

# 函数

Swift 语法中最具特色的就是声明与调用函数的方式了，没什么比这更重要！就像第 1 章所介绍的那样，所有代码都位于函数中；而动作则是由函数触发的。

## 2.1 函数参数与返回值

函数就像是小学数学课本中所讲的那种能够处理各种东西的机器一样。你知道我说的是什么：上面是一个漏斗，中间是一些齿轮和曲柄，下面的管子就会出来东西。函数就像这样的机器一样：你提供一些东西，然后由特定的机器进行处理，最后东西就生成出来了。

提供的东西是输入，出来的东西是输出。从技术的角度来看，函数的输入叫作参数，输出叫作结果。比如，下面这个简单的函数有两个 Int 值输入，然后将它们相加，最后输出相加的和：

```
func sum (x:Int, _ y:Int) -> Int {
    let result = x + y
    return result
}
```

这里所用的语法是非常严格且定义良好的，如果理解不好你就没法用好 Swift。下面来详细介绍；我将第 1 行分为几块来分别介绍：

```
func sum ❶
    (x:Int, _ y:Int) ❷❸
    -> Int { ❹
        let result = x + y ❺
        return result ❻
}
```

❶ 声明从关键字 func 开始，后跟函数的名字，这里就是 sum。调用函数时必须使用这个名字，所谓调用就是运行函数所包含的代码。

❷ 函数名后面跟着的是其参数列表，至少要包含一对圆括号。如果函数接收参数（输入），那么它们就会列在圆括号中，中间用逗号隔开。每个参数都有严格的格式：参数名，一个冒号，然后是参数类型。这里的 sum 函数接收两个参数：一个是 Int，其名字为 x；另一个也是 Int，其名字为 y。

值得注意的是，名字 x 与 y 是随意起的，它们只对该函数起作用。它们与其他函数或是更高层次作用域中所使用的其他 x 与 y 是不同的。定义这些名字的目的在于让参数值能有自己的名字，这样就可以在函数体的代码块中引用它们了。实际上，参数声明是一种变量声明：我们声明变量 x 与 y 以便在该函数中使用它们。

❸ 该函数声明的参数列表中第 2 个参数名前还有一个下划线（_）和一个空格。现在我不打算介绍这个下划线，只是这个示例需要用到它，因此相信我就行了。

❹ 圆括号之后是一个箭头运算符→，后跟函数所返回的值类型。接下来是一对花括号，包围了函数体——实际代码。

❺ 在花括号中（即函数体），作为参数名定义的变量进入生命周期，其类型是在参数列表中指定的。我们知道，只有当函数被调用并将值作为参数传递进去时，这些代码才会运行。

这里的参数叫作 x 与 y，这样我们就可以安全地使用这些值，通过其名字来引用它们，我们知道这些值都会存在，并且是 Int 值，这是通过参数列表来指定的。不仅是程序员，编译器也可以确保这一点。

❻ 如果函数有返回值，那么它必须要使用关键字 return，后跟要返回的值。该值的类型要与之前声明的返回值类型（位于箭头运算符之后）相匹配。

这里返回了一个名为 result 的变量；它是通过将两个 Int 值相加创建出来的，因此也是个 Int，这正是该函数所生成的结果。如果尝试返回一个 String（return "howdy"）或是省略掉 return 语句，那么编译器就会报错。

关键字 return 实际上做了两件事。它返回后面的值，同时又终止了函数的执行。return 语句后面可以跟着多行代码，不过如果这些代码行永远都不会执行，那么编译器就会发出警告。

花括号之前的函数声明是一种契约，描述了什么样的值可以作为输入，产生的输出会是什么样子。根据该契约，函数可以接收一定量的参数，每个参数都有确定的类型，并会生成确定类型的结果。一切都要遵循这个契约。花括号中的函数体可以将参数作为局部变量。返回值要与声明的返回类型相匹配。

这个契约会应用到调用该函数的任何地方。如下代码调用了 sum 函数：

```
let z = sum(4,5)
```

重点关注等号右侧——sum(4,5)，这是个函数调用。它是如何构建的呢？它使用了函数的名字、名字后跟一对圆括号；在圆括号里面是逗号分隔的传递给函数参数的值。从技术角度来说，这些值叫作实参。我这里使用了 Int 字面值，不过还可以使用 Int 变量；唯一的要求就是它要有正确的类型：

```
let x = 4
let y = 5
let z = sum(y,x)
```

在上述代码中，我故意使用了名字 x 与 y，这两个变量值会作为参数传递给函数，而且还在调用中将它们的顺序有意弄反，从而强调这些名字与函数参数列表及函数体中的名字 x 与 y 没有任何关系。对于函数来说，名字并不重要，值才重要，它们的值叫作实参。

函数的返回值呢？该值会在函数调用发生时替换掉函数调用。上述代码就是这样的，其结果是 9。因此，最后一行就像我说过的：

```
let z = 9
```

程序员与编译器都知道该函数会返回什么类型的值，还知道在什么地方调用该函数是合法的，什么地方调用是不合法的。作为变量 z 声明的初始化来调用该函数是没问题的，这相当于将 9 作为声明的初始化部分：在这两种情况下，结果都是 Int，因此 z 也会声明为 Int。不过像下面这样写就不合法了：

```
let z = sum(4,5) + "howdy" // compile error
```

由于 sum 返回一个 Int，这相当于将 Int 加到一个 String 上。在默认情况下，Swift 中不允许这么做。

忽略函数调用的返回值也是可以的：

```
sum(4,5)
```

上述代码是合法的；它既不会导致编译错误，也不会造成运行错误。这么做有点无聊，因为我们历尽辛苦使得 sum 函数能够实现 4 与 5 相加的结果，但却没有用到这个结果，而是将其丢弃掉了。不过，很多时候我们都会忽略掉函数调用的返回值；特别是除了返回值之外，函数还会做一些其他事情（从技术上来说叫作副作用），而调用函数的目的只是让函数能够做一些其他事情。

如果在使用 Int 的地方调用 sum，而且 sum 的参数都是 Int 值，那是不是就表示你可以在 sum 调用中再调用 sum 呢？当然了！这么做是完全合法的，也是合情合理的：

```
let z = sum(4,sum(5,6))
```

这样写代码存在着一个争议，那就是你可能被自己搞晕，而且会导致调试变得困难。不过从技术上来说，这么做是正常的。

## 2.1.1 Void 返回类型与参数

下面回到函数声明。关于函数参数与返回类型，存在以下两种情况能够让我们更加简洁地表示函数声明。

*无返回类型的函数*

没有规定说函数必须要有返回值。函数声明可以没有返回值。在这种情况下，有 3 种声明方式：可以返回 Void，可以返回 ()，还可以完全省略箭头运算符与返回类型。如下声明都是合法的：

```
func say1(s:String) -> Void { print(s) }
func say2(s:String) -> () { print(s) }
func say3(s:String) { print(s) }
```

如果函数没有返回值，那么函数体就无须包含 return 语句。如果包含了 return 语句，那么其目的就纯粹是在该处终止函数的执行。

return 语句通常只会包含 return。不过，Void（无返回值的函数的返回类型）是 Swift 中的一个类型。从技术上来说，无返回值的函数实际上返回的就是该类型的值，可以表示为字面值 ()。（第 3 章将会介绍字面值 () 的含义。）因此，函数声明 return() 是合法的；无论是否声明，() 就是函数所返回的。写成 return() 或 return;（后面加上一个分号）有助于消除歧义，否则 Swift 可能认为函数会返回下一行的内容。

如果函数无返回值，那么调用它纯粹就是为了函数的副作用；其调用结果无法作为更大的表达式的一部分，这样函数中代码的执行就是函数要做的唯一一件事，返回的 () 值会被忽略。不过，也可以将捕获的值赋给 Void 类型的变量；比如：

```
let pointless : Void = say1("howdy")
```

*无参数的函数*

没有规定说函数必须要有参数。如果没有参数，那么函数声明中的参数列表就可以完全为空。不过，省略参数列表圆括号本身是不可以的！圆括号需要出现在函数声明中，位于函数名之后：

```
func greet1() -> String { return "howdy" }
```

显然，函数可以既没有返回值，也没有参数；如下代码的含义是一样的：

```
func greet1() -> Void { print("howdy") }
func greet2() -> () { print("howdy") }
func greet3() { print("howdy") }
```

就像不能省略函数声明中的圆括号（参数列表）一样，你也不能省略函数调用中的圆括号。如果函数没有参数，那么这些圆括号就是空的，但必须要有。比如：

```
greet1()
```

注意上述代码中的圆括号！

### 2.1.2 函数签名

如果省略函数声明中的参数名，那就完全可以通过输入与输出的类型来描述一个函数了，比如，下面这个表达式：

```
(Int, Int) -> Int
```

事实上，这在 Swift 中是合法的表达式。它是函数签名。在该示例中，它表示的是 sum 函数的签名。当然，还可能有其他函数也接收两个 Int 参数并返回一个 Int，这就是要点。该签名描述了所有接收这些类型、具有这些数量的参数，并返回该类型结果的函数。实际上，函数的签名是其类型——函数的类型。稍后将会看到，函数拥有类型是非常重要的。

函数的签名必须要包含参数列表（无参数名）与返回类型，即便其中一样或两者都为空亦如此；因此，不接收参数且无返回值的函数签名可以有 4 种等价的表示方式，包括 `Void -> Void` 与 `() -> ()`。

## 2.2 外部参数名

函数可以外化其参数名。外部名称要作为实参的标签出现在对函数的调用中。这么做是有意义的，原因如下：

- 阐明了每个实参的作用；每个实参名都能表示出自己对函数动作的作用。
- 将函数区分开来；两个函数可能有相同的名字和签名，但却拥有不同的外化参数名。
- 有助于 Swift 与 Objective-C 和 Cocoa 的通信，后者的方法参数几乎总是有外化名字的。

要想外化参数名，请在函数声明中将外部名字放在内部参数名之前，中间用空格隔开。外部名字可以与内部名字相同，也可以不同。不过在 Swift 中，外化参数名已经形成了标准，因此有如下规则：在默认情况下，除了第一个参数，所有参数名都会自动外化。

这样，如果需要外化一个参数名，并且它不是第一个参数，并且希望外化名与内部名相同，那么你什么都不需要做，Swift 会自动帮你实现。

如下是一个函数声明，该函数会将一个字符串拼接到自身 times 次：

```
func repeatString(s:String, #times:Int) -> String {
    var result = ""
    for _ in 1...times { result += s }
    return result
}
```

该函数的第 1 个参数只有内部名字，不过第 2 个参数有一个外部名字，它与内部名字相同，都叫作 times。下面是调用该函数的代码：

```
let s = repeatString("hi", times:3)
```

如你所见，在调用中，外部名作为一个标签位于实参之前，中间用冒号隔开。

我之前曾经说过，参数的外部名可以与内部名不同。比如，在 repeatString 函数中，我们将 times 作为外部名，将 n 作为内部名。那么，函数声明将会如下所示：

```
func repeatString(s:String, times n:Int) -> String {
    var result = ""
    for _ in 1...n { result += s}
    return result
}
```

函数体中现在已经看不到 times 了；times 只用作外部名，在调用时使用。内部名是 n，也是代码所引用的名字。

外部名的存在并不意味着调用可以使用与声明不同的参数顺序。比如，repeatString 接受一个 String 参数和一个 Int 参数，顺序就是这样的。虽然标签可以消除实参对应于哪个参数的歧义，但调用的顺序也需要如此（不过稍后我将给出该规则的一个例外情况）。

repeatString 函数说明了默认规则，即第 1 个参数没有外部名，其他参数则有。为何说这是默认规则呢？一个原因在于第 1 个参数通常不需要外部名，因为函数名通常能够清晰表明第 1 个参数的含义——就像 repeatString 函数一样（重复一个字符串，而该字符串应该由第 1 个参数提供）。另一个原因也是在实际情况中更为重要的，那就是这个约定使得 Swift 函数能够与 Objective-C 方法交互，而后者采取的就是这种方式。

比如，下面是 Cocoa NSString 方法的 Objective-C 声明：

```
- (NSString *)stringByReplacingOccurrencesOfString:(NSString *)target
                                        withString:(NSString *)replacement
```

该方法接收两个 NSString 参数并返回一个 NSString。第 2 个参数的外部名是显而易见的，即 withString。不过第 1 个参数的名字却不那么明显。一方面，你可以说它是 stringByReplacingOccurrencesOfString。另一方面，它实际上并非参数真正的名字；它更像是方法名。实际上，该方法的正式名字是整个字符串 stringByReplacingOccurrencesOfString:withString:。不过，Swift 函数调用语法通过圆括号将函数名与外部参数名区分开来。因此，如果 Swift 想要调用这个 Objective-C 方法，那么冒号前首先就应该是函数名（位于圆括号之前），然后是第 2 个实参的标签（位于圆括号中）。Swift String 与 Cocoa NSString 能够自动桥接彼此，因此可以在 Swift String 上调用这个 Cocoa 方法，如以下代码所示：

```
let s = "hello"
let s2 = s.stringByReplacingOccurrencesOfString("ell", withString:"ipp")
// s2 is now "hippo"
```

如果函数是你自己定义的（即是你声明的），并且 Objective-C 永远不会调用它（这样就没必要遵循 Objective-C 的要求了），那么你可以自由改变其默认行为。你可以在自己的函数声明中做如下事情。

*外化第 1 个参数名*

如果想要外化第 1 个参数名，那么请将外部名放到内部名之前。这两个名字可以相同。

*修改除第 1 个参数之外的其他参数名*

如果想要修改除第 1 个参数外的其他参数的外部名，那么请将所需的外部名放到内部名之前。

*防止对除第 1 个参数外的其他参数进行外化*

要想防止对除第 1 个参数外的其他参数进行外化，请在其前面加上一个下划线和一个空格：

```
func say(s:String, _ times:Int) {
```

现在在调用这个方法时不能对第 2 个参数加标签：

```
let d = Dog()
d.say("woof", 3)
```

（这就解释了本章一开始所作的声明 func sum (x:Int, _ y:Int) -> Int：这里阻止了第 2 个参数名的外化，从而无须提供实参标签。）

### 这个函数的名字是什么？

从技术上来说，一个Swift函数的名字是由圆括号之前的名字加上参数的外部名共同构成的。如果阻止了参数的外部名，那么可以使用一个下划线来表示其外部名。结果就是外部参数名会位于圆括号中，后跟一个冒号。比如，函数声明 `func say(s:String, times:Int)` 从技术上来看就是 `say(_:times:)`，函数声明 `func say(s:String, _ times:Int)` 从技术上来看就是 `say(_:_:)`。这么表示有点烦琐，本书也不会这么使用，不过其优点在于准确和无歧义。

## 2.3 重载

在Swift中，函数重载是合法的，也是常见的。这意味着具有相同函数名（包括外部参数名）的两个函数可以共存，只要签名不同即可。

比如，如下两个函数可以共存：

```
func say (what:String) {
}
func say (what:Int) {
}
```

重载可行的原因在于Swift拥有严格的类型。String一定不是Int。Swift能够在声明中将二者区分开，在函数调用时也能区分开。这样，Swift就能够毫无歧义地知道 `say("what")` 不同于 `say(1)`。

重载也适用于返回类型。具有相同名字与相同参数类型的两个函数可以有不同的返回类型。不过调用上下文一定不能有歧义；也就是说，一定要清楚调用者需要什么样的返回类型。

比如，如下两个函数可以共存：

```
func say() -> String {
    return "one"
}
func say() -> Int {
    return 1
}
```

但现在就不能像下面这样调用了：

```
let result = say() // compile error
```

上述调用是有歧义的，编译器会告诉你这一点。调用上下文一定要清楚期望的返回类型是什么。比如，假设我们有另一个没有重载的函数，它接收一个 String 参数：

```
func giveMeAString(s:String) {
    print("thanks!")
}
```

那么 giveMeAString(say()) 就是合法的，因为只有一个 String 符合，因此我们必须调用返回 String 的 say。与之类似：

```
let result = say() + "two"
```

只有 String 可以“加到” String 上，因此这个 say() 必须是个 String。

如果之前用过 Objective-C，那么你会对 Swift 中重载的合法性感到惊讶，因为在 Objective-C 中是不允许重载的。如果在 Objective-C 中声明了相同方法的两个重载版本，那么编译器就会报“Duplicate declaration”错误。实际上，如果在 Swift 中声明了两个重载方法，但 Objective-C 却能看到它们（参见附录 A 了解详情），那么就会遇到一个 Swift 编译错误，因为这种重载与 Objective-C 是不兼容的。

具有相同签名和不同外部参数名的两个函数并不构成重载；由于函数有着不同的外部参数名，因此它们是名字不同的两个不同函数。

## 2.4 默认参数值

参数可以有一个默认值。这意味着调用者可以完全省略参数，不为其提供实参；那么，其值就是默认值。

要想提供默认值，在声明中的参数类型后追加一个 = 号和默认值：

```
class Dog {
    func say(s:String, times:Int = 1) {
        for _ in 1...times {
            print(s)
        }
    }
}
```

事实上，现在有两个函数，分别是 say 与 say(times:)。如果只想说一次，那么你可以直接调用 say，同时 times: 参数值 1 会提供给你：

```
let d = Dog()
d.say("woof") // same as saying d.say("woof", times:1)
```

如果想要重复，那么就调用 say(times:)：

```
let d = Dog()
d.say("woof", times:3)
```

如果具有外部名的参数有默认值，那就需要按照顺序调用。比如，如果一个函数的声明如下所示：

```
func doThing (a a:Int = 0, b:Int = 3) {}
```

那么，像下面这样调用就是合法的：

```
doThing(b:5, a:10)
```

不过，这可能是 Swift 的一个疏忽，当然，如果有一个参数没有默认值，那么这么调用就是非法的。因此，我建议不要这么做：请保证调用时实参的顺序与声明时形参的顺序一致。

## 2.5 可变参数

参数可以是可变参数。这意味着调用者可以根据需要提供多个该参数类型的值，中间用逗号分隔；函数体会将这些值当作数组。

要想将参数标记为可变参数，参数后要跟着 3 个点，如下所示：

```
func sayStrings(arrayOfStrings:String ...) {
    for s in arrayOfStrings { print(s) }
}
```

下面是调用方式：

```
sayStrings("hey", "ho", "nonny nonny no")
```

在 Swift 的早期版本中，可变参数只能是最后一个参数；不过，Swift 2.0 放宽了这个限制。现在的限制是一个函数最多只能声明一个可变参数（否则就无法确定值列表结束的位置）。比如：

```
func sayStrings(arrayOfStrings:String ..., times:Int) {
    for _ in 1...times {
        for s in arrayOfStrings { print(s) }
    }
}
```

下面是调用方式：

```
sayStrings("Mannie", "Moe", "Jack", times:3)
```

全局 print 函数的第 1 个参数就是个可变参数，因此可以通过一条命令输出多个值：

```
print("Mannie", 3, true) // Mannie 3 true
```

默认参数对输出还做了进一步的细化。默认的 separator: 是个空格（当提供了多个值），默认的 terminator: 是个换行符；你可以修改它们：

```
print("Mannie", "Moe", separator:", ", terminator: ", ")
print("Jack")
// output is "Mannie, Moe, Jack" on one line
```

遗憾的是，Swift 语言中有一个陷阱：没办法将数组转换为逗号分隔的参数列表（相比于 Ruby 中的 splat）。如果一开始就有一个某种类型的数组，那么你不能在需要该类型可变参数的地方使用它。

# 2.6 可忽略参数

局部名为下划线的参数会被忽略。调用者必须要提供一个实参，不过函数体中并没有它的名字，因此无法引用。比如：

```
func say(s:String, times:Int, loudly _:Bool) {
```

函数体中无法使用 loudly 参数，不过调用者还是需要提供第 3 个参数：

```
say("hi", times:3, loudly:true)
```

声明不需要为忽略的参数提供外部名：

```
func say(s:String, times:Int, _:Bool) {
```

不过调用者必须要提供：

```
say("hi", times:3, true)
```

该特性的目的是什么呢？它并非为了满足编译器的要求，因为如果函数体中没有引用某个参数，那么编译器并不会报错。我主要将其作为对自己的一个提示，表示“我知道这里有个参数，只不过故意不使用它而已”。

# 2.7 可修改参数

在函数体中，参数本质上是个局部变量。在默认情况下，它是个隐式使用 let 声明的变量。你无法对其赋值：

```
func say(s:String, times:Int, loudly:Bool) {
    loudly = true // compile error
}
```

如果代码需要在函数体中为参数名赋值，那么请显式使用 var 声明参数名：

```
func say(s:String, times:Int, var loudly:Bool) {
    loudly = true // no problem
}
```

在上述代码中，loudly 依旧是个局部变量。为其赋值并不会修改函数体外任何变量的值。不过，还可以这样配置参数，使得它修改的是函数体外的变量值！一个典型用例就是你希望函数会返回多个结果。比如，我下面要编写一个函数，它会将给定字符串中出现的某个字符全部删除，然后返回删除的字符数量：

```
func removeFromString(var s:String, character c:Character) -> Int {
    var howMany = 0
    while let ix = s.characters.indexOf(c) {
        s.removeRange(ix...ix)
        howMany += 1
    }
    return howMany
}
```

可以这样调用：

```
let s = "hello"
let result = removeFromString(s, character:Character("l")) // 2
```

很好，不过我们忘记了一件事：初始字符串 s 依旧是 "hello"！在函数体中，我们从 String 参数的本地副本中删除了所有出现的 character，不过这个改变并未影响原来的字符串。

如果希望函数能够修改传递给它的实参的初始值，那就需要做出如下 3 个改变：

- 要修改的参数必须声明为 inout。
- 在调用时，持有待修改值的变量必须要声明为 var，而不是 let。
- 相比于将变量作为实参进行传递，我们传递的是地址。这是通过在名字前加上 & 符号做到的。

下面来修改，removeFromString 的声明现在如下所示：

```
func removeFromString(inout s:String, character c:Character) -> Int {
```

对 removeFromString 的调用现在如下所示：

```
var s = "hello"
let result = removeFromString(&s, character:Character("l"))
```

调用之后，结果是 2，s 为 "heo"。注意，名字 s 前的 & 符号是函数调用中的第 1 个参数！我喜欢这么做，因为它强制我显式告诉编译器和我自己，我们要做的事情存在一些潜在的风险：函数会修改函数体之外的一个值，这会产生副作用。

当调用具有 inout 参数的函数时，地址作为实参传递给参数的变量总是会被设定，即便函数没有修改该参数亦如此。

在使用 Cocoa 时，你常常会遇到该模式的变种。Cocoa API 是使用 C 与 Objective-C 编写的，因此你看不到 Swift 术语 inout。你可能会看到一些奇怪的类型，如 UnsafeMutablePointer。不过从调用者的视角来看，它们是一回事。依然是准备 var 变量并传递其地址。

比如，考虑 Core Graphics 函数 CGRectDivide。CGRect 是个表示矩形的结构体。在将一个矩形切分成两个矩形时需要调用 CGRectDivide。CGRectDivide 需要告诉你生成的两个矩形都是什么。因此，它需要返回两个 CGRect。其策略就是函数本身不返回值；相反，它会说："将两个 CGRect 作为实参传递给我，我会修改它们，这样它们就是操作的结果了"。

下面是 CGRectDivide 在 Swift 中的声明：

```
func CGRectDivide(rect: CGRect,
    slice: UnsafeMutablePointer<CGRect>,
    remainder: UnsafeMutablePointer<CGRect>,
    amount: CGFloat,
    edge: CGRectEdge)
```

第 2 个和第 3 个参数都是针对 CGRect 的 UnsafeMutablePointer。如下代码来自于我开发的一个应用，它调用了这个函数；请注意我是如何处理第 2、3 两个实参的：

```
var arrow = CGRectZero
var body = CGRectZero
CGRectDivide(rect, &arrow, &body, Arrow.ARHEIGHT, .MinYEdge)
```

我需要事先创建两个 var CGRect 变量，它们要有值，不过其值立刻会被对 CGRectDivide 的调用所替换，因此我为其赋值 CGRectZero 作为占位符。

Swift 扩展了 CGRect，提供了一个 divide 方法。作为一个 Swift 方法，它实现了一些 Cocoa C 函数做不到的事情：返回两个值（以元组的形式，参见第 3 章）。这样，一开始就无需调用 CGRectDivide 了。不过，你依然可以调用 CGRectDivide，因此了解其调用方式还是很有必要的。

有时，Cocoa 会通过 UnsafeMutablePointer 参数调用你的函数，你可能想要修改其值。为了做到这一点，你不能直接对其赋值，就像 removeFromString 实现中对 inout 变量 s 所做的那样。你使用的是 Objective-C 而非 Swift，这是个 UnsafeMutablePointer 而非 inout 参数。从技术上来说，这是将其赋给了 UnsafeMutablePointer 的内存属性。下面来自于我所编写的代码的一个片段（不做更多的解释）：

```
func popoverPresentationController(
    popoverPresentationController: UIPopoverPresentationController,
    willRepositionPopoverToRect rect: UnsafeMutablePointer<CGRect>,
    inView view: AutoreleasingUnsafeMutablePointer<UIView?>) {
        view.memory = self.button2
        rect.memory = self.button2.bounds
}
```

有时当参数是某个类的实例时，函数需要修改这个没有声明为 inout 的参数，这种情况比较常见。这是类的一个特殊的特性，与其他两种对象类型（枚举与结构体）风格不同。String 不是类，它是个结构体。这也是我们要使用 inout 才能修改 String 参数的原因所在。下面声明一个具有 name 属性的 Dog 类来说明这一点：

```
class Dog {
    var name = ""
}
```

下面这个函数接收一个 Dog 实例参数和一个 String，并将该 Dog 实例的 name 设为该 String。注意这里并未使用 inout：

```
func changeNameOfDog(d:Dog, to tostring:String) {
    d.name = tostring
}
```

下面是调用方式，该调用没有使用 inout，因此直接传递一个 Dog 实例：

```
let d = Dog()
d.name = "Fido"
print(d.name) // "Fido"
changeNameOfDog(d, to:"Rover")
print(d.name) // "Rover"
```

注意，虽然没有将 Dog 实例 d 作为 inout 参数传递，但我们依然可以修改它的属性，即便它一开始是使用 let 而非 var 进行的声明。这似乎违背了参数修改的规则，但实际上

却并非如此。这是类实例的一个特性，即实例本身是可变的。在 changeNameOfDog 中，我们实际上并没有修改参数本身。为了做到这一点，我们本应该用一个不同的 Dog 实例进行替换。但这并非我们所采取的做法，如果想要这么做，那就需要将 Dog 参数声明为 inout（同时需要用 var 来声明 d，并将其地址作为参数进行传递）。

从技术上来说，类是引用类型，而其他对象类型风格则是值类型。在将结构体的实例作为参数传递给函数时，实际上使用的是该结构体实例的一个独立的副本。不过，在将类实例作为参数传递给函数时，传递的则是类实例本身。第 4 章将会对此进行深入介绍。

## 2.8 函数中的函数

可以在任何地方声明函数，包括在函数体中声明。声明在函数体中的函数（也叫作局部函数）可以被相同作用域的后续代码所调用，不过在作用域之外则完全不可见。

对于那些旨在辅助其他函数的函数，这是个非常优雅的架构。如果只有函数 A 需要调用函数 B，那么函数 B 就可以放在函数 A 中。

如下示例来自于我所写的应用（仅保留了结构）：

```
func checkPair(p1:Piece, and p2:Piece) -> Path? {
    // ...
    func addPathIfValid(midpt1:Point, _ midpt2:Point) {
    // ...
    }
    for y in -1..._yct {
        addPathIfValid((pt1.x,y),(pt2.x,y))
    }
    for x in -1..._xct {
        addPathIfValid((x,pt1.y),(x,pt2.y))
    }
    // ...
}
```

第 1 个循环（for y）与第 2 个循环（for x）所做的事情是一样的，不过起始值集合是不同的。我们可以在每个 for 循环中编写整个功能，不过这么做没有必要，并且会导致重复（这种重复违背了 DRY 原则，即“不要重复自己”）。为了防止这种重复，我们可以将重复代码重构到实例方法中，然后由两个 for 循环调用，不过这么做会将该功能所公开的范围扩大，因为它只会由 checkPair 中的这两个 for 循环所调用。对于这种情况，局部函数就是很好的折中。

有时，即便函数只会在一个地方调用，使用局部函数也是值得的。如下示例也来自于我所编写的代码（它是同一个函数的另外一个部分）：

```
func checkPair(p1:Piece, and p2:Piece) -> Path? {
    // ...
    if arr.count > 0 {
        func distance(pt1:Point, _ pt2:Point) -> Double {
            // utility to learn physical distance between two points
            let deltax = pt1.0 - pt2.0
            let deltay = pt1.1 - pt2.1
            return sqrt(Double(deltax * deltax + deltay * deltay))
        }
        for thisPath in arr {
            var thisLength = 0.0
            for ix in 0..<(thisPath.count-1) {
                thisLength += distance(thisPath[ix],thisPath[ix+1])
            }
           // ...
        }
    }
    // ...
}
```

上述代码的结构很清晰（不过代码使用了尚未介绍的一些Swift特性）。进入函数checkPair中，我有一个路径的数组（arr），我需要知道每条路径的长度。每条路径本身都是点的一个数组，因此为了获取其长度，我需要计算出每两个点之间的距离总和。为了得到两个点之间的距离，我使用了勾股定理。我可以使用勾股定理，在for循环（for ix）中进行计算。不过，我将其作为单独的一个函数distance，这样在for循环中就可以调用该函数了。

这么做并没有减少代码的行数；事实上，声明distance还会增加行数！严格来说，我并没有重复自己；勾股定理会使用多次，不过代码中只出现了一次，位于for循环中。不过，将代码抽象为更加通用的距离计算功能使代码变得更加整洁了：实际上，我是先说明要做什么（要计算两个点之间的距离），然后再去做。函数名distance为代码赋予了含义；这么做相比于直接写出距离计算步骤来说，可理解性与可维护性都更好。

局部函数就是带有函数值的局部变量（本章后面将会介绍这个概念）。因此，局部函数不能与相同作用域中的局部变量同名，相同作用域中的两个局部函数也不能同名。

## 2.9 递归

函数可以调用自身，这叫作递归。递归似乎有些可怕，就像从悬崖上跳下来一样，因为

要冒着创建一个无限循环的风险；不过，如果函数编写正确，那么总是会有一个“停止”条件，它会处理降级情况，并防止无限循环的发生：

```
func countDownFrom(ix:Int) {
    print(ix)
    if ix > 0 { // stopper
        countDownFrom(ix-1) // recurse!
    }
}
```

在 Swift 2.0 之前，Swift 对递归施加了一个限制：函数中的函数（局部函数）不可以调用自身。在 Swift 2.0 中，这个限制已经解除了。

## 2.10 将函数作为值

如果从未使用过将函数当作一等公民的编程语言，那么你现在应该坐好了，因为我所说的话可能会让你昏倒：在 Swift 中，函数是一等公民。这意味着函数可以用在任何可以使用值的地方。比如，函数可以赋给变量；函数可以作为函数调用的参数；函数可以作为函数的结果返回。

Swift 有严格的类型。你只能将一个值赋给变量，或是将值传递给函数以及从函数传递出来，前提是它的类型是正确的。为了将函数当成值来使用，函数需要有一个类型。事实上，函数就是有类型的。能猜出是什么吗？函数的签名就是其类型。

将函数当作值的主要目的在于稍后可以在不知道函数是什么的情况下调用该函数。

下面是个简单的示例，只展示了语法与结构：

```
func doThis(f:()->()) {
    f()
}
```

`doThis` 函数接收一个参数，并且无返回值。参数 `f` 本身是个函数；因为参数类型不是 `Int`、`String` 或 `Dog`，而是一个函数签名 `()->()`，这表示一个不接收参数、无返回值的函数。接下来，`doThis` 函数会调用接收到的函数 `f` 作为其参数，这正是函数体中参数名后面圆括号的含义。

那么该如何调用函数 `doThis` 呢？你需要传递一个函数作为实参。一种方式是将函数名作为参数，如以下代码所示：

```
func whatToDo() {
    print("I did it")
}
doThis(whatToDo)
```

首先，我们声明了一个恰当类型的函数，该函数不接收参数且无返回值。接下来调用 doThis，将函数名作为实参传递给它。注意，这里并没有调用 whatToDo，只是将其传递进去。这是因为其名字后面并没有小括号。当然，这么做没问题：将 whatToDo 作为实参传递给 doThis；doThis 会将接收到的函数作为参数进行调用；然后控制台会打印出 "I did it"。

不过，这么做的意义何在？如果目标是调用 whatToDo，那为何不直接调用它呢？让其他函数调用它有什么好处呢？在刚才给出的示例中看不出来这么做的好处；我只是介绍一下语法与结构。不过在实际情况下，这么做是很有价值的，因为其他函数可以以特殊的方式来调用参数函数。比如，可以在完成其他一些事情后再调用它，或稍后再调用。

比如，将函数调用封装到函数中的一个原因是可以减少重复，降低出错的可能。如下示例来自于我所编写的代码。Cocoa 中常做的一件事就是在代码中直接绘图，这涉及 4 个步骤：

```
let size = CGSizeMake(45,20)
UIGraphicsBeginImageContextWithOptions(size, false, 0) ❶
let p = UIBezierPath(
    roundedRect: CGRectMake(0,0,45,20), cornerRadius: 8)
p.stroke() ❷
let result = UIGraphicsGetImageFromCurrentImageContext() ❸
UIGraphicsEndImageContext() ❹
```

❶ 打开图形上下文。

❷ 在上下文中绘制。

❸ 提取图像。

❹ 关闭图形上下文。

这么做丑陋至极。所有代码的唯一目的就是获取 result，即图像；不过这个目的散落到了其他代码中。同时，整个结构是样本式的；无论在哪个应用中，步骤 1、步骤 3 和步骤 4 都是一样的。此外，我很担心会忘记其中某一步；比如，如果不小心漏掉了步骤 4，那么结果就会非常糟糕。

每次绘制时唯一不同的就是步骤 2。因此，步骤 2 是唯一一个需要编写的部分！只要编写一个辅助函数来表示出这个样板化过程就能解决所有问题：

```
func imageOfSize(size:CGSize, whatToDraw:() -> ()) -> UIImage {
    UIGraphicsBeginImageContextWithOptions(size, false, 0)
```

```
    whatToDraw()
    let result = UIGraphicsGetImageFromCurrentImageContext()
    UIGraphicsEndImageContext()
    return result
}
```

imageOfSize 辅助函数很有用，因此我将其声明在文件顶层，这样所有文件就都能看到它了。为了绘制图像，我在函数中执行了步骤 2（实际的绘制），然后将该函数作为实参传递给 imageOfSize 辅助函数：

```
func drawing() {
    let p = UIBezierPath(
        roundedRect: CGRectMake(0,0,45,20), cornerRadius: 8)
    p.stroke()
}
let image = imageOfSize(CGSizeMake(45,20), drawing)
```

这是将绘制指令转换为图像的一种漂亮的表示方式。

Cocoa API 很多时候都需要传递函数，然后由运行时以某种方式或稍后调用。比如，当一个视图控制器展现视图时，你所调用的方法会接收 3 个参数：展现的视图控制器、表示展现是否要添加动画的 Bool 值，以及展现完毕后所调用的函数：

```
let vc = UIViewController()
func whatToDoLater() {
    print("I finished!")
}
self.presentViewController(vc, animated:true, completion:whatToDoLater)
```

Cocoa 文档常常将这样的函数描述为处理器，并将其称作块，因为这里需要的是 Objective-C 语法结构；在 Swift 中，它是个函数，因此将其当作函数并传递就可以了。

有些常见的 Cocoa 场景甚至会将两个函数传递给一个函数。比如，在执行视图动画时，你常常会传递两个函数，一个函数规定动画动作，另一个函数指定接下来要做的事情：

```
func whatToAnimate() { // self.myButton is a button in the interface
    self.myButton.frame.origin.y += 20
}
func whatToDoLater(finished:Bool) {
    print("finished: \(finished)")
}
UIView.animateWithDuration(
    0.4, animations: whatToAnimate, completion: whatToDoLater)
```

这表示：改变界面中按钮的帧原点（即位置），动作持续时间为 0.4 秒；接下来，当完成时，在控制台中打印出一条日志消息，说明动画执行是否完成。

为了让函数类型说明符更加清晰，请通过Swift的typealias特性创建一个类型别名，为函数类型赋予一个名字。这个名字可以是描述性的，请不要与箭头运算符符号搞混。比如，如果定义 typealias VoidVoidFunction = () -> ()，那就可以在通过该签名指定函数类型时使用 VoidVoidFunction 了。

# 2.11 匿名函数

再来看看之前的示例：

```
func whatToAnimate() { // self.myButton is a button in the interface
    self.myButton.frame.origin.y += 20
}
func whatToDoLater(finished:Bool) {
    print("finished: \(finished)")
}
UIView.animateWithDuration(
    0.4, animations: whatToAnimate, completion: whatToDoLater)
```

上述代码有些丑陋。我声明函数 whatToAnimate 与 whatToDoLater 的唯一目的就是将它们传递给最下面一行的函数。我其实并不需要 whatToAnimate 与 whatToDoLater 这两个名字，只不过就是为了在最后一行代码中引用它们而已；无论是名字还是这两个函数后面都不会再用到了。因此，要是不需要名字而只传递这两个函数的函数体就好了。

这叫作匿名函数，在 Swift 中是合法且常见的。为了构造匿名函数，你需要完成两件事：

1. 创建函数体本身，包括外面的花括号，但不需要函数声明。
2. 如果必要，将函数的参数列表与返回类型作为花括号中的第 1 行，后跟关键字 in。

下面将之前的具名函数声明转换为匿名函数。如下是 whatToAnimate 的具名函数声明：

```
func whatToAnimate() {
    self.myButton.frame.origin.y += 20
}
```

下面是完成相同事情的匿名函数。注意我是如何将参数列表与返回类型移到花括号中的：

```
{
    () -> () in
    self.myButton.frame.origin.y += 20
}
```

下面是 whatToDoLater 的具名函数声明：

```
func whatToDoLater(finished:Bool) {
```

```
    print("finished: \(finished)")
}
```

下面是完成相同事情的匿名函数：

```
{
    (finished:Bool) -> () in
    print("finished: \(finished)")
}
```

现在我们既然已经知道了如何创建匿名函数，下面就来使用它们。在向 animateWithDuration 传递参数时需要函数。我们可以在这个地方创建并传递匿名函数，如以下代码所示：

```
UIView.animateWithDuration(0.4, animations: {
    () -> () in
    self.myButton.frame.origin.y += 20
    }, completion: {
        (finished:Bool) -> () in
        print("finished: \(finished)")
})
```

我们可以像 2.10 节调用 imageOfSize 函数那样做出相同的改进。之前是这样调用函数的：

```
func drawing() {
    let p = UIBezierPath(
        roundedRect: CGRectMake(0,0,45,20), cornerRadius: 8)
    p.stroke()
}
let image = imageOfSize(CGSizeMake(45,20), drawing)
```

不过，现在知道并不需要单独声明 drawing 函数。我们可以通过匿名函数来调用 imageOfSize：

```
let image = imageOfSize(CGSizeMake(45,20), {
    let p = UIBezierPath(
        roundedRect: CGRectMake(0,0,45,20), cornerRadius: 8)
    p.stroke()
})
```

匿名函数在 Swift 中使用非常普遍，因此请确保你能够读懂并编写这样的代码！事实上，匿名函数非常常见且非常重要，因此 Swift 提供了以下一些便捷写法。

*省略返回类型*

如果编译器知道匿名函数的返回类型，那么你就可以省略箭头运算符及返回类型说明：

```
UIView.animateWithDuration(0.4, animations: {
    () in
    self.myButton.frame.origin.y += 20
```

```
    }, completion: {
        (finished:Bool) in
        print("finished: \(finished)")
})
```

*如果没有参数，那么可以省略 in 这一行*

如果匿名函数不接收参数，并且返回类型可以省略，那么 in 这一行就可以被完全省略：

```
UIView.animateWithDuration(0.4, animations: {
    self.myButton.frame.origin.y += 20
    }, completion: {
        (finished:Bool) in
        print("finished: \(finished)")
})
```

*省略参数类型*

如果匿名函数接收参数，并且编译器知道其类型，那么类型就是可以省略的：

```
UIView.animateWithDuration(0.4, animations: {
    self.myButton.frame.origin.y += 20
    }, completion: {
        (finished) in
        print("finished: \(finished)")
})
```

*省略圆括号*

如果省略参数类型，那么包围参数列表的圆括号也可以省略：

```
UIView.animateWithDuration(0.4, animations: {
    self.myButton.frame.origin.y += 20
    }, completion: {
        finished in
        print("finished: \(finished)")
})
```

*有参数时也可以省略 in 这一行*

如果返回类型可以省略，并且编译器知道参数类型，那就可以省略 in 这一行，直接在匿名函数体中引用参数，方式是使用魔法名 $0、$1 等，并且要按照顺序引用：

```
UIView.animateWithDuration(0.4, animations: {
    self.myButton.frame.origin.y += 20
    }, completion: {
        print("finished: \($0)")
})
```

*省略参数名*

如果匿名函数体不需要引用某个参数，那就可以在 in 这一行通过下划线来代替参数列表中该参数的名字；事实上，如果匿名函数体不需要引用任何参数，那就可以通过一个下划线来代替整个参数列表：

```
UIView.animateWithDuration(0.4, animations: {
    self.myButton.frame.origin.y += 20
    }, completion: {
        _ in
        print("finished!")
})
```

不过请注意，如果匿名函数接收参数，那就必须要以某种方式承认它们的存在。可以省略 in 这一行，然后通过魔法名 $0 等来使用参数，或是保留 in 这一行，然后通过下划线省略参数，但不能在省略 in 这一行的同时又不通过魔法名来使用参数！如果这么做了，那么代码将无法编译通过。

*省略函数实参标签*

如果匿名函数是函数调用的最后一个参数，那么你可以在最后一个参数前通过右圆括号关闭函数调用，然后放置匿名函数体且不带任何标签（这叫作尾函数）：

```
UIView.animateWithDuration(0.4, animations: {
    self.myButton.frame.origin.y += 20
    }) {
        _ in
        print("finished!")
}
```

*省略调用函数圆括号*

如果使用尾函数语法，并且调用的函数只接收传递给它的函数，那就可以在调用中省略空的圆括号。这是唯一一个可以从函数调用中省略圆括号的情形。下面声明并调用一个不同的函数：

```
func doThis(f:()->()) {
    f()
}
doThis { // no parentheses!
    print("Howdy")
}
```

*省略关键字* return

如果匿名函数体只包含一条语句，并且该语句使用关键字 return 返回一个值，那么关键字 return 就可以省略。换句话说，在函数会返回一个值的上下文中，如果匿名函数体只包含了一条语句，那么 Swift 就会假设该语句是个表达式，其值会从匿名函数中返回：

```
func sayHowdy() -> String {
    return "Howdy"
}
func performAndPrint(f:()->String) {
    let s = f()
    print(s)
```

```
}
performAndPrint {
    sayHowdy() // meaning: return sayHowdy()
}
```

在编写匿名函数时，你可以充分利用上面介绍的各种省略形式。此外，你还可以将整个匿名函数作为一行放到函数调用中，从而减少代码占据的行数（但不会减少代码量）。这样，涉及匿名函数的 Swift 代码就会变得非常紧凑了。

下面是个典型的示例。首先定义了一个 Int 值的数组，然后生成一个新数组，新数组中的每个值都是原数组值乘以 2，方式是调用 map 实例方法。数组的 map 方法接收一个函数，该函数接收一个参数，并返回一个与数组元素相同类型的值；这里的数组由 Int 值构成，因此需要向 map 方法传递一个接收一个 Int 值并返回一个 Int 值的函数。整个函数的代码如下所示：

```
let arr = [2, 4, 6, 8]
func doubleMe(i:Int) -> Int {
    return i*2
}
let arr2 = arr.map(doubleMe) // [4, 8, 12, 16]
```

不过，这么写不太符合 Swift 的风格。其他地方并不需要 doubleMe 这个名字，因此它可以作为一个匿名函数。其返回类型是已知的，因此无须指定；其参数类型是已知的，因此也无须指定。我们只需要使用一个参数，因此并不需要 in 这一行，只要用 $0 来引用该参数即可。函数体只包含了一条语句，它是个 return 语句，因此可以省略 return。map 不再接收其他参数，因此可以省略圆括号，在名字后直接跟着尾函数即可：

```
let arr = [2, 4, 6, 8]
let arr2 = arr.map {$0*2}
```

## 2.12 定义与调用

Swift 中非常常见的一种模式就是定义一个匿名函数然后调用它，如以下代码所示：

```
{
    // ... code goes here
}()
```

注意花括号后面的圆括号。花括号定义了一个匿名函数体；圆括号则调用了这个匿名函数。

为什么会这么做呢？如果想要运行一些代码，直接运行就行了；为什么还要将其嵌入更深的层次作为函数体，反过来再运行它呢？

首先，匿名函数是降低代码的命令性，增强函数性的一种行之有效的方式：动作在需要时才发生，而无须借助一系列的准备步骤。如下是个常见的 Cocoa 示例：创建并配置一个 NSMutableParagraphStyle，然后在对 addAttribute:value:range: 的调用中使用（content 是个 NSMutableAttributedString）。

```
let para = NSMutableParagraphStyle()
para.headIndent = 10
para.firstLineHeadIndent = 10
// ... more configuration of para ...
content.addAttribute(
    NSParagraphStyleAttributeName,
    value:para, range:NSMakeRange(0,1))
```

我觉得上面的代码丑陋至极。我们只在 addAttribute:value:range 调用中才需要将 para 作为 value: 实参传递进去，因此在调用中创建并配置它才是更好的做法。Swift 允许我们这么做，我更倾向于下面这种写法：

```
content.addAttribute(
    NSParagraphStyleAttributeName,
    value: {
        let para = NSMutableParagraphStyle()
        para.headIndent = 10
        para.firstLineHeadIndent = 10
        // ... more configuration of para ...
        return para
    }(),
    range:NSMakeRange(0,1))
```

第 3 章将会进一步介绍定义与调用的使用场景。

## 2.13 闭包

Swift 函数就是闭包。这意味着它们可以在函数体作用域中捕获对外部变量的引用。这是什么意思呢？回忆一下第 1 章，花括号中的代码构成了一个作用域，这些代码能够看到外部作用域中声明的变量与函数：

```
class Dog {
    var whatThisDogSays = "woof"
    func bark() {
        print(self.whatThisDogSays)
    }
}
```

在上述代码中，bark 函数体引用了变量 whatThisDogSays，该变量是函数体的外部变量，因为它声明在函数体外部。它位于函数体的作用域中，因为函数体内部的代码可以看到它。函数体内部的代码能够引用 whatThisDogSays。

一切都很好；不过，我们现在知道函数 bark 可以当作值来传递。实际上，它可以从一个环境传递到另一个环境中！如果这样做，那么对 whatThisDogSays 的引用会发生什么情况呢？下面就来看看：

```
func doThis(f : Void -> Void) {
    f()
}
let d = Dog()
d.whatThisDogSays = "arf"
let f = d.bark
doThis(f) // arf
```

运行上述代码，控制台会打印出 "arf"。

也许结果不会让你感到惊讶，不过请思考一下。我们并未直接调用 bark。我们创建了一个 Dog 实例，然后将其 bark 函数作为值传递给函数 doThis，然后被调用。现在，whatThisDogSays 是某个 Dog 实例的一个实例属性。在函数 doThis 中并没有 whatThisDogSays。实际上，在函数 doThis 中并没有 Dog 实例！不过，调用 f() 依然可以使用。函数 d.bark 还是可以看到变量 whatThisDogSays（声明在外部），虽然它的调用环境中并没有任何 Dog 实例，也没有任何实例属性 whatThisDogSays。

bark 函数在传递时会持有其所在的环境，甚至在传递到另一个全新环境中再调用时亦如此。对于“捕获”，我的意思是当函数作为值被传递时，它会持有对外部变量的内部引用。这使得函数成为一个闭包。

你可能利用了函数就是闭包这一特性，但却根本就没有注意到过。回忆一下之前的示例，在界面上以动画的形式移动按钮的位置：

```
UIView.animateWithDuration(0.4, animations: {
    self.myButton.frame.origin.y += 20
    }) {
        _ in
        print("finished!")
}
```

上述代码看起来很简单，但请注意第 2 行，匿名函数作为实参被传递给了 animations: 参数。真是这样的吗？这与 Cocoa 相差甚远，这个匿名函数会在未来的某个时间被调用来启动动画，Cocoa 会找到 myButton，这个对象是 self 的一个属性，代码中早就是这样的了？是的，Cocoa 可以做到这一点，因为函数就是个闭包。对该属性的引用会被捕获并由匿名函数维护；这样，当匿名函数真正被调用时，它就会执行，按钮也会移动。

## 2.13.1 闭包是如何改善代码的

如果理解了函数就是闭包这一理念，那么你就可以利用这一点来改善代码的语法了。闭

包会让代码变得更加通用，实用性也更强。下面这个函数是之前提及的一个示例，它接收绘制指令，然后执行来生成一张图片：

```
func imageOfSize(size:CGSize, _ whatToDraw:() -> ()) -> UIImage {
    UIGraphitsBeginImageContextWithOptions(size, false, 0)
    whatToDraw()
    let result = UIGraphicsGetImageFromCurrentImageContext()
    UIGraphicsEndImageContext()
    return result
}
```

我们可以通过一个尾匿名函数来调用 imageOfSize：

```
let image = imageOfSize(CGSizeMake(45,20)) {
    let p = UIBezierPath(
        roundedRect: CGRectMake(0,0,45,20), cornerRadius: 8)
    p.stroke()
}
```

不过，上述代码还有一个讨厌的重复情况。这是个会根据给定大小（包含该尺寸的圆角矩形）创建图片的调用。我们重复了这个尺寸；数值对（45,20）出现了两次，这么做可不好。下面将尺寸放到起始位置处的变量中来避免重复。

```
let sz = CGSizeMake(45,20)
let image = imageOfSize(sz) {
    let p = UIBezierPath(
        roundedRect: CGRect(origin:CGPointZero, size:sz), cornerRadius: 8)
    p.stroke()
}
```

内部可以看到声明在更高层的匿名函数中的变量 sz。这样，我们就可以在匿名函数中引用它了，我们也是这么做的。匿名函数就是个函数，因此也是闭包。匿名函数会捕获引用，将其放到对 imageOfSize 的调用中。当 imageOfSize 调用 whatToDraw，而 whatToDraw 引用了变量 sz 时，这么做是没问题的，即便在 imageOfSize 中并没有 sz 也可以。

下面更进一步。到目前为止，我们硬编码了所需圆角矩形的大小。不过，假设创建各种大小的圆角矩形图片是经常性的事情，那么将代码放到函数中就是更好的做法，其中 sz 不是固定值，而是一个参数；接下来，函数会返回创建好的图片：

```
func makeRoundedRectangle(sz:CGSize) -> UIImage {
    let image = imageOfSize(sz) {
        let p = UIBezierPath(
            roundedRect: CGRect(origin:CGPointZero, size:sz),
            cornerRadius: 8)
        p.stroke()
    }
    return image
}
```

代码依然可以正常使用。匿名函数中的 sz 会引用传递给外围函数 makeRounded-Rectangle 的 sz 参数。外围函数的参数对于外部以及匿名函数都是可见的。匿名函数是个闭包，因此在传递给 imageOfSize 时它会捕获对该参数的引用。

代码现在变得很紧凑了。为了调用 makeRoundedRectangle，提供一个尺寸即可；创建好的图片就会返回。这样，就可以执行调用，获取图片，然后将图片放到界面上，所有这些只需一步即可实现，如以下代码所示：

```
self.myImageView.image = makeRoundedRectangle(CGSizeMake(45,20)).
```

## 2.13.2 返回函数的函数

现在再进一步！相对于返回一张图片，函数可以返回一个函数，这个函数可以创建出指定大小的圆角矩形。如果从来没有见过一个函数可以以值的形式从另一个函数中返回，那么现在就是见证奇迹的时刻了。毕竟，函数可以当作值。在函数调用中，我们已经将函数作为实参传递给另一个函数了，现在来从一个函数中接收一个函数作为其结果：

```
func makeRoundedRectangleMaker(sz:CGSize) -> () -> UIImage { ❶
    func f () -> UIImage { ❷
        let im = imageOfSize(sz) {
            let p = UIBezierPath(
                roundedRect: CGRect(origin:CGPointZero, size:sz),
                cornerRadius: 8)
            p.stroke()
        }
        return im
    }
    return f ❸
}
```

下面来分析一下上述代码：

❶ 声明是最难理解的部分。函数 makeRoundedRectangleMaker 的类型（签名）到底是什么呢？它是 (CGSize) -> () -> UIImage。该表达式有两个箭头运算符。为了理解这一点，请记住每个箭头运算符后面的内容就是返回值的类型。因此，makeRoundedRectangleMaker 是个函数，它接收 CGSize 参数并返回 a() -> UIImage。那 () -> UIImage 又是什么意思呢？我们其实已经知道了：它是个函数，不接收参数，并且返回一个 UIImage。这样，makeRoundedRectangleMaker 就是个函数，接收一个 CGSize 参数并返回一个函数，如果不传递参数，那么该函数本身就会返回一个 UIImage。

❷ 现在来看 makeRoundedRectangleMaker 函数体，首先声明一个函数（函数中的函数或是局部函数），其类型是我们期望返回的，即它不接收参数并返回一个 UIImage。

我们将该函数命名为 f，该函数的工作方式非常简单：调用 imageOfSize，传递一个匿名函数（创建一个圆角矩形图片 im），然后将图片返回。

❸ 最后，返回创建的函数（f）。我们已经实现了契约：返回一个函数，它不接收参数并返回一个 UIImage。

也许你还对 makeRoundedRectangleMaker 感到好奇，想知道该如何调用它，以及调用之后会得到什么结果。下面就来试一下：

```
let maker = makeRoundedRectangleMaker(CGSizeMake(45,20))
```

代码运行后变量 maker 值是什么呢？它是个函数，不接收参数，当调用后会生成一张大小为（45,20）的圆角矩形图片。不相信？那我就来证明一下，调用这个函数（它现在是 maker 的值）：

```
let maker = makeRoundedRectangleMaker(CGSizeMake(45,20))
self.myImageView.image = maker()
```

现在应该理解函数可以将函数作为结果了，下面来看看 makeRoundedRectangleMaker 的实现，再次对其进行分析，这次是以不同的方式。记住，我并没有向你演示函数可以产生函数，我这么写只是为了说明闭包！来看看环境是如何被捕获的：

```
func makeRoundedRectangleMaker(sz:CGSize) -> () -> UIImage {
    func f () -> UIImage {
        let im = imageOfSize(sz) { // *
            let p = UIBezierPath(
                roundedRect: CGRect(origin:CGPointZero, size:sz), // *
                cornerRadius: 8)
            p.stroke()
        }
        return im
    }
    return f
}
```

函数 f 不接收参数。不过，在 f 的函数体中（参见 * 注释）引用了尺寸值 sz 两次。f 的函数体可以看到 sz，它是外围函数 makeRoundedRectangleMaker 的参数，因为 sz 位于外围作用域中。函数 f 在 makeRoundedRectangleMaker 调用时捕获对 sz 的引用，并且在将 f 返回并赋给 maker 时保持该引用：

```
let maker = makeRoundedRectangleMaker(CGSizeMake(45,20))
```

这正是 maker 现在是一个函数的原因所在，当调用时，它会创建并返回一个尺寸为（45,20）的图片，虽然它本身被调用时并没有任何参数。我们已经将所要生成的图片尺寸传递给了 maker。

从另一个角度再来看看，makeRoundedRectangleMaker是一个工厂，用于创建类似于maker的一系列函数，其中每个函数都会生成特定尺寸的一张图片。这是对闭包功能的最好说明。

继续之前，我准备以更加Swift的风格重写该函数。在函数f中，我们无须创建im再将其返回；可以直接返回调用imageOfSize的结果：

```
func makeRoundedRectangleMaker(sz:CGSize) -> () -> UIImage {
    func f () -> UIImage {
        return imageOfSize(sz) {
            let p = UIBezierPath(
                roundedRect: CGRect(origin:CGPointZero, size:sz),
                cornerRadius: 8)
            p.stroke()
        }
    }
    return f
}
```

不过没必要声明f再将其返回；可以将其定义为匿名函数，然后直接返回：

```
func makeRoundedRectangleMaker(sz:CGSize) -> () -> UIImage {
    return {
        return imageOfSize(sz) {
            let p = UIBezierPath(
                roundedRect: CGRect(origin:CGPointZero, size:sz),
                cornerRadius: 8)
            p.stroke()
        }
    }
}
```

不过，匿名函数只包含了一条语句，返回imageOfSize的调用结果。（imageOfSize的匿名函数参数有很多行，不过imageOfSize调用本身只有一行Swift语句。）因此，没必要使用return：

```
func makeRoundedRectangleMaker(sz:CGSize) -> () -> UIImage {
    return {
        imageOfSize(sz) {
            let p = UIBezierPath(
                roundedRect: CGRect(origin:CGPointZero, size:sz),
                cornerRadius: 8)
            p.stroke()
        }
    }
}
```

## 2.13.3 使用闭包设置捕获变量

闭包可以捕获其环境的能力甚至要超过之前所介绍的。如果闭包捕获了对外部变量的引用，并且该变量的值是可以修改的，那么闭包就可以设置该变量。

比如，我声明了下面这个简单的函数。它所做的是接收一个函数，该函数会接收一个 Int 参数，然后通过实参 100 调用该函数：

```
func pass100 (f:(Int)->()) {
    f(100)
}
```

仔细看看如下代码，猜猜运行结果会是什么：

```
var x = 0
print(x)
func setX(newX:Int) {
    x = newX
}
pass100(setX)
print(x)
```

第 1 个 `print(x)` 显然会打印出 0。第 2 个 `print(x)` 调用会打印出 100！ `pass100` 函数进入我的代码，并修改变量 x 的值！这是因为传递给 `pass100` 的函数包含了对 x 的引用；它不仅包含了对其引用，还能够捕获它；而且不仅能捕获它，还会设置 x，就像直接调用 `setX` 一样。

## 2.13.4 使用闭包保存捕获的环境

当闭包捕获其环境后，即便什么都不做，它也可以保存该环境。如下示例可能会颠覆你的三观——这是一个可以修改函数的函数。

```
func countAdder(f:()->()) -> () -> () {
    var ct = 0
    return {
        ct = ct + 1
        print("count is \(ct)")
        f()
    }
}
```

函数 `countAdder` 接收一个函数作为参数，结果也返回一个函数。它所返回的函数会调用它所接收的函数；此外，它会增加变量值，然后打印出结果。现在猜猜如下代码运行后的结果会是什么：

```
func greet () {
    print("howdy")
```

```
}
let countedGreet = countAdder(greet)
countedGreet()
countedGreet()
countedGreet()
```

上述代码首先定义函数greet，它会打印出"howdy"，然后将其传递给函数countAdder。countAdder返回的是一个新函数，我们将其命名为countedGreet。接下来调用countedGreet 3次。下面是控制台的输出：

```
count is 1
howdy
count is 2
howdy
count is 3
howdy
```

显然，countAdder向传递给它的函数增加了调用次数的功能。现在来想想：维护这个数量的变量到底是什么呢？在countAdder内部，它是个局部变量ct；不过，它并未声明在countAdder所返回的匿名函数中。这么做是故意的！如果声明在匿名函数中，那么每次调用countedGreet时都会将ct设置为0，这就达不到计数的目的了。相反，ct只会被初始化为0一次，然后会由匿名函数所捕获。这样，该变量就会被保存为countedGreet环境的一部分了。在某些奇怪的保留环境的情况下，它会位于countedGreet外面，这样每次调用countedGreet时，它的值都会增加。这正是闭包的强大之处。

这个示例（可以保存环境状态）还有助于说明函数是引用类型的。为了证明这一点，我先来个对比示例。对一个函数工厂方法的两次单独调用会生成两个函数，正如你期望的那样：

```
let countedGreet = countAdder(greet)
let countedGreet2 = countAdder(greet)
countedGreet() // count is 1
countedGreet2() // count is 1
```

在上述代码中，两个函数countedGreet与countedGreet2会分别维护各自的数量。仅仅是赋值或是参数传递就会生成对相同函数的新引用，下面就来证明这一点：

```
let countedGreet = countAdder(greet)
let countedGreet2 = countedGreet
countedGreet() // count is 1
countedGreet2() // count is 2
```

# 2.14 柯里化函数

再次回到 makeRoundedRectangleMaker：

```
func makeRoundedRectangleMaker(sz:CGSize) -> () -> UIImage {
    return {
        imageOfSize(sz) {
            let p = UIBezierPath(
                roundedRect: CGRect(origin:CGPointZero, size:sz),
                cornerRadius: 8)
            p.stroke()
        }
    }
}
```

我对上述方法的一个地方不太满意：它所创建的圆角矩形的尺寸是个参数（sz），不过圆角矩形的 cornerRadius 却是硬编码的 8，我希望能够为圆角半径指定值。有两种方式可以做到这一点。一种是为 makeRoundedRectangleMaker 本身再提供一个参数：

```
func makeRoundedRectangleMaker(sz:CGSize, _ r:CGFloat) -> () -> UIImage {
    return {
        imageOfSize(sz) {
            let p = UIBezierPath(
                roundedRect: CGRect(origin:CGPointZero, size:sz),
                cornerRadius: r)
            p.stroke()
        }
    }
}
```

然后像下面这样调用：

```
let maker = makeRoundedRectangleMaker(CGSizeMake(45,20), 8)
```

还有另外一种方式。现在，makeRoundedRectangleMaker 所返回的函数不接收参数，我们可以让它接收一个参数：

```
func makeRoundedRectangleMaker(sz:CGSize) -> (CGFloat) -> UIImage {
    return {
        r in
        imageOfSize(sz) {
            let p = UIBezierPath(
                roundedRect: CGRect(origin:CGPointZero, size:sz),
                cornerRadius: r)
            p.stroke()
        }
    }
}
```

现在，makeRoundedRectangleMaker 所返回的函数会接收一个参数，因此在调用时需要将这个参数提供给它：

```
let maker = makeRoundedRectangleMaker(CGSizeMake(45,20))
self.myImageView.image = maker(8)
```

如果不需要保存 maker 供其他地方使用，那就可以在一行完成所有这些事情：函数调用会生成一个函数，我们再立刻调用该函数来获取图片：

```
self.myImageView.image = makeRoundedRectangleMaker(CGSizeMake(45,20))(8)
```

如果函数返回的函数接收一个参数，就像该示例这样，那么它就叫作柯里化函数（为了纪念计算机科学家 Haskell Curry）。Swift 提供了柯里化函数的便捷声明方式，可以省略第 1 个箭头运算符与顶层匿名函数，如以下代码所示：

```
func makeRoundedRectangleMaker(sz:CGSize)(_ r:CGFloat) -> UIImage {
   return imageOfSize(sz) {
       let p = UIBezierPath(
           roundedRect: CGRect(origin:CGPointZero, size:sz),
           cornerRadius: r)
       p.stroke()
   }
}
```

表达式 (sz:CGSize)(_ r:CGFloat)（一行有两个参数列表，并且中间没有箭头运算符）表示“Swift，请对该函数进行柯里化”。Swift 会将函数划分到两个函数中，一个是 makeRoundedRectangleMaker，接收 CGSize 参数，另一个是匿名函数，接收 CGFloat。代码看起来好像是 makeRoundedRectangleMaker 会返回一个 UIImage，不过实际上它返回的是一个函数，该函数会返回一个 UIImage，就像之前那样。我们可以像之前所采用的两种方式那样调用它。

第 3 章

# 变量与简单类型

本章将会深入介绍变量的声明与初始化，同时还会介绍所有主要的 Swift 内建简单类型（这里的“简单”是相对于集合来说的；第 4 章最后将会介绍主要的内建集合类型）。

## 3.1 变量作用域与生命周期

回忆一下第 1 章所讲的，变量就是个类型明确的具名盒子。每个变量都必须要显式声明。为了将对象放到盒子中，即让变量名引用该对象，你需要将对象赋给变量（第 2 章介绍过，函数也有类型，也可以赋给变量）。

除了给引用赋予一个名字，根据所声明的位置，变量还会对所引用的对象赋予一个特定的作用域（可见性）与生命周期；将某个对象赋给变量可以确保它能被所需的代码看到，并且持续足够长的时间来满足这个目的。

在 Swift 文件的结构中（参见示例 1-1），变量实际上可以在任何地方声明。不过，区分变量作用域与生命周期的几个层次还是非常有必要的：

*全局变量*

全局变量指的是声明在 Swift 文件顶层的变量（在示例 1-1 中，变量 one 就是个全局变量）。

全局变量的生命周期与文件一样长。这意味着它会一直存在。不过，说一直存在有点不太严格，但只要程序运行时它就会存在。

全局变量在任何地方都是可见的，这正是“全局”一词的含义。相同文件中的所有代码都可以看到它；因为它位于顶层，因此相同文件中的任何其他代码都会位于顶

层或是更低的层次，这都是作用域所包含的层次。此外，在默认情况下，相同模块中的任何其他文件中的代码也可以看到它，因为同一个模块中的 Swift 文件会自动看到彼此，因此也会看到彼此的顶层内容。

*属性*

属性指的是声明在对象类型声明（枚举、结构体或类；在示例 1-1 中，3 个 `name` 变量就是属性）顶层的变量。有两种类型的属性：实例属性与静态 / 类属性。

*实例属性*

在默认情况下，属性就是实例属性。其值对于该对象类型的每个实例来说都是不同的；其生命周期与实例的生命周期相同。回忆一下第 1 章，实例创建后（通过实例化）就存在了；实例随后的生命周期取决于该实例所赋予的变量的生命周期。

*静态 / 类属性*

通过关键字 `static` 或 `class` 声明的属性就是静态 / 类属性（第 4 章将会对其进行详细介绍）。其生命周期与对象类型的生命周期相同。如果对象类型声明在文件顶层，或是声明在另一个对象类型的顶层，而该对象类型又声明在顶层，那么这就意味着它会一直存在（只要程序运行就会存在）。

属性对于对象声明中的所有代码都是可见的。比如，对象的方法可以看到该对象的属性。代码可以通过 `self` 加上点符号来引用属性，我总是这样做，不过除了一些可能会产生歧义的场景， 通常可以省略掉。

在默认情况下，实例属性对于其他代码也是可见的，前提是其他代码持有该实例的引用；在这种情况下，可以通过实例引用与点符号来引用属性。在默认情况下，静态 / 类属性对于其他代码也是可见的，只要其他代码能够看到该对象类型的名字即可；在这种情况下，可以通过对象类型与点符号来引用属性。

*局部变量*

局部变量指的是声明在函数体中的变量（在示例 1-1 中，变量 `two` 就是个局部变量）。局部变量的生命周期取决于外围花括号的生命周期：当执行路径进入作用域中并到达变量声明处时，局部变量就产生了；当执行路径退出作用域时，局部变量就会消亡。局部变量有时也叫作自动变量，表示它们会自动产生和消亡。

局部变量只能被相同作用域的后续代码看到（包括相同作用域中后续更深层次的代码）。

## 3.2 变量声明

正如第 1 章所述，变量是通过 `let` 或 `var` 声明的：

- let 声明的变量是常量，其值在首次赋值（初始化）后就不会再变化了。
- var 声明的变量才是真正的变量，其值可以被后续的赋值所改变。

不过，变量的类型是绝对不会改变的。使用 var 声明的变量可以被赋予不同的值，但该值必须要符合变量的类型。这样，在声明变量时，需要为其指定好类型，然后变量就会一直具有该类型。你可以显式或隐式地指定变量的类型：

*显式变量类型声明*

在声明中的变量名后面，添加一个冒号和类型名：

```
var x : Int
```

*通过初始化创建隐式变量类型*

如果将变量初始化作为声明的一部分，并且没有提供显式的类型，那么 Swift 就会根据初始化值推断其类型：

```
var x = 1 // and now x is an Int
```

完全可以显式声明变量的类型，然后为其赋予一个初始值，并将这些工作在一步中完成：

```
var x : Int = 1
```

在该示例中，显式类型声明是多余的，因为类型（Int）可以通过初始值推断出来。但有时，提供显式类型的同时又赋予一个初始值并不多余。比如，下面列出的各种情况：

*Swift 的推断可能是错误的*

我遇到的一个常见情况就是当我提供初始值作为数值字面值时，Swift 会将其推断为 Int 或 Double，这取决于字面值是否包含了小数点。不过还有很多其他的数值类型！如果遇到这种情况，我会显式提供类型，如下代码所示：

```
let separator : CGFloat = 2.0
```

*Swift 无法推断出类型*

在这种情况下，显式变量类型可以让 Swift 推断出初始值的类型。选项集合就是一种非常常见的情况（第 4 章将会介绍）。如下代码无法编译通过：

```
var opts = [.Autoreverse, .Repeat] // compile error
```

问题在于名字 .Autoreverse 与 .Repeat 分别是 UIViewAnimationOptions.Autoreverse 与 UIViewAnimationOptions.Repeat 的简写，不过除非我们告诉 Swift，否则它是不知道这一点的：

```
let opts : UIViewAnimationOptions = [.Autoreverse, .Repeat]
```

*程序员无法推断出类型*

我常常会加上多余的显式类型声明来提醒我自己。如下示例来自于我所编写的代码：

```
let duration : CMTime = track.timeRange.duration
```

在上述代码中，track 的类型是 AVAssetTrack。Swift 非常清楚 AVAssetTrack 的 timeRange 属性的 duration 属性是个 CMTime。不过，我不知道！为了提醒自己，我显式添加了类型。

由于可以使用显式变量类型，因此变量在声明时无需初始化。下面这样写是合法的：

```
let x : Int
```

现在，x 是个空盒子，一个没有初始值的 Int 变量。不过，如果可以避免，我强烈建议你不要对局部变量采取这种做法。这么做并非灾难，因为 Swift 编译器会阻止你使用从未赋过值的变量，只不过不是一个好习惯而已。

能够证明该规则的一个例外情况是条件初始化。有时，我们只有在执行了某些条件测试后才知道某个变量的初始值是什么。不过，变量本身只能声明一次，因此它必须要提前声明，然后根据条件进行初始化。下面这么做是合理的（不过还有更好的写法）：

```
let timed : Bool
if val == 1 {
    timed = true
} else {
    timed = false
}
```

在将变量的地址作为实参传递给函数时，变量必须要提前声明并初始化，即便初始值是假的亦如此。回忆一下第 2 章的示例：

```
var arrow = CGRectZero
var body = CGRectZero
CGRectDivide(rect, &arrow, &body, Arrow.ARHEIGHT, .MinYEdge)
```

代码运行后，两个 CGRectZero 值将被替换掉；它们仅仅是占位符而已，为了满足编译器的要求。

有时，你希望调用的 Cocoa 方法会立刻返回一个值，然后在传递给相同方法的函数中使用该值。比如，Cocoa 有一个 UIApplication 实例方法，其声明如下所示：

```
func beginBackgroundTaskWithExpirationHandler(handler: (() -> Void)?)
    -> UIBackgroundTaskIdentifier
```

该函数会返回一个数字（UIBackgroundTaskIdentifier 就是个 Int），然后再调用传递给

它的函数（handler），该函数会使用一开始返回的数字。Swift 的安全规则不允许你使用一行代码声明持有该数字的变量，然后在匿名函数中使用它：

```
let bti = UIApplication.sharedApplication()
    .beginBackgroundTaskWithExpirationHandler({
        UIApplication.sharedApplication().endBackgroundTask(bti)
    }) // error: variable used within its own initial value
```

因此，你需要提前声明好变量；不过，Swift 还会提示另一个错误：

```
var bti : UIBackgroundTaskIdentifier
bti = UIApplication.sharedApplication()
    .beginBackgroundTaskWithExpirationHandler({
        UIApplication.sharedApplication().endBackgroundTask(bti)
    }) // error: variable captured by a closure before being initialized
```

解决办法就是提前声明好变量，然后为其赋予一个假的初始值作为占位符：

```
var bti : UIBackgroundTaskIdentifier = 0
bti = UIApplication.sharedApplication()
    .beginBackgroundTaskWithExpirationHandler({
        UIApplication.sharedApplication().endBackgroundTask(bti)
    })
```

对象的实例属性（在枚举、结构体或类声明的顶层）可以在对象的初始化器函数中进行初始化，而不必在声明中赋值。对于常量实例属性（let）与变量实例属性（var），具有显式类型而不直接赋予初始值是合法且常见的。第 4 章将会深入介绍这一点。

## 3.3 计算初始化器

有时，你希望通过运行几行代码来计算出变量的初始值。完成这件事简单且紧凑的方式就是使用匿名函数，然后立刻调用（参见 2.12 节）。下面就来改写之前的示例进行说明：

```
let timed : Bool = {
    if val == 1 {
        return true
    } else {
        return false
    }
}()
```

在初始化实例属性时也可以这么做。在这个类中有一个图片（UIImage），后面将会用到多次。合理的方式是提前创建好该图片，并将其作为类的常量实例属性。创建图片意味着要绘制它，这需要几行代码才能实现。因此，我通过定义和调用一个匿名函数来声明并初始化该属性，如下代码所示（请参见第 2 章了解 imageOfSize 这个辅助函数）：

```
class RootViewController : UITableViewController {
    let cellBackgroundImage : UIImage = {
         return imageOfSize(CGSizeMake(320,44)) {
             // ... drawing goes here ...
         }
    }()
}
```

事实上，定义与调用匿名函数常常是通过多行代码来计算出实例属性初始值的唯一合法方式。原因在于，当初始化实例属性时是无法调用实例方法的，因为这个时候实例还不存在；毕竟，实例正在创建过程中。

# 3.4 计算变量

到目前为止，本章所介绍的变量都是存储下来的变量，就像盒子一样。变量是个名字，就像盒子一样；值可以通过赋给变量放到盒子中，然后等待通过引用该变量进行获取，只要变量存在就行。

此外，变量还可以计算出来。这意味着变量不再持有值，而是持有函数。在给变量赋值时，函数 *setter* 会被调用。当引用变量时，另一个函数 *getter* 会被调用。如下代码演示了声明计算变量的语法：

```
var now : String { ❶
    get { ❷
        return NSDate().description ❸
    }
    set { ❹
        print(newValue) ❺
    }
}
```

❶ 变量必须是个 var（不能是 let）。其类型必须要显式声明，后跟一对花括号。

❷ getter 函数叫作 get。注意到这里并没有正式的函数声明；单词 get 后跟一对花括号，里面是函数体。

❸ getter 函数必须要返回与变量类型相同的值。

❹ setter 函数叫作 set。这里并没有正式的函数声明；单词 set 后跟一对花括号，里面是函数体。

❺ setter 的行为就像是接收一个参数的函数。在默认情况下，参数通过局部名 newValue 进入 setter 函数中。

如下代码演示了计算变量的用法，这与其他变量没什么大的差别。该赋值时就赋值，该使用时就使用。不过在幕后，setter 与 getter 函数会被调用：

```
now = "Howdy" // Howdy ❶
print(now) // 2015-06-26 17:03:30 +0000 ❷
```

❶ 为 now 赋值会调用其 setter。传递给调用的参数就是所赋的值，这里就是 "Howdy"。该值进入 set 函数后成为 newValue。set 函数会将 newValue 打印到控制台上。

❷ 获取 now 会调用其 getter。get 函数会获取到当前的日期时间，然后将其转换为字符串并返回。接下来将该字符串打印到控制台上。

注意到第 1 行将 now 设为 "Howdy" 时，字符串 "Howdy" 并没有存储下来。比如，它对于第 2 行的 now 值就没起任何作用。set 函数可以存储值，不过它无法将其存储到计算变量中；计算变量是不可存储的！它只是调用 getter 与 setter 函数的便捷方法而已。

上述语法有几个变种：

- set 函数的参数名不一定非得叫作 newValue。要想指定不同的名字，将其放到单词 set 后面的圆括号中即可，如下代码所示：

```
set (val) { // now you can use "val" inside the setter function body
```

- 不一定要有 setter。如果省略 setter，那么变量就变成只读的了。为其赋值会导致编译器报错。没有 setter 的计算变量是 Swift 中创建只读变量的主要方式。
- 一定要有 getter！如果没有 setter，那么单词 get 与后面的花括号就可以省略了。如下代码是声明只读变量的合法方式：

```
var now : String {
    return NSDate().description
}
```

计算变量在很多地方都很有用。下面是其在实际使用中经常用到的地方：

*只读变量*

计算变量是创建只读变量最简单的方式，只需在声明中省略 setter 即可。通常情况下，变量是全局变量或属性；局部只读变量的意义并不大。

*函数门面*

如果每次需要一个值时，它都会由一个函数计算出来，那么可以通过更简单的语法将其表示为一个只读的计算变量。如下示例来自我所编写的代码：

```
var mp : MPMusicPlayerController {
    return MPMusicPlayerController.systemMusicPlayer()
}
```

每次想要引用时，都可以调用 MPMusicPlayerController.systemMusicPlayer()，

通过简单的名字 mp 来引用会更加紧凑一些。mp 表示一个事物而非动作表现，因此将 mp 当作变量会更好一些，这样在所有出现 mp 的地方，它都表示一个事物而非返回事物的函数。

*其他变量的门面*

计算变量可以位于存储变量之前，作为一个守护者，来确定如何设置和获取那些存储变量。这是相比于 Objective-C 的访问器方法的。在极端情况下，公开的计算变量是由一个私有的存储变量所维护的：

```
private var _p : String = ""
var p : String {
    get {
        return self._p
    }
    set {
        self._p = newValue
    }
}
```

这个示例本身没什么意义，因为对于访问器没什么可做的：我们只是直接设置和获取私有的存储变量而已，因此 p 与 _p 之间并没有什么实际的差别。但是基于该模板，你可以添加一些功能，在设置和获取时完成一些额外的事情。

正如上面的示例所示，计算实例属性函数可以引用其他实例属性，还可以调用实例方法。这是很重要的，因为一般来说，存储属性的初始化器这两件事都做不了。计算属性之所以可以，是因为直到实例存在了才可以调用它的函数。

如下示例演示了如何将计算变量用作存储门面。类有一个实例属性，它存储的数据很大，并且可以为 nil（它是一个 Optional，稍后将会介绍）：

```
var myBigDataReal : NSData! = nil
```

当应用进入后台时，我想减少内存使用（因为 iOS 会杀掉占据大量内存的后台应用）。因此，我计划将 myBigDataReal 数据保存为文件并存储到磁盘上，然后将变量本身设为 nil，这样可以释放内存中的数据。现在来考虑当应用回到前台，并且代码尝试获取 myBigDataReal 时会发生什么。如果它不为 nil，那么我们只需获取其值即可。但如果它为 nil，可能是因为我们将其值保存到了磁盘上。现在我想读取磁盘来恢复其值，然后获取。这正是计算变量门面的用武之地：

```
var myBigData : NSData! {
    set (newdata) {
        self.myBigDataReal = newdata
    }
```

```
        get {
            if myBigDataReal == nil {
                // ... get a reference to file on disk, f ...
                self.myBigDataReal = NSData(contentsOfFile: f)
                // ... erase the file ...
            }
            return self.myBigDataReal
        }
    }
```

## 3.5 setter 观察者

计算变量并不需要成为存储变量门面，这一点与你想的可能会不同。这是因为 Swift 提供了另一个漂亮的特性，可以让你将功能注入存储变量的 setter 中，即 setter 观察者。这些函数会在其他代码设置存储变量前后被调用。

声明具有 setter 观察者变量的语法非常类似于声明计算变量的语法：你可以编写一个 `willSet` 函数、一个 `didSet` 函数，二者也可以都提供：

```
var s = "whatever" { ❶
    willSet { ❷
        print(newValue) ❸
    }
    didSet { ❹
        print(oldValue) ❺
        // self.s = "something else"
    }
}
```

❶ 变量必须是 `var`（不能是 `let`）。可以为它赋初值，后跟一对花括号。

❷ `willSet` 函数，如果有，那就是 `willSet`，后跟一对花括号，里面是函数体。当其他代码设置该变量时它会被调用，就在变量接收到新值之前。

❸ 在默认情况下，`willSet` 函数会将接收到的新值设为 `newValue`。你可以在单词 `willSet` 后面的圆括号中提供不同的名字来改变这一点。旧值依然位于存储变量中，`willSet` 函数可以访问到它。

❹ `didSet` 函数，如果有，那就是 `didSet`，后跟一对花括号，里面是函数体。当其他代码设置该变量时它会被调用，就在变量接收到新值之后。

❺ 在默认情况下，`didSet` 函数会接收到旧值，它已经被变量值所替换，名字为 `oldValue`。你可以在单词 `didSet` 后面的圆括号中提供不同的名字来改变这一点。新值已经位于存储变量中，`didSet` 函数可以访问到它。此外，`didSet` 函数也可以将存储变量设为不同的值。

如果存储变量被初始化了或是 didSet 函数修改了存储变量值，那么 Setter 观察者函数就不会被调用，这是个循环！

实际上，在使用 Objective-C 中的 Setter 重写的大多数情况下，相对于计算变量来说，我更倾向于使用 Setter 观察者。如下示例来自于 Apple 提供的代码（Master–Detail Application 模板），它说明了一种典型场景，即在设置了某个属性后改变界面：

```
var detailItem: AnyObject? {
    didSet {
        // Update the view.
        self.configureView()
    }
}
```

这是视图控制器类的一个实例属性。每次修改该属性时，我们都需要改变界面，因为界面的一部分职责是显示该属性的值。因此，每次设置属性时，我们只需调用一个实例方法即可。该实例方法会读取属性值并相应地设置界面。

如下示例来自于我所编写的代码，我们不仅要修改界面，还要将设置的值限定在一个范围内：

```
var angle : CGFloat = 0 {
    didSet {
        // angle must not be smaller than 0 or larger than 5
        if self.angle < 0 {
            self.angle = 0
        }
        if self.angle > 5 {
            self.angle = 5
        }
        // modify interface to match
        self.transform = CGAffineTransformMakeRotation(self.angle)
    }
}
```

计算变量是没有 Setter 观察者的，它也不需要！有一个 Setter 函数，在设置值时的一些额外处理可以直接在该 Setter 函数中以编程的方式完成。

# 3.6 延迟初始化

术语延迟并非贬义；它是对一种重要行为的正式描述。如果存储变量在声明时被赋予一

个初始值，并且使用了延迟初始化，那么直到运行着的代码访问了该变量的值时才会计算初始值并完成赋值。

在 Swift 中，有 3 种类型的变量可以做到延迟初始化：

*全局变量*

全局变量自动就是延迟初始化的，如果你问自己，它们何时应该初始化，那么这就是答案。当应用启动时，文件与顶层代码都会执行，这时初始化全局变量是没有意义的，因为应用甚至还没有运行。这样，全局初始化必须要延迟到后面某个有意义的时间点处。因此，全局变量初始化直到其他代码首次引用它们时才会发生。在底层，该行为是由 `dispatch_once` 保护的；这使得初始化只会执行一次并且是线程安全的。

*静态属性*

静态属性的行为非常类似于全局变量，并且也是出于相同的原因。（Swift 中并没有存储类属性，因此类属性是无法初始化的，也不能做到延迟初始化。）

*实例属性*

在默认情况下，实例属性不是延迟初始化的，不过可以在声明中通过关键字 `lazy` 让它变成延迟初始化。该属性必须要通过 `var` 声明，而不是 `let`。如果在代码获取属性值之前有其他代码对该属性赋值，那么属性的初始化器就永远都不会执行。

延迟初始化器通常用于实现单例。单例是一种设计模式，所有代码都可以访问某个类的一个单独的共享实例：

```
class MyClass {
    static let sharedMyClassSingleton = MyClass()
}
```

其他代码可以通过 `MyClass.sharedMyClassSingleton` 获取到对 `MyClass` 单例的引用。直到其他代码首次这么调用时，单例实例才会创建出来；随后，无论调用多少次，返回的总是这个相同的实例。（如果这是计算只读属性，其 getter 调用了 `MyClass()` 并返回该实例，那么情况就不是这样的了，知道原因吗？）

现在来谈谈实例属性的延迟初始化。为何需要这个特性呢？一个原因是显而易见的：初始值的生成代价可能会很高，因此你希望只在需要时才生成。不过还有另外一个不那么明显的原因，而且这个原因更为重要：延迟初始化器可以做到正常的初始化器做不到的事情。特别地，它可以引用到实例，正常的初始化器却做不到这一点，因为在正常的初始化器运行时，实例还不存在（我们还在创建实例的过程中，因此实例尚未准备好）。与之相反，延迟初始化器直到实例已经创建出来后的某个时间点才会运行，因此可以引用到实例。比如，如果没有将 `arrow` 属性声明为 `lazy`，那么如下代码就是不合法的：

```
class MyView : UIView {
    lazy var arrow : UIImage = self.arrowImage()
    func arrowImage () -> UIImage {
        // ... big image-generating code goes here ...
    }
}
```

常见的写法是通过一个定义与调用匿名函数来初始化延迟实例属性：

```
lazy var prog : UIProgressView = {
    let p = UIProgressView(progressViewStyle: .Default)
    p.alpha = 0.7
    p.trackTintColor = UIColor.clearColor()
    p.progressTintColor = UIColor.blackColor()
    p.frame = CGRectMake(0, 0, self.view.bounds.size.width, 20)
    p.progress = 1.0
    return p
}()
```

语言中有一些小陷阱：延迟实例属性不能拥有 Setter 观察者，并且也没有 `lazy let`（因此无法将延迟实例属性设为只读）。不过，这些限制并不严重，因为对于存储属性来说，计算属性做不到的事情也不要指望 `lazy` 属性能够做到，如示例 3-1 所示。

示例 3-1：手工实现延迟属性

```
private var lazyOncer : dispatch_once_t = 0
private var lazyBacker : Int = 0
var lazyFront : Int {
    get {
        dispatch_once(&self.lazyOncer) {
            self.lazyBacker = 42 // expensive initial value
        }
        return self.lazyBacker
    }
    set {
        dispatch_once(&self.lazyOncer) {}
        // will set
        self.lazyBacker = newValue
        // did set
    }
}
```

在示例 3-1 中，原则在于只有 `lazyFront` 可以被外界访问；`lazyBacker` 是其底层存储，`lazyOncer` 使得一切只出现正确的次数。`lazyFront` 现在是个普通的计算变量，因此我们可以在设置时观察它（在其 Setter 函数中加入额外代码，位于“will set”与“did setter”注释处），也可以将其设为只读（将整个 Setter 删除）。

# 3.7 内建简单类型

每个变量与每个值都必须有一个类型。不过类型是什么呢？到目前为止，我已经假设存

在一些类型了，如 Int 与 String，不过并没有正式对其进行介绍。下面是 Swift 提供的主要的简单类型，以及适合于这些内建类型的实例方法、全局函数与运算符。（集合类型将会在第 4 章最后介绍。）

## 3.7.1 Bool

Bool 对象类型（结构体）只有两个值，真与假（或是与非）。你可以通过字面关键字 `true` 与 `false` 来表示这些值；显然，一个 Bool 值要么为 `true`，要么为 `false`：

```
var selected : Bool = false
```

在上述代码中，selected 是个 Bool 变量，并被初始化为 `false`；随后可以将其设为 `false` 或 `true`，但不能是其他值。由于其简单的真或假状态，这种 Bool 变量通常也叫作标识。

Cocoa 有很多方法都接收 Bool 参数或是返回 Bool 值。比如，当应用启动时，Cocoa 会调用如下声明的方法：

```
func application(application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
    -> Bool
```

你可以在该方法中做任何事情；但通常什么都不会做，不过必须要返回一个 Bool！在实际情况下，该 Bool 值总是 `true`。该函数最简单的实现如下所示：

```
func application(application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
    -> Bool {
        return true
}
```

Bool 在条件判断中很有用；第 5 章将会介绍，在说 `if` *something* 时，那么 *something* 就是个条件，它是个 Bool 值，或是会得到一个 Bool 值的表达式。比如，在使用相等比较运算符 `==` 来比较两个值时，结果就是个 Bool；如果两个值相等，那么结果就为 `true`，否则为 `false`：

```
if meaningOfLife == 42 { // ...
```

（稍后在谈及如 Int 和 String 等可以进行比较的类型时，我们还会继续介绍相等比较。）

在准备判断条件时，有时提前将 Bool 值存储到变量中会增强可读性：

```
let comp = self.traitCollection.horizontalSizeClass == .Compact
if comp { // ...
```

注意到在使用这种方式时，我们直接将 Bool 变量作为条件。写成 `if comp == true` 这样是非常愚蠢的做法，也是错误的，因为“如果 `comp` 为 `true`”，那就没必要显式测试它为 `true` 还是 `false`；条件表达式本身已经测试过了。

既然 Bool 可以用作条件，那么对返回一个 Bool 值的函数的调用也可以作为条件。如下示例来自于我所编写的代码。我声明了一个返回 Bool 值的函数，判断用户所选择的底牌是否是谜题的正确答案：

```
func evaluate(cells:[CardCell]) -> Bool { // ...
```

在其他地方可以这样调用：

```
if self.evaluate(cellsToTest) { // ...
```

与很多计算机语言不同，Swift 中没有任何东西可以隐式转换为或被当作 Bool。比如，在 C 中，boolean 实际上是个数字，0 是 `false`。不过在 Swift 中，除了 `false`，没有任何东西是 `false`，`true` 亦如此。

类型名 Bool 源自英国数学家 George Boole；布尔代数提供了逻辑运算。Bool 值可以应用到这些操作：

`!`

非。! 一元运算符用在 Bool 值前面，它会反转该 Bool 值。如果 `ok` 为 `true`，那么 `!ok` 就为 `false`，反之亦然。

`&&`

逻辑与。只有两个操作数都为 `true` 才会返回 `true`，否则返回 `false`。如果第 1 个操作数为 `false`，那么第 2 个操作数甚至都不会计算（从而避免可能的副作用）。

`||`

逻辑或。如果两个操作数有一个为 `true` 就返回 `true`，否则返回 `false`。如果第 1 个操作数为 `true`，那么第 2 个操作数甚至都不会计算（从而避免可能的副作用）。

如果逻辑运算很复杂，那么对子表达式加上圆括号会有助于厘清运算逻辑与顺序。

## 3.7.2 数字

主要的数字类型是 Int 与 Double，这表示你应该使用这两种类型。其他数字类型存在的主要目的就是与 C 和 Objective-C API 兼容，因为在编写 iOS 程序时，Swift 需要与它们通信。

### 1. Int

Int 对象类型（结构体）表示介于 Int.max 与 Int.min（包含首尾两个数字）之间的一个整数。实际的限定值取决于应用运行的平台与架构，因此不要完全依赖它们；在我的测试中，它们分别是 $2^{63}$-1 与 -$2^{63}$（64 位）。

表示一个 Int 最简单的方式就是将其作为一个数字字面值。在默认情况下，没有小数点的简单数字字面值都会被当作 Int。可以在数字间使用下划线，这有助于增强长数字的可读性。前导的 0 也是合法的，这有助于填补与对齐代码中的值。

可以通过二进制、八进制与十六进制来表示 Int 字面值。要想做到这一点，请分别在数字前加上 `0b`、`0o` 或 `0x`。比如，`0x10` 表示十进制 16。

### 2. Double

Double 对象类型（结构体）表示一个精度大约为小数点后 15 位的浮点数（64 位存储）。

表示一个 Double 值最简单的方式就是将其作为一个数字字面值。在默认情况下，包含小数点的任何数字字面值都会被当作 Double。可以在数字间使用下划线与前导 0。

Double 字面值不能以小数点开头！如果值介于 0 到 1 之间，那么请以前导 0 作为字面值的开始。（强调这一点的原因在于这与 C 和 Objective-C 有着明显的差别。）

可以通过科学计数法表示 Double 字面值。字母 `e` 后面的内容就是 10 的指数。如果小数部分为 0，那就可以省略小数点。比如，`3e2` 就表示 3 乘以 $10^2$（300）。

还可以通过十六进制表示 Double 字面值。要想做到这一点，请以 `0x` 作为字面值的开头。这里也可以使用乘方（还是可以省略小数点）；字母 `p` 后面的内容就是 2 的指数。比如，`0x10p2` 就表示十进制 64，因为是 16 乘以 $2^2$。

除了其他属性，`Double` 还有一个静态属性 `Double.infinity` 和一个实例属性 `isZero`。

### 3. 强制类型转换

强制类型转换指的是将一种数字类型的值转换为另一种。Swift 并没有提供显式类型转换，不过通过实例化来达到相同的目的。要想将一个 Int 显式转换为 Double，请在圆括号中使用 Int 来实例化一个 Double。要想将一个 Double 显式转换为 Int，请在圆括号中使用 Double 来实例化一个 Int；这么做会截断原始值（小数点后的一切都会被丢弃）：

```
let i = 10
let x = Double(i)
print(x) // 10.0, a Double
let y = 3.8
let j = Int(y)
```

```
print(j) // 3, an Int
```

在将数字值赋给变量或作为参数传递给函数时，Swift 只会执行字面值的隐式转换。如下代码是合法的：

```
let d : Double = 10
```

不过如下代码是非法的，因为你所赋予的变量（非字面值）是另外一种类型；编译器会阻止你这么做：

```
let i = 10
let d : Double = i // compile error
```

解决办法就是在赋值或传递变量时进行显式转换：

```
let i = 10
let d : Double = Double(i)
```

使用算术运算合并数字值时也需要遵循该原则。Swift 只会执行隐式转换。常见的情况是对 Int 与 Double 执行算术运算；Int 会被当作 Double：

```
let x = 10/3.0
print(x) // 3.33333333333333
```

不过，如果对不同数字类型的变量执行算术运算，这些变量需要进行显式转换，这样才能确保它们都是相同类型。比如：

```
let i = 10
let n = 3.0
let x = i / n // compile error; you need to say Double(i)
```

这些原则显然都是 Swift 严格类型的结果；不过，与其他现代计算机语言相比，Swift 对待数字值的方式有着很大的不同，可能会让你叫苦不迭。到目前为止，我所给出的示例都很容易，不过如果算术表达式很长，那么事情就会变得更加复杂，而且问题还会同为了保持与 Cocoa 兼容所需的其他数字类型交织在一起，下面就来谈谈。

### 4. 其他数值类型

如果没有编写 iOS 应用，而是单纯使用 Swift，那么你可能只会用到 Int 与 Double 来完成所有的算术运算。但遗憾的是，编写 iOS 程序需要 Cocoa，而 Cocoa 中还有很多其他的数值类型，Swift 也提供了与之匹配的类型。因此，除了 Int，还有各种大小的有符号整型（如 Int8、Int16、Int32 及 Int64），以及无符号整型 UInt、UInt8、UInt16、UInt32 及 UInt64。除了 Double，还有低精度的 Float（32 位存储、大约保留小数点后 6 或 7 位

精度）以及扩展精度的 Float80；在 Core Graphics 框架中还有 CGFloat（其大小可以是 Float 或 Double，这取决于架构的位数）。

在使用 C API 时还会遇到 C 数值类型。对于 Swift 来说，这些类型只是类型别名而已，这意味着它们是其他类型的别名；比如，CDouble（对应于 C 的 `double`）只是 Double 的另一个名字，CLong（C 中的 `long`）是 Int 类型等。很多其他的数值类型别名都会出现在各种 Cocoa 框架中；比如，NSTimeInterval 只是 Double 的类型别名而已。

问题来了。我之前曾说过，不能通过变量赋值、传递或组合不同数值类型的值；你只能显式将这些值转换为正确的类型才行。不过，现在你面对的是 Cocoa 中众多类型的数值！Cocoa 传递给你的数值很可能既不是 Int 也不是 Double，你可能根本就发现不了，直到编译器告诉你出现了类型不匹配的情况。接下来，你需要搞清楚到底什么地方错了，然后将这些变量转换为相同的类型。

如下这个典型示例来自于我的应用。我有一个 UIImage，将其 CGImage 抽取出来，现在想要通过 CGSize 来表示该 CGImage 的大小：

```
let mars = UIImage(named:"Mars")!
let marsCG = mars.CGImage
let szCG = CGSizeMake( // compile error
    CGImageGetWidth(marsCG),
    CGImageGetHeight(marsCG)
)
```

问题在于 `CGImageGetWidth` 与 `CGImageGetHeight` 返回的是 Int，而 `CGSizeMake` 接收的却是 CGFloat。这并非 C 或 Objective-C 的问题，因为它们可以实现从前者到后者的隐式类型转换。问题在于 Swift，你只能执行显式类型转换：

```
var szCG = CGSizeMake(
    CGFloat(CGImageGetWidth(marsCG)),
    CGFloat(CGImageGetHeight(marsCG))
)
```

下面是另一个实际的例子。界面中的滑块是个 UISlider，其 `minimumValue` 与 `maximumValue` 都是 Float。在如下代码中，s 是个 UISlider，g 是个 UIGestureRecognizer，我们要通过手势识别器将滑块移动到用户轻拍的位置处：

```
let pt = g.locationInView(s)
let percentage = pt.x / s.bounds.size.width
let delta = percentage * (s.maximumValue - s.minimumValue) // compile error
```

上述代码无法编译通过。pt 是个 CGPoint，因此 `pt.x` 是个 CGFloat。幸好，`s.bounds.size.width` 也是个 CGFloat，因此第 2 行代码可以编译通过；现在的 `percentage` 被推

断为是个 CGFloat。不过在第 3 行，percentage 与 s.maximumValue 和 s.minimumValue 一同参与运算，后两者是 Float，并非 CGFloat。必须要进行显式类型转换：

```
let delta = Float(percentage) * (s.maximumValue - s.minimumValue)
```

图 3-1：快速帮助会显示出变量的类型

唯一的好消息是，如果大部分代码都能编译通过，那么 Xcode 的快速帮助特性会告诉你 Swift 推断出某个变量的类型到底是什么（如图 3-1 所示）。这可以帮助你定位关于数值类型的问题。

有时，你需要赋值或传递一种整型类型，但目标需要的却是另一种整型类型，而你也不知道到底需要哪一种整型类型，这时可以通过调用 numericCast 让 Swift 进行动态类型转换。比如，如果 i 与 j 是之前声明的不同整型类型的变量，那么 i = numericCast(j) 就会将 j 强制转换为 i 的整型类型。

## 5. 算术运算

Swift 的算术运算符与你想的一样；它们与其他计算机语言和真正的算术运算非常类似：

+

加运算符。将第 2 个操作数加到第 1 个并返回结果。

-

减运算符。从第 1 个操作数中减掉第 2 个并返回结果。一元减运算符用作操作数的前缀，看起来与它一样，但返回的却是操作数的相反数（事实上，还有个一元加运算符，它原样返回操作数）。

*

乘运算符。将第 1 个操作数与第 2 个相乘并返回结果。

/

除运算符。将第 1 个操作数除以第 2 个并返回结果。

与 C 一样，两个 Int 相除得到的还是 Int；小数部分均会丢弃掉。10/3 的结果为 3，而不是 3 又 1/3。

%

余数运算符。将第 1 个操作数除以第 2 个并返回余数。如果第 1 个操作数是负数，那么结果就是负数；如果第 2 个操作数是负数，那么结果为正数。浮点操作数是合法的。

整型类型可以看作二进制位，因此可以进行二进制位运算：

&

按位与。如果两个操作数的同一位均为 1，那么结果就为 1。

|

按位或。如果两个操作数的同一位均为 0，那么结果就为 0。

^

按位异或。如果两个操作数的同一位不同，那么结果就为 1。

~

按位取反。它用在单个操作数之前，对每一位取反并返回结果。

<<

左移。将第 1 个操作数向左移动第 2 个操作数所指定的位数。

>>

右移。将第 1 个操作数向右移动第 2 个操作数所指定的位数。

从技术上来说，如果整型是无符号的，那么位移运算符会执行逻辑位移；如果整型是有符号的，那么它会执行算术位移。

整型上溢或下溢（比如，将两个 Int 相加，导致结果超出 Int.max）是个运行时错误（应用会崩溃）。对于简单的情况来说，编译器会阻止你这么做，但是你可以轻松绕过编译器的检查：

```
let i = Int.max - 2
let j = i + 12/2 // crash
```

在某些情况下，你希望强制这种操作能够成功，因此需要提供特殊的上溢 / 下溢方法。这些方法会返回一个元组；虽然还没有介绍过元组，但是我还是打算展示这样一个示例：

```
let i = Int.max - 2
let (j, over) = Int.addWithOverflow(i,12/2)
```

现在，j 值为 Int.min + 3（因为其值已经由原来的对 Int.max 的包装变成了对 Int.min 的包装），over 值为 true（用于报告溢出情况）。

如果你对是否存在上溢 / 下溢的情况不在乎，那么可以通过特殊的算术运算符来消除错误：&+、&- 和 &*。

你常常会将现有变量值与另一个值合并起来，然后将结果存储到相同的变量中。请记住，为了做到这一点，你需要将变量声明为 var：

```
var i = 1
i = i + 7
```

作为一种简便写法，你可以通过一个运算符一步完成算术运算与赋值：

```
var i = 1
i += 7
```

简便（复合）赋值算术运算符有 +=、-=、*=、/=、%=、&=、|=、^=、~=、<<= 和 >>=。

我们常常需要将某个数值加 1 或减 1，Swift 提供了一元增加与减少运算符 ++ 和 --。区别在于它们用作前缀还是后缀。如果用作前缀（++i、--i），那么值就会增加或减少，并存储到相同的变量中，然后用于外部表达式中；如果用作后缀（i++、i--），那么变量当前值就会用在外部表达式中，然后值再增加或减少，并存储到相同变量中。显然，变量必须要通过 var 声明才可以。

运算优先级也是非常直观的：比如，* 的优先级比 + 要高，因此 x+y*z 会先执行 y*z，然后再将结果与 x 相加。如果有问题，可以通过圆括号消除歧义；比如，(x+y)*z 就会先执行加法操作。

全局函数包含了 abs（取绝对值）、max 和 min：

```
let i = -7
let j = 6
print(abs(i)) // 7
print(max(i,j)) // 6
```

其他数学函数（如取平方根、四舍五入、伪随机数、三角函数等）都来自于 C 标准库，可以正常使用它们，因为你已经导入了 UIKit。还得小心数值类型，即便对于字面值来说也没有隐式转换。

比如，sqrt 接收一个 C double，它是个 CDouble 类型，也是个 Double 类型。因此，不能写成 sqrt(2)，只能写成 sqrt(2.0)。与之类似，arc4random 会返回一个 UInt32 类型。如果 n 是个 Int 类型，同时希望得到一个介于 0 到 n-1 之间的随机数，那么你不能写成 arc4random()%n；只能将调用 arc4random 的结果强制转换为 Int。

### 6. 比较

数字是通过比较运算符进行比较的，运算符返回一个 Bool。比如，表达式 i==j 用于判断 i 与 j 是否相等；如果 i 与 j 是数字，那么相等就表示数值上的相等。因此，只有 i 和 j 是相同的数字，i==j 才为 true，这与你的期望是完全一致的。

比较运算符有：

==
: 相等运算符，操作数相等才会返回 true。

!=
: 不等运算符。操作数相等会返回 false。

<
: 小于运算符。如果第 1 个操作数小于第 2 个，那么会返回 true。

<=
: 小于等于运算符。如果第 1 个操作数小于或等于第 2 个，那么会返回 true。

>
: 大于运算符。如果第 1 个操作数大于第 2 个，那么会返回 true。

>=
: 大于等于运算符。如果第 1 个操作数大于或等于第 2 个，那么会返回 true。

请记住，基于计算机存储数字的方式，Double 值的相等性比较可能会与你期望的不一致。要想判断两个 Double 是否相等，更可靠的方式是将它们的差值与一个非常小的值进行比较（通常叫作 ε）。

```
let isEqual = abs(x - y) < 0.000001
```

## 3.7.3 String

String 对象类型（结构体）表示文本。表示 String 值最简单的方式是使用字面值，并由一对双引号围起来：

```
let greeting = "hello"
```

Swift 字符串是非常现代化的；在底层，它是个 Unicode，你可以在字符串字面值中直接包含任意字符。如果不想敲 Unicode 字符，同时又知道它的代码，那么可以使用符号 \u{...?}，其中花括号之间最多会有 8 个十六进制数字：

```
let leftTripleArrow = "\u{21DA}"
```

字符串中的反斜杠是转义字符；它表示“我并不是一个反斜杠，而是告诉你要特别对待下一个字符”。各种不可打印以及容易造成歧义的字符都是转义字符，最重要的转义字符有：

`\n`
: UNIX 换行符。

`\t`
: 制表符。

`\"`
: 引号（这里的转义是表示它并非字符串字面值的结束）。

`\\`
: 反斜杠（因为单独一个反斜杠是转义字符）。

Swift 最酷的特性之一就是字符串插入。你可以将待输出的任何值使用 `print` 嵌入字符串字面值中作为字符串，即便它本身并非字符串也可以，使用的是转义圆括号 `\(...?)`，比如：

```
let n = 5
let s = "You have \(n) widgets."
```

现在，`s` 表示字符串“`You have 5 widgets`”。该示例本身没什么太大价值，因为我们知道 `n` 是什么，并且可以直接在字符串中输入 `5`；不过，如果我们不知道 `n` 是什么呢！此外，转义圆括号中的内容不一定非得是变量的名字；它可以是 Swift 中任何合法的表达式。如果不知道怎么用，如下示例会更具价值：

```
let m = 4
let n = 5
let s = "You have \(m + n) widgets."
```

转义圆括号中不能有双引号。这令人感到失望，但却不是什么障碍；这时只需将其赋给一个变量，然后在圆括号中使用该变量即可。比如，你不能这么做：

```
let ud = NSUserDefaults.standardUserDefaults()
let s = "You have \(ud.integerForKey("widgets")) widgets." // compile error
```

对双引号转义也无济于事，你只能写成多行，如下代码所示：

```
let ud = NSUserDefaults.standardUserDefaults()
let n = ud.integerForKey("widgets")
let s = "You have \(n) widgets."
```

要想拼接两个字符串，最简单的方式是使用 `+` 运算符（以及 `+=` 赋值简写方式）：

```
let s = "hello"
let s2 = " world"
let greeting = s + s2
```

这种便捷符号是可以的，因为 + 运算符已经被重载了：对于操作数是数字以及操作数是字符串的情况，它的行为是不同的，前者执行数字相加，后者执行字符串拼接。第 5 章将会介绍，所有运算符都可以重载，你可以重载它们以便对自己定义的类型执行恰当的操作。

作为 += 的替代，你还可以调用 appendContentsOf 实例方法：

```
var s = "hello"
let s2 = " world"
s.appendContentsOf(s2) // or: s += s2
```

拼接字符串的另一种方式是使用 joinWithSeparator 方法。通过一个待拼接的字符串数组调用它（没错，我们还没开始介绍数组呢），并将插入其中的字符串传递给该数组：

```
let s = "hello"
let s2 = "world"
let space = " "
let greeting = [s,s2].joinWithSeparator(space)
```

比较运算符也进行了重载，这样它们就都可以用于 String 操作数。如果两个 String 包含相同的文本，那么它们就是相等的（==）。如果一个 String 按照字母表顺序位于另一个之前，那么前一个就小于后一个。

Swift 还提供了一些附加的便捷实例方法与属性。isEmpty 会返回一个 Bool，表示字符串是否为空字符串（""）。hasPrefix 与 hasSuffix 判断字符串是否以另一个字符串开始或结束；比如，"hello".hasPrefix("he") 返回 true。uppercaseString 与 lowercaseString 属性提供了原始字符串的大写与小写版本。

可以在 String 与 Int 之间进行强制类型转换。要想创建一个表示 Int 的字符串，使用字符串插入即可；此外，还可以使用 Int 作为 String 初始化器，就好像在数字类型之间进行强制类型转换一样：

```
let i = 7
let s = String(i) // "7"
```

字符串还可以通过其他进制来表示 Int，提供一个 radix: 参数来表示进制：

```
let i = 31
let s = String(i, radix:16) // "1f"
```

能够表示数字的 String 还可以强制转换为数字类型；整型类型会接收一个 radix: 参数来

表示基数。不过，这个转换可能会失败，因为String可能不是表示指定类型的数字；这样，结果就不是数字，而是一个包装了数字的Optional（现在还没有介绍过Optional，相信我就好；第4章将会介绍可失败的初始化器）：

```
let s = "31"
let i = Int(s) // Optional(31)
let s2 = "1f"
let i2 = Int(s2, radix:16) // Optional(31)
```

实际上，String的强制类型转换是字符串插值与使用print在控制台打印的基础。你可以将任何对象转换为String，方式是让其遵循如下3个协议之一：Streamable、CustomStringConvertible与CustomDebugStringConvertible。第4章介绍协议时会给出相关的示例。

可以通过characters属性的count方法获得String的字符长度：

```
let s = "hello"
let length = s.characters.count // 5
```

为何String没有提供length属性呢？这是因为String并没有一个简单意义上的长度概念。String是以Unicode编码序列的形式存在的，不过多个Unicode编码才能构成一个字符；因此，为了知道一个序列表示多少个字符，我们需要遍历序列，将其解析为所表示的字符。

你也可以遍历String的字符。最简单的方式是使用for...?in结构（参见第5章）。这么做所得到的是Character对象，稍后将会对其进行深入介绍。

```
let s = "hello"
for c in s.characters {
    print(c) // print each Character on its own line
}
```

在更深的层次上，可以通过utf8与utf16属性将String分解为UTF-8编码与UTF-16编码。

```
let s = "\u{BF}Qui\u{E9}n?"
for i in s.utf8 {
    print(i) // 194, 191, 81, 117, 105, 195, 169, 110, 63
}
for i in s.utf16 {
    print(i) // 191, 81, 117, 105, 233, 110, 63
}
```

还有一个unicodeScalars属性，它将String的UTF-32编码集合表示为一个UnicodeScalar结构体。要想从数字编码构造字符串，请通过数字实例化一个

UnicodeScalar 并将其 append 到 String 上。下面这个辅助函数会将一个两字母的国家缩写转换为其国旗的表情符号：

```
func flag(country:String) -> String {
    let base : UInt32 = 127397
    var s = ""
    for v in country.unicodeScalars {
        s.append(UnicodeScalar(base + v.value))
    }
    return s
}
// and here's how to use it:
let s = flag("DE")
```

奇怪的是 Swift 并没有提供更多关于标准字符串操作的方法。比如，如何将一个字符串转换为大写，如何判断某个字符串是否包含了给定的子字符串。大多数现代编程语言都提供了紧凑、方便的方式来做到这一点，但 Swift 却不行。原因在于 Foundation 框架所提供的特性的缺失，在实际开发中你总是会导入它（导入 UIKit 就会导入 Foundation）。Swfit String 桥接了 Foundation NSString。这意味着在很大程度上，当使用 Swift String 时，真正使用的却是 Foundation NSString 方法。比如：

```
let s = "hello world"
let s2 = s.capitalizedString // "Hello World"
```

capitalizedString 属性来自于 Foundation 框架，它由 Cocoa 而非 Swift 提供。这是个 NSString 属性，它是附着在 String 上的。与之类似，如下代码展示了如何定位某个字符串中的一个子字符串：

```
let s = "hello"
let range = s.rangeOfString("ell") // Optional(Range(1..<4))
```

现在尚未介绍过 Optional 和 Range（本章后面将会对其进行介绍），不过上述代码起到了连接 Swift 与 Cocoa 的作用：Swift String s 变成了一个 NSString，NSString rangeOfString 方法被调用，返回了一个 Foundation NSRange 结构体，然后 NSRange 又被转换为 Swift Range 并被包装为一个 Optional。

不过有时，你并不希望进行这种转换。出于各种各样的原因，你只想使用 Foundation，并接收 Foundation NSRange。为了做到这一点，你需要通过 as 运算符（第 4 章将会介绍类型转换）显式将字符串转换为 NSString：

```
let s = "hello"
let range = (s as NSString).rangeOfString("ell") // (1,3), an NSRange
```

再来看一个示例，该示例也涉及 NSRange。假设你想要根据范围（第 2、3、4 个字符）从“hello”中获取到字符串“ell”。Foundation NSString 的方法 substringWithRange:

要求你提供一个范围，表示一个 NSRange。你可以直接通过 Foundation 函数构造一个 NSRange，不过如果这么做，代码将无法通过编译：

```
let s = "hello"
let ss = s.substringWithRange(NSMakeRange(1,3)) // compile error
```

编译报错的原因在于 Swift 已经吸纳了 NSString 的 `substringWithRange:`，它这里希望你提供一个 Swift Range。稍后将会介绍如何做到这一点，不过通过类型转换让 Swift 使用 Foundation 会更简单一些，如下代码所示：

```
let s = "hello"
let ss = (s as NSString).substringWithRange(NSMakeRange(1,3)) // "ell"
```

## 3.7.4 Character

Character 对象类型（结构体）表示单个 Unicode 字母，即字符串中的一个字符。可以通过 `characters` 属性将 String 对象分解为一系列 Character 对象。形式上，它是一个 `String.CharacterView` 结构体；不过习惯称为字符序列。如前所述，可以通过 `for...in` 遍历字符序列来获取 String 的 Characters，一个接着一个：

```
let s = "hello"
for c in s.characters {
    print(c) // print each Character on its own line
}
```

在字符序列之外遇到 Character 对象的情况并不多，甚至都没有创建 Character 字面值的方式。要想从头创建一个 Character，请通过单字符的 String 进行初始化：

### String 与 NSString 元素的失配

Swift 与 Cocoa 对字符串包含什么元素有着不同的理解。Swift 涉及字符，而 NSString 则涉及 UTF-16 编码。每种方式都有自己的优点。相比于 Swift 来说，NSString 速度更快，效率更高；Swift 必须要遍历字符串才能知晓字符是如何构建的；不过，Swift 的做法与你的直觉是相一致的。为了强调这种差别，非字面值的 Swift 字符串没有 `length` 属性；它与 NSString 的 `length` 的对应之物则是其 `utf16` 属性的 `count`。

幸好，元素失配在实际情况中并不常见；不过，还是存在这种可能的，下面是一个测试：

```
let s = "Ha\u{030A}kon"
print(s.characters.count) // 5
let length = (s as NSString).length // or: s.utf16.count
```

```
print(length) // 6
```

上述代码通过一个 Unicode 编码创建了一个字符串（挪威语 Håkon），这个 Unicode 编码与前面的编码一同构成了一个字符，该字符上面会有一个圆圈。Swift 会遍历整个字符串，因此它会规范化这个字符串组合并返回 5 个字符；Cocoa 只会看到该字符串包含了 6 个 16 位编码。

```
let c = Character("h")
```

出于同样的原因，你可以通过一个 Character 初始化 String。

```
let c = Character("h")
let s = (String(c)).uppercaseString
```

可以比较 Character，“小于”的含义与你的理解是一致的。

字符序列有很多方便好用的属性与方法。由于是个集合（CollectionType），所以它拥有 `first` 与 `last` 属性；它们都是 Optional，因为字符串可能为空：

```
let s = "hello"
let c1 = s.characters.first // Optional("h")
let c2 = s.characters.last // Optional("o")
```

`indexOf` 方法会在序列中找到给定字符首次出现的位置并返回其索引。它也是个 Optional，因为给定的字符可能在序列中并不存在：

```
let s = "hello"
let firstL = s.characters.indexOf("l") // Optional(2)
```

所有的 Swift 索引都是从数字 0 开始的，因此 2 表示第 3 个字符。不过，这里的索引值并不是 Int；稍后将会介绍它到底是什么以及这么做的好处。

由于是个序列（SequenceType），字符序列有一个返回 Bool 的方法 `contains`，它表示序列中是否存在某个字符：

```
let s = "hello"
let ok = s.characters.contains("o") // true
```

此外，`contains` 还可以接收一个函数，这个函数会接收一个 Character 并返回 Bool（`indexOf` 方法也可以这么做）。如下代码判断目标字符串是否包含元音：

```
let s = "hello"
let ok = s.characters.contains {"aeiou".characters.contains($0)} // true
```

`filter` 方法接收一个函数，这个函数接收一个 Character 并返回 Bool，它会排除掉返回

false的那些字符。其结果是个字符序列，不过你可以将其强制转换为String。如下代码展示了如何删除一个String中出现的所有辅音：

```
let s = "hello"
let s2 = String(s.characters.filter {"aeiou".characters.contains($0)}) // "eo"
```

dropFirst与dropLast方法分别会返回一个排除掉第一个与最后一个字符的新字符序列：

```
let s = "hello"
let s2 = String(s.characters.dropFirst()) // "ello"
```

prefix与suffix会从初始字符序列的起始与末尾处提取出给定长度的字符序列：

```
let s = "hello"
let s2 = String(s.characters.prefix(4)) // "hell"
```

split会根据一个函数（该函数接收一个Character并返回Bool）将字符序列转换为数组。在如下示例中，我得到了一个String中的单词，这里的“单词”指的是除了空格外的其他字符。

```
let s = "hello world"
let arr = s.characters.split{$0 == " "}
```

不过，得到的结果是个相当奇怪的SubSlice对象数组；为了获得String对象，我们需要使用map函数将其转换为String。第4章将会介绍map函数，现在使用它就好了：

```
let s = "hello world"
let arr = split(s.characters){$0 == " "}.map{String($0)} // ["hello", "world"]
```

我们还可以像操作数组那样操作String（实际上是其底层的字符序列）。比如，你可以通过下标获得指定位置处的字符。但遗憾的是，这其实并不是那么容易的。比如，“hello”的第2个字符是什么？如下代码无法编译通过：

```
let s = "hello"
let c = s[1] // compile error
```

原因在于String上的索引（实际上是其字符序列上的索引）是一种特殊的嵌套类型String.Index（实际上是String.CharacterView.Index的类型别名）。创建该类型的对象并不是那么容易的事情。首先使用String（或字符序列）的startIndex或endIndex，或indexOf方法的返回值；接下来调用advancedBy方法获得所需的索引：

```
let s = "hello"
let ix = s.startIndex
let c = s[ix.advancedBy(1)] // "e"
```

这种做法非常笨拙，原因在于 Swift 只有遍历完序列后才能知道字符序列中的字符到底在哪里；调用 advancedBy 就是为了让 Swift 做到这一点。

除了 advancedBy 方法，还可以通过 ++ 与 -- 来增加或是减少索引值，可以通过 successor 与 predecessor 方法得到下一个与前一个索引值。这样，可以将上述示例修改为下面这样：

```
let s = "hello"
var ix = s.startIndex
let c = s[++ix] // "e"
```

也可以写成这样：

```
let s = "hello"
let ix = s.startIndex
let c = s[ix.successor()] // "e"
```

得到了所需的字符索引值后，你就可以通过它来修改 String 了。比如，insertContentsOf(at:) 方法会将一个字符序列（不是 String）插入 String 中：

```
var s = "hello"
let ix = s.characters.startIndex.advancedBy(1)
s.insertContentsOf("ey, h".characters, at: ix) // s is now "hey, hello"
```

与之类似，removeAtIndex 会删除单个字符（并返回该字符）。

（涉及更多字符的操作需要用到 Range，3.7.5 节将会对其进行介绍）。

值得注意的是，我们可以将字符序列直接转换为 Character 对象数组，如 Array("hello".characters)。这么做是很值得的，因为数组索引是 Int，使用起来很容易。操纵完 Character 数组后，你可以直接将其转换为 String。3.7.5 节将会介绍相关示例（第 4 章将会介绍数组，还会再次谈及集合与序列）。

## 3.7.5 Range

Range 对象类型（结构体）表示一对端点。有两个运算符可以构造一个 Range 字面值；提供一个起始值和一个终止值，中间是一个 Range 运算符：

`...`

闭区间运算符。符号 a...b 表示“从 a 到 b，包括 b”。

`..<`

半开半闭区间运算符。符号 a..<b 表示“从 a 到 b，但不包含 b”。

可以在 Range 运算符左右两侧使用空格。

不存在反向 Range：Range 的起始值不能大于终止值（编译器不会报错，但运行时会崩溃）。

Range 端点的类型通常是某种数字，大多数情况下是 Int：

```
let r = 1...3
```

如果终止值是负数，那么必须将其放到圆括号中：

```
let r = -1000...(-1)
```

Range 的常见用法是在 `for...in` 中遍历数字：

```
for ix in 1 ... 3 {
    print(ix) // 1, then 2, then 3
}
```

还可以使用 Range 的 `contains` 实例方法判断某个值是否在给定的范围内；在这种情况下，Range 实际上是个间隔（严格来说是个 IntervalType）：

```
let ix = // ... an Int ...
if (1...3).contains(ix) { // ...
```

为了测试包含，Range 的端点还可以是 Double：

```
let d = // ... a Double ...
if (0.1...0.9).contains(d) { // ...
```

Range 的另一个常见使用场景是对序列进行索引。比如，如下代码获取到一个 String 的第 2、3、4 个字符。正如 3.7.4 节最后所介绍的那样，我们将 String 的 `characters` 转换为了一个 Array；接下来将 Int Range 作为该数组的索引，然后再将其转换为 String：

```
let s = "hello"
let arr = Array(s.characters)
let result = arr[1...3]
let s2 = String(result) // "ell"
```

此外，可以直接将 Range 作为 String（或其底层字符序列）的索引，不过这时它必须是 `String.Index` 的 Range，正如之前所说的，这么做非常笨拙。更好的方式是让 Swift 将从 Cocoa 方法调用中得到的 NSRange 转换为 Swift Range：

```
let s = "hello"
let r = s.rangeOfString("ell") // a Swift Range (wrapped in an Optional)
```

还可以将Range端点作为索引值，比如，使用String startIndex的advancedBy，如前所述。得到了恰当类型的Range后，你就可以通过下标来抽取出子字符串了：

```
let s = "hello"
let ix1 = s.startIndex.advancedBy(1)
let ix2 = ix1.advancedBy(2)
let s2 = s[ix1...ix2] // "ell"
```

一种优雅的便捷方式是从序列的indices属性开始，它会返回一个介于序列startIndex与endIndex之间的半开Range区间；接下来就可以修改该Range并使用它了：

```
let s = "hello"
var r = s.characters.indices
r.startIndex++
r.endIndex--
let s2 = s[r] // "ell"
```

replaceRange方法会拼接为一个范围，这样就可以修改字符串了：

```
var s = "hello"
let ix = s.startIndex
let r = ix.advancedBy(1)...ix.advancedBy(3)
s.replaceRange(r, with: "ipp") // s is now "hippo"
```

与之类似，可以通过removeRange方法来删除一系列字符：

```
var s = "hello"
let ix = s.startIndex
let r = ix.advancedBy(1)...ix.advancedBy(3)
s.removeRange(r) // s is now "ho"
```

Swift Range与Cocoa NSRange的构建方式存在着很大的差别。Swift Range是由两个端点定义的，Cocoa NSRange则是由一个起始点和一个长度定义的。不过，你可以将端点为Int的Swift Range转换为NSRange，也可以通过toRange方法将NSRange转换为Swift Range（返回一个包装了Range的Optional）。

有时，Swift会更进一步。比如，当调用"hello".rangeOfString("ell")时，Swift会桥接Range与NSRange，它能够正确处理好Swift与Cocoa在字符解释与字符串长度上的差别，以及NSRange的值是Int，而描述Swift子字符串的Range端点是String.Index这些情况。

## 3.7.6 元组

元组是个轻量级、自定义、有序的多值集合。作为一种类型，它是通过一个圆括号，里面是所含值的类型，类型之间通过逗号分隔来表示的。比如，下面是一个包含Int与String的元组类型变量的声明：

```
var pair : (Int, String)
```

元组字面值的表示方式也是一样的，圆括号中是所包含的值，值与值之间通过逗号分隔：

```
var pair : (Int, String) = (1, "One")
```

这些类型可以推导出来，因此没必要在声明中显式指定类型：

```
var pair = (1, "One")
```

元组是纯粹的 Swift 语言特性，它们与 Cocoa 和 Objective-C 并不兼容，因此只能将其用在 Cocoa 无法触及之处。不过在 Swift 中，它们有很多用武之地。比如，元组显然就是函数只能返回一个值这一问题的解决之道；元组本身是一个值，但它可以包含多个值，因此将元组作为函数的返回类型可以让函数返回多个值。

元组具有很多语言上的便捷性，你可以赋值给变量名元组，以此作为同时给多个变量赋值的一种方式：

```
var ix: Int
var s: String
(ix, s) = (1, "One")
```

这么做非常方便，Swift 可以在一行完成对多个变量同时初始化的工作：

```
var (ix, s) = (1, "One") // can use let or var here
```

可以通过元组安全地实现变量值的互换：

```
var s1 = "Hello"
var s2 = "world"
(s1, s2) = (s2, s1) // now s1 is "world" and s2 is "Hello"
```

全局函数 swap 能以更加通用的方式实现值的互换。

要想忽略掉其中一个赋值，请在接收元组中使用下划线表示：

```
let pair = (1, "One")
let (_, s) = pair // now s is "One"
```

enumerate 方法可以通过 for...in 遍历序列，然后在每次迭代中接收到每个元素的索引号与元素本身；这两个结果是以元组的形式返回的：

```
let s = "hello"
for (ix,c) in s.characters.enumerate() {
```

```
    print("character \(ix) is \(c)")
}
```

我之前曾指出过，addWithOverflow 等数字的实例方法会返回一个元组。

可以直接引用元组的每个元素。第 1 种方式是通过索引号，将字面数字（不是变量值）作为消息名发送给元组，并使用点符号：

```
let pair = (1, "One")
let ix = pair.0 // now ix is 1
```

如果对元组的引用不是常量，那么可以通过相同手段为其赋值：

```
var pair = (1, "One")
pair.0 = 2 // now pair is (2, "One")
```

访问元组元素的第 2 种方式是给元组命名，这类似于函数参数，并且要作为显式或隐式类型声明的一部分。下面是创建元组元素名的一种方式：

```
let pair : (first:Int, second:String) = (1, "One")
```

下面是另一种方式：

```
let pair = (first:1, second:"One")
```

名字现在是该值类型的一部分，并且要通过随后的赋值来访问。接下来可以将其用作字面消息名，就像数字字面值一样：

```
var pair = (first:1, second:"One")
let x = pair.first // 1
pair.first = 2
let y = pair.0 // 2
```

可以将没有名字的元组赋给相应的有名字的元组，反之亦然：

```
let pair = (1, "One")
let pairWithNames : (first:Int, second:String) = pair
let ix = pairWithNames.first // 1
```

在传递或是从函数返回一个元组时可以省略元组名：

```
func tupleMaker() -> (first:Int, second:String) {
    return (1, "One") // no names here
}
let ix = tupleMaker().first // 1
```

如果在程序中会一以贯之地使用某种类型的元组，那么为它起个名字就很有必要了。要想做到这一点，请使用 Swift 的 typealias 关键字。比如，在我开发的 LinkSame 应用

中有一个 Board 类，它描述并且操纵着游戏格局。Board 是由 Piece 对象构成的网格，我需要通过一种方式来描述网格的位置，它是一对整型，因此将其定义为元组：

```
class Board {
    typealias Point = (Int,Int)
    // ...
}
```

这么做的好处在于现在在代码中可以轻松使用 Point 了。比如，给定一个 Point，我可以获取到相应的 Piece：

```
func pieceAt(p:Point) -> Piece? {
    let (i,j) = p
    // ... error-checking goes here ...
    return self.grid[i][j]
}
```

拥有元素名的元组与函数参数列表之间的相似性并非巧合。参数列表就是个元组！事实上，每个函数都接收一个元组参数并返回一个元组。这样就可以向接收多个参数的函数传递单个元组了。比如，一个函数如下代码所示：

```
func f (i1:Int, _ i2:Int) -> () {}
```

f 的参数列表是个元组。这样，调用 f 时就可以将元组作为实参传递进去了：

```
let tuple = (1,2)
f(tuple)
```

在该示例中，f 没有外部参数名。如果函数有外部参数名，那么你可以向其传递一个带有具名元素的元组。如下面这个函数：

```
func f2 (i1 i1:Int, i2:Int) -> () {}
```

可以像下面这样调用：

```
let tuple = (i1:1, i2:2)
f2(tuple)
```

不过，出于我也尚不清楚的一些原因，以这种方式作为函数参数传递的元组必须是常量。如下代码将无法编译通过：

```
var tuple = (i1:1, i2:2)
f2(tuple) // compile error
```

与之类似，Void（不返回值的函数所返回的值类型）实际上是空元组的类型别名，这也是可以将其写成 () 的原因所在。

## 3.7.7 Optional

Optional 对象类型（枚举）用于包装任意类型的其他对象。单个 Optional 对象只能包装一个对象。此外，一个 Optional 对象还可能不包装任何对象。这正是 Optional 这个名字的由来，即可选：它可以包装其他对象，也可以不包装。你可以将 Optional 看作一种盒子，这个盒子可能是空的。

首先创建包装一个对象的 Optional。假设我们需要一个包装了字符串 `"howdy"` 的 Optional，一种创建方式就是使用 Optional 初始化器：

```
var stringMaybe = Optional("howdy")
```

如果使用 `print` 将 `stringMaybe` 的值输出到控制台上，那么我们会看到与相应的初始化器 `Optional("howdy")` 相同的表达式。

在声明与初始化后，`stringMaybe` 就拥有了类型，它既不是 String，也不是简单的 Optional，实际上它是包装了 String 的 Optional。这意味着只能将包装了 String 的 Optional（而不能是包装了其他类型的 Optional）赋给它。如下代码是合法的：

```
var stringMaybe = Optional("howdy")
stringMaybe = Optional("farewell")
```

如下代码则是不合法的：

```
var stringMaybe = Optional("howdy")
stringMaybe = Optional(123) // compile error
```

`Optional(123)` 是一个包装了 Int 的 Optional，如果需要包装了 String 的 Optional，那么你无法将其赋给它。

Optional 对于 Swift 非常重要，因此语言本身提供了使用它的特殊语法。创建 Optional 的常规方法并不是使用 Optional 初始化器（当然了，你可以这么做），而是将某个类型的值赋给或是传递给包装该类型的 Optional 引用。比如，如果 `stringMaybe` 的类型是包装了 String 的 Optional，那么你可以直接将字符串赋给它。这么做貌似不合法，但实际上却是可以的。结果就是被赋值的 String 被自动包装到了那个 Optional 中：

```
var stringMaybe = Optional("howdy")
stringMaybe = "farewell" // now stringMaybe is Optional("farewell")
```

我们还需要一种方式能够显式地将某个变量声明为包装了 String 的 Optional；否则就无法声明 Optional 类型的变量了，同时也无法声明 Optional 类型的参数。本质上，Optional 是个泛型，因此包装了 String 的 Optional 其实是 `Optional<String>`（第 4 章将

会介绍该语法）。不过，你不用非得这么写。Swift 语言支持 Optional 类型表示的语法糖：使用包装类型名，后跟一个问号。比如：

```
var stringMaybe : String?
```

这样就完全不需要使用 Optional 初始化器了。我可以将变量声明为包装 String 的 Optional，然后将一个 String 赋给它进行包装，一步就能搞定：

```
var stringMaybe : String? = "howdy"
```

事实上，这才是在 Swift 中创建 Optional 的常规方式。

在得到了包装某个具体类型的 Optional 后，你可以将其用在需要包装该类型的 Optional 的场合中，就像其他任何值一样。如果函数参数是一个包装了 String 的 Optional，那就可以将 stringMaybe 作为实参传递给该参数：

```
func optionalExpecter(s:String?) {}
let stringMaybe : String? = "howdy"
optionalExpecter(stringMaybe)
```

此外，在需要包装某个类型值的 Optional 时，你可以将被包装类型的值传递进去。这是因为参数传递就像是赋值：未包装的值会被隐式包装。比如，如果函数需要一个包装了 String 的 Optional，那么你可以传递一个 String 实参，它会在接收参数中被包装为 Optional：

```
func optionalExpecter(s:String?) {
    // ... here, s will be an Optional wrapping a String ...
    print(s)
}
optionalExpecter("howdy") // console prints: Optional("howdy")
```

但反过来则不行，你不能在需要被包装类型的地方使用包装该类型的 Optional，这么做将无法编译通过：

```
func realStringExpecter(s:String) {}
let stringMaybe : String? = "howdy"
realStringExpecter(stringMaybe) // compile error
```

错误消息是：“Value of optional type Optional<String> not unwrapped; did you mean to use! or??”。你经常会在 Swift 中看到这类消息！正如消息所表示的，如果需要被 Optional 包装的类型，但使用的却是 Optional，那就需要展开 Optional；也就是说，你需要进入 Optional 中，取出它包装的实际内容。下面就来介绍如何做到这一点。

### 1. 展开 Optional

之前已经介绍过将对象包装到 Optional 中的多种方法。不过相反的过程会怎样呢？如何展开 Optional 得到其中的对象呢？一种方式是使用展开运算符（或是强制展开运算符），它是个后缀感叹号，如下代码所示：

```
func realStringExpecter(s:String) {}
let stringMaybe : String? = "howdy"
realStringExpecter(stringMaybe!)
```

在上述代码中，stringMaybe! 语法表示进入 Optional stringMaybe 中，获取被包装的值，然后在该处使用这个值。由于 stringMaybe 是个包装了 String 的 Optional，因此里面的内容就是个 String。这正是 realStringExpecter 函数的参数类型！因此，我们可以将展开的 Optional 作为实参传递给 realStringExpecter。stringMaybe 是个包装了 String "howdy" 的 Optional，不过 stringMaybe! 却是 String "howdy"。

如果 Optional 包装了某个类型，那么你无法向其发送该类型所允许的消息；首先需要展开它。比如，我们想要获得 stringMaybe 的大写形式：

```
let stringMaybe : String? = "howdy"
let upper = stringMaybe.uppercaseString // compile error
```

解决方法就是展开 stringMaybe 获得里面的 String。可以通过展开运算符直接达成所愿：

```
let stringMaybe : String? = "howdy"
let upper = stringMaybe!.uppercaseString
```

如果需要使用 Optional 多次来获得其中包装的类型，并且每次都需要使用展开运算符获取里面的对象，那么代码很快就会变得非常冗长。比如，在 iOS 编程中，应用的窗口就是应用委托的 Optional UIWindow 属性（self.window）：

```
// self.window is an Optional wrapping a UIWindow
self.window = UIWindow()
self.window!.rootViewController = RootViewController()
self.window!.backgroundColor = UIColor.whiteColor()
self.window!.makeKeyAndVisible()
```

这么做太笨拙了，立刻可以想到的一种解决办法就是将展开值赋给包装类型的一个变量，然后使用该变量即可：

```
// self.window is an Optional wrapping a UIWindow
self.window = UIWindow()
let window = self.window!
// now window (not self.window) is a UIWindow, not an Optional
window.rootViewController = RootViewController()
window.backgroundColor = UIColor.whiteColor()
window.makeKeyAndVisible()
```

其实还有别的方法，现在就来介绍一下。

### 2. 隐式展开 Optional

Swift 提供了在需要被包装类型时使用 Optional 的另一种方式：你可以将 Optional 类型声明为隐式未包装的。这其实是另一种类型，即 ImplicitlyUnwrappedOptional。ImplicitlyUnwrappedOptional 是一种 Optional，不过编译器允许它使用一些特殊的魔法操作：在需要被包装类型时，可以直接使用它。你可以显式展开 ImplicitlyUnwrappedOptional，但不必这么做，因为它可以隐式展开（这也是其名字的由来）。比如：

```
func realStringExpecter(s:String) {}
var stringMaybe : ImplicitlyUnwrappedOptional<String> = "howdy"
realStringExpecter(stringMaybe) // no problem
```

与 Optional 一样，Swift 提供了语法糖来表示隐式展开的 Optional 类型。就像包装了 String 的 Optional 可以表示为 `String?` 一样，包装了 String 的隐式展开 Optional 可以表示为 `String!`。这样，我们可以将上述代码重写为（这也是实际开发中的写法）：

```
func realStringExpecter(s:String) {}
var stringMaybe : String! = "howdy"
realStringExpecter(stringMaybe)
```

请记住，隐式展开的 Optional 也是个 Optional，它只是个便捷的写法而已。通过将对象声明为隐式展开的 Optional，你告诉编译器，如果在需要被包装类型的地方使用了它，那么编译器能够将其展开。

就它们的类型来说，常规 Optional 会包装某个类型（如 `String?`），而隐式展开的 Optional 也包装了相同的类型（如 `String!`），它们之间是可以互换的：在需要其中一个的地方都可以使用另外一个。

### 3. 魔法词 nil

我一直在说 Optional 会包含一个包装值，不过不包含任何包装值的 Optional 是什么呢？正如我之前所说的，这种 Optional 也是合法的实体；事实上，这两种情况构成了完整的 Optional。

你需要通过一种方式来判断一个 Optional 是否包含了包装值，以及指定没有包装值的 Optional。Swift 让这一切变得异常简单，这是通过一个特殊的关键字 `nil` 来实现的：

*判断一个 Optional 是否包含了包装值*

测试 Optional 是否与 `nil` 相等。如果相等，那么该 Optional 就是空的。一个空的 Optional 在控制台中也会打印出 `nil`。

*指定没有包装值的 Optional*

在需要 Optional 类型时赋值或传递一个 nil，结果就是期望类型的 Optional，它不包含包装值。

比如：

```
var stringMaybe : String? = "Howdy"
print(stringMaybe) // Optional("Howdy")
if stringMaybe == nil {
    print("it is empty") // does not print
}
stringMaybe = nil
print(stringMaybe) // nil
if stringMaybe == nil {
    print("it is empty") // prints
}
```

魔法词 nil 可以表达这个概念：一个 Optional 包装了恰当的类型，但实际上不包含该类型的任何对象。显然，这是非常方便的；你可以充分利用它。不过重要的是，你要理解它只是个魔法而已：Swift 中的 nil 并不是对象，也不是值。它只不过是个简便写法而已。你可以认为这个简便写法就是真正存在的。比如，我可以说某个东西是 nil。但实际上，没有什么东西会是 nil；nil 并不是具体的事物。我的意思是这个东西相当于 nil（因为它是个没有包装任何东西的 Optional）。

没有包装对象的 Optional 的实际值是 Optional.None，包装了 String 的 Optional 里面如果没有 String 对象，那么其实际值是 Optional<String>.None。不过在实际开发中，你是不需要这么编写代码的，因为只需写成 nil 即可。第 4 章将会介绍这些表达式的真正含义。

由于类型为 Optional 的变量可能为 nil，所以 Swift 使用了一种特殊的初始化规则：如果变量（var）的类型为 Optional，那么其值自动就为 nil。如下代码是合法的：

```
func optionalExpecter(s:String?) {}
var stringMaybe : String?
optionalExpecter(stringMaybe)
```

上述代码很有趣，因为看起来好像是不合法的。我们声明了一个变量 stringMaybe，但却没有给它赋值。不过却将其传递给了一个函数，就好像它是有值一样。这是因为它的的确确是有值的。该变量会被隐式初始化为 nil。在 Swift 中，类型为 Optional 的变量（var）是唯一一种会被隐式初始化的变量类型。

现在来谈谈也许是 Swift 中最为重要的一个原则：不能展开不包含任何东西的 Optional（即

等于 nil 的 Optional）。这种 Optional 不包含任何东西；没有什么需要展开的。事实上，显式展开不包含任何东西的 Optional 会造成程序在运行时崩溃。

```
var stringMaybe : String?
let s = stringMaybe! // crash
```

崩溃消息的内容是："Fatal error: unexpectedly found nil while unwrapping an Optional value"。习惯吧，因为你会经常看到这个消息。这是个很容易犯的错误。事实上，展开一个不包含值的 Optional 可能是导致 Swift 程序崩溃最常见的一个原因，你应该好好利用这种崩溃的情况。事实上，如果某个 Optional 中不包含值，那么你希望应用崩溃，因为这个 Optional 本应该包含值的，既然不包含值，那就说明其他地方出错了。

要想消除这种崩溃的情况，你需要确保 Optional 中包含值，如果不包含，那么请不要将其展开。显而易见的一种做法是首先将其与 nil 进行比较：

```
var stringMaybe : String?
// ... stringMaybe might be assigned a real value here ...
if stringMaybe != nil {
    let s = stringMaybe!
    // ...
}
```

### 4. Optional 链

有时，你想向被 Optional 所包装的值发送消息。要想做到这一点，你可以将 Optional 展开。如下面这个示例：

```
let stringMaybe : String? = "howdy"
let upper = stringMaybe!.uppercaseString
```

这种形式的代码叫作 Optional 链。在点符号链的中间，你已经将 Optional 展开了。

如果不展开，那就无法向 Optional 发送消息。Optional 本身并不会响应任何消息（实际情况是，它们会响应一些消息，不过非常少，你基本上不会用到——它们也不是 Optional 里面的对象所要响应的消息）。如果向 Optional 发送了本该发送给里面的对象的消息，那么编译器就会报错：

```
let stringMaybe : String? = "howdy"
let upper = stringMaybe.uppercaseString // compile error
```

不过，我们已经看到，如果展开一个不包含对象的 Optional，那么应用将会崩溃。这样，如果不确定一个 Optional 是否包含了对象该怎么办呢？在这种情况下，如何向一个 Optional 发送消息呢？ Swift 针对这个目的提供了一个特殊的简写形式。要想安全地向可能为空的 Optional 发送消息，你可以展开这个 Optional。在这种情况下，请通过问号后缀运算符而非感叹号将 Optional 展开：

```
var stringMaybe : String?
// ... stringMaybe might be assigned a real value here ...
let upper = stringMaybe?.uppercaseString
```

这是个 Optional 链，你通过问号展开了该 Optional。通过使用该符号，你可以有条件地将 Optional 展开。条件就是一种安全保障；会帮助我们执行与 `nil` 的比较。代码表示的意思是：”如果 `stringMaybe` 包含了一个 String，那么将其展开并向其发送 `uppercaseString` 消息；如果不包含（也就是说等于 `nil`），那就不要展开它，也不要向其发送任何消息”。

这种代码是个双刃剑。一方面，如果 `stringMaybe` 为 `nil`，那么应用在运行期不会崩溃；另一方面，如果 `stringMaybe` 为 `nil`，那么这一行代码其实什么都没做，并不会得到任何大写字符串。

不过现在又有了一个新问题。在上述代码中，我们使用一个表达式（该表达式会发送 `uppercaseString` 消息）初始化了变量 `upper`。结果却是 `uppercaseString` 这条消息可能发送了，也可能根本就没有发送。那么，`upper` 被初始化成了什么呢？

为了处理这种情况，Swift 有一个特殊的原则。如果一个 Optional 链包含了可选的展开 Optional，并且如果该 Optional 链生成了一个值，那么该值本身就会被包装到 Optional 中。这样，`upper` 的类型就是包装了 String 的 Optional。这么做非常棒，因为它涵盖了两种可能的情况。首先，假设 `stringMaybe` 包含了一个 String：

```
var stringMaybe : String?
stringMaybe = "howdy"
let upper = stringMaybe?.uppercaseString // upper is a String?
```

上述代码执行后，`upper` 并不是一个 String；它不是 `"HOWDY"`。实际上，它是个包装了 `"HOWDY"` 的 Optional！另一方面，如果尝试展开 Optional 的操作失败了，那么该 Optional 链会返回 `nil`：

```
var stringMaybe : String?
let upper = stringMaybe?.uppercaseString // upper is a nil String?
```

以这种方式展开 Optional 是优雅且安全的；不过请考虑一下执行结果。一方面，即便 `stringMaybe` 是 `nil`，应用也不会在运行时崩溃。另一方面，这么做并不比之前的做法更好：我们实际上得到了另一个 Optional！无论 `stringMaybe` 是否为 `nil`，`upper` 的类型都是一个包装了 String 的 Optional，为了使用其中的 String，你需要展开 `upper`。我们不知道 `upper` 是否为 `nil`，因此会遇到与之前一样的问题——需要确保能够安全展开 `upper`，并且不会意外展开一个空的 Optional。

更长的 Optional 链也是合法的。它们的工作方式与你想象的完全一致：无论链中要展开多少个 Optional，如果其中一个被展开了，那么整个表达式就会生成一个 Optional，它包装的是 Optional 被正常展开后所得到的类型，并且在这个过程中会安全地失败。比如：

```
// self.window is a UIWindow?
let f = self.window?.rootViewController?.view.frame
```

视图的 `frame` 属性是个 CGRect。不过在上述代码执行后，`f` 并非 CGRect，它是个包装了 CGRect 的 Optional。如果链中的任何一个展开失败了（由于要展开的 Optional 为 `nil`），那么整个链就会返回 `nil` 以表示失败。

注意到上述代码并没有嵌套使用 Optional；并不会因为链中有两个 Optional 就生成包装到 Optional 中的 CGRect，然后这个 Optional 又包装到另一个 Optional 中。不过，出于其他一些原因，我们可以生成包装到另一个 Optional 中的 Optional，第 4 章将会给出一个示例。

如果涉及可选展开 Optional 的 Optional 链生成了一个结果，那么你可以通过检查结果来判断链中的所有 Optional 是否可以安全展开：如果它不为 `nil`，那么一切都可以成功展开。但如果没有得到结果呢？比如：

```
self.window?.rootViewController = UIViewController()
```

现在真是进退维谷。程序当然是不会崩溃的了；如果 `self.window` 为 `nil`，那么它不会展开，因此安全。但如果 `self.window` 为 `nil`，我们也没办法为窗口赋一个根视图控制器了！最好要知道该 Optional 链的展开是否是成功的。幸好，我们可以通过一个技巧来实现这个目标。在 Swift 中，不返回值的语句都会返回一个 Void。因此，对拥有可选展开的 Optional 的赋值会返回一个包装了 Void 的 Optional；你可以捕获到这个 Optional，这意味着你可以判断它是否为 `nil`；如果不为 `nil`，那么赋值就成功了。比如：

```
let ok : Void? = self.window?.rootViewController = UIViewController()
if ok != nil {
    // it worked
}
```

显然，无须显式地将包装了 `Void` 的 Optional 赋给变量；你可以在一步中完成捕获和与 `nil` 的比较两件事：

```
if (self.window?.rootViewController = UIViewController()) != nil {
    // it worked
}
```

如果函数调用返回一个 Optional，那么你可以展开结果并使用，无须先捕获结果，可以直接展开，方式是在函数调用后使用一个感叹号或问号（即在右圆括号后面）。这与之前所做的别无二致，只不过相对于 Optional 属性或变量来说，这里使用的是返回 Optional 的函数调用。比如：

```
class Dog {
    var noise : String?
    func speak() -> String? {
        return self.noise
    }
}
let d = Dog()
let bigname = d.speak()?.uppercaseString
```

最后不要忘记，bigname 并非 String，它是个包装了 String 的 Optional。

第 5 章介绍流程控制时还会继续介绍检查 Optional 是否为 nil 的其他 Swift 语法。

！与？后缀运算符（分别表示无条件与有条件展开 Optional）和表示 Optional 类型时与类型名搭配使用的！和？语法糖（如 String? 表示包装了 String 的 Optional，String! 表示隐式展开包装了 String 的 Optional）没有任何关系。二者之间表面上的相似性迷惑了很多初学者。

### 5. 与 Optional 的比较

在与除 nil 的其他值比较时，Optional 会特殊一些：比较的是包装值而非 Optional 本身。比如，如下代码是合法的：

```
let s : String? = "Howdy"
if s == "Howdy" { // ... they _are_ equal!
```

上述代码看起来不可行，但实际上却是可行的；展开一个 Optional，但却只是为了将其包装值与其他值进行比较，这么做非常麻烦（特别是，你还得先检查 Optional 是否为 nil）。相对于将 Optional 本身与 "Howdy" 进行比较，Swift 会自动（且安全）将其包装值（如果有）与 "Howdy" 比较，而且比较成功了。如果被包装值不是 "Howdy"，那么比较就会失败。如果没有被包装值（s 为 nil），那么比较也会失败，这非常安全！这样，你就可以将 s 与 nil 或 String 进行比较了，在所有情况下比较都可以正确进行。

同样地，如果 Optional 包装了可以使用大于和小于运算符类型的值，那么这些运算符也可以直接应用到 Optional 上：

```
let i : Int? = 2
if i < 3 { // ... it _is_ less!
```

## 6. 为何使用 Optional？

既然已经知道如何使用 Optional，那么你可能想知道为何要使用 Optional。Swift 为何要提供 Optional，好处又是什么呢？

Optional 一个非常重要的目的就是提供可与 Objective-C 交换的对象值。在 Objective-C 中，任何对象引用都可能为 `nil`。因此需要通过一种方式向 Objective-C 发送 `nil` 并接收来自 Objective-C 的 `nil`。Swift Optional 就提供了这一方式。

Swift 会帮助你正确使用 Cocoa API 中的恰当类型。比如，考虑 UIView 的 `backgroundColor` 属性，它是个 UIColor，不过可能为 `nil`，你也可以将其设为 `nil`。这样，其类型就是 `UIColor?`。为了设置这个值，你无须直接使用 Optional。请记住，将被包装值赋给 Optional 是合法的，因为系统会将其包装起来。这样，你可以将 `myView.backgroundColor` 设为 UIColor 或 `nil`。不过，如果获得了 UIView 的 `backgroundColor`，那么你就有了一个包装 UIColor 的 Optional，你需要清楚这一事实，否则就可能出现奇怪的结果：

```
let v = UIView()
let c = v.backgroundColor
let c2 = c.colorWithAlphaComponent(0.5) // compile error
```

上述代码向 `c` 发送了 `colorWithAlphaComponent` 消息，就好像它是个 UIColor 一样。它其实不是 UIColor，而是包装了 UIColor 的 Optional。Xcode 会在这种情况下帮助你；如果使用代码完成输入了 `colorWithAlphaComponent` 方法的名字，那么 Xcode 会在 `c` 后面插入一个问号，这样就会展开 Optional 并得到合法的代码：

```
let v = UIView()
let c = v.backgroundColor
let c2 = c?.colorWithAlphaComponent(0.5)
```

不过在大多数情况下，Cocoa 对象类型都不会被标记为 Optional。这是因为，虽然从理论上说，它可能为 `nil`（因为任何 Objective-C 对象引用都可能为 `nil`）；但实际上，它不会。Swift 会将值看作对象类型本身。这个魔法是通过对 Cocoa API（又叫作审计）采取一定的处理实现的。在 Swift 早期公开版中（2014 年 6 月），从 Cocoa 接收到的所有对象值实际上都是 Optional 类型（通常是隐式展开的 Optional）。不过后来，Apple 花了大力气调整 API，从而去除了那些不需要作为 Optional 的 Optional。

在一些情况下，你还是会遇到来自于Cocoa的隐式展开Optional。比如，在本书编写之际，NSBundle 方法 `loadNibNamed:owner:options:` 的 API 如下代码所示：

```
func loadNibNamed(name: String!,
    owner: AnyObject!,
    options: [NSObject : AnyObject]!)
    -> [AnyObject]!
```

这些隐式展开Optional表明这个头文件还没有被处理过。它们无法精确表示出现状(比如，你永远不会将 `nil` 作为第一个参数传递进去），不过问题倒也不太大。

使用Optional的另一个重要目的在于推断出实例属性的初始化。如果变量(通过 `var` 声明)类型是 Optional，那么即便没有对其初始化，它也会有一个值，即 `nil`。如果你知道某个对象将会具有值，但不是现在，那么 Optional 就非常方便了。在实际的 iOS 编程中，一个典型示例就是插座变量，它是指向界面中某个东西（如按钮）的一个引用：

```
class ViewController: UIViewController {
    @IBOutlet var myButton: UIButton!
    // ...
}
```

现在可以忽略 `@IBOutlet` 指令，它是对 Xcode 的一个内部提示（第 7 章将会介绍）。重要之处在于属性 `myButton` 在 ViewController 实例首次创建出来之后还没有值，但在视图控制器的视图加载后，`myButton` 值会被设定好，这样它就会指向界面中实际的 UIButton 对象了。因此，该变量的类型是个隐式展开 Optional。之所以是 Optional，是因为当 ViewController 实例首次创建出来后，`myButton` 需要一个占位符值。它是个隐式展开 Optional，这样代码就可以将 `self.myButton` 当作对实际的 UIButton 的引用，不必强调它是个 Optional 了。

另一种相关情况是当一个变量（通常是实例属性）所表示的数据需要一些时间才能获取到该怎么办。比如，在我写的 Albumen 应用中，当应用启动时，我会创建一个根视图控制器的实例。我还想获取用户音乐库的数据，然后将数据存储到根视图控制器实例的实例属性中。不过获取这些数据需要时间。因此，需要先实例化根视图控制器，然后再获取数据，因为如果在实例化根视图控制器之前获取数据，那么应用的启动时间就会很长，这种延迟会很明显，甚至可能会造成应用崩溃（因为 iOS 不允许过长的启动时间）。因此，数据属性都是 Optional 类型的；在获取到数据之前，它们都是 `nil`；当数据获取到后，它们才会被赋予“真正的”值：

```
class RootViewController : UITableViewController {
    var albums : [MPMediaItemCollection]! = nil // initialized to nil
    // ...
```

最后，Optional 最重要的用处之一就是可以将值标记为空或使用不正确的值，上述代码已经很好地说明了这一点。当 Albumen 应用启动时，它会显示出一个表格，列出了用户所有的音乐专辑。不过在启动时，数据尚未取得。展示表格的代码会检查 `albums` 是否为 `nil`；如果是，那就显示一个空的表格。在获取到数据后，表格会再一次展示数据。这次，展示表格的代码会发现 `albums` 不为 `nil`，而是包含了实际的数据，它现在就会将数据显示出来。借助 Optional，`albums` 可以存储数据，也可以表示其中没有数据。

很多内建的 Swift 函数都以类似的方式使用 Optional，比如，之前提到的将 String 转换为 Int：

```
let s = "31"
let i = Int(s) // Optional(31)
```

从 String 初始化 Int 会返回一个 Optional，因为转换可能会失败。如果 `s` 是 `"howdy"`，那么它就不是数字。这样，返回的类型就不是 Int，因为没有一个 Int 可以表示“我没有找到 Int”这一含义。返回一个 Optional 优雅地解决了这一问题：`nil` 表示我没有找到 Int，否则实际的 Int 结果就会位于 Optional 所包装的对象中。

Swift 在这方面要比 Objective-C 更加聪明。如果引用是个对象，那么 Objective-C 可以返回 `nil` 来报告失败；不过在 Objective-C 中，并非一切都是对象。因此，很多重要的 Cocoa 方法都会返回一个特殊值来表示失败，你需要知道这一点，并记得对其进行测试。比如，NSString 的 `rangeOfString:` 可能在目标字符串中找不到给定的子字符串；在这种情况下，它会返回一个长度为 0 的 NSRange，其位置（索引）是个特殊值，即 `NSNotFound`，它实际上是个非常大的负数。幸好，这种特殊值已经内建在 Swift 对 Cocoa API 的桥接中：Swift 会将返回值的类型作为包装了 Range 的 Optional，如果 `rangeOfString:` 返回一个 NSRange，其位置是 `NSNotFound`，那么 Swift 会将其表示为 `nil`。

不过，并非 Swift–Cocoa 桥接的每一处都如此便捷。如果调用了 NSArray 的 `indexOfObject:`，那么结果就是个 Int，而不是包装了 Int 的 Optional；结果还有可能是 NSNotFound，你要记得对其进行测试：

```
let arr = [1,2,3]
let ix = (arr as NSArray).indexOfObject(4)
if ix == NSNotFound { // ...
```

另一种做法是使用 Swift，然后调用 `indexOf` 方法，它会返回一个 Optional：

```
let arr = [1,2,3]
let ix = arr.indexOf(4)
if ix == nil { // ...
```

# 第 4 章

# 对象类型

第 3 章介绍了一些内建的对象类型，不过还没有谈及对象类型本身。正如我在第 1 章所说的，Swift 对象类型有 3 种风格：枚举、结构体与类。它们之间的差别是什么？如何创建自定义的对象类型？这正是本章所要回答的问题。

我首先会大体介绍一下对象类型，然后详细介绍对象类型的 3 种风格。接下来，我会介绍 Swift 所提供的用于增强对象类型灵活性的 3 种方式：协议、泛型与扩展。最后，我会以 3 种保护类型与 3 种集合类型来结束对 Swift 内建类型的介绍。

## 4.1 对象类型声明与特性

对象类型是通过一种对象类型风格（enum、struct 与 class）、对象类型的名字（应该以一个大写字母开头）和一对花括号进行声明的：

```
class Manny {
}
struct Moe {
}
enum Jack {
}
```

对象类型声明可以出现在任何地方：在文件顶部、在另一个对象类型声明顶部，或在函数体中。对象类型相对于其他代码的可见性（作用域）与可用性取决于它声明的位置（参见第 1 章）：

- 在默认情况下，声明在文件顶部的对象类型对于项目（模块）中的所有文件都可见，对象类型通常都会声明在这个地方。

- 有时需要在其他类型的声明中声明一个类型，从而赋予它一个命名空间，这叫作嵌套类型。
- 声明在函数体中的对象类型只会在外围花括号的作用域内存活；这种声明是合法的，但并不常见。

任何对象类型声明的花括号中都可能包含如下内容：

*初始化器*

对象类型仅仅是一个对象的类型而已。声明对象类型的目的通常是（但不总是这样）创建该类型的实际对象，即实例。初始化器是个特殊的函数，它的声明和调用方式都与众不同，你可以通过它创建对象。

*属性*

声明在对象类型声明顶部的变量就是属性。在默认情况下，它是个实例属性。实例属性的作用域是实例：可以通过该类型的特定实例来访问它，该类型的每个实例的实例属性值可能都不同。

此外，属性还可以是静态 / 类属性。对于枚举或结构体来说，它是通过关键字 `static` 声明的；对于类来说，它是通过关键字 `class` 声明的。这种属性属于对象类型本身；可以通过类型来访问它，它只有一个值，与所属类型关联。

*方法*

声明在对象类型声明顶部的函数就是方法。在默认情况下，方法都是实例方法；可以通过向该类型的特定实例发送消息来调用它。在实例方法内部，`self` 就是实例本身。

此外，函数还可以是静态 / 类方法。对于枚举或结构体来说，它是通过关键字 `static` 声明的；对于类来说，它是通过关键字 `class` 声明的。可以通过向类型发送消息来调用它。在静态 / 类方法内部，`self` 就是类型。

*下标*

下标是一种特殊类型的实例方法，可以通过向实例引用附加方括号来调用它。

*对象类型声明*

对象类型声明还可以包含对象类型声明，即嵌套类型。从外部对象类型内部看，嵌套类型位于其作用域中；从外部对象类型外部看，嵌套类型必须要通过外部对象类型才能使用。这样，外部对象类型是嵌套类型的命名空间。

## 4.1.1 初始化器

初始化器是一个函数，用来生成对象类型的一个实例。严格来说，它是个静态 / 类方法，因为它是通过对象类型调用的。调用时通常会使用特殊的语法：类型名后面直接跟着一

对圆括号，就好像类型本身是函数一样。当调用初始化器时，新的实例会被创建出来并作为结果返回。你通常会用到返回的实例，比如，将其赋给变量，从而将其保存起来并在后续代码中使用它。

比如，假设有一个 Dog 类：

```
class Dog {
}
```

接下来可以创建一个 Dog 实例：

```
Dog()
```

上述代码虽然合法，但却没什么用，甚至连编译器都会发出警告。我们创建了一个 Dog 实例，但却没有引用该实例。如果没有引用，那么 Dog 实例创建出来后立刻就会消亡。一般来说，我们会这么做：

```
let fido = Dog()
```

现在，只要变量 fido 存在，Dog 实例就会存在（参见第 3 章），变量 fido 引用了 Dog 实例，这样就可以使用它了。

注意，虽然 Dog 类没有声明任何初始化器，但 Dog() 还是调用了一个初始化器！原因在于对象类型会有隐式初始化器。这样你就不必费力编写自定义的初始化器了。不过，你还是可以编写自定义的初始化器，而且会经常这么做。

初始化器是一种函数，其声明语法与函数非常像。要想声明初始化器，你需要使用关键字 init，后跟一个参数列表，然后是包含代码的花括号。一个对象类型可以有多个初始化器，由参数进行区分。在默认情况下，参数名（包括第一个参数）都是外化的（当然了，你可以在参数名前通过下划线阻止这一点）。参数常常用于设置实例属性的值。

比如，下面是拥有两个实例属性的 Dog 类：name（String）与 license（Int）。我们为这些实例属性赋予了默认值，这些默认值起到了占位符的作用——一个空字符串和一个数字 0。接下来声明了 3 个初始化器，这样调用者就可以通过 3 种不同方式来创建 Dog 实例了：提供一个名字、提供一个登记号，或提供这二者。在每个初始化器中，所提供的参数都用于设置相应属性的值：

```
class Dog {
    var name = ""
    var license = 0
    init(name:String) {
        self.name = name
    }
    init(license:Int) {
```

```
        self.license = license
    }
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
}
```

注意，在上述代码的每个初始化器中，我为每个参数起了与其相应的属性相同的名字，这么做只是一种编程风格而已。在每个初始化器中，我可以通过 `self` 访问属性将参数与属性区分开。

上述声明的结果就是我可以通过 3 种不同方式来创建 Dog：

```
let fido = Dog(name:"Fido")
let rover = Dog(license:1234)
let spot = Dog(name:"Spot", license:1357)
```

我无法做的是不使用初始化器参数创建 Dog 实例。我编写了初始化器，因此隐式初始化器就不复存在了。如下代码是不合法的：

```
let puff = Dog() // compile error
```

当然，可以显式声明一个不带参数的初始化器，这样上述代码就合法了：

```
class Dog {
    var name = ""
    var license = 0
    init() {
    }
    init(name:String) {
        self.name = name
    }
    init(license:Int) {
        self.license = license
    }
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
}
```

其实不需要这么多初始化器，因为初始化器是个函数，函数的参数可以有默认值。这样，我可以将所有代码放到单个初始化器中，如下代码所示：

```
class Dog {
    var name = ""
    var license = 0
    init(name:String = "", license:Int = 0) {
        self.name = name
```

```
            self.license = license
        }
    }
```

现在依然可以通过 4 种不同的方式创建一个 Dog 实例：

```
let fido = Dog(name:"Fido")
let rover = Dog(license:1234)
let spot = Dog(name:"Spot", license:1357)
let puff = Dog()
```

现在来看看有趣的地方。在属性声明中，我可以去掉默认初始值的赋值（只要显式声明每个属性的类型即可）：

```
class Dog {
    var name : String // no default value!
    var license : Int // no default value!
    init(name:String = "", license:Int = 0) {
        self.name = name
        self.license = license
    }
}
```

上述代码是合法的，也很常见，因为初始化器执行的确实是初始化工作！换言之，我无须在声明中为属性赋初值，而是在所有的初始化器中为它们赋初值。通过这种方式，我可以保证当实例创建出来后，所有实例属性都有值了，这正是重要之处。相反，当实例创建出来后，没有初值的实例属性是不合法的。属性要么在声明中初始化，要么被每个初始化器初始化，否则编译器会报错。

Swift 编译器认为所有实例属性都要被恰当初始化是 Swift 的一个重要特性（这与 Objective-C 相反，它的实例属性可以没有初始化，这常常会导致后续一些奇怪的错误）。不要挑战编译器，请适应它。编译器会通过错误消息（“Return from initializer without initializing all stored properties”）帮助你，直到初始化器初始化了所有实例属性。

```
class Dog {
    var name : String
    var license : Int
    init(name:String = "") {
        self.name = name // compile error
    }
}
```

由于在初始化器中设置实例属性算是初始化，所以即便实例属性是通过 `let` 声明的常量也是合法的：

```
class Dog {
    let name : String
    let license : Int
```

```
    init(name:String = "", license:Int = 0) {
        self.name = name
        self.license = license
    }
}
```

在这个示例中，我们没有对初始化器做任何限制：调用者可以在不提供 name 或 license 实参的情况下实例化 Dog。但通常，初始化器的目的正好相反：我们会强制调用者在实例化时提供所有必要的信息。在实际情况下，Dog 类更可能像是下面这样：

```
class Dog {
    let name : String
    let license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
}
```

在上述代码中，Dog 有一个 name 和一个 license，这两个变量的值是在实例化时提供的（它们没有默认值），并且之后这两个值就无法再改变了（这些属性是常量）。通过这种方式，我们强制要求每个 Dog 都必须要有一个有意义的名字与许可证号。现在，创建 Dog 只有一种方式：

```
let spot = Dog(name:"Spot", license:1357)
```

### 1. Optional 属性

有时，在初始化时并没有可赋给实例属性的有意义的默认值。比如，也许直到实例创建出来一段时间后才能获取到属性的初始值。这种情况与所有实例属性要么在声明中，要么通过初始化器进行初始化的要求相冲突。当然，你可以通过给实例属性赋一个默认初始值来绕过这个问题，不过它并非“真正的”值。

正如我在第 3 章所提及的，这个问题合理且常见的解决方案是使用 `var` 将实例属性声明为 Optional 类型。值为 `nil` 的 Optional 表示没有提供“真正的”值；Optional `var` 会被自动初始化为 `nil`。这样，代码就可以比较该实例属性与 `nil`，如果为 `nil`，那就不使用该属性。稍后，属性会被赋予“真正的”值。当然，这个值现在被包装到了一个 Optional 中；但如果将其声明为隐式展开 Optional，那么你还可以直接使用被包装的值，无须显式将其展开（就好像它根本就不是 Optional 一样），如果确定，那就可以这样做：

```
// this property will be set automatically when the nib loads
@IBOutlet var myButton: UIButton!
// this property will be set after time-consuming gathering of data
var albums : [MPMediaItemCollection]!
```

## 2. 引用 self

除了设置实例属性，初始化器不能引用 self，无论显式还是隐式都不可以，除非所有实例属性都完成了初始化。这个原则可以确保实例在使用前已经完全构建完毕。比如，如下代码是不合法的：

```
struct Cat {
    var name : String
    var license : Int
    init(name:String, license:Int) {
        self.name = name
        meow() // too soon - compile error
        self.license = license
    }
    func meow() {
        print("meow")
    }
}
```

对实例方法 meow 的调用隐式引用了 self，它表示 self.meow()。初始化器可以这么做，但需要在初始化完所有未初始化的属性后才可以。对实例方法 meow 的调用只需要下移一行即可，这样在完成了 name 与 license 的初始化后就可以调用它了。

## 3. 委托初始化器

对象类型中的初始化器可以通过语法 self.init(...) 调用其他初始化器。调用其他初始化器的初始化器叫作委托初始化器。当一个初始化器委托另一个初始化器时，被委托的初始化器必须要先完成实例的初始化，接下来委托初始化器才能使用初始化完毕的实例，可以再次设置被委托初始化器已经设定的 var 属性。

委托初始化器看起来好像是之前介绍的关于 self 的规则的一个例外。但实际上并非如此，因为它要通过 self 才能委托，而且委托会导致所有实例属性都被初始化。事实上，关于委托初始化器使用 self 的规则要更加严格：委托初始化器不能引用 self，也不能设置属性，直到对其他初始化器的调用完毕后才可以。比如：

```
struct Digit {
    var number : Int
    var meaningOfLife : Bool
    init(number:Int) {
        self.number = number
        self.meaningOfLife = false
    }
    init() { // this is a delegating initializer
        self.init(number:42)
        self.meaningOfLife = true
    }
}
```

此外，委托初始化器不能设置不可变属性（即 let 变量）。这是因为只有在调用了其他初始化器后它才可以引用属性，而这时实例已经构建完毕——初始化已经结束，通往不可变属性的初始化之门已经关闭。这样，如果 meaningOfLife 是通过 let 声明的，那么上述代码就不合法，因为第 2 个初始化器是委托初始化器，它无法设置不可变属性。

请注意，不要递归委托！如果让初始化器委托给自身，或是创建了循环委托初始化器，那么编译器不会报错（我认为这是个 Bug），不过运行着的应用会挂起。比如，不要这么做：

```
struct Digit { // do not do this!
    var number : Int = 100
    init(value:Int) {
        self.init(number:value)
    }
    init(number:Int) {
        self.init(value:number)
    }
}
```

### 4. 可失败初始化器

初始化器可以返回一个包装新实例的 Optional。通过这种方式，可以返回 nil 来表示失败。具备这种行为的初始化器叫作可失败初始化器。在声明时要想将某个初始化器标记为可失败的，请在关键字 init 后面放置一个问号（对于隐式展开 Optional，放置一个感叹号）。如果可失败初始化器需要返回 nil，请显式写明 return nil。判断返回的 Optional 与 nil 是否相等是调用者的事，请展开它，然后比较，与其他 Optional 的做法一样。

下面这个版本的 Dog 有一个返回隐式展开 Optional 的初始化器，如果 name: 或是 license: 实参无效，那么它会返回 nil：

```
class Dog {
    let name : String
    let license : Int
    init!(name:String, license:Int) {
        self.name = name
        self.license = license
        if name.isEmpty {
            return nil
        }
        if license <= 0 {
            return nil
        }
    }
}
```

返回值的类型是 Dog，Optional 会隐式展开，因此以这种方式实例化 Dog 的调用者可以直接使用该结果，就好像它是个 Dog 实例一样。不过如果返回的是 nil，那么调用者访问 Dog 实例的成员就会导致程序在运行时崩溃：

```
let fido = Dog(name:"", license:0)
let name = fido.name // crash
```

按照惯例，Cocoa 与 Objective-C 会从初始化器中返回 nil 来表示失败；如果初始化真的可能失败，那么这种初始化器 API 已经被转换为了 Swift 可失败初始化器。比如，UIImage 初始化器 init?(named:) 就是个可失败初始化器，因为给定的名字可能并不表示一张图片。它不会隐式展开，因此结果值是一个 UIImage?，并且在使用前需要展开（不过，大多数 Objective-C 初始化器都没有被桥接为可失败初始化器，即便从理论上说，任何 Objective-C 初始化器都可能返回 nil）。

## 4.1.2 属性

属性是个变量，它声明在对象类型声明的顶部。这意味着第 3 章所介绍的关于变量的一切都适用于属性。属性拥有确定的类型；可以通过 var 或 let 声明属性，它可以是存储变量，也可以是计算变量；它也可以拥有 Setter 观察者。实例属性也可以声明为 lazy。

存储实例属性必须要赋予一个初始值。不过，正如我之前说过的，这不一定非得通过声明中的赋值来实现；也可以通过初始化器。Setter 观察者在属性的初始化过程中是不会被调用的。

初始化属性的代码不能获取实例属性，也不能调用实例方法。这种行为需要一个对 self 的显式或隐式引用；在初始化过程中还不存在 self，self 是在初始化过程中所创建的。这个错误所导致的 Swift 编译错误消息令人感到很费解。比如，如下代码是不合法的（删除对 self 的显式引用也不行）：

```
class Moi {
    let first = "Matt"
    let last = "Neuburg"
    let whole = self.first + " " + self.last // compile error
}
```

一种解决办法就是将 whole 作为一个计算属性：

```
class Moi {
    let first = "Matt"
    let last = "Neuburg"
    var whole : String {
        return self.first + " " + self.last
    }
}
```

这是合法的，因为计算直到 self 存在后才会执行。另一个解决办法是将 whole 声明为 lazy：

```
class Moi {
    let first = "Matt"
    let last = "Neuburg"
    lazy var whole : String = self.first + " " + self.last
}
```

这也是合法的，因为直到 `self` 存在后对它的引用才会执行。与之类似，属性初始化器是无法调用实例方法的，不过，计算属性却可以，`lazy` 属性也可以。

正如第 3 章所述，变量的初始化器可以包含多行代码，前提是将其写成定义与调用匿名函数。如果变量是实例属性，并且代码引用了其他的实例属性或实例方法，那么变量就可以声明为 `lazy`：

```
class Moi {
    let first = "Matt"
    let last = "Neuburg"
    lazy var whole : String = {
        var s = self.first
        s.appendContentsOf(" ")
        s.appendContentsOf(self.last)
        return s
    }()
}
```

如果属性是实例属性（默认情况），那么只能通过实例来访问它，并且对于每个实例来说，其值都是独立的。比如，再来看看这个 Dog 类：

```
class Dog {
    let name : String
    let license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
}
```

这个 Dog 类有一个 `name` 实例属性，接下来可以通过两个不同的 `name` 值创建两个不同的 Dog 实例，并通过实例访问每个 Dog 的 `name` 属性：

```
let fido = Dog(name:"Fido", license:1234)
let spot = Dog(name:"Spot", license:1357)
let aName = fido.name // "Fido"
let anotherName = spot.name // "Spot"
```

另一方面，静态 / 类属性是通过类型访问的，其作用域是类型，这意味着它是全局且唯一的，这里使用一个结构体作为示例：

```
struct Greeting {
    static let friendly = "hello there"
```

```
    static let hostile = "go away"
}
```

现在，其他地方的代码可以获取到 Greeting.friendly 与 Greeting.hostile 的值。该示例非常有代表性：不变的静态 / 类属性可以作为一种非常便捷且有效的方式为代码提供命名空间下的常量。

与实例属性不同，静态属性可以通过对其他静态属性的引用进行实例化，这是因为静态属性初始化器是延迟的（参见第 3 章）：

```
struct Greeting {
    static let friendly = "hello there"
    static let hostile = "go away"
    static let ambivalent = friendly + " but " + hostile
}
```

注意到上述代码中没有使用 self。在静态 / 类代码中，self 表示类型本身。即便在 self 会被隐式使用的场景下，我也倾向于显式使用它，不过这里却无法使用 self，虽然编译器不会报错（我认为这是个 Bug）。为了表示 friendly 与 hostile 的状态，我可以使用类型名字，就像其他代码一样：

```
struct Greeting {
    static let friendly = "hello there"
    static let hostile = "go away"
    static let ambivalent = Greeting.friendly + " but " + Greeting.hostile
}
```

另外，如果将 ambivalent 作为计算属性，那就可以使用 self 了：

```
struct Greeting {
    static let friendly = "hello there"
    static let hostile = "go away"
    static var ambivalent : String {
        return self.friendly + " but " + self.hostile
    }
}
```

此外，如果初始值是通过定义与调用匿名函数所设置的，那就无法使用 self（我认为这也是个 Bug）：

```
struct Greeting {
    static let friendly = "hello there"
    static let hostile = "go away"
    static var ambivalent : String = {
        return self.friendly + " but " + self.hostile // compile error
    }()
}
```

### 4.1.3 方法

方法就是函数，只是声明在对象类型声明顶部的函数，这意味着第 2 章介绍的关于函数的一切也都适用于方法。

在默认情况下，方法是实例方法，这意味着只能通过实例来进入它。在实例方法体中，self 指的就是实例。为了说明这一点，我们继续在 Dog 类中添加一些内容：

```
class Dog {
    let name : String
    let license : Int
    let whatDogsSay = "Woof"
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
    func bark() {
        print(self.whatDogsSay)
    }
    func speak() {
        self.bark()
        print("I'm \(self.name)")
    }
}
```

现在可以创建 Dog 实例并调用它的 speak 方法：

```
let fido = Dog(name:"Fido", license:1234)
fido.speak() // Woof I'm Fido
```

在 Dog 类中，speak 方法通过 self 调用了实例方法 bark，然后又通过 self 获取到实例属性 name 的值；而 bark 实例方法则通过 self 获取到实例属性 whatDogsSay 的值。这是因为实例代码可以通过 self 引用到该实例；如果引用没有歧义，那么代码就可以省略 self；比如，代码可以写成这样：

```
func speak() {
    bark()
    print("I'm \(name)")
}
```

不过，我从来都不会这么写（仅仅是偶尔为之）。我认为，省略 self 会导致代码的可读性与可维护性变差；仅仅使用 bark 与 name 看起来会令人费解且困惑。此外，有时 self 是不可以省略的。比如，在 init(name:license:) 实现中，我必须得使用 self 消除参数 name 与属性 self.name 之间的差别。

静态 / 类属性是通过类型访问的，self 表示的是类型。参见如下 Greeting 结构体示例：

```
struct Greeting {
    static let friendly = "hello there"
    static let hostile = "go away"
    static var ambivalent : String {
        return self.friendly + " but " + self.hostile
    }
    static func beFriendly() {
        print(self.friendly)
    }
}
```

下面展示了如何调用静态方法 beFriendly：

```
Greeting.beFriendly() // hello there
```

虽然声明在相同的对象类型中，但静态 / 类成员与实例成员之间在概念上还是存在一些差别，它们位于不同的世界中。静态 / 类方法不能引用“实例”，因为根本就没有实例存在；静态 / 类方法不能直接引用任何实例属性，也不能调用任何实例方法。另外，实例方法却可以通过名字引用类型，也可以访问静态 / 类属性，调用静态 / 类方法（本章后面将会介绍实例方法引用类型的另一种方式）。

比如，回到 Dog 类上来，解决一下狗会叫的问题。假设所有狗叫的都一样。因此，我们倾向于在类级别而非实例级别表示 whatDogsSay。这正是静态属性的用武之地，下面是一个用于说明问题的简化的 Dog 类：

## 实例方法揭秘

有这样一个秘密：实例方法实际上可以访问静态 / 类方法。比如，如下代码是合法的（但看起来很奇怪）：

```
class MyClass {
    var s = ""
    func store(s:String) {
        self.s = s
    }
}
let m = MyClass()
let f = MyClass.store(m) // what just happened!?
```

虽然 store 是个实例方法，但我们能以类方法的形式调用它，即通过将类实例作为参数！原因在于实例方法实际上是由两个函数构成的调制静态 / 类方法：一个函数接收一个实例，另一个函数接收实例方法的参数。这样，在上述代码执行后，f 就成为第 2 个函数，调用它就相当于调用实例 m 的 store 方法一样：

```
f("howdy")
print(m.s) // howdy
```

```
class Dog {
    static var whatDogsSay = "Woof"
    func bark() {
        print(Dog.whatDogsSay)
    }
}
```

接下来创建一个 Dog 实例并调用其 bark 方法：

```
let fido = Dog()
fido.bark() // Woof
```

## 4.1.4 下标

下标是一种实例方法，不过调用方式比较特殊：在实例引用后面使用方括号，方括号可以包含传递给下标方法的参数。你可以通过该特性做任何想做的事情，不过它特别适合于通过键或索引号访问对象类型中的元素的场景。第 3 章曾介绍过该语法搭配字符串的使用方式，字典与数组也经常见到这种使用方式；你可以对字符串、字典与数组使用方括号，因为 Swift 的 String、Dictionary 与 Array 类型都声明了下标方法。

声明下标方法的语法类似于函数声明和计算属性声明，这并非巧合！下标类似于函数，因为它可以接收参数：当调用下标方法时，实参位于方括号中。下标类似于计算属性，因为调用就好像是对属性的引用：你可以获取其值，也可以对其赋值。

为了说明问题，我声明一个结构体，它对待整型的方式就像是字符串，通过在方括号中使用索引数的方式返回一个数字；出于简化的目的，我有意省略了错误检查代码：

```
struct Digit {
    var number : Int
    init(_ n:Int) {
        self.number = n
    }
    subscript(ix:Int) -> Int { ❶❷
        get { ❸
            let s = String(self.number)
            return Int(String(s[s.startIndex.advancedBy(ix)]))!
        }
    }
}
```

❶ 关键字 `subscript` 后面有一个参数列表，指定什么参数可以出现在方括号中；在默认情况下，其名字不是外化的。

❷ 接下来，在箭头运算符后面指定了传出（调用 getter 时）或传入（调用 setter 时）的值类型；这与计算属性的类型声明是类似的，不过箭头运算符的语法类似于函数声明中的返回值。

❸ 最后，花括号中的内容就像是计算属性的内容。你可以为 getter 提供 get 与花括号，为 setter 提供 set 与花括号。如果只有 getter 没有 setter，那么单词 get 及后面的花括号就可以省略。setter 会将新值作为 newValue，不过你可以在圆括号中单词 set 后面提供不同的名字来改变它。

下面是调用 getter 的一个示例；实例名后面跟着方括号，里面是实参值，调用时相当于获取一个属性值一样：

```
var d = Digit(1234)
let aDigit = d[1] // 2
```

现在来扩展 Digit 结构体，使其下标方法包含 setter（再次省略错误检查代码）：

```
struct Digit {
    var number : Int
    init(_ n:Int) {
        self.number = n
    }
    subscript(ix:Int) -> Int {
        get {
            let s = String(self.number)
            return Int(String(s[s.startIndex.advancedBy(ix)]))!
        }
        set {
            var s = String(self.number)
            let i = s.startIndex.advancedBy(ix)
            s.replaceRange(i...i, with: String(newValue))
            self.number = Int(s)!
        }
    }
}
```

下面是调用 setter 的一个示例；实例名后面跟着方括号，里面是实参值，调用时相当于设置一个属性值一样：

```
var d = Digit(1234)
d[0] = 2 // now d.number is 2234
```

一个对象类型可以声明多个下标方法，前提是其签名不同。

## 4.1.5 嵌套对象类型

一个对象类型可以声明在另一个对象类型声明中，从而形成嵌套类型：

```
class Dog {
    struct Noise {
        static var noise = "Woof"
    }
    func bark() {
```

```
            print (Dog.Noise.noise)
        }
    }
```

嵌套对象类型与一般的对象类型没有区别，不过从外部引用它的规则发生了变化；外部对象类型成为一个命名空间，必须要显式通过它才能访问到嵌套对象类型：

```
Dog.Noise.noise = "Arf"
```

Noise 结构体位于 Dog 类命名空间下面，该命名空间增强了清晰性：名字 Noise 不能随意使用，必须要显式关联到所属的 Dog 类。借助命名空间，我们可以创建多个 Noise 结构体，而不会造成名字冲突。Swift 内建对象类型通常都会利用命名空间；比如，有一些结构体包含了 Index 结构体，而 String 结构体就是其中之一，它们之间不会造成名字冲突。

（借助于 Swift 的隐私原则，我们还可以隐藏嵌套对象类型，这样就无法在外部引用它了。这样，如果一个对象类型需要另一个对象类型作为辅助，而其他对象类型无须了解这个辅助对象类型，那么通过这种方式就可以很好地起到组织和封装的目的。第 5 章将会介绍隐私。）

## 4.1.6 实例引用

总的来说，对象类型的名字是全局的，只需通过其名字就可以引用它们，不过实例则不同。实例必须要显式地逐一创建，这正是实例化的目的之所在。创建好实例后，你可以将它存储到具有足够长生命周期的变量中以保证实例一直存在；将该变量作为引用，你可以向实例发送实例消息，访问实例属性并调用实例方法。

对对象类型的实例化是直接创建该类型全新实例的一种方式，这需要调用初始化器。不过在很多情况下，其他对象会创建对象并将其提供给你。

一个简单的例子就是像下面这样操纵一个 String 时会发生什么：

```
let s = "Hello, world"
let s2 = s.uppercaseString
```

上述代码执行完毕后会生成两个 String 实例。第 1 个 s 是通过字符串字面值创建的；第 2 个 s2 是通过访问第 1 个字符串的 uppercaseString 属性创建的。因此，我们会得到两个实例，只要对它们的引用存在，这两个实例就会存在而且相互独立；不过，在创建它们时并未调用初始化器。

有时，你所需要的实例已经以某种持久化形式存在了；接下来的问题就在于如何获得对该实例的引用。

比如，有一个实际的iOS应用。你当然会有一个根视图控制器，它是某种UIViewController的实例。假设它是ViewController类的实例。当应用启动并运行后，该实例就已经存在了。接下来，通过实例化ViewController类来与根视图控制器进行通信显然与我们的想法是背道而驰的：

```
let theVC = ViewController()
```

上述代码会创建另一个完全不同的ViewController类实例，向该实例发送的消息都是毫无意义的，因为它并非你想要与之通信的那个特定实例。这是初学者常犯的一个错误，请注意。

获取对已经存在的实例的引用是个很有意思的话题。显然，实例化并不是解决之道，那该怎么做呢？要具体问题具体分析。在这个特定的情况下，我们的目标是从代码中获取到对应用根视图控制器实例的引用。下面来介绍一下该怎么做。

获取引用总是从你已经具有引用的对象开始，通常这是个类。在iOS编程中，应用本身就是个实例，有一个类会持有一个对该实例的引用，它会在你需要时将其传递给你。这个类就是UIApplication，我们可以通过调用其sharedApplication类方法来获得对应用实例的引用：

```
let app = UIApplication.sharedApplication()
```

现在，我们拥有了对应用实例的引用，该应用实例有一个keyWindow属性：

```
let window = app.keyWindow
```

现在，我们有了对应用主窗口的引用。该窗口拥有根视图控制器，并且会将对其的引用给我们，即其rootViewController属性；应用的keyWindow是个Optional，因此需要将其展开才能得到rootViewController：

```
let vc = window?.rootViewController
```

现在，我们有了对应用根视图控制器的引用。为了获得对该持久化实例的引用，我们实际上创建了一个方法调用与属性链，从已知到未知，从全局类到特定实例：

```
let app = UIApplication.sharedApplication()
let window = app.keyWindow
let vc = window?.rootViewController
```

显然，可以通过一个链来表示上述代码，使用重复的点符号即可：

```
let vc = UIApplication.sharedApplication().keyWindow?.rootViewController
```

无需将实例消息链接为单独一行：使用多个 `let` 赋值会更具效率、更加清晰、也更易于调试。不过这么做会更加便捷，也是 Swift 这种使用点符号的面向对象语言的一个特性。

获取对已经存在的实例的引用是个很有趣的话题，应用也非常广泛，第 13 章将会对其进行深入介绍。

## 4.2 枚举

枚举是一种对象类型，其实例表示不同的预定义值，可以将其看作已知可能的一个列表。Swift 通过枚举来表示彼此可替代的一组常量。枚举声明中包含了若干 case 语句。每个 case 都是一个选择名。一个枚举实例只表示一个选择，即其中的一个 case。

比如，在我开发的 Albumen 应用中，相同视图控制器的不同实例可以列出 4 种不同的音乐库内容：专辑、播放列表、播客、有声书。视图控制器的行为对于每一种音乐库内容来说存在一些差别。因此，在实例化视图控制器时，我需要一个四路 switch 进行设置，表示该视图控制器会显示哪一种内容。这就像枚举一样！

下面是该枚举的基本声明；称为 Filter，因为每个 case 都表示过滤音乐库内容的不同方式：

```
enum Filter {
    case Albums
    case Playlists
    case Podcasts
    case Books
}
```

该枚举并没有初始化器。你可以为枚举编写初始化器，稍后将会介绍；不过它提供了默认的初始化模式，你可以在大多数时候使用该模式：使用枚举名，后跟点符号以及一个 case。比如，如下代码展示了如何创建表示 Albums case 的 Filter 实例：

```
let type = Filter.Albums
```

作为一种简写，如果类型提前就知道了，那就可以省略枚举的名字，不过前面还是要有一个点。比如：

```
let type : Filter = .Albums
```

不能在其他地方使用 `.Albums`，因为 Swift 不知道它属于哪个枚举。在上述代码中，变量被显式声明为 Filter，因此 Swift 知道 `.Albums` 的含义。类似的情况出现在将枚举实例作为实参传递给函数调用时：

```
func filterExpecter(type:Filter) {}
filterExpecter(.Albums)
```

第 2 行创建了一个 Filter 实例并传递给函数，无须使用枚举的名字。这是因为 Swift 从函数声明中已经知道这里需要一个 Filter 类型。

在实际开发中，省略枚举名所带来的空间上的节省可能会相当可观，特别是在与 Cocoa 通信时，枚举类型名通常都会很长。比如：

```
let v = UIView()
v.contentMode = .Center
```

UIView 的 contentMode 属性是 UIViewContentMode 枚举类型。上述代码很简洁，因为我们无须在这里显式使用名字 UIViewContentMode。.Center 要比 UIViewContentMode.Center 更加整洁，但二者都是合法的。

枚举声明中的代码可以在不使用点符号的情况下使用 case 名。枚举是个命名空间，声明中的代码位于该命名空间下面，因此能够直接看到 case 名。

相同 case 的枚举实例是相等的。因此，你可以比较枚举实例与 case 来判断它们是否相等。第 1 次比较时就获悉了枚举的类型，因此第 2 次之后就可以省略枚举名字了：

```
func filterExpecter(type:Filter) {
    if type == .Albums {
        print("it's albums")
    }
}
filterExpecter(.Albums) // "it's albums"
```

## 4.2.1 带有固定值的 Case

在声明枚举时，你可以添加类型声明。接下来，所有 case 都会持有该类型的一个固定值（常量）。如果类型是整型数字，那么值就会隐式赋予，并且默认从 0 开始。在如下代码中，.Mannie 持有值 0，.Moe 持有值 1，以此类推：

```
enum PepBoy : Int {
    case Mannie
    case Moe
    case Jack
}
```

如果类型为 String，那么隐式赋予的值就是 case 名字的字符串表示。在如下代码中，.Albums 持有值 "Albums"，以此类推：

```
enum Filter : String {
    case Albums
    case Playlists
    case Podcasts
```

```
    case Books
}
```

无论类型是什么，你都可以在 case 声明中显式赋值：

```
enum Filter : String {
    case Albums = "Albums"
    case Playlists = "Playlists"
    case Podcasts = "Podcasts"
    case Books = "Audiobooks"
}
```

以这种方式附加到枚举上的类型只能是数字与字符串，赋的值必须是字面值。case 所持有的值叫作其原生值。该枚举的一个实例只会有一个 case，因此只有一个固定的原始值，并且可以通过 rawValue 属性获取到：

```
let type = Filter.Albums
print(type.rawValue) // Albums
```

让每个 case 都有一个固定的原始值会很有意义。在我开发的 Albumen 应用中，Filter case 持有的就是上述 String 值，当视图控制器想获取标题字符串并展现在屏幕顶部时，它只需获取到当前类型的 rawValue 即可。

与每个 case 关联的原生值在当前枚举中必须唯一；编译器会强制施加该规则。因此，我们还可以进行反向匹配：给定一个原生值，可以得到与之对应的 case。比如，你可以通过 rawValue: 初始化器实例化具有该原生值的枚举：

```
let type = Filter(rawValue:"Albums")
```

不过，以这种方式来实例化枚举可能会失败，因为提供的原生值可能不对应任何一个 case；因此，这是一个可失败初始化器，其返回值是 Optional。在上述代码中，type 并非 Filter，它是个包装了 Filter 的 Optional。这可能不那么重要，不过由于你要做的事情很可能是比较枚举与其 case，因此可以使用 Optional 而无须展开。如下代码是合法的，并且执行正确：

```
let type = Filter(rawValue:"Albums")
if type == .Albums { // ...
```

## 4.2.2 带有类型值的 Case

4.2.1 节介绍的原生值是固定的：给定的 case 会持有某个原生值。此外，你可以构建这样一个 case，其常量值是在实例创建时设置的。为了做到这一点，请不要为枚举声明任何类型；相反，请向 case 的名字附加一个元组类型。通常来说，该元组中只会有一个类型；因此，其形式就是圆括号中会有一个类型名，其中可以声明任何类型，如下示例所示：

```
enum Error {
    case Number(Int)
    case Message(String)
    case Fatal
}
```

上述代码的含义是：在实例化期间，带有 .Number case 的 Error 实例必须要赋予一个 Int 值，带有 .Message case 的 Error 实例必须要赋予一个 String 值，带有 .Fatal case 的 Error 实例不能赋予任何值。带有赋值的实例化实际上会调用一个初始化函数；若想提供值，你需要将其作为实参放到圆括号中：

```
let err : Error = .Number(4)
```

这里的附加值叫作关联值。这里所提供的实际上是个元组，因此它可以包含字面值或值引用；如下代码是合法的：

```
let num = 4
let err : Error = .Number(num)
```

元组可以包含多个值，可以提供名字，也可以不提供名字；如果值有名字，那么必须在初始化期间使用：

```
enum Error {
    case Number(Int)
    case Message(String)
    case Fatal(n:Int, s:String)
}
let err : Error = .Fatal(n:-12, s:"Oh the horror")
```

声明了关联值的枚举 case 实际上是个初始化函数，这样就可以捕获到对该函数的引用并在后面调用它：

```
let fatalMaker = Error.Fatal
let err = fatalMaker(n:-1000, s:"Unbelievably bad error")
```

第 5 章将会介绍如何从这样的枚举实例中提取出关联值。

下面我来揭示 Optional 的工作原理。Optional 实际上是一个带有两个 case 的枚举：.None 与 .Some。如果为 .None，那么它就没有关联值，并且等于 nil；如果为 .Some，那么它就会将包装值作为关联值。

## 4.2.3 枚举初始化器

显式的枚举初始化器必须要实现与默认初始化相同的工作：它必须返回该枚举特定的一个 case。为了做到这一点，请将 self 设定给 case。在该示例中，我扩展了 Filter 枚举，使之可以通过数字参数进行初始化：

```
enum Filter: String {
    case Albums = "Albums"
    case Playlists = "Playlists"
    case Podcasts = "Podcasts"
    case Books = "Audiobooks"
    static var cases : [Filter] = [Albums, Playlists, Podcasts, Books]
    init(_ ix:Int) {
        self = Filter.cases[ix]
    }
}
```

现在有 3 种方式可以创建 Filter 实例：

```
let type1 = Filter.Albums
let type2 = Filter (rawValue:"Playlists")!
let type3 = Filter (2) // .Podcasts
```

在该示例中，如果调用者传递的数字超出了范围（小于 0 或大于 3），那么第 3 行将会崩溃。为了避免这种情况的出现，我们可以将其作为可失败初始化器，如果数字超出了范围就返回 `nil`：

```
enum Filter: String {
    case Albums = "Albums"
    case Playlists = "Playlists"
    case Podcasts = "Podcasts"
    case Books = "Audiobooks"
    static var cases : [Filter] = [Albums, Playlists, Podcasts, Books]
    init!(_ ix:Int) {
        if !(0...3).contains(ix) {
            return nil
        }
        self = Filter.cases[ix]
    }
}
```

一个枚举可以有多个初始化器。枚举初始化器可以通过调用 `self.init(...)` 委托给其他初始化器，前提是在调用链的某个点上将 `self` 设定给一个 case；如果不这么做，那么枚举将无法编译通过。

该示例改进了 Filter 枚举，这样它可以通过一个 String 原生值进行初始化而无须调用 `rawValue:`。为了做到这一点，我声明了一个可失败初始化器，它接收一个字符串参数，并且委托给内建的可失败 `rawValue:` 初始化器：

```
enum Filter: String {
    case Albums = "Albums"
    case Playlists = "Playlists"
    case Podcasts = "Podcasts"
    case Books = "Audiobooks"
    static var cases : [Filter] = [Albums, Playlists, Podcasts, Books]
    init!(_ ix:Int) {
```

```
            if !(0...3).contains(ix) {
                return nil
            }
            self = Filter.cases[ix]
        }
        init!(_ rawValue:String) {
            self.init(rawValue:rawValue)
        }
    }
```

现在有 4 种方式可以创建 Filter 实例：

```
let type1 = Filter.Albums
let type2 = Filter (rawValue:"Playlists")
let type3 = Filter (2) // .Podcasts
let type4 = Filter ("Playlists")
```

## 4.2.4 枚举属性

枚举可以拥有实例属性与静态属性，不过有一个限制：枚举实例属性不能是存储属性。这是有意义的，因为如果相同 case 的两个实例拥有不同的存储实例属性值，那么它们彼此之间就不相等了——这有悖于枚举的本质与目的。

不过，计算实例属性是可以的，并且属性值会根据 `self` 的 case 发生变化。如下示例来自于我所编写的代码，我将搜索函数关联到了 Filter 枚举的每个 case 上，用于从音乐库中获取该类型的歌曲：

```
enum Filter : String {
    case Albums = "Albums"
    case Playlists = "Playlists"
    case Podcasts = "Podcasts"
    case Books = "Audiobooks"
    var query : MPMediaQuery {
        switch self {
        case .Albums:
            return MPMediaQuery.albumsQuery()
        case .Playlists:
            return MPMediaQuery.playlistsQuery()
        case .Podcasts:
            return MPMediaQuery.podcastsQuery()
        case .Books:
             return MPMediaQuery.audiobooksQuery()
        }
    }
```

如果枚举实例属性是个带有 Setter 的计算变量，那么其他代码就可以为该属性赋值了。不过，代码中对枚举实例的引用必须是个变量（`var`）而不能是常量（`let`）。如果试图通过 `let` 引用为枚举实例属性赋值，那么编译器就会报错。

## 4.2.5 枚举方法

枚举可以有实例方法（包括下标）与静态方法。编写枚举方法是相当直接的。如下示例来自于我之前编写的代码。在纸牌游戏中，每张牌分为矩形、椭圆与菱形。我将绘制代码抽象为一个枚举，它会将自身绘制为一个矩形、椭圆或菱形，取决于其 case 的不同：

```
enum ShapeMaker {
    case Rectangle
    case Ellipse
    case Diamond
    func drawShape (p: CGMutablePath, inRect r : CGRect) -> () {
        switch self {
        case Rectangle:
            CGPathAddRect(p, nil, r)
        case Ellipse:
            CGPathAddEllipseInRect(p, nil, r)
        case Diamond:
            CGPathMoveToPoint(p, nil, r.minX, r.midY)
            CGPathAddLineToPoint(p, nil, r.midX, r.minY)
            CGPathAddLineToPoint(p, nil, r.maxX, r.midY)
            CGPathAddLineToPoint(p, nil, r.midX, r.maxY)
            CGPathCloseSubpath(p)
        }
    }
}
```

修改枚举自身的枚举实例方法应该被标记为 mutating。比如，一个枚举实例方法可能会为 self 的实例属性赋值；虽然这是个计算属性，但这种赋值还是不合法的，除非将该方法标记为 mutating。枚举实例方法甚至可以修改 self 的 case；不过，方法依然要标记为 mutating。可变实例方法的调用者必须要有一个对该实例的变量引用（var）而非常量引用（let）。

在该示例中，我向 Filter 枚举添加了一个 advance 方法。想法在于 case 构成了一个序列，序列可以循环。通过调用 advance，我可以将 Filter 实例转换为序列中的下一个 case：

```
enum Filter : String {
    case Albums = "Albums"
    case Playlists = "Playlists"
    case Podcasts = "Podcasts"
    case Books = "Audiobooks"
    static var cases : [Filter] = [Albums, Playlists, Podcasts, Books]
    mutating func advance() {
        var ix = Filter.cases.indexOf(self)!
        ix = (ix + 1) % 4
        self = Filter.cases[ix]
    }
}
```

下面是调用代码：

```
var type = Filter.Books
type.advance() // type is now Filter.Albums
```

（下标 Setter 总被认为是 mutating，不必显式标记。）

## 4.2.6 为何使用枚举

枚举是个拥有状态名的 switch。很多时候我们都需要使用枚举。你可以自己实现一个多状态值；比如，如果有 5 种可能的状态，你可以使用一个值介于 0 到 4 之间的 Int。不过接下来还有不少工作要做，要确保不会使用其他值，并且要正确解释这些数值。对于这种情况来说，5 个具名 case 会更好一些！即便只有两个状态，枚举也比 Bool 好，这是因为枚举的状态拥有名字。如果使用 Bool，那么你就得知道 `true` 与 `false` 到底表示什么；借助枚举，枚举的名字与 case 的名字会告诉你这一切。此外，你可以在枚举的关联值或原生值中存储额外的信息，但 Bool 却做不到这些。

比如，在我实现的 LinkSame 应用中，用户可以使用定时器开始真正的游戏，也可以不使用定时器进行练习。在代码的不同位置处，我需要知道进行的是真正的游戏还是练习。游戏类型是枚举的 case：

```
enum InterfaceMode : Int {
    case Timed = 0
    case Practice = 1
}
```

当前的游戏类型存储在实例属性 `interfaceMode` 中，其值是个 InterfaceMode。这样就可以轻松根据 case 的名字设定游戏了：

```
// ... initialize new game ...
self.interfaceMode = .Timed
```

也可以轻松根据 case 名字检测游戏类型：

```
// notify of high score only if user is not just practicing
if self.interfaceMode == .Timed { // ...
```

那原生整型值起什么作用呢？它们对应于界面中 UISegmentedControl 的分割索引。当修改了 `interfaceMode` 属性时，Setter 观察者会选择 UISegmentedControl 中相应的分割部分（`self.timedPractice`），这只需获取到当前枚举 case 的 `rawValue` 即可：

```
var interfaceMode : InterfaceMode = .Timed {
    willSet (mode) {
        self.timedPractice?.selectedSegmentIndex = mode.rawValue
    }
}
```

# 4.3 结构体

结构体是 Swift 中非常重要的对象类型。枚举的 case 数量固定，实际上它是一种精简、特殊的对象。类则是另一个极端，它的能力过于强大；它拥有一些结构体缺乏的特性，不过如果不需要这些特性，那么结构体就是最佳选择。

在 Swift 头文件所声明的大量对象类型中，只有 4 个是类（你不大可能用到它们）。与之相反，Swift 本身提供的几乎所有内建对象类型都是结构体。String 是个结构体、Int 是个结构体、Range 是个结构体、Array 也是结构体。这表明了结构体的强大功能。

## 4.3.1 结构体初始化器、属性与方法

如果一个结构体没有显式初始化器，或不需要显式初始化器（因为结构体没有存储属性，或在声明中为所有存储属性都赋予了默认值），那么它就会自动拥有一个不带参数的隐式初始化器 `init()`。比如：

```
struct Digit {
    var number = 42
}
```

可以通过调用`Digit()`来初始化上面的结构体。不过，如果添加了自定义的显式初始化器，那么隐式初始化器就不复存在了：

```
struct Digit {
    var number = 42
    init(number:Int) {
        self.number = number
    }
}
```

现在可以调用 `Digit(number:42)`，不过不能再调用 `Digit()` 了。当然了，你可以添加一个显式初始化器完成相同的事情：

```
struct Digit {
    var number = 42
    init() {}
    init(number:Int) {
        self.number = number
    }
}
```

现在又可以调用 `Digit()` 了，也可以调用 `Digit(number:42)`。

拥有存储属性，并且没有显式初始化器的结构体会自动获得来自于实例属性的隐式初始化器。这叫作成员初始化器。比如：

```
struct Digit {
    var number : Int // can use "let" here
}
```

上述结构体是合法的(事实上,即便使用let而非var来声明number属性,它也是合法的),不过似乎我们并没有实现在声明中或初始化器中初始化所有存储属性的契约。原因在于该结构体自动具有了一个成员初始化器,它会初始化所有属性。在该示例中,成员初始化器叫作 init(number:)。

即便在声明中为可变存储属性赋了默认值,成员初始化器也是存在的;这样,除了隐式 init() 初始化器,该结构体还有一个成员初始化器 init(number:):

```
struct Digit {
    var number = 42
}
```

如果添加了自定义的显式初始化器,那么成员初始化器就不复存在了(当然,你还是可以提供显式初始化器完成相同的事情)。

如果结构体拥有显式初始化器,那么它必须要实现这个契约:要么在声明中,要么在所有初始化器中完成对所有存储属性的初始化。如果结构体有多个显式初始化器,那么可以通过调用 self.init(...) 进行委托。

结构体可以拥有实例属性与静态属性,它们既可以是存储变量,也可以是计算变量。如果其他代码想要设置结构体实例的某个属性,那么对该实例的引用就必须是个变量(var)而不能是常量(let)。

结构体可以拥有实例方法(包括下标)与静态方法。如果实例方法设置某个属性,那么必须要将其标记为 mutating,调用者对该结构体实例的引用必须是个变量(var)而不能是常量(let)。mutating 实例方法甚至可以用别的实例替换掉当前实例,只需将 self 设置为相同结构体的不同实例即可(下标 Setter 总是 mutating 的,因此不必显式标记)。

## 4.3.2 将结构体作为命名空间

我经常将退化的结构体作为常量的命名空间。之所以称一个结构体为“退化的”,是因为它只由静态成员构成;我不会通过该对象类型创建任何实例。不过,这么使用结构体是完全没问题的。

比如,假设要在 Cocoa 的 NSUserDefaults 中存储用户偏好信息。NSUserDefaults 是一种字典:每一项都可以通过键来访问,键通常是字符串。一个常见的程序错误就是在每次需要键时都手工写出这些字符串键;如果拼错了键名,那么在编译期是不会有任何错误

出现的，不过代码将无法正常工作。好的方式是将这些键作为常量字符串，并使用字符串的名字；通过这种方式，如果在输入字符串名的时候出错了，那么编译器会提醒你。拥有静态成员的结构体非常适合定义这些常量字符串，并且将这些名字形成到一个命名空间中：

```
struct Default {
    static let Rows = "CardMatrixRows"
    static let Columns = "CardMatrixColumns"
    static let HazyStripy = "HazyStripy"
}
```

上述代码表示我现在可以通过一个名字来引用NSUserDefaults键，如`Default.HazyStripy`。

如果结构体声明了静态成员，其值是相同结构体类型的实例，那么在需要该结构体类型实例的情况下，提供结构体静态成员时就可以省略结构体的名字，就好像该结构体是个枚举一样：

```
struct Thing {
    var rawValue : Int = 0
    static var One : Thing = Thing(rawValue:1)
    static var Two : Thing = Thing(rawValue:2)
}
let thing : Thing = .One // no need to say Thing.One here
```

示例本身是我们设想的，不过使用场景却不是；很多Objective-C枚举都是通过这种结构体桥接到Swift的（后面还会对此进行介绍）。

## 4.4 类

类与结构体相似，但存在如下一些主要差别：

*引用类型*

类是引用类型。这意味着类实例有两个结构体或枚举实例所不具备的关键特性。

*可变性*

类实例是可变的。虽然对类实例的引用是个常量（`let`），不过你可以通过该引用修改实例属性的值。类的实例方法绝不能标记为`mutating`。

*多引用*

如果给定的类实例被赋予多个变量或作为参数传递给函数，那么你就拥有了对相同对象的多个引用。

*继承*

类可以拥有子类。如果一个类有父类，那么它就是这个父类的子类。这样，类就可以构成一种树形结构了。

在 Objective-C 中，类是唯一一种对象类型。一些内建的 Swift 结构体类型会桥接到 Objective-C 的类类型，不过自定义的结构体类型却做不到这一点。因此，在使用 Swift 进行 iOS 编程时，使用类而非结构体的一个主要原因就是它能够与 Objective-C 和 Cocoa 互换。

## 4.4.1 值类型与引用类型

枚举与结构体是一类，类是另一类，这两类之间的主要差别在于前者是值类型，而后者是引用类型。

值类型是不可变的。实际上，这意味着你无法修改值类型实例属性的值。看起来可以修改，但实际上是不行的。比如，我们考虑一个结构体。结构体是值类型：

```
struct Digit {
    var number : Int
    init(_ n:Int) {
        self.number = n
    }
}
```

看起来好像可以修改 Digit 的 number 属性。毕竟，这是将该属性声明为 var 的唯一目的；Swift 的赋值语法使我们相信修改 Digit 的 number 是可行的：

```
var d = Digit(123)
d.number = 42
```

但实际上，在上述代码中，我们并未修改这个 Digit 实例的 number 属性；我们实际上创建了一个不同的 Digit 实例并替换掉了之前那个。要想证明这一点，我们添加一个 Setter 观察者：

```
var d : Digit = Digit(123) {
    didSet {
        print("d was set")
    }
}
d.number = 42 // "d was set"
```

一般来说，当修改一个实例值类型时，你实际上会通过另一个实例替换掉当前这个实例。这说明了如果对该实例的引用是通过 let 声明的，那么这就是无法修改值类型实例的原因。如你所知，使用 let 声明的初始化变量是不能被赋值的。如果该变量指向了值类型

实例，并且该值类型实例有一个属性，即便这个属性是通过 var 声明的，如果我们对该属性赋值，那么编译器就会报错：

```
struct Digit {
    var number : Int
    init(_ n:Int) {
        self.number = n
    }
}
let d = Digit(123)
d.number = 42 // compile error
```

原因在于这种修改需要替换掉 d 盒子中的 Digit 实例。不过，我们无法通过另一个 Digit 实例替换掉 d 所指向的 Digit 实例，因为这意味着要对 d 赋值，而 let 声明是不允许这么做的。

反过来，这正是设置实例属性的结构体或枚举的实例方法要被显式标记为 mutating 关键字的原因所在。比如：

```
struct Digit {
    var number : Int
    init(_ n:Int) {
        self.number = n
    }
    mutating func changeNumberTo(n:Int) {
        self.number = n
    }
}
```

如果不使用 mutating 关键字，那么上述代码将无法编译通过。mutating 关键字会让编译器相信你知道这里会产生什么样的结果：如果方法被调用了，那么它会替换掉这个实例，这样该方法只能在使用 var 声明的引用上进行调用，let 则不行：

```
let d = Digit(123)
d.changeNumberTo(42) // compile error
```

不过，我所说的这一切都不适用于类实例！类实例是引用类型，而非值类型。如果一个类的实例属性可以被修改，那么显然要用 var 声明；不过，若想通过类实例的引用来设置属性，那么引用是无须声明为 var 的：

```
class Dog {
    var name : String = "Fido"
}
let rover = Dog()
rover.name = "Rover" // fine
```

在上面最后一行代码中，rover 所指向的类实例会在原地被修改。这里不会对 rover 进行

隐式赋值，因此 let 声明是无法阻止修改的。在设置属性时，Dog 变量上的 Setter 观察者是不会被调用的：

```
var rover : Dog = Dog() {
    didSet {
        print("did set rover")
    }
}
rover.name = "Rover" // nothing in console
```

如果显式设置 rover（设为另一个 Dog 实例），那么 Setter 观察者会被调用；不过，这里仅仅是修改了 rover 所指向的 Dog 实例的一个属性，因此 Setter 观察者不会被调用。

这些示例都涉及声明的变量引用。对于函数调用的参数来说，值类型与引用类型之间的差别依然存在，并且与之前所述一致。如果尝试对枚举参数的实例属性或结构体参数的实例属性赋值，那么编译器就会报错。如下代码无法编译通过：

```
func digitChanger(d:Digit) {
    d.number = 42 // compile error
}
```

要想让上述代码编译通过，请使用 var 来声明参数：

```
func digitChanger(var d:Digit) {
    d.number = 42
}
```

但如下函数声明没有使用 var 依然也能编译通过：

```
func dogChanger(d:Dog) {
    d.name = "Rover"
}
```

值类型与引用类型存在这些差别的深层次原因在于：对于引用类型来说，在对实例的引用与实例本身之间实际上存在一个隐藏的间接层次；引用实际上引用的是对实例的指针。这又引申出了另一个重要的隐喻：在将类实例赋给变量或作为参数传递给函数时，你可以使用针对同一个对象的多个引用。但结构体与枚举却不是这样。在将枚举实例或结构体实例赋给变量、传递给函数，或从函数返回时，真正赋值或传递的本质上是该实例的一个新副本。不过，在将类实例赋给变量、传递给函数，或从函数返回时，赋值或传递的是对相同实例的引用。

为了证明这一点，我将一个引用赋给另一个引用，然后修改第 2 个引用，接下来看看第 1 个引用会发生什么。先来看看结构体：

```
var d = Digit(123)
print(d.number) // 123
```

```
var d2 = d // assignment!
d2.number = 42
print(d.number) // 123
```

上述代码修改了结构体实例 d2 的 number 属性；这并不会影响 d 的 number 属性。下面再来看看类：

```
var fido = Dog()
print(fido.name) // Fido
var rover = fido // assignment!
rover.name = "Rover"
print(fido.name) // Rover
```

上述代码修改了类实例 rover 的 name 属性，fido 的 name 属性也随之发生了变化！这是因为第 3 行的赋值语句执行后，fido 与 rover 都指向了相同的实例。在对枚举或结构体实例赋值时，实际上会执行复制，创建出全新的实例。不过在对类实例进行赋值时，得到的是对相同实例的新引用。

参数传递亦如此。先来看看结构体：

```
func digitChanger(var d:Digit) {
    d.number = 42
}
var d = Digit(123)
print(d.number) // 123
digitChanger(d)
print(d.number) // 123
```

我们将 Digit 结构体实例 d 传递给了函数 digitChanger，它会将局部参数 d 的 number 属性设为 42。不过，Digit 实例 d 的 number 属性依然为 123。这是因为，传递给 digitChanger 的 Digit 是个完全不同的 Digit。作为函数实参传递 Digit 的动作会创建一个全新的副本。不过对于类实例来说，传递的是对相同实例的引用：

```
func dogChanger(d:Dog) { // no "var" needed
    d.name = "Rover"
}
var fido = Dog()
print(fido.name) // "Fido"
dogChanger(fido)
print(fido.name) // "Rover"
```

函数 dogChanger 中对 d 的修改会影响 Dog 实例 fido！将类实例传递给函数并不会复制该实例，而更像是将该实例借给函数一样。

可以生成相同实例的多个引用的能力在基于对象编程的世界中是非常重要的，其中对象可以持久化，并且其中的属性也会随之持久化。如果对象 A 与对象 B 都是长久存在的对象，并且它们都拥有一个 Dog 属性（Dog 是个类），将对相同 Dog 实例的引用分别传递

给这两个对象，对象 A 与对象 B 都可以修改其 Dog 属性，那么一个对象对 Dog 属性的修改就会影响另一个对象。你持有着一个对象，然后发现它已经被其他人修改了。这个问题在多线程应用中变得更为严重，相同的对象可能会被两个不同的线程修改；值类型就不存在这些问题；实际上，正是由于这个差别的存在，在设计对象类型时，你会更倾向于使用结构体而非类。

引用类型有缺点，但同样也有优点！优点在于传递类实例变得非常简单，你所传递的只是一个指针而已。无论对象实例有多大，多复杂；无论包含了多少属性，拥有多少数据量，传递实例都是非常快速且高效的，因为整个过程中不会产生新数据。此外，在传递时，类实例更为长久的生命周期对于其功能性和完整性是至关重要的；UIViewController 需要是类而不能是结构体，因为无论如何传递，每个 UIViewController 实例都会表示运行着的应用的视图控制器体系中同一个真实存在且持久的视图控制器。

### 递归引用

值类型与引用类型的另一个主要差别在于值类型从结构上来说是不能递归的：值类型的实例属性不能是相同类型的实例。如下代码无法编译通过：

```
struct Dog { // compile error
    var puppy : Dog?
}
```

如 Dog 包含了 Puppy 属性，同时 Puppy 又包含了 Dog 属性等更为复杂的循环链也是不合法的。不过，如果 Dog 是类而不是结构体，那就没问题了。这是值类型与引用类型在内存管理上的不同导致的（第 5 章将会详细介绍引用类型内存管理，第 12 章会介绍这个话题）。

在 Swift 2.0 中，枚举 case 的关联值可以是该枚举的实例，前提是该 case（或整个枚举）被标记为 indirect：

```
enum Node {
    case None(Int)
    indirect case Left(Int, Node)
    indirect case Right(Int, Node)
    indirect case Both(Int, Node, Node)
}
```

## 4.4.2 子类与父类

两个类彼此间可以形成子类与父类的关系。比如，我们有个名为 Quadruped 的类和名为 Dog 的类，并让 Quadruped 成为 Dog 的父类。一个类可以有多个子类，但一个类只能有一个直接父类。这里的“直接”指的是父类本身也可能有父类，这样会形成一个链条，

直到到达最终的父类，我们称为基类或根类。由于一个类可以有多个子类，并且只有一个父类，因此会形成一个子类层次树，每个子类都从其父类分支出来，同时顶部只有一个基类。

对于 Swift 语言本身来说，并不要求一个类必须要有父类；如果有父类，那么最终也是从某个特定的基类延伸出来的。因此，Swift 程序中可能会有很多类没有父类，会有很多独立的层次化子类树，每棵树都从不同的基类延伸出来。

不过，Cocoa 却不是这样的。在 Cocoa 中只有一个基类：NSObject，它提供了一个类需要的所有必备功能，其他所有类都是该基类不同层次上的子类。因此，Cocoa 包含了一个巨大的类层次树，甚至在你编写代码或创建自定义类之前就是这样的。我们可以将这棵树画出来作为一个大纲。事实上，Xcode 可以呈现出这个大纲（如图 4-1 所示）：在 iOS 项目窗口中，选择 View → Navigators → Show Symbol Navigator 并单击 Hierarchical，选中过滤栏上的第 1 个与第 3 个图标（标记为蓝色）。Cocoa 类是 NSObject 下面的树形结构的一部分。

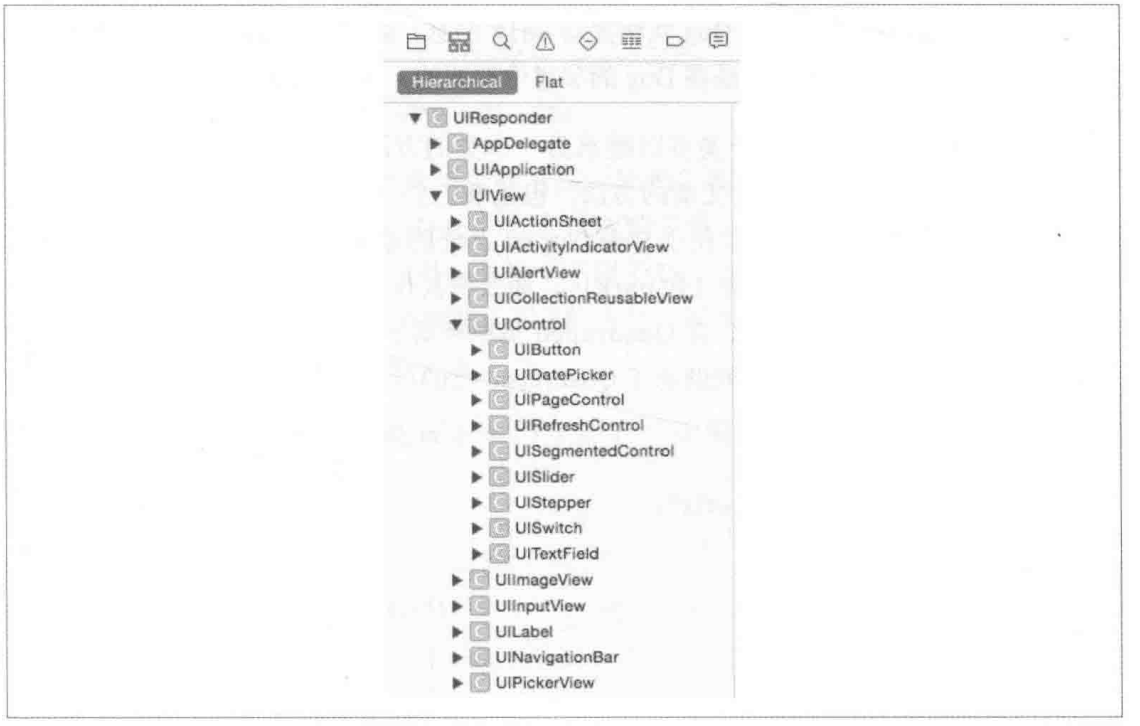


图 4-1：Xcode 中呈现的 Cocoa 类层次关系的一部分

起初，设定父类与子类关系的目的在于可以让相关类共享一些功能。比如，我们有一个 Dog 类和一个 Cat 类，考虑为这两个类声明一个 `walk` 方法。因为狗与猫都是四肢动物，因此可以想象它们走路的方式大体上是相似的。这样，将 `walk` 作为 Quadruped 类的方

法会更合理一些，并且让 Dog 与 Cat 成为 Quadruped 的子类。结果就是虽然 Dog 与 Cat 没有定义 walk 方法，但却可以向它们发送 walk 消息，这是因为它们都有一个拥有 walk 方法的父类。我们可以说子类继承了父类的方法。

要想将某个类声明为另一个类的子类，请在类声明的类名后面加上一个冒号和父类的名字，比如：

```
class Quadruped {
    func walk () {
        print("walk walk walk")
    }
}
class Dog : Quadruped {}
class Cat : Quadruped {}
```

现在来证明 Dog 实际上继承了 Quadruped 的 walk 方法：

```
let fido = Dog()
fido.walk() // walk walk walk
```

注意，在上述代码中，可以向 Dog 实例发送 walk 消息，就好像 walk 实例方法是在 Dog 类中声明的一样，虽然实际上是在 Dog 的父类中声明的，这正是继承所起的作用。

子类化的目的不仅在于让一个类可以继承另一个类的方法；子类还可以声明自己的方法。通常，子类会包含继承自父类的方法，但远非这些。如果 Dog 没有定义自己的方法，那么我们就很难看到它存在于 Quadruped 之外的原因。不过，如果 Dog 知道一些 Quadruped 所不知道的事情（如 bark），那么将其作为单独一个类才有意义。如果在 Dog 类中声明了 bark 方法，在 Quadruped 类中声明了 walk 方法，并且让 Dog 成为 Quadruped 的子类，那么 Dog 就继承了 Quadruped 类的行走能力，而且还可以 bark：

```
class Quadruped {
    func walk () {
        println("walk walk walk")
    }
}
class Dog : Quadruped {
    func bark () {
        println("woof")
    }
}
```

下面证明一下：

```
let fido = Dog()
fido.walk() // walk walk walk
fido.bark() // woof
```

一个类是否有一个实例方法并不是什么重要的事情，因为方法可以声明在该类中，也可以声明在父类中并继承下来。发送给 self 的消息在这两种情况下都可以正常运作。如下代码声明了一个 barkAndWalk 实例方法，它向 self 发送了两条消息，并没有考虑相应的方法是在哪里声明的（一个在当前类中声明的，另一个则继承自父类）：

```
class Quadruped {
    func walk () {
        print("walk walk walk")
    }
}
class Dog : Quadruped {
    func bark () {
        print("woof")
    }
    func barkAndWalk() {
        self.bark()
        self.walk()
    }
}
```

下面证明一下：

```
let fido = Dog()
fido.barkAndWalk() // woof walk walk walk
```

子类还可以重新定义从父类继承下来的方法。比如，也许一些狗的 bark 不同于别的狗。我们可以定义一个类 NoisyDog，它是 Dog 的子类。Dog 声明了 bark 方法，不过 NoisyDog 也声明了 bark 方法，并且其定义不同于 Dog 对其的定义，这叫作重写。本质原则在于，如果子类重写了从父类继承下来的方法，那么在向该子类实例发送消息时，被调用的方法是子类所声明的那一个。

在 Swift 中，当重写从父类继承下来的东西时，你需要在声明前显式使用关键字 override。比如：

```
class Quadruped {
    func walk () {
        print("walk walk walk")
    }
}
class Dog : Quadruped {
    func bark () {
        print("woof")
    }
}
class NoisyDog : Dog {
    override func bark () {
        print("woof woof woof")
    }
}
```

下面试一下：

```
let fido = Dog()
fido.bark() // woof
let rover = NoisyDog()
rover.bark() // woof woof woof
```

值得注意的是，子类函数与父类函数同名并不一定就是重写。回忆一下，只要签名不同，Swift 就可以区分开同名的两个函数，它们是不同的函数，因此子类中的实现并不是对父类中实现的重写。只有当子类重新定义了继承自父类的相同函数才是重写，所谓相同函数指的是名字相同（包括外部参数名相同）和签名相同。

很多时候，我们想要在子类中重写某个东西，同时又想访问父类中被重写的对应物。这可以通过向关键字 `super` 发送消息来达成所愿。NoisyDog 中的 `bark` 实现就是个很好的示例。NoisyDog 的吠叫与 Dog 基本上是一样的，只不过次数不同而已。我们想要在 NoisyDog 的 `bark` 实现中表示出这种关系。为了做到这一点，我们让 NoisyDog 的 `bark` 实现发送 `bark` 消息，但不是发送给 `self`（这会导致循环），而是发送给 `super`；这样就会在父类而不是自己的类中搜索 `bark` 实例方法实现：

```
class Dog : Quadruped {
    func bark () {
        print("woof")
    }
}
class NoisyDog : Dog {
    override func bark () {
        for _ in 1...3 {
            super.bark()
        }
    }
}
```

下面是调用：

```
let fido = Dog()
fido.bark() // woof
let rover = NoisyDog()
rover.bark() // woof woof woof
```

下标函数是个方法。如果父类声明了下标，那么子类可以通过相同的签名声明下标，只要使用关键字 `override` 指定即可。为了调用父类的下标实现，子类可以在关键字 `super` 后使用方括号（如 `super[3]`）。

除了方法，子类还可以继承父类的属性。当然，子类还可以声明自己的附加属性，可以重写继承下来的属性（稍后将会介绍一些限制）。

可以在类声明前加上关键字 final 防止类被继承，也可以在类成员声明前加上关键字 final 防止它被子类重写。

## 4.4.3 类初始化器

类实例的初始化要比结构体或枚举实例的初始化复杂得多，这是因为类存在继承。初始化器的主要工作是确保所有属性都有初值，这样当实例创建出来后其格式就是良好的；初始化器还可以做一些对于实例的初始状态与完整性来说是必不可少的工作。不过，类可能会有父类，也有可能拥有自己的属性与初始化器。这样，除了初始化子类自身的属性并执行初始化器任务，我们必须要确保在初始化子类时，父类的属性也被初始化了，并且初始化器会按照良好的顺序执行。

Swift 以一种一致、可靠且巧妙的方式解决了这个问题，它强制施加了一些清晰且定义良好的规则，用于指导类初始化器要做的事情。

### 1. 类初始化器分类

这些规则首先对类可以拥有的初始化器种类进行了区分：

*隐式初始化器*

类没有存储属性，或是存储属性都作为声明的一部分进行初始化，没有显式初始化器，有一个隐式初始化器 init()。

*指定初始化器*

在默认情况下，类初始化器是个指定初始化器。如果类中有存储属性没有在声明中完成初始化，那么这个类至少要有一个指定初始化器，当类被实例化时，一定会有一个指定初始化器被调用，并且要确保所有存储属性都被初始化。指定初始化器不可以委托给相同类的其他初始化器；指定初始化器不能使用 self.init(...)。

*便捷初始化器*

便捷初始化器使用关键字 convenience 标记。它是个委托初始化器，必须调用 self.init(...)。此外，便捷初始化器必须要调用相同类的一个指定初始化器，否则就必须调用相同类的另一个便捷初始化器，这就构成了一个便捷初始化器链，并且最后要调用相同类的一个指定初始化器。

如下是一些示例。类没有存储属性，因此它具有一个隐式 init() 初始化器：

```
class Dog {
}
let d = Dog()
```

下面这个类的存储属性有默认值，因此它也有一个隐式 init() 初始化器：

```
class Dog {
    var name = "Fido"
}
let d = Dog()
```

下面这个类的存储属性没有默认值，它有一个指定初始化器，所有这些属性都是在该指定初始化器中初始化的：

```
class Dog {
    var name : String
    var license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
}
let d = Dog(name:"Rover", license:42)
```

下面这个类与上面的类似，不过它还有两个便捷初始化器。调用者无须提供任何参数，因为不带参数的便捷初始化器会沿着便捷初始化器链进行调用，直到遇到一个指定初始化器：

```
class Dog {
    var name : String
    var license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
    convenience init(license:Int) {
        self.init(name:"Fido", license:license)
    }
    convenience init() {
        self.init(license:1)
    }
}
let d = Dog()
```

值得注意的是，本章之前介绍的初始化器可以做什么，什么时候做等原则依然有效。除了初始化属性，只有当类的所有属性都初始化完毕后，指定初始化器才能使用 `self`。便捷初始化器是个委托初始化器，因此只有在直接或间接地调用了指定初始化器后，它才可以使用 `self`（也不能设置不可变属性）。

## 2. 子类初始化器

介绍完指定初始化器与便捷初始化器并了解了它们之间的差别后，我们来看看当一个类本身是另一个类的子类时，关于初始化器的这些原则会发生什么变化：

*无声明的初始化器*

如果子类没有声明自己的初始化器，那么其初始化器就会包含从父类中继承下来的初始化器。

*只有便捷初始化器*

如果子类没有自己的初始化器，那么它就可以声明便捷初始化器，并且与一般的便捷初始化器工作方式别无二致，因为继承向 `self` 提供了便捷初始化器一定会调用的指定初始化器。

*指定初始化器*

如果子类声明了自己的指定初始化器，那么整个规则就会发生变化。现在，初始化器都不会被继承下来！显式的指定初始化器的存在阻止了初始化器的继承。子类现在只拥有你显式编写的初始化器（不过有个例外，稍后将会介绍）。

现在，子类中的每个指定初始化器都有一个额外的要求：它必须要调用父类的一个指定初始化器，通过 `super.init(...)` 调用。此外，调用 `self` 的规则依然适用。子类的指定初始化器必须要按照如下顺序调用执行：

1. 必须确保该类（子类）的所有属性都被初始化。
2. 必须调用 `super.init(...)`，它所调用的初始化器必须是个指定初始化器。
3. 满足上面两条之后，该初始化器才可以使用 `self`，调用实例方法或访问继承的属性。

子类中的便捷初始化器依然适用于上面列出的各种规则。它们必须调用 `self.init(...)`，直接或间接（通过便捷初始化器链）调用指定初始化器。如果没有继承下来的初始化器，那么便捷初始化器所调用的初始化器必须显式声明在子类中。

如果指定初始化器没有调用 `super.init(...)`，那么在可能的情况下 `super.init()` 就会被隐式调用。如下代码是合法的：

```
class Cat {
}
class NamedCat : Cat {
    let name : String
    init(name:String) {
        self.name = name
    }
}
```

在我看来，Swift 的这个特性是错误的：Swift 不应该使用这种秘密行为，即便这个行为看起来是“有益的”。我认为上述代码不应该编译通过；指定初始化器应该总是显式调用 `super.init(...)`。

*重写初始化器*

子类可以重写父类初始化器，但要遵循如下限定：

- 签名与父类便捷初始化器匹配的初始化器必须也是个便捷初始化器，无须标记为 override。
- 签名与父类指定初始化器匹配的初始化器可以是指定初始化器，也可以是便捷初始化器，但必须要标记为 override。在重写的指定初始化器中可以通过 super.init(...) 调用被重写的父类指定初始化器。

一般来说，如果子类有指定初始化器，那就不会继承任何初始化器。不过，如果子类重写了父类所有的指定初始化器，那么子类就会继承父类的便捷初始化器。

*可失败初始化器*

只有在完成了自己的全部初始化任务后，可失败指定初始化器才能够调用 return nil。比如，可失败子类指定初始化器必须要完成所有子类属性的初始化，在调用 return nil 前必须要调用 super.init(...)（其实就是在实例销毁前，必须要先构建出实例。不过，这是必要的，目的是确保父类能够完成自己的初始化）。

如果可失败初始化器所调用的初始化器是可失败的，那么调用语法并不会发生变化，也不需要额外的测试。如果被调用的可失败初始化器失败了，那么整个初始化过程就会立刻失败（而且会终止）。

针对重写与委托的目的，返回隐式展开 Optional 的可失败初始化器（init!）就像是个常规的初始化器（init）一样。对于返回常规 Optional (init?) 的可失败初始化器，有一些额外的限制：

- init 可以重写 init?，反之则不行。
- init? 可以调用 init。
- init 可以调用 init?，方式是调用 init 并将结果展开（要使用感叹号，因为如果 init? 失败了，程序将会崩溃）。

如下示例展示了合法的语法：

```
class A:NSObject {
    init?(ok:Bool) {
        super.init()        // init? can call init
    }
}
class B:A {
    override init(ok:Bool) { // init can override init?
        super.init(ok:ok)!   // init can call init? using "!"
    }
}
```

无论何时，子类初始化器都不能设置父类的常量属性（let）。这是因为，当子类可以做除了初始化自己的属性以及调用其他初始化器之外的事情时，父类已经完成了自己的初始化，子类已经没有机会再初始化父类的常量属性了。

下面是一些示例。首先来看这样一个类，它的子类没有声明自己的显式初始化器：

```
class Dog {
    var name : String
    var license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
    convenience init(license:Int) {
        self.init(name:"Fido", license:license)
    }
}
class NoisyDog : Dog {
}
```

根据上述代码，我们可以像下面这样创建一个 NoisyDog：

```
let nd1 = NoisyDog(name:"Fido", license:1)
let nd2 = NoisyDog(license:2)
```

上述代码是合法的，因为 NoisyDog 继承了父类的初始化器。不过，你不能像下面这样创建 NoisyDog：

```
let nd3 = NoisyDog() // compile error
```

上述代码是不合法的。虽然 NoisyDog 没有声明自己的属性，它也没有隐式初始化器；但它的初始化器是继承下来的，其父类 Dog 也没有可供继承的隐式 init() 初始化器。

来看看下面这个类，其子类唯一的显式初始化器是便捷初始化器：

```
class Dog {
    var name : String
    var license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
    convenience init(license:Int) {
        self.init(name:"Fido", license:license)
    }
}
class NoisyDog : Dog {
    convenience init(name:String) {
        self.init(name:name, license:1)
```

```
    }
}
```

注意到NoisyDog的便捷初始化器是如何通过self.init(...)调用一个指定初始化器(正好是继承下来的)来满足其契约的。根据上述代码，有3种方式可以创建NoisyDog，如下所示：

```
let nd1 = NoisyDog(name:"Fido", license:1)
let nd2 = NoisyDog(license:2)
let nd3 = NoisyDog(name:"Rover")
```

下面这个类的子类声明了一个指定初始化器：

```
class Dog {
    var name : String
    var license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
    convenience init(license:Int) {
        self.init(name:"Fido", license:license)
    }
}
class NoisyDog : Dog {
    init(name:String) {
        super.init(name:name, license:1)
    }
}
```

现在，NoisyDog的显式初始化器是个指定初始化器。它通过在super调用指定初始化器满足了契约。现在的NoisyDog阻止了所有初始化器的继承；创建NoisyDog的唯一方式如下所示：

```
let nd1 = NoisyDog(name:"Rover")
```

最后，下面这个类的子类重写了其指定初始化器：

```
class Dog {
    var name : String
    var license : Int
    init(name:String, license:Int) {
        self.name = name
        self.license = license
    }
    convenience init(license:Int) {
        self.init(name:"Fido", license:license)
    }
}
class NoisyDog : Dog {
```

```
    override init(name:String, license:Int) {
        super.init(name:name, license:license)
    }
}
```

NoisyDog 重写了父类所有的指定初始化器，因此它继承了父类的便捷初始化器。有两种方式可以创建 NoisyDog：

```
let nd1 = NoisyDog(name:"Rover", license:1)
let nd2 = NoisyDog(license:2)
```

这些示例阐释了你应该牢牢记住的主要规则。你可能不需要记住其他规则，因为编译器会强制应用这些规则，并确保你所做的一切都是正确的。

### 1. 必备初始化器

关于类初始化器还有一点值得注意：类初始化器前面可以加上关键字 `required`，这意味着子类不可以省略它。反过来，这又表示如果子类实现了指定初始化器，从而阻止了继承，那么它必须要重写该初始化器，参见如下示例：

```
class Dog {
    var name : String
    required init(name:String) {
        self.name = name
    }
}
class NoisyDog : Dog {
    var obedient = false
    init(obedient:Bool) {
        self.obedient = obedient
        super.init(name:"Fido")
    }
} // compile error
```

上述代码无法编译通过。`init(name:)` 被标记为 `required`，因此除非在 NoisyDog 中继承或重写 `init(name:)`，否则代码编译是通不过的。但我们不能继承，因为通过实现 NoisyDog 的指定初始化器 `init(obedient:)`，继承已经被阻止了。因此必须要重写它：

```
class Dog {
    var name : String
    required init(name:String) {
        self.name = name
    }
}
class NoisyDog : Dog {
     var obedient = false
    init(obedient:Bool) {
        self.obedient = obedient
        super.init(name:"Fido")
```

```
    }
    required init(name:String) {
        super.init(name:name)
    }
}
```

注意，被重写的必备初始化器并没有标记 `override`，但却被标记了 `required`，这样就可以确保无论子类层次有多深都可以满足需求。

我已经介绍过了将初始化器声明为 `required` 的含义，但尚未介绍这么做的原因，本章后面将会通过一些示例进行说明。

## 2. Cocoa 的特殊之处

在继承 Cocoa 类时，初始化器继承规则可能会产生一些奇怪的结果。比如，在编写 iOS 程序时，你肯定会声明 UIViewController 子类。假设该子类声明了一个指定初始化器。父类 UIViewController 中的指定初始化器是 `init(nibName:bundle:)`，因此为了满足规则，你需要像下面这样从指定初始化器中调用它：

```
class ViewController: UIViewController {
    init() {
        super.init(nibName:"MyNib", bundle:nil)
    }
}
```

现在看来一切正常；不过，你会发现创建 ViewController 实例的代码无法编译通过了：

```
let vc = ViewController(nibName:"MyNib", bundle:nil) // compile error
```

只有声明了自己的指定初始化器后，上面的代码才能编译通过；但现在并没有这么做。原因在于，通过在子类中实现指定初始化器，你阻止了初始化器的继承！ ViewController 类过去会继承 UIViewController 的 `init(nibName:bundle:)` 初始化器，但现在却不是这样。你还需要重写该初始化器，即便实现只是调用被重写的初始化器亦如此：

```
class ViewController: UIViewController {
    init() {
        super.init(nibName:"MyNib", bundle:nil)
    }
    override init(nibName: String?, bundle: NSBundle?) {
        super.init(nibName:nibName, bundle:bundle)
    }
}
```

现在，如下实例化 ViewController 的代码可以编译通过了：

```
let vc = ViewController(nibName:"MyNib", bundle:nil) // fine
```

不过，现在又有一个令人惊诧之处：ViewController 本身无法编译通过了！原因在于还有一个施加于 ViewController 之上的必备初始化器，你还需要将其实现出来。之前你是不知道这一点的，因为当 ViewController 没有显式初始化器时，你会将必备初始化器继承下来；现在，你又阻止了继承。幸好，Xcode 的 Fix-It 特性提供了一个桩实现；它什么都没做（事实上，如果调用，程序将会崩溃），不过却满足了编译器的要求：

```
required init?(coder aDecoder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}
```

本章后面将会介绍该必备初始化器是如何应用的。

## 4.4.4 类析构器

只有类才会拥有析构器。它是个通过关键字 deinit 声明的函数，后跟一对花括号，里面是函数体。你永远不会自己调用这个函数；它是当类的实例消亡时由运行时调用的。如果一个类有父类，那么子类的析构器（如果有）会在父类的析构器（如果有）调用之前调用。

析构器的想法在于你可以在实例消亡前执行一些清理工作，或是向控制台打印一些日志，证明操作执行顺序是正确的。我将在第 5 章介绍内存管理主题时使用析构器。

## 4.4.5 类属性与方法

子类可以重写继承下来的属性。重写的属性必须要与继承下来的属性拥有相同的名字与类型，并且要标记为 override（属性与继承下来的属性不能只名字相同而类型不同，因为这样就无法区分它们了）。需要遵循如下新规则：

- 如果父类属性是可写的（存储属性或带有 setter 的计算属性），那么子类在重写时可以添加对该属性的 setter 观察者。
- 此外，子类可以使用计算变量进行重写。在这种情况下：
  - 如果父类属性是存储属性，那么子类的计算变量重写就必须要有 getter 与 setter。
  - 如果父类属性是计算属性，那么子类的计算变量重写就必须要重新实现父类实现的所有访问器。如果父类属性是只读的（只有 getter），那么重写可以添加 setter。

重写属性的函数可以通过 super 关键字引用（读或写）继承下来的属性。

类可以有静态成员，只需将其标记为 static，就像结构体或枚举一样；还可以有类成员，标记为 class。静态与类成员都可以由子类继承（分别作为静态与类成员）。

从程序员的视角来看，静态方法与类方法之间的主要差别在于静态方法无法重写；static 就好像是 class final 的同义词一样。

比如，使用一个静态方法表示狗叫：

```
class Dog {
    static func whatDogsSay() -> String {
        return "woof"
    }
    func bark() {
        print(Dog.whatDogsSay())
    }
}
```

子类现在继承了 whatDogsSay，但却无法重写。Dog 的子类不能包含签名相同的名为 whatDogsSay 的类方法或静态方法实现。

下面使用一个类方法表示狗叫：

```
class Dog {
    class func whatDogsSay() -> String {
        return "woof"
    }
    func bark() {
        print(Dog.whatDogsSay())
    }
}
```

子类继承了 whatDogsSay，并且可以重写，要么作为类函数，要么作为静态函数：

```
class NoisyDog : Dog {
    override class func whatDogsSay() -> String {
        return "WOOF"
    }
}
```

静态属性与类属性之间的差别是类似的，不过还要再增加一条重要差别：静态属性可以是存储属性，而类属性只能是计算属性。

下面通过一个静态类属性来表示狗叫：

```
class Dog {
    static var whatDogsSay = "woof"
    func bark() {
        print(Dog.whatDogsSay)
    }
}
```

子类继承了whatDogsSay，但却无法重写；Dog的子类无法声明类或静态属性whatDogsSay。

现在通过类属性来表示狗叫。它不能是存储属性，因此只能使用计算属性：

```
class Dog {
    class var whatDogsSay : String {
        return "woof"
    }
    func bark() {
        print(Dog.whatDogsSay)
    }
}
```

子类继承了whatDogsSay，并且可以通过类属性或静态属性重写它。不过，正如子类重写的静态属性不能是存储属性一样，这符合之前介绍的关于属性重写的原则：

```
class NoisyDog : Dog {
    override static var whatDogsSay : String {
        return "WOOF"
    }
}
```

## 4.5 多态

如果某个计算机语言有类型与子类型层次，那么它必须要解决这样一个问题：对于对象类型与声明的指向该对象的引用类型之间的关系，这种层次体系意味着什么。Swift遵循着多态的原则。我认为，正是多态的作用才使得基于对象的语言彻底演变为完善的面向对象语言。Swift的多态原则如下所示：

*替代*

在需要某个类型时，我们可以使用该类型的子类型。

*内在一致性*

对象类型的关键在于内部特性，而与对象是如何被引用的毫无关系。

下面来看看这些原则的含义。假设有一个Dog类，它有一个子类NoisyDog：

```
class Dog {
}
class NoisyDog : Dog {
}
let d : Dog = NoisyDog()
```

替代法则表示上述代码最后一行是合法的：我们可以将NoisyDog实例赋给类型为Dog的引用d。内在一致性法则表示，在底层d现在就是个NoisyDog。

你可能会问：如何证明内在一致性规则？如果对 NoisyDog 的引用类型是 Dog，那么它怎么就是 NoisyDog 呢？为了说明这一问题，我们来看看当子类重写了继承下来的方法时会发生什么。下面重新定义 Dog 与 NoisyDog 进行说明：

```
class Dog {
    func bark() {
        print("woof")
    }
}
class NoisyDog : Dog {
    override func bark() {
        super.bark(); super.bark()
    }
}
```

看看下面的代码，想想能否编译通过，如果能，那么结果是什么：

```
func tellToBark(d:Dog) {
    d.bark()
}
var d = NoisyDog()
tellToBark(d)
```

上述代码可以编译通过。我们创建了一个 NoisyDog 实例并将其传给了需要 Dog 参数的函数。这么做是可以的，因为 NoisyDog 是 Dog 的子类（替代）。NoisyDog 可以用在需要 Dog 的地方。从类型上来看，NoisyDog 就是一种 Dog。

不过当代码实际运行并调用 `tellToBark` 函数时，它里面的局部变量 `d` 所引用的对象的 `bark` 会做什么呢？一方面，`d` 的类型是 Dog，Dog 的 `bark` 函数会打印出 `"woof"` 一次；另一方面，当调用 `tellToBark` 时，实际传递的是 NoisyDog 实例，而 NoisyDog 的 bark 函数会打印出 `"woof"` 两次，那结果会是什么呢？下面就来看一下：

```
func tellToBark(d:Dog) {
    d.bark()
}
var d = NoisyDog()
tellToBark(d) // woof woof
```

结果是 `"woof woof"`。内在一致性法则表明在发送消息时，重要的事情并不是如何通过引用来判断消息接收者的类型，而是接收者的实际类型到底是什么。无论持有的变量类型是什么，传递给 `tellToBark` 的是 NoisyDog；因此，`bark` 消息会使得该对象打印出两次 `"woof"`。它是个 NoisyDog！

下面是多态的另一个重要影响：关键字 `self` 的含义。它指的是实际实例，其含义取决于实际实例的类型，即便单词 `self` 出现在父类代码中亦如此。比如：

```
class Dog {
    func bark() {
        print("woof")
    }
    func speak() {
        self.bark()
    }
}
class NoisyDog : Dog {
    override func bark() {
        super.bark(); super.bark()
    }
}
```

调用 NoisyDog 的 speak 方法时会打印什么呢？下面来试一下：

```
let d = NoisyDog()
d.speak() // woof woof
```

speak 方法声明在 Dog 而非 NoisyDog 中，即声明在父类中。speak 方法会调用 bark 方法，这是通过关键字 self 实现的（这里其实可以省略对 self 的显式引用，不过即便如此，self 还是会隐式使用，因此我还是显式使用了 self）。Dog 中有个 bark 方法，NoisyDog 中有个重写的 bark 方法。那么到底会调用哪个 bark 方法呢？

关键字 self 位于 Dog 类的 speak 方法实现中。不过，重要的事情并不是单词 self 在哪里，而是它表示什么含义。它表示当前实例。内在一致性法则告诉我们，当前实例是个 NoisyDog！因此，调用的是 NoisyDog 重写的 bark。

归功于多态，你可以充分利用子类为已有的类增加功能并做更多的定制。这在 iOS 编程世界中尤为重要，其中大多数类都是由 Cocoa 定义的，并不属于你。比如，UIViewController 是由 Cocoa 定义的；它有大量 Cocoa 会调用的内建方法，这些方法会执行各种重要的任务，而且是以一种通用的方式执行的。在实际情况中，你会声明 UIViewController 的子类并重写这些方法来完成适合于特定应用的任务。这对 Cocoa 不会造成任何影响，因为替代法则在发挥着作用，在 Cocoa 期望接收或要调用 UIViewController 时，它可以接收你自己定义的 UIViewController 子类，这么做不会产生任何问题。而且，这种替代行为与你的期望是一致的，因为（内在一致性法则）当 Cocoa 调用子类中的 UIViewController 方法时，真正调用的实际上是子类重写的版本。

多态很酷，不过其速度会慢一些。它需要动态分发，这意味着运行时要思考向类实例发送的消息到底表示什么。这也是在可能的情况下优先使用结构体而非类的另一个原因：结构体无须动态分发。此外，可以通过将类或类成员声明为 final 或 private，以及打开全模块优化（参见第 6 章）来减少动态分发的使用。

# 4.6 类型转换

Swift 编译器有严格的类型限制，它会限制什么消息可以发送给某个对象引用。编译器允许发送给某个对象引用的消息是该引用类型所允许的那些消息，包括继承下来的那些。

由于多态的内在一致性法则，对象可以接收到编译器不允许发送的消息。这有时会让我们不知所措。比如，假设在 NoisyDog 中声明了一个 Dog 所没有的方法：

```
class Dog {
    func bark() {
        print("woof")
    }
}
class NoisyDog : Dog {
    override func bark() {
        super.bark(); super.bark()
    }
    func beQuiet() {
        self.bark()
    }
}
```

在上述代码中，我们在 NoisyDog 中增加了一个 `beQuiet` 方法。现在来看看调用 Dog 类型对象的 `beQuiet` 方法时会发生什么：

```
func tellToHush(d:Dog) {
    d.beQuiet() // compile error
}
let d = NoisyDog()
tellToHush(d)
```

代码无法编译通过。我们不能向该对象发送 `beQuiet` 消息，即便事实上它是个 NoisyDog 并且具有 NoisyDog 方法。这是因为，函数体中的引用 `d` 的类型为 Dog，而 Dog 是没有 `beQuiet` 方法的。这里有点讽刺：我们知道的比编译器还要多！我们知道上述代码是可以正确运行的，因为 `d` 实际上是个 NoisyDog，只要让代码能够编译通过就行。我们需要通过一种方式告知编译器，“请相信我：当程序真正运行时，它实际上是个 NoisyDog，请允许我发送这条消息”。

实际上是有办法做到这一点的，那就是通过类型转换。要想实现类型转换，你需要使用关键字 `as`，后跟真正的类型名。Swift 不允许将一种类型转换为不相干的另一种类型，不过可以将父类转换为子类，这叫作向下类型转换。在进行向下类型转换时，你需要在关键字 `as` 后面加上一个感叹号，即 `as!`。感叹号提醒你在让编译器做一些它本不会做的事情：

```
func tellToHush(d:Dog) {
```

```
    (d as! NoisyDog).beQuiet()
}
let d = NoisyDog()
tellToHush(d)
```

上述代码可以编译通过，并且正常运行。对于该示例来说，更好的写法是下面这样：

```
func tellToHush(d:Dog) {
    let d2 = d as! NoisyDog
    d2.beQuiet()
    d2.beQuiet()
}
let d = NoisyDog()
tellToHush(d)
```

之所以说上面这种写法更好是因为如果还会向该对象发送其他 NoisyDog 消息，那就不用每次都执行类型转换了，我们可以根据内在一致性类型只转换对象一次，并将其赋给一个变量。既然可以根据类型转换推测出变量的类型（即内在一致性类型），我们就可以向该变量发送多条消息了。

我说过 as! 运算符的感叹号会提醒你强制编译器进行转换。它还有警告的作用：代码可能会崩溃！原因在于你可能对编译器撒谎。向下类型转换会让编译器放松其严格的类型检查，让你能够正常调用。如果使用类型转换做了错误的声明，那么编译器还是会允许你这么做，不过当应用运行时就会崩溃：

```
func tellToHush(d:Dog) {
    (d as! NoisyDog).beQuiet() // compiles, but prepare to crash...!
}
let d = Dog()
tellToHush(d)
```

在上述代码中，我们告诉编译器该对象是个 NoisyDog，编译器选择相信我们，并允许我们向该对象发送 beQuiet 消息。不过事实上，当代码运行时，该对象是个 Dog，因此由于该对象并不是 NoisyDog，类型转换会失败，程序将会崩溃。

为了防止这种错误，你可以在运行时测试实例的类型。一种方式是使用关键字 is。你可以在条件中使用 is；判断通过后再转换，这样转换就是安全的了：

```
func tellToHush(d:Dog) {
    if d is NoisyDog {
        let d2 = d as! NoisyDog
        d2.beQuiet()
    }
}
```

结果是这样的：除非 d 真的是 NoisyDog，否则我们不会将其转换为 NoisyDog。

解决这个问题的另一种方式是使用 Swift 的 as? 运算符。它也会进行向下类型转换，不过提供了失败的选项；因此，它转换的结果是个 Optional（你可能已经猜出来了），现在回到了我们熟知的领域，因为我们已经知道如何安全地处理 Optional 了：

```
func tellToHush(d:Dog) {
    let noisyMaybe = d as? NoisyDog // an Optional wrapping a NoisyDog
    if noisyMaybe != nil {
        noisyMaybe!.beQuiet()
    }
}
```

这与之前的做法相比并没有简洁多少。不过，还记得我们可以通过展开 Optional 向一个 Optional 发送消息吧！因此，我们可以省略赋值并将代码压缩到一行：

```
func tellToHush(d:Dog) {
    (d as? NoisyDog)?.beQuiet()
}
```

首先，我们通过 as? 运算符获取到一个包装了 NoisyDog（或者是 nil）的 Optional。接下来展开该 Optional，并向其发送了一条消息。如果 d 不是 NoisyDog，那么该 Optional 就是 nil，消息也不会发送。如果 d 是 NoisyDog，那么该 Optional 将会展开，消息也会发送出去。这样，代码就是安全的。

回忆一下第 3 章，对 Optional 使用比较运算符会自动应用到该 Optional 所包装的对象上。as!、as? 与 is 运算符的工作方式是一样的。如果有一个包装了 Dog 的 Optional d（也就是说，d 是个 Dog? 对象），那么它实际上会包装一个 Dog 或 NoisyDog；替换法则对 Optional 类型也适用，因为它对 Optional 所包装的类型适用。要想知道它到底包装的是什么，你可能会使用 is，是吗？毕竟，这个 Optional 既不是 Dog 也不是 NoisyDog，它是个 Optional！好消息是 Swift 知道你的想法；如果 is 左边的是个 Optional，那么 Swift 就会认为它是包装在 Optional 中的值。这样，其工作方式与你期望的就一致了：

```
let d : Dog? = NoisyDog()
if d is NoisyDog { // it is!
```

如果对 Optional 使用 is，那么如果该 Optional 为 nil，测试就会失败。is 实际上做了两件事：它会检查 Optional 是否为 nil，如果不是，那么它会继续检查被包装的值是否是我们所指定的类型。

那么类型转换呢？你不能将 Optional 转换为任何其他类型。不过，你可以对 Optional 使用 as! 运算符，因为 Swift 知道你的想法；如果 as! 左侧是 Optional，那么 Swift 就会将其当作被包装的类型。此外，使用 as! 运算符会做两件事情：Swift 首先展开 Optional，然后进行类型转换。如下代码可以正常运行，因为 d 被展开得到 d2，它是个 NoisyDog：

```
let d : Dog? = NoisyDog()
let d2 = d as! NoisyDog
d2.beQuiet()
```

不过，上述代码并不安全。你不应该在不测试的情况下就进行类型转换，除非你对要做的事情很有把握。如果 d 为 nil，那么第 2 行代码就会崩溃，因为这时你所展开的是一个 nil Optional。如果 d 是个 Dog 而非 NoisyDog，那么类型转换还是会失败，第 2 行代码依然会崩溃。这正是 as? 运算符存在的原因，它是安全的，不过会生成一个 Optional：

```
let d : Dog? = NoisyDog()
let d2 = d as? NoisyDog
d2?.beQuiet()
```

还有一种情况会用到类型转换，那就是在进行 Swift 与 Objective-C 值交换时（两个类型是相同的）。比如，你可以将 Swift String 转换为 Cocoa NSString，反之亦然。这并不是因为其中一个是另一个的子类，而是因为它们之间可以彼此桥接；它们本质上是相同的类型。在从 String 转换为 NSString 时，其实并没有做向下类型转换，你所做的事情并没有什么不安全的，因此可以使用 as 运算符，不需要使用感叹号。第 3 章给出了一个示例，介绍了什么情况下需要这么做，如下代码所示：

```
let s = "hello"
let range = (s as NSString).rangeOfString("ell") // (1,3), an NSRange
```

从 String 到 NSString 的转换告诉 Swift，在调用 rangeOfString 时要使用 Cocoa，这样结果就是 Cocoa 了，即一个 NSRange 而非 Swift Range。

Swift 与 Objective-C 中的很多常见类都是通过这种方式桥接的。通常，在从 Swift 到 Objective-C 时并不需要进行转换，因为 Swift 会自动进行转换。比如，Swift Int 与 Cocoa NSNumber 是完全不同的两种类型；不过，你可以在需要 NSNumber 的地方使用 Int，无须进行转换，如下代码所示：

```
let ud = NSUserDefaults.standardUserDefaults()
ud.setObject(1, forKey: "Test")
```

在上述代码中，我们在 Objective-C 期望 NSObject 实例的地方使用了 Int（即 1）。Int 并非 NSObject 实例；它甚至都不是类实例（它是个结构体实例）。不过，Swift 发现这个地方需要 NSObject，并且确定 NSNumber 最适合表示 Int，于是帮你进行了桥接。因此，存储在 NSUserDefaults 中的实际上是个 NSNumber。

不过，在调用 objectForKey: 时，Swift 并不知道这个值实际上是什么，因此如果需要 Int 时就得显式进行转换，这里做的就是向下类型转换（稍后将会对其进行详细介绍）：

```
let i = ud.objectForKey("Test") as! Int
```

上述转换是正确的，因为ud.objectForKey("Test")会生成一个包装整型的NSNumber，将其转换为Swift Int是可行的，类型之间会桥接起来。不过，如果ud.objectForKey("Test")不是NSNumber(或是nil)，那么程序将会崩溃。如果不确定，请使用as?确保安全。

# 4.7 类型引用

实例引用自己的类型是很有用的，比如，向该类型发送消息。在之前的示例中，Dog实例方法通过显式向Dog类型发送一条消息来获取到一个Dog类属性，这是通过使用Dog这个单词做到的：

```
class Dog {
    class var whatDogsSay : String {
        return "Woof"
    }
    func bark() {
        print(Dog.whatDogsSay)
    }
}
```

表达式Dog.whatDogsSay看起来很笨拙并且不灵活。为什么要在Dog类中使用硬编码的类名呢？它有一个类；它应该知道是什么。

在Objective-C中，我们习惯使用类实例方法来处理这种情况。在Swift中，实例可能不是类（可能是结构体实例或枚举实例）；Swift实例拥有类型。Swift针对这一目的提供了一个名为dynamicType的实例方法。实例可以通过该方法访问其类型。因此，如果不想显式使用Dog来通过Dog实例调用Dog类方法，那么下面就是另一种解决方案：

```
class Dog {
    class var whatDogsSay : String {
        return "Woof"
    }
    func bark() {
        print(self.dynamicType.whatDogsSay)
    }
}
```

使用dynamicType而非硬编码类名的重要之处在于它遵循了多态：

```
class Dog {
    class var whatDogsSay : String {
        return "Woof"
    }
    func bark() {
        print(self.dynamicType.whatDogsSay)
```

```
    }
}
class NoisyDog : Dog {
    override class var whatDogsSay : String {
        return "Woof woof woof"
    }
}
```

下面来看一下结果：

```
let nd = NoisyDog()
nd.bark() // Woof woof woof
```

如果调用 NoisyDog 的 bark 方法，那么会打印出 "Woof woof woof"。原因在于 dynamicType 的含义是“该实例现在的实际类型，这正是该类型成为动态的原因所在”。我们向 NoisyDog 实例发送了“bark”消息。bark 实现引用了 self.dynamicType；self 表示当前实例，即 NoisyDog，因此 self.dynamicType 是 NoisyDog 类，它是获取到的 NoisyDog 的 whatDogsSay。

还可以通过 dynamicType 获取到对象类型的名字（字符串），这通常用于调试的目的。在调用 print(myObject.dynamicType) 时，控制台上会打印出类型名。

在某些情况下，你会将对象类型作为值进行传递。这么做是合法的；对象类型本身也是对象。下面是你需要知道的几点：

- 声明接收某个对象类型（如作为变量或参数的类型），请使用点符号加上类型名与关键字 Type。
- 将对象类型作为值（比如，为变量指定类型或将类型传递给函数），请使用类型引用（类型名，或实例的 dynamicType），后面可以通过点符号跟着关键字 self。

比如，下面这个函数的参数会接收一个 Dog 类型：

```
func typeExpecter(whattype:Dog.Type) {
}
```

如下代码调用了该函数：

```
typeExpecter(Dog) // or: typeExpecter(Dog.self)
```

还可以像下面这样调用：

```
let d = Dog() // or: let d = NoisyDog()
typeExpecter(d.dynamicType) // or: typeExpecter(d.dynamicType.self)
```

为何要这么做呢？典型场景就是函数是个实例工厂：给定一个类型，它会创建出该类型的实例，可能还会做一些处理，然后将其返回。你可以使用指向类型的变量引用来创建该类型的实例，方式是向其发送一条 init(...) 消息。

比如，如下 Dog 类带有一个 init(name:) 初始化器，同时它还有一个子类 NoisyDog：

```
class Dog {
    var name : String
    init(name:String) {
        self.name = name
    }
}
class NoisyDog : Dog {
}
```

下面是创建 Dog 或 NoisyDog 的工厂方法（通过参数来指定），为实例指定一个名字，然后将实例返回：

```
func dogMakerAndNamer(whattype:Dog.Type) -> Dog {
    let d = whattype.init(name:"Fido") // compile error
    return d
}
```

如你所见，由于 whattype 引用了一个类型，所以可以调用其初始化器创建该类型的实例，不过有一个问题，上述代码无法编译通过。原因在于编译器不能确定 init(name:) 初始化器是否会被 Dog 的每个子类型所实现。为了消除编译器的这个疑虑，我们需要通过 required 关键字声明该初始化器：

```
class Dog {
    var name : String
    required init(name:String) {
        self.name = name
    }
}
class NoisyDog : Dog {
}
```

现在来解释一下需要将初始化器声明为 required 的原因：required 会消除编译器的疑虑；Dog 的每个子类都要继承或重新实现 init(name:)；因此，可以向指向 Dog 或其子类的类型引用发送 init(name:) 消息。现在代码可以编译通过，我们也可以调用函数：

```
let d = dogMakerAndNamer(Dog) // d is a Dog named Fido
let d2 = dogMakerAndNamer(NoisyDog) // d2 is a NoisyDog named Fido
```

在类方法中，self 表示类，这正是多态发挥作用之处。这意味着在类方法中，你可以向 self 发送消息，以多态的形式调用初始化器。下面是一个示例，我们将实例工厂方法移到 Dog 中，作为一个类方法，并将这个类方法命名为 makeAndName。我们需要这

个类方法创建并返回一个具名 Dog，返回的 Dog 类型取决于向哪个类发送 makeAndName 消息。如果调用 Dog.makeAndName()，那么返回的是 Dog；如果调用 NoisyDog.makeAndName()，那么返回的是 NoisyDog。类型是多态的 self 类，因此 makeAndName 类方法可以初始化 self：

```
class Dog {
    var name : String
    required init(name:String) {
        self.name = name
    }
    class func makeAndName() -> Dog {
        let d = self.init(name:"Fido")
        return d
    }
}
class NoisyDog : Dog {
}
```

其工作方式与我们期望的一致：

```
let d = Dog.makeAndName() // d is a Dog named Fido
let d2 = NoisyDog.makeAndName() // d2 is a NoisyDog named Fido
```

不过有一个问题。虽然 d2 实际上是个 NoisyDog，但其类型却是 Dog。这是因为 makeAndName 类方法在声明时返回的类型是 Dog，这并不是我们想要的结果。我们希望该方法所返回的实例类型与接收 makeAndName 消息的类型保持一致。换句话说，我们需要一种多态化的类型声明！这个类型是 Self（注意，首字母是大写的）。它用作方法声明的返回类型，表示“运行时该类型的实例”。如下：

```
class Dog {
    var name : String
    required init(name:String) {
        self.name = name
    }
    class func makeAndName() -> Self {
        let d = self.init(name:"Fido")
        return d
    }
}
class NoisyDog : Dog {
}
```

现在，当调用 NoisyDog.makeAndName() 时，我们将会得到类型为 NoisyDog 的 NoisyDog。

Self 也可以用于实例方法声明。因此，我们可以编写出该工厂方法的实例方法版本。下面是 Dog 类与 NoisyDog 类的声明，同时在 Dog 类中声明了一个返回 Self 的 havePuppy 方法：

```
class Dog {
    var name : String
    required init(name:String) {
        self.name = name
    }
   func havePuppy(name name:String) -> Self {
       return self.dynamicType.init(name:name)
    }
}
class NoisyDog : Dog {
}
```

下面是测试代码：

```
let d = Dog(name:"Fido")
let d2 = d.havePuppy(name:"Fido Junior")
let nd = NoisyDog(name:"Rover")
let nd2 = nd.havePuppy(name:"Rover Junior")
```

如你所想，d2 是个 Dog，不过 nd2 却是个类型为 NoisyDog 的 NoisyDog。

讲了这么多可能有些乱，下面来总结一下：

`.dynamicType`

用在代码中，发送给实例：表示该实例的多态化（内部）类型，无论实例引用的类型是什么均如此。静态 / 类成员可以通过实例的 dynamicType 进行访问。

`.Type`

用在声明中，发送给类型：多态类型而不是类型的实例。比如，在函数声明中，Dog 表示需要一个 Dog 实例（或其子类的实例），而 Dog.Type 则表示需要 Dog 类型本身（或是其子类的类型）。

`.self`

用在代码中，发送给类型。比如，在需要 Dog.Type 时，你可以传递 Dog.self（向实例发送 .self 是合法的，但却毫无意义）。

`self`

用在实例代码中表示多态语义下的当前实例。

用在静态 / 类代码中表示多态语义下的类型；self.init(...) 会实例化该类型。

`Self`

在方法声明中，如果指定了返回类型，那么它表示多态化的类或实例的类。

# 4.8 协议

协议是一种表示不相关类型共性的方式。比如，Bee 对象与 Bird 对象可能有一些共性，

因为蜜蜂与鸟都能飞。因此，定义一个 Flier 类型会好一些；但问题在于：让 Bee 与 Bird 都成为 Fliers 会有多大的意义呢？

当然了，一种可能是使用类继承。如果 Bee 与 Bird 都是类，那就存在一种父类与子类的类继承。这样，Flier 就是 Bee 与 Bird 的父类。问题在于，可能存在其他一些原因使得 Flier 不能作为 Bee 与 Bird 的父类。Bee 是个 Insect，而 Bird 不是；但它们都可以飞，这是彼此独立的能力。我们需要一种类型可以某种方式透过类继承体系，将不相关的类集成到一起。

此外，如果 Bee 与 Bird 都不是类该怎么办呢？在 Swift 中，这是非常有可能的。重要且强大的对象可以是结构体而非类，不过并不存在父结构体与子结构体的结构体层次体系。毕竟，这是结构体与类之间的一个主要差别。但结构体也需要像类一样拥有和表达正常的共性特性。Bee 结构体与 Bird 结构体怎么可能都是 Fliers 呢？

Swift 通过协议解决了这一问题。协议在 Swift 中是非常重要的；Swift 头文件中定义了 70 多个协议！此外，Objective-C 也支持协议；Swift 协议大体上与 Objective-C 协议一致，并且可以与之交换。Cocoa 大量使用了协议。

协议是一种对象类型，不过并没有协议对象——你无法实例化协议。协议要更加轻量级一些。协议声明仅仅是一些属性与方法列表而已。属性没有值，方法没有代码！其想法是“真实”的对象类型可以声明它属于某个协议类型；这叫作使用或遵循协议。使用协议的对象类型会遵守这样一个契约：它会实现协议所列出的属性与方法。

比如，假设成为 Flier 需要实现一个 `fly` 方法；那么，Flier 协议可以指定必须要有一个 `fly` 方法；为了做到这一点，它会列出 `fly` 方法，但却没有函数体，如下代码所示：

```
protocol Flier {
    func fly()
}
```

任何类型（枚举、结构体、类，甚至是另一个协议）都可以使用该协议。为了做到这一点，它需要在声明中的名字后面加上一个冒号，后跟协议名（如果使用者是个拥有父类的类，那么父类后面还需要加上一个逗号，协议则位于该逗号后面）。

假设 Bird 是个结构体，那么它可以像下面这样使用 Flier：

```
struct Bird : Flier {
} // compile error
```

目前来看一切都没问题，不过上述代码无法编译通过。Bird 结构体承诺要实现 Flier 协议的特性，现在它必须要履行承诺！`fly` 方法是 Flier 协议的唯一要求。为了满足这一点，我在 Bird 中增加了一个空的 `fly` 方法：

```
protocol Flier {
    func fly()
}
struct Bird : Flier {
    func fly() {
    }
}
```

这么做就没问题了！我们定义了一个协议，并且让一个结构体使用该协议。当然了，在实际开发中，你可能希望使用者对协议方法的实现能够做一些事情；不过，协议对此并没有做任何规定。

在 Swift 2.0 中，协议可以声明方法并提供实现，这要归功于协议扩展，本章后面将会对此进行介绍。

## 4.8.1 为何使用协议

也许到这个时候你还不太理解协议到底有什么用。我们让 Bird 成为一个 Flier，然后呢？如果想让 Bird 知道如何飞，为什么不在 Bird 中声明一个 `fly` 方法，这样就无须使用任何协议了。这个问题的答案与类型有关。别忘了，协议是一种类型；我们的协议 Flier 是一种类型。因此，我可以在需要类型的时候使用 Flier。比如，可以用它声明变量的类型，或函数参数的类型：

```
func tellToFly(f:Flier) {
    f.fly()
}
```

仔细想想上面的代码，因为它体现了协议的精髓。协议是一种类型，因此适用于多态。协议赋予我们表达类与子类概念的另一种方式。这意味着，根据替代法则，这里的 Flier 可以是任何对象类型的实例：枚举、结构体或类。对象类型是什么不重要，只要它使用了 Flier 协议即可。如果使用了 Flier 协议，那么它就会有 `fly` 方法，因为这是使用 Flier 协议所要求的！因此，编译器允许我们向该对象发送 `fly` 消息。根据定义，Flier 是个可以接收 `fly` 消息的对象。

不过，反过来就不行了：拥有 `fly` 方法的对象不一定就是 Flier。它不一定遵循了协议的要求；对象类型必须要使用协议。如下代码将无法编译通过：

```
struct Bee {
    func fly() {
    }
}
```

```
let b = Bee()
tellToFly(b) // compile error
```

Bee 可以接收 fly 消息，这是以 Bee 的身份做的。不过，tellToFly 并不接收 Bee 参数；它接收的是 Flier 参数。形式上，Bee 并非 Flier。要想让 Bee 成为 Flier，只需形式上声明 Bee 使用了 Flier 协议。如下代码可以编译通过：

```
struct Bee : Flier {
    func fly() {
    }
}
let b = Bee()
tellToFly(b)
```

关于鸟与蜜蜂的示例到此为止，下面来看看实际的示例吧！如前所述，Swift 已经提供了大量的协议，下面让我们自定义的类型使用其中一个协议。Swift 提供的最有用的协议之一是 CustomStringConvertible。CustomStringConvertible 协议要求我们实现一个 description String 属性。如果这么做了，那就会有奇迹发生：在将该类型的实例用在字符串插入或 print 中时（或是控制台中的 po 命令），description 属性就会自动用来表示该实例。

回忆一下本章之前介绍的 Filter 枚举，我向其中添加一个 description 属性：

```
enum Filter : String {
    case Albums = "Albums"
    case Playlists = "Playlists"
    case Podcasts = "Podcasts"
    case Books = "Audiobooks"
    var description : String { return self.rawValue }
}
```

不过，这么做还不足以让 Filter 具备 CustomStringConvertible 协议的功能；要想做到这一点，我们还需要正式使用 CustomStringConvertible 协议。Filter 声明中已经有了一个冒号与类型，因此所使用的协议需要放在逗号后面：

```
enum Filter : String, CustomStringConvertible {
    case Albums = "Albums"
    case Playlists = "Playlists"
    case Podcasts = "Podcasts"
    case Books = "Audiobooks"
    var description : String { return self.rawValue }
}
```

现在，Filter 已经正式使用 CustomStringConvertible 协议了。CustomStringConvertible 协议要求我们实现一个 description String 属性；我们已经实现了一个 description String 属性，因此代码可以编译通过。现在可以向 print 传递一个 Filter 或将其插入到一个字符串中，其 description 将会被自动打印出来：

```
let type = Filter.Albums
print(type) // Albums
print("It is \(type)") // It is Albums
```

看到协议的强大威力了吧，你可以通过相同方式为任何对象类型赋予字符串转换的能力。

一个类型可以使用多个协议！比如，内建的Double类型就使用了CustomStringConvertible、Hashable、Comparable和其他内建协议。要想声明使用多个协议，请在声明中将每个协议列在第一个协议后面，中间用逗号分隔。比如：

```
struct MyType : CustomStringConvertible, Hashable, Comparable {
    // ...
}
```

（当然，除非在MyType中声明所需的方法，否则上述代码将无法编译通过；声明完之后，MyType就真正使用了这些协议）。

## 4.8.2 协议类型测试与转换

协议是一种类型，协议的使用者是其子类型，这里使用了多态。因此，用于对象真实类型的那些运算符也可以用于声明为协议类型的对象。比如，Flier协议被Bird与Bee使用了，那么我们就可以通过 `is` 运算符测试某个Flier是否为Bird：

```
func isBird(f:Flier) -> Bool {
    return f is Bird
}
```

与之类似，`as!` 与 `as?` 可用于将声明为协议类型的对象向下转换为其真正的类型。这是非常重要的，因为使用协议的对象可以接收协议无法接收的消息。比如，假设Bird有个getWorm方法：

```
struct Bird : Flier {
    func fly() {
    }
    func getWorm() {
    }
}
```

Bird能以Flier身份 `fly`，但却只能以Bird身份getWorm，你不能让任意一个Flier去getWorm：

```
func tellGetWorm(f:Flier) {
    f.getWorm() // compile error
}
```

不过，如果这个Flier是个Bird，那么它显然可以getWorm，这正是类型转换要做的事情：

```
func tellGetWorm(f:Flier) {
    (f as? Bird)?.getWorm()
}
```

## 4.8.3 声明协议

只能在文件顶部声明协议。要想声明协议，请使用关键字 protocol，后跟协议名；作为一种对象类型，协议名首字母应该是大写的。接下来是一对花括号，里面可以包含如下内容：

*属性*

协议中的属性声明包含了 var（不是 let）、属性名、冒号、类型，以及包含单词 get 或 get set 的一对花括号。对于前者来说，使用者对该属性的实现是可写的；对于后者来说，它需要满足如下规则：使用者不可以将 get set 属性实现为只读计算属性或常量（let）存储属性。

要想声明静态 / 类属性，请在前面加上关键字 static。类使用者可以将其实现为类属性。

*方法*

协议中的方法声明是个没有函数体的函数声明，即没有花括号，因此也没有代码。任何对象函数类型都是合法的，包括 init 与下标（在协议中声明下标的语法与在对象类型中声明下标的语法是相同的，只不过没有函数体，就像协议中的属性声明一样，它也可以包含 get 或 get set）。

要想声明静态 / 类方法，请在前面加上关键字 static。类使用者可以将其实现为类方法。

如果方法（由枚举或结构体实现）想要声明为 mutating，那么协议就必须指定 mutating 指令；如果协议没有指定 mutating，那么使用者将无法添加。不过，如果协议指定了 mutating，那么使用者可以将其省略。

*类型别名*

协议可以通过声明类型别名为声明中的类型指定局部同义词。比如，通过 typealias Time = Double 可以在协议花括号中使用 Time 类型；在其他地方（比如，使用协议的对象类型中）则不存在 Time 类型，不过可以使用 Double 类型。

在协议中还可以通过其他方式使用类型别名，稍后将会介绍。

*协议使用*

协议本身还可以使用一个或多个协议；语法与你想象的一样，声明中的协议名后面是一个冒号，后面跟着它所使用的协议列表，中间用逗号分隔。事实上，这种方式创建了一个二级类型层次！ Swift 头文件中大量充斥了这种用法。

出于清晰的目的，使用了另一个协议的协议可以重复被使用的协议花括号中的内容，但不必这么做，因为这种重复是隐式的。使用了这种协议的对象类型必须要满足该协议以及该协议使用的所有协议的要求。

如果协议的唯一目的是将其他协议组合起来，但不会添加任何新功能，并且这种组合仅仅用在代码中的一个地方，那么可以通过即时创建组合协议以避免声明协议。要想做到这一点，请使用类型名 `protocol<...,...>`，其中尖括号中的内容是个逗号分隔的协议列表。

## 4.8.4 可选协议成员

在 Objective-C 中，协议成员可以声明为 optional，表示该成员不必被使用者实现，但也可以实现。为了与 Objective-C 保持兼容，Swift 也支持可选协议成员，不过只用于显式与 Objective-C 桥接的协议，方式是在声明前加上 `@objc` 属性。在这种协议中，可选成员（方法或属性）是通过在声明前加上 `optional` 关键字实现的：

```
@objc protocol Flier {
    optional var song : String {get}
    optional func sing()
}
```

只有类可以使用这种协议，并且符合如下两种情况之一才能使用该特性：类是 NSObject 子类，或者可选成员被标记为 `@objc` 特性：

```
class Bird : Flier {
    @objc func sing() {
        print("tweet")
    }
}
```

可选成员不保证会被使用者实现，因此 Swift 并不知晓向 Flier 发送 `song` 消息或 `sing` 消息是否安全。

对于 `song` 这样的可选属性来说，Swift 通过将其值包装到 Optional 中来解决这个问题。如果 Flier 使用者没有实现该属性，那么结果就是 `nil`，并不会出现什么问题：

```
let f : Flier = Bird()
let s = f.song // s is an Optional wrapping a String
```

这是很少会出现的要使用双重 Optional 的一种情况。比如，如果可选属性 `song` 的值是个 `String?`，那么从 Flier 中获取其值就会得到一个 `String??`。

可选属性可以由协议声明为{get set}，不过并没有相关的语法可以设置该协议类型对象中的这种属性。比如，如果f是个Flier，其song被声明为{get set}，那么你就不能设置f.song。我认为这是语言的一个Bug。

对于像sing这样的可选方法来说，事情将变得更为复杂。如果方法没有实现，那么我们就不可以调用它。为了解决这一问题，方法本身会被自动变成其所声明类型的Optional版本。因此，要想向Flier发送sing消息，你需要将其展开。安全的做法是以可选的方式展开它，使用一个问号：

```
let f : Flier = Bird()
f.sing?()
```

上述代码可以编译通过，也可以安全地运行。效果相当于只有当f实现了sing时才向其发送sing消息。如果使用者的实际类型并未实现sing，那么什么都不会发生。虽然可以强制展开调用（f.sing!()），不过如果使用者没有实现sing，那么应用将会崩溃。

如果可选方法返回一个值，那么它也会被包装到Optional中。比如：

```
@objc protocol Flier {
    optional var song : String {get}
    optional func sing() -> String
}
```

如果现在在Flier上调用sing?()，那么结果就是一个包装了String的Optional：

```
let f : Flier = Bird()
let s = f.sing?() // s is an Optional wrapping a String
```

如果强制展开调用（sing!()），那么结果要么是一个String（如果使用者实现了sing），要么应用崩溃（如果使用者没有实现sing）。

很多Cocoa协议都有可选成员。比如，iOS应用会有一个应用委托类，它使用了UIApplicationDelegate协议；该协议有很多方法，所有方法都是可选的。不过，这对如何实现这些方法是没有影响的；你无须通过任何特殊的方式标记它们。应用委托类已经是NSObject的子类，因此该特性可以正常使用，无论是否实现了方法都如此。与之类似，你常常会让UIViewController子类使用带有可选成员的Cocoa委托协议；它也是NSObject的子类，因此你只需实现想要实现的那些方法，不必做任何特殊的标记。（第10章将会深入介绍Cocoa协议，第11章则会深入介绍委托协议。）

## 4.8.5 类协议

名字后面的冒号后使用关键字 `class` 声明的协议是类协议，表示该协议只能由类对象类型使用：

```
protocol SecondViewControllerDelegate : class {
    func acceptData(data:AnyObject!)
}
```

（如果协议已经被标记为 `@objc`，那就无须使用 `class`；`@objc` 特性隐含表示这还是个类协议。）

声明类协议的典型目的在于利用专属于类的内存管理特性。目前还没有介绍过内存管理，不过还是先给出示例吧（第 5 章介绍内存管理时还会探讨这个主题）。

```
class SecondViewController : UIViewController {
    weak var delegate : SecondViewControllerDelegate?
    // ...
}
```

关键字 `weak` 标识 `delegate` 属性将会使用特殊的内存管理，只有类实例可以使用这种特殊的内存管理。`delegate` 属性的类型是个协议，而协议可以由结构体或枚举类型使用。为了告诉编译器该对象实际上是个类实例而非结构体或枚举实例，这里的协议被声明成了类协议。

## 4.8.6 隐式必备初始化器

假设协议声明了一个初始化器，同时一个类使用了该协议。根据协议的约定，该类及其子类必须要实现这个初始化器。因此，该类不仅要实现该初始化器，还要将其标记为 `required`。这样，声明在协议中的初始化器就是隐式必备的，而类则需要显式满足这个要求。

下面这个简单的示例是无法通过编译的：

```
protocol Flier {
    init()
}
class Bird : Flier {
    init() {} // compile error
}
```

上述代码会产生一段详细且信息丰富的编译错误消息：“Initializer requirement `init()` can only be satisfied by a `required` initializer in non-final class Bird.”要想让代码编译通过，我们需要将初始化器指定为 `required`。

```
protocol Flier {
    init()
}
class Bird : Flier {
    required init() {}
}
```

正如编译错误消息所示，我们可以将 Bird 类标记为 final。这意味着它不能有任何子类，从而确保这个问题不会再出现。如果将 Bird 标记为 final，那就没必要将 init 标记为 required 了。

在上述代码中，我们并未将Bird标记为final，但其init被标记为了required。如前所述，这意味着如果 Bird 实现了指定初始化器（从而丧失了初始化器的继承），那么其子类就必须要实现必备初始化器，并将其标记为 required。

该解决方案用于处理本章之前提到的 Swift iOS 编程中一个奇怪、恼人的特性。假设继承了内建的 Cocoa 类 UIViewController（很多时候你都会这么做），并且为子类添加了一个初始化器（很多时候你也会这么做）：

```
class ViewController: UIViewController {
    init() {
        super.init(nibName: "ViewController", bundle: nil)
    }
}
```

上述代码无法编译通过，编译器会报错："required initializer init(coder:) must be provided by subclass of UIViewController."

我们需要理解所发生的事情。UIViewController 使用了协议 NSCoding。该协议需要一个初始化器 init(coder:)。不过，这些都不是你做的；UIViewController 与 NSCoding 是由 Cocoa 而不是你声明的。但这都没关系！这与上述情况一样。你的 UIViewController 子类要么继承 init(coder:)，要么显式实现它并将其标记为 required。由于子类已经实现了自己的指定初始化器（从而丧失了初始化器继承），因此它还需要实现 init(coder:) 并将其标记为 required！

不过，如果不希望在 UIViewController 子类中调用 init(coder:)，这样做就没什么用了。这么做只不过是提供了一个没什么用处的初始化器而已。幸好，Xcode 的 Fix-It 特性会帮助你生成这个初始化器，如下代码所示：

```
required init?(coder aDecoder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}
```

上述代码符合编译器的要求。（第 5 章将会介绍为什么说它不符合初始化器的契约，但

还是一个合法的初始化器。）如果调用这个初始化器，那么程序就会崩溃，这是有意而为之的。

如果希望这个初始化器完成一些功能，那么请删除 `fatalError` 这一行，然后插入自己的功能实现代码。一个有意义且代码量最小的实现是 `super.init(coder:aDecoder)`；当然，如果类有需要初始化的属性，那就需要先初始化它们。

除了 UIViewController，还有很多内建的 Cocoa 类都使用了 NSCoding。在继承这些类并实现自己的初始化器时就会遇到这个问题，你得习惯才行。

## 4.8.7 字面值转换

Swift的精妙之处在于，相对于内建以及魔法实现，它的很多特性都是由Swift本身实现的，并且可以通过 Swift 头文件一探究竟，字面值就是这样的。相对于通过 `Int(5)` 来初始化一个 Int，你可以直接将 5 赋给它，其原因并不是来自于什么神奇魔法，而是因为 Int 使用了协议 IntegerLiteralConvertible。除了 Int 字面值，所有字面值均如此。如下字面值转换协议都声明在 Swift 头文件中：

- NilLiteralConvertible
- BooleanLiteralConvertible
- IntegerLiteralConvertible
- FloatLiteralConvertible
- StringLiteralConvertible
- ExtendedGraphemeClusterLiteralConvertible
- UnicodeScalarLiteralConvertible
- ArrayLiteralConvertible
- DictionaryLiteralConvertible

你自己定义的对象类型也可以使用字面值转换协议，这意味着可以在需要对象类型实例的情况下使用字面值！比如，下面声明了一个 Nest 类型，它包含了一些鸡蛋（即 `eggCount`）：

```
struct Nest : IntegerLiteralConvertible {
    var eggCount : Int = 0
    init() {}
    init(integerLiteral val: Int) {
        self.eggCount = val
```

```
    }
}
```

由于 Nest 使用了 IntegerLiteralConvertible，我们可以在需要 Nest 的地方使用 Int，init(integerLiteral:) 会自动调用，这会创建一个具有指定 eggCount 的全新 Nest 对象：

```
func reportEggs(nest:Nest) {
    print("this nest contains \(nest.eggCount) eggs")
}
reportEggs(4) // this nest contains 4 eggs
```

# 4.9 泛型

泛型是一种类型占位符，实际的类型会在稍后进行填充。由于 Swift 有严格的类型，所以泛型是非常有用的一个特性。在不牺牲严格类型的情况下，有时你不能或是不想在代码中的某处精确指定类型。

重要的是要理解泛型并没有放松 Swift 严格的类型。特别地，泛型并未将类型解析推迟到运行期。在使用泛型时，代码依然要指定真实的类型；这个真实的类型在编译期就会完全指定好！代码中如果需要某个类型，那么可以使用泛型，这样就不必完全指定好类型了，不过当其他代码使用这部分代码时就需要指定好类型。占位符就是泛型，不过在使用泛型时，它会被解析为实际的特定类型。

Optional 就是个很好的示例。任何类型的值都可以包装到 Optional 中，不过你永远不必担心某个 Optional 中包装的是什么类型，这是怎么做到的呢？因为 Optional 是个泛型类型，这正是 Optional 的工作原理。

我之前已经说过 Optional 是个枚举，它有两个 Case：.None 与 .Some。如果 Optional 的 Case 是 .Some，那么它就会有一个关联值，即被该 Optional 所包装的值。不过这个关联值的类型是什么呢？一方面，我们会说它可以是任何类型；毕竟，任何东西都可以被包装到 Optional 中。另一方面，包装某个值的任何 Optional 都会包装某个特定类型的值。在展开 Optional 时，被展开的值需要转换为它原本的类型，这样才能向其发送恰当的消息。

该问题的解决方案就是 Swift 泛型。Swift 头文件中 Optional 枚举声明的开头如下代码所示：

```
enum Optional< Wrapped> {
    // ...
}
```

上述语法表示："在声明中，我使用了一个假的类型（类型占位符），叫作 Wrapped。"它是个真实且单一的类型，不过现在不想过多地表示它的信息。你需要知

道的是，当我说 Wrapped 时，我指的是一个特定的类型。在创建实际的 Optional 时，类型 Wrapped 的含义就一目了然了，接下来我再说 Wrapped 时，你应该将其替换为它所表示的类型。

下面再来看看 Optional 声明：

```
enum Optional<Wrapped> {
    case None
    case Some(Wrapped)
    init(_ some: Wrapped)
    // ...
}>
```

我们已经将 Wrapped 声明成了一个占位符，接下来就可以使用它了。有一个 Case 为 .None，还有一个 Case 为 .Some，它有一个关联值，类型为 Wrapped。我们还有一个初始化器，它接收一个类型为 Wrapped 的参数。因此，初始化时所使用的类型就是 Wrapped，它也是关联到 .Some Case 的值的类型。

正是由于初始化器参数的类型与 .Some 关联值的类型之间的这种同一性才使得后者能够被解析出来。在 Optional 枚举的声明中，Wrapped 是个占位符。不过在实际情况下，当创建实际的 Optional 时，它会被某个确定的类型值所初始化。很多时候，我们会使用问号语法糖（String? 类型），初始化器则会在背后得到调用。出于清晰的目的，下面来显式调用初始化器：

```
let s = Optional("howdy")
```

上述代码会针对这个特定的 Optional 实例对 Wrapped 类型进行解析。显然，"howdy" 是个 String，因此编译器知道，对于这个特定的 Optional<Wrapped> 来说，Wrapped 是个 String。在底层 Optional 枚举声明中凡是出现 Wrapped 的地方，编译器都会将其替换为 String。因此，从编译器的角度来看，变量 s 所引用的这个特定 Optional 的声明如下所示：

```
enum Optional <String>{
    case None
    case Some(String)
    init(_ some: String)
    // ...
}
```

这是 Optional 声明的伪代码，其中 Wrapped 占位符已经被 String 类型所替换。我们可以说 s 是个 Optional<String>。事实上，这是合法的语法！我们可以像下面这样创建相同的 Optional：

```
let s : Optional<String> = "howdy"
```

大量内建的 Swift 类型都涉及泛型。事实上，该语言特性在设计时就充分考虑了 Swift 类型；正是由于泛型的存在，Swift 类型才能实现自己的目的。

## 4.9.1 泛型声明

下面列出了在什么地方可以声明 Swift 泛型：

*使用 Self 的泛型协议*

在协议中，关键字 `Self`（注意首字母大写）会将协议转换为泛型。`Self` 是个占位符，表示使用者的类型。比如，下面这个 Flier 协议声明了一个接收 `Self` 参数的方法：

```
protocol Flier {
    func flockTogetherWith(f:Self)
}
```

这表示，如果 Bird 对象类型使用了 Flier 协议，那么 `flockTogetherWith` 的实现就需要将其 `f` 参数声明为 Bird。

*使用空类型别名的泛型协议*

协议可以声明类型别名，不必定义类型别名表示什么。也就是说，`typealias` 语句并不会包含等号。这会将协议转换为泛型；别名的名字（也叫作关联类型）是个占位符。比如：

```
protocol Flier {
    typealias Other
    func flockTogetherWith(f:Other)
    func mateWith(f:Other)
}
```

使用者会在泛型使用类型别名的地方声明特定的类型，从而解析出占位符。如果 Bird 结构体使用了 Flier 协议，并将 `flockTogetherWith` 的 `f` 参数声明为 Bird，那么该声明就会针对这个特定的使用者将 Other 解析为 Bird，现在 Bird 也需要将 `mateWith` 的 `f` 参数声明为 Bird 类型：

```
struct Bird : Flier {
    func flockTogetherWith(f:Bird) {}
    func mateWith(f:Bird) {}
}
```

这种形式的泛型协议从根本上来说与前一种形式一样；如果写成 `f:Other`，那么 Swift 就会知道它表示 `f:Self.Other`，实际上这么写是合法的（也更加清晰）。

*泛型函数*

函数声明可以对其参数、返回类型以及在函数体中使用泛型占位符。请在函数名后的尖括号中声明占位符的名字：

```
func takeAndReturnSameThing<T> (t:T) -> T {
    return t
}
```

调用者会在函数声明中占位符出现的地方使用特定的类型，从而解析出占位符：

```
let thing = takeAndReturnSameThing("howdy")
```

调用中所用的实参 "howdy" 的类型会将 T 解析为 String；因此，对 takeAndReturnSameThing 的调用也会返回一个 String，捕获结果的变量 thing 也会被推断为 String。

*泛型对象类型*

对象类型声明可以在花括号中使用泛型占位符类型。请在对象类型名后面的尖括号中声明占位符名字：

```
struct HolderOfTwoSameThings<T> {
    var firstThing : T
    var secondThing : T
    init(thingOne:T, thingTwo:T) {
        self.firstThing = thingOne
        self.secondThing = thingTwo
    }
}
```

该对象类型的使用者会在对象类型声明中占位符出现的地方使用特定的类型，从而解析出占位符：

```
let holder = HolderOfTwoSameThings(thingOne:"howdy", thingTwo:"getLost")
```

初始化器调用中所使用的 thingOne 实参 "howdy" 的类型会将 T 解析为 String；因此，thingTwo 也一定是个 String，属性 firstThing 与 secondThing 都是 String。

对于使用了尖括号语法的泛型函数与对象类型，尖括号中可以包含多个占位符名，中间通过逗号分隔，比如：

```
func flockTwoTogether<T, U>(f1:T, _ f2:U) {}
```

现在，flockTwoTogether 的两个参数可以被解析为两个不同的类型（不过也可以相同）。

## 4.9.2 类型约束

到目前为止，所有示例都可以使用任何类型替代占位符。此外，你可以限制用于解析特

定占位符的类型，这叫作类型限制。最简单的类型限制形式是其首次出现时，在占位符名后面加上一个冒号和一个类型名。冒号后面的类型名可以是类名或是协议名。

回到 Flier 及其 flockTogetherWith 函数。假设 flockTogetherWith 的参数类型需要被使用者声明为使用了 Flier 的类型。你不能在协议中将参数类型声明为 Flier：

```
protocol Flier {
    func flockTogetherWith(f:Flier)
}
```

上述代码表示：只有声明的函数 flockTogetherWith 的 f 参数是 Flier 类型，你才能使用该协议：

```
struct Bird : Flier {
    func flockTogetherWith(f:Flier) {}
}
```

这并不是我们想要的！我们需要的是，Bird 应该可以使用 Flier 协议，同时将 f 声明为某个 Flier 使用者类型，如 Bird。方式就是将占位符限制为 Flier。比如，我们可以这样做：

```
protocol Flier {
    typealias Other : Flier
    func flockTogetherWith(f:Other)
}
```

遗憾的是，这么做是不合法的：协议不能将自身作为类型约束。解决办法就是再声明一个协议，然后让 Flier 使用这个协议，并且将 Other 约束到这个协议上：

```
protocol Superflier {}
protocol Flier : Superflier {
    typealias Other : Superflier
    func flockTogetherWith(f:Other)
}
```

现在，Bird 就是个合法的使用者了：

```
struct Bird : Flier {
    func flockTogetherWith(f:Bird) {}
}
```

在泛型函数或泛型对象类型中，类型限制位于尖括号中。比如：

```
func flockTwoTogether<T:Flier>(f1:T, _ f2:T) {}
```

现在不能使用两个 String 参数调用 flockTwoTogether 了，因为 String 并不是 Flier。此外，如果 Bird 与 Insect 都使用了 Flier，那么 flockTwoTogether 可以通过两个 Bird 参数或两

个 Insect 参数调用，但不能一个是 Bird，另一个是 Insect，因为 T 仅仅是一个占位符而已，表示 Flier 使用者类型。

对于占位符的类型限制通常用于告诉编译器，某个消息可以发送给占位符类型的实例。比如，假设我们要实现一个函数 myMin，它会从相同类型的一个列表中返回最小值。下面是一个看起来还不错的泛型函数实现，不过有个问题，即它无法编译通过：

```
func myMin<T>(things:T ...) -> T {
    var minimum = things[0]
    for ix in 1..<things.count {
        if things[ix] < minimum { // compile error
            minimum = things[ix]
        }
    }
    return minimum
}
```

问题在于比较 things[ix] < minimum。编译器怎么知道类型 T（things[ix] 与 minimum 的类型）所解析出的类型能够使用小于运算符进行比较呢？它不知道，这也是上述代码无法编译通过的原因所在。解决方案就是向编译器承诺，T 解析出的类型能够使用小于运算符。方式就是将 T 限制为 Swift 内建的 Comparable 协议；使用 Comparable 协议可以确保使用者能够使用小于运算符：

```
func myMin<T:Comparable>(things:T ...) -> T {
```

现在的 myMin 可以编译通过，因为只有将 T 解析为使用了 Comparable 的对象类型它才能被调用，因此它也可以使用小于运算符进行比较。自然地，你觉得可以进行比较的内建对象类型（如 Int、Double、String 及 Character 等）实际上都使用了 Comparable 协议！查阅 Swift 头文件，你会发现内建的 min 全局函数就是按照这种方式声明的，原因与此相同。

泛型协议（声明中使用了 Self 或拥有关联类型的协议）只能用在泛型类型中，并且作为类型限制。如下代码无法编译通过：

```
protocol Flier {
    typealias Other
    func fly()
}
func flockTwoTogether(f1:Flier, _ f2:Flier) { // compile error
    f1.fly()
    f2.fly()
}
```

要想将泛型 Flier 协议作为类型，你需要编写一个泛型并将 Flier 作为类型限制，如下代码所示：

```
protocol Flier {
    typealias Other
    func fly()
}
func flockTwoTogether<T1:Flier, T2:Flier>(f1:T1, f2:T2) {
    f1.fly()
    f2.fly()
}
```

## 4.9.3 显式特化

到目前为止，所有示例中泛型的使用者都是通过推断来解析占位符类型的。不过，还有一种解析方式：使用者可以手工解析类型，这叫作显式特化。在某些情况下，显式特化是强制的，即如果占位符类型无法通过推断得出，那就需要使用显式特化。有两种形式的显式特化：

*拥有关联类型的泛型协议*

协议使用者可以通过 `typealias` 声明手工解析出协议的关联类型，方式是使用协议别名与显式类型赋值。比如：

```
protocol Flier {
    typealias Other
}
struct Bird : Flier {
    typealias Other = String
}
```

*泛型对象类型*

泛型对象类型的使用者可以通过相同的尖括号语法手工解析出对象的占位符类型，尖括号用于声明泛型，里面的是实际的类型名。比如：

```
class Dog<T> {
    var name : T?
}
let d = Dog<String>()
```

（这解释了本章之前与第 3 章介绍的 `Optional<String>` 类型。）

不能显式特化泛型函数。不过，你可以使用非泛型函数（使用了泛型类型的占位符）来声明泛型类型；泛型类型的显式特化会解析出占位符，因此也能解析出函数：

```
protocol Flier {
    init()
}
struct Bird : Flier {
    init() {}
}
struct FlierMaker<T:Flier> {
```

```
    static func makeFlier() -> T {
        return T()
    }
}
let f = FlierMaker<Bird>.makeFlier() // returns a Bird
```

如果类是泛型的，那么你可以对其子类化，前提是可以解析出泛型（这是 Swift 2.0 的新特性）。可以通过匹配的泛型子类或显式解析出父类泛型来做到这一点。比如，下面是个泛型 Dog：

```
class Dog<T> {
   var name : T?
}
```

你可以将其子类化为泛型，其占位符与父类占位符相匹配：

```
class NoisyDog<T> : Dog<T> {}
```

这么做是合法的，因为对 NoisyDog 占位符 T 的解析也会解析 Dog 占位符 T。另一种方式是子类化一个明确指定的 Dog：

```
class NoisyDog : Dog<String> {}
```

## 4.9.4 关联类型链

如果具有关联类型的泛型协议使用了泛型占位符，那么我们可以通过对占位符名使用点符号将关联类型名链接起来，从而指定其类型。

来看下面这个示例。假设有一个游戏程序，士兵与弓箭手彼此为敌。我通过将 Soldier 结构体与 Archer 结构体纳入拥有 Enemy 关联类型的 Fighter 协议中来表示这一点，Enemy 本身又被限制为是一个 Fighter（这里还是需要另外一个协议，Fighter 会使用该协议）：

```
protocol Superfighter {}
protocol Fighter : Superfighter {
    typealias Enemy : Superfighter
}
```

下面手工为这两个结构体解析这个关联类型：

```
struct Soldier : Fighter {
    typealias Enemy = Archer
}
struct Archer : Fighter {
    typealias Enemy = Soldier
}
```

现在来创建一个泛型结构体，表示这些战士对面的营地：

```
struct Camp<T:Fighter> {
}
```

假设一个营地可以容纳来自对方阵营的一个间谍。那么间谍的类型应该是什么呢？如果是 Soldier 营地，那么它就是 Archer；如果是 Archer 营地，那么它就是 Soldier。更为一般地，由于 T 是个 Fighter，那么它应该是 Fighter 的 Enemy 类型。我可以通过将关联类型名链接到占位符名来清楚地表达这一点：

```
struct Camp<T:Fighter> {
    var spy : T.Enemy?
}
```

结果就是，针对某个特定的 Camp，如果 T 被解析为 Soldier，那么 `T.Enemy` 就表示 Fighter，反之亦然。我们为 Capm 的 spy 类型创建了正确的规则。如下代码无法编译通过：

```
var c = Camp<Soldier>()
c.spy = Soldier() // compile error
```

我们尝试将错误类型的对象赋给这个 Camp 的 spy 属性。但如下代码可以编译通过：

```
var c = Camp<Soldier>()
c.spy = Archer()
```

使用更长的关联类型名链也是可以的，特别是当泛型协议有一个关联类型，这个关联类型本身又被强制约束为一个拥有关联类型的泛型协议时更是如此。

比如，下面为每一类 Fighter 赋予一个有特色的武器：士兵有剑，弓箭手有弓。创建一个 Sword 结构体和一个 Bow 结构体，并将它们置于 Wieldable 协议之下：

```
protocol Wieldable {
}
struct Sword : Wieldable {
}
struct Bow : Wieldable {
}
```

向 Fighter 添加一个 Weapon 关联类型，Weapon 被强制约束为 Wieldable，这次还是手工解析每一种 Fighter 类型的 Weapon：

```
protocol Superfighter {
    typealias Weapon : Wieldable
}
protocol Fighter : Superfighter {
    typealias Enemy : Superfighter
}
struct Soldier : Fighter {
    typealias Weapon = Sword
    typealias Enemy = Archer
```

```
}
struct Archer : Fighter {
    typealias Weapon = Bow
    typealias Enemy = Soldier
}
```

假设每个Fighter都可以窃取敌人的武器，我为Fighter泛型协议添加一个steal(weapon:from:)方法。Fighter泛型协议该如何表示参数类型才能让其使用者通过恰当的类型来声明这个方法呢？

from:参数类型是该Fighter的Enemy。我们已经知道该如何表示它了：它是由占位符、点符号以及关联类型名构成的。这里的占位符就是该协议的使用者，即Self。因此，from:参数类型就是Self.Enemy。那么weapon:参数类型又是什么呢？它是Enemy的Weapon！因此，weapon:参数类型就是Self.Enemy.Weapon：

```
protocol Fighter : Superfighter {
    typealias Enemy : Superfighter
    func steal(weapon:Self.Enemy.Weapon, from:Self.Enemy)
}
```

（上述代码可以编译通过，省略Self表达的也是相同的含义。不过，Self依然是整个链条的隐式起始点，我觉得加上Self会让代码的含义变得更加清晰。）

如下Soldier与Archer的声明正确地使用了Fighter协议，代码会编译通过：

```
struct Soldier : Fighter {
    typealias Weapon = Sword
    typealias Enemy = Archer
    func steal(weapon:Bow, from:Archer) {
    }
}
struct Archer : Fighter {
    typealias Weapon = Bow
    typealias Enemy = Soldier
    func steal (weapon:Sword, from:Soldier) {
    }
}
```

这个示例是假想出来的（但我希望能说明问题），不过其表示的概念却不是。Swift头文件大量使用了关联类型链，关联类型链Generator.Element使用得非常多，因为它表示了序列元素的类型。SequenceType泛型协议有一个关联类型Generator，它被约束为泛型GeneratorType协议的使用者，反过来它会有一个关联类型Element。

## 4.9.5 附加约束

简单的类型约束会对类型进行限制，使其能够将占位符解析为单个类型。有时，你需要对可解析的类型做进一步的限制：这就需要附加约束了。

在泛型协议中，类型别名约束中的冒号与类型声明中的冒号是一个意思。这样，其后面可以跟着多个协议，或是后跟一个父类再加上多个协议：

```
class Dog {
}
class FlyingDog : Dog, Flier {
}
protocol Flier {
}
protocol Walker {
}
protocol Generic {
    typealias T : Flier, Walker
    typealias U : Dog, Flier
}
```

在 Generic 协议中，关联类型 T 只能被解析为使用了 Flier 协议与 Walker 协议的类型，关联类型 U 只能被解析为 Dog（或 Dog 子类）并使用了 Flier 协议的类型。

在泛型函数或对象类型的尖括号中，这种语法是非法的；相反，你可以附加一个 `where` 字句，其中包含一个或多个逗号分隔的对所声明的占位符的附加约束：

```
func flyAndWalk<T where T:Flier, T:Walker> (f:T) {}
func flyAndWalk2<T where T:Flier, T:Dog> (f:T) {}
```

`Where` 子句还可以对已经包含了占位符的泛型协议的关联类型进行附加限制，方式是使用关联类型链（参见 4.9.4 节的介绍）。如下伪代码表明了我的意图：我省略了 `where` 子句的内容，将注意力放在 `where` 子句所限制的内容上：

```
protocol Flier {
    typealias Other
}
func flockTogether<T:Flier where T.Other /*???*/ > (f:T) {}
```

如你所见，占位符 T 已经被限制为了一个 Flier。Flier 本身是个泛型协议，并且有一个关联类型 Other。这样，无论什么类型解析 T，它都会解析 Other。`Where` 子句进一步限制了到底什么类型可以解析 T，这是通过限制可解析 Other 的类型来做到的。

我们可以对关联类型链施加什么限制呢？一种可能是与上述示例相同的限制，一个冒号，后跟它需要使用的协议；或是通过它必须继承的类来做到这一点。如下示例使用了协议：

```
protocol Flier {
    typealias Other
}
struct Bird : Flier {
    typealias Other = String
}
struct Insect : Flier {
```

```
    typealias Other = Bird
}
func flockTogether<T:Flier where T.Other:Equatable> (f:T) {}
```

Bird 与 Insect 都使用了 Flier，不过这并不是说它们都可以作为 flockTogether 函数调用的参数。flockTogether 函数可以通过 Bird 实参调用，因为 Bird 的 Other 关联类型会被解析为 String，而 String 使用了内建的 Equatable 协议。不过，flockTogether 却不能通过 Insect 实参调用，因为 Insect 的 Other 关联类型会被解析为 Bird，而 Bird 并没有使用 Equatable 协议：

```
flockTogether(Bird()) // okay
flockTogether(Insect()) // compile error
```

如下示例使用了类：

```
protocol Flier {
    typealias Other
}
class Dog {
}
class NoisyDog : Dog {
}
struct Pig : Flier {
    typealias Other = NoisyDog // or Dog
}
func flockTogether<T:Flier where T.Other:Dog> (f:T) {}
```

flockTogether 函数可以通过 Pig 实参调用，因为 Pig 使用了 Flier，并且会将 Other 解析为 Dog 或 Dog 的子类：

```
flockTogether(Pig()) // okay
```

除了冒号，我们还可以使用等号 == 并且后跟一个类型。关联类型链最后的类型必须是这个精确的类型，而不能仅仅是协议使用者或子类。比如：

```
protocol Flier {
    typealias Other
}
protocol Walker {
}
struct Kiwi : Walker {
}
struct Bird : Flier {
    typealias Other = Kiwi
}
struct Insect : Flier {
    typealias Other = Walker
}
func flockTogether<T:Flier where T.Other == Walker> (f:T) {}
```

flockTogether 函数可以通过 Insect 实参调用，因为 Insect 使用了 Flier 并且会将 Other 解析为 Walker。不过，它不能通过 Bird 实参调用。Bird 使用了 Flier，并且会将 Other 解析为 Walker 的使用者，即 Kiwi；不过，这并不满足 == 限制。

在上一个示例中使用 == Dog 也会得到同样的结果。如果 Pig 将 Other 解析为 NoisyDog，那么 Pig 实参就不再是可接受的了；Pig 必须要将 Other 解析为 Dog 本身，这样才能成为可接受的实参。

== 运算符右侧的类型本身可以是个关联类型链。两个链中末尾被解析出的类型必须要相同。比如：

```
protocol Flier {
    typealias Other
}
struct Bird : Flier {
    typealias Other = String
}
struct Insect : Flier {
    typealias Other = Int
}
func flockTwoTogether<T:Flier, U:Flier where T.Other == U.Other>
(f1:T, _ f2:U) {}
```

flockTwoTogether 函数可以通过 Bird 与 Bird 调用，也可以通过 Insect 与 Insect 调用；不过，不能一个是 Insect，另一个是 Bird，因为它们不会将 Other 关联类型解析为相同的类型。

Swift 头文件大量使用了带有 == 运算符的 where 子句，特别是用它来限制序列类型。比如，String 的 appendContentsOf 方法声明了两次，如下代码所示：

```
mutating func appendContentsOf(other: String)
mutating func appendContentsOf<S : SequenceType
where S.Generator.Element == Character>(newElements: S)
```

第 3 章介绍过 appendContentsOf 可以将一个 String 连接到另一个 String 上。不过 appendContentsOf 并不仅仅可以将 String 连接到 String 上！字符序列也可以：

```
var s = "hello"
s.appendContentsOf(" world".characters) // "hello world"
```

Character 数组也可以：

```
s.appendContentsOf(["!" as Character])
```

它们都是字符序列，第 2 个 appendContentsOf 方法声明中的泛型指定了这一点。它是个序列，因为其类型使用了 SequenceType 协议。不过，并不是任何序列都可以；

其 Generator.Element 关联类型链必须要解析为 Character。如前所述，Generator.Element 链是 Swift 用于表示序列元素类型概念的一种方式。

Array 结构体也有一个 appendContentsOf 方法，不过其声明有些不同：

```
mutating func appendContentsOf<S : SequenceType
    where S.Generator.Element == Element>(newElements: S)
```

序列只能是一种类型。如果序列包含了 String 元素，那么你可以向其添加更多的元素，但只能是 String 元素；你不能向 String 元素序列添加 Int 元素序列。数组是序列；它是个泛型，其 Element 占位符是其元素的类型。因此，Array 结构体在其 appendContentsOf 方法声明中通过 == 运算符来强制使用这个规则：实参序列的元素类型必须要与现有数组的元素类型相同。

# 4.10 扩展

扩展是将自己的代码注入其他地方声明的对象类型中的一种方式；你所扩展的是一个已有的对象类型。你可以扩展自定义的对象类型，也可以扩展 Swift 或 Cocoa 的对象类型，在这种情况下，你实际上是将功能添加到了不属于你自己的类型当中！

扩展声明只能位于文件的顶部。要想声明扩展，请使用关键字 extension，后跟已有的对象类型名，然后可以添加冒号，后跟该类型需要使用的协议列表名（这一步是可选的），最后是花括号，里面是通常的对象类型声明的内容，其限制如下所示：

- 扩展不能重写已有的成员（不过它可以重载已有的方法）。
- 扩展不能声明存储属性（不过可以声明计算属性）。
- 类的扩展不能声明指定初始化器和析构器（不过可以声明便捷初始化器）。

## 4.10.1 扩展对象类型

根据以往的经验，我有时会扩展内建的 Swift 或 Cocoa 类型，从而以属性或方法的形式封装一些缺失的功能。如下示例来自于真实的应用。

在纸牌游戏中，我需要洗牌，而纸牌会存储在数组中。我会扩展内建的 Array 类型，并添加一个 shuffle 方法：

```
extension Array {
    mutating func shuffle () {
        for i in (0..<self.count).reverse() {
            let ix1 = i
            let ix2 = Int(arc4random_uniform(UInt32(i+1)))
```

```
                (self[ix1], self[ix2]) = (self[ix2], self[ix1])
            }
        }
    }
```

Cocoa 的 Core Graphics 框架有很多有用的函数都与 CGRect 结构体有关，Swift 已经扩展了 CGRect，并添加了一些辅助性的属性与方法；不过它并未提供获取 CGRect 中心点（CGPoint）的便捷方法，这在实际开发中是经常需要的，于是我扩展了 CGRect，为其添加一个 center 属性：

```
extension CGRect {
    var center : CGPoint {
        return CGPointMake(self.midX, self.midY)
    }
}
```

扩展可以声明静态或类方法；由于对象类型通常都是全局可见的，所以这是给全局函数指定恰当命名空间的绝佳方式。比如，在我开发的一个应用中，我经常会使用某个颜色（UIColor）。相对于重复创建这个颜色，更好的方式是将生成它的代码封装到全局函数中。不过，相对于让这个函数成为全局的，我使之成为 UIColor 的一个类方法，这么做是非常恰当的：

```
extension UIColor {
    class func myGoldenColor() -> UIColor {
        return self.init(red:1.000, green:0.894, blue:0.541, alpha:0.900)
    }
}
```

现在，我只需通过 UIColor.myGolden() 就可以在代码中使用该颜色了，这与内建的类方法如 UIColor.redColor() 是非常相似的。

扩展的另一个用途是让内建的 Cocoa 类能够处理你的私有数据类型。比如，在我开发的 Zotz 应用中，我定义了一个枚举，其原始值是归档或反归档 Card 属性时所用的键字符串：

```
enum Archive : String {
    case Color = "itsColor"
    case Number = "itsNumber"
    case Shape = "itsShape"
    case Fill = "itsFill"
}
```

这里唯一的问题在于为了在归档时能够使用该枚举，我每次都需要带上其 rawValue：

```
coder.encodeObject(s1, forKey:Archive.Color.rawValue)
coder.encodeObject(s2, forKey:Archive.Number.rawValue)
coder.encodeObject(s3, forKey:Archive.Shape.rawValue)
coder.encodeObject(s4, forKey:Archive.Fill.rawValue)
```

这么做太丑陋了。优雅的解决办法（WWDC 2015 视频中所推荐的做法）是告诉 coder 所属的类 NSCoder 当 forKey: 参数是归档而非 String 时应该怎么做。在扩展中，我重载了 encodeObject:forKey: 方法：

```
extension NSCoder {
    func encodeObject(objv: AnyObject?, forKey key: Archive) {
        self.encodeObject(objv, forKey:key.rawValue)
    }
}
```

实际上，我将对 rawValue 的调用从代码中移出并放到了 NSCoder 的代码中。现在，归档 Card 时就可以不调用 rawValue 了：

```
coder.encodeObject(s1, forKey:Archive.Color)
coder.encodeObject(s2, forKey:Archive.Number)
coder.encodeObject(s3, forKey:Archive.Shape)
coder.encodeObject(s4, forKey:Archive.Fill)
```

对自定义对象类型的扩展有助于代码组织。经常应用的一个约定是为对象类型需要使用的每个协议添加扩展，比如：

```
class ViewController: UIViewController {
    // ... UIViewController method overrides go here ...
}
extension ViewController : UIPopoverPresentationControllerDelegate {
    // ... UIPopoverPresentationControllerDelegate methods go here ...
}
extension ViewController : UIToolbarDelegate {
    // ... UIToolbarDelegate methods go here ...
}
```

如果你认为多个小文件要比一个大文件好，那么对自定义对象类型的扩展也是将该对象类型分散到多个文件中的一种方式。

在扩展 Swift 结构体时，初始化器会有一件奇怪的事情出现：我们可以声明一个初始化器，同时又保留隐式初始化器：

```
struct Digit {
    var number : Int
}
extension Digit {
    init() {
        self.init(number:42)
    }
}
```

上述代码表示，你可以通过调用显式声明的初始化器 Digit()，或是调用隐式初始化器 Digit(number:7) 来实例化一个 Digit。因此，通过扩展显式声明的初始化器并不会导致

隐式初始化器的丢失，如果在原来的结构体声明中就声明了相同的初始化器，那么就会出现这种情况。

## 4.10.2 扩展协议

在 Swift 2.0 中，你可以对协议进行扩展。在扩展协议时，你可以向其中添加方法与属性，就像扩展对象类型那样。与协议声明不同的是，这些方法与属性并不仅仅要被协议使用者所实现，它们还是要被协议使用者所继承的实际方法与属性！比如：

```
protocol Flier {
}
extension Flier {
    func fly() {
        print("flap flap flap")
    }
}
struct Bird : Flier {
}
```

现在，Bird 可以使用 Flier 而无须实现 `fly` 方法。即便我们将 `func fly()` 作为一种要求添加到了 Flier 协议的声明中，Bird 依然可以使用 Flier 而无须实现 `fly` 方法。这是因为 Flier 协议扩展支持 `fly` 方法！这样，Bird 就继承了 `fly` 的实现：

```
let b = Bird()
b.fly() // flap flap flap
```

使用者可以实现从协议扩展继承下来的方法，因此也可以重写这个方法：

```
struct Insect : Flier {
    func fly() {
        print("whirr")
    }
}
let i = Insect()
i.fly() // whirr
```

不过你要知道，这种继承并不是多态。使用者的实现并非重写；它只不过是另一个实现而已。内在一致性原则并不适用；重要的是引用类型到底是什么：

```
let f : Flier = Insect()
f.fly() // flap flap flap
```

虽然 `f` 本质上是个 Insect（这一点通过 `is` 运算符可以看到），但 `fly` 消息却发送给了类型为 Flier 的对象引用，因此调用的是 `fly` 方法的 Flier 实现而非 Insect 实现。

要想实现多态继承，我们需要在原始协议中将 `fly` 声明为必须要实现的方法：

```
protocol Flier {
    func fly() // *
}
extension Flier {
    func fly() {
        print("flap flap flap")
    }
}
struct Insect : Flier {
    func fly() {
        print("whirr")
    }
}
```

现在，Insect 会维护其内在一致性：

```
let f : Flier = Insect()
f.fly() // whirr
```

这种差别有其现实意义，因为协议使用者并不会引入（也不能引入）动态分派的开销。因此，编译器要做出静态的决定。如果方法在原始协议中声明为必须要实现的方法，那么我们就可以确保使用者会实现它，因此可以调用（也只能这么调用）使用者的实现。但如果方法只存在于协议扩展中，那么决定使用者是否重新实现了它就需要运行期的动态分派，这违背了协议的本质，因此编译器会将消息发送给协议扩展。

协议扩展的主要好处在于可以将代码移到合适的范围中。如下示例来自于我开发的 Zotz 应用。我有 4 个枚举，每个都表示 Card 的一个特性：Fill、Color、Shape 和 Number。它们都有一个 Int 原始值。我已经对每次通过其原始值初始化这些枚举时都要调用 `rawValue:` 感到厌烦，因此为每个枚举添加了一个没有外部参数名的委托初始化器，它会调用内建的 `init(rawValue:)` 初始化器：

```
enum Fill : Int {
    case Empty = 1
    case Solid
    case Hazy
    init?(_ what:Int) {
        self.init(rawValue:what)
    }
}
enum Color : Int {
    case Color1 = 1
    case Color2
    case Color3
    init?(_ what:Int) {
        self.init(rawValue:what)
    }
}
// ... and so on ...
```

我不喜欢重复初始化器声明，不过在 Swift 1.2 及之前的版本中只能这么做。在 Swift 2.0 中，我可以将其声明移到协议扩展中。带有原始值的枚举会自动使用内建的泛型 RawRepresentable 协议，其中的原始值类型是个名为 RawValue 的类型别名。因此，我可以将初始化器放到 RawRepresentable 协议中：

```
extension RawRepresentable {
    init?(_ what:RawValue) {
        self.init(rawValue:what)
    }
}
enum Fill : Int {
    case Empty = 1
    case Solid
    case Hazy
}
enum Color : Int {
    case Color1 = 1
    case Color2
    case Color3
}
// ... and so on ...
```

在 Swift 标准库中，协议扩展使得很多全局函数都可以转换为方法。比如，在 Swift 1.2 及之前的版本中，enumerate（参见第 3 章）是个全局函数：

```
func enumerate<Seq:SequenceType>(base:Seq) -> EnumerateSequence<Seq>
```

enumerate 是个全局函数，因为只能如此。该函数只能应用于序列，即 SequenceType 协议的使用者。在 Swift 2.0 之前该如何表示这一点呢？enumerate 方法被声明为 SequenceType 协议中必须要实现的方法，不过这仅仅意味着 SequenceType 的每个使用者都要实现它；协议本身无法提供实现。要想做到这一点，唯一的办法就是全局函数，将序列作为参数，使用泛型约束来做好把控，因此实参只能是序列。

不过在 Swift 2.0 中，enumerate 是个方法，声明在 SequenceType 协议的扩展中：

```
extension SequenceType {
    func enumerate() -> EnumerateSequence<Self>
}
```

现在没必要再使用泛型约束了。没必要使用泛型了。也没必要使用参数了！它已经成为 SequenceType 中的方法；要枚举的序列就是接收 enumerate 消息的那个序列。

以此类推，在 Swift 2.0 中，有大量 Swift 标准库全局函数都变成了方法。这种转变改变了语言的风格。

## 4.10.3 扩展泛型

在扩展泛型类型时，占位符类型名对于扩展声明来说是可见的。这很棒，因为你可能会用到它；不过，这会导致代码变得令人困惑，因为你看起来在使用未定义的类型。添加注释是个好主意，用来提醒自己要做的是什么：

```
class Dog<T> {
    var name : T?
}
extension Dog {
    func sayYourName() -> T? { // T is the type of self.name
        return self.name
    }
}
```

在 Swift 2.0 中，泛型类型扩展可以使用一个 where 子句。这与泛型约束的效果是一样的：它会限制泛型的哪个解析者可以调用该扩展所注入的代码，并向编译器保证代码对于这些解析者来说是合法的。

与协议扩展一样，这意味着全局函数可以转换为方法了。回忆一下本章之前的这个示例：

```
func myMin<T>(things:T...) -> T {
    var minimum = things[0]
    for ix in 1..<things.count {
        if things[ix] < minimum { // complile error
            minimum = things[ix]
        }
    }
    return minimum
}
```

我为何要将其作为全局函数呢？因为在 Swift 2.0 之前，我只能这么做。假设将其作为 Array 的一个方法。在 Swift 1.2 及之前的版本中，你可以扩展 Array，扩展会引用到 Array 的泛型占位符；不过，它无法对占位符做进一步的约束。这样，我们没办法在将方法注入 Array 的同时又确保占位符是个 Comparable，因此编译器不允许对数组的元素使用 < 运算符。在 Swift 2.0 中，我可以进一步约束泛型占位符，因此可以将其作为 Array 的一个方法：

```
extension Array where Element:Comparable { // Element is the element type
    func min() -> Element {
        var minimum = self[0]
        for ix in 1..<self.count {
            if self[ix] < minimum {
                minimum = self[ix]
            }
        }
        return minimum
```

```
        }
    }
```

该方法只能在 Comparable 元素的数组上调用；它不会注入其他类型的数组中，因此编译器不允许下面这样调用：

```
let m = [4,1,5,7,2].min() // 1
let d = [Digit(12), Digit(42)].min() // compile error
```

第 2 行代码无法编译通过，因为 Digit 结构体并未使用 Comparable 协议。

重申一次，Swift 语言的这种变化导致 Swift 标准库发生了大规模的重组，可以将全局函数移到结构体扩展与协议扩展中并作为方法。比如，Swift 1.2 及之前版本中的全局函数 `find` 在 Swift 2.0 中成为 CollectionType 的 `indexOf` 方法；它是受约束的，这样集合的元素就都是 Equatable 的，因为大海捞针是不可能的，除非你有办法能找到针：

```
extension CollectionType where Generator.Element : Equatable {
    func indexOf(element: Self.Generator.Element) -> Self.Index?
}
```

这是个协议扩展，也是个带有 `where` 子句约束的泛型扩展，这些特性都是 Swift 2.0 中新增的。

# 4.11 保护类型

Swift 提供了几个内建的保护类型，它们可以通过一个声明表示多种实际类型。

## 4.11.1 AnyObject

在实际的 iOS 编程中，最常使用的保护类型是 AnyObject。它实际上是个协议；作为协议，其内部完全是空白的，没有属性，也没有方法。它有个很特别的特性，那就是所有的类类型都会自动遵循它。因此，在需要 AnyObject 的地方，我们可以赋值或传递任何类实例，并且可以进行双向的类型转换：

```
class Dog {
}
let d = Dog()
let any : AnyObject = d
let d2 = any as! Dog
```

某些非类类型的 Swift 类型（如 String 和基本的数字类型）都可以桥接到 Objective-C 类型上，它们是由 Foundation 框架定义的类类型。这意味着，在 Foundation 框架下，Swift 桥接类型可以赋值、传递或转换为 AnyObject，即便它并非类类型也可以，这是因

为在背后，它首先会被自动转换为相应的 Objective-C 桥接类类型，AnyObject 也可以向下类型转换为 Swift 桥接类型。比如：

```
let s = "howdy"
let any : AnyObject = s // implicitly casts to NSString
let s2 = any as! String
let i = 1
let any2 : AnyObject = i // implicitly casts to NSNumber
let i2 = any2 as! Int
```

我们常常会在与 Objective-C 交换的过程中遇到 AnyObject。Swift 可以将任何类类型转换为 AnyObject 以及将 AnyObject 转换为类类型的能力类似于 Objective-C 可以将任何类类型转换为 `id` 以及将 `id` 转换为类类型的能力。实际上，AnyObject 就是 Swift 版本的 `id`。

比如，你可以通过 NSUserDefaults、NSCoding 与键值编码（参见第 10 章）根据字符串的键名来获取到不确定的类对象；这种对象会以 AnyObject 的形式进入 Swift 中；特别地，其形式是包装了 AnyObject 的一个 Optional，因为键可能不存在，这样 Cocoa 要能返回 `nil`。不过，一般来说，你很少会用到 AnyObject；你要让 Swift 知道对象到底是什么类型的。展开 Optional 并将其从 AnyObject 向下类型转换是你的职责。如果你知道其类型是什么，那就可以强制展开并通过 `as!` 运算符进行强制转换：

```
required init ( coder decoder: NSCoder ) {
    let s = decoder.decodeObjectForKey(Archive.Color) as! String
    // ...
}
```

当然，在使用 `as!` 对 AnyObject 进行向下类型转换时要将其转换为正确的类型，否则当代码运行时程序就会崩溃，转换也是不可能的事情了。如果不确定，那么可以使用 `is` 与 `as?` 运算符来确保转换是安全的。

### 1. 压制类型检查

AnyObject 一个令人惊叹的特性是它可以将编译器对某条消息是否可以发送给某个对象的判断推迟，这类似于 Objective-C，在 Objective-C 中，`id` 类型会导致编译器推迟对什么消息可以发送给它的判断。因此，你可以向 AnyObject 发送消息，而不必将其转换为真正的类型。（不过，如果知道对象的真实类型，那么你可能还是想将其转换到该类型上。）

你不能随意向 AnyObject 发送消息；消息必须要对应于满足如下条件之一的类成员：

- 它要是 Objective-C 类的成员。
- 它要是你自己定义的 Objective-C 类的 Swift 子类（或扩展）的成员。

- 它要是 Swift 类的成员，并且标记为 @objc（或 dynamic）。

本质上，该特性类似于本章之前介绍的可选协议成员，只不过有一些微小的差别。先从两个类开始：

```
class Dog {
    @objc var noise : String = "woof"
    @objc func bark() -> String {
        return "woof"
    }
}
class Cat {}
```

Dog 的属性 noise 与方法 bark 都被标记为了 @objc，这样它们就可以作为发送给 AnyObject 的潜在消息了。为了证明这一点，我来创建一个 AnyObject 类型的 Cat，并向其发送一条消息。首先从 noise 属性开始：

```
let c : AnyObject = Cat()
let s = c.noise
```

上述代码竟然可以编译通过。此外，代码运行时也不会崩溃！ noise 属性的类型是包装其原始类型的一个 Optional，即包装 String 的 Optional。如果类型为 AnyObject 的对象没有实现 noise，那么结果就是 nil，什么都不会发生。另外，与可选协议属性不同，本示例中的 Optional 是隐式展开的。因此，如果 AnyObject 实际上有 noise 属性（比如，它是个 Dog），那么生成的隐式展开 String 就可以直接当成 String 了。

下面再来看看方法调用：

```
let c : AnyObject = Cat()
let s = c.bark?()
```

上述代码依然可以编译通过。如果类型为 AnyObject 的对象没有实现 bark，那么 bark() 调用就不会执行；方法结果类型已经被包装到了 Optional 中，因此 s 的类型是个 String?，并且已经被设为了 nil。如果 AnyObject 具有 bark 方法（比如，如果它是个 Dog），那么结果就是一个包装了返回 String 的 Optional。如果在 AnyObject 上调用 bark!()，那么结果就是个 String，不过如果 AnyObject 没有实现 bark，那么应用将会崩溃。与可选协议成员不同，在发送消息时甚至都不用将其展开。如下代码是合法的：

```
let c : AnyObject = Cat()
let s = c.bark()
```

上述代码就好像是强制展开调用一样：结果是个 String，不过这么做可能导致应用崩溃。

### 2. 对象恒等性与类型恒等性

有时，你想要知道的并非某个对象是什么类型，而是某个对象本身是否是你所认为的那个特定的对象。值类型不会出现这个问题，但引用类型却会出现，因为可能会有多个不同的引用指向同一个对象。类是引用类型，因此类实例会遇到这个问题。

Swift 的解决方案就是恒等运算符（===）。该运算符可用于使用了 AnyObject 协议的对象类型实例，就像类一样！它会比较对象引用。这并不是比较两个值是否相等，就像相等运算符（==）那样；你所做的是判断两个对象引用是否指向了相同的对象。恒等运算符还有一个否定版本（!==）。

一个典型的使用场景是类实例来自于 Cocoa，你需要知道它是否是已经拥有的某个引用所指向的特定对象。比如，NSNotification 有一个 `object` 属性，它用于标识通知（通常情况下，它是通知最初的发送者）；Cocoa 对其底层类型一无所知，因此你会接收到一个包装了 AnyObject 的 Optional。就像 == 一样，=== 运算符可与 Optional 无缝衔接，因此你可以使用它来确保通知的 `object` 属性就是你所期望的对象：

```
func changed(n:NSNotification) {
    let player = MPMusicPlayerController.applicationMusicPlayer()
    if n.object === player {
        // ...
    }
}
```

## 4.11.2 AnyClass

AnyClass 是 AnyObject 对应的类，它对应于 Objective-C 的 `Class` 类型。通常在 Cocoa API 的声明中如果需要一个类，那这正是 AnyClass 的用武之地。

比如，UIView 的 `layerClass` 类方法对应的 Swift 声明如下代码所示：

```
class func layerClass() -> AnyClass
```

上述代码表示：如果重写了该方法，那么需要让其返回一个类。这可能是一个 CALayer 子类。要想在自己的实现中返回一个实际的类，请向类名发送 `self` 消息：

```
override class func layerClass() -> AnyClass {
    return CATiledLayer.self
}
```

对 AnyClass 对象的引用与对 AnyObject 对象的引用行为相似。你可以向其发送 Swift 知道的任何 Objective-C 消息，即任何 Objective-C 类消息。为了说明问题，我们从两个类开始：

```
class Dog {
    @objc static var whatADogSays : String = "woof"
}
class Cat {}
```

Objective-C 会看到 whatADogSays 并将其作为一个类属性。因此，你可以将 whatADogSays 发送给 AnyClass 引用：

```
let c : AnyClass = Cat.self
let s = c.whatADogSays
```

对类的引用（可以通过向实例引用发送 dynamicType 获得，也可以通过向类型名发送 self 获得）类型使用了 AnyClass，你可以通过 === 运算符比较这种类型的引用。实际上，这种方式可以判断指向类的两个引用是否指向了相同的类。比如：

```
func typeTester(d:Dog, _ whattype:Dog.Type) {
    if d.dynamicType === whattype {
        // ...
    }
}
```

只有当 d 与 whattype 是相同类型时条件才为 true（不考虑多态）；比如，如果 Dog 有个子类 NoisyDog，那么如果参数为 Dog() 与 Dog.self 或 NoisyDog 与 NoisyDog.self，条件就为 true；但如果参数为 NoisyDog() 与 Dog.self，那么条件就为 false。尽管没有使用多态，但这么做是有意义的，因为当右侧是类型引用时，你没法使用 is 运算符，必须要是字面类型名才行。

## 4.11.3 Any

Any 类型是被所有类型自动使用的一个空协议的类型别名。这样，在需要一个 Any 对象时，你可以传递任何对象：

```
func anyExpecter(a:Any) {}
anyExpecter("howdy")     // a struct instance
anyExpecter(String)      // a struct
anyExpecter(Dog())       // a class instance
anyExpecter(Dog)         // a class
anyExpecter(anyExpecter) // a function
```

类型为 Any 的对象可以与任何对象或函数类型进行比较，也可以向下类型转换为它们。为了说明这一点，下面是一个带有关联类型的协议，并且有两个使用者显式地进行解析：

```
protocol Flier {
    typealias Other
}
struct Bird : Flier {
    typealias Other = Insect
```

```
}
struct Insect : Flier {
    typealias Other = Bird
}
```

现在有一个函数接收一个 Flier 参数和一个类型为 Any 的参数，并测试第 2 个参数的类型是否与 Flier 解析出的 Other 类型一样；这个测试是合法的，因为 Any 可以与任何类型进行比较：

```
func flockTwoTogether<T:Flier>(flier:T, _ other:Any) {
    if other is T.Other {
        print("they can flock together")
    }
}
```

如果使用 Bird 与 Insect 调用 `flockTwoTogether`，那么控制台会打印出“they can flock together”。如果使用 Bird 与其他类型的对象调用 `flockTwoTogether`，那么控制台就不会打印出任何内容。

# 4.12 集合类型

与大多数现代计算机语言一样，Swift 也拥有内建的集合类型，Array 与 Dictionary，以及第 3 种类型 Set。Array 与 Dictionary 是非常重要的，语言也对其提供了特殊的语法支持。同时，就像大多数 Swift 类型一样，Swift 为其提供的相关函数也是有限的，一些缺失的功能是通过 Cocoa 的 NSArray 与 NSDictionary 补充的，Array 与 NSArray，Dictionary 与 NSDictionary 之间是彼此桥接的。Set 集合类型则与 Cocoa 的 NSSet 桥接。

## 4.12.1 Array

数组（Array，是个结构体）是对象实例的一个有序集合（即数组元素），可以通过索引号进行访问，索引号是个 Int，值从 0 开始。因此，如果一个数组包含了 4 个元素，那么第 1 个元素的索引就为 0，最后一个元素的索引为 3。Swift 数组不可能是稀疏数组：如果有一个元素的索引为 3，那么肯定还会有一个元素的索引为 2，以此类推。

Swift 数组最显著的特征就是其严格的类型。与其他一些计算机语言不同，Swift 数组的元素必须是统一的，也就是说，数组必须包含相同类型的元素。甚至连空数组都必须要有确定的元素类型，尽管此时数组中并没有元素存在。数组本身的类型与其元素的类型是一致的。所包含的元素类型不同的数组被认为是两个不同类型的数组：Int 元素的数组与 String 元素的数组就是不同类型的数组。数组类型与其元素类型一样也是多态的：如果 NoisyDog 是 Dog 的子类，那么 NoisyDog 的数组就可以用在需要 Dog 数组的地方。

如果这些让你想起了 Optional，那就对了。就像 Optional 一样，Swift 数组也是泛型。它被声明为 `Array<Element>`，其中占位符 Element 是特定数组元素的类型。

一致性约束并没有初看起来那么苛刻。数组只能包含一种类型的元素，不过类型却是非常灵活的。通过精心选取类型，你所创建的数组中，内部元素的类型可以是不同的。比如：

- 如果 Dog 类有个 NoisyDog 子类，那么 Dog 数组既可以包含 Dog 对象，也可以包含 NoisyDog 对象。
- 如果 Bird 与 Insect 都使用了 Flier 协议，那么 Flier 数组既可以包含 Bird 对象，也可以包含 Insect 对象。
- AnyObject 数组可以包含任何类以及任何 Swift 桥接类型的实例，如 Int、String 和 Dog 等。
- 类型本身也可以是不同的可能类型的载体。本章之前介绍的 Error 枚举就是一个例子；其关联值可能是 Int 或是 String，这样 Error 元素的数组就可以包含 Int 值与 String 值。

要想声明或表示给定数组元素的状态，你应该显式解析出泛型占位符；Int 元素的数组就是 `Array<Int>`。不过，Swift 提供了语法糖来表示数组的元素类型，通过将方括号包围元素类型名来表示，如 `[Int]`。你在绝大多数时候都会使用这种语法。

字面值数组表示为一个方括号，里面包含着用逗号分隔的元素列表（以及可选的空白字符）：如 `[1,2,3]`。空数组的字面值就是空的方括号 `[]`。

数组的默认初始化器 `init()` 是通过在数组类型后面使用一对空的圆括号来调用的，它会生成该类型的一个空数组。因此，你可以像下面这样创建一个空的 Int 数组：

```
var arr = [Int]()
```

另外，如果提前已经知道了引用类型，那么空数组 [] 就可以推断为该类型。因此，你还可以像下面这样创建一个 Int 类型的空数组：

```
var arr : [Int] = []
```

如果从包含元素的字面值数组开始，那么通常无须声明数组的类型，因为 Swift 会根据元素推断出其类型。比如，Swift 会将 `[1,2,3]` 推断为 Int 数组。如果数组元素类型包含了类及其子类，如 Dog 与 NoisyDog，那么 Swift 会将父类推断为数组的类型。甚至 `[1, "howdy"]` 也是合法的数组字面值；它会被推断为 NSObject 数组。不过，在某些情况下，你还是需要显式声明数组引用的类型，同时将字面值赋给该数组：

```
let arr : [Flier] = [Insect(), Bird()]
```

数组也有一个初始化器，其参数是个序列。这意味着，如果类型是个序列，那么你可以将其实例分割到数组元素中。比如：

- `Array(1...3)` 会生成数组 `Int [1,2,3]`。
- `Array("hey".characters)` 会生成数组 `Character ["h","e","y"]`。
- `Array(d)`（其中 `d` 是个 Dictionary）会生成 `d` 键值对的元组数组。

另一个数组初始化器 `init(count:repeatedValue:)` 可以使用相同的值来装配数组。在该示例中，我创建了一个由 100 个 Optional 字符串构成的数组，每个 Optional 都被初始化为 `nil`：

```
let strings : [String?] = Array(count:100, repeatedValue:nil)
```

这是 Swift 中最接近稀疏数组的数组创建方式；我们有 100 个槽，每个槽都可能包含或不包含一个字符串（一开始都不包含）。

### 1. 数组转换与类型检测

在将一个数组类型赋值、传递或转换为另一个数组类型时，你操作的实际上是数组中的每个元素。比如：

```
let arr : [Int?] = [1,2,3]
```

上述代码实际上是个简写：将 Int 数组看作包装了 Int 的 Optional 数组意味着原始数组中的每个 Int 都必须要包装到 Optional 中。如下代码所示：

```
let arr : [Int?] = [1,2,3]
print(arr) // [Optional(1), Optional(2), Optional(3)]
```

与之类似，假设有一个 Dog 类及其 NoisyDog 子类；那么如下代码就是合法的：

```
let dog1 : Dog = NoisyDog()
let dog2 : Dog = NoisyDog()
let arr = [dog1, dog2]
let arr2 = arr as! [NoisyDog]
```

在第 3 行，我们有一个 Dog 数组。在第 4 行，我们将该数组向下类型转换为 NoisyDog 数组，这意味着我们将第 1 个数组中的每个 Dog 都转换为了 NoisyDog（这么做应用并不会崩溃，因为第 1 个数组中的每个元素实际上都是个 NoisyDog）。

你可以将数组的所有元素与 `is` 运算符进行比较来判断数组本身。比如，考虑之前代码中的 Dog 数组，可以这么做：

```
if arr is [NoisyDog] { // ...
```

如果数组中的每个元素都是 NoisyDog，那么结果就为 true。

与之类似，as? 运算符会将数组转换为包装数组的 Optional，如果底层的转换无法进行，那么结果就为 nil：

```
let dog1 : Dog = NoisyDog()
let dog2 : Dog = NoisyDog()
let dog3 : Dog = Dog()
let arr = [dog1, dog2]
let arr2 = arr as? [NoisyDog] // Optional wrapping an array of NoisyDog
let arr3 = [dog2, dog3]
let arr4 = arr3 as? [NoisyDog] // nil
```

对数组进行向下类型转换与对任何值进行向下类型转换的原因相同，这样就可以向数组的元素发送恰当的消息了。如果 NoisyDog 声明了一个 Dog 没有的方法，那么你就不能向 Dog 数组中的元素发送该消息。有时需要将元素向下类型转换为 NoisyDog，这样编译器就允许你发送该消息了。你可以向下类型转换单个元素，也可以转换整个数组；你要做的就是选择一种安全并且在特定上下文中有意义的方式。

## 2. 数组比较

数组相等与你想的是一样的：如果两个数组包含相同数量的元素，并且相同位置上的元素全都相等，那么这两个数组就是相等的：

```
let i1 = 1
let i2 = 2
let i3 = 3
if [1,2,3] == [i1,i2,i3] { // they are equal!
```

如果比较两个数组，那么这两个数组不必非得是相同类型的，不过除非它们包含的对象都彼此相等，否则这两个数组就不会相等。如下代码比较了一个 Dog 数组和一个 NoisyDog 数组；它们实际上是相等的，因为它们实际上是以相同顺序包含了相同的狗：

```
let nd1 = NoisyDog()
let d1 = nd1 as Dog
let nd2 = NoisyDog()
let d2 = nd2 as Dog
if [d1,d2] == [nd1,nd2] { // they are equal!
```

## 3. 数组是值类型

由于数组是个结构体，因此它是个值类型而非引用类型。这意味着每次将数组赋给变量或作为参数传递给函数时，数组都会被复制。不过，我并不是说仅仅赋值或传递数组就是代价高昂的操作，这些复制每次都会发生。如果对数组的引用是个常量，那显然就没

必要执行复制了；甚至从另一个数组生成新数组，或是修改数组的操作都是非常高效的。你要相信 Swift 的设计者肯定已经考虑过这些问题了，并且在背后会高效地实现数组。

虽然数组本身是个值类型，但其元素却会按照元素本身的情况来对待。特别地，如果一个数组是类实例数组，将其赋给多个变量，那么结果就是会有多个引用指向相同的实例。

### 4. 数组下标

Array 结构体实现了 subscript 方法，可以通过在对数组的引用后使用方括号来访问元素。你可以在方括号中使用 Int。比如，在一个包含着 3 个元素的数组中，如果数组是通过变量 `arr` 引用的，那么 `arr[1]` 就可以访问第 2 个元素。

还可以在方括号中使用 Int 的 Range。比如，如果 `arr` 是个包含 3 个元素的数组，那么 `arr[1...2]` 就表示第 2 与第 3 个元素。从技术上来说，如 `arr[1...2]` 之类的表达式会生成一个 ArraySlice。不过，ArraySlice 非常类似于数组；比如，你可以像对数组一样对 ArraySlice 使用下标，在需要数组的地方也可以传递 ArraySlice 过去。一般来说，你可以认为 ArraySlice 就是个数组。

如果对数组的引用是可变的（`var` 而非 `let`），那么就可以对下标表达式赋值。这么做会改变槽中的值。当然，赋的值一定要与数组元素的类型保持一致：

```
var arr = [1,2,3]
arr[1] = 4 // arr is now [1,4,3]
```

如果下标是个范围，那么赋的值就必须是个数组。这会改变被赋值的数组的长度：

```
var arr = [1,2,3]
arr[1..<2] = [7,8] // arr is now [1,7,8,3]
arr[1..<2] = [] // arr is now [1,8,3]
arr[1..<1] = [10] // arr is now [1,10,8,3] (no element was removed!)
```

如果访问数组元素时使用的下标大于最大值或小于最小值，那就会产生运行时错误。如果 `arr` 有 3 个元素，那么 `arr[-1]` 与 `arr[3]` 从语义上来说并没有错，但程序将会崩溃。

### 5. 嵌套数组

数组元素也可以是数组，比如：

```
let arr = [[1,2,3], [4,5,6], [7,8,9]]
```

这是个 Int 数组的数组。因此，其类型声明是 `[[Int]]`。（被包含的数组不一定非得是相同长度的，我这么做只是为了清晰。）

要想访问嵌套数组中的每个 Int，你可以将下标运算符链接起来：

```
let arr = [[1,2,3], [4,5,6], [7,8,9]]
let i = arr[1][1] // 5
```

如果外层数组引用是可变的，那么你还可以对嵌套数组赋值：

```
var arr = [[1,2,3], [4,5,6], [7,8,9]]
arr[1][1] = 100
```

还可以通过其他方式修改内部数组；比如，可以插入新的元素。

## 6. 基本的数组属性与方法

数组是一个集合（CollectionType 协议），集合本身又是个序列（SequenceType 协议）。你可能对此有所了解：String 的 characters 就是这样的，我在第 3 章将其称作字符序列。出于这个原因，数组与字符序列是非常相似的。

作为集合，数组的 count 是个只读属性，返回数组中的元素个数。如果数组的 count 为 0，那么其 isEmpty 属性就为 true。

数组的 first 与 last 只读属性会返回其第一个和最后一个元素，不过这些元素会被包装到 Optional 中，因为数组可能为空，因此这些属性就会为 nil。这会出现 Swift 中很少会遇到的将一个 Optional 包装到另一个 Optional 中的情况。比如，考虑包装 Int 的 Optional 数组，如果获取该数组最后一个属性会发生什么。

数组最大的可访问索引要比其 count 小 1。你常常会使用对 count 的引用来计算索引值；比如，要想引用 arr 的最后两个元素，可以这样做：

```
let arr = [1,2,3]
let arr2 = arr[arr.count-2...arr.count-1] // [2,3]
```

Swift 并未使用现在普遍采用的通过负数来计算索引这一约定。另一方面，如果想要访问数组的最后 n 个元素，你可以使用 suffix 方法：

```
let arr = [1,2,3]
let arr2 = arr.suffix(2) // [2,3]
```

suffix 与 prefix 有一个值得注意的特性，那就是超出范围后并不会出错：

```
let arr = [1,2,3]
let arr2 = arr.suffix(10) // [1,2,3] (and no crash)
```

相对于通过数量来描述后缀或前缀的大小，你可以通过索引来表示后缀或前缀的限制：

```
let arr = [1,2,3]
let arr2 = arr.suffixFrom(1) // [2,3]
let arr3 = arr.prefixUpTo(1) // [1]
```

```
let arr4 = arr.prefixThrough(1) // [1,2]
```

数组的 startIndex 属性值是 0，其 endIndex 属性值是其 count。此外，数组的 indices 属性值是个半开区间，其端点是 startIndex 与 endIndex，即访问整个数组的范围。如果通过可变引用来访问这个范围，那就可以修改其 startIndex 与 endIndex 来生成一个新的范围。第 3 章对字符序列就是这么做的；不过，数组的索引值是 Int，因此你可以使用普通的算术运算符：

```
let arr = [1,2,3]
var r = arr.indices
r.startIndex = r.endIndex-2
arr2 = arr[r] // [2,3]
```

indexOf 方法会返回一个元素在数组中首次出现位置处的索引，不过它被包装到了 Optional 中，这样如果数组中不存在这个元素就会返回 nil。如果数组包含了 Equatables，那比较时就可以通过 == 来识别待寻找的元素：

```
let arr = [1,2,3]
let ix = arr.indexOf(2) // Optional wrapping 1
```

即便数组没有包含 Equatables，你也可以提供自定义的函数，它接收一个元素类型并返回一个 Bool，你会得到返回 true 的第一个元素。在如下示例中，Bird 结构体有一个 name String 属性：

```
let aviary = [Bird(name:"Tweety"), Bird(name:"Flappy"), Bird(name:"Lady")]
let ix = aviary.indexOf {$0.name.characters.count < 5} // Optional(2)
```

作为序列，数组的 contains 方法会判断它是否包含了某个元素。如果元素是 Equatables，那么你依然可以使用 == 运算符；也可以提供自定义函数，该函数接收一个元素类型并返回一个 Bool：

```
let arr = [1,2,3]
let ok = arr.contains(2) // true
let ok2 = arr.contains {$0 > 3} // false
```

startsWith 方法判断数组的起始元素是否与给定的相同类型序列的元素相匹配。这里依然可以使用 == 运算符来比较 Equatables；你也可以提供自定义函数，该函数接收两个元素类型值，并返回表示它们是否匹配的 Bool：

```
let arr = [1,2,3]
let ok = startsWith(arr, [1,2]) // true
let ok2 = arr.startsWith([1,-2]) {abs($0) == abs($1)} // true
```

elementsEqual 方法是数组比较的序列泛化：两个序列长度必须相同，其元素要么是 Equatables，要么你自己提供匹配函数。

minElement 与 maxElement 方法分别返回数组中最小与最大的元素，并且被包装到 Optional 中，目的是防止数组为空。如果数组包含了 Comparables，那么你可以使用 < 运算符；此外，你可以提供一个返回 Bool 的函数，表示两个给定元素中较小的那个是不是第一个：

```
let arr = [3,1,-2]
let min = arr.minElement() // Optional(-2)
let min2 = arr.minElement {abs($0)<abs($1)} // Optional(1)
```

如果对数组的引用是可变的，那么 append 与 appendContentsOf 实例方法就会将元素添加到数组末尾。这两个方法的差别在于 append 会接收单个元素类型值，而 appendContentsOf 则接收元素类型的一个序列。比如：

```
var arr = [1,2,3]
arr.append(4)
arr.appendContentsOf([5,6])
arr.appendContentsOf(7...8) // arr is now [1,2,3,4,5,6,7,8]
```

若 + 运算符左侧是个数组，那么 + 就会被重载，其行为类似于 appendContentsOf（而非 append!），只不过它会生成一个新数组，因此即便对数组的引用是个常量，这么做也是可行的。如果对数组的引用是可变的，那么你可以通过 += 运算符对其进行扩展。比如：

```
let arr = [1,2,3]
let arr2 = arr + [4] // arr2 is now [1,2,3,4]
var arr3 = [1,2,3]
arr3 += [4] // arr3 is now [1,2,3,4]
```

如果对数组的引用是可变的，那么实例方法 insert(atIndex:) 就会在指定的索引处插入一个元素。要想同时插入多个元素，请使用范围下标数组进行赋值，就像之前介绍的那样（此外，还有一个 insertContentsOf(at:) 方法）。

如果对数组的引用是可变的，那么实例方法 removeAtIndex 会删除指定索引处的元素；实例方法 removeLast 会删除最后一个元素，removeFirst 则会删除第一个元素。这些方法还会返回从数组中删除的值；如果不需要，那么可以忽略返回值。这些方法不会将返回值包装到 Optional 中，访问越界的索引会导致程序崩溃。另一种形式的 removeFirst 可以指定删除的元素数量，不过它不返回值；如果数组中没有那么多元素，那么程序将会崩溃。

另外，popFirst 与 popLast 则会展开 Optional 中的返回值；这样，即便数组为空也是安全的。如果引用不可变，那么你可以通过 dropFirst 与 dropLast 方法返回删除了最后一个元素后的数组（实际上是个切片）。

joinWithSeparator 实例方法接收一个数组的数组。它会提取出每个元素，并将参数数

组中的元素插入到提取出的每个元素序列之间。结果是个叫作 JoinSequence 的中间序列；如果必要，还需要将其转换为 Array。比如：

```
let arr = [[1,2], [3,4], [5,6]]
let arr2 = Array(arr.joinWithSeparator([10,11]))
// [1, 2, 10, 11, 3, 4, 10, 11, 5, 6]
```

调用 joinWithSeparator 时将空数组作为参数可以将数组的数组打平：

```
let arr = [[1,2], [3,4], [5,6]]
let arr2 = Array(arr.joinWithSeparator([]))
// [1, 2, 3, 4, 5, 6]
```

还有一个 flatten 实例方法也可以做到这一点。它会返回一个中间序列（或集合），你可能需要将其转换为 Array：

```
let arr = [[1,2], [3,4], [5,6]]
let arr2 = Array(arr.flatten())
// [1, 2, 3, 4, 5, 6]
```

reverse 实例方法会生成一个新数组，其元素顺序与原始数组的相反。

sortInPlace 实例方法会对原始数组排序（如果对数组的引用是可变的），而 sort 实例方法则会根据原始数组生成一个新数组。你有两个选择：如果是 Comparables 数组，那就可以通过 < 运算符指定新顺序；此外，你可以提供一个函数，该函数接收两个元素类型参数并返回一个 Bool，表示第 1 个参数是否应该位于第 2 个参数之前（就像 minElement 与 maxElement 一样）。比如：

```
var arr = [4,3,5,2,6,1]
arr.sortInPlace() // [1, 2, 3, 4, 5, 6]
arr.sortInPlace {$0 > $1} // [6, 5, 4, 3, 2, 1]
```

在最后一行代码中，我提供了一个匿名函数。当然，你还可以将一个声明好的函数名作为参数传递进来。在 Swift 中，比较运算符其实就是函数名。因此，我能以更加简洁的方式完成相同的功能，比如：

```
var arr = [4,3,5,2,6,1]
arr.sortInPlace(>) // [6, 5, 4, 3, 2, 1]
```

split 实例方法会在通过测试的元素位置处将一个数组分解为数组的数组，这里的测试指的是一个函数，它接收一个元素类型值并返回 Bool；通过测试的元素会被去除：

```
let arr = [1,2,3,4,5,6]
let arr2 = arr.split {$0 % 2 == 0} // split at evens: [[1], [3], [5]]
```

### 7. 数组枚举与转换

数组是一个序列，因此你可以对其进行枚举，并按照顺序查看或操纵每个元素。最简单的方式是使用 for...in 循环；第 5 章将会对此进行更为详尽的介绍：

```
let pepboys = ["Manny", "Moe", "Jack"]
for pepboy in pepboys {
    print(pepboy) // prints Manny, then Moe, then Jack
}
```

此外，你还可以使用 forEach 实例方法。其参数是个函数，该函数接收数组（或是其他序列）中的一个元素并且没有返回值。你可以将其看作命令式 for...in 循环的函数式版本：

```
let pepboys = ["Manny", "Moe", "Jack"]
pepboys.forEach {print($0)} // prints Manny, then Moe, then Jack
```

如果既需要索引号又需要元素，那么请调用 enumerate 实例方法并对结果进行循环；每次迭代得到的将是一个元组：

```
let pepboys = ["Manny", "Moe", "Jack"]
for (ix,pepboy) in pepboys.enumerate() {
    print("Pep boy \(ix) is \(pepboy)") // Pep boy 0 is Manny, etc.
}
// or:
pepboys.enumerate().forEach {print("Pep boy \($0.0) is \($0.1)")}
```

Swift 还提供了 3 个强大的数组转换实例方法。就像 forEach 一样，这些方法都会枚举数组，这样循环就被隐藏到了方法调用中，代码也变得更加紧凑和整洁。

首先来看看 map 实例方法。它会生成一个新数组，数组中的每个元素都是将原有数组中相应元素传递给你所提供的函数进行处理后的结果。该函数接收一个元素类型的参数，并返回可能是其他类型的结果；Swift 通常会根据函数返回的类型推断出生成的数组元素的类型。

比如，如下代码演示了如何将数组中的每个元素乘以 2：

```
let arr = [1,2,3]
let arr2 = arr.map {$0 * 2} // [2,4,6]
```

如下示例演示了 map 实际上可以生成不同元素类型的新数组：

```
let arr = [1,2,3]
let arr2 = arr.map {Double($0)} // [1.0, 2.0, 3.0]
```

下面是个实际的示例，展示了使用 map 后代码将会变得多么简洁和紧凑。为了删除 UITableView 中一个段中的所有单元格，我需要将单元格定义为 NSIndexPath 对象的数组。如果 sec 是段号，那么我就可以像下面这样分别构建这些 NSIndexPath 对象：

```
let path0 = NSIndexPath(forRow:0, inSection:sec)
let path1 = NSIndexPath(forRow:1, inSection:sec)
// ...
```

这里有个规律！我可以通过 for...in 循环行值来生成 NSIndexPath 对象的数组。不过，要是使用 map，那么表示相同的循环就会变得更加紧凑（ct 是段中的行数）：

```
let paths = Array(0..<ct).map {NSIndexPath(forRow:$0, inSection:sec)}
```

实际上，map 是 CollectionType 的一个实例方法，Range 本身是个 CollectionType。因此，我不需要转换为数组：

```
let paths = (0..<ct).map {NSIndexPath(forRow:$0, inSection:sec)}
```

filter 实例方法也会生成一个新数组。新数组中的每个元素都是老数组中的，顺序也相同；不过，老数组中的一些元素可能会被去除，它们被过滤掉了。起过滤作用的是你所提供的函数；它接收一个元素类型的参数并返回一个 Bool，表示这个元素是否应该被放到新数组中。

比如：

```
let pepboys = ["Manny", "Moe", "Jack"]
let pepboys2 = pepboys.filter{$0.hasPrefix("M")} // [Manny, Moe]
```

最后来看看 reduce 实例方法。如果学过 LISP 或 Scheme，那么你对 reduce 就会很熟悉；否则，初学起来会觉得很困惑。这是一种将数组中（实际上是序列）所有元素合并为单个值的方式。这个值的类型（结果类型）不必与数组元素类型相同。你提供了一个函数，它接收两个参数：第 1 个是结果类型，第 2 个是元素类型，结果是这两个参数的组合，它们作为结果类型。每次迭代的结果会作为下一次迭代的第一个参数，同时数组的下一个元素会作为第二个参数。因此，组合对不断累积的输出，以及最终的累积值就是 reduce 函数最终的输出。不过，这并没有说明第一次迭代的第一个参数来自哪里。答案就是你需要自己提供 reduce 调用的第一个参数。

通过一个简单的示例说明会加强理解。假设有一个 Int 数组，接下来我们可以通过 reduce 得到数组中所有元素的和。如下伪代码省略了 reduce 调用的第一个参数，这样你就可以思考它应该是什么了：

```
let sum = arr.reduce(/*???*/) {$0 + $1}
```

每一对参数都会一起添加进去，从而得到下一次迭代时的第一个参数。每次迭代时的第 2 个参数都是数组中的元素。那么问题来了，数组的第一个元素会与什么相加呢？我们想要得到所有元素的和，既不多也不少；显然，数组的第一个元素应该与 0 相加！下面是具体代码：

```
let arr = [1, 4, 9, 13, 112]
let sum = arr.reduce(0) {$0 + $1} // 139
```

上述代码可以更加简洁一些，因为 + 运算符是所需类型的函数名：

```
let sum = arr.reduce(0, combine:+)
```

在我的 iOS 编程生涯中，我大量使用了这些方法，通常使用其中 2 个，或 3 个都用，将它们嵌套起来、链接起来，或二者结合起来使用。下面来看个示例；这个示例很复杂，但却能非常好地展现出通过 Swift 来处理数组是多么简洁。我有一个表视图，它将数据以段的形式展现出来。在底层，数据是个 String 数组的数组，每个子数组都表示段的行。现在，我想要过滤该数据，去除不包含某个子字符串的所有字符串。我想要保持段的完整性，不过如果删除字符串导致一个段的所有字符串都被删除，那么我需要删除整个段数组。

其核心是判断一个字符串是否包含了子字符串。这里使用的是 Cocoa 方法，因为可以通过它们执行不区分大小写的搜索。如果 s 是数组中的字符串，并且 target 是我们要搜索的子字符串，那么判断 s 是否不区分大小写地包含了 target 的代码如下所示：

```
let options = NSStringCompareOptions.CaseInsensitiveSearch
let found = s.rangeOfString(target, options: options)
```

回忆一下第 3 章介绍的 rangeOfString。如果 found 不为 nil，那就说明找到了子字符串。下面是具体代码，前面加上了一些示例数据：

```
let arr = [["Manny", "Moe", "Jack"], ["Harpo", "Chico", "Groucho"]]
let target = "m"
let arr2 = arr.map {
    $0.filter {
        let options = NSStringCompareOptions.CaseInsensitiveSearch
        let found = $0.rangeOfString(target, options: options)
        return (found != nil)
    }
}.filter {$0.count > 0}
```

前两行代码设定了示例数据，剩下的是一条命令：一个 map 调用，其函数包含了一个 filter 调用，后面又链接了一个 filter 调用。如果上述代码都说明不了 Swift 有多么酷，那就没别的能够说明了。

## 8. Swift Array 与 Objective-C NSArray

在编写 iOS 应用时，你会导入 Foundation 框架（或是 UIKit，因为它会导入 Foundation），它包含了 Objective-C NSArray 类型。Swift 的 Array 类型与 Objective-C 的 NSArray 类型是桥接的；不过，前提是数组中的元素类型可以桥接。相比于 Swift，

Objective-C 对于 NSArray 中可以放置什么元素的规则既宽松又严格。一方面，NSArray 中的元素不必是相同类型的。另一方面，NSArray 中的元素必须是对象，因为只有对象才能为 Objective-C 所理解。一般来说，如果类型能够向上转换为 AnyObject（这意味着它是个类类型），或是如 Int、Double 及 String 这样特殊的桥接结构体，那么它才可以桥接到 Objective-C。

将 Swift 数组传递给 Objective-C 通常是很简单的。如果 Swift 数组包含了可以向上类型转换为 AnyObject 的对象，那么直接传递数组即可；要么通过赋值，要么作为函数调用的实参：

```
let arr = [UIBarButtonItem(), UIBarButtonItem()]
self.navigationItem.leftBarButtonItems = arr
self.navigationItem.setLeftBarButtonItems(arr, animated: true)
```

要想在 Swift 数组上调用 NSArray 方法，你需要将其转换为 NSArray：

```
let arr = ["Manny", "Moe", "Jack"]
let s = (arr as NSArray).componentsJoinedByString(", ")
// s is "Manny, Moe, Jack"
```

Swift 数组可以看出 `var` 引用是可变的，不过无论怎么看，NSArray 都是不可变的。要想在 Objective-C 中获得可变数组，你需要 NSArray 的子类 NSMutableArray。你不能将 Swift 数组通过类型转换、赋值或传递的方式赋给 NSMutableArray，必须要强制进行。最佳方式是调用 NSMutableArray 的初始化器 `init(array:)`，你可以直接向其传递一个 Swift 数组：

```
let arr = ["Manny", "Moe", "Jack"]
let arr2 = NSMutableArray(array:arr)
arr2.removeObject("Moe")
```

将 NSMutableArray 转换回 Swift 数组只需直接转换即可；如果需要一个拥有原始 Swift 类型的数组，那就需要转换两次才能编译通过：

```
var arr = ["Manny", "Moe", "Jack"]
let arr2 = NSMutableArray(array:arr)
arr2.removeObject("Moe")
arr = arr2 as NSArray as! [String]
```

如果 Swift 对象类型不能向上转换为 AnyObject，那么它就无法桥接到 Objective-C；如果需要一个 NSArray，但你传递了一个包含这种类型的 Array，那么编译器就会报错。在这种情况下，你需要手工“桥接”数组元素。

比如，我有一个 Swift 的 CGPoint 数组。这在 Swift 中是没问题的，不过由于 CGPoint 是个结构体，而 Objective-C 并不会将其视为一个对象，因此你不能将其放到 NSArray

中。如果在需要 NSArray 的地方传递了这个数组，那就会导致编译错误："[CGPoint] is not convertible to NSArray."。解决办法就是将每个 CGPoint 包装为 NSValue，这是个 Objective-C 对象类型，专门用作各种非对象类型的载体；现在，我们有了一个 Swift 的 NSValue 数组，接下来就可以由 Objective-C 进行处理了：

```
let arrNSValues = arrCGPoints.map { NSValue(CGPoint:$0) }
```

另一种情况是 Swift 的 Optional 数组。Objective-C 集合不能包含 nil（因为在 Objective-C 中，`nil` 不是对象）。因此，你不能将 Optional 放到 NSArray 中。在需要 NSArray 时如果传递 Optional 数组，那就需要事先对这些 Optional 进行处理。如果 Optional 包装了值，那么你可以将其展开。不过，如果 Optional 没有包装值（它是个 `nil`），那么就无法将其展开。一种解决办法就是采取 Objective-C 中的做法。Objective-C NSArray 不能包含 `nil`，因此 Cocoa 提供了一个特殊的类 NSNull，当需要一个对象时，其单例 `NSNull()` 可以代替 `nil`。这样，如果有一个包装了 String 的 Optional 数组，那么我可以将那些不为 `nil` 的元素展开，并使用 `NSNull()` 替代 `nil` 元素：

```
let arr2 : [AnyObject] =
    arr.map{if $0 == nil {return NSNull()} else {return $0!}}
```

（第 5 章将会进一步简化上述代码。）

现在来看看将 NSArray 从 Objective-C 传递给 Swift 时会发生什么。跨越桥接不会有任何问题：NSArray 会安全地变成 Swift Array。不过，这是个什么类型的 Swift Array 呢？就其本身来说，NSArray 并没有携带关于它所包含的元素类型的任何信息。因此，默认就是 Objective-C NSArray 会转换为 Swift 的 AnyObject 数组。

幸好，现在不会像之前那样再遇到这种默认情况了。从 Xcode 7 开始，Objective-C 语言发生了变化，NSArray、NSDictionary 与 NSSet 的声明（这 3 种集合类型会桥接到 Swift）已经包含了元素类型信息（Objective-C 称为轻量级泛型）。在 iOS 9 中，Cocoa API 也得到了改进，它们包含了这些信息。这样，在大多数情况下，从 Cocoa 接收到的数组都是带有类型的。

比如，如下优雅的代码在之前是不可能的：

```
let arr = UIFont.familyNames().map {
    UIFont.fontNamesForFamilyName($0)
}
```

结果是一个 String 数组的数组，根据字体系列列出了所有可用的字体。上述代码可以编译通过，因为 Swift 会看到 UIFont 的这两个类方法返回了一个 String 数组。之前，这两个数组是没有类型信息的（它们都是 AnyObject 数组），你需要将其向下类型转换为 String 数组。

虽然不常见，但你还是有可能从 Objective-C 接收到 AnyObject 数组。如果出现了这种情况，那么通常你会将其进行向下类型转换，或是将其转换为特定 Swift 类型的数组。如下 Objective-C 类包含了一个方法，其 NSArray 返回类型不带元素类型：

```
@implementation Pep
- (NSArray*) boys {
    return @[@"Mannie", @"Moe", @"Jack"];
}
@end
```

要想调用该方法并对结果进行处理，你需要将结果向下类型转换为 String 数组。如果确信这一点，那就可以执行强制类型转换：

```
let p = Pep()
let boys = p.boys() as! [String]
```

不过，与任何类型转换一样，请确保你的转换是正确的！ Objective-C 数组可以包含多种对象类型。不要将这样的数组强制向下类型转换为并非每个元素都能转换的类型，否则一旦转换失败，程序将会崩溃；在排除或转换有问题的元素时要深思熟虑。

## 4.12.2 Dictionary

字典（Dictionary，是个结构体）是成对对象的一个无序集合。对于每一对对象来说，第 1 个对象是键，第 2 个对象是值。其用法是通过键来访问值。键通常是字符串，不过并不局限为字符串；形式上的要求是键的类型要使用 Hashable 协议，这意味着它们使用了 Equatable，并且有一个 `hashValue` 属性（一个 Int），这样两个相等的键就会拥有相等的散列值，而两个不相等的键的散列值也不等。因此，背后可以通过散列值实现键的快速访问。Swift 的数字类型、字符串与枚举都是 Hashable。

就像数组一样，给定的字典类型必须是统一的。键类型与值类型不必是相同类型，通常情况下其类型也不相同。不过在任何字典中，所有键的类型都必须是相同的，所有值的类型也必须是相同的。字典其实是个泛型，其占位符类型先是键类型，然后是值类型：`Dictionary<Key,Value>`。不过与数组一样，Swift 为字典类型的表示提供了语法糖，通常情况下都会这么用：`[Key: Value]`，即方括号中包含了一个冒号（以及可选的空格），两边是键类型与值类型。如下代码创建了一个空的字典，其键（如果存在）是 String，值（如果存在）也是 String：

```
var d = [String:String]()
```

冒号还用于字典字面值语法中，用于分隔每一对键值。键值对位于方括号中，中间用逗号分隔，就像数组一样。如下代码通过字面值方式创建了一个字典（字典的类型 `[String:String]` 会被推断出来）：

```
var d = ["CA": "California", "NY": "New York"]
```

空字典的字面值是个里面只包含了一个冒号 [:] 的方括号。如果通过其他方式获悉了字典的类型，那就可以使用这个符号表示。下面是创建空字典（[String:String]）的另一种方式：

```
var d : [String:String] = [:]
```

如果通过不存在的键获取值，那么不会出现错误，不过 Swift 需要通过一种方式告知你这个操作失败了；因此，它会返回 nil。这反过来又会表示，如果成功通过一个键访问到了值，那么返回的值一定是个包装真实值的 Optional！

我们常常通过下标访问字典的内容。要想根据键获取其值，请对字典引用应用下标，下标中是键：

```
let d = ["CA": "California", "NY": "New York"]
let state = d["CA"]
```

不过请记住，在上述代码执行后，state 并不是 String，它是个包装了 String 的 Optional！忘记这一点是很多初学者常犯的错误。

如果对字典的引用是可变的，那么你还可以对键下标表达式赋值。如果键已经存在，那么其值就会被替换。如果键不存在，那么它会被创建，并且将值关联到键上：

```
var d = ["CA": "California", "NY": "New York"]
d["CA"] = "Casablanca"
d["MD"] = "Maryland"
// d is now ["MD": "Maryland", "NY": "New York", "CA": "Casablanca"]
```

还可以调用 updateValue(forKey:)；好处在于它会将旧值包装到 Optional 中返回，如果键不存在则会返回 nil。

作为一种便捷方式，如果键存在，那么将 nil 赋给键下标表达式会将该键值对删除：

```
var d = ["CA": "California", "NY": "New York"]
d["NY"] = nil // d is now ["CA": "California"]
```

还可以调用 removeValueForKey；好处在于在删除键值对之前它会返回被删除的值。返回的被删除的值会包装到一个 Optional 中；因此，如果返回 nil，那就表示这个键本来就不在字典中。

与数组一样，字典类型也可以进行向下类型转换，这意味着其中的每个元素都会进行向下类型转换。通常只有值类型会不同：

```
let dog1 : Dog = NoisyDog()
let dog2 : Dog = NoisyDog()
```

```
let d = ["fido": dog1, "rover": dog2]
let d2 = d as! [String : NoisyDog]
```

与数组一样，is 可用于测试字典中的实际类型，as? 可用于测试并安全地进行类型转换。与数组相等性一样，字典相等性的工作方式与你想的是一样的。

### 1. 基本的字典属性与枚举

字典有个 count 属性，它会返回字典中所包含的键值对数量；还有一个 isEmpty 属性，用于判断这个数量是否为 0。

字典有个 keys 属性，它会返回字典中所有的键；还有一个 values 属性，它会返回字典中所有的值。它们都是没有对外公开的结构体（实际类型是 LazyForwardCollection），不过在通过 for...in 枚举它们时，你会得到期望的类型：

```
var d = ["CA": "California", "NY": "New York"]
for s in d.keys {
    print(s) // s is a String
}
```

字典是无序的！你可以枚举它（键或值），但不要期望元素会以任何特定的顺序返回。

可以通过将 keys 属性或 values 属性转换为数组一次性获得字典的键或值：

```
var d = ["CA": "California", "NY": "New York"]
var keys = Array(d.keys)
```

还可以枚举字典本身。你可能已经想到了，每次迭代都会得到一个键值对元组：

```
var d = ["CA": "California", "NY": "New York"]
for (abbrev, state) in d {
    print("\(abbrev) stands for \(state)")
}
```

可以通过将字典转换为数组，从而一次性以数组（键值对元组）的形式获得字典的全部内容：

```
var d = ["CA": "California", "NY": "New York"]
let arr = Array(d) // [("NY", "New York"), ("CA", "California")]
```

就像数组一样，字典、其 keys 属性与 values 属性都是集合（CollectionType）与序列（SequenceType）。因此，上面所介绍的关于作为集合与序列的数组的一切也都适用于字典！比如，如果字典 d 有 Int 值，那么你可以通过 reduce 实例方法求出其和：

```
let sum = d.values.reduce(0, combine:+)
```

可以获取其最小值（包装在 Optional 中）：

```
let min = d.values.minElement()
```

可以列出符合某个标准的值：

```
let arr = Array(d.values.filter{$0 < 2})
```

（这里需要转换为 Array，因为 `filter` 得到的序列是延迟的：直到枚举它或将其放到数组中后，它里面才会有内容）。

## 2. Swift Dictionary 与 Objective-C NSDictionary

Foundation 框架中的字典类型是 NSDictionary，而 Swift 的 Dictionary 类型会与其桥接。在双方之间传递字典就像之前介绍的数组那样。NSDictionary 的无类型信息的桥接 API 的类型是 `[NSObject:AnyObject]`，它使用 Objective-C Foundation 对象基类作为键；之所以这么做有几个原因，不过从 Swift 的视角来看，主要的原因在于 AnyObject 并非 Hashable。另一方面，NSObject 被 Swift API 扩展了，并且使用了 Hashable；由于 NSObject 是 Cocoa 类的基类，因此任何 Cocoa 类型都是 NSObject。这样，NSDictionary 就可以桥接了。

就像 NSArray 一样，NSDictionary 的键与值类型现在可以在 Objective-C 中标记了。在实际的 Cocoa NSDictionary 中，最常使用的键类型是 NSString，因此接收到的 NSDictionary 会是个 `[String:AnyObject]`。不过，NSDictionary 中拥有特定类型值的情况并不多见；传给 Cocoa 或从 Cocoa 接收的字典通常具有不同类型的值。一个字典的键是字符串，但值包含了字符串、数字、颜色以及数组这一情况是非常常见的。出于这一原因，你通常不会对整个字典的类型进行向下类型转换；相反，你在使用字典时会将值看作 AnyObject，在从字典中获取到单个值时才进行类型转换。由于从下标键中返回的值本身是个 Optional，所以常常需要先展开值，然后再进行类型转换。

下面是个示例。Cocoa NSNotification 对象有个 `userInfo` 属性。它是个 NSDictionary，本身可能为 `nil`，因此 Swift API 是这样描述它的：

```
var userInfo: [NSObject : AnyObject]? { get }
```

假设这个字典包含一个 `"progress"` 键，其值是个 NSNumber，值里面包含着一个 Double。我的目标是将该 NSNumber 提取出来，并将其包含到 Double 赋给属性 `self.progress`。下面是一种安全的做法，使用可选展开与可选类型转换（n 是个 NSNotification 对象）：

```
let prog = (n.userInfo?["progress"] as? NSNumber)?.doubleValue
if prog != nil {
    self.progress = prog!
}
```

这是个 Optional 链，链的最后会获取 NSNumber 的 doubleValue 属性；因此，prog 的隐式类型是个包装了 Double 的 Optional。上述代码是安全的，因为如果没有 userInfo 属性，或字典中不包含 "progress" 键，或键的值不是个 NSNumber，那么什么都不会发生，prog 值就是 nil。接下来判断 prog 是否为 nil；如果不是，那我就可以安全地强制展开它了，并且展开的值就是我所期望的 Double。

（第 5 章将会介绍完成相同事情的另一种语法，使用了条件绑定。）

与之相反，下面这个示例会创建一个字典并将其传递给 Cocoa。该字典是个混合体：其值包含了 UIFont、UIColor 及 NSShadow；其键都是字符串，并且是从 Cocoa 中获取的常量。我以字面值形式构造这个字典，然后将其传递过去，整个过程一步搞定，完全不需要类型转换：

```
UINavigationBar.appearance().titleTextAttributes = [
    NSFontAttributeName : UIFont(name: "ChalkboardSE-Bold", size: 20)!,
    NSForegroundColorAttributeName : UIColor.darkTextColor(),
    NSShadowAttributeName : {
        let shad = NSShadow()
        shad.shadowOffset = CGSizeMake(1.5,1.5)
        return shad
    }()
]
```

与 NSArray 和 NSMutableArray 一样，如果希望 Cocoa 能够修改字典，那就需要将其转换为 NSMutableDictionary。在如下示例中，我想要连接两个字典，因此使用了 NSMutableDictionary，它有一个 addEntriesFromDictionary: 方法：

```
var d1 = ["NY":"New York", "CA":"California"]
let d2 = ["MD":"Maryland"]
let mutd1 = NSMutableDictionary(dictionary:d1)
mutd1.addEntriesFromDictionary(d2)
d1 = mutd1 as NSDictionary as! [String:String]
// d1 is now ["MD": "Maryland", "NY": "New York", "CA": "California"]
```

这种事情经常会遇到，因为并没有将一个字典的元素添加到另一个字典中的原生方法。实际上在 Swift 中，与字典相关的原生辅助方法数量是非常少的：其实根本就没有。Cocoa 与 Foundation 框架还可以为我们所用，也许 Apple 觉得没必要在 Swift 标准库中重复 Foundation 中已经存在的那些功能。如果觉得使用 Cocoa 很麻烦，那么你可以编写自己的库；比如，我们可以通过扩展轻松将 addEntriesFromDictionary: 重新实现为 Swift Dictionary 的实例方法：

```
extension Dictionary {
    mutating func addEntriesFromDictionary(d:[Key:Value]) { // generic types
        for (k,v) in d {
            self[k] = v
        }
    }
}
```

## 4.12.3 Set

集合（Set，是个结构体）是不重复对象的一个无序集合。它非常类似于字典的键！其元素必须是相同类型的，它有一个 `count` 属性和一个 `isEmpty` 属性；可以通过任意序列进行初始化；你可以通过 `for...in` 遍历其元素。不过，元素的顺序是不确定的，你不应该假定元素的顺序。

Set 元素的唯一性是通过限制其类型使用 Hashable 协议来做到的，就像 Dictionary 的键一样。因此，背后可以使用散列值来加速访问。你可以通过 `contains` 实例方法判断一个集合中是否包含了给定的元素，其效率要比对数组进行相同的操作高很多。因此，如果元素的唯一性是可以接受的（或需要这样），并且不需要索引，也不需要确保顺序，那么相比于数组，Set 会是一个更好的选择。

集合的元素是 Hashables，这意味着它们一定也都是 Equatables。这是非常有意义的，因为唯一性这个概念取决于能够回答给定对象在集合中是否已经存在这一问题。

Swift 中并没有 Set 字面值，你也不需要，因为在需要集合的地方可以传递一个数组字面值。Swift 也没有提供集合类型表示的语法糖，因为 Set 结构体是一个泛型，因此可以通过显式特化泛型来表示类型信息：

```
let set : Set<Int> = [1, 2, 3, 4, 5]
```

不过在上述示例中，我们无须特化泛型，因为 Int 类型可以通过数组推断出来。

很多时候你想要获取到集合中的某一个元素作为样本。由于顺序是无意义的，因此获取任意一个元素就可以，如第一个元素。如果出于这个目的，你可以使用 `first` 实例属性；它会返回一个 Optional，以防止集合为空，没有第一个元素。

集合的标志性特性在于其对象的唯一性。如果将对象添加到集合中，同时集合中已经包含了该对象，那么它就不会被再次添加进去。将数组转换为集合，然后又将集合转换为数组是一种快速且可靠的确保数组元素唯一性的方式，不过数组元素的顺序并不会保留下来：

```
let arr = [1,2,1,3,2,4,3,5]
let set = Set(arr)
```

```
let arr2 = Array(set) // [5,2,3,1,4], perhaps
```

Set是一种集合（CollectionType），也是个序列（SequenceType），这类似于数组与字典，之前介绍的关于这两种类型的一切也都适用于Set。比如，Set有map实例方法；它返回一个数组，当然，如果需要也可以将其转换回Set：

```
let set : Set = [1,2,3,4,5]
let set2 = Set(set.map {$0+1}) // {6, 5, 2, 3, 4}, perhaps
```

如果对集合的引用是可变的，那就有很多实例方法可供使用了。你可以通过insert向集合添加对象；如果对象已经在集合中，那就什么都不会发生，但也没有问题。可以通过remove方法从集合中删除指定对象并返回；它会返回包装在Optional中的对象，如果对象不存在，那么该方法会返回nil。可以通过removeFirst方法删除并返回集合中的第1个对象（无论第1个指的是什么）；如果集合为空，那么应用就会崩溃，因此请小心行事（或使用安全的popFirst）。

集合的相等性比较（==）与你期望的是一致的；如果一个集合中的每个元素都与另一个集合中的元素相等，那么这两个集合就是相等的。

如果集合的概念让你想起了小学时学到的文氏图，那就太好了，因为集合提供的实例方法可以让你实现当初学到的所有集合操作。参数可以是集合，也可以是序列（会被转换为集合）；比如，可以是数组、范围，甚至是字符序列：

intersect、intersectInPlace
: 找出该集合与参数中都存在的元素。

union、unionInPlace
: 找出该集合与参数中元素的合集。

exclusiveOr、exclusiveOrInPlace
: 找出在该集合，但不在参数中的元素，以及在参数，但不在该集合中的元素的合集。

subtract、subtractInPlace
: 找出在该集合，但不在参数中的元素。

isSubsetOf、isStrictSubsetOf
isSupersetOf、isStrictSupersetOf
: 返回一个Bool值，判断该集合中的元素是否都在参数中，或判断参数中的元素是否都在该集合中。如果两个集合包含了相同的元素，那么“strict版本”就会返回false。

isDisjointWith
: 返回一个Bool值，判断该集合和参数是否没有相同的元素。

如下示例演示了如何优雅地使用 Set，它来自于我所编写的一个应用。应用中有很多带编号的图片，我们要从中随机选取一个。不过，我不想选取最近已经选取过的图片。因此，我维护了一个最近选取过的所有图片的编号列表。在选取新的图片时，我会将所有编号的列表转换为一个 Set，同时将最近选取过的图片的编号列表转换为一个 Set，然后二者相减得到一个未使用过的图片编号列表！现在，我可以随机选取一个图片编号，并将其添加到最近使用过的图片编号列表中：

```
let ud = NSUserDefaults.standardUserDefaults()
var recents = ud.objectForKey(RECENTS) as? [Int]
if recents == nil {
    recents = []
}
var forbiddenNumbers = Set(recents!)
let legalNumbers = Set(1...PIXCOUNT).subtract(forbiddenNumbers)
let newNumber = Array(legalNumbers)[
    Int(arc4random_uniform(UInt32(legalNumbers.count)))
]
forbiddenNumbers.insert(newNumber)
ud.setObject(Array(forbiddenNumbers), forKey:RECENTS)
```

## 1. Option Set

Option Set（从技术上来说是 OptionSetType）是 Swift 提供的将 Cocoa 中常用的一些枚举类型当作结构体的一种方式。严格来说，它并不是 Set；不过看起来像是个 Set，它通过 SetAlgebraType 协议实现了 Set 的诸多特性。因此，Option Set 也拥有 `contains`、`insert`、`remove` 方法，以及各种集合操作方法。

Option Set 的目的在于帮助你处理 Objective-C 的位掩码。位掩码是个整型，当同时指定多个选项时，它们用作开关。这种位掩码在 Cocoa 中用得非常多。在 Objective-C 以及 Swift 2.0 之前，我们通过算术按位或和按位与运算符来操纵位掩码。这种操作令人感到不可思议，并且极易出错。多亏了 Option Set，在 Swift 2.0 中，我们可以通过集合操作来操纵位掩码。

比如，在指定 UIView 动画时，我们可以传递一个 `options:` 实参，它的值来自于 UIViewAnimationOptions 枚举，其定义（在 Objective-C 中）以如下内容开始：

```
typedef NS_OPTIONS(NSUInteger, UIViewAnimationOptions) {
    UIViewAnimationOptionLayoutSubviews            = 1 << 0,
    UIViewAnimationOptionAllowUserInteraction      = 1 << 1,
    UIViewAnimationOptionBeginFromCurrentState     = 1 << 2,
    UIViewAnimationOptionRepeat                    = 1 << 3,
    UIViewAnimationOptionAutoreverse               = 1 << 4,
    // ...
};
```

假设一个 NSUInteger 是 8 位（实际上不是，这里做了一些简化）。那么，该枚举（在 Swift 中）定义了如下名值对：

```
UIViewAnimationOptions.LayoutSubviews            0b00000001
UIViewAnimationOptions.AllowUserInteraction      0b00000010
UIViewAnimationOptions.BeginFromCurrentState     0b00000100
UIViewAnimationOptions.Repeat                    0b00001000
UIViewAnimationOptions.Autoreverse               0b00010000
```

可以将这些值组合为单个值（位掩码），并将其作为动画的 `options:` 实参传递过去。为了理解你的意图，Cocoa 只需查看你所传递的值中哪些位被设为了 1。比如，`0b00011000` 表示 `UIViewAnimationOptions.Repeat` 与 `UIViewAnimationOptions.Autoreverse` 都为 true（也就表示其他都是 false）。

问题在于如何构造值 `0b00011000` 来传递。你可以直接将其构造为字面值，并将 `options:` 实参设为 `UIViewAnimationOptions(rawValue:0b00011000)`；不过，这么做可不太好，因为极易出错，并且会导致代码难以理解。在 Objective-C 中，你可以使用算术按位或运算符，这类似于如下 Swift 代码：

```
let val =
    UIViewAnimationOptions.Autoreverse.rawValue |
    UIViewAnimationOptions.Repeat.rawValue
let opts = UIViewAnimationOptions(rawValue: val)
```

不过在 Swift 2.0 中，UIViewAnimationOptions 类型是个 Option Set 结构体（因为它在 Objective-C 中被标记为 `NS_OPTIONS`），因此可以像 Set 一样使用它。比如，给定一个 UIViewAnimationOptions 值，你可以通过 `insert` 向其添加一个选项：

```
var opts = UIViewAnimationOptions.Autoreverse
opts.insert(.Repeat)
```

此外，还可以从数组字面值开始，就像初始化 Set 一样：

```
let opts : UIViewAnimationOptions = [.Autoreverse, .Repeat]
```

要想不设定选项，请传递一个空的 Option Set（`[]`）。这是相对于 Swift 1.2 及之前的版本一个较大的变化（之前的约定是传递 `nil`），这是不合逻辑的，因为该值永远不会为 Optional。

相反的情况是 Cocoa 传递给你一个位掩码，你想知道是否设置了其中某一位。在这个来自于 UITableViewCell 子类的示例中，单元格的 state 以位掩码的形式传递给了我们；我

们想要知道表示单元格是否显示其编辑控件的位信息。过去，我们需要提取出原始值并使用按位与运算符：

```
override func didTransitionToState(state: UITableViewCellStateMask) {
    let editing = UITableViewCellStateMask.ShowingEditControlMask.rawValue
    if state.rawValue & editing != 0 {
        // ... the ShowingEditControlMask bit is set ...
    }
}
```

这么做太容易出错了。在 Swift 2.0 中，它是个 Option Set，因此使用 contains 方法即可：

```
override func didTransitionToState(state: UITableViewCellStateMask) {
    if state.contains(.ShowingEditControlMask) {
        // ... the ShowingEditControlMask bit is set ...
    }
}
```

## 2. Swift Set 与 Objective-C NSSet

Swift 的 Set 类型会桥接到 Objective-C NSSet，中间类型是 Set<NSObject>，因为 NSObject 会被看作 Hashable。当然，同样的规则也适用于数组。Objective-C NSSet 要求元素是类实例，Swift 则会进行桥接。在实际开发中，你可能会使用一个数组，然后将其转换为集合或传递给需要集合的地方，如下示例来自于我所编写的代码：

```
let types : UIUserNotificationType = [.Alert, .Sound] // a bitmask
let category = UIMutableUserNotificationCategory()
category.identifier = "coffee"
let settings = UIUserNotificationSettings( // second parameter is an NSSet
    forTypes: types, categories: [category])
```

如果 Objective-C 不知道这个 Set 是什么类型，那么从 Objective-C 返回的就是一个 NSObject Set，在这种情况下，你可以对其进行向下类型转换。不过与 NSArray 一样，现在可以对 NSSet 进行标记以表示其元素类型；很多 Cocoa API 都已经被标记了，因此无需类型转换：

```
override func touchesBegan(touches: Set<UITouch>, withEvent event: UIEvent?) {
    let t = touches.first // an Optional wrapping a UITouch
    // ...
}
```

# 第 5 章

# 流程控制与其他

本章将会介绍 Swift 语言剩余的其他方面。首先将会介绍 Swift 分支、循环与跳转流程控制结构的语法，然后再来介绍如何重写运算符以及如何创建自定义运算符。最后将会介绍 Swift 的隐私性与内省特性，以及用于引用类型内存管理的专用模式。

## 5.1 流程控制

计算机程序都有通过代码语句表示的执行路径。正常来说，这个路径会遵循着一个简单的规则：连续执行每一条语句。不过还有另外的可能。流程控制用于让执行路径跳过某些代码语句，或是重复执行一些代码语句。流程控制使得计算机程序变得“智能”，而不只是执行简单、固定的一系列步骤。通过测试条件（结果为 Bool 的表达式，因此值为 `true` 或 `false`）的真值，程序可以确定如何继续。基于条件测试的流程控制大体上可以分为以下两种类型。

*分支*

代码被划分为不同的区块，就像树林中分叉的路一样，程序有几种可能进行下去的方式：条件真值用于确定哪一个代码区块会被真正执行。

*循环*

将代码块划分出来以重复执行：条件真值用于确定代码块是否应该执行，然后是否应该再次执行。每次重复都叫作一次迭代。一般来说，每次迭代时都会改变一些环境特性（比如，变量的值），这样重复就不是一样的了，而是整个任务处理中的连续阶段。

流程控制中的代码块（称为块）是由花括号包围的。这些花括号构成了一个作用域。可

以在里面声明新的局部变量，当执行路径离开花括号时，这些局部变量就会自动消亡。对于循环来说，这意味着局部变量在每次迭代时都会创建出来，然后消亡。就像其他作用域那样，花括号中的代码可以看到外部更高层次的作用域。

Swift 流程控制相当简单，总的来说类似于 C 及相关语言。Swift 与 C 存在两种基本的语法差别，这些差别使得 Swift 变得更加简单和整洁：在 Swift 中，条件不必放到圆括号中，不过花括号是不能省略的。此外，Swift 还添加了一些专门的流程控制特性，帮助你更方便地使用 Optional，同时又提供了更为强大的 switch 语句。

## 5.1.1 分支

Swift 有两种形式的分支：if 结构以及 switch 语句。此外，我还会介绍条件求值，它是 if 结构的一种紧凑形式。

### 1. if 结构

Swift 的 `if` 分支结构类似于 C，本书之前已经出现过很多 if 结构的示例。示例 5-1 总结了 if 结构的形式。

示例 5-1：Swift if 结构

```
if condition {
    statements
}

if condition {
    statements
} else {
    statements
}

if condition {
    statements
} else if condition {
    statements
} else {
    statements
}
```

第 3 种形式包含了 `else if`，其实它可以根据需要包含多个 `else if`，最后的 `else` 块可以省略。

下面的 if 结构来自于我所编写的一个应用：

## 自定义嵌套作用域

有时，当知道某个局部变量只需要在几行代码中存在时，你可能会自己定义一个作用域——自定义嵌套作用域，在作用域的开头引入该局部变量，在作用域的结尾该变量会离开，并且其值会自动销毁。

不过，Swift 却不允许你使用空的花括号来这么做。在 Swift 1.2 及之前的版本中，通常的做法是采取一些欺骗的手段，比如，滥用某种形式的流程控制，使之引入合法的嵌套作用域，如 `if true`。Swift 2.0 则提供了 `do` 结构来实现这个目的：

```
do {
    var myVar = "howdy"
    // ... use myVar here ...
}
// now myVar is out of scope and its value is destroyed
```

```
// okay, we've tapped a tile; there are three cases
if self.selectedTile == nil { // no selected tile: select and play this tile
    self.selectTile(tile)
    self.playTile(tile)
} else if self.selectedTile == tile { // selected tile tapped: deselect it
    self.deselectAll()
    self.player?.pause()
} else { // there was a selected tile, another tile tapped: swap them
    self.swap(self.selectedTile, with:tile, check:true, fence:true)
}
```

### 2. 条件绑定

在 Swift 中，`if` 后可以紧跟变量声明与赋值，也就是说，其后面可以是 `let` 或 `var`，然后是新的局部变量名，后面还可以加上一个冒号以及类型声明，然后是等号和一个值。该语法（叫作条件绑定）实际上是条件展开 Optional 的简写。赋的值是个 Optional（如果不是，则编译器会报错），说明如下。

- 如果 Optional 为 `nil`，那么条件会失败，块也不会执行。
- 如果 Optional 不为 `nil`，那么：
  1. Optional 会被展开。
  2. 展开的值会被赋给声明的局部变量。
  3. 块执行时局部变量会处于作用域中。

因此，条件绑定是将展开的 Optional 安全地传递给块的一种便捷方式。只有 Optional 可以展开块才会执行。

条件绑定中的局部变量可以与外部作用域中的已有变量同名。它甚至可以与被展开的Optional同名！没必要创建新的名字，块中的Optional展开值会覆盖原来的Optional，这样就不可能无意中访问到它。

下面是条件绑定的一个示例。如下代码来自于第4章，我展开了NSNotification的`userInfo`字典，尝试通过“progress”键从字典中获取值，并且只在该值为NSNumber时才继续：

```
let prog = (n.userInfo?["progress"] as? NSNumber)?.doubleValue
if prog != nil {
    self.progress = prog!
}
```

可以通过条件绑定重写上述代码：

```
if let prog = (n.userInfo?["progress"] as? NSNumber)?.doubleValue {
    self.progress = prog
}
```

还可以对条件绑定进行嵌套。为了说明这一点，我要重写上一个示例，对链中的每个Optional使用单独的条件绑定：

```
if let ui = n.userInfo {
    if let prog : AnyObject = ui["progress"] {
        if let prog = prog as? NSNumber {
            self.progress = prog.doubleValue
        }
    }
}
```

结果更为冗长，嵌套层次也更深（Swift程序员管这叫作“末日金字塔”），不过我却认为其可读性更好，因为其结构能很好地反映出连续的测试过程。为了避免缩进，可以将连续的条件绑定放到一个列表中，中间通过逗号分隔：

```
if let ui = n.userInfo, prog = ui["progress"] as? NSNumber {
    self.progress = prog.doubleValue
}
```

列表中的绑定甚至可以后跟一个`where`子句，将另一个条件放到一行当中。整个列表可以从一个条件开始，位于单词`let`或`var`前。如下示例来自于我曾编写过的代码（第11章将会对此进行介绍）。“末日金字塔”包含4个嵌套条件：

```
override func observeValueForKeyPath(keyPath: String?,
    ofObject object: AnyObject?, change: [String : AnyObject]?,
    context: UnsafeMutablePointer<()>) {
        if keyPath == "readyForDisplay" {
            if let obj = object as? AVPlayerViewController {
```

```
                if let ok = change?[NSKeyValueChangeNewKey] as? Bool {
                    if ok {
                        // ...
                    }
                }
            }
        }
}
```

可以将这 4 个条件组合放到单个列表中：

```
override func observeValueForKeyPath(keyPath: String?,
    ofObject object: AnyObject?, change: [String : AnyObject]?,
    context: UnsafeMutablePointer<()>) {
        if keyPath == "readyForDisplay",
        let obj = object as? AVPlayerViewController,
        let ok = change?[NSKeyValueChangeNewKey] as? Bool where ok {
            // ...
        }
}
```

不过，至于第 2 个版本是否更清晰可读则是个见仁见智的问题了。

在 Swift 2.0 中，你可以通过一系列 guard 语句来表示这个条件链；我觉得下面这种方式是最好的：

```
override func observeValueForKeyPath(keyPath: String?,
    ofObject object: AnyObject?, change: [String : AnyObject]?,
    context: UnsafeMutablePointer<()>) {
        guard keyPath == "readyForDisplay" else {return}
        guard let obj = object as? AVPlayerViewController else {return}
        guard let ok = change?[NSKeyValueChangeNewKey] as? Bool else {return}
        guard ok else {return}
        // ...
}
```

### 3. Switch 语句

Switch 语句是一种更为简洁的 `if...else if...else` 结构编写方式。在 C 及 Objective-C 中，switch 语句有个隐藏的陷阱；Swift 消除了这个陷阱，并增加了功能与灵活性。这样，switch 语句在 Swift 中得到了广泛的应用（但在我所编写的 Objective-C 代码中用得却很少）。

在 switch 语句中，条件位于不同可能值与单个值的比较（叫作标记）中，这叫作 case。case 比较是按照顺序执行的。如果某个 case 比较成功，那么 case 的代码就会执行，整个 switch 语句将会退出。示例 5-2 展示了其模式，根据需要可以有多个 case，`default` case 则可以省略（有一些限制，稍后将会介绍）。

示例 5-2：Swift switch 语句

```
switch tag {
case pattern1:
    statements
case pattern2:
    statements
default:
    statements
}
```

下面是个实际的示例：

```
    switch i {
    case 1:
        print("You have 1 thingy!")
    case 2:
        print("You have 2 thingies!")
    default:
        print("You have \(i) thingies!")
    }
```

在上述代码中，变量 i 作为标记。i 的值首先会与 1 进行比较。如果 i 为 1，那么该 case 的代码就会执行，然后 switch 语句退出。如果 i 不为 1，那么它会与 2 进行比较。如果 i 为 2，那么该 case 的代码就会执行，然后 switch 语句退出。如果 i 值与所有 case 值都不相等，那么 default case 的代码就会执行。

在 Swift 中，switch 语句必须是完备的。这意味着标记的每个可能值都必须要被 case 覆盖到。如果违背了这一原则，编译器就会报错。如果一个值只有有限的可能，这个原则就会显得很直观；这通常来说是枚举，它本身只有为数不多的 case 作为可能值。不过在上述示例中，标记是个 Int，而 Int 的可能值是很大的，因此 case 也会有很多。这样就必须要有一个扫尾的 case，不过你不必显式提供。常见的扫尾 case 是使用一个 default case。

每个 case 的代码都可以有多行，不必像上述示例那样只有单行代码。不过，每个 case 至少要包含一行代码；Swift switch case 不允许没有代码。case 代码的第 1 行（也只有这行）可以与 case 位于同一行；这样就可以像下面这样改写上述示例了：

```
    switch i {
    case 1: print("You have 1 thingy!")
    case 2: print("You have 2 thingies!")
    default: print("You have \(i) thingies!")
    }
```

最少的单行 case 代码只包含关键字 break；在这种情况下，break 表示一个占位符，什么都不会做。“很多时候，switch 语句会包含一个 default（或其他扫尾 case），它只

包含了关键字 break”；这样就可以穷尽标记的所有可能值，不过如果值与哪一个 case 都不匹配，那就什么都不会做。

现在来看看标记值与 case 值的比较。在上述示例中，这种比较类似于相等比较（==）；不过还有其他情况存在。在 Swift 中，case 值实际上是一个叫作模式的特殊表达式，该模式会通过“隐秘的模式匹配运算符 ~=”与标记值进行比较。你对构建模式的语法认识越深刻，case 值与 switch 语句就会越强大。

模式还可以包含一个下划线（_）来表示所有其他值。实际上，下划线 case 是“扫尾 case”的一种替代形式：

```
switch i {
case 1:
    print("You have 1 thingy!")
case _:
    print("You have many thingies!")
}
```

模式可以包含局部变量名的声明（无条件绑定）来表示所有值，并将实际值作为该局部变量的值。这实际上是“扫尾 case”的另一种替代方案：

```
switch i {
case 1:
    print("You have 1 thingy!")
case let n:
    print("You have \(n) thingies!")
}
```

如果标记是个 Comparable，那么 case 还可以包含 Range；比较时会向 Range 发送 contains 消息：

```
switch i {
case 1:
    print("You have 1 thingy!")
case 2...10:
    print("You have \(i) thingies!")
default:
    print("You have more thingies than I can count!")
}
```

如果标记是个 Optional，那么 case 可以将其与 nil 进行比较。这样，安全展开 Optional 的一种方式就是先将其与 nil 进行比较，并在随后的 case 中将其展开，因为如果 nil 比较通过，那么流程永远也不会进入到展开 case。在如下示例中，i 是个包装了 Int 的 Optional：

```
switch i {
case nil: break
```

```
default:
    switch i! {
    case 1:
        print("You have 1 thingy!")
    case let n:
        print("You have \(n) thingies!")
    }
}
```

不过，这看起来有点笨拙，因此 Swift 2.0 提供了一个新的语法：将 ? 追加到 case 模式上可以安全地展开一个 Optional 标记。这样，我们就可以像下面这样重写该示例：

```
switch i {
case 1?:
    print("You have 1 thingy!")
case let n?:
    print("You have \(n) thingies!")
case nil: break
}
```

如果标记是个 Bool 值，那么 case 就可以将其与条件进行比较了。通过巧妙的使用，你可以通过 case 测试任何条件：将 `true` 作为标记！这样，switch 语句就变成了扩展的 `if...else if` 结构的替代者。如下示例来自于我所编写的代码，我本可以使用 `if...else if`，但每个 case 只有一行代码，因此使用 switch 语句会更加整洁一些：

```
func positionForBar(bar: UIBarPositioning) -> UIBarPosition {
    switch true {
    case bar === self.navbar:  return .TopAttached
    case bar === self.toolbar: return .Bottom
    default:                   return .Any
    }
}
```

模式还可以包含 `where` 子句来添加条件，从而限制 case 的真值。它常常会与绑定搭配使用，但这并非强制要求；条件可以引用绑定中声明的变量：

```
switch i {
case let j where j < 0:
    print("i is negative")
case let j where j > 0:
    print("i is positive")
case 0:
    print("i is 0")
default:break
}
```

模式可以通过 `is` 运算符来判断标记的类型。在如下示例中，假设有个 Dog 类及其 NoisyDog 子类，`d` 的类型为 Dog：

```
switch d {
```

```
case is NoisyDog:
    print("You have a noisy dog!")
case _:
    print("You have a dog.")
}
```

模式可以通过 as（不是 as?）运算符进行类型转换。一般来说，你会将其与声明局部变量的绑定搭配使用；虽然使用了无条件的 as，但值的类型转换却是有条件的，如果转换成功，那么局部变量就会将转换后的值带入 case 代码中。假设 Dog 实现了 bark，NoisyDog 实现了 beQuiet：

```
switch d {
case let nd as NoisyDog:
    nd.beQuiet()
case let d:
    d.bark()
}
```

在与特定的值进行比较时，你还可以使用 as（不是 as!）运算符根据情况对标记进行向下类型转换（可能还会展开）；在如下示例中，i 可能是个 AnyObject，也可能是个包装了 AnyObject 的 Optional：

```
switch i {
case 0 as Int:
    print("It is 0")
default:break
}
```

你可以将标记表示为元组，同时将相应的比较也包装到元组中，这样就可以同时进行多个比较了。只有当比较元组与相应的标记元组中的每一项比较都通过，这个 case 才算通过。在如下示例中，我们从一个类型为 [String:AnyObject] 的字典 d 开始。借助元组，我们可以安全地抽取并转换两个值：

```
switch (d["size"], d["desc"]) {
case let (size as Int, desc as String):
    print("You have size \(size) and it is \(desc)")
default:break
}
```

如果标记是个枚举，那么 case 就可以是枚举的 case。这样，switch 语句就成为处理枚举的绝佳方式，下面是枚举声明：

```
enum Filter {
    case Albums
    case Playlists
    case Podcasts
    case Books
}
```

下面是个 switch 语句，其中的标记 type 是个 Filter：

```
switch type {
case .Albums:
    print("Albums")
case .Playlists:
    print("Playlists")
case .Podcasts:
    print("Podcasts")
case .Books:
    print("Books")
}
```

这里不需要“扫尾”case，因为代码已经穷尽了所有 case。（在该示例中，case 名前的点是必不可少的。不过，如果代码位于枚举声明中，那么点就可以省略。）

Switch 语句提供了从枚举 case 中抽取出关联值的方式。回忆一下第 4 章介绍的这个枚举：

```
enum Error {
    case Number(Int)
    case Message(String)
    case Fatal
}
```

要想从 Error 中抽取出错误号（其 case 是 .Number），或从 Error 中抽取出消息字符串（其 case 是 .Message），我可以使用 switch 语句。回忆一下，关联值实际上是个元组。匹配 case 名后的模式元组会应用到关联值上。如果模式是个绑定变量，那么它会捕获到关联值。let（或 var）可以位于圆括号中，或在 case 关键字后；如下代码演示了这两种情况：

```
switch err {
case .Number(let theNumber):
    print("It is a .Number: \(theNumber)")
case let .Message(theMessage):
    print("It is a .Message: \(theMessage)")
case .Fatal:
    print("It is a .Fatal")
}
```

如果 let（或 var）位于 case 关键字之后，那就可以添加一个 where 子句：

```
switch err {
case let .Number(n) where n > 0:
    print("It's a positive error number \(n)")
case let .Number(n) where n < 0:
    print("It's a negative error number \(n)")
case .Number(0):
    print("It's a zero error number")
default:break
}
```

如果不想抽取出错误号，只想进行匹配，那就可以在圆括号中使用另外一种模式：

```
switch err {
case .Number(1..<Int.max):
    print("It's a positive error number")
case .Number(Int.min...(-1)):
    print("It's a negative error number")
case .Number(0):
    print("It's a zero error number")
default:break
}
```

该模式提供了另外一种处理 Optional 标记的方式。正如第 4 章所述，Optional 实际上是个枚举。它有两种 case，分别是 .None 与 .Some，其中被包装的值是与 .Some case 相关联的值。不过现在我们知道了该如何提取出相关联的值！这样，我们就可以再次重写之前的那个示例，其中 i 是个包装了 Int 的 Optional：

```
switch i {
case .None: break
case .Some(1):
    print("You have 1 thingy!")
case .Some(let n):
    print("You have \(n) thingies!")
}
```

在 Swift 2.0 中，我们可以通过轻量级的 if case 结构在条件中使用与 switch 语句的 case 相同模式的语法。Switch case 模式类似于之前介绍的标记，if case 模式后面则会跟着一个等号和标记。实际上，这对于通过单个条件绑定来从枚举中提取出关联值是非常有用的（下面的 err 是 Error 枚举）。

```
if case let .Number(n) = err {
    print("The error number is \(n)")
}
```

甚至还可以在 switch case 中附加一个 where 子句：

```
if case let .Number(n) = err where n < 0 {
    print("The negative error number is \(n)")
}
```

要想将 case 测试组合起来（使用隐式的逻辑或），可以用逗号将其分隔开：

```
switch i {
case 1,3,5,7,9:
    print("You have a small odd number of thingies.")
case 2,4,6,8,10:
    print("You have a small even number of thingies.")
default:
    print("You have too many thingies for me to count.")
}
```

在该示例中，i 是个 AnyObject：

```
switch i {
case is Int, is Double:
    print("It's some kind of number.")
default:
    print("I don't know what it is.")
}
```

不过，你不能使用逗号组合声明绑定变量的模式，因为在这种情况下，对变量的赋值是不明确的。

组合 case 的另一种方式是通过 fallthrough 语句从一个 case 落到下一个 case 上。虽然一个 case 执行完一些代码后落到下一个 case 上是合法的，但很多时候一个 case 只会包含一个 fallthrough 语句：

```
switch pep {
case "Manny": fallthrough
case "Moe": fallthrough
case "Jack":
    print("\(pep) is a Pep boy")
default:
    print("I don't know who \(pep) is")
}
```

注意，fallthrough 会跳过下一个 case 的测试；它会直接开始执行下一个 case 的代码。因此，下一个 case 不能声明任何绑定变量，因为无法对变量赋值。

### 4. 条件求值

在确定使用什么值时会出现一个有意思的问题，比如，将什么值赋给变量。这看起来像是分支结构的用武之地。当然，你可以先声明变量但不对其初始化，并在随后的分支结构中设置其值。不过，使用分支结构作为变量值会更好一些。比如，下面是个变量赋值语句，其中等号后面会直接跟着一个分支结构（代码无法编译通过）：

```
let title = switch type { // compile error
case .Albums:
    "Albums"
case .Playlists:
    "Playlists"
case .Podcasts:
    "Podcasts"
case .Books:
    "Books"
}
```

有些语言允许这么做，但 Swift 不行。不过可以采取一个易于实现的变通办法：使用定义与调用匿名函数：

```
let title : String = {
    switch type {
    case .Albums:
        return "Albums"
    case .Playlists:
        return "Playlists"
    case .Podcasts:
        return "Podcasts"
    case .Books:
        return "Books"
    }
}()
```

有时，值通过一个二路条件才能确定下来，Swift提供了类似于C语言的三元运算符（:?）。其模式如下所示：

```
condition ? exp1 : exp2
```

如果条件为true，那么表达式“*exp1*”会求值并将值作为结果；否则，表达式“*exp2*”会求值并将值作为结果。这样，在赋值时就可以使用三元运算符了，模式如下所示：

```
let myVariable = condition ? exp1 : exp2
```

myVariable的初始值取决于条件的真值情况。我在代码中大量使用了三元运算符，比如：

```
cell.accessoryType =
    ix.row == self.currow ? .Checkmark : .DisclosureIndicator
```

上下文不一定非得是个赋值；如下示例会确定将什么值作为函数实参传递进去：

```
CGContextSetFillColorWithColor(
    context, self.hilite ? purple.CGColor : beige.CGColor)
```

在现代Objective-C所使用的C版本中，有一种压缩形式的三元运算符，它可以将值与nil进行比较。如果为nil，那么你可以提供一个替代值。如果不为nil，那么使用的就是被测试的值本身。在Swift中，类似的操作涉及对Optional的判断：如果被测试的Optional为nil，那就会使用替代值；否则会展开Optional，并使用被包装的值。Swift提供了这样一个运算符：?? 运算符（叫作nil合并运算符）。

回忆一下第4章的示例，其中arr是个Swift的Optional String数组，我对其进行了转换，使得转换后的结果可以以NSArray的形式传递给Objective-C：

```
let arr2 : [AnyObject] =
    arr.map{if $0 == nil {return NSNull()} else {return $0!}}
```

可以通过三元运算符以更加整洁的方式完成相同的事情：

```
let arr2 = arr.map { $0 != nil ? $0! : NSNull() }
```

nil 合并运算符甚至更加整洁：

```
let arr2 = arr.map{ $0 ?? NSNull() }
```

可以将使用 ?? 的表达式链接起来：

```
let someNumber = i1 as? Int ?? i2 as? Int ?? 0
```

上述代码尝试将 i1 类型转换为 Int 并使用该 Int。如果失败，那么它会尝试将 i2 类型转换为 Int 并使用该 Int。如果这也失败，那么它就会放弃并使用 0。

## 5.1.2 循环

循环的目的在于重复一个代码块的执行，并且在每次迭代时会有一些差别。这种差别通常还作为循环停止的信号。Swift 提供了两种基本的循环结构：while 循环与 for 循环。

### 1. while 循环

while 循环有两种形式，如示例 5-3 所示。

示例 5-3：Swift while 循环

```
while condition {
    statements
}
repeat {
    statements
} while condition
```

这两种形式之间的主要差别在于测试的次数。在第 2 种形式中，代码块执行后才会测试条件，这意味着代码块至少会被执行一次。

一般来说，块中的代码会修改一些东西，这会影响环境与条件，最终让循环结束。如下典型示例来自于我之前所编写的代码（movenda 是个数组）：

```
while self.movenda.count > 0 {
    let p = self.movenda.removeLast()
    // ...
}
```

每次迭代都会从 movenda 中删除一个元素，最终其数量会变为 0，循环也将终止；接下来，执行会进入到右花括号的下一行代码。

while 循环第一种形式的条件可以是个 Optional 的条件绑定。这提供了一种紧凑且安全的方式来展开 Optional，然后循环，直到 Optional 为 nil；包含了展开的 Optional 的局部变量位于花括号的作用域中。这样，我们就能以更加简洁的方式改写代码：

```
while let p = self.movenda.popLast() {
    // ...
}
```

在我的代码中，while 循环的另一种常见用法是沿着层次结构向上或向下遍历。在如下示例中，首先从表视图单元的一个子视图（textField）开始，我想知道它属于哪个表视图单元。因此，我会沿着视图层次向上遍历，比较每个父视图，直到遇到了表视图单元：

```
var v : UIView = textField
repeat { v = v.superview! } while !(v is UITableViewCell)
```

上述代码执行完毕后，v 就是我们要找的表视图单元。不过，上述代码有些危险：如果在到达视图层次结构的最顶层时没有遇到 UITableViewCell（也就是说 superview 为 nil 的视图），那么程序就会崩溃。下面以一种安全的方式来编写同样的代码：

```
var v : UIView = textField
while let vv = v.superview where !(vv is UITableViewCell) {v = vv}
if let c = v.superview as? UITableViewCell { // ...
```

类似于 if case 结构，在 while case 中也可以使用 switch case 模式。在下面这个想象出来的示例中有一个由各种 Error 枚举所构成的数组：

```
let arr : [Error] = [
    .Message("ouch"), .Message("yipes"), .Number(10), .Number(-1), .Fatal
]
```

我们可以从数组起始位置开始提取出与 .Message 关联的字符串值，如以下代码所示：

```
var i = 0
while case let .Message(message) = arr[i++] {
    print(message) // "ouch", then "yipes"; then the loop stops
}
```

## 2. for 循环

Swift for 循环有两种形式，如示例 5-4 所示。

示例 5-4：Swift for 循环

```
for variable in sequence {
    statements
}

for before-all; condition; after-each {
    statements
}
```

第 1 种形式（for...in 结构）类似于 Objective-C 的 for...in 结构。在 Objective-C 中，

如果一个类遵循了NSFastEnumeration协议，那么就可以使用该for循环语法。在Swift中，如果一个类型使用了SequenceType协议，那么就可以使用该for循环语法。

在`for...in`结构中，变量会在每次迭代中隐式通过`let`进行声明；因此在默认情况下它是不可变的。（如果需要对块中的变量进行赋值，那么请写成`for var`。）该变量对于块来说也是局部变量。在每次迭代中，序列中连续的元素用于初始化变量，它位于块作用域中。你会经常使用这种for循环形式，因为在Swift中，创建序列是非常轻松的事情。在C中，遍历数字1到15的方式是使用第2种形式的for循环，在Swift中当然也可以这么做：

```
for var i = 1; i < 6; i++ {
    print(i)
}
```

不过在Swift中，你可以很方便地创建数字1到5的序列（是个Range），一般来说你会这么做：

```
for i in 1...5 {
    print(i)
}
```

SequenceType有个`generate`方法，它会生成一个“迭代器”对象，这个迭代器对象本身有个mutating的`next`方法，该方法会返回序列中的下一个对象，并被包装到Optional中，如果没有下一个对象，那么它会返回`nil`。在底层，`for...in`实际上是一种while循环：

```
var g = (1...5).generate()
while let i = g.next() {
    print(i)
}
```

有时，你会发现像上面这样显式使用while循环会使得循环更易于控制和定制。

序列通常是个已经存在的值。它可以是字符序列，这样变量值就是连续的Character。它可以是Array，这样变量值就是连续的数组元素。它可以是字典，这样变量值就是键值对元组，你可以将变量表示为两个名字的元组，从而可以捕获到它们。之前的章节中已经介绍了一些示例。

正如第4章所述，你可能会遇到来自于Objective-C的数组，其元素需要从AnyObject进行向下类型转换。我们常常会将其作为序列规范的一部分：

```
let p = Pep()
for boy in p.boys() as! [String] {
    // ...
}
```

序列的 enumerate 方法会生成一个元组序列，并在原始序列的每个元素前加上其索引号：

```
for (i,v) in self.tiles.enumerate() {
    v.center = self.centers[i]
}
```

如果想要跳过序列的某些值，在 Swift 2.0 中可以附加一个 where 子句：

```
for i in 0...10 where i % 2 == 0 {
    print(i) // 0, 2, 4, 6, 8, 10
}
```

就像 if case 与 while case 一样，还有一个 for case。回到之前 Error 枚举数组那个示例：

```
let arr : [Error] = [
    .Message("ouch"), .Message("yipes"), .Number(10), .Number(-1), .Fatal
]
```

遍历整个数组，只提取出与 .Number 关联的值：

```
for case let .Number(i) in arr {
    print(i) // 10, then -1
}
```

序列还有实例方法，如 map、filter 和 reverse；如下示例倒序输出了偶数数字：

```
let range = (0...10).reverse().filter{$0 % 2 == 0}
for i in range {
    print(i) // 10, 8, 6, 4, 2, 0
}
```

另一种方式是通过调用 stride 方法来生成序列。它是 Strideable 协议的一个实例方法，并且被数字类型与可以增加及减少的所有类型所使用。它拥有两种形式：

- stride(through:by:)
- stride(to:by:)

使用哪种形式取决于你是否希望序列包含最终值。by: 参数可以是负数：

```
for 10.stride(through: 0, by: -2) {
    print(i) // 10, 8, 6, 4, 2, 0
}
```

可以通过全局的 zip 函数同时遍历两个序列，它接收两个序列，并生成一个 Zip2 结构体，其本身也是个序列。每次遍历 Zip2 得到的值都是原来的两个序列中相应元素所构成的元组；如果原来的一个序列比另一个长，那么额外的元素会被忽略：

```
let arr1 = ["CA", "MD", "NY", "AZ"]
let arr2 = ["California", "Maryland", "New York"]
var d = [String:String]()
for (s1,s2) in zip(arr1,arr2) {
    d[s1] = s2
} // now d is ["MD": "Maryland", "NY": "New York", "CA": "California"]
```

第2种形式的for循环来源于C的循环（参见示例5-4），它主要用于增加或减少计数器值。*before-all* 语句会在进入for循环时执行一次，它通常用于计数器的初始化。接下来会测试条件，如果为true，那么代码块就会执行；条件通常用来测试计数器是否到达了某个限值。接下来会执行 *after-each* 语句，它通常用于增加或减少计数器值；接下来会再次测试条件。因此，要想使用整数值1、2、3、4与5执行代码块，这种形式的for循环的标准做法如下所示：

```
var i : Int
for i = 1; i < 6; i++ {
    print(i)
}
```

要想将计数器的作用域限制到花括号中，请在 *before-all* 语句中声明它：

```
for var i = 1; i < 6; i++ {
    print(i)
}
```

不过，没有规则说这种形式的for循环就一定是与计数或值增加相关的。回忆一下之前介绍的关于while循环的示例，我们遍历了视图层次，查找某个表视图单元：

```
var v : UIView = textField
repeat { v = v.superview! } while !(v is UITableViewCell)
```

下面是另一种做法，使用一个for循环，其代码块是空的：

```
var v : UIView
for v = textField; !(v is UITableViewCell); v = v.superview! {}
```

就像C一样，声明中的语句（用分号分隔）可以包含多个代码声明（用逗号分隔）。这是一种表明意图的便捷、优雅的方式。下面这个示例来自于我之前编写的代码，我在 *before-all* 语句中声明了两个变量，然后在 *after-each* 语句中修改了它们；当然，完成这个任务不止这一种方法，不过这种方式看起来最简洁：

```
var values = [0.0]
for (var i = 20, direction = 1.0; i < 60; i += 5, direction *= -1) {
    values.append( direction * M_PI / Double(i) )
}
```

## 5.1.3 跳转

虽然分支与循环构成了代码执行的大多数决策流程，但有时它们还不足以表达出所需的逻辑。我们偶尔还需要完全终止代码的执行，并跳转到其他地方。

从一个地方跳转到另一个地方最常见的方式是使用 `goto` 命令，这在早期编程语言中是非常普遍的，不过现在却被认为是“有害的”。Swift 并未提供 `goto` 命令，不过它提供了一些跳转控制方式，在实际情况下可以涵盖所有的情况。Swift 的跳转模式都是以从当前代码流中提前退出的形式而存在的。

你已经对最重要的一种提前退出模式很熟悉了，那就是 `return`，它会立即终止当前的函数，并在函数调用处继续执行。这样，我们可以认为 `return` 就是一种跳转形式。

### 1. 短路与标签

Swift 提供了几种方式来短路分支与循环结构流：

`fallthrough`

Switch case 中的 `fallthrough` 语句会终止当前 case 代码的执行，并立刻开始下一个 case 代码的执行。必须要有下一个 case，否则编译器会报错。

`continue`

循环结构中的 `continue` 语句会终止当前迭代的执行，然后继续下一次迭代：

- 在 while 循环中，`continue` 表示立刻执行条件测试。
- 在第 1 种 for 循环中（`for...in`），`continue` 表示如果有下一次迭代，那么它会立刻进入下一次迭代中。
- 在第 2 种 for 循环中（C 语言中的 for 循环），`continue` 表示立刻执行 *after-each* 语句和条件测试。

`break`

`break` 语句会终止当前结构：

- 在循环中，`break` 会完全终止循环。
- 在 switch case 代码中，`break` 会终止整个 switch 结构。

如果结构是嵌套的，那么你可能就需要指定想要 `continue` 或 `break` 哪一个结构。因此，Swift 支持在 do 块、if 结构、switch 语句、while 循环或 for 循环前放置一个标签。标签可以是任何名字，后跟一个冒号。接下来可以在任意深度结构中的 `continue` 或 `break` 后面加上标签名，指定你所引用的结构。

如下示例用于说明语法。首先，我不使用标签嵌套了两个 for 循环：

```
for i in 1...5 {
    for j in 1...5 {
        print("\(i), \(j);")
        break
    }
}
// 1, 1; 2, 1; 3, 1; 4, 1; 5, 1;
```

从输出可以看到，上述代码会在一次迭代后终止内部循环，而外部循环则会正常执行 5 次迭代。但如果想要终止整个嵌套结构该怎么办呢？解决办法就是使用标签：

```
outer: for i in 1...5 {
    for j in 1...5 {
        print("\(i), \(j);")
        break outer
    }
}
// 1, 1;
```

在 Swift 2.0 中，你可以在单词 `if` 前放置一个标签，还可以在 `if` 或 `else` 块的代码中将 `break` 与标签名搭配使用；与之类似，你可以在单词 `do` 之前放置一个标签，并在 `do` 块中将 `break` 与标签名搭配使用。借助这些举措，我们可以认为 Swift 的短路功能是特性完备的。

### 2. 抛出与捕获错误

有时会出现无法达成一致的情况：我们所进入的整个操作失败了。接下来就需要终止当前作用域，这可能是当前函数，甚至还可能是调用它的函数等，然后退出到能接收到这个失败的地方，并通过其他方式继续进行。

出于这个目的，Swift 2.0 提供了一种抛出与捕获错误的机制。为了保持其一以贯之的安全性与清晰性，Swift 对这种机制的使用施加了一些严格的条件，编译器会确保你遵守了这些条件。

从这个意义上来说，错误是一种消息，指出了出错的地方。作为错误处理过程的一部分，该消息会沿着作用域与函数调用向上传递，如果需要，从失败中恢复的代码会读取该消息，然后决定该如何继续。在 Swift 中，错误一定是使用了 ErrorType 协议的类型的对象，它只有两点要求：一个 String 类型的 `_domain` 属性，以及一个 Int 类型的 `_code` 属性。实际上，它指的是如下两者之一：

*NSError*

NSError 是 Cocoa 中用于与问题本质通信的类。如果对 Cocoa 方法的调用导致了失

败，那么 Cocoa 会向你发送一个 NSError 实例。还可以通过调用其指定初始化器 `init(domain:code:userInfo:)` 来创建自己的 NSError 实例。

*使用了 ErrorType 的 Swift 类型*

如果一个类型使用了 ErrorType 协议，那么就可以将其作为错误对象；在背后，协议的要求会神奇般地得到满足。一般来说，该类型是个枚举，它会通过其 case 来与消息通信：不同的 case 会区分不同类型的失败，也许一些原始值或关联值还会持有更多的信息。

有两个错误机制阶段需要考虑：抛出错误与捕获错误。抛出错误会终止当前的执行路径，并将错误对象传递给错误处理机制。捕获错误会接收错误对象并对其进行响应，在捕获点后执行路径会继续。实际上，我们会从抛出点跳转到捕获点。

要想抛出错误，请使用关键字 `throw` 并后跟错误对象。这会导致当前代码块终止执行，同时错误处理机制会介入。为了确保 `throw` 命令的使用能够做到前后一致，Swift 应用了一条规则，你只能在如下两个地方使用 `throw`：

*在 `do...catch` 结构的 do 块中*

`do...catch` 结构至少包含了两个块，即 do 块与 catch 块。该结构的要点在于 catch 块可以接收 do 块所抛出的任何错误。因此，我们就可以前后一致地处理这种错误，错误可以被捕获到。稍后将会更加详尽地介绍 `do...catch` 结构。

*在标记了 `throws` 的函数中*

如果不在 `do...catch` 结构的 do 块中抛出错误，或在 do 块中抛出了错误，但 catch 块没有将其捕获，那么错误消息就会跳出当前函数。这样就需要依赖于其他函数了，即调用该函数的函数，或更外层的函数，以此类推一直沿着调用栈向上，通过这种方式来捕获错误。要想告知任何调用者（以及编译器）错误发生了，函数需要在其声明中加上关键字 `throws`。

要想捕获错误，请使用 `do...catch` 结构。从 do 块中抛出的错误可以被与之相伴的 catch 块所捕获。`do...catch` 结构的模式类似于示例 5-5。

示例 5-5：Swift do...catch 结构

```
do {
    statements // a throw can happen here
} catch errortype {
    statements
} catch {
    statements
}
```

一个 do 块后面可以跟着多个 catch 块。Catch 块类似于 switch 语句中的 case，通常也都

具有同样的逻辑：首先，你会有专门的 catch 块，其中每一个都用于处理可能会出现的一部分错误；最后会有一个一般性的 catch 块，它作为默认值，处理其他专门的 catch 块所没有捕获到的错误。

实际上，catch 块所用的捕获指定错误的语法就是 switch 语句中的 case 所用的模式语法！可以将其看作一个 switch 语句，标记就是错误对象。接下来，错误对象与特定 catch 块的匹配就好像使用的是 case 而非 catch 一样。通常，如果 ErrorType 是个枚举，那么专门的 catch 块至少会声明它所捕获到的枚举，也许还有该枚举的 case；它可以通过绑定来捕获到该枚举或与其关联的类型；还可以通过 where 子句来进一步限定可能性。

为了说明问题，我首先定义两个错误：

```
enum MyFirstError : ErrorType {
    case FirstMinorMistake
    case FirstMajorMistake
    case FirstFatalMistake
}
enum MySecondError : ErrorType {
    case SecondMinorMistake(i:Int)
    case SecondMajorMistake(s:String)
    case SecondFatalMistake
}
```

下面的 do...catch 结构用于说明在不同的 catch 块中捕获不同错误的方式：

```
do {
    // throw can happen here
} catch MyFirstError.FirstMinorMistake {
    // catches MyFirstError.FirstMinorMistake
} catch let err as MyFirstError {
    // catches all other cases of MyFirstError
} catch MySecondError.SecondMinorMistake(let i) where i < 0 {
    // catches e.g. MySecondError.SecondMinorMistake(i:-3)
} catch {
    // catches everything else
}
```

在使用了伴随模式的 catch 块中，你可以在模式中决定捕获关于错误的何种信息。比如，如果希望将错误本身当作变量传递到 catch 块中，那就需要在模式中进行绑定。在没有使用伴随模式的 catch 块中，错误对象会以一个名为 error 的变量形式进入块中。

如果函数中的代码使用了 throw，同时代码又不处于拥有“收尾”catch 块的 do 块中，那么该函数本身就要标记为 throws，因为如果没有捕获到每一个可能的错误，同时代码又抛出了错误，那么该错误就会离开所在的函数。语法要求关键字 throws 要紧跟参数列表之后（如果有箭头运算符，则还要位于它之前）。比如：

```
enum NotLongEnough : ErrorType {
    case ISaidLongIMeantLong
}
func giveMeALongString(s:String) throws {
    if s.characters.count < 5 {
        throw NotLongEnough.ISaidLongIMeantLong
    }
    print("thanks for the string")
}
```

向函数声明添加的 throws 创建了一个新的函数类型。giveMeALongString 的类型不是 (String) -> ()，而是 (String) throws -> ()。如果一个函数接收另一个会 throw 的函数作为参数，那么其参数类型就需要进行相应的指定：

```
func receiveThrower(f:(String) throws -> ()) {
    // ...
}
```

现在，这个函数可以作为 giveMeALongString 的参数进行调用了：

```
func callReceiveThrower() {
    receiveThrower(giveMeALongString)
}
```

如果必要，匿名函数可以在正常的函数声明中使用关键字 throws。不过，如果匿名函数的类型可以推断出来，那么这么做就没必要了：

```
func callReceiveThrower() {
    receiveThrower {
        s in
        if s.characters.count < 5 {
            throw NotLongEnough.ISaidLongIMeantLong
        }
        print("thanks for the string")
    }
}
```

Swift 对 throws 函数的调用者也有要求：调用者必须要在调用前使用关键字 try。该关键字告诉程序员和编译器，我们知道这个函数会 throw。它还有这样一个要求：调用必须出现在 throw 为合法的情况下！使用 try 调用的函数会 throw，因此 try 的含义就类似于 throw：你必须要在 do...catch 结构的 do 块中或标记为 throws 的函数中使用它。

比如：

```
func stringTest() {
    do {
        try giveMeALongString("is this long enough for you?")
    } catch {
        print("I guess it wasn't long enough: \(error)")
```

```
        }
    }
```

不过，如果你非常确定会 throw 的函数肯定不会 throw，那么你就可以使用关键字 `try!` 而非 `try` 来调用它。这么做会简化使用：你可以在任何地方使用 `try!` 而无须捕获可能的 throw。不过请注意：如果你做错了，当程序运行时这个函数抛出了异常，那么程序就会崩溃，因为你允许错误继续而没有捕获，一直到调用链的顶部。

因此，下面这种做法是合法的，但却是危险的：

```
func stringTest() {
    try! giveMeALongString("okay")
}
```

介于 `try` 与 `try!` 之间的是 `try?`。它拥有 `try!` 的优点，你可以在任何地方使用它而无须捕获可能的异常。此外，如果真的抛出了异常，那么程序并不会崩溃；相反，它会返回 `nil`。因此，`try?` 在表达式返回一个值的情况下特别有用。如果没有抛出异常，那么它会将值包装到一个 Optional 中。一般来说，你可以通过条件绑定在同一行上安全地展开这个 Optional。稍后将会介绍一个示例。

如果一个函数接收一个会抛出异常的函数作为参数，然后调用该函数（使用 `try`），但结果没有抛出异常，那么我们可以将该函数本身标记为 `rethrows` 而非 `throws`。二者的差别在于当调用一个 `rethrows` 函数时，调用者可以传递一个不抛出异常的函数作为参数。这样，就不必对调用使用 `try` 了（调用函数也无须标记为 `throws`）：

```
func receiveThrower(f:(String) throws -> ()) rethrows {
    try f("ok?")
}
func callReceiveThrower() { // no throws needed
    receiveThrower { // no try needed
        s in
        print("thanks for the string!")
    }
}
```

下面来介绍一下 Swift 的错误处理机制与 Cocoa 和 Objective-C 之间的关系。常见的 Cocoa 模式是方法会通过返回 `nil` 来表示失败，并且接收一个 `NSError**` 参数作为与方法外部结果调用者之间通信的方式。Swift 中该参数类型为 NSErrorPointer，它是一个指向包装了 NSError 的 Optional 的指针。比如，NSString 在 Objective-C 中有一个初始化器声明，如下所示：

```
- (instancetype)initWithContentsOfFile:(NSString *)path
    encoding:(NSStringEncoding)enc
    error:(NSError **)error;
```

在 Swift 2.0 之前，该声明对应的 Swift 代码如下所示：

```
convenience init?(contentsOfFile path: String,
    encoding enc: UInt,
    error: NSErrorPointer)
```

你可以将包装了 NSError 的一个 Optional 的地址作为最后一个参数传递给它：

```
var err : NSError?
let s = String(contentsOfFile: f, encoding: NSUTF8StringEncoding, error: &err)
```

调用完毕后，s 要么是个 String（包装在一个 Optional 中），要么是个 nil。如果为 nil，那么调用就失败了，你可以检查 err，系统会设置其值来存储失败的原因。

不过在 Swift 2.0 中，该 Objective-C 方法会被自动进行类型转换，从而利用错误处理机制。error: 参数已经从该声明的 Swift 版本中被移除了，并且被一个 throws 标记所替代：

```
init(contentsOfFile path: String, encoding enc: NSStringEncoding) throws
```

因此，现在没必要提前声明好 NSError 变量了，也没必要间接地接收 NSError。相反，你只需在 Swift 的控制条件中调用该方法即可：你需要在可能会抛出异常的地方使用 try。结果永远不会为 nil，因此也不会再有包装到 Optional 中的 String 了；它就是个 String，因为如果初始化失败，那么调用会抛出异常，并不会产生任何结果：

```
do {
    let f = // path to some file, maybe
    let s = try String(contentsOfFile: f, encoding: NSUTF8StringEncoding)
    // ... if successful, do something with s ...
} catch {
    print((error as NSError).localizedDescription)
}
```

如果非常确定初始化不会失败，那就可以省略 do...catch 结构，转而使用 try!：

```
let f = // path to some file, maybe
let s = try! String(contentsOfFile: f, encoding: NSUTF8StringEncoding)
```

不过，如果有疑虑，那还可以省略 do...catch 结构并继续安全地使用 try?，在这种情况下返回的值是个 Optional，你可以安全地展开它，如以下代码所示：

```
let f = // path to some file, maybe
if let s = try? String(contentsOfFile: f, encoding: NSUTF8StringEncoding) {
    // ...
}
```

Objective-C NSError 与 Swift ErrorType 是彼此桥接的。这样，在之前的 catch 块中，我

将 error 变量类型转换为了 NSError，并使用 NSError 属性检查它。不过，你不必这么做；相对于将捕获到的错误看作 NSError，你可以将其看作 Swift 枚举。

对于常见的 Cocoa 错误类型，桥接枚举的名字就是 NSError 域的名字，同时将 "Domain" 从其名字中删除。假设文件不存在，调用会抛出异常，我们捕获到了错误。这个 NSError 的域就是 "NSCocoaErrorDomain"。因此，Swift 会将其看作一个 NSCocoaError 枚举。此外，其代码是 260，这在 Objective-C 表示的是 NSFileReadNoSuchFileError，在 Swift 则表示 FileReadNoSuchFileError 枚举。因此，我们可以像下面这样捕获这个错误：

```
do {
    let f = // path to some file, maybe
    let s = try String(contentsOfFile: f, encoding: NSUTF8StringEncoding)
    // ... if successful, do something with s ...
} catch NSCocoaError.FileReadNoSuchFileError {
    print("no such file")
} catch {
    print(error)
}
```

参见 Objective-C 中的 *FoundationError.h* 头文件来了解关于 Cocoa 内建标准错误域的更多信息。

反之亦然。如前所述，采用了 ErrorType 的 Swift 类型会在背后自动实现其要求：特别地，其 _domain 是类型的名字，如果是枚举，那么其 _code 就是 case 的索引值（否则就是 1）。如果在需要 NSError 的地方使用了 ErrorType（或类型转换为 NSError），那么它就会成为 NSError 的 domain 和 code 值。

### 3. Defer

Swift 2.0 新增的 defer 语句的目的是确保某个代码块会在执行路径流经当前作用域时（无论如何流经）一定会执行。

Defer 语句适用于它所出现的作用域，如函数体、while 块、if 结构等。无论在哪里使用 defer，请在其外面使用花括号；当执行路径离开这些花括号时，defer 块就会执行。离开花括号包括到达花括号的最后一行代码，或是本节之前介绍的任何一种形式的提前退出。

为了理解 defer 的作用，请看看如下两个命令：

```
UIApplication.sharedApplication().beginIgnoringInteractionEvents()
```

阻止所用用户触摸动作到达应用的任何视图。

```
UIApplication.sharedApplication().endIgnoringInteractionEvents()
```

恢复用户触摸到达应用视图的功能。

在一些耗时操作的开始停止用户交互，接下来当操作完毕时再恢复交互，特别是在操作过程中，用户轻拍按钮等会导致应用出错的场景下，使用 defer 是非常有用的。因此，有时我们会这样编写一些方法：

```
func doSomethingTimeConsuming() {
    UIApplication.sharedApplication().beginIgnoringInteractionEvents()
    // ... do stuff ...
    UIApplication.sharedApplication().endIgnoringInteractionEvents()
}
```

很不错，如果我们可以保证该函数的执行路径一定会到达最后一行。但如果需要从函数中提前返回呢？参见如下代码：

```
func doSomethingTimeConsuming() {
    UIApplication.sharedApplication().beginIgnoringInteractionEvents()
    // ... do stuff ...
    if somethingHappened {
        return
    }
    // ... do more stuff ...
    UIApplication.sharedApplication().endIgnoringInteractionEvents()
}
```

糟糕！我们犯了一个严重的错误。通过向 doSomethingTimeConsuming 函数提供一个额外的路径，代码可能永远都不会执行到对 endIgnoringInteractionEvents() 的调用。我们可以通过 return 语句离开函数，用户接下来就无法与界面交互了。显然，我们需要在 if 结构中添加另外一个 endIgnoring... 调用，就在 return 语句之前。不过，当继续编写代码时，要记住，如果添加离开函数的其他方式，那就需要对每一种方式都添加另外一个 endIgnoring... 调用，这简直太可怕了！

Defer 语句可以解决这个问题。我们可以通过它指定当离开某个作用域时（无论如何离开）会发生什么事情。代码现在看起来如下所示：

```
func doSomethingTimeConsuming() {
    UIApplication.sharedApplication().beginIgnoringInteractionEvents()
    defer {
        UIApplication.sharedApplication().endIgnoringInteractionEvents()
    }
    // ... do stuff ...
    if somethingHappened {
        return
    }
    // ... do more stuff ...
}
```

Defer 块中的 endIgnoring... 调用会执行，其执行时间并不取决于其出现的位置，而是在 return 语句之前或是在方法的最后一行代码之前，即执行路径离开函数的时刻。Defer 语句表示："最终（尽可能晚地），请执行该代码"。我们确保了关闭用户交互与打开用户交互之间的平衡。大多数 defer 语句的使用方式都是这样的：你通过它平衡一个命令或恢复受到干扰的状态。

如果当前作用域有多个 defer 块挂起，那么它们的调用顺序与其出现的顺序是相反的。实际上，有一个 defer 栈；每个后续的 defer 语句都会将其代码推至栈顶，离开 defer 语句的作用域会将代码弹出来并执行。

### 4. 终止

终止是流程控制的一种极端形式；程序会在执行轨迹中突然停止。实际上，你可以故意让自己的程序崩溃。虽然，很少会这么做，但这却是给出危险信号的一种方式：你其实不想终止，这样一旦终止，那就表示一定出现了你无法解决的问题。

终止的一种方式是通过调用全局函数 fatalError。它接收一个 String 参数，可以向控制台打印一条消息。如下示例之前已经介绍过了：

```
required init?(coder aDecoder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}
```

上述代码表示，执行永远也不会到达这个点。我们并没有实现 init(coder:)，也不希望通过这种方式进行初始化。如果以这种方式初始化，那就说明有问题了，我们需要让程序崩溃，因为程序有严重的 Bug。

包含了 fatalError 调用的初始化器不必初始化任何属性。这是因为 fatalError 是通过 @noreturn 特性声明的，它会让编译器放弃任何上下文的需求。与之类似，如果遇到了 fatalError 调用，那么拥有返回值的函数就不必返回任何值了。

还可以通过调用 assert 函数实现条件式终止。其第 1 个参数是个条件，值为一个 Bool。如果条件为 false，那就会终止；第 2 个参数是个 String 消息，如果终止，它会打印到控制台上。其功能是我们断言条件为 true，如果条件为 false，那么极有可能是程序中出现了 Bug，你想让应用崩溃，这样就可以找出 Bug 并进行修复了。

在默认情况下，assert 只在程序开发时会使用。当程序开发完毕并发布后，你会使用不同的构建开关，告诉编译器忽略 assert。实际上，assert 调用中的条件会被忽略；它

们都会被当作 true 来看待。这意味着你可以放心地将 assert 调用放到代码中。当然，在交付程序时，断言是不应该失败的；会导致其失败的任何 Bug 都应该已经被解决了。

在交付代码时，对断言的禁用是通过一种很有意思的方式执行的。条件参数上会增加一个额外的间接层，这是通过将其声明为 @autoclosure 函数实现的。这意味着，虽然参数实际上不是函数，但编译器会将其包装为一个函数；这样一来，除非必要，否则运行时是不会调用该函数的。在交付代码时，运行时是不会调用该函数的。这种机制避免了代价高昂且不必要的求值：assert 条件测试可能会有边际效应，不过如果程序中关闭了断言，那么测试就不会执行。

此外，Swift 还提供了先决函数。它类似于断言，只不过在交付的程序中它依然是可用的。

## 5. Guard

如果需要跳转，那么你可能会测试一个条件来决定是否跳转。Swift 2.0 为这种情况提供一个特殊的语法：guard 语句。实际上，guard 语句就是个 if 语句，你需要在条件失败时提前退出。其形式如示例 5-6 所示。

示例 5-6：Swift guard 语句

```
guard condition else {
    statements
    exit
}
```

如你所见，guard 语句只包含一个条件和一个 else 块。else 块必须要通过 Swift 所提供的任何一种方式跳出当前作用域，如 return、break、continue、throw 或 fatalError 等，只要确保编译器在条件失败时，执行不会在包含 guard 语句的块中继续即可。

这种架构的优雅结果在于，由于 guard 语句确保了在条件失败时退出，所以编译器就知道，如果没有退出，那么 guard 语句后的条件就是成功的了。这样，guard 语句后条件中的条件绑定就处于作用域中，无须引入嵌套作用域。比如：

```
guard let s = optionalString else {return}
// s is now a String (not an Optional)
```

如前所述，该结构是“末日金字塔”的一个很好的替代方案。它与 try? 搭配起来使用也是非常方便的。假设只有在 String(contentsOfFile:encoding:) 成功时流程才能继续。接下来，我们可以重写之前的示例，如以下代码所示：

```
let f = // path to some file, maybe
guard let s = try? String(contentsOfFile: f, encoding: NSUTF8StringEncoding)
    else {return}
// s is now a String (not an Optional)
```

还有一个guard case结构，逻辑上与if case相反。为了说明，我们再次使用Error枚举：

```
guard case let .Number(n) = err else {return}
// n is now the extracted number
```

注意，guard语句的条件绑定不能使用等号左侧相同作用域中已经声明的名字。如下写法是非法的：

```
let s = // ... some Optional
guard let s = s else {return} // compile error
```

原因在于，与if let和while let不同，guard let并不会为嵌套作用域声明绑定变量；它只会在当前作用域中声明。这样，我们就不能在这里声明s，因为s已经在相同作用域中声明了。

## 5.2 运算符

Swift运算符（如+和>等）并不是语言提供的神奇之物。事实上，它们都是函数；它们是被显式声明和实现的，就像其他函数一样。这也是我在第4章指出的+可以作为reduce调用的最后一个参数进行传递的原因所在；reduce接收一个函数（该函数接收两个参数）并返回一个与第一个参数类型相同的值；+实际上是函数的名字。这还解释了Swift运算符如何针对不同的值类型进行重载的方式。你可以对数字、字符串或数组使用+，每种情况下+的含义都不同，因为名字相同但参数类型不同（签名不同）的两个函数是不同的；根据参数类型，Swift可以确定你调用的是哪个+函数。

这些事实不仅仅是有趣的背后实现细节。它们对于你和代码来说都有实际的含义。你可以重载已有的运算符，并应用到自定义的对象类型上。甚至还可以创建新的运算符！本节将会对此进行介绍。

首先，我们来介绍运算符是如何声明的。显然，要有某种句法形式（这是个计算机科学术语），因为调用运算符函数的方式与通常的函数的方式是不同的。你不会说+(1,2)，而是说1+2。即便如此，第2个表达式中的1和2都是+函数调用的参数。那么，Swift是如何知道+函数使用了这种特殊语法呢？

为了探究问题的答案，我们来看看Swift头文件：

```
infix operator + {
```

```
        associativity left
        precedence 140
    }
```

这是个运算符声明。运算符声明表示这个符号是个运算符，它有多少个参数，关于这些参数存在哪些使用语法。真正重要的地方在于花括号之前的部分：关键字 operator，它前面是运算符类型，这里是 infix，后跟运算符的名字。类型有：

infix
: 该运算符接收两个参数，并且运算符位于两个参数中间。

prefix
: 该运算符接收一个参数，并且运算符位于参数之前。

postfix
: 该运算符接收一个参数，并且运算符位于参数之后。

运算符也是个函数，因此你还需要一个函数声明，表明参数的类型与函数的结果类型。Swift 头文件就是一个示例：

```
func +(lhs: Int, rhs: Int) -> Int
```

这是 Swift 头文件中声明的诸多 + 函数中的一个。特别地，它是两个参数都是 Int 的声明。在这种情况下，结果本身就是个 Int（局部参数名 lhs 与 rhs 并不会影响特殊的调用语法，它表示左侧与右侧）。

运算符声明与相应的函数声明都要位于文件顶部。如果运算符是个 prefix 或 postfix 运算符，那么函数声明就必须要以单词 prefix 或 postfix 开头；默认是 infix，可以省略。

我们可以重写运算符来应用到自定义的对象类型上！下面看个示例，假设有一个装有细菌的瓶子（Vial）：

```
struct Vial {
    var numberOfBacteria : Int
    init(_ n:Int) {
        self.numberOfBacteria = n
    }
}
```

在将两个 Vial 合并起来时，你会得到一个由两个 Vial 中的细菌共同构成的一个 Vial。因此，将两个 Vial 加起来的方式就是将它们中的细菌加到一起：

```
func +(lhs:Vial, rhs:Vial) -> Vial {
    let total = lhs.numberOfBacteria + rhs.numberOfBacteria
    return Vial(total)
}
```

如下代码用于测试新的 + 运算符重写：

```
let v1 = Vial(500_000)
let v2 = Vial(400_000)
let v3 = v1 + v2
print(v3.numberOfBacteria) // 900000
```

对于复合赋值运算符来说，第 1 个参数是被赋值的一方。因此，要想实现这种运算符，必须要将第 1 个参数声明为 inout。下面为 Vial 类实现该运算符：

```
func +=(inout lhs:Vial, rhs:Vial) {
    let total = lhs.numberOfBacteria + rhs.numberOfBacteria
    lhs.numberOfBacteria = total
}
```

下面是测试 += 重写的代码：

```
var v1 = Vial(500_000)
let v2 = Vial(400_000)
v1 += v2
print(v1.numberOfBacteria) // 900000
```

对 Vial 类重写 == 比较运算符也是很有必要的。这需要让 Vial 使用 Equatable 协议，当然，它不会自动使用 Equatable 协议，需要我们来实现：

```
func ==(lhs:Vial, rhs:Vial) -> Bool {
    return lhs.numberOfBacteria == rhs.numberOfBacteria
}
extension Vial:Equatable{}
```

既然 Vial 是个 Equatable，那么它就可以用于 indexOf 这样的方法上了：

```
let v1 = Vial(500_000)
let v2 = Vial(400_000)
let arr = [v1,v2]
let ix = arr.indexOf(v1) // Optional wrapping 0
```

此外，互补的不等运算符 != 也会自动应用到 Vial 上，这是因为它已经根据 == 运算符定义到所有的 Equatable 上了。出于同样的原因，如果对 Vial 重写了 < 并让其使用 Comparable，那么另外 3 个比较运算符也会自动应用上。

接下来实现一个全新的运算符。作为示例，我向 Int 注入一个运算符，它会将第 1 个参数作为底数，将第 2 个参数作为指数。我将 ^^ 作为运算符符号（我本想使用 ^，不过它已经被占用了）。出于简化的目的，我省略了边际情况的错误检查（如指数小于 1 等）：

```
infix operator ^^ {
}
func ^^(lhs:Int, rhs:Int) -> Int {
```

```
    var result = lhs
    for _ in 1..<rhs {result *= lhs}
    return result
}
```

代码就是这些！下面来测试一下：

```
print(2^^2) // 4
print(2^^3) // 8
print(3^^3) // 27
```

在定义运算符时，考虑到运算符与其他包含了运算符的表达式之间的关系，你应该指定优先级与结合性规则。我不打算介绍细节，如果感兴趣可以参考 Swift 手册。手册还列出了可作为自定义运算符名的特殊字符：

```
/ = - + ! * % < > & | ^ ? ~
```

运算符名还可以包含其他很多符号字符（除了其他字母数字的字符），这些字符更难输入；请参考手册了解正式的列表。

# 5.3 隐私性

隐私性（也叫作访问控制）指的是对正常的作用域规则的显式修改。第 1 章曾介绍过下面这个示例：

```
class Dog {
    var name = ""
    private var whatADogSays = "woof"
    func bark() {
        print(self.whatADogSays)
    }
}
```

这里的意图是限制其他对象访问 Dog 属性 `whatADogSays` 的方式。它是个私有属性，主要供 Dog 类自身内部使用：Dog 可以使用 `self.whatADogSays`，不过其他对象甚至都意识不到它的存在。

Swift 提供了 3 级私有性：

`internal`

默认规则为声明都是内部的，这意味着它们对所在模块中的所有文件中的所有代码可见。这正是相同模块中的 Swift 文件能够自动看到彼此顶层内容的原因所在，你无须为此做任何额外的处理工作（这与 C 和 Objective-C 不同，在这两种语言中，除非显式通过 `include` 或 `import` 语句将一个文件展示给别的文件，否则它们之间是看不到彼此的）。

private（比 internal 范围要小）

声明为 private 的内容只在包含它的文件中可见。这个规则可能与你想象的不太一样。在某些语言中，private 表示对对象声明是私有的。在 Swift 中，private 并不是这个意思；如果一个文件包含了两个类，这两个类都声明为了 private，但它们还是可以看到彼此。因此，我们可以将代码划分到多个文件中，遵循一个文件一个类这一通用规则。

public（比 internal 范围要大）

声明为 public 的内容可在模块外可见。另一个模块要先导入这个模块才能看到其中的内容。不过，一旦另一个模块导入了这个模块，那么它还是看不到这个模块中没有显式声明为 public 的内容。如果没有编写过任何模块，那么你可能并不需要将任何东西声明为公开的。如果编写过模块，那就需要将某些东西声明为公开的，否则模块将会毫无作用。

## 5.3.1 Private 声明

通过将对象成员声明为 private，你也就间接指定了该对象会有哪些公开 API。如下示例来自于我所编写的代码：

```
class CancelableTimer: NSObject {
    private var q = dispatch_queue_create("timer",nil)
    private var timer : dispatch_source_t!
    private var firsttime = true
    private var once : Bool
    private var handler : () -> ()
    init(once:Bool, handler:()->()) {
        // ...
    }
    func startWithInterval(interval:Double) {
        // ...
    }
    func cancel() {
        // ...
    }
}
```

初始化器 init(once:handler:)、startWithInterval: 与 cancel 方法没有被标记为 private，它们都是这个类的公开 API。也就是说，你可以随意调用它们。不过，属性都是私有的；其他代码（即该文件外面的代码）都看不到它们，也无法获取其值和设置其值。它们纯粹都是用于该类方法的内部。它们维护着状态，不过这些状态是外部代码所不知道的。

值得注意的是，隐私性只限于当前文件的级别。比如：

```
class Cat {
    private var secretName : String?
}
class Dog {
    func nameCat(cat:Cat) {
        cat.secretName = "Lazybones"
    }
}
```

为何说上述代码是合法的呢？Cat 的 `secretName` 是私有的，那为什么 Dog 可以修改它呢？这是因为，私有性并不是单个对象类型级别的；它是文件级别的。我在同一个文件中定义了 Cat 与 Dog，因此它们可以看到彼此的私有成员。

在实际开发中，我们常常会将每个类定义在自己的文件中。事实上，文件名常常是在里面的类名定义好之后才确定下来的；包含了 ViewController 类声明的文件常常会命名为 *ViewController.swift*。不过，这仅仅是个约定而已，并不强求。在 Swift 中，文件名并没有什么意义，同一个模块中的 Swift 文件能够自动看到其他文件中的内容，不必指定其他文件的名字。文件名只不过是为了程序员的方便而已：在 Xcode 中，我会看到程序文件的列表；因此，当我想要寻找 ViewController 类声明时，通过文件名会很方便，我可以查找 *ViewController.swift* 文件。

将代码划分到不同文件中的真实原因在于确保私有性能够起作用。比如，我有两个文件，*Cat.swift* 与 *Dog.swift*：

```
// Cat.swift:
class Cat {
    private var secretName : String?
}

// Dog.swift:
class Dog {
    func nameCat(cat:Cat) {
        cat.secretName = "Lazybones" // compile error
    }
}
```

上述代码将无法编译通过：编译器会报错“Cat does not have a member named `secretName`.”。现在，代码被放到了不同的文件中，因此私有性起作用了。

有些时候，你想将设置变量和读取变量时的私有性区分开。为了实现这种差别，请将单词 `set` 放在私有性声明后面的圆括号中。这样，`private(set) var myVar` 就表示对该变量的设置局限在了同一个文件的代码中。它并没有对变量的获取作任何限制，因此取默认值。与之类似，你可以写成 `public private(set) var myVar` 来使得变量读取变成公开的，同时保持变量的设置为私有的（可以对 `subscript` 函数使用相同的语法）。

### 5.3.2 Public 声明

如果编写模块，那就至少需要将一些对象声明为公开的，否则导入你的模块的代码就无法看到它们。未声明为公开的其他声明是内部的，这意味着它们对于模块来说是私有的。因此，请合理使用 public 声明来配置模块的公开 API。

比如，在我编写的 Zotz 应用（一个纸牌游戏）中，用于发牌和描述牌的对象类型与将纸牌形成一副牌的对象被绑定到了名为 ZotzDeck 的框架中。其中很多类型（如 Card 和 Deck）都被声明为公开的。不过，很多辅助对象类型则不是；ZotzDeck 模块中的类可以看到并使用它们，不过模块外的代码就完全不需要知道它们的存在。

公开的对象类型中的成员本身不会自动变成公开的。如果希望让某个方法是公开的，你需要将其声明为公开的。这是个非常棒的默认行为，因为这意味着除非你想这么做，否则这些成员不会在模块外被共享（正如 Apple 所做的，你必须“显式发布”对象成员）。

比如，在 ZotzDeck 模块中，Card 类被声明为了公开的，但其初始化器却不是这样。为什么呢？因为没必要。获得牌的方式是通过初始化 Deck 来实现的（这意味着要导入 ZotzDeck 模块）；Deck 的初始化器被声明为了公开的，因此你可以这么做。没有任何理由让牌脱离 Deck 单独存在，这都归功于隐私性规则，你做不到这一点。

### 5.3.3 隐私性规则

在语言发布早期，Apple 花了不少时间向 Swift 添加访问控制，很大程度上是因为编译器需要知道非常多的规则才能确保相关内容的隐私级别是一致的。比如：

- 如果类型是私有的，那么变量就不能是公开的，因为其他代码无法使用这种变量。
- 如果父类不是公开的，那么子类也不能是公开的。
- 子类可以改变重写成员的访问级别，但它看不到父类的私有成员，除非它们声明在同一个文件中。

诸如此类。我可以列出所有的规则，但不会这么做，因为没必要。Swift 手册对此有详尽的介绍，如果需要可以参考。一般来说，你可能用不上；这本身都是很直观的，如果违背了规则，编译器会通过有用的错误消息提示你。

## 5.4 内省

Swift 提供了有限的能力来内省对象，即让一个对象显示其属性名和属性值。该特性用于调试的目的，而不是用于程序的逻辑。比如，你可以通过它在 Xcode 调试窗格中修改对象的显示方式。

要想内省一个对象，在实例化 Mirror 时请将其作为 reflecting 参数。Mirror 的 children 是个名值对元组，描述了原始对象的属性。下面这个 Dog 类有个 description 属性，它利用了内省特性。相比于硬编码类实例属性的列表，我们通过内省实例来获取属性名与属性值。这意味着，我们可以在后面添加更多的属性，而无须修改 description 实现：

```
struct Dog : CustomStringConvertible {
    var name = "Fido"
    var license = 1
    var description : String {
       var desc = "Dog ("
       let mirror = Mirror(reflecting:self)
       for (k,v) in mirror.children {
           desc.appendContentsOf("\(k!): \(v), ")
       }
       let c = desc.characters.count
       return String(desc.characters.prefix(c-2)) + ")"
    }
}
```

如果实例化 Dog 并将实例传递给 print，那么控制台会打印出如下内容：

```
Dog (name: Fido, license: 1)
```

如果对象类型使用了 CustomStringConvertible（description 属性）与 CustomDebugStringConvertible（debugDescription 属性）协议，那么我们首选 description，不过还可以通过 debugPrint 函数输出 debugDescription。

通过使用 CustomReflectable 协议，我们可以接管 Mirror 的 children。要想做到这一点，我们实现了 customMirror 方法，返回自定义的 Mirror 对象，其 children 属性是我们所配置的名值对元组集合。

在下面这个简单的示例中，我们实现了 customMirror 来支持对属性的修改：

```
struct Dog : CustomReflectable {
    var name = "Fido"
    var license = 1
    func customMirror() -> Mirror {
        let children : [Mirror.Child] = [
            ("ineffable name", self.name),
            ("license to kill", self.license)
        ]
        let m = Mirror(self, children:children)
        return m
    }
}
```

结果就是，当我们在 Xcode 调试窗格控制台中打印出 Dog 实例时，自定义属性名会显示：

```
- ineffable name : "Fido"
- license to kill : 1
```

# 5.5 内存管理

Swift 内存管理是自动进行的，你通常不必考虑这个问题。对象在实例化后生成，如果不再需要则会消亡。不过在底层，引用类型对象的内存管理则是很复杂的；第 12 章将会介绍底层机制。甚至对于 Swift 用户来说，就这一点而言，事情有时也会出错（值类型不需要引用类型那种复杂的内存管理，因此对于值类型来说，内存管理不会出现什么问题）。

麻烦之事常常出现在两个类实例彼此引用的情况下。这种情况会出现保持循环，而这会导致内存泄漏，这意味着两个对象永远不会消亡。一些计算机语言通过周期性的“垃圾收集”阶段来解决这类问题，它会检测保持循环并进行清理，不过 Swift 并未采取这种做法；你只能手工避免保持循环的出现。

检测与观察内存泄漏的方式是实现类的 `deinit`。当实例消亡时会调用该方法。如果实例永远不会消亡，那么 `deinit` 就永远不会调用。如果你期望实例应该消亡但却没有消亡，这就是个危险信号。

下面是个示例。首先，我生成了两个类实例，并观测它们的消亡：

```
func testRetainCycle() {
    class Dog {
        deinit {
            print("farewell from Dog")
        }
    }
    class Cat {
        deinit {
            print("farewell from Cat")
        }
    }
    let d = Dog()
    let c = Cat()
}
testRetainCycle() // farewell from Cat, farewell from Dog
```

上述代码运行后，控制台会打印出两条“farewell”消息。我们创建了 Dog 实例与 Cat 实例，不过对它们的引用是位于“`testRetainCycle`”函数中的自动（局部）变量。当函数体执行完毕后，所有自动变量都会销毁；这正是其得名为自动变量的原因所在。再没有其他引用指向 Dog 与 Cat 实例，因此它们也会随之销毁。

现在修改一下代码，让 Dog 与 Cat 实例彼此引用：

```
func testRetainCycle() {
    class Dog {
        var cat : Cat?
        deinit {
            print("farewell from Dog")
        }
    }
    class Cat {
        var dog : Dog?
        deinit {
            print("farewell from Cat")
        }
    }
    let d = Dog()
    let c = Cat()
    d.cat = c // create a...
    c.dog = d // ...retain cycle
}
testRetainCycle() // nothing in console
```

上述代码运行后，控制台不会打印出任何“farewell”消息。Dog 与 Cat 对象会彼此引用，它们都是持久化引用（也叫作强引用）。持久化引用会保证，只要 Dog 引用了特定的 Cat，那么 Cat 就不会销毁。这是好事，也是明智的内存管理的基本原则。不好之处在于，Dog 与 Cat 彼此都有持久化引用。这是个保持循环！ Dog 实例与 Cat 实例都无法销毁，因为没有一个能先销毁掉，就像 Alphonse 与 Gaston 都无法进门一样，因为它们都要求对方先走。Dog 无法先销毁，因为 Cat 有对其的持久化引用，Cat 也不能先销毁，因为 Dog 有对其的持久化引用。

因此，这两个对象会造成泄漏。代码执行结束了；d 与 c 也不复存在了。再没有引用指向这两个对象；这些对象也无法再被引用。没有代码可以触及它们；没有代码能够延伸到它们。但它们还会继续存在，但毫无用处，只是占据着内存而已。

## 5.5.1 弱引用

对于保持循环的一种解决方案就是将有问题的引用标记为 weak。这意味着引用不再是持久化引用了。它是个弱引用。现在，即便引用者依旧存在，被引用的对象还是可以消亡。当然，这么做是有风险的，因为现在被引用的对象可能会在引用者背后销毁。不过 Swift 对此也提供了解决方案：只有 Optional 引用可以标记为 weak。通过这种方式，如果被引用的对象在引用者背后销毁，那么引用者会看到 nil。此外，引用必须是个 var 引用，这是因为只有它才可以变为 nil。

如下代码破坏了保持循环，并防止了内存泄漏：

```
func testRetainCycle() {
    class Dog {
        weak var cat : Cat?
        deinit {
            print("farewell from Dog")
        }
    }
    class Cat {
        weak var dog : Dog?
        deinit {
            print("farewell from Cat")
        }
    }
    let d = Dog()
    let c = Cat()
    d.cat = c
    c.dog = d
}
testRetainCycle() // farewell from Cat, farewell from Dog
```

上述代码做得有些过头了。为了破坏保持循环，没必要让 Dog 的 cat 与 Cat 的 dog 都成为弱引用；只需让其中一个成为弱引用就足以破坏这个循环了。事实上，这是解决保持循环问题的一种常规解决方案。只要二者之中的一个引用比另一个更强就可以；不太强的那个就会拥有一个弱引用。

如前所述，虽然值类型不会遇到引用类型才会遇到的内存管理问题，但值类型在与类实例一起使用时依然会遇到保持循环问题。在这个保持循环示例中，如果 Dog 是个类，Cat 是个结构体，那么依然会出现保持循环问题。解决方案是一样的：让 Cat 的 dog 成为一个弱引用（不能让 Dog 的 cat 成为弱引用，因为 Cat 是个结构体，只有对类类型的引用才能声明为 weak）。

请确保只在必要时才使用弱引用！内存管理不是儿戏。不过，在实际开发中，有时弱引用是正确之道，即便没有遇到保持循环问题时亦如此。比如，视图控制器对自身视图的父视图的引用通常是个弱引用，因为视图本身已经拥有了对子视图的持久化引用，我们不希望在视图本身不存在的情况下还保留对这些子视图的引用：

```
class HelpViewController: UIViewController {
    weak var wv : UIWebView?
    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)
        let wv = UIWebView(frame:self.view.bounds)
        // ... further configuration of wv here ...
        self.view.addSubview(wv)
        self.wv = wv
    }
    // ...
}
```

在上述代码中，self.view.addSubview(wv) 会导致 UIWebView wv 持久化；因此，我们自己对其的引用（self.wv）就是弱引用。

## 5.5.2 无主引用

Swift 还对保持循环提供了另一种解决方案。相对于将引用标记为 weak，你可以将其标记为 unowned。这个方案对于一个对象如果没有对另一个对象的引用就完全不复存在这一特殊情况很有用，不过该引用无须成为持久化引用。

比如，假设一个 Boy 可能有，也可能没有一个 Dog，但每个 Dog 一定会有一个对应的 Boy，因此我在 Dog 中声明了一个 init(boy:) 初始化器。Dog 需要一个对其 Boy 的引用，Boy 如果有 Dog，也需要一个对其的引用；这可能会形成一个保持循环：

```
func testUnowned() {
    class Boy {
        var dog : Dog?
        deinit {
            print("farewell from Boy")
        }
    }
    class Dog {
        let boy : Boy
        init(boy:Boy) { self.boy = boy }
        deinit {
            print("farewell from Dog")
        }
    }
    let b = Boy()
    let d = Dog(boy: b)
    b.dog = d
}
testUnowned() // nothing in console
```

可以通过将 Dog 的 boy 属性声明为 unowned 来解决这一问题：

```
func testUnowned() {
    class Boy {
        var dog : Dog?
        deinit {
            print("farewell from Boy")
        }
    }
    class Dog {
        unowned let boy : Boy // *
        init(boy:Boy) { self.boy = boy }
        deinit {
            print("farewell from Dog")
        }
    }
    let b = Boy()
```

```
    let d = Dog(boy: b)
    b.dog = d
}
testUnowned() // farewell from Boy, farewell from Dog
```

使用unowned引用的好处在于它不必非得是个Optional；实际上，它也不能是Optional，它可以是个常量（let）。不过，unowned引用也是有风险的，因为被引用的对象可能会在引用者背后消亡，这时如果使用该引用就会导致程序崩溃，如以下代码所示：

```
var b = Optional(Boy())
let d = Dog(boy: b!)
b = nil // destroy the Boy behind the Dog's back
print(d.boy) // crash
```

因此，只有在确保被引用对象的存活时间比引用者长时才应该使用unowned。

## 5.5.3 匿名函数中的弱引用与无主引用

如果实例属性的值是个函数，并且该函数引用了实例本身，那就会出现保持循环的一个变种情况：

```
class FunctionHolder {
    var function : (Void -> Void)?
    deinit {
        print("farewell from FunctionHolder")
    }
}
func testFunctionHolder() {
    let f = FunctionHolder()
    f.function = {
        print(f)
    }
}
testFunctionHolder() // nothing in console
```

我创建了一个保持循环，在匿名函数中引用了一个对象，该对象又引用了这个匿名函数。由于函数就是闭包，所以声明在匿名函数外部的FunctionHolder f会被匿名函数当作持久化引用。不过，该FunctionHolder的function属性包含了该匿名函数，它也是个持久化引用。因此形成了保持循环：FunctionHolder会一直引用函数，而函数也会一直引用FunctionHolder。

在这种情况下，我无法通过将function属性声明为weak或unowned来破坏保持循环。只有对类类型的引用才能声明为weak或unowned，而函数并不是类。因此，我需要在匿名函数中将捕获到的值f声明为weak或unowned。

Swift为此提供了一种精妙的语法。在匿名函数体开头（即in这一行，如果这一行有代

码就在 in 之前）加上一个方括号，里面是逗号分隔的会被外部环境捕获的有问题的类类型引用，每个引用前面加上 weak 或 unowned。这个列表叫作捕获列表。如果有捕获列表，那么捕获列表后面必须要跟着关键字 in。就像下面这样：

```
class FunctionHolder {
     var function : (Void -> Void)?
     deinit {
         print("farewell from FunctionHolder")
     }
}
func testFunctionHolder() {
    let f = FunctionHolder()
    f.function = {
        [weak f] in // *
        print(f)
    }
}
testFunctionHolder() // farewell from FunctionHolder
```

上述语法能够解决问题。不过，在捕获列表中将引用标记为 weak 会产生一个副作用，这需要你多加注意：这种引用会以 Optional 的形式传递给匿名函数。这么做很好，因为如果被引用的对象消亡了，那么 Optional 的值就为 nil。当然，你需要相应地修改代码，根据需要展开 Optional 来使用它。通常的做法是进行弱引用强引用跳跃：条件绑定中，在函数一开始就展开 Optional 一次：

```
class FunctionHolder {
     var function : (Void -> Void)?
     deinit {
         print("farewell from FunctionHolder")
     }
}
func testFunctionHolder() {
    let f = FunctionHolder()
    f.function = { // here comes the weak-strong dance
        [weak f] in // weak
        guard let f = f else { return }
        print(f) // strong
    }
}
testFunctionHolder() // farewell from FunctionHolder
```

条件绑定 let f = f 完成了两件事。首先，它展开了进入匿名函数中的 Optional f。其次，它声明了另一个常规（强）引用 f。这样，如果展开成功，那么新的 f 就会在作用域的其他地方继续存在。

在这个特定的示例中，如果匿名函数依旧存活，那么 FunctionHolder 实例 f 是不可能消亡的。并没有其他引用指向这个匿名函数；它只作为 f 的属性而存在。因此，我可

以避免背后的一些额外工作，就像弱引用强引用跳跃一样，在捕获列表中将 f 声明为 unowned。

在实际开发中，我常常会在这种情况下使用 unowned。很多时候，捕获列表中标记为 unowned 的引用都是 self。如下示例来自于我之前编写的代码：

```
class MyDropBounceAndRollBehavior : UIDynamicBehavior {
    let v : UIView
    init(view v:UIView) {
        self.v = v
        super.init()
    }
    override func willMoveToAnimator(anim: UIDynamicAnimator!) {
        if anim == nil { return }
        let sup = self.v.superview!
        let grav = UIGravityBehavior()
        grav.action = {
            [unowned self] in
            let items = anim.itemsInRect(sup.bounds) as! [UIView]
            if items.indexOf(self.v) == nil {
                anim.removeBehavior(self)
                self.v.removeFromSuperview()
            }
        }
        self.addChildBehavior(grav)
        grav.addItem(self.v)
        // ...
    }
    // ...
}
```

这里存在一个潜在的（相当不易察觉）保持循环可能：self.addChildBehavior(grav) 会导致对 grav 持有一个持久化引用，grav 有一个对 grav.action 的持久化引用，赋给 grav.action 的匿名函数引用了 self。为了破坏保持循环，我在匿名函数的捕获列表中将对 self 的引用声明为 unowned。

别惊慌！初学者可能会谨慎地对所有匿名函数使用 [weak self]。这么做是不必要的，也是错误的。只有保持的函数才会引起保持循环的可能性。仅仅传递一个函数并不会引入这种可能性，特别是在被传递的函数会被立刻调用的情况下。请在预防保持循环问题前确保一定会遇到保持循环问题。

## 5.5.4 协议类型引用的内存管理

只有对类类型实例的引用可以声明为 weak 或 unowned。对结构体或枚举类型实例的引用不能这么声明，因为其内存管理方式不同（不会遇到保持循环问题）。

因此，声明为协议类型的引用就会有问题。协议可以被结构体或枚举使用。因此，你不能随意地将这种引用声明为 weak 或 unowned。只有协议类型的引用是类协议，你才能将其声明为 weak 或 unowned，也就是说，其被标记为了 @objc 或 class。

在如下代码中，SecondViewControllerDelegate 是我声明的协议。如果不将 SecondViewControllerDelegate 声明为类协议，那么代码是无法编译通过的：

```
class SecondViewController : UIViewController {
    weak var delegate : SecondViewControllerDelegate?
    // ...
}
```

下面是 SecondViewControllerDelegate 的声明；它被声明为了类协议，因此上述代码是合法的：

```
protocol SecondViewControllerDelegate : class {
    func acceptData(data:AnyObject!)
}
```

Objective-C 中声明的协议会被隐式标记为 @objc，并且是类协议。因此，如下声明是合法的：

```
weak var delegate : WKScriptMessageHandler?
```

WKScriptMessageHandler 是由 Cocoa 声明的协议（由 Web Kit 框架声明）。因此，它会被隐式标记为 @objc；只有类才能使用 WKScriptMessageHandler，因此编译器认为 delegate 变量是个类实例，其引用可以标记为 weak。

# 第二部分

# IDE

到现在为止，你肯定迫不及待地想要开始编写应用了。这时，你需要对所用的工具有比较好的了解。诸多工具可以总结为一个词儿：Xcode。这部分将会探索 Xcode，它是你在编写 iOS 应用时所用的 IDE（集成开发环境）。Xcode 是个庞大的应用，编写一个应用涉及大量内容；这部分将会帮助你熟悉 Xcode。在这个过程中，我们会通过一些手把手的教程来创建一个可用的应用。

- 第 6 章会概览 Xcode 并介绍项目的结构，同时还会介绍用于生成应用的诸多文件。
- 第 7 章将会介绍 nib。nib 是个包含界面绘制的文件。理解 nib（知道其工作原理及其与代码之间的交互方式）对于 Xcode 的使用以及应用的开发是至关重要的。
- 第 8 章将会介绍 Xcode 文档以及关于 API 的其他信息来源。
- 第 9 章将会介绍如何编辑、测试与调试代码，以及向 App Store 提交应用所需的各项步骤。一开始可以快速浏览一遍本章内容，待到开发并提交实际的应用时再回过头来将其作为详尽的参考。

第 6 章

# Xcode 项目剖析

Xcode 是个用于开发 iOS 应用的应用。Xcode 项目是应用的源泉；它包含了构建应用所需的全部文件与设置。要想创建、开发和维护应用，你需要了解如何操控 Xcode 项目。你需要掌握 Xcode、了解 Xcode 项目的本质与结构，以及 Xcode 的展现方式。这正是本章的主题。

术语“Xcode”实际上有两层含义。一方面，它是你编辑和构建应用时所用的应用的名字；另一方面，它是与之相伴的整个工具套件的名字。从后者的角度来看，Instruments 与 Simulator 都是 Xcode 的一部分。通常，这种模糊的划分并不会带来什么问题。

Xcode 是个强大、复杂且非常庞大的程序。在学习 Xcode 时，我建议的做法是不要过于发散：如果不理解某些东西，那不必担心，不用管它，也不要碰它，因为你可能不小心修改了某些重要的东西。我们对 Xcode 的介绍会从一个安全、受限且基本的路径开始，重点关注那些必要的功能而忽略其他功能。

要想了解完整信息，请参阅 Apple 的文档（选择 Help → Xcode Overview）；乍一看文档内容相当多，但你所需了解的只是其中一小部分。现在还有专门介绍 Xcode 的图书。

## 6.1 新建项目

甚至在编写代码前，Xcode 项目就已经非常复杂了。为了更好地理解，我们创建一个全新的“空”项目；你很快就会发现这根本就不是一个空项目。

1. 打开 Xcode 并选择 File → New → New Project。

2. 这时会弹出“Choose a template”对话框。模板是项目初始文件与设置的集合。在选择模板时，你实际上选择的是现有的包含了文件的目录；基本上，这些目录位于 *Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/Library/Xcode/Templates/Project Templates/iOS/Application* 中。本质上，Xcode 会复制该目录并填充一些值来创建项目。

   对于该示例来说，请选择左边 iOS 下的 Application。在右边选择 Single View Application，然后单击 Next。

3. 现在需要为项目起个名字（Product Name），请输入 Empty Window 作为项目名。

   在真实的项目中，你需要好好想想项目名，因为你要经常与它打交道。由于 Xcode 会复制模板目录，所以它会使用项目名“填充几处空白”，包括应用名。这样，你在这里所输入的名字将会被整个项目所用。不过，项目名确定好之后并不是永远不会变的，有单独的设置可以让你在任何时候都可以修改应用名。稍后将会介绍如何修改项目名（见 6.6 节）。

   项目名中也可以包含空格。可以在项目名、应用名以及 Xcode 自动生成的各种文件和目录名中使用空格；空格只在很少的地方会出现问题（比如，下面将会介绍的包标识符等），你在 Product Name 中输入的名字中的空格会被转换为连字符。不过，请不要在项目名中使用任何其他的标点符号！这些标点符号会导致 Xcode 的某些特性出现问题。

4. 注意到 Organization Identifier 域。第一次创建项目时该域是空的，你应该填写其中的内容。你需要填入一个唯一的字符串来标识自己或组织。约定的做法是以 `com.` 作为组织标识符的开头并且后跟其他人不大会使用的字符串（可能包含多个点分隔的字符串）。比如，我会使用 `com.neuburg.matt`。设备上或提交到 App Store 的每个应用都需要一个唯一的包标识符。应用的包标识符位于组织标识符下方，显示为灰色，它由组织标识符和项目名的版本号构成；如果为自己开发的每个项目起一个唯一的名字，那么包标识符就可以唯一区分项目以及它所生成的应用（如果需要，后面也可以手工修改包标识符）。

5. 你可以通过 Language 弹出菜单选择 Swift 与 Objective-C。这个选择并不是一成不变的；它只是规定了项目模板的初始结构与代码，不过你可以自由向 Objective-C 项目中添加 Swift 文件，也可以向 Swift 项目中添加 Objective-C 文件。你甚至还可以从 Objective-C 项目开始，稍后再将其转换为 Swift。现在，请选择 Swift。

6. 将 Device 弹出菜单设为 iPhone。重申一次，这个选择并不是一成不变的；不过现在，假设我们的应用只会运行在 iPhone 上。

7. 不要勾选 Use Core Data、Include Unit Tests 与 Include UI Tests，单击 Next。

8. 现在 Xcode 已经知道该如何构建项目。基本上，它会从方才提到的 *Project Templates* 目录中复制 *Single View Application.xctemplate* 目录。但你需要告诉它将目录复制到何处。这正是 Xcode 现在会弹出保存对话框的原因所在。你需要指定待创建的目录位置，即该项目的项目目录。项目目录可以位于任何地方，你可以在创建后移动它。我常常在桌面上创建新项目。
9. Xcode 还可以为项目创建 git 仓库。在实际开发中，这是非常方便的（参见第 9 章），但现在请不要勾选该复选框，单击 Create。
10. 磁盘上会创建好 Empty Window 项目目录（如果你指定在桌面上创建项目，那么目录就在桌面上），Xcode 会打开 Empty Window 项目的项目窗口。

我们刚才创建的项目是个可运行的项目；它确实可以构建出名为 Empty Window 的 iOS 应用。要想做到这一点，请确保项目窗口工具栏中的方案与目标显示为 Empty Window → iPhone 6（方案与目标实际上是个弹出菜单，如果需要可以单击它们修改其值）。选择 Product → Run。过一会儿，iOS Simulator 应用就会出现并运行你的应用，即一个空白界面。

构建项目需要编译代码、将编译好的代码和其他各种资源装配到实际的应用中。通常，如果想要了解代码是否编译通过、项目的构建是否正确，你需要构建项目（Product → Build）。此外，还可以编译单个文件（选择 Product → Perform Action → Compile [文件名]）。要想运行项目以便启动构建好的应用，可以在 Simulator 或连接的设备上运行；如果想要了解代码运行是否正常，那就需要运行项目（Product → Run），如果必要，运行之前会自动构建。

# 6.2 项目窗口

Xcode 项目包含了大量的信息，这些信息描述了构成项目的文件以及在构建应用时该如何使用这些文件，比如：

- 待编译的源文件（代码）。
- *.storyboard* 或 *.xib* 文件，它们会以图形化方式表示界面对象，在应用运行时进行实例化。
- 资源，如图标、图片或声音文件，它们是应用的组成部分。
- 在构建应用时需要遵循的所有设置（编译器、链接器的指令等）。
- 代码运行时所需的框架。

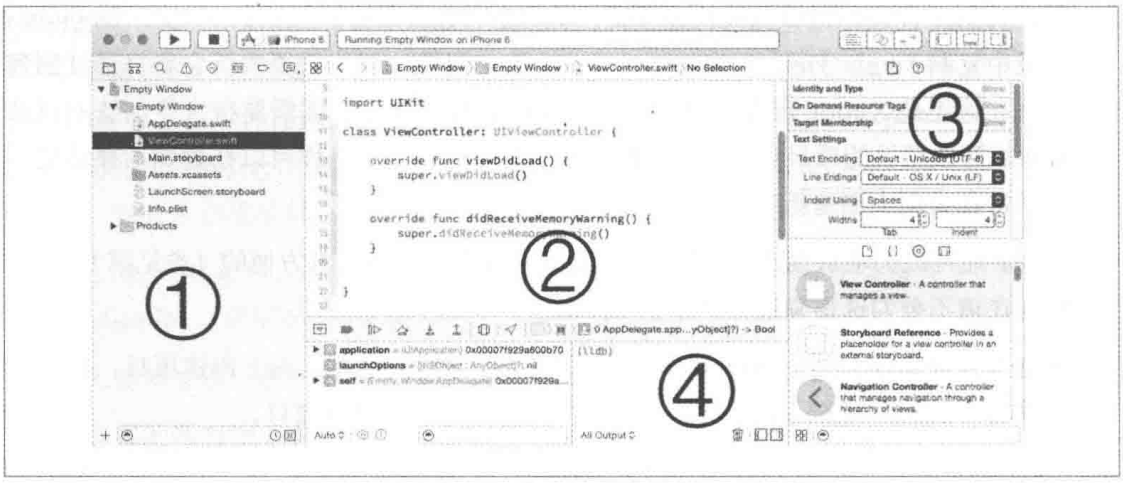


图 6-1：项目窗口

Xcode 项目窗口会展现出所有这些信息，还可以使用、编辑，并在代码间导航；此外，它还能够呈现出构建或调试应用的进度与结果。该窗口显示出了大量信息与功能！项目窗口非常强大和复杂；学习如何使用需要花些时间。下面就来探索这个窗口，看看其构造方式。

项目窗口有 4 个主要的构成（如图 6-1 所示）：

1. 左边是导航窗格。你可以通过 View → Navigators → Show/Hide Navigator（Command-0）或单击工具栏中最右侧的第 1 个 View 按钮来显示或隐藏。
2. 中间是编辑窗格（简称为“编辑器”）。这是项目窗口的主要区域。项目窗口几乎总会显示一个编辑窗格，并且还可以同时显示多个编辑窗格。
3. 右边是辅助窗格。你可以通过 View → Utilities → Show/Hide Utilities（Command-Option-0）或单击工具栏最右侧的第 3 个 View 按钮来显示或隐藏。
4. 底部是调试窗格。你可以通过 View → Debug Area → Show/Hide Debug Area（Shift-Command-Y）或单击工具栏最右侧的第 2 个 View 按钮来显示或隐藏。

所有的 Xcode 键盘快捷键都可以定制；参见 Preferences 窗口的 Key Binding 窗格。我这里所用的键盘快捷键都是默认值。

## 6.2.1 导航窗格

导航窗格就是项目窗口左侧的信息列。你主要通过它来控制项目窗口的主要区域会显示

什么（编辑器）。对于Xcode来说，一个重要的使用模式就是：在导航窗格中选中某个东西，它就会显示在编辑器中。

你可以切换导航窗格的可见性（View → Navigators → Hide/Show Navigator或Command-0）；比如，如果通过导航窗格找到了所需的条目，那么你可能想暂时隐藏导航窗格来增加屏幕的尺寸（特别是在小显示器上）。你可以通过拖曳右边的竖线来改变导航窗格的宽度。

导航窗格本身可以显示8种不同的信息；这样实际上会有8种导航器。这是通过上方的8个图标来表示的；要想在导航器间切换，请使用这8个图标或相应的键盘快捷键（Command-1、Command-2等）。如果导航窗格被隐藏了，那么请按下导航器的键盘快捷键，这会显示出导航窗格并切换到相应的导航器上。

根据你在Xcode首选项Behaviors窗格中设置的不同，在执行某个动作时，导航器可能会自动显示出来。比如，默认情况下，在构建项目时，如果出现了警告或错误消息，那么Issue导航器就会出现。这种自动的行为并不会让人生厌，因为通常这就是你需要的行为，如果不是，那么你可以修改它；此外你还可以随时切换到其他的导航器上。

首先来体验一下各种导航器吧：

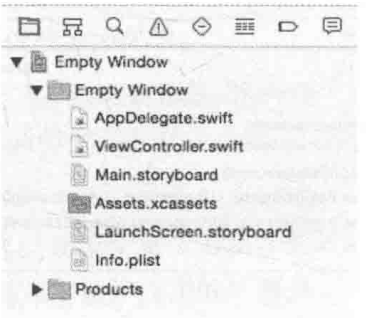


图6-2：项目导航器

*项目导航器（Command-1）*

单击项目导航器中的条目可以在构成项目的文件间导航。比如，在Empty Window目录中（在项目导航器中，这些类似于目录的东西实际上叫作分组），单击*AppDelegate.swift*文件可以在编辑器中查看其代码（如图6-2所示）。

在项目导航器顶层有个蓝色的Xcode图标，它代表Empty Window项目自身；单击它可以查看与项目及其目标相关的各种设置。在不了解的情况下请不要修改任何设置！稍后我将介绍这些设置。

可以通过项目导航器底部的过滤栏限制显示的文件；如果文件有很多，那么通过它可以快速找到已知名字的文件。比如，可以在过滤栏搜索框中输入“delegate”。试验完后请不要忘记删除过滤信息。

如果对导航器进行了过滤，那么它会一直进行过滤，除非将其删除，否则连关闭项目也不行！常见的一个错误是对导航器进行了过滤，但却忘记了，这样就看不到结果了（因为你一直在盯着导航器本身，没看到下面的过滤栏），你还纳闷：“我的文件去哪儿了？”

*符号导航器（Command-2）*

符号就是个名字，通常是类或方法的名字。它有助于代码导航。比如，你可以选择过滤器栏中的前两个图标（前两个是蓝色的，后一个是深色的），然后快速查看 AppDelegate 的 `applicationDidBecomeActive:` 方法定义。

你可以通过各种方式选择过滤器栏的图标以查看符号导航器内容的变化。在过滤栏的搜索框中输入一些字符以限制符号导航器中的内容；比如，你可以尝试在搜索框中输入“active”，然后看看发生了什么变化。

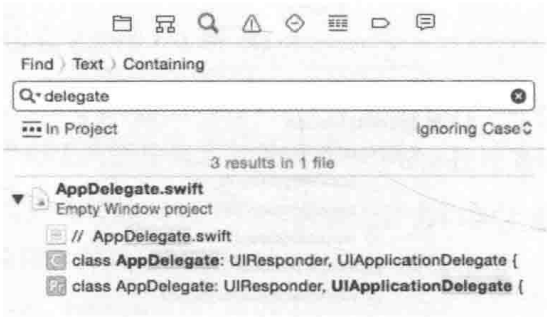


**图 6-3：搜索导航器**

如果第 2 个过滤器图标没有高亮，那就会显示出所有符号，包括 Swift 与 Cocoa 所定义的内容（如图 4-1 所示）。这是查看对象类型的一种绝佳方式，同时还可以快速进入声明这些类型的头文件中（一种重要的文档形式，参见第 8 章）。

*搜索导航器（Command-3）*

该强大的搜索功能用于寻找项目中的文本。你还可以通过 Find → Find in Project（Command-Shift-F）调出搜索导航器。搜索框上的单词会展示出现在所用的选项；

他们是弹出菜单，可以通过单击其中一个来改变选项。试着搜索“delegate”（如图6-3所示）。单击任何一条搜索结果就会跳转到代码中该文本出现的位置。

在搜索框的左下方是当前的搜索区域。可以单击它来查看搜索域面板。你可以限制只对项目中的某个分组（目录）进行搜索，还可以定义新的域：单击New Scope会弹出域配置窗口，你可以从中查看选项。域是针对每个用户而不是项目定义的；这里所创建的域也会出现在其他项目中。

可以在其他搜索域(位于底部的过滤栏)中输入内容来进一步限制显示的搜索结果(现在我不打算再介绍过滤栏了，不过每个导航器都有某种形式的过滤栏）。

*问题导航器（Command-4）*

问题导航器主要在代码中出现问题时使用。它指的并不是项目中有问题；而是Xcode的一个术语，指的是项目构建时所出现的警告与错误消息。

要想查看问题导航器，你需要给代码制造点问题才行。转到（你应该知道如何做了，至少有3种方式）文件*AppDelegate.swift*，在文件顶部最后一行注释后面的空行下及`import`行之上输入`howdy`。构建项目（Command-B）。这时问题导航器会显示出一些错误消息，表示编译器无法处理这种出现在不合法位置处的不合法的单词。单击其中一个问题可以在文件中查看它。在代码中，问题“气球”会出现在有问题这一行的右侧；如果觉得有干扰，你可以通过Editor → Issues → Hide/Show All Issues（Command-Control-M）改变其可见性。

既然你已经让Xcode报错了，那么请选择“howdy”并将其删除；保存并再次构建，这时问题会消失不见。真希望实际开发中也能这么简单！

*测试导航器（Command-5）*

该导航器会列出测试文件以及每个测试方法，同时还可以运行测试并查看测试是通过还是失败了。测试代码并不属于应用的一部分；它会调用应用代码以检测其行为是否与期望一致。第9章将会对测试进行更多的介绍。

*调试导航器（Command-6）*

在默认情况下，该导航器只在调试暂停时才会出现。对于Xcode来说，运行与调试之间的差别并不明显；环境都是一样的。差别只不过在于是否使用了断点（第9章将会详细介绍关于调试的相关信息）。

要想查看调试导航器，你需要为代码设定断点。再一次转到文件*AppDelegate.swift*，选中`return true`这一行，并选择Debug → Breakpoints → Add Breakpoint at Current Line，这时蓝色的断点箭头就会出现在该行。运行项目。在默认情况下，当遇到断点时，导航窗格会切换到调试导航器，调试窗格会出现在窗口下方。调试项目时，你很快就会熟悉这一总体布局（如图6-4所示）。

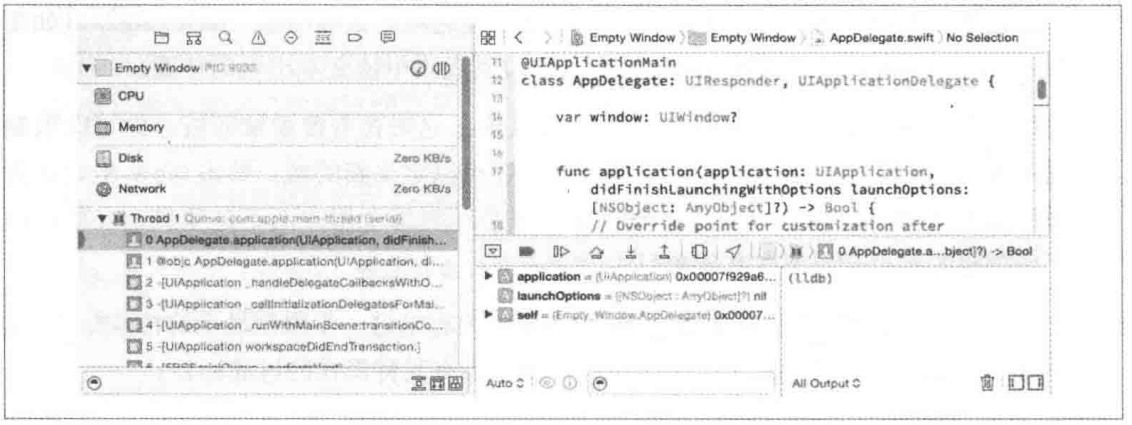


图 6-4：调试布局

调试导航器以几个数字与图形化的分析信息展示开始（至少会有 CPU、内存、磁盘与网络）；单击其中一个可以在编辑器中看到更多的图形化信息。在应用运行时，可以通过这些信息追踪应用可能的错误行为，从而避免了运行 Instruments 辅助工具（第 9 章将会介绍）的复杂性。要想切换调试导航器顶部分析信息的可见性，请单击“gauge”图标（位于进程名右侧）。

调试导航器还会显示出调用堆栈，其中会显示出暂停位置处的嵌套方法名；如你所想，你可以通过单击方法名来导航。你可以通过导航器底部过滤栏中的第 1 个按钮来缩短或增加列表。可以通过进程名右侧的第 2 个图标在根据线程显示与根据队列显示之间进行切换。

调试窗格包含了两个子窗格，你可以根据需要显示或隐藏它们（View → Hide/Show Debug Area 或 Command-Shift-Y）：

*变量列表（位于左侧）*

里面是调用堆栈中所选方法作用域中的变量。

*控制台（位于右侧）*

调试器会将文本消息显示在这里；你可以通过这里查看到运行的应用所抛出的异常，同时还可以让代码有意发送描述应用进程与行为的日志消息。这些消息非常重要，因此在应用运行时请密切关注控制台。还可以使用控制台向调试器输入命令。在暂停时，这通常是比变量列表更好的查看变量值的方式。

可以通过窗格右下角的两个按钮来隐藏变量列表和控制台。还可以通过 View → Debug Area → Activate Console 来显示出控制台。

可以通过视图调试查看应用的视图层次结构。要想切换到视图调试，请选择 Debug → View Debugging → Capture View Hierarchy（或单击调试窗格顶部栏中的调试视图层次按钮）。

*断点导航器（Command-7）*

该导航器会列出所有断点。目前只有一个断点导航器，但在调试有很多断点的大型项目时，你会觉得该导航器很有用。此外，你可以在这里创建特殊断点（如符号断点），通常这是管理现有断点的中心位置。我们会在第 9 章对其进行详细介绍。

*报告导航器（Command-8）*

该导航器会列出最近的主要动作，如项目的构建或运行（调试）。在执行某个动作时，单击清单就可以查看（在编辑器中）到所生成的报告。报告可能包含其他途径无法显示的信息，此外，你还可以通过它回想起最近的一些消息（比如，“刚才调试时出现了哪个异常”）。

举个例子，通过单击成功构建的清单，然后通过报告上方的过滤器开关来选择显示所有消息，那么我们可以看到构建的每一个步骤（如图 6-5 所示）。要想显示某一步的所有文本，请单击该步骤，然后单击最右边的 Expand Transcript 按钮（参见 Editor 菜单中的菜单项）。

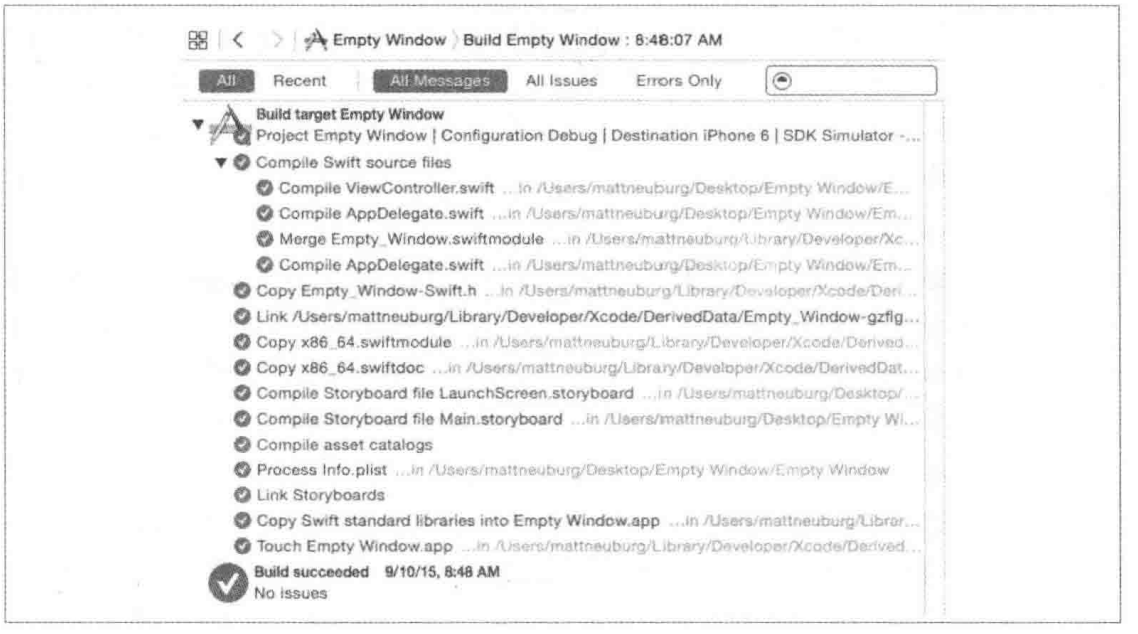


图 6-5：查看报告

在通过单击导航窗格进行导航时，不同的单击方式会影响导航的结果。在默认情况下，按住 Option 并单击会在辅助窗格中导航（稍后将会介绍）、双击会打开新窗口，按住

Option与Shift并单击会弹出一个小的智能窗格，你可以指定导航到哪里（新窗口、新页签，或新的辅助窗格）。要想了解管理这些单击的设置，请访问Xcode首选项的导航窗格。

## 6.2.2 辅助窗格

辅助窗格是位于项目窗口右边的那一列。它包含了查看器，这些查看器提供了关于当前所选以及设置的信息；如果设置可以修改，那么可以在这里进行修改。它还包含了编辑项目时所需的一些库（所需的对象来源）的信息。在编辑 *.storyboard* 或 *.xib* 文件（第7章将会介绍）时会凸显其重要性。不过，它对于代码编辑来说也很有用，因为快速帮助（文档的一种形式，第8章将会介绍）也会显示在这里；辅助窗格还是代码片段的来源（参见第9章）。要想显示或隐藏辅助窗格，请选择 View → Utilities → Hide/Show Utilities（Command-Option-0）。你可以通过拖曳其左边的竖线来改变辅助窗格的宽度。

辅助窗格包含了大量控制板，它们会形成多个集合并被划分到两个主要的分组中：窗格的上半部分与下半部分。你可以通过拖曳它们之间的水平线来改变二者的相对高度。

*上半部分*

辅助窗格上半部分会有哪些内容取决于当前编辑器所选内容。主要情况有如下4种：

*正在编辑代码文件*

辅助窗格的上半部分要么显示文件查看器，要么显示快速帮助。你可以通过辅助窗格上半部分顶部的图标或键盘快捷键（Command-Option-1、Command-Option-2）来切换它们。文件查看器很少会用到，但快速帮助则可作为文档来用（参见第8章）。文件查看器包含了多个部分，每个部分都可以通过单击头部来展开或收起。

*正在编辑 .storyboard 或 .xib 文件*

除了文件查看器和快速帮助，辅助窗格的上半部分还可以显示身份查看器（Command-Option-3）、属性查看器（Command-Option-4）、尺寸查看器（Command-Option-5）和连接查看器（Command-Option-6）。它们包含了多个部分，每一部分都可以通过单击头部来展开或收起。

*正在编辑资源文件*

除了文件查看器和快速帮助，属性查看器还会列出关于所选资源集或资源的更多信息。你可以通过它确定列出所选资源集的哪些变体，并设置资源标签；如果所选资源是图片，那么你可以配置关于它的一些额外信息。

*正在调试视图层次*

除了文件查看器和快速帮助，对象查看器可以显示关于当前所选视图的信息，尺寸查看器可以显示当前所选视图的大小、位置与约束。

*下半部分*

辅助窗格的下半部分会显示四种库之一。你可以通过单击顶部的图标或键盘快捷键来切换显示的库。这些库分别是文件模板库（Command-Option-Control-1）、代码片段库（Command-Option-Control-2）、对象库（Command-Option-Control-3）以及媒体库（Command-Option-Control-4）。对象库是其中最为重要的；在编辑 *.storyboard* 或 *.xib* 文件时会经常使用到它。

要想查看关于库中当前所选条目的描述信息，请按空格键。

## 6.2.3 编辑器

项目窗口中间是编辑器。你在这里完成实际的工作，阅读并编写代码（参见第 9 章），在 *.storyboard* 或 *.xib* 文件中设计界面（参见第 7 章）。编辑器是项目窗口的核心。你可以关闭导航窗格、辅助窗格和调试窗格，但如果项目窗口中没有编辑器就完全不行了（虽然你可以通过调试窗格来使用编辑器）。

编辑器提供了自己的导航形式，即顶部的跳转栏。它不仅会以分层方式显示出当前正在编辑的文件，你还可以通过它切换到不同的文件。特别地，跳转栏中的每个路径组成也是个弹出菜单。这些弹出菜单可通过单击每个路径组成来调出，或使用键盘快捷键（在 View → Standard Editor 子菜单中的第 2 部分）。比如，Control-4 会弹出分层的弹出菜单，你完全可以通过键盘在菜单中导航，这样就可以选择项目中的不同文件来编辑了。此外，跳转栏中的每个弹出菜单也是个搜索框；你可以从跳转栏中弹出菜单并输入内容。通过这种方式，即便项目导航器没有显示出来，你也可以导航项目。

跳转栏最左侧的符号（Control-1）会弹出一个层次化菜单（相关条目菜单），可以导航到与当前文件相同的文件上。这里出现的内容不仅取决于当前正在编辑的文件，还与该文件当前所选内容相关。这是个非常强大且便捷的菜单，你应该花些时间好好研究它。你可以导航到相关类文件和头文件（父类、子类与兄弟类；兄弟类指的是拥有共同父类的类）；你可以查看被当前所选方法调用的方法，并调用当前所选的方法。在 Xcode 7 中，选择 Generated Interface 可以查看到 Swift 文件的公开 API 以及 Swift 可以看到的 Objective-C 头文件。

编辑器会记住所显示的历史信息，你可以通过跳转栏中的后退按钮回到之前查看的内容，这也是个弹出菜单，可以从中进行选择。此外，还可以选择 Navigate → Go Back（Command-Control-Left）。

在开发项目时，你很有可能想要同时编辑多个文件，或获得同一个文件的多个视图以便能够同时编辑文件的两个区域。这可以通过 3 种方式实现：辅助窗格、页签与第二窗口。

*辅助窗格*

你可以通过辅助窗格将一个编辑器分割为多个编辑器。要想做到这一点，请单击工具栏中的第 2 个编辑器按钮（“显示辅助编辑器”），或选择 View → Assistant Editor → Show Assistant Editor（Command-Option-Return）。此外在默认情况下，在导航时按住 Option 键会打开一个辅助窗格；比如，按住 Option 键并单击导航窗格或按住 Option 键并选择跳转栏，这会打开一个辅助窗格（如果已经有辅助窗格，那么就会导航到现有的辅助窗格）。要想移除辅助窗格，请单击工具栏中的第 1 个编辑器按钮，或选择 View → Standard Editor → Show Standard Editor（Command-Return），还可以单击辅助窗格右上角的 X 按钮。

你可以确定辅助窗格的布局。要想做到这一点，请选择 View → Assistant Layout 子菜单。一般情况下，我更喜欢 All Editors Stacked Vertically，但这仅仅是个人习惯而已。一旦打开了辅助窗格，你就可以进一步将其分割为更多的辅助窗格。要想做到这一点，请单击辅助窗格右上方的“+”按钮。要想隐藏辅助窗格，请单击右上方的 X 按钮。

辅助窗格之所以是辅助窗格，而不仅仅是一种分割窗格编辑器，原因在于它与主编辑器窗格之间可以保持着特殊的关系。默认情况下，主编辑器窗格的内容是由你在导航窗格中所单击的条目来决定的；同时，辅助窗格可以响应主编辑器窗格中正在编辑的文件，它会智能地改变正在编辑（辅助窗格）的文件，这叫作追踪。要想配置辅助窗格的追踪行为，请使用跳转栏中的第 1 个组件（Control-4）。这是个追踪菜单；它类似于刚才介绍的相关条目菜单，不过选择某个类别会决定自动化的追踪行为。如果某个类别有多个文件，那么一对箭头按钮就会出现在跳转栏的最右侧，你可以通过它们进行导航（或使用第 2 个跳转栏组件，Control-5）。可以通过将辅助窗格的第 1 个跳转栏组件设为 Manual 来关闭追踪。

如果想要关闭辅助窗格，但又想继续编辑内容，请先将辅助窗格的内容移动到主编辑器窗格中（Navigate → Open In Primary Editor）。

*页签*

你可以将整个项目窗口界面表示为一个页签。要想做到这一点，请选择 File → New → Tab（Command -T），如果之前并未显示出页签栏，那么这会将其显示出来（就在工具栏下面）。页签界面的使用就像 Safari 等应用一样。你可以通过单击页签或使用 Command-Shift-} 来切换页签。一开始，新的页签看起来与建立页签的原始窗口别无二致。不过现在你可以在页签上做一些修改，即改变显示的窗格或编辑的文件，

同时又不会影响其他页签。这样就可以得到项目的多个视图。你可以为每个页签添加一个描述性的名字：双击页签名就可以进行编辑。

*第二窗口*

第二项目窗口类似于页签，但它是个独立的窗口，而页签则位于相同的窗口中。要想创建第二窗口，请选择 File → New → Window（Command-Shift-T）。此外，你还可以将页签拖曳出当前的窗口而使之成为一个窗口。

页签与第二窗口之间的差别并不明显；无论使用哪一个，无论出于何种目的其实都是习惯和方便的问题。我发现第二窗口的优势在于你可以同时将其看作主窗口，这个窗口会小一些。这样，如果有文件频繁被引用，那么我会使用第二窗口作为编辑器来显示该文件，它不会占据太多屏幕空间，也不需要额外的窗格。

页签与窗口都拥有自定义行为。比如，如前所述，调试时能够查看控制台是非常重要的；我喜欢在满屏项目窗口中查看，不过还希望可以切换回来查看代码。因此，我创建了一个自定义行为（单击 Preferences 窗口 Behaviors 窗格底部的 + 按钮），它会执行两个动作：在活动窗口中显示出名为 Console 的页签，并显示出 Console View 调试器。此外，我还为该行为指定了一个键盘快捷键。这样，任何时候就都可以通过键盘快捷键切换至 Console 页签了（如果不存在则创建），这只会显示出控制台。它就是一个页签，因此可以通过 Command-Shift-} 在它与代码间切换。

有多种方式可以改变编辑器中的内容，导航器并不会自动与这些变更进行同步。要想从项目导航器中选择在当前编辑器中显示的文件，请选择 Navigate → Reveal in Project Navigator。

# 6.3 项目文件及其依赖

项目导航器（Command-1）中的第 1 项表示项目本身（在本章之前创建的 Empty Window 项目中，它叫作 Empty Window）。以层次方式依赖它的则是用于项目构建的各种条目。这些条目与项目本身都对应于磁盘上项目目录中的条目。

为了了解这种对应关系，我们同时在 Finder 和 Xcode 项目窗口中看一看。在项目导航器中选中项目清单，然后选择 File → Show in Finder。Finder 会显示出项目目录中的内容（如图 6-6 所示）。

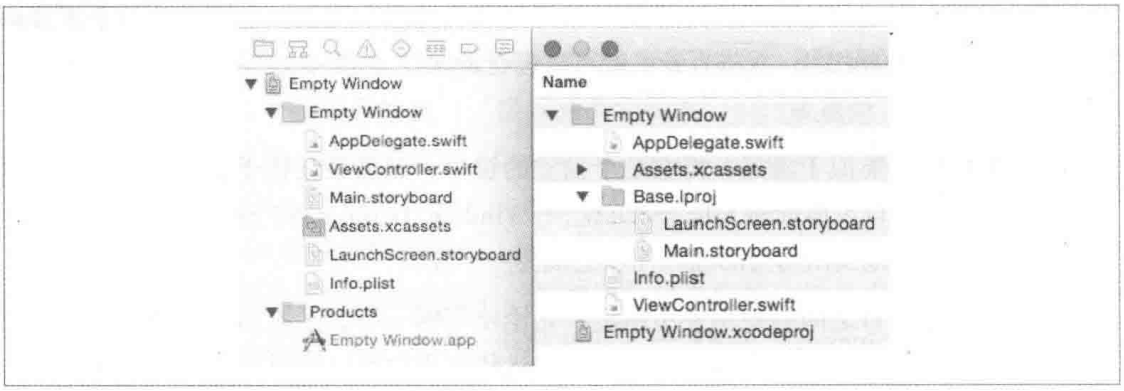


图 6-6：项目导航器与项目目录

然而，绝对不要在 Finder 中修改项目目录中的任何文件，除了双击项目文件以打开项目。不要将任何文件直接放到项目目录中。不要删除项目目录中的任何文件。不要重命名项目目录中的任何文件。重申一次，不要修改项目目录中的任何文件！请通过 Xcode 的项目窗口来处理项目（如果你是个 Xcode 超级用户，那么你应该清楚何时可以违背该原则。但现在请严格遵守该原则）。

之所以会有上述警告，原因在于项目期望项目目录中的文件能够符合一定的格式。如果直接在 Finder 中对项目目录进行修改，那么项目目录就会被破坏，这也会破坏项目。当在项目窗口中工作时，Xcode 本身会对项目目录做必要的修改，这不会导致任何问题。

项目目录中最重要的文件是 Empty *Window.xcodeproj*。它是项目文件，对应于项目导航器中所列出的项目。Xcode 关于项目的所有信息（包含了哪些文件以及如何构建项目）都存储在该文件中。要想从 Finder 中打开项目，请双击项目文件。此外，还可以将项目目录拖曳到 Xcode 的图标（位于 Finder、Dock 或应用目录中）上，Xcode 就会找到该项目文件并将其打开；你永远都不需要打开项目目录！

项目导航器中显示的分组与文件是按照层次结构依赖于项目文件的，它们对应于磁盘上的文件，这一点可以通过 Finder 看到（图 6-6）。回忆一下，分组是一个技术术语，对应于项目导航器中所显示的类似于目录的对象：

- Empty Window 分组直接对应于磁盘上的 *Empty Window* 目录。项目导航器中的分组并不一定对应于 Finder 中磁盘上的目录；同时，Finder 中磁盘上的目录也不一定对应于项目导航器中的分组。不过在该示例中，二者之间存在着对应关系！
- Empty Window 分组中的文件（如 *AppDelegate.swift*）对应于磁盘上 *Empty Window* 目录中的真实文件。如果创建了其他代码文件（在实际开发中，你肯定会在项目开发的过程中创建文件），那么你会将这些文件放在项目导航器的 Empty Window 分

组中，这样它们就会位于磁盘上的 *Empty Window* 目录中。（然而这么做并不是必需的；文件可以位于任何地方，项目也不会出现任何问题。）

- Empty Window 分组中的两个文件 *Main.storyboard* 与 *LaunchScreen.storyboard* 位于 Finder 的 *Base.lproj* 目录中，该目录并没有出现在项目导航器中。这与本地化有关，我将在第 9 章对其进行介绍。
- 项目导航器中的条目 *Assets.xcassets* 对应于磁盘上专门的结构化目录 *Assets.xcassets*。这是个资源目录；你可以在 Xcode 中向资源目录添加图片，它会在磁盘上维护该目录。在本章后面以及第 9 章将会详细介绍资源目录。

你可能想要找到诸如此类的所有不一致的地方。千万不要这么做！记住，不要直接通过 Finder 操纵磁盘上的项目目录。你已经看到了，并且知道它里面有内容，同时也清楚它与项目导航器之间存在一定的关联关系。将注意力放在项目导航器上，在这里修改项目，这样就不会出现任何问题了。

在开发项目和向其中添加文件时，你可以随意向项目导航器中添加额外的分组。分组的目的旨在让项目导航器更好用！它们并不会影响应用的构建方式，在默认情况下，也不会对应于磁盘上的任何目录；它们只是为了在项目导航器中更方便地进行组织。要想创建新的分组，请选择 File → New → Group。要想重命名分组，请在项目导航器中将分组选中，然后按下回车键使分组名变成可编辑状态。比如，如果有些代码文件与应用有时会弹出的登录界面相关，那么你可以将其放到 Login 分组中。如果应用包含了一些声音文件，那就可以将其放到 Sounds 分组中，诸如此类。

Products 分组及其内容并不对应于项目目录中的任何一项。Xcode 会生成对可执行包的引用（通过构建项目中的每个目标生成），按照惯例，这些引用会出现在 Products 分组中。

另一个便捷的分组是 Frameworks 分组，它会列出代码所依赖的框架。代码会依赖一些框架，但在默认情况下，这些框架并不会出现在项目导航器中，项目导航器中没有 Frameworks 分组，因为这些框架并没有显式链接到构建中；相反，代码会使用模块，这意味着文件顶部的 import 语句就可以隐式链接了。不过，如果显式链接到了某个框架，那么它就会列在项目导航器中；接下来，你会创建一个 Frameworks 分组，将这些框架放到该分组中。本章后面将会对框架进行介绍。

## 6.4 目标

所谓目标就是规则与设置的集合，这些规则与设置用于指定如何构建产品。在构建时，你所构建的实际上是个目标（可能还会有多个目标）。

在项目导航器顶部选中 Empty Window 项目，你会在编辑器左侧看到两部分内容（如图 6-7 所示）：项目本身以及目标列表。Empty Window 项目有一个目标：应用目标，叫作 Empty Window（就像项目本身一样）。应用目标是用于构建和运行应用的目标。其设置会告诉 Xcode 如何构建应用；其产品就是应用本身。

在某些情况下，你可能会向项目添加更多的目标：

- 你想要执行单元测试或界面测试；为了做到这一点，你会添加一个目标。（第 9 章将会对测试进行更多的介绍。）
- 你想要编写一个框架并作为自己的 iOS 应用的一部分；借助自定义框架，你可以将共同代码重构到一处，可以通过命名空间来重构其私有性细节信息。自定义框架是需要构建的，因此它也是个目标。（本章后面将会对框架进行更多的介绍。）
- 你想要编写应用扩展，比如，今日扩展（出现在通知中心的内容），或图片编辑扩展（出现在照片应用上的图片编辑界面）。它们也都是目标。

项目名与目标列表可以通过两种方式呈现（如图 6-7 所示）：一种是作为列显示在编辑器左侧；如果将该列收起以节省空间，那么它还可以作为弹出菜单显示在编辑器左上角。如果在列或弹出菜单中选中了项目，那就可以编辑项目；如果选中了某个目标，那就可以编辑该目标。我会在后面的表述中使用这种说法。

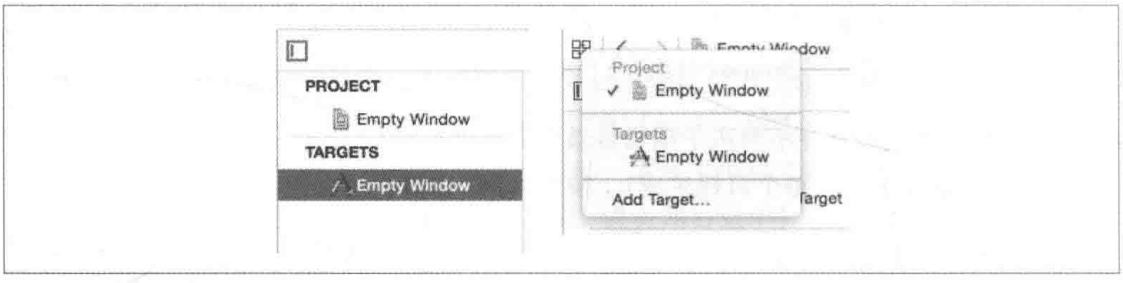


图 6-7：展示项目与目标的两种方式

## 6.4.1 构建阶段

编辑应用目标并单击编辑器顶部的 Build Phases（如图 6-8 所示），这些是应用的构建阶段。构建阶段既描述了目标的构建方式，也包含了一套指令，指示 Xcode 该如何构建目标；如果修改了构建阶段，那么构建过程也会随之修改。单击每个构建阶段可以查看该构建阶段所应用的目标中的文件列表。

有两个构建阶段是有内容的。这些构建阶段的含义也是一目了然的：

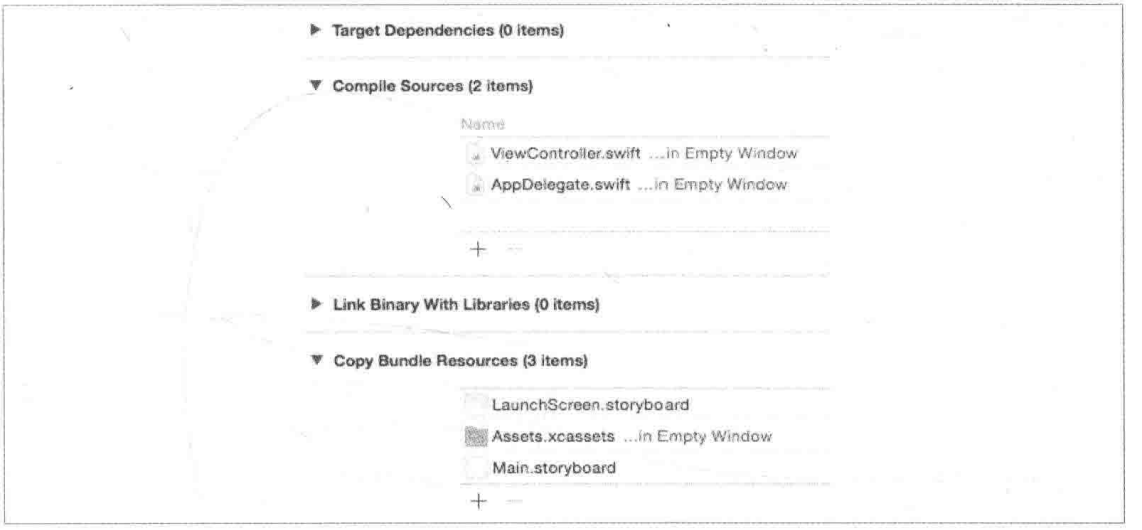


图 6-8：应用目标的构建阶段

*编译源*

会编译某些文件（代码），生成的编译代码会被复制到应用中。

该构建阶段通常会作用于所有目标的 *.swift* 文件上；它们是构成目标的代码文件。显然，它会包含 *ViewController.swift* 与 *AppDelegate.swift*。如果向项目添加了新的 Swift 文件（通常是为了声明另一个类），那么你应该将其指定为应用目标的一部分，它会自动添加到编译源构建阶段中。

*复制包资源*

某些文件会被复制到应用中，这样在应用运行时，代码或系统就能找到它们了。

该构建阶段目前会作用于资源目录上；添加到资源目录中的任何资源都会被复制到应用中作为目录的一部分。它还会列出启动 storyboard 文件 *LaunchScreen.storyboard* 与应用的界面 storyboard 文件 *Main.storyboard* 文件。

复制并不意味着进行完全的复制。某些类型的文件在复制到应用包中时会通过几种方式被特殊对待。比如，复制资源目录意味着目录中的图标会被写到应用包的顶层，资源目录本身会被转换为 *.car* 文件；复制 *.storyboard* 文件意味着它会被转换为 *.storyboardc* 文件，该文件本身是个包含 nib 文件的包。

你可以手工修改这些列表，有时也需要这么做。比如，如果项目中的某些文件（如声音文件）不在复制包资源中，但你又想在构建过程中将其复制到应用中，那么你可以将其从项目导航器中拖曳到复制包资源列表中，或（更加简单的方式）单击复制包资源列表下方的“+”按钮来弹出一个对话框，该对话框会列出项目中的所有内容。相反，如果项目中的某些文件位于复制包资源中，但你又不想将其复制到应用中，那么你可以从列

表中将其删除；这么做并不会将其从项目、项目导航器或 Finder 中删除，而只会从复制到应用中的条目列表中删除。

有时需要修改 Link Binary With Libraries 构建阶段，某些库（通常是框架）会链接到编译后的代码上（编译之后叫作库），这会告诉代码，当应用运行时，设备上需要这些库。Empty Window 项目会链接到一些框架，不过它并未使用该构建阶段；相反，它将框架导入为模块，框架就会自动链接。不过，在某些情况下，你需要显式将二进制文件链接到额外的框架上；本章后面将会对此进行介绍。

一个实用的技巧是添加 Run Script 构建阶段，它会在构建过程中运行一个自定义 shell 脚本。要想做到这一点，请选择 Editor → Add Build Phase → Add Run Script Build Phase。打开新添加的 Run Script 构建阶段可以编辑自定义 shell 脚本。最简单的一个 shell 脚本如下所示：

```
echo "Running the Run Script build phase"
```

勾上“Show environment variables in build log”复选框会在 Run Script 构建阶段的构建报告中列出构建过程中的环境变量及其值。单凭这一点我们就应该加上 Run Script 构建阶段；通过查看环境变量可以了解到关于构建过程如何进行的很多信息。

## 6.4.2 构建设置

构建阶段只是目标了解如何构建项目的一个方面。另外一个方面就是构建设置。要想查看构建设置，请编辑目标并在编辑器顶部单击构建设置（如图 6-9 所示）。你会看到一个长长的设置列表，其中大部分设置都不用修改。Xcode 会检查该列表以便了解在构建过程的各个阶段应该做什么。项目之所以会按照期望的方式编译和构建，完全是因为构建设置在起作用。

图 6-9：目标构建设置

你可以通过单击 Basic 或 All 来显示不同的构建设置。设置会被组合为类别，你可以关闭或打开每个类别标题以节省空间。如果了解想要查看的某个设置，如它的名字，那么你就可以通过右上角的搜索框来过滤显示的设置。

你可以通过单击 Combined 或 Levels 来决定构建设置的显示方式；在图 6-9 中，我单击了 Levels，目的是介绍何谓 Levels。不仅目标包含了构建设置的值，项目也包含了相同构建设置的值；此外，Xcode 拥有一些内建的默认构建设置值。Levels 会同时显示出这些层级，这样你就能清楚每个构建设置的实际值的来源了。

要想理解这张图，请从右向左看。比如，Build Active Architecture Only 设置的 Debug 配置（最右侧）默认为 No。不过，项目中将其设为了 Yes（右侧第 2 列）。目标并未修改该设置（右边第 3 列），因此结果就是设置被解析为了 Yes（右侧第 4 列）。

如果想要改变其值，你可以现在就改。你可以在项目层次或目标层次上修改其值。我并不建议你这么做；事实上，你很少会直接操纵构建设置，因为通常情况下，默认值就可以了。然而，你还是可以修改构建设置值的，就在这里修改。要想了解各种构建设置的详细信息，请参考 Apple 的文档，特别是 Xcode 构建设置参考文档。此外，你还可以选择某个构建设置，然后在辅助窗格中显示快速帮助以了解更多信息。要想了解各种构建设置的详细信息，请参考 Apple 的文档，特别是 Xcode 构建设置参考文档。

## 6.4.3 配置

实际上，构建设置值会有多个列表，但在执行构建时只有一个列表会发挥作用。每个列表都叫作一个配置。我们需要多个配置，因为你会在不同的时间针对不同的目的采用不同的构建方式，这样就需要某些构建设置在不同的情况下接受不同的值。

默认情况下，有如下两个配置：

*调试*

该配置用于开发过程，也就是编写和运行应用的时候。

*发布*

该配置用于后期的测试，也就是在设备上检查性能，以及将应用打包提交到 App Store 的时候。

之所以需要配置完全是因为项目的需要。要想查看项目中的配置，请编辑项目并单击编辑器顶部的 Info（如图 6-10 所示）。注意到这些配置仅仅是名字而已。你可以添加更多的配置，这只不过是向名字列表中添加名字而已。配置的重要性只在这些名字与构建设置值关联起来时才会显现出来。配置会在项目与目标级别上影响构建设置值。

图 6-10：配置

比如，回到目标构建设置（如图 6-9 所示）并在搜索框中输入“Optim”。你会看到 Optimization Level 构建设置（如图 6-11 所示）。Optimization Level 的调试配置值是 None：在开发应用时，你会使用调试配置来构建，这样代码就会以一种直观的方式一行一行地进行编译。Optimization Level 的发布配置值是 Fast；当应用准备发布时，你会使用发布配置进行构建，这样生成二进制文件时就会针对速度进行优化，非常适合于用户在设备上运行应用，但却不适合开发，因为调试器中的断点与步进将无法使用。

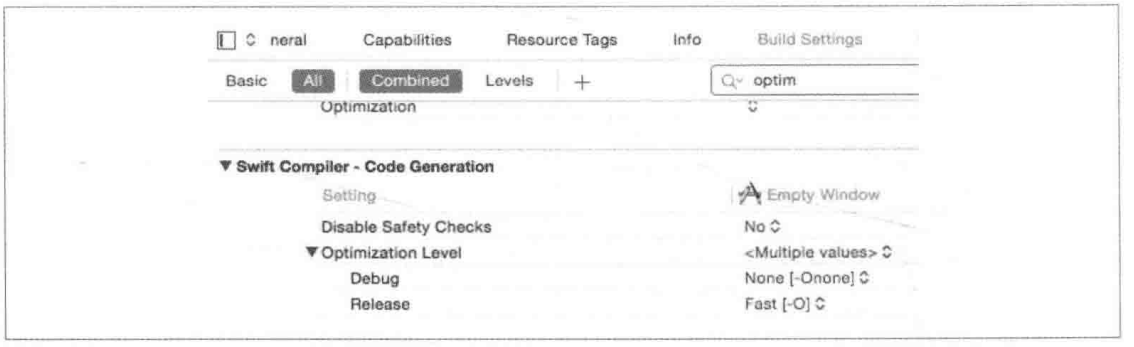


图 6-11：配置是如何影响构建设置的

对于发布配置 Optimization Level 设置，更好的选择是 Fast, Whole Module Optimization。这样，Swift 编译器就会立刻检查所有代码文件。编译时间可能会长一些，不过优化结果会更好；比如，编译器可能会推断出某些类成员无须动态分派，这会加快代码的运行速度。

## 6.4.4 方案与目标

到目前为止，我还没有介绍在某次构建过程中，Xcode 是如何知道配置的。这是由方案来决定的。

方案会根据构建目的将一个或多个目标与构建配置组合起来。在默认情况下，新的项目都自带一个方案，并以项目名命名。这样，Empty Window 项目的方案就叫作 Empty Window。要想查看它，请选择 Product → Scheme → Edit Scheme，这会打开模式编辑器对话框（如图 6-12 所示）。

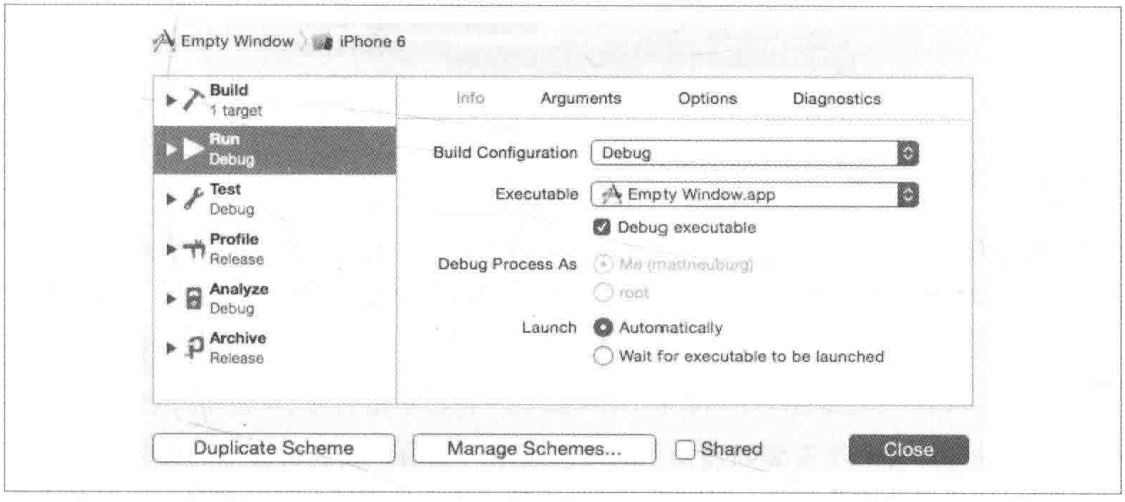


图 6-12：方案编辑器

方案编辑器左边列出的是你可以从 Product 菜单中执行的各种动作。单击某个动作可以在该方案中查看到其相应的设置。

第 1 个动作构建动作与其他动作不同，因为其他动作都会用到构建动作，其他动作都会隐式涉及构建。构建动作只是在其他动作执行时用来决定构建哪些目标。对于我们的项目来说，它意味着无论执行的动作是什么，应用目标总是会被构建。

第 2 个动作（运行动作）决定了构建和运行时所需的设置。构建配置弹出菜单（在信息窗格中）被设为了调试。这说明了当前的构建配置的来源：现在，在构建和运行时（Product → Run，或单击工具栏中的 Run 按钮），你使用的是调试构建配置和与其相对应的构建设置值，因为你使用的是该方案，这正是构建与运行时该方案所要做的事情。

你可以编辑已有的方案，不过一般来说不需要这么做。另一种可能是创建新的方案。一种方式是从项目窗口工具栏的Scheme弹出菜单中选择Manage Schemes（如图6-13所示）。

方案弹出菜单是经常会用到的一个功能。所有方案都会在此列出，因此在构建和运行前可以轻松在各个方案间切换。目标（Destination）会以层次方式附加到每个方案上。目标就是运行应用的机器。比如，你可能会在物理设备或模拟器中运行应用。如果是模拟器，那就需要指定要模拟的特定类型的设备。可以在方案弹出菜单中选择目标。

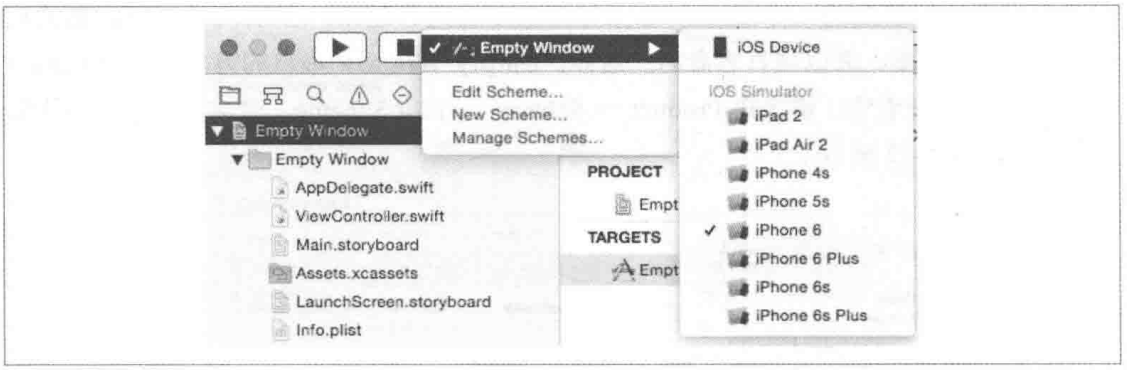


图 6-13：方案弹出菜单

目标与方案之间并没有什么关系。方案弹出菜单中会有目标主要是起到方便的作用，这样你就可以使用弹出菜单一次性地来选择方案或目标了，也可以同时选择这两者。要想在不改变方案的情况下切换目标，请在方案弹出菜单中单击目标名。要想切换方案，或确定目标（如图 6-13 所示），请在方案弹出菜单中单击方案名。

每个模拟设备都有一个安装到设备上的系统版本。目前，我们所模拟的设备都运行着 iOS 9.0；这样就没有差别了，系统版本就没有显示出来。不过，可以在 Xcode 的首选项窗格中下载其他 SDK（系统）。如果下载了并且应用可以运行在多个系统版本上，你还会在 Scheme 弹出菜单中看到系统版本成为目标名的一部分。比如，如果安装了 iOS 8.4 SDK，同时项目的部署目标（参见第 9 章）是 8.0，那么项目窗口工具栏中的方案弹出菜单就会在目标名后面显示“iOS 9.0”或是“iOS 8.4”。

如果下载了额外的 SDK，并且应用配置为在多个系统上运行，要是看不到使用了这些系统的任何模拟设备，那么请选择 Window → Device 来弹出 Devices 窗口。这是管理模拟设备的地方。可以在这里创建、删除和重命名模拟设备，还可以指定某个模拟设备是否会作为目标出现在方案弹出菜单中。

# 6.5 从项目到运行应用

应用文件实际上是一种特殊的目录，叫作包（*package*，而特殊的 package 则叫作 *bundle*）。通常情况下，Finder 会将包当作文件，并不会将其内容显示给用户，但你可以绕过这种防护措施并使用 Show Package Contents 命令查看应用包的内容。这样就可以了解到应用包的结构了。

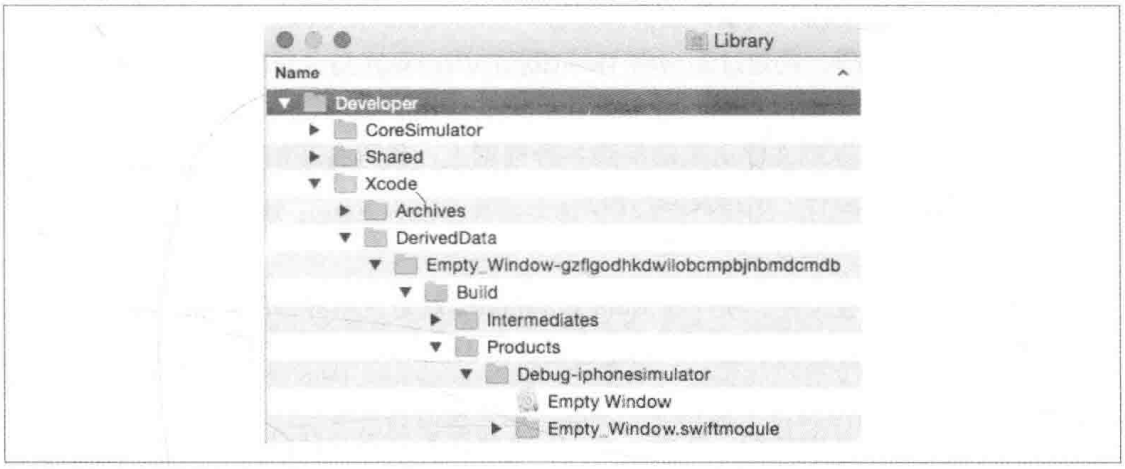


图 6-14：在 Finder 中查看构建好的应用

我们将使用之前构建的 Empty Window 应用作为示例应用来一探究竟。你需要在 Finder 中找到它；默认情况下，它应该位于用户的 *Library/Developer/Xcode/DerivedData* 目录下，如图 6-14 所示。

在 Finder 中，按下 Control 键并单击 Empty Window 应用，从上下文菜单中选择 Show Package Contents。你会看到构建过程的结果（如图 6-15 所示）。

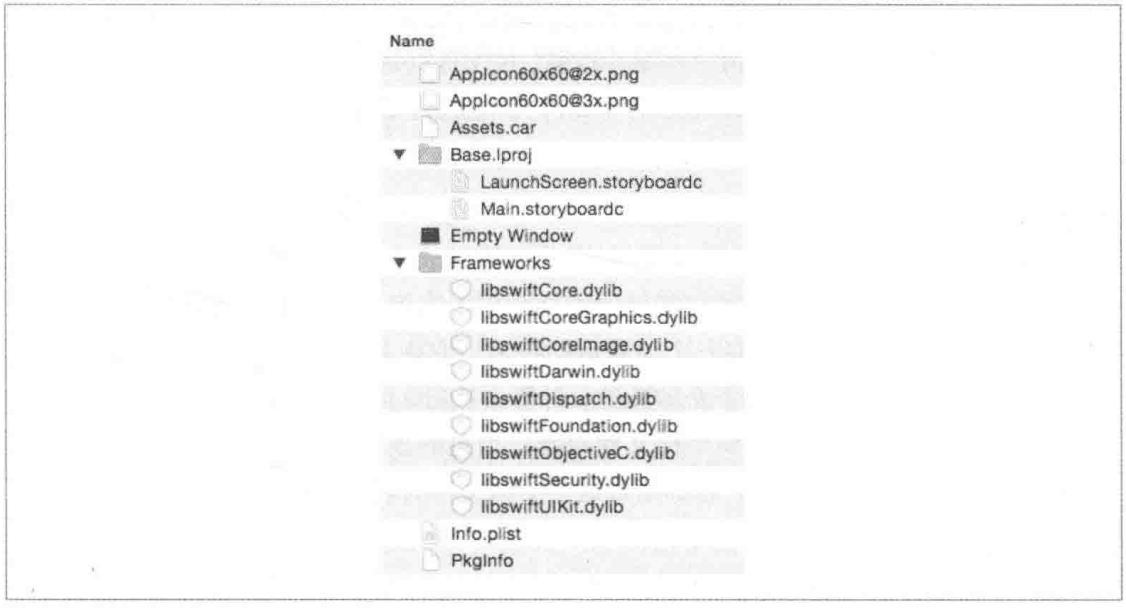


图 6-15：应用包的内容

可以将应用包看作项目目录的一种变换：

*Empty Window*

应用编译后的代码。构建过程会将 *ViewController.swift* 与 *AppDelegate.swift* 文件编译到这个文件中，即应用的二进制文件。它是应用的核心，实际执行的内容。当应用启动时，该二进制文件会被链接到各种框架上，代码会开始运行（本章后面将会详细介绍“开始运行”所涉及的东西）。

*Main.storyboardc*

应用的界面故事板文件。项目的 *Main.storyboard* 就是应用界面的来源。在该示例中，一个空白视图会占据整个窗口。构建过程会将 *Main.storyboard* 编译为更加紧凑的格式（使用 ibtool 命令行工具），即 *.storyboardc* 文件，它实际上包含了多个 nib 文件，当应用启动时会按需加载。其中一个 nib 文件会在应用启动时加载进来，它就是界面中所显示的空白视图的来源。*Main.storyboardc* 与项目目录中的 *Main.storyboard* 位于同一个子目录中（在 Base.lproj 中）；如前所述，该目录结构与本地化有关（第 9 章将会介绍）。

*LaunchScreen.storyboardc*

应用的启动界面文件。该文件（*LaunchScreen.storyboard* 的编译版本）包含了应用启动的短暂时间内所显示的界面。

*Assets.car*、*AppIcon60x60@2x.png* 与 *AppIcon60x60@3x.png*

资源目录与一对图标文件。在构建准备时，我向原来的资源目录 *Images.xcassets* 中添加了一些图标图片和其他一些图片资源。该文件处理后（使用 `actool` 命令行工具）会生成一个编译后的资源目录文件（*.car*），它包含了添加到目录中的所有资源。同时，图标文件会被写到应用包的顶层，系统会在这里寻找它们。

*Info.plist*

这是个遵循严格文本格式的配置文件（属性列表文件）。它来自于项目的 *Info.plist*，但与之并不完全相同。其包含的指令用于告诉系统该如何对待并启动应用。比如，项目的 *Info.plist* 有一个计算后的报名，它来自于产品的名字 `$(PRODUCT_NAME)`；在构建后的应用的 *Info.plist* 中，计算会执行，值会读取 `Empty Window`。此外，它还会连同资源目录一同将图标文件写到应用包的顶层，这时会向构建好的应用的 *Info.plist* 中添加一项设置，告诉系统这些图标文件的名字是什么。

*Frameworks*

有几个框架会添加到构建好的应用中。我们的应用使用了 Swift；这些框架会包含完整的 Swift 语言！应用所用的其他框架会被构建到系统中，但 Swift 不会。将 Swift 框架打包到应用包中能够允许 Apple 快速演化 Swift 语言，同时又独立于任何系统版本，并且还可以让 Swift 向后兼容老系统。其副作用就是这些框架会增加应用的大小；

不过，相比于 Swift 的强大与灵活性，这么做是值得的。（也许未来，当 Swift 语言稳定下来后，它会被构建到系统而不是每个应用中，这样 Swift 应用就会变得小一些。）

*PkgInfo*

这是一小段文本，内容是 APPL????，表示该应用的类型和创建者代码。*PkgInfo* 文件已经过时了；对于 iOS 应用的功能来说并没有什么用，并且是自动生成的。你永远不会用到它。

在实际开发中，应用包可能会包含多个文件，但差别主要还是量而不是种类的问题。比如，我们的项目可能会有额外的 *.storyboard* 或 *.xib* 文件、框架或声音文件等资源。所有这些文件都会按照自己的方式放到应用包当中。此外，在设备上运行的应用包还会包含一些安全相关的文件。

你现在应该能体会到这种项目组件的处理方式以及组装到应用中的方式所带来的好处了，同时也清楚为了确保应用能够正确构建，程序员应该做哪些事情。本节后面的内容将会介绍项目中的哪些内容会被放到应用的构建中，以及应用的构成是如何使得应用能够运行起来的。

## 6.5.1 构建设置

我们已经介绍了该如何使用构建设置。Xcode 本身、项目以及目标都可以修改最终的构建设置值，根据构建配置的不同，其中一些可能会有所不同。在构建前，你需要指定好方案；方案会决定构建配置，这样在构建时特定的构建设置值才会应用上。

## 6.5.2 属性列表设置

项目中会包含一个属性列表文件，它用于生成构建的应用的 *Info.plist* 文件。项目中的这个文件不一定非得叫作 *Info.plist*！应用目标知道该文件是什么，因为其名字位于 *Info.plist* 文件的构建设置中。比如，在我们这个项目中，应用目标的 *Info.plist* 文件构建设置值被设为了 *Empty Window/Info.plist*（看看构建设置就知道了）。

属性列表文件是个键值对的集合。你可以编辑它，有时也需要这么做。编辑项目的 *Info.plist* 主要有 3 种方式：

- 在项目导航器中选中 *Info.plist* 文件并在编辑器中对其进行编辑。在默认情况下，键名（以及一些值）会通过一些描述性信息显示，这是由其功能决定的；比如，键名可能用“Bundle name”表示而非实际的键 CFBundleName。不过可以通过在编辑器中单击，然后选择 Editor → Show Raw Keys & Values 或使用上下文菜单来查看实际的键。

此外，可以通过实际的 XML 格式来查看和编辑 *Info.plist* 文件：按住 Control 并在项目导航器中单击 *Info.plist* 文件，并从上下文菜单中选择 Open As → Source Code。（不过，以原生 XML 形式编辑 *Info.plist* 是有风险的，因为一旦出错，XML 就无效了，这会导致出现问题，但却看不到警告。）

- 编辑目标并切换至信息窗格。Custom iOS Target Properties 部分会显示出与在编辑器中编辑 *Info.plist* 时相同的信息。
- 编辑目标并切换至 General 窗格。这里的一些设置可用于编辑 *Info.plist*。比如，单击 Device Orientation 复选框会改变 *Info.plist* 中“Supported interface orientations”键的值。（这里的其他一些设置可用于编辑构建设置。比如，如果修改了 Deployment Target，那就会修改 iOS Deployment Target 构建设置的值。）

项目 *Info.plist* 中的一些值会被处理以将其转换为构建好的应用的 *Info.plist* 中的最终值。这个步骤会在构建过程后期进行。比如，项目 *Info.plist* 中的“Executable file”键值是 `$(EXECUTABLE_NAME)`；它会被 `EXECUTABLE_NAME` 构建环境变量的值所替换（可以通过 Run Script 构建阶段查看它）。此外，在处理过程中还会向 *Info.plist* 注入一些额外的键值对。

若想了解键的完整列表及其含义，请参见 Apple 的文档：*Information Property List Key Reference*。第 9 章将会介绍 *Info.plist* 中你很有可能会修改的一些设置。

## 6.5.3 nib 文件

nib 文件是以一种编译好的格式对用户界面的描述，它包含在一个扩展名为 *.nib* 的文件中。你所编写的每个应用都至少包含一个 nib 文件。通过在 Xcode 中以图形化方式编辑 *.storyboard* 或 *.xib* 文件来准备这些 nib 文件；实际上，你正在设计一些对象，这些对象会在应用运行以及 nib 文件加载时实例化。

nib 文件是在构建过程中生成的，要么通过编辑 *.xib* 文件（使用 `ibtool` 命令行工具），这会生成一个 nib 文件；要么通过编辑 *.storyboard* 文件，这会生成包含了多个 nib 文件的 *.storyboardc* 包。编辑是对 *.storyboard* 或 *.xib* 文件进行的，它们位于应用目标的 Copy Bundle Resources 构建阶段中。

Single View Application 模板所生成的 Empty Window 项目包含了一个名为 *Main.storyboard* 的界面 *.storyboard* 文件。这个文件会被特殊对待，它是应用的主故事板；之所以是这样并非因为它的名字，而是因为它被 *Info.plist* 文件中的键“Main storyboard file base name”(`UIMainStoryboardFile`)所指向；使用其名字(“Main”)并去掉 *.storyboard* 扩展来编辑 *Info.plist* 文件就会看到了！结果是，当应用启动时，从 *.storyboard* 文件中所生成的第 1 个 nib 会自动加载以帮助创建应用的初始界面。

本章后面将会详细介绍应用启动过程与主故事板。请参考第 7 章以了解关于编辑 *.storyboard* 与 *.xib* 文件，以及代码运行时它们是如何创建实例的更多信息的。

## 6.5.4 其他资源

资源是嵌入应用包中的辅助文件，当应用启动时会根据需要获取，比如，想要展示的图片或想要播放的声音等。实际应用很可能会包含多个附加资源。当应用运行时确保这些资源可用通常取决于你的代码（或加载 nib 文件的代码）：基本上，运行时只是进入应用包中，然后拉取所需的资源。实际上，应用包会被看作充满了很多东西的目录。

可以在两个地方向项目添加资源，这对应于两个不同的位置，都位于应用包当中：

*项目导航器*

如果向项目导航器添加了资源，那么还要确保它会出现在 Copy Bundle Resources 构建阶段中，它会被构建过程复制到应用包的顶层。在图 6-15 中，它与图标图像文件处于同一层级，如 *AppIcon60x60@2x.png*。

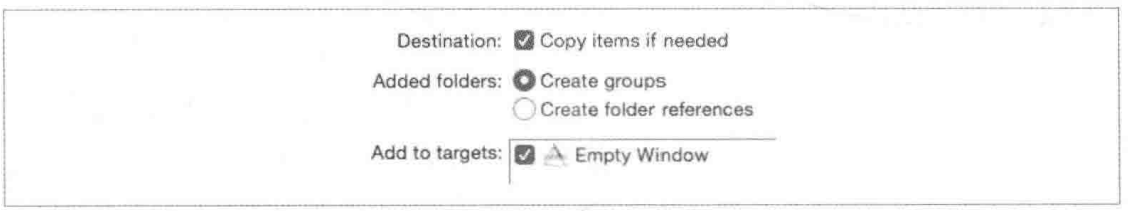


图 6-16：向项目添加资源时的选项

*资源目录*

如果向资源目录添加了资源，那么当构建过程向应用包的顶层复制并编译资源目录时（就像图 6-15 中的 *Assets.car*），资源就会位于其中。

后面将会介绍这两种向项目添加资源的方式。

### 1. 项目导航器中的资源

要想通过项目导航器向项目添加资源，请选择 File → Add Files to [Project]；还可以将资源从 Finder 拖曳到项目导航器中。无论哪种方式都会出现一个对话框（如图 6-16 所示），你可以做如下设置：

*目标*

你几乎总是应该勾上这个复选框（“Copy items if needed”）。这么做会将资源复制到项目目录中。如果不勾选这个复选框，那么项目就会依赖于项目目录外的文件，你可能会不小心删除或修改它。请将项目所需的一切文件放到项目目录中。

*添加目录*

只有向项目中添加的是目录时该选项才会起作用；区别在于项目引用目录内容的方式：

*创建分组*

目录名会成为项目导航器中普通分组的名字；目录内容会出现在该分组中，不过它们会列在 Copy Bundle Resources 构建阶段中，这样在默认情况下，它们都会被复制到应用包的顶层。

*创建目录引用*

在项目导航器中，目录显示为蓝色（一个目录引用）；此外，它会作为目录显示在 Copy Bundle Resources 构建阶段中，这意味着构建过程会将整个目录及其内容复制到应用包中。目录中的所有资源都不会位于应用包的顶层，而是在一个子目录中。如果有很多资源，并且想要对其分门别类（而非将所有资源都放在应用包的顶层），或目录层次对于应用是有意义的，那么这种布局就很有价值了。这种布局的副作用就是你所编写的用于访问资源的代码将会特定于包含该资源的目录的子目录。

*添加到目标*

选中该复选框会将资源添加到目标的 Copy Bundle Resources 构建阶段中。这样，大多数情况下都需要针对应用目标将其选中；为何需要将资源添加到项目中呢？如果不小心未选中该复选框，稍后发现项目导航器中所列出的资源需要针对某个特定的目标被添加到 Copy Bundle Resources 构建阶段中，那么你可以手工添加，具体做法如前所述。

## 2. 资源目录中的资源

在 Xcode 7 之前，资源目录只是用于图片文件。其他资源（如音频文件等）只能在项目导航器中添加。在 Xcode 7 中，资源目录可以包含任何种类的数据文件。还可以通过资源目录指定一个资源的不同版本以提供给不同硬件配置，比如，设备的屏幕分辨率（对于图片），或 iPhone 与 iPad（对于任何类型的资源）。

对于图片文件来说，资源目录可以帮助你轻松区分图像文件名特殊约定上的差别。比如，由于 iOS 9 可以运行在一倍分辨率、两倍分辨率及三倍分辨率的设备上，因此需要为每个图片都提供 3 种尺寸。为了能够与框架的图片加载方法协同工作，这种资源都使用了特殊的命名约定：比如，*listen.png*、*listen@2x.png* 与 *listen@3x.png*。项目导航器中图片文件数量的增长速度会非常快，也很容易出错。资源目录就是为了缓解这个问题而出现的。

相对于在添加到项目时手工单调地命名 *listen.png* 文件的多个版本，我可以让资源目录帮助我。编辑资源目录，单击第 1 列底部的 + 按钮，选择 New Image Set。结果是一个名为 Image 的图片集，它带有 3 个不同尺寸的图片。我将图片从 Finder 拖曳到恰当的地方。原始图片文件名并不重要！图片会被自动复制到项目目录中（位于资源目录中），无须指定这些图片文件的目标成员，因为它们都是资源目录的一部分，已经拥有了正确的目标成员。我可以重命名图片集，将其改为更具描述性的名字，比如，叫它 *listen*。结果是代码现在可以针对当前的屏幕分辨率加载正确的图片，方式是通过 `"listen"` 引用它，不用管图片的原始名或扩展是什么。

可以通过选中一张图片，然后使用属性查看器（Command-Option-4）来查看资源目录中的图片。这会显示出原始名与图片的像素大小（这一点更为重要）。

在 Xcode 7 中，类似的处理过程也适用于其他类型的资源。假设我想要向应用包中添加一个名为 *Theme.mp3* 的音频文件。我会编辑资源目录，单击 + 按钮，并选择 New Data Set。这时，一个名为 Data 的数据集会出现，它有一个 Universal 位置，我现在就可以将音频文件拖曳进去了。我对数据集进行了重命名（改为了 *theme*）；现在，我的代码可以通过名字 `"theme"` 来访问该资源了（通过 iOS 9 新增的 NSDataAsset 类）。

此外，资源目录中的目录可用于提供命名空间：比如，如果 *theme* 数据集位于名为 *music* 的资源目录中，如果对该目录勾选上了 Provides Namespace in the Attributes inspector 选项，那么就可以通过名字 `"music/theme"` 来访问该数据集了。

这样，组织就可以通过资源目录约定来使用它们了，而不会因为资源文件搞乱项目导航器与应用包的顶层结构。

在 Xcode 7 中，应用中的资源可以存储在 Apple 的服务器上而不必添加到用户从 App Store 所下载的应用包当中了。代码随后可以在后台下载用户所需的任何资源，并且在不需要时还可以清除。请访问 Apple 的 *On-Demand Resources Guide* 了解更多信息。

## 6.5.5 代码文件与应用启动过程

构建过程知道要编译什么代码文件以形成应用的二进制文件，这是因为它们都在应用目标的 Compile Sources 构建阶段中。对于 Empty Window 项目来说，它们是 *ViewController.swift* 与 *AppDelegate.swift*。随着应用开发的进行，你可能会不断向项目

添加代码文件，这时要确保它们是目标的一部分，并且位于 Compile Sources 构建阶段中。经常出现的情况是你想向代码中添加新的对象类型声明；这通常是通过向项目添加新文件来实现的，因为这会使得对象类型声明很容易就能找到，而且 Swift 的私有性依赖于将代码隔离到不同的文件中（参见第 5 章）。

在通过 File → New → File 新建文件时，你可以指定 Cocoa Touch Class 模板或 Swift File 模板。Swift File 模板只不过是个空白文件而已：它仅仅导入了 Foundation 框架而已。如果你希望继承某个 Cocoa 类，那么 Cocoa Touch Class 模板通常就更为适合了；Xcode 会帮你生成初始的类声明，对于某些常见的父类继承来说，如 UIViewController 与 UITableViewController，它甚至提供了一些类方法的桩声明。

当应用启动时，系统知道在应用包的何处寻找二进制文件，因为应用包的 *Info.plist* 文件有个“Executable file”键（`CFBundleExecutable`），其值是二进制文件的文件名；在默认情况下，二进制文件名来自于 `EXECUTABLE_NAME` 环境变量（如“Empty Window”）。

### 1. 入口点

应用启动过程中最为复杂的部分就是开始。找到并加载了二进制后，系统必须要调用它，但在哪里调用呢？如果应用是个 Objective-C 程序，那么答案就是显而易见的。Objective-C 是 C，因此入口点就是 `main` 函数。我们的项目有个 *main.m* 文件，它包含了 `main` 函数，如以下代码所示：

```
int main(int argc, char *argv[]) {
    @autoreleasepool {
        return UIApplicationMain(argc, argv, nil,
            NSStringFromClass([AppDelegate class]));
    }
}
```

`main` 函数只做了两件事：

- 创建了内存管理环境：`@autoreleasepool` 与后面的花括号。
- 它会调用 `UIApplicationMain` 函数，该函数做了很多事情，它会让应用启动并运行。

不过，我们的应用是个 Swift 程序，它没有 `main` 函数！相反，Swift 有个特殊的特性：`@UIApplicationMain`。查看 *AppDelegate.swift* 文件，你会看到这个特性，它位于 AppDelegate 类的声明之上：

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
```

该特性本质上完成了 Objective-C *main.m* 文件所做的全部事情：它会创建一个入口点并调用 `UIApplicationMain` 来启动应用。

在某些情况下，你可能想要移除 @UIApplicationMain 特性并替换为一个 *main* 文件，没问题。文件可以是 Objective-C 文件或 Swift 文件。假设是个 Swift 文件。你可以创建一个 *main.swift* 文件并确保将其添加到应用委托中。其名字很重要，因为名为 *main.swift* 的文件会得到特殊对待：可以将可执行代码放到文件的顶层。文件中应该包含与 Objective-C 对 UIApplicationMain 的调用相当的代码：

```
import UIKit
UIApplicationMain(
    Process.argc, Process.unsafeArgv, nil, NSStringFromClass(AppDelegate))
```

为什么要这么做呢？大概是因为你想要在 *main.swift* 文件中做些其他事情，或想要定制对 UIApplicationMain 的调用。

## 2. UIApplicationMain

无论是编写自己的 *main.swift* 文件还是使用 Swift @UIApplicationMain 特性都会调用 UIApplicationMain。这个函数调用是应用要做的一件重要事情。整个应用除了调用 UIApplicationMain，就没别的了！此外，UIApplicationMain 负责解决应用运行时会遇到的一些棘手问题。应用在何处创建最初的实例呢？一开始会调用这些实例上的哪些实例方法呢？应用的初始界面来自于何处？下面来看看 UIApplicationMain 到底做了哪些事情：

1. UIApplicationMain 会创建应用的首个实例，即共享的应用实例，随后可以通过调用 UIApplication.sharedApplication() 来访问该实例。对 UIApplicationMain 调用的第 3 个参数（一个字符串）指定了该共享应用实例是哪个类的实例。如果该参数为 nil（通常情况下均如此），那么默认类就是 UIApplication。如果不为 nil，那就需要继承 UIApplication，这时需要将子类替换为一个显式的值，比如，NSStringFromClass（MyApplicationSubclass，取决于调用的子类），并将其作为第 3 个参数来调用 UIApplicationMain。
2. UIApplicationMain 还会创建应用的第 2 个实例，即应用实例的委托。委托是一种重要且应用广泛的 Cocoa 模式，第 11 章将会对其进行详细介绍。它非常重要，你所编写的每个应用都会有一个应用委托实例。对 UIApplicationMain 调用的第 4 个参数（是个字符串）指定了应用委托实例是什么类。在手工版本的 *main.swift* 中，它就是 NSStringFromClass（AppDelegate）。如果使用 @UIApplicationMain 特性，那么在默认情况下，该特性会被放到 *AppDelegate.swift* 中的 AppDelegate 类声明之上；该特性表示：“这是应用委托类。”
3. 如果 *Info.plist* 文件指定了主故事板文件，那么 UIApplicationMain 就会加载它并寻找故事板的初始视图控制器（或是故事板的入口点）；它会实例化该视图控制器，

从而创建应用的第 3 个实例。对于我们的 Empty Window 项目，它由 Single View Application 模板创建，该视图控制器是名为 ViewController 的类实例；定义了该类以及 *ViewController.swift* 的代码文件也是由该模板创建的。

4. 如果有主故事板文件，那么 `UIApplicationMain` 现在就会创建应用的窗口，这是应用的第 4 个实例，即 UIWindow 的实例（或者，应用委托可以替换 UIWindow 子类的实例）。它会将该窗口实例作为应用委托的 `window` 属性；它还会将初始的视图控制器实例作为窗口实例的 `rootViewController` 属性。该视图现在是应用的根视图控制器。

5. `UIApplicationMain` 现在转向了应用委托实例并开始调用它的一些代码，如 `application:didFinishLaunchingWithOptions:`。自定义代码可以借此机会运行！`application:didFinishLaunchingWithOptions:` 就是放置用于初始化值以及执行启动任务的代码的绝佳之处；不过，请不要将耗费时间的代码放置在这里，因为这时应用的界面尚未出现。

6. 如果有主故事板，那么 `UIApplicationMain` 现在就可以让应用界面出现了。这是通过调用 UIWindow 的实例方法 `makeKeyAndVisible` 实现的。

7. 现在窗口要出现了。这反过来又会导致窗口转向根视图控制器，并告诉它获取其主视图，它会出现并占据窗口。如果视图控制器从 *.storyboard* 或 *.xib* 文件获取视图，那么相应的 nib 文件就会加载进来；其对象会被实例化和初始化，它们会成为初始界面的对象：视图会被放到窗口中，它与子视图会对用户可见。视图控制器的 `viewDidLoad` 这时也会被调用——这是你的代码提前开始运行的另一个时机。

应用现在就会启动并运行！它有一组初始实例，至少有共享应用实例、窗口、初始视图控制器与初始视图控制器的视图，以及它所包含的界面对象。你的一些代码已经开始运行：`UIApplicationMain` 依旧在运行（它永远不会返回），就在那儿，注视着用户的一举一动，维护着事件循环，而事件循环会在用户动作发生时对其做出响应。

### 3. 没有故事板的应用

在对应用启动过程的描述中，我使用几次短语“如果有主故事板”。在 Xcode 7 应用模板中，比如，用于生成 Empty Window 项目的 Single View Application 模板，有一个主故事板。不过，也可以没有主故事板。在这种情况下，如创建窗口实例、为其赋予根视图控制器、将其赋给应用委托的 `window` 属性，以及调用窗口的 `makeKeyAndVisible` 来显示界面等，都需要通过代码来实现。

为了说明这一点，请通过 Single View Application 模板新建一个 iPhone 项目，叫作 Truly Empty。然后按照如下步骤进行：

1. 编辑目标。在 General 窗格中，选择 Main Interface 域中的“Main”，然后将其删除（并按下 Tab 键使之生效）。
2. 从项目中删除 *Main.storyboard* 与 *ViewController.swift*。
3. 选中并删除 *AppDelegate.swift* 中的所有代码。

现在的项目有一个应用委托，但却没有故事板，没有代码！为了创建一个最小可运行的应用，你需要按照这种方式编辑 *AppDelegate.swift* 来重新创建 AppDelegate 类，只保留创建和显示窗口的代码，如示例 6-1 所示。

示例 6-1：没有故事板的应用委托类

```
import UIKit
@UIApplicationMain
class AppDelegate : UIResponder, UIApplicationDelegate {
    var window : UIWindow?
    func application(application: UIApplication,
        didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
        -> Bool {
            self.window = UIWindow()
            self.window!.rootViewController = UIViewController()
            self.window!.backgroundColor = UIColor.whiteColor()
            self.window!.makeKeyAndVisible()
            return true
    }
}
```

这会生成一个可运行的最小应用，它有一个空白窗口；可以通过将窗口的 `backgroundColor` 改为其他颜色（如 `UIColor.redColor()`）并再次运行应用来验证创建窗口的代码的正确性。

这是个可运行的应用，不过却没什么用。它什么都没做，也做不了，因为其根视图控制器是通用的 UIViewController。我们这里需要的是自己的视图控制器实例（包含自己的代码），可以在 nib 中配置的视图。下面来创建一个 UIViewController 子类以及包含其视图的 *.xib* 文件：

1. 选择 File → New → File。在“Choose a template”对话框中，iOS 下面，单击左侧的 Source，然后选择 Cocoa Touch Class。单击 Next。
2. 将类命名为 MyViewController，指定它为 UIViewController 的子类。勾选“Also create XIB file”复选框。指定语言为 Swift。单击 Next。
3. 这时会弹出 Save 对话框。请确保将文件保存到 Truly Empty 目录中、将 Group 弹出菜单设为 Truly Empty，并勾选 Truly Empty 目标，这些文件会成为应用目标的组成部分。单击 Create。

Xcode 已经为我们创建了两个文件：*MyViewController.swift*（将 MyViewController 作为 UIViewController 的子类）与 *MyViewController.xib*（nib 的源，MyViewController 实例会在这里获得其视图）。

4. 在 *AppDelegate.swift* 中，回到应用委托的 `application:didFinishLaunchingWithOptions:`，将根视图控制器的类修改为 MyViewController，并将其关联到 nib，如以下代码所示：

```
self.window!.rootViewController =
    MyViewController(nibName:"MyViewController", bundle:nil)
```

我们现在没有使用故事板创建了一个可用且最小的应用项目。代码完成了存在主故事板的情况下 `UIApplicationMain` 所自动完成的一些工作：我们实例化了 UIWindow，将窗口实例设为了应用委托的 `window` 属性；实例化了一个初始视图控制器，让窗口能够出现。此外，窗口的出现会自动导致 `MyViewController` 实例从 *MyViewController.xib* 编译好的 nib 中获取到其视图；这样，我们可以通过 *MyViewController.xib* 来自定义应用的初始界面。除了说明 `UIApplicationMain` 隐式所做的事情，这也是一种构建应用的合理方式。

## 6.5.6 框架与 SDK

框架就是你的代码所用的编译好的代码库。在进行 iOS 编程时，你所用的大多数框架都是 Apple 内建的框架。这些框架已经成为应用运行的设备系统的一部分了，位于 */System/Library/Frameworks* 下，但在 iPhone 或 iPad 上就没法指定了，因为我们没法（通常情况下如此）直接查看文件的层次结构。

在构建项目并在电脑上运行时，编译好的代码还需要连接到这些框架上。为了达成所愿，iOS 设备的 *System/Library/Frameworks* 在电脑上会有个副本，就在 Xcode 中。这种设备系统的副本子集称作 SDK（即“软件开发包”）。到底使用哪个 SDK 取决于构建目标是什么。

链接指的是将编译好的代码与所需框架连接起来的过程，这些框架在构建期位于一个地方，但在运行期却在另一个地方。比如：

*当构建代码以在设备上运行时*

会使用所需框架的副本。该副本位于 iPhone SDK 的 *System/Library/Frameworks* 中，它在 *Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS.sdk* 中。

*当代码在设备上运行时*

代码开始运行时会在设备顶层目录 */System/Library/Frameworks* 中查找所需框架。

通过框架这种方式，当应用运行时，Apple 的代码就会动态合并到你的应用中。框架是每个应用所需的东西的集散地；它们就是 Cocoa。内容有很多，编译好的代码也有很多。应用会共享框架的精华与功能，因为应用会链接到框架上。代码运行时就好像框架代码是其一部分一样。不过，应用只会完成很少的一部分工作；框架才是真正的能量之源。

链接会负责将编译好的代码连接到所需的框架上，不过这还不足以让代码编译通过。框架中有很多你的代码会调用的类（如 NSString）与方法（如 `rangeOfString:`）。为了满足编译器的要求，框架会在其头文件中发布 API，代码则会导入框架头文件。比如，代码可以使用 NSString 并调用 `rangeOfString:`，这是因为它导入了 *NSString* 头文件。事实上，你的代码所导入的是 UIKit 头文件，它反过来又导入了 *Foundation* 头文件，而 Foundation 又导入了 *NSString* 头文件。可以在自己代码的头文件中看到这一点：

```
import UIKit
```

按住 Command 键并单击 `UIKit` 会转到 Swift 的 *UIKit* 头文件中。文件顶部是 `import Foundation`。查看 *Foundation* 头文件并向下滚动，你会看到 `import Foundation.NSString`。查看 *NSString* 头文件，你会看到 `rangeOfString:` 方法的声明。

这样，使用框架需要两步：

*导入框架的头文件*

代码需要这个信息才能成功编译。代码会通过 `import` 关键字导入框架的头文件，从而导入框架，或导入的框架再导入所需的框架。在 Swift 中，可以通过模块名指定框架。

*链接到框架*

运行时，编译后的可执行二进制文件需要连接到所用的框架上，这会将编译后的代码与这些框架合并起来。代码构建完毕后，它会链接到所需的框架上，按照目标 Link Binary With Libraries 构建阶段所列出的框架进行。

不过，我们的项目并没有任何显式的链接。查看应用目标的 Link Binary With Libraries 构建阶段，你会发现它是空的。这是因为 Swift 使用了模块，模块可以进行自动链接。在 Objective-C 中，这两个特性都是可选的，并且由构建设置管理。不过在 Swift 中，模块与自动链接的使用则是自动的。

模块是存储在电脑 *Library/Developer/Xcode/DerivedData/ModuleCache* 中的缓存信息。仅仅打开一个 Swift 项目就会使得任何被导入的模块都缓存在那里。进入 ModuleCache 目录中，你会看到大量框架与头文件（*.pcm* 文件）所构成的模块。Swift 对模块的使用简化了导入与链接的过程，并加快了编译时间。

模块是精巧且便捷的，不过有时还需要手工链接到框架上。比如，假设你想在界面中使用 MKMapView（Map Kit View）。你可以在 *.storyboard* 或 *.xib* 文件中配置，不过当构建和运行应用时，应用会崩溃，消息是“Could not instantiate class named MKMapView.”。原因在于 nib 在加载时会发现它包含了一个 MKMapView，但却不知道 MKMapView 是什么。MKMapView 定义在 MapKit 框架中，但 nib 并不知道这一点。

在代码顶部加上 `import MapKit` 也无法解决这个问题；如果代码想要使用 MKMapView，你就需要这么做，不过这却没办法在 nib 加载时让其知道何为 MKMapView。解决办法就是手工链接到 MapKit 框架：

1. 编辑目标，查看构建阶段窗格。
2. 在 Link Binary With Libraries 下单击 + 按钮。
3. 这时会出现一个可用框架列表（还有一些动态库）。向下滚动到 *MapKit.framework*，将其选中并单击 Add。

这可以解决问题：应用现在可以构建并运行了。

你还可以创建自己的框架并作为项目的一部分。框架是个模块，因此也有助于结构化你的代码，正如第 5 章介绍 Swift 隐私性时所述。要想创建新的框架：

1. 编辑目标并选择 Editor → Add Target。
2. 在对话框左侧，iOS 下，选择 Framework & Library；在右侧，选择 Cocoa Touch Framework，单击 Next。
3. 为框架取个名字；叫它 Coolness。你可以选择语言，不过我不确定这么做是否有用，因为现在还没有创建任何代码文件。默认情况下，应用弹出菜单中的 Project and Embed 应该已经被正确设定好了，单击 Finish。

这时，项目中会创建好一个新的 Coolness 框架目标。如果向 Coolness 目标添加一个 *.swift* 文件，并在里面定义一个对象类型，将其声明为 `public`；回到一个主应用目标文件上，如 *AppDelegate.swift*，那么代码就可以 `import Coolness`，并且能够看到该对象类型及里面的公共成员了。

## 6.6 对项目内容进行重命名

创建项目时为项目所指定的名字会在项目中的很多地方使用，这导致一些初学者担心重命名项目会破坏一些东西。不过请不必担心！

首先，通常情况下你不需要对项目进行重命名。一般来说，你想要修改的是应用的名字，即用户在设备上看到的名字，与应用图标关联在一起。它并不是项目名！实际上，用户是看不到项目名的。如果你想要修改的是设备上与应用关联的那个名字，那么请在 *Info.plist* 中修改（或创建）“Bundle Display Name”。

不过还是可以对项目进行重命名的，做起来也很容易：请在项目导航器顶部选中项目列表，按下回车键即可编辑其名字、输入新的名字，然后再次按下回车键。Xcode 会弹出一个对话框，提示修改与之匹配的其他名字，包括应用目标与构建后的应用，以及各种相关的构建设置。你可以自由选择。

修改项目名（或目标名）并不会自动修改与之匹配的方案名。因为没必要这么做，不过你可以自由修改方案名：选择 Product → Manage Schemes，单击方案名即可编辑。

修改项目名（或目标名）并不会自动修改与之匹配的主分组名。因为没必要这么做，不过你可以在项目导航器中自由修改分组名，因为这些名字是任意的；它们对于构建设置或构建过程没有什么影响。不过，主分组比较特殊，因为它对应于磁盘上的真实目录，该目录位于项目目录顶层项目文件的旁边。修改分组名是没问题的，不过初学者不应该在磁盘上修改其目录名，因为它是被硬编码到几处构建设置中的。

你可以随时在 Finder 中修改项目目录名，也可以移动项目目录，因为针对项目目录中的文件与目录条目的所有构建设置首选项都是相对的。

# 第 7 章

# nib 管理

第 4 章介绍过获取实例的方式。可以直接实例化一个对象类型：

```
let v = UIView()
```

也可以获取对已有实例的引用：

```
let v = self.view.subviews[0]
```

不过还有第 3 种方式：可以加载 *nib*。*nib* 是个特殊格式的文件，文件中是一些用于创建与配置实例的指令。加载 nib 实际上就是告诉 nib 遵循这些指令：它会创建并配置这些实例。

刚才提到的 UIView 实例就适合于使用这种方式创建，因为 UIView 常常都是通过 nib 创建的。我们在 Xcode 中通过图形化界面来编辑 nib，就像绘图程序一样。其想法是设计一些界面对象（几乎都是 UIView 与 UIView 子类的实例），当应用运行时会使用到这些对象。当应用运行时，当真正开始需要这些界面对象时（通常是要在可视化界面中将其显示出来），你会加载 nib，nib 加载过程会创建并配置实例，你会接收到这些实例并将其添加到应用的界面中。

创建界面对象不必非得使用 nib。nib 加载过程中所做的事情完全可以通过代码完成。可以实例化 UIView 或 UIView 子类，配置它们，构建视图层次体系，可以将该视图层次体系添加到界面上，手工一步步完成，完全在 Xcode 中进行。nib 只不过是一种让这个过程变得更简单、更便捷的方式而已。提前以图形化方式设计好 nib；当应用运行时，代码不必实例化或配置任何视图，只需加载 nib 并获得生成的实例，然后将其放到界面中即可。实际上，由于你一定会用到视图控制器（UIViewController），它们本身在设计时

就考虑到了 nib，因此你甚至都不用使用 nib！视图控制器会加载 nib，获取生成的实例，并将它们放到界面上，这一切都是自动完成的。

相比于编写代码，nib 是一种简单且精巧的方式，它使得设计与配置应用界面的过程变得更加简单和便捷。不过，它们可能也是 iOS 编程中最不易理解的方面。很多初学者从开始学习 iOS 第一天就知道 nib，并且一直使用了很多年，但却不知道 nib 到底是什么，其工作原理是什么。这么做是完全错误的。nib 不是魔法，理解起来并不难。重要的是，你要知道 nib 是什么，其工作原理是什么，如何在代码中操纵 nib。没有完全理解 nib 会导致你陷入各种低级、混乱的问题中；而实际上，只需掌握一些基本的知识就可以完全避免或纠正这些问题。这些都是本章将要介绍的主题。

### nib 有必要吗？

从根本上来说，nib 是实例之源，你可能想问是否可以不使用 nib。这些实例也可以通过代码生成，因此完全去除 nib 不也可以吗？简单的答案就是：是的，没问题。我们完全可以编写一个没有 *.storyboard* 或 *.xib* 文件的复杂应用（我就这么干过）。不过，实际问题是如何做好平衡。大多数应用都至少会将 nib 文件作为一些界面对象之源；不过，有一些界面对象只能通过代码来定制，有时从一开始就完全通过代码来生成这些界面对象会更简单。在实际开发中，项目可能会涉及一些代码生成的界面对象与 nib 生成的界面对象（后者还可以通过代码做进一步的修改或是配置）。

名字 *nib* 或 *nib* 文件与钢笔或巧克力没有任何关系。Xcode 提供的图形化 nib 设计器（称为 *nib* 编辑器）过去（一直到 Xcode 3.2.x）是个单独的应用，叫作 Interface Builder（Xcode 中的 *nib* 编辑器环境依然被称作 Interface Builder）。Interface Builder 所创建的文件拥有 *.nib* 文件扩展名，这是“NeXTStep Interface Builder”的首字母缩写。时至今日，你在 nib 编辑器中直接编辑的文件要么是 *.storyboard* 文件，要么是 *.xib* 文件；在构建应用时，这些文件会被编译到 nib 文件中（参见第 6 章）。

## 7.1 nib 编辑器界面概览

下面探索 Xcode 的 nib 编辑器界面。第 6 章直接通过 Single View Application 模板创建了一个简单的 iPhone 项目 Empty Window；它包含了一个故事板文件，我们就来使用它。在 Xcode 中，打开 Empty Window 项目，在项目导航器中找到 *Main.storyboard*，单击进行编辑。

图 7-1 显示了选中 *Main.storyboard* 后的项目窗口（我做了一些调整，使得屏幕截图适合于图书的页面大小）。界面可以划分为 4 部分：

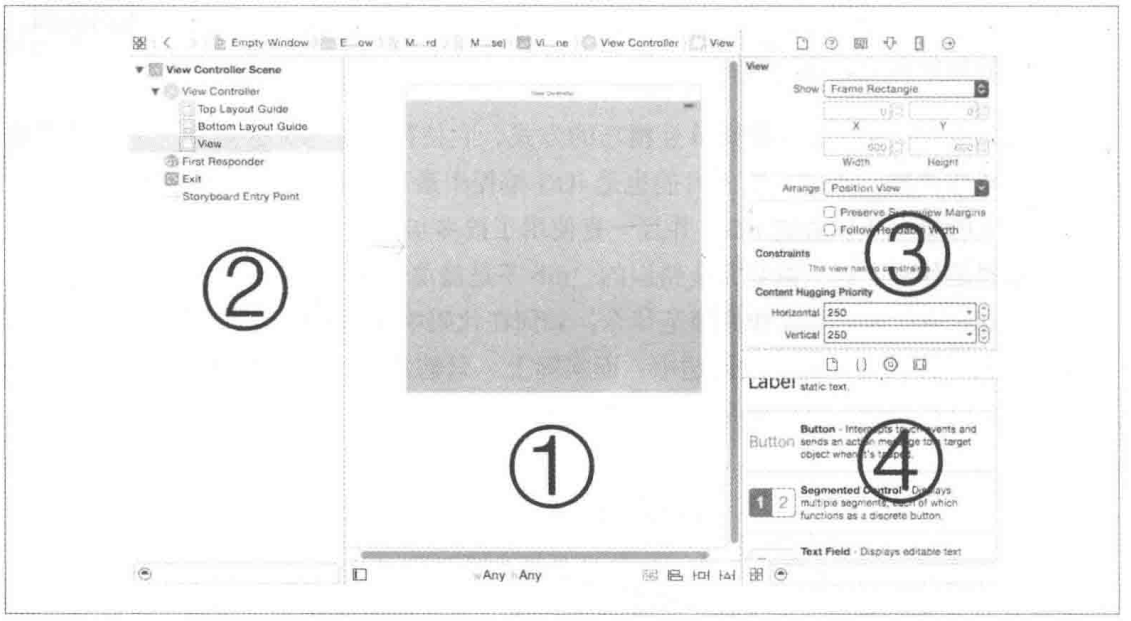


图 7-1：编辑 nib 文件

1. 编辑器的大部分是画布，这是你设计应用界面的地方。画布描绘了应用界面中的视图以及包含视图的部分。视图是个界面对象，它将自身绘制为一个矩形区域。“包含视图的部分”指的是我用于包含视图控制器的方式，虽然它们并不在应用界面中绘制，但也会展现在画布中；视图控制器并不是视图，但它有一个视图（并且可以控制它）。
2. 编辑器左侧是文档大纲，它根据名字以层次化方式列出了故事板的内容。可以通过拖曳右边缘或单击画布左下角的按钮将其隐藏。
3. 辅助窗格中的查看器是编辑当前所选对象详细信息的地方。
4. 辅助窗格中的库，特别是对象库用于将界面对象添加到 nib 中。

## 7.1.1 文档大纲

文档大纲以层次化方式描述了 nib 中对象间的关系。根据编辑的是 *.storyboard* 文件还是 *.xib* 文件，其结构会有些许的不同。

在故事板文件中，主要部分是场景。大概来说，场景指的是一个视图控制器，外加上一些辅助材料；每个场景顶层都会有一个视图控制器。

视图控制器并不是界面对象，不过它管理着一个界面对象，我们称这个界面对象为其视图（或主视图）。nib 中视图控制器的主视图未必位于与之相同的 nib 中，不过通常都在

一个 nib 中；这样，在 nib 编辑器中，视图通常都位于画布中的视图控制器内。这样在图 7-1 中，画布中大大的高亮矩形就是视图控制器的主视图，它实际上位于视图控制器中。

可以在文档大纲中查看并选取视图控制器。在场景停靠栏中，它显示为一个图标；如果选择了场景中的任何东西，那么场景停靠栏就会出现在画布中视图控制器的上方（如图 7-2 所示）。故事板文件中的每个视图控制器都构成了一个场景。这个场景在文档大纲中表示为一种层次化的名字集合。文档大纲的顶部就是场景本身。每个场景的顶部基本上是与视图控制器场景停靠栏中相同的对象：视图控制器本身以及两个代理对象 First Responder 标记与 Exit 标记。这些对象（以图标形式显示在场景停靠栏中的对象，以及位于文档大纲中场景顶层的对象）都是场景的顶层对象。

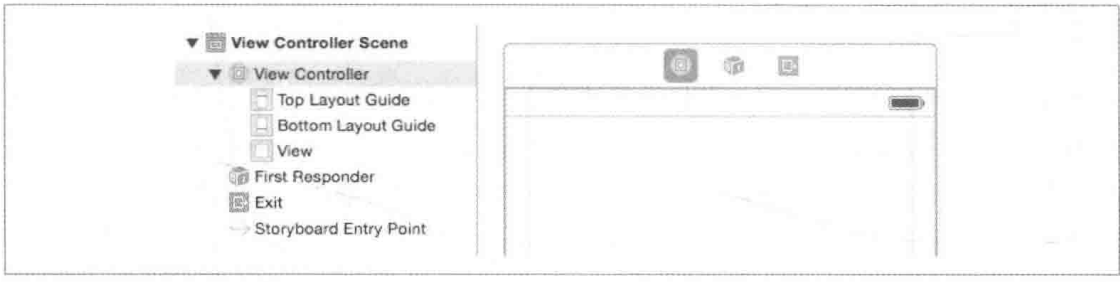


图 7-2：在故事板中选择一个视图控制器

显示在文档大纲中的对象可以分为两类：

*nib 对象*

视图控制器、主视图与我们想要放到该视图中的任何子视图是实际的对象，这些都是潜在的对象，当 nib 被运行着的应用加载时，它们会转换为实际的实例。这种从 nib 实例化的真实对象也叫作 nib 对象。

*代理对象*

加载 nib 时会实例化一些实例，不过代理对象（这里就是 First Responder 与 Exit 标记）却并不表示这些实例。相反，它们表示的是其他对象，并且有助于 nib 对象与其他对象之间的通信（本章后面将会介绍相关示例）。你不能创建或删除代理对象；nib 编辑器会自动将其显示出来。

（文档大纲中还会显示故事板入口点。它并非任何类型的对象；只是表示该视图控制器是故事板的初始视图控制器（在其属性查看器中，Is Initial View Controller 选项会被勾选上），并且对应于画布中该视图控制器左侧的向右箭头。）

故事板文档大纲中所列出的大多数 nib 对象都会按照层次依赖于场景的视图控制器。比如，在图 7-2 中，视图控制器有一个主视图；该视图会以层次方式依赖于视图控制器。

这是有意义的，因为该视图属于这个视图控制器。此外，拖曳到画布主视图中的任何其他界面对象都会列在文档大纲中，并且以层次方式依赖于视图。这也是有意义的。一个视图可以包含其他视图（其子视图），并且还可以被其他视图所包含（其父视图）。一个视图可以包含多个子视图，这些子视图本身又会包含子视图。不过每个视图都只有一个直接父视图。这样就会形成一个子视图的层次树，其父视图会包含这棵树，同时顶层会有唯一一个对象。文档大纲将这棵树表示为大纲的形式，这正是其名字的由来。

*.xib* 文件中是没有场景的。如果 *.storyboard* 文件中场景的顶层对象成为 *.xib* 文件中 nib 的顶层对象会出现什么情况呢？不要求这些顶层对象一定要是视图控制器；它可以是，不过大多数时候，*.xib* 文件的顶层界面对象都是个视图。这个视图可以作为视图控制器的主视图，不过这并非强制的。图 7-3 展示了一个 *.xib* 文件的结构，它等价于图 7-2 的单个场景。

图 7-3 中的文档大纲列出了 3 个顶层对象。其中两个是代理对象，它们在文档大纲中起到占位符的作用：File's Owner 与 First Responder。第 3 个对象是实际的对象，它是个视图；应用运行加载 nib 时会将其实例化。*.xib* 文件中的文档大纲不能完全隐藏起来；相反，它会收起为一组图标，表示 nib 的顶层对象，类似于故事板文件中的场景停靠栏，通常也叫作停靠栏（如图 7-4 所示）。

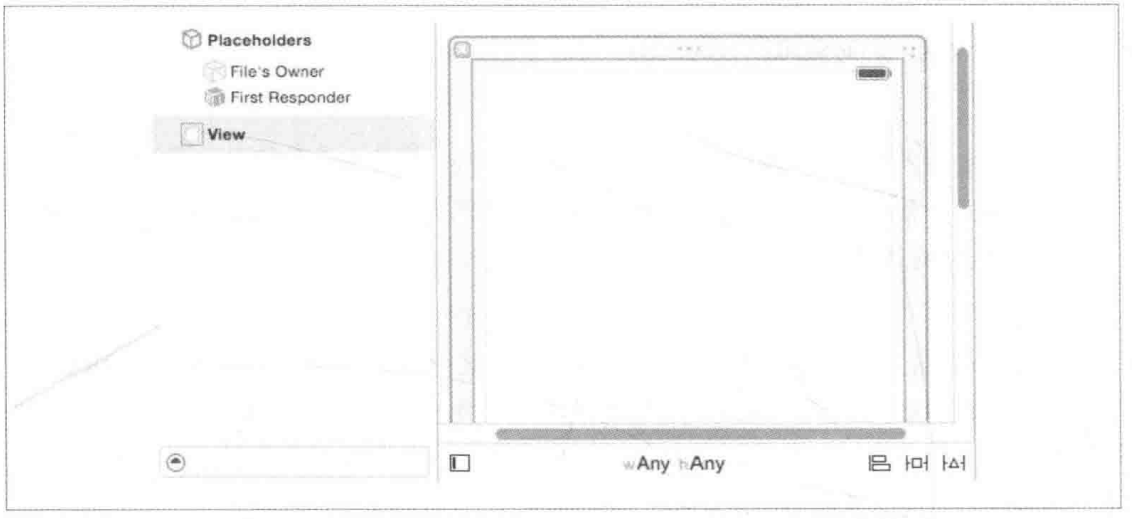


图 7-3：包含了一个视图的 .xib 文件

现在，文档大纲看起来似乎有些多余，因为其层次结构非常少；图 7-2 与图 7-3 中的所有对象都可以通过画布来访问。不过，如果故事板包含了多个场景，一个视图包含了多层对象（及其自动布局约束），这时文档大纲就很有用了，你可以通过它以非常直观的层次化结构来查看 nib 的内容，并且能够找到和选择所需的对象。此外，还可以在这里

重新整理层次结构；比如，如果错误地将某个对象作为了一个视图的子视图，那么你可以拖曳其名字在大纲中对其进行重新定位。

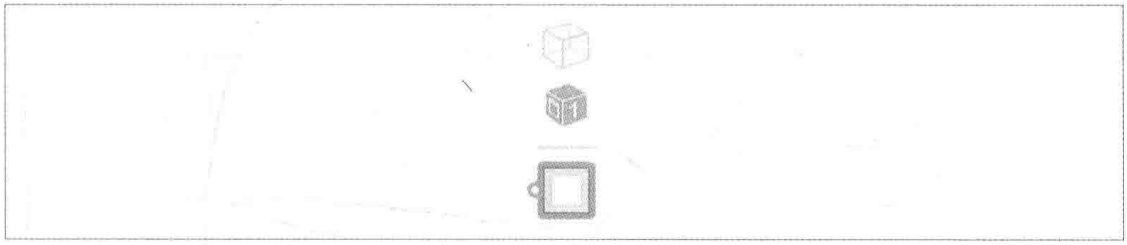

图 7-4：.xib 文件中的停靠栏

如果文档大纲中的 nib 对象名是泛泛的名字且没有给出什么有价值的信息，那么你可以修改它们。从技术上来说，名字是个标签，没什么特殊含义，可以随意为 nib 对象分配适合的标签。在文档大纲中选择一个 nib 对象的标签，按下回车键使之可以编辑；或选中该对象，然后在身份查看器的 Document 部分编辑 Label 域。

## 7.1.2 画布

画布能以图形化的方式表示出顶层的接口 nib 对象及其子视图，类似于你经常使用的绘图程序。画布是可以滚动的，并且能够自动容纳其所包含的多个图形化元素；故事板画布还可以缩放大小（选择 Editor → Canvas → Zoom 或使用上下文菜单）。

（在 *.xib* 文件中，可以在不删除对象的情况下删除顶层 nib 对象的画布表示，方式是单击左上角的 X，如图 7-3 所示。还可以在文档大纲中单击 nib 对象来恢复画布的图形化表示。）

这个简单的 Empty Window 项目的 *Main.storyboard* 只包含一个场景，因此它在画布中只会以图形化方式表示一个顶层的 nib 对象，即场景的视图控制器。视图控制器中是其主视图，通常无法将其与画布中的表示区分开来。当应用运行时，该视图控制器会成为应用窗口的根视图控制器；因此，其视图会占据整个窗口，实际上会成为应用的初始界面（参见第 6 章）。可以在这里做一些尝试：我们在该视图中所做的任何修改都会在随后构建并运行应用后显现出来。为了证明这一点，下面添加一个子视图：

1. 从图 7-1 所示的 nib 编辑器开始。
2. 打开对象库（Command-Option-Control-3）。如果显示为图标视图（没有文本的图标网格），请单击过滤栏左侧的按钮将其显示为列表视图。单击过滤栏（或选择 Edit → Filter → Filter in Library，Command-Option-L）并输入“button”，这样列表中只会显示出按钮对象。Button 对象会列在最上面。

3. 将 Button 对象从对象库拖曳到画布中视图控制器的主视图上（如图 7-5 所示），松开鼠标。

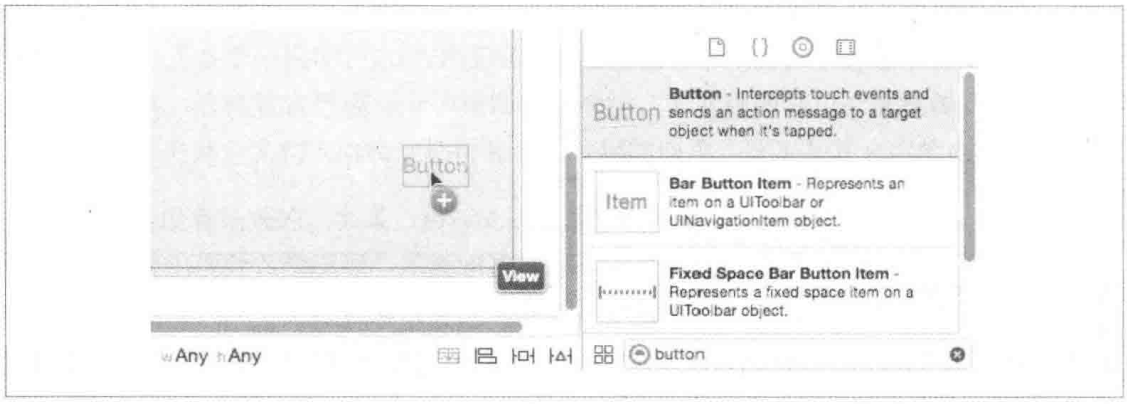


图 7-5：将一个按钮拖曳到视图中

现在按钮会出现在画布中的窗口上。我们所执行的动作（从对象库拖曳到画布上）是非常典型的一个动作；在设计界面时经常会这么做。

就像绘图程序一样，nib 编辑器可以帮助你设计界面。下面是一些典型使用场景：

- 选中按钮：修改大小处理器会出现（如果不小心选中了两次，修改大小处理器就会消失，请再次选中视图，然后选中按钮）。
- 使用修改大小处理器，让按钮变得宽一些：尺寸信息会出现。
- 将按钮拖曳到视图边缘：会出现一个指示，展示出标准的间隔。与之类似，将按钮拖曳到视图中心附近，当按钮居中时，指示会告诉你。
- 选中按钮后，按下 Option 键并将鼠标悬浮在按钮外面：箭头与数字会出现，展示出按钮与视图边缘之间的距离。（如果在按下 Option 键时不小心单击或拖曳了，你就会看到两个按钮。这是因为按住 Option 并拖曳对象会将对象复制出来。选中不想要的按钮，按下 Delete 键将其删除。）
- 按住 Control 与 Shift 键并单击按钮：会出现一个菜单，可以通过它选择按钮或按钮下面的对象（在该示例中就是视图与视图控制器，因为视图控制器是一切的顶层背景）。
- 双击按钮标题。标题就变成可编辑的了。指定好一个新标题，如“Howdy!”。按下回车键来设置新的标题。

现在来验证刚才的设计，我们来运行应用：

1. 将按钮拖曳到画布左上角附近（如果不这么做，那么当应用运行时按钮可能就会脱离屏幕）。
2. 查看 Debug → Activate / Deactivate Breakpoints 菜单项。如果显示的是 Deactivate Breakpoints，那就请选择它；我们不希望应用运行时在之前章节所创建的断点处暂停下来。
3. 确保方案弹出菜单中的目标是 iPhone 6。
4. 选择 Product → Run（或单击工具栏上的 Run 按钮）。

暂停一段时间后，iOS 模拟器会打开，这个窗口不再是空的了（如图 7-6 所示）；它包含了一个按钮！你可以使用鼠标单击该按钮，模拟用户手指的操作；在单击时按钮会高亮显示。

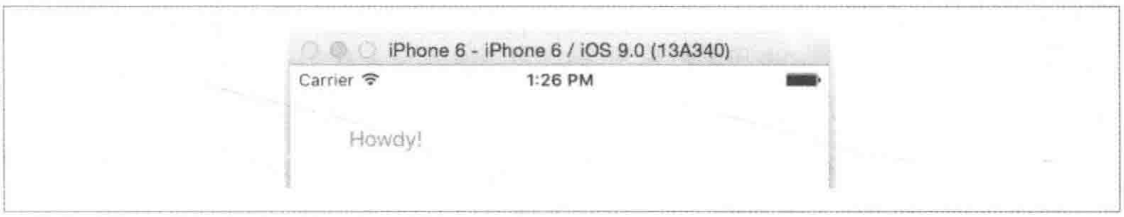


图 7-6：Empty Window 应用的窗口不再是空白的了

## 7.1.3 查看器与库

有 4 个查看器会与 nib 编辑器一同出现，并且会作用于在文档大纲、停靠栏或画布中所选择的对象上：

*身份查看器（Command-Option-3）*

该查看器的第一部分 Custom Class 是最为重要的。可以在这里查看并修改所选对象的类。有时需要修改 nib 中对象所属的类，本章后面将会对此进行介绍。

*属性查看器（Command-Option-4）*

这里的设置对应于代码中用于配置对象的属性与方法。比如，在属性查看器中选中视图，然后从后面的弹出菜单中进行选择相当于在代码中设置视图的 `backgroundColor` 属性。与之类似，选中按钮并在 Title 域中输入相当于调用按钮的 `setTitle:forState:` 方法。

属性查看器分为几部分，对应于所选对象的类层次结构。比如，UIButton 的属性查看器有 3 部分：除了 Button 部分，还有一个 Control 部分（因为 UIButton 也是个 UIControl）和一个 View 部分（因为 UIControl 也是个 UIView）。

*尺寸查看器（Command-Option-5）*

X、Y、Width 与 Height 域确定了对象在其父视图中的位置与大小，对应于代码中其 `frame` 属性；可以通过拖曳和缩放的方式在画布中完成这些操作，但这么做无法满足数字精度。

如果开启了自动布局（对于新的 *.storyboard* 与 *.xib* 文件，这是默认情况），那么尺寸查看器的其他部分就与所选对象的自动布局约束相关；此外，画布右下角的按钮可以自动管理对齐、定位与约束。

*连接查看器（Command-Option-6）*

本章后面将会介绍连接查看器的使用。

在编辑 nib 时有两个非常重要的库：

*对象库（Control-Option-Command-3）*

这个库是想要添加到 nib 中的对象来源。

*媒体库（Control-Option-Command-4）*

该库会列出项目中的媒体，比如，想要拖曳到 UIImageView 或直接拖曳到界面中的图片（在这种情况下会创建一个 UIImageView）。

刚才多次提到自动布局与约束，不过这里还不打算对其进行介绍，也不会介绍尺寸等级和条件约束（画布底部的“Any”按钮）。这些都是涵盖范围广泛的主题，与视图和视图控制器紧密相关，这已经超出了本书的讨论范围。我在另一本书《Programming iOS 9》中对其进行了详尽的介绍，包括如何在 nib 编辑器中处理约束与尺寸等级。

## 7.2 nib 加载

nib 文件是个关于潜在实例的集合，这些实例就是其 nib 对象。当应用运行并加载 nib 时，这些实例才会创建出来。这时，包含在 nib 中的 nib 对象会转换为应用可以使用的实例。

这种架构颇具效率。nib 通常会包含界面；界面是相对重量级的对象。nib 只在需要时才会加载；实际上，它可能永远都不会加载。通过这种方式，我们可以确保内存使用量最低，而这是非常重要的，因为移动设备上的内存是非常昂贵的资源。此外，加载 nib 是需要时间的，因此启动时加载少量 nib（足够生成应用的初始界面即可）会加快启动速度。

并没有所谓的“卸载”nib。nib 加载过程所完成的事情是生成一些实例；当这些实例生成后，nib 的工作就完成了。此后将会由运行着的应用来决定该如何使用生成的这些实例。只要需要这些实例，应用就得保持对其的引用；如果不再需要，那就可以将其销毁。

可以将 nib 文件看作用于生成实例的一组指令；当 nib 加载时会执行这些指令。相同的 nib 文件可以加载多次，每次都生成一组新的指令。比如，一个 nib 文件可能包含应用中多处都会用到的一些界面。代表表格中一行的 nib 文件可能会加载多次，从而生成表格中的多行。

## 7.2.1 何时加载 nib

当应用运行时，有一些主要的场景通常会加载 nib 文件：

*从故事板中实例化视图控制器*

故事板是场景的集合。每个场景都从一个视图控制器开始。当需要该视图控制器时，它会通过故事板实例化出来。这意味着包含该视图控制器的 nib 会加载进来。

视图控制器会通过故事板自动实例化出来。比如，当应用启动时，如果有主故事板，那么运行时就会寻找该故事板的初始化视图控制器（入口点）并将其实例化（参见第 6 章）。与之类似，故事板常常会包含由 Segue 连接的几个场景；在执行 Segue 时，目标场景的视图控制器会被实例化出来。

还可以在代码中以手工方式从故事板中实例化视图控制器。要想做到这一点，请从一个 UIStoryboard 实例开始，然后：

- 可以通过调用 `instantiateInitialViewController` 来实例化故事板的初始视图控制器。
- 可以通过调用 `instantiateViewControllerWithIdentifier:` 来实例化视图控制器，前提是该视图控制器的场景是在故事板中通过标识符字符串来命名的。

*视图控制器从 nib 中加载主视图*

视图控制器有一个主视图。不过视图控制器是个轻量级对象（只包含少量代码），而主视图则是个相对重量级的对象。因此，在实例化时，视图控制器会缺少主视图。当需要将视图放到界面中时，它才会将主视图生成出来。视图控制器可以通过几种方式来包含主视图；其中一种方式就是从 nib 中加载主视图。

如果视图控制器属于故事板中的某个场景，并且在故事板的画布中包含了视图（通常来说均如此），就像 Empty Window 示例项目一样，这就会涉及两个 nib：包含视图控制器的 nib 与包含主视图的 nib。包含视图控制器的 nib 会加载进来以便实例化视图控制器，就像之前所介绍的那样；现在，当该视图控制器实例包含了主视图时，主视图 nib 会自动加载，连接到该视图控制器的整个界面就会创建出来。

如果视图控制器是通过其他方式实例化的，那就会有一个与之相关的 *.xib*generated nib 文件，其中包含了主视图。重申一次，视图控制器会自动加载这个 nib，然后在需要时抽取出主视图。这种视图控制器与主视图 nib 文件之间的关联是通过 nib 文件

名来实现的。第 6 章曾通过 UIViewController 初始化器 `init(nibName:bundle:)` 以代码的方式配置这种关联，如下所示：

```
self.window!.rootViewController =
    MyViewController(nibName:"MyViewController", bundle:nil)
```

上述代码会让视图控制器将自己的 `nibName` 属性设为 `"MyViewController"`。这意味着当视图控制器需要其视图时，它是通过加载来自于 *MyViewController.xib* 的 nib 实现的。

*代码显式加载 nib 文件*

如果 nib 文件来自于 *.xib* 文件，那么代码可以手工加载它，这是通过调用如下方法之一来做到的：

`loadNibNamed:owner:options:`

这是个 NSBundle 实例方法。通常，你会直接调用 `NSBundle.mainBundle()`。

`instantiateWithOwner:options:`

这是个 UINib 实例方法。在实例化 UINib 并通过 `init(nibName:bundle:)` 对其初始化时会确定好 nib。

应用运行时指定 nib 文件实际上需要两部分信息：其名字与包含它的包。视图控制器不仅有 nibName 属性，还有一个 `nibBundle` 属性，用于指定 nib 的方法，如 `init(nibName:bundle:)`，会有一个 `bundle:` 参数。不过实际上，这个包都是应用包（或 `NSBundle.mainBundle()`，它们是一回事）；这是默认的，因此无须再指定包了。可以直接传递一个 nil，不必再显式提供一个包了。

## 7.2.2 手工加载 nib

在实际情况下，你会将应用配置为会自动加载大多数 nib，这与刚才提到的各种机制与场景相一致。不过为了理解 nib 加载过程，手工加载 nib 也是非常有益的，下面就来实现。

首先在 Empty Window 项目中创建并配置一个 *.xib* 文件：

1. 在 Empty Window 项目中，选择 File → New → File 并指定 iOS → User Interface → View。这是个 *.xib* 文件，包含了一个 UIView 实例。单击 Next 按钮。
2. 在 Save 对话框中，对新的 *.xib* 文件保持默认名字 View，单击 Create 按钮。
3. 现在回到项目导航器中；*View.xib* 文件已经创建出来并被选中，可以在编辑器中查看其内容。这些内容包含了一个 UIView。它非常大，因此请将其选中，在属性查看器中，在 Simulated Metric 下，将 Size 弹出菜单修改为 Freeform。这时，处理器会出现在画布中的视图旁边；拖曳它将视图缩小。

4. 使用任意子视图来装配视图，方式是将它们从对象库中拖曳到视图上。还可以配置视图本身；比如，在属性查看器中修改背景色（如图 7-7 所示）。

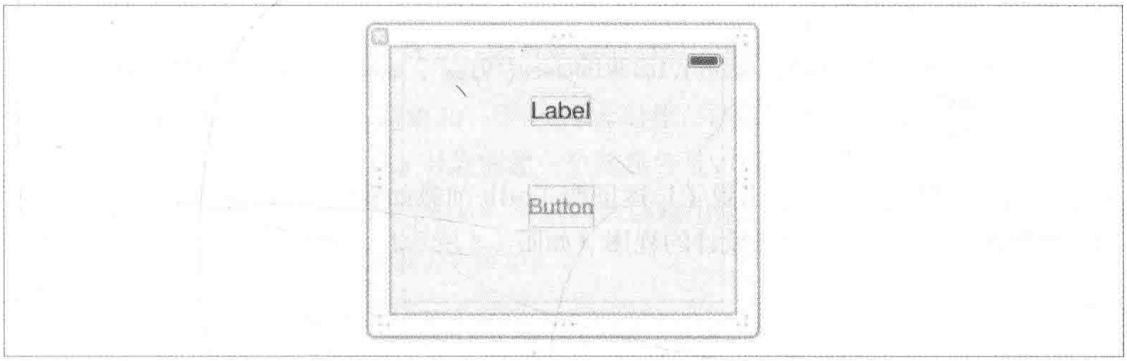


图 7-7：在 .xib 文件中设计视图

现在的目标是当应用运行时在代码中手工加载这个 nib 文件。编辑 *ViewController.swift*，在 `viewDidLoad` 方法体中，插入下面这行代码：

```
NSBundle.mainBundle().loadNibNamed("View", owner: nil, options: nil)
```

构建并运行应用。发生了什么？来自于 *View.xib* 的设计好的视图去哪儿了？ nib 加载失败了吗？

不是。nib 加载没有失败。它已经加载了！不过，我们还差两步。记住，在加载 nib 时要执行 3 个任务。

1. 加载 nib。
2. 加载时获取它所创建的实例。
3. 对实例进行一些处理。

我们执行了第 1 个任务（加载 nib），但并没有从中获取实例。这样，实例虽然创建出来，但随后又烟消云散了。为了防止这种情况的发生，我们需要通过某种方式捕获到这些实例。对 `loadNibNamed:owner:options:` 的调用会返回一个顶层 nib 对象数组，这些 nib 对象是从 nib 加载过程中实例化的。这就是我们需要捕获的实例！我们只有一个顶层 nib 对象（UIView），因此捕获数组的第 1 个元素（也只有这唯一一个元素）即可。重写代码如下所示：

```
let arr = NSBundle.mainBundle().loadNibNamed("View", owner: nil, options: nil)
let v = arr[0] as! UIView
```

现在执行了第 2 个任务：捕获到加载 nib 时所创建的实例。现在，变量 `v` 会引用全新的 UIView 实例。

不过，在构建并运行应用时，依然什么都不会出现，这是因为我们并未对该 UIView 进行任何处理。这是第 3 个任务。下面就对该 UIView 进行一些处理：将其放到界面上！再次重写代码，如下所示：

```
let arr = NSBundle.mainBundle().loadNibNamed("View", owner: nil, options: nil)
let v = arr[0] as! UIView
self.view.addSubview(v)
```

构建并运行应用，视图终于出现了！这证明了 nib 加载如我们所愿：我们可以在运行着的应用界面上看到在 nib 中所设计的视图（如图 7-8 所示）。

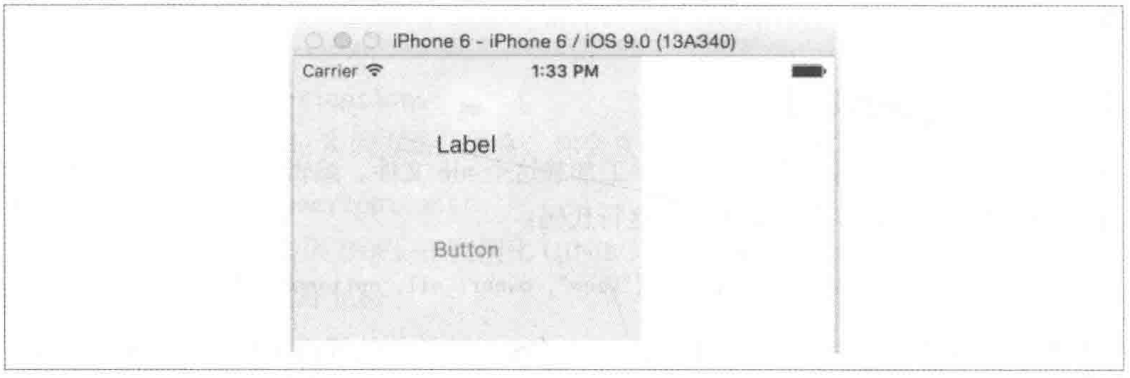


图 7-8：nib 加载的视图出现在了界面上

# 7.3 连接

连接指的是 nib 文件中的操作。它联合了两个 nib 对象，从一个运行到另一个。连接有个方向：这也是我为什么要用“从哪里到哪里”来描述它。这两个对象叫作连接的源与目标。

有两种类型的连接，插座变量连接与动作连接。本节后面的内容将会介绍它们、如何创建和配置，同时还会介绍它们所解决的问题的本质。

## 7.3.1 插座变量

nib 加载并且其实例生成时会产生一个问题：如果没有引用它们，那么这些实例就是无用的。7.2 节中，我们是通过捕获 nib 加载时所实例化的顶层对象数组来解决这个问题的。不过还有另外一种方式：使用插座变量。这种方式会复杂一些，它需要提前进行一些配置工作，而这项工作很容易出错。不过这种方式也是更加常用的，特别在 nib 是自动加载的情况下更是如此。

插座变量连接属于一类连接，它有个名字，名字实际上是个字符串。当 nib 加载时，一些奇妙的事情就会发生。源对象与目标对象不再仅仅是 nib 中的潜在对象；它们会成为真实存在的实例。插座变量的名字用于定位插座变量源对象中相同名字的实例属性，而目标对象则会被赋给该属性。现在，源对象就会有一个指向目标对象的引用！

比如，假设 nib 中有个 Dog 对象和一个 Person 对象，Dog 有个 `master` 实例属性。如果从 nib 中的 Dog 对象到 Person 对象创建一个插座变量，并且将其命名为 `"master"`，那么当 nib 加载时，Dog 实例与 Person 实例就会创建出来，并且 Person 实例会被赋给该 Dog 实例的 `master` 属性（如图 7-9 所示）。

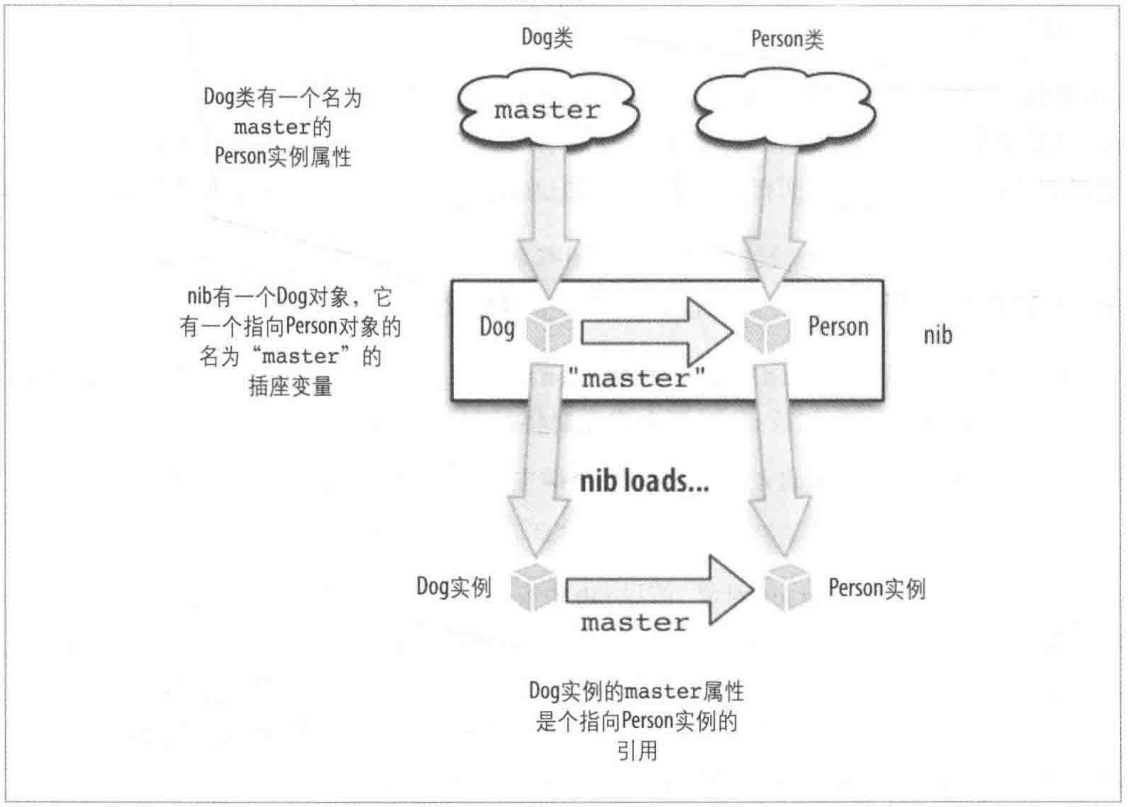


**图 7-9：插座变量是如何通过引用来指向 nib 所实例化的对象的**

nib 加载机制并不会神奇地创建出实例属性。也就是说，如果源对象不具有某个属性，那么当其实例化后，它并不会自动拥有这个属性。源对象所对应的类需要事先通过这个实例属性进行定义。这样，要想使用插座变量，我们需要在两处进行准备工作：一是源对象所对应的类，二是 nib。这有点棘手，Xcode 会帮助你，但也有可能把事情搞乱。（本章后面将会对此进行详细介绍。）

## 7.3.2 nib 拥有者

要想让插座变量捕获到从 nib 中创建的实例引用，我们需要一个从 nib 外部的对象到 nib 内部的对象的一个插座变量。这看起来似乎是不可能的事情，但实际上是可以的。nib 编辑器可以通过 nib 拥有者对象创建出这样的插座变量。首先，介绍如何在 nib 编辑器中找到 nib 拥有者对象；接下来介绍它到底是什么：

- 在故事板场景中，nib 拥有者是顶层的视图控制器。它是文档大纲中所列出场景中的第 1 个对象，也是场景停靠栏中所显示的第 1 个对象。
- 在 *.xib* 文件中，nib 拥有者是个代理对象。它是文档大纲或停靠栏中所显示的第 1 个对象，并且作为 File's Owner 列在 Placeholders 下面。

nib 编辑器中的 nib 拥有者对象表示 nib 加载时 nib 之外已经存在的实例。当 nib 加载时，nib 加载机制并不会实例化该对象；它已经是个实例了。实际上，nib 加载机制会用真正的、已经存在的实例代替 nib 拥有者对象，使用它来实现涉及 nib 拥有者的任何连接。

不过请等等！ nib 加载机制是如何知道该用哪个真正的、已经存在的实例来代替 nib 中的 nib 拥有者对象呢？这是因为在 nib 加载时，系统会通过两种方式告知它：

- 如果代码通过调用 `loadNibNamed:owner:options:` 或 `instantiateWithOwner:options:` 来加载 nib，那么你需要将拥有者对象作为 `owner:` 参数。
- 如果视图控制器实例自动加载 nib 来获得主视图，那么视图控制器实例会将自身作为拥有者对象。

比如，回到 Dog 对象与 Person 对象。假设 nib 中有个 Person nib 对象，但没有 Dog nib 对象。Nib 拥有者对象是个 Dog。Dog 有个 `master` 实例属性。我们配置一个从 Dog nib 拥有者对象到 Person 对象的插座变量，叫作 `"master"`。接下来加载 nib，将现有的 Dog 实例作为拥有者。nib 加载机制会匹配 Dog nib 拥有者对象与这个已经存在的实际的 Dog 实例，并将新实例化的 Person 实例作为该 Dog 实例的 `master`（如图 7-10 所示）。

回到 Empty View，下面通过重新配置来说明这个机制。我们已经通过 *ViewController.swift* 的代码加载了 View nib。代码会在 ViewController 实例中运行。因此，我们将该实例作为 nib 拥有者。这种配置有些乏味，不过请耐心一些，因为理解如何使用这种机制是非常重要的。下面就来看看具体步骤：

1. 首先，ViewController 需要一个实例属性。在 ViewController 类声明体的开头，插入属性声明，如下所示：

```
class ViewController: UIViewController {
    @IBOutlet var coolview : UIView!
```

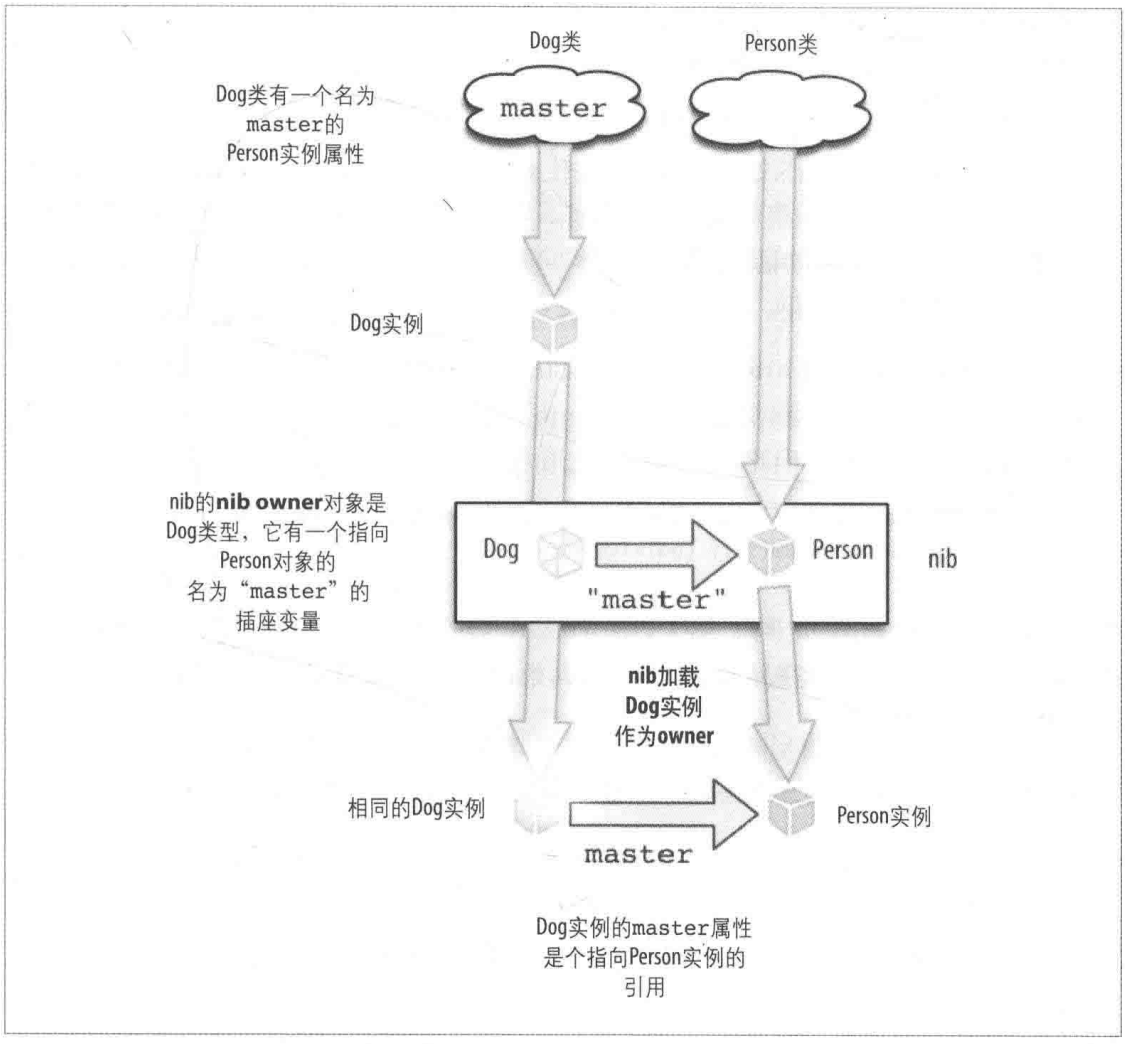


图 7-10：来自于 nib 拥有者对象的插座变量

你已经理解了 var 声明的含义；我们声明了一个名为 coolview 的实例属性。它声明为 Optional，这是因为在 ViewController 实例创建时它才会拥有“真正的”值；在 nib 加载时它会持有该值。@IBOutlet 属性告诉 Xcode 允许我们在 nib 编辑器中创建插座变量。

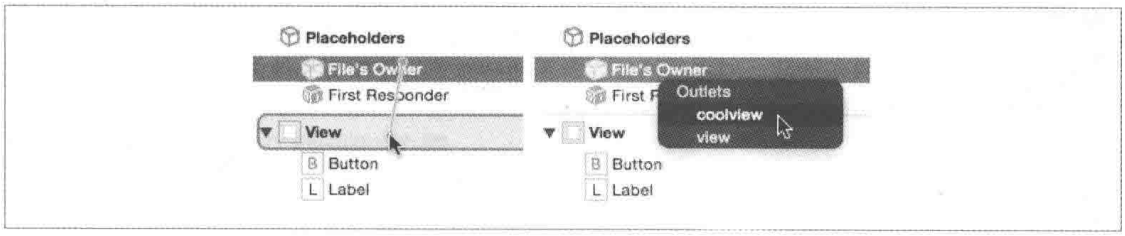


图 7-11：创建插座变量

2. 编辑 *View.xib*。首先要确保 nib 拥有者对象作为一个 ViewController 实例。选中 File's Owner 代理对象并切换到身份查看器。在第一个文本框中（Custom Class 下面），将 Name 值设为 `ViewController`。在文本框外单击并保存。

3. 现在创建插座变量！在文档大纲中，按住 Control 键并将 File's Owner 对象拖曳到 View 上；拖曳时有一根线会跟随着鼠标，松开鼠标。这时会出现一个提示，列出了可以创建的所有可能的插座变量（如图 7-11 所示）。其中有两个，分别是 `coolview` 与 `view`。单击 `coolview`（不是 `view`!）。

4. 最后，我们需要修改 nib 加载代码。现在不需要捕获实例化对象的顶层数组。现在要改变一下方式，我们会自己加载 nib 并将 self 作为拥有者。这样会自动设置 `coolview` 实例属性，因此我们就可以使用它了：

```
NSBundle.mainBundle().loadNibNamed("View", owner: self, options: nil)
self.view.addSubview(self.coolview)
```

构建并运行，一切如预期一样！第 1 行会加载 nib，并将 *`coolview`* 实例属性设为从 nib 实例化的视图。第 2 行会在界面上显示 `self.coolview`，因为 `self.coolview` 现在就是该视图。

下面总结一下。预先的配置有些棘手，因为要在两个地方进行配置，即代码中和 nib 中：

- 当 nib 加载时，如果一个类的实例是拥有者，那么这个类中一定会有一个实例属性（不仅要创建该属性，还要将其标记为 `@IBOutlet`）。
- 在 nib 编辑器中，当 nib 加载时，如果一个类的实例是拥有者，那么 nib 拥有者对象的类必须要设为这个类。
- 在 nib 编辑器中，一定要创建插座变量，其名字与属性名相同，并且从 nib 拥有者到某个 nib 对象（只有当另外两项配置做好了才可以执行这个步骤）。

如果上面一切都做好了，那么当 nib 加载时，如果使用正确的类拥有者，该拥有者的实例属性就会被设为插座变量的目标。

Xcode 7 中，当在 nib 中配置指向一个对象的插座变量时，文档大纲中所列出的对象名不再是泛泛的名字（如“View”），它会显示插座变量的名字（如“coolview”）。该名字只不过是个标签而已，它对于插座变量的操作没有任何影响，你可以在身份查看器中修改它。

### 7.3.3 自动配置 nib

在某些情况下，拥有者类与 nib 的配置可以自动进行。既然已经了解了如何手工配置拥有者与 nib，我们也可以理解这些自动化配置。

一个重要的示例是视图控制器是如何获取其主视图的。视图控制器有一个 `view` 属性。实际的视图通常来自于 nib。这样，当 nib 加载时，视图控制器就需要充当拥有者的角色，还需要有一个从 nib 拥有者对象到该视图的 `view` 插座变量。如果查看持有视图控制器主视图的实际 nib，你就会发现这一点。

回到 Empty Window 项目。编辑 *Main.storyboard*。它有一个场景，其 nib 拥有者对象是 View Controller 对象。在文档大纲中选中 View Controller，切换至身份查看器。它会显示出 nib 拥有者对象的类实际上就是 ViewController！

保持文档大纲中 View Controller 为选中状态，切换至连接查看器。它显示出实际上有一个从 View Controller 到 View 对象的插座变量连接，这个插座变量叫作 `"view"`！如果将鼠标悬浮在该插座变量连接上，那么画布中的 View 对象就会高亮显示，帮助你进行识别。

这说明了视图控制器是如何获取到其主视图的！当视图控制器需要其主视图时（因为视图要显示在界面上），view nib 就会加载——视图控制器会成为拥有者。这样，视图控制器的 `view` 属性会被设为这里所设计的视图。接下来，视图会显示在界面上：它与上面的内容会出现在运行的应用上。

对于第 6 章的 Truly Empty 项目亦如此。编辑 *MyViewController.xib*。nib 拥有者对象是 File's Owner 代理对象。选中 File's Owner 对象，切换至身份查看器。它会显示出 nib 拥有者对象的类实际上是 MyViewController！切换至连接查看器，它会显示出有一个连接到 View 对象的插座变量，名为 `"view"`！

这说明了视图控制器是如何获得其主视图的。在调用 `MyViewController(nibName:"MyViewController", bundle:nil)` 实例化视图控制器时，我们会告诉它去哪里寻找其 nib。不过，nib 本身已经正确配置了，因为在创建 MyViewController 类并勾选上“Also create XIB file”复选框时，Xcode 会帮我们做这件事。视图控制器加载 nib 时会将自身作为拥有者，插座变量即可生效：来自于 nib 文件的视图会成为视图控制器的 `view`，并显示在界面上。

### 7.3.4 误配置的插座变量

创建插座变量并能正常使用涉及几件事情。我敢保证你今后肯定会在这个地方栽跟头，插座变量无法正常使用。别生气，也别担心；请准备好！每个人都会遇到这个事情。重

要的是，要能识别出问题所在，这样才能知道到底哪里出错了。接下来我们有意做错一些事情，目的是看看哪些情况会导致插座变量配置不正确：

*插座变量名与源类中的属性名不匹配*

从 Empty Window 示例开始。运行项目来证明一切都如我们所愿。现在，在 *ViewController.swift* 中，将属性名改为 badview：

```
@IBOutlet var badview : UIView!
```

为了让代码编译通过，你还需要在 viewDidLoad 中修改对该属性的引用：

```
self.view.addSubview(self.badview)
```

代码编译通过。不过当运行时，应用会崩溃，控制台上转出的消息是："This class is not key value coding-compliant for the key coolview"。

从技术上来说，这条消息表示当 nib 加载时，nib 中的插座变量名（依旧是 coolview）与 nib 拥有者的属性名不匹配，这是因为我们将该属性名修改成了 badview，这导致配置出现了问题。实际上，一切都是正确的，不过我们绕过了 nib 编辑器，从插座变量源的类中删除了对应的实例属性。当 nib 加载时，运行时无法匹配插座变量的名字与插座变量源（ViewController 实例）中的任何属性，应用因此崩溃。

还有一些情况也会导致这种误配置。比如，你可以修改一些东西，导致 nib 拥有者成为错误类的实例：

```
NSBundle.mainBundle().loadNibNamed("View", owner: NSObject(), options: nil)
```

我们将 owner 设为一个通用的 NSObject 实例。结果一样：NSObject 类没有与插座变量名相同的属性，这样当 nib 加载时应用就会崩溃，提示拥有者并不是"键值编码兼容的"。会导致同样错误的另一个常见做法是在 nib 中将 nib 拥有者类设为错误的类。

*nib 中没有插座变量*

在 *ViewController.swift* 中，将之前示例对属性名的引用由 badview 改回到 coolview。运行项目来证明修改是正确的。现在来做一些破坏！编辑 *View.xib*。选中 File's Owner 并切换至连接查看器，单击第 2 个椭圆形左侧的 X 来取消 coolview 插座变量的连接。运行项目，应用会崩溃，控制台上转出的消息是："Fatal error: unexpectedly found nil while unwrapping an Optional value"。

将插座变量从 nib 中删除。当 nib 加载时，ViewController 实例属性 coolview（其类型是个隐式、展开的 Optional，它包装了一个 UIView，即 UIView!）不会被设置。这样，它会保持其初始值，即 nil。接下来，将其放到界面中来使用隐式展开的 Optional：

```
self.view.addSubview(self.coolview)
```

Swift 会展开 Optional，不过你无法展开 nil，因此程序会崩溃。

*没有视图插座变量*

对于这种情况来说，你需要使用第 6 章的 Truly Empty 示例，该示例会从一个 *.xib* 文件中加载视图控制器的主视图；我无法使用 *.storyboard* 文件来说明问题，因为故事板编辑器不允许这么做。在 Truly Empty 项目中，编辑 *MyViewController.xib* 文件。选中 File's Owner 对象并切换至连接查看器，取消 view 插座变量的连接。运行项目。程序启动时会崩溃，控制台转出的消息是："Loaded the 'MyViewController' nib but the viewoutlet was not set"。

控制台消息已经说明了情况。作为视图控制器主视图源的 nib 必须要有一个从视图控制器（nib 拥有者对象）到视图的 view 插座变量。

## 7.3.5 删除插座变量

一致性地删除插座变量（也就是说，不会导致上面提及的那些问题）涉及要同时修改几处地方，就像创建插座变量一样。建议按照下面这个顺序进行：

1. 取消 nib 中插座变量的连接。
2. 从代码中删除插座变量声明。
3. 尝试编译，让编译器捕获其余的问题。

比如，假设要从 Empty Window 项目中删除 coolview 插座变量，遵循上面提到的 3 个步骤，做法如下所示：

1. 取消 nib 中插座变量的连接。要想做到这一点，请编辑 *View.xib*，选中源对象（File's Owner 代理对象），然后在连接查看器中单击 X 取消 coolview 插座变量的连接。
2. 从代码中删除插座变量声明。要想做到这一点，请编辑 *ViewController.swift*，删除或注释掉 @IBOutlet 声明这一行。
3. 删除对属性的其他引用。最简单的方式是构建项目；编译器会对 *ViewController.swift* 中 self.coolview 这一行代码报错，因为现在已经没有 v 属性了。删除或注释掉该行，再次构建，验证一切正常。

## 7.3.6 创建插座变量的其他方式

之前创建插座变量的方式是这样的：首先在类文件中声明一个实例属性，然后在 nib 编辑器中，按住 Control 键，从文档大纲的源（该类的实例）拖曳到目标处，从弹出列表

中选择所需的插座变量属性。Xcode 提供了多种方式来创建插座变量，这里就不再一一列举了。我会介绍一些常见的方式。

继续使用 Empty Window 项目与 *View.xib* 文件。请记住，所有这些与对 *.storyboard* 文件的操作是一样的。

通过连接查看器删除 *View.xib* 中的插座变量（如果之前没有做过）。在 *ViewController.swift* 中，创建（或取消注释）属性声明，然后保存：

```
@IBOutlet var coolview : UIView!
```

现在来试一下！

*从源连接查看器中拖曳*

可以拖曳 nib 编辑器的连接查看器中的圆圈来连接插座变量。在 *View.xib* 中，选中 File's Owner 并切换至连接查看器。`coolview` 插座变量会列出来，不过它尚未连接：右侧的圆圈是打开的。从 `coolview` 旁边的圆圈拖曳到 nib 中的 UIView 对象上。可以拖曳到画布或文档大纲中的视图上。在拖曳圆圈时不需要按住 Control 键，也没有提示列表，因为你是从特定的插座变量上拖曳的，Xcode 知道是哪个。

*从目标连接查看器中拖曳*

现在按照相反的方向完成相同的步骤。删除 nib 中的插座变量。选中 View 并打开连接查看器。我们需要一个将该视图作为目标的插座变量：这是个“引用插座变量”。从 New Referencing Outlet 旁边的圆圈拖曳到 File's Owner 对象上。提示列表会出现：单击 `coolview` 来创建插座变量连接。

*从源提示列表拖曳*

可以使用与连接查看器相同的提示列表。下面就从这个提示列表开始。再一次，删除连接查看器中的插座变量。按住 Control 键并单击 File's Owner。这时会弹出一个提示列表，看起来像是连接查看器！从 `coolview` 右侧的圆圈拖曳到 UIView 上。

*从目标提示列表拖曳*

再一次，我们按照相反的方向完成相同的步骤。删除连接查看器中的插座变量。在画布或文档大纲中，按住 Control 键并单击视图。这时会出现一个提示列表，显示其连接查看器。从 New Referencing Outlet 旁边的圆圈拖曳到 File's Owner 上。这时又会出现一个提示列表，列出了可能的插座变量；单击 `coolview`。

再一次，删除插座变量。现在通过在代码与 nib 编辑器间拖曳来创建插座变量。这要求你同时在两处进行操作：你需要一个辅助窗格。在主编辑器窗格中，打开 *ViewController.swift*。在辅助窗格中，打开 *View.xib*，这样视图就是可编辑的了。

*从属性声明拖曳到 nib*

代码中，属性声明旁边有个空心圆圈。你觉得它是干什么的呢？将其拖曳到 nib 编辑器的 View 上（如图 7-12 所示）。这时，nib 中会形成插座变量连接；可以通过连接查看器看到这一点，回到代码，你会发现圆圈不再是空心的了。将鼠标悬浮在填充后的圆圈上或单击它，看看它连接到了 nib 中的哪个插座变量上。单击这个实心圆圈会弹出一个菜单，可以单击菜单转向目标对象。

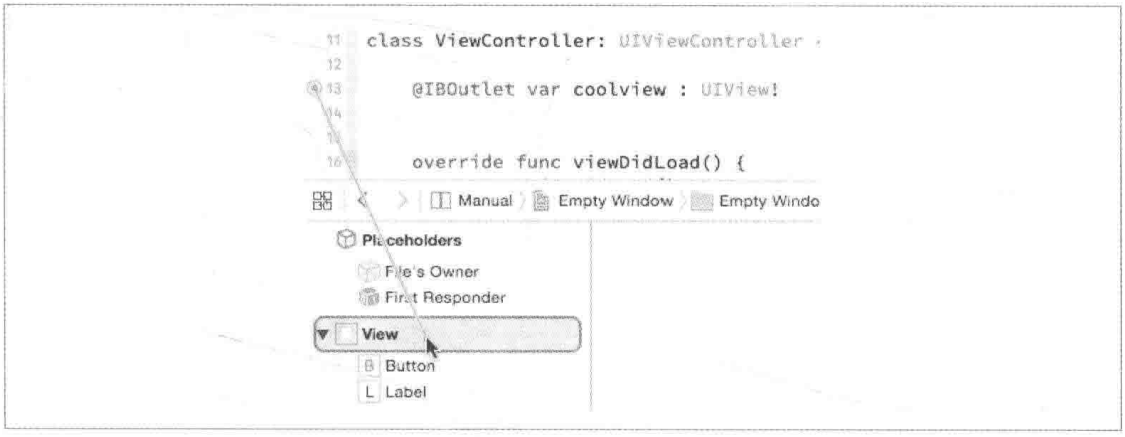


**图 7-12：从代码拖曳到 nib 编辑器来连接插座变量**

还有一种方式，也是最令人惊讶的方式。保留上一示例的两个窗格排列。再一次，删除插座变量（你可能需要使用连接查看器或 nib 编辑器中的弹出列表）。再从代码中删除 `@IBOutlet` 这一行！我们来创建属性声明并连接插座变量，一步即可搞定！

*从 nib 拖曳到代码*

按住 Control 键将 nib 编辑器中的视图拖曳到类 ViewController 的声明体中。提示列表会显示出 Insert Outlet 或 Outlet Collection（如图 7-13 所示）。松开鼠标。这时会出现一个弹出层，可以配置插入代码中的声明。按照图 7-14 进行配置：你需要一个插座变量，该属性应该命名为 `coolview`。单击 Connect。这时，属性声明会插入代码中，插座变量会在 nib 中进行连接，这一切只需一步操作即可。

直接连接代码与 nib 编辑器来创建插座变量确实非常酷，也非常方便，不过要当心：并不存在这种直接的连接。要想让插座变量能够正常使用，总是要有两部分内容：类中的实例属性以及 nib 中的插座变量，其名字是相同的，来自于该类的实例。正是名字与类的同一性才使得 nib 加载时它们能在运行期匹配上。Xcode 会帮助你将一切准备好，不过实际上并不是什么魔法将代码连接到 nib 上。

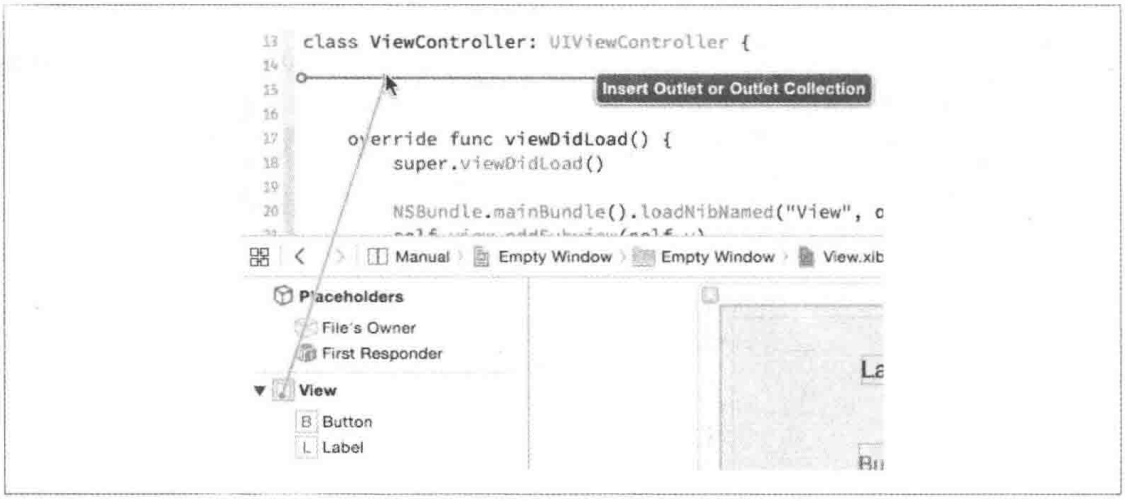


图 7-13：从 nib 编辑器拖曳到代码来创建插座变量

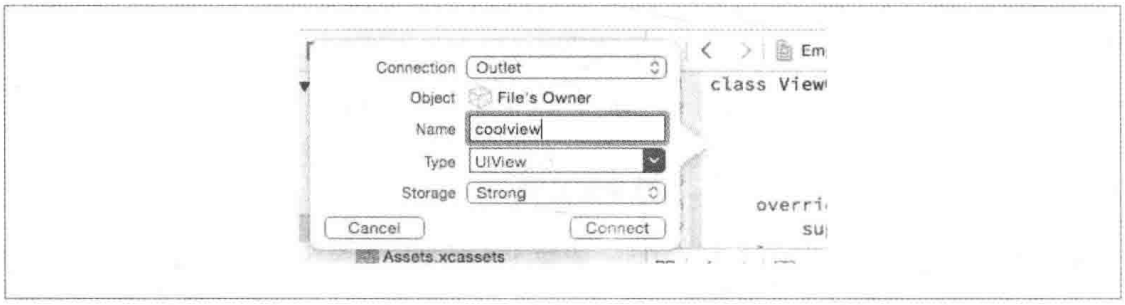


图 7-14：配置属性声明

## 7.3.7 插座变量集合

插座变量集合指的是与相同类型对象的多个连接匹配（nib 中）的数组实例属性（代码中）。

比如，假设一个类包含了如下属性声明：

```
@IBOutlet var coolviews: [UIView]!
```

结果就是在 nib 编辑器中，如果选中了该类的实例，那么连接查看器就会在 Outlet Collections 而非 Outlets 下面列出 `coolviews`。这意味着你可以构建多个 `coolviews` 插座变量，每个都会连接到 nib 中的不同 UIView 对象上。当 nib 加载时，这些 UIView 实例会成为数组 `coolviews` 的元素；插座变量构建的顺序就是数组中元素的排列顺序。

这么做的好处在于代码可以通过数字（数组索引）来引用从 nib 实例化的多个界面对象，而不必为每个对象使用不同的名字。在构建如自动布局约束与手势识别的插座变量时，这么做是非常有用的。

## 7.3.8 动作连接

就像插座变量连接一样，动作连接也是让 nib 中的一个对象能够引用另外一个对象的方式。不过，它并非属性引用，而是消息发送引用。

所谓动作，就是当用户对 Cocoa UIControl 界面对象（一个控件）执行了某种操作后由其产生并发送给其他对象的消息，如轻拍控件等。会导致控件发出动作消息的各种用户行为叫作事件。要想查看完整的事件列表，请查看 UIControl 类的文档，位于“Control Events”下。比如，对于 UIButton，用户轻拍按钮对应于 `UIControlEvents.TouchUpInside` 事件。

控件对象要知道下面 3 点才能让这个架构正常运作：

- 响应什么控件事件。
- 当该控件事件（动作）发生时会发送什么消息（调用的方法）。
- 将消息发送给哪个对象（目标）。

nib 中的动作连接会将这 3 点融入自身当中。它拥有作为源的控件对象；其终点就是目标；在构建时，你会指明动作连接，以及哪个控件事件与动作消息。为了构建动作连接，首先需要配置目标对象所对应的类，使之拥有适合于作为动作消息的方法。

为了尝试动作连接，nib 中需要有一个 UIControl 对象，如按钮。Empty Window 项目的 *Main.storyboard* 文件中可能已经有了这样的按钮。不过，当应用运行时，从 *View.xib* 中所加载的视图可能会覆盖这个按钮。因此，首先需要在 *ViewController.swift* 中清除 `ViewController` 类声明，使之不再有插座变量属性与手工加载 nib 代码；如下就是清理之后的代码：

```
class ViewController: UIViewController {
}
```

下面将 Empty Window 项目中的视图控制器作为按钮 `UIControlEvents.TouchUpInside` 事件（表示轻拍了按钮）所发出的动作消息的目标。我们需要在视图控制器中添加一个方法，当轻拍按钮后会调用该方法。为了看起来明显一些，我们让视图控制器弹出一个警告窗口。将如下方法添加到 *ViewController.swift* 声明体中：

```
class ViewController: UIViewController {
    @IBAction func buttonPressed(sender:AnyObject) {
        let alert = UIAlertController(
            title: "Howdy!", message: "You tapped me!", preferredStyle: .Alert)
        alert.addAction(
            UIAlertAction(title: "OK", style: .Cancel, handler: nil))
        self.presentViewController(alert, animated: true, completion: nil)
```

```
    }
}
```

@IBAction 属性就像是 @IBOutlet 一样：它用来提示 Xcode，让 Xcode 在 nib 编辑器中加入这个方法。实际上，如果查看 nib 编辑器，你会发现它就在那儿：编辑 *Main.storyboard*，选中 View Controller 对象并切换至连接查看器，你会看到 buttonPressed: 现在位于 Received Actions 下面。

在 *Main.storyboard* 中的唯一一个场景中，顶层 View Controller 的 View 应该会包含一个按钮（本章之前创建的，如图 7-5 所示）。如果不存在，请添加一个，然后将其放到视图左上角。我们的目标是将按钮的 Touch Up Inside 事件（作为动作）连接到 ViewController 中的 buttonPressed: 方法上。

与插座变量连接一样，动作连接也有一个源和一个目标。这里的源就是按钮，目标是 View Controller，View Controller 实例作为包含按钮的 nib 的拥有者。有多种方式可以构建这个插座变量连接，这与动作连接都是完全对应的。区别在于必须要配置连接的两端。在按钮（源）端，需要将 Touch Up Inside 指定为所要使用的控件事件；幸好，这是 UIButton 的默认值，因此可以省略这一步。在视图控制器（目标）端，需要将 buttonPressed: 指定为要调用的动作方法。

下面按住 Control 键从按钮拖曳到 nib 编辑器中的视图控制器来构建动作连接：

1. 按住 Control 键从按钮（在画布或文档大纲中）拖曳到文档大纲中所列出的 View Controller（或拖曳到画布中视图上面的场景停靠栏中的视图控制器图标上）来创建连接。
2. 这时会出现一个提示列表（如图 7-15 所示），列出了可能的连接；它会列出 Segue，以及 Sent Event，特别是 buttonPressed:。
3. 单击提示列表中的 buttonPressed:。

现在会形成动作连接。这意味着当应用运行时，只要按钮的 Touch Up Inside 事件触发（表示按钮被按下），那么它就会向目标（视图控制器实例）发送消息 buttonPressed:。我们知道这个方法应该做什么事情：它会弹出一个警告窗口。试一下吧！构建并运行应用，当应用出现在模拟器中时，单击按钮，事情与你想的一样！

## 7.3.9 创建动作的其他方式

如果在 *ViewController.swift* 中创建了动作方法，那么还有下面几种方式可以在 nib 中创建动作连接：

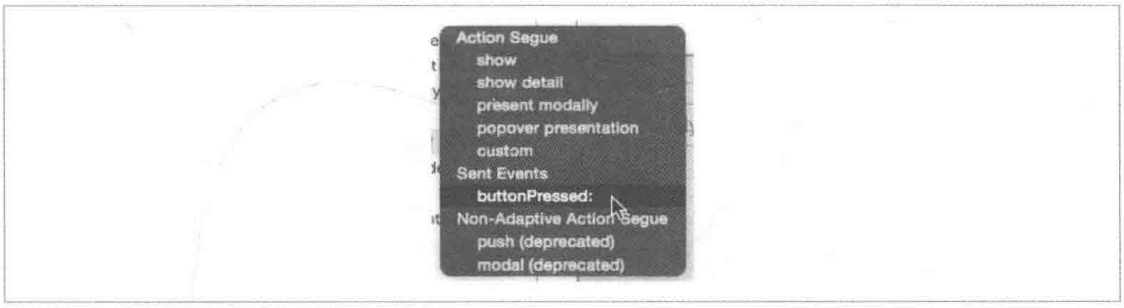


图 7-15：展示出动作方法的提示列表

- 按住 Control 键并单击视图控制器，这会弹出一个提示列表，类似于连接查看器。从 `buttonPressed:`（位于 Received Actions 下）拖曳到按钮。这会弹出另一个提示列表，列出可能的控件事件，单击其中的 Touch Up Inside。
- 选中按钮并使用连接查看器。从 Touch Up Inside 的圆圈拖曳到视图控制器。这时会弹出一个提示列表，列出视图控制器中已知的动作方法；单击 `buttonPressed:`。
- 按住 Control 键并单击按钮。这时会弹出一个提示列表，类似于连接查看器。像之前一样操作。
- 在一个窗格中显示 *ViewController.swift*，在另一个窗格中显示故事板。*ViewController.swift* 中的 `buttonPressed:` 方法左侧有一个圆圈。从该圆圈拖曳到 nib 中的按钮上。

与插座变量连接一样，创建动作连接最为直观的方式就是从 nib 编辑器拖曳到代码中，插入动作方法并在 nib 中构建动作连接，一步即可完成。首先请删除代码中的 `buttonPressed:` 方法以及 nib 中的动作连接。在一个窗格中显示 *ViewController.swift*，在另一个窗格中显示故事板。现在：

1. 按住 Control 键，从 nib 编辑器中的按钮拖曳到 ViewController 类声明体的空白区域。代码中会弹出一个提示列表，提示创建插座变量或动作，松开鼠标。
2. 一个弹出框会出现，这一块比较棘手。在默认情况下，该弹出框用于创建插座变量连接，但这并非你想要的；你需要的是动作连接！将连接弹出菜单修改为 Action。现在输入动作方法的名字（`buttonPressed`）并配置声明的其他部分（默认值就可以了，如图 7-16 所示）。

Xcode 会在 nib 中构建动作连接，并向代码中插入一个桩方法：

```
@IBAction func buttonPressed(sender: AnyObject) {
}
```

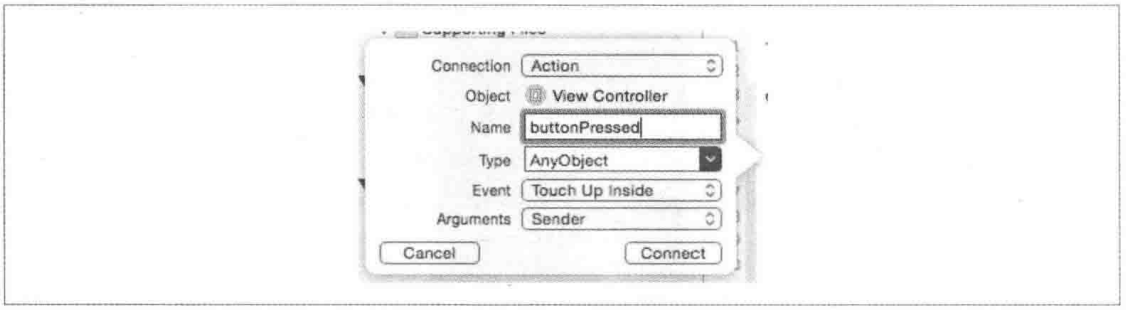


图 7-16：配置动作方法声明

这仅仅是个桩方法（Xcode 肯定不知道这个方法要做什么），在实际情况下，你需要在花括号之间插入一些功能代码。就像插座变量连接一样，动作方法代码旁边的实心圆圈表示 Xcode 已经认为连接就绪，可以单击这个实心圆圈查看、导航到连接中的源对象。

## 7.3.10 误配置的动作

与插座变量连接一样，配置动作连接涉及两端（nib 与代码）的一些处理与配置，这样它们才能匹配上。因此，可以有意破坏动作连接的配置，并让应用崩溃。通常的误配置出现在嵌入 nib 动作连接中的动作方法名与代码中的动作方法名不匹配。

为了证明这一点，请将代码中的函数名由 `buttonPressed` 修改为其他名字，如 `buttonPushed`。运行应用并轻拍按钮。应用会崩溃并在控制台中显示错误消息："Unrecognized selector sent to instance"。选择器是一条消息，即方法的名字。运行时尝试向对象发送一条消息，不过该对象并没有与之对应的方法（因为方法已经重命名了）。如果看一下错误消息开头，你会发现它甚至告诉你了这个方法的名字：

```
-[Empty_Window.ViewController buttonPressed:]
```

运行时表示它尝试调用 Empty Window 模块 ViewController 类中的 `buttonPressed:` 方法，但 ViewController 类却没有这个方法。

## 7.3.11 nib 之间的连接——不行！

不能在一个 nib 中的对象与另一个 nib 中的对象之间创建插座变量连接与动作连接。比如：

- 不能在两个不同的 *.xib* 文件中打开 nib 编辑器，然后按住 Control 键从一个文件拖曳到另一个文件来创建连接。
- 在 *.storyboard* 文件中，不能按住 Control 键从一个场景中的对象拖曳到另一个场景中的对象来创建连接。

如果想这么做，那就说明你还没有理解 nib（或场景、连接）到底是什么。

原因很简单：nib 中的对象会在 nib 加载时成为实例，因此在 nib 中将其连接起来是合理的，因为我们知道当 nib 加载时会有哪些实例。两个对象可以从 nib 中实例化，其中一个可能是代理对象（nib 拥有者），不过它们必须位于同一个 nib 中，这样当 nib 加载时，我们可以根据具体情况来配置实际实例之间的关系。

如果插座变量连接或动作连接是从一个 nib 中的对象到另一个 nib 中的对象建立的，那么就没有办法知道真正连接的实例是什么，因为它们是不同的 nib，会在不同时刻加载。从一个 nib 生成的实例与从另一个 nib 生成的实例之间的通信问题是程序中实例之间通信方式的一种特殊情况，详见第 13 章。

## 7.4 nib 实例的其他配置

当 nib 加载完毕后，其实例已经是功能完备的了；它们已经通过属性与尺寸查看器中的所有属性初始化和配置好了，其插座变量用于设置相应实例变量的值。然而，当对象从加载的 nib 中实例化后，你可能还想向初始化过程附加自己的代码。本节将会介绍几种做法。

一种常见的情况是视图控制器（当包含主视图的 nib 加载时，它作为拥有者，因此在 nib 中表示为 nib 拥有者对象）拥有一个插座变量，它指向从 nib 实例化的界面对象。在这种架构中，视图控制器可以对该界面对象做进一步的配置，因为 nib 加载后它有一个指向它的引用——相应的实例属性。进行这种配置最方便的地方就是其 `viewDidLoad` 方法。在调用 `viewDidLoad` 时，视图控制器的视图已经加载了，也就是说，视图控制器的 `view` 属性已经设为了实际的主视图，这是从 nib 实例化的，所有插座变量都会连接起来；不过视图尚未出现在可视化界面上。

另一种可能是除了在 nib 中进行配置，你还希望 nib 对象能够自我配置。通常来说，这是因为你有一个内建界面对象类的自定义子类。事实上，你想要创建自定义类，从而放置一些自定义代码。你要解决的问题是 nib 编辑器不允许你进行后续配置，或有很多对象，并且希望以一种一致且精心设定好的方式进行配置，这样相对于单独配置每一个，通过共同类进行配置才更有意义。

一种方式是在自定义类中实现 `awakeFromNib`。对象通过 nib 加载实例化后（对象初始化并配置完毕，其连接也建立起来了），`awakeFromNib` 会向所有这些对象发送消息。

比如，下面创建一个按钮，无论在 nib 中如何配置，其背景色总是红色的（这个例子没什么意义，不过能说明问题）。在 Empty Window 项目中，创建一个按钮子类 RedButton：

1. 在项目导航器中，选择 File → New → File。然后选择 iOS → Source → Cocoa Touch Class，单击 Next 按钮。
2. 将新建的类命名为 RedButton，让它成为 UIButton 的子类，单击 Next 按钮。
3. 确保将其保存到项目目录中，位于 Empty Window 分组下，同时勾选 Empty Window 应用目标，单击 Create 按钮。Xcode 会创建 *RedButton.swift*。
4. 在 *RedButton.swift* 的 RedButton 类的声明中，实现 `awakeFromNib`：

```
override func awakeFromNib() {
    super.awakeFromNib()
    self.backgroundColor = UIColor.redColor()
}
```

现在有一个 UIButton 子类，在其从 nib 实例化时，它会变成红色。不过，任何 nib 中都没有该子类的实例。编辑故事板，选中已经位于主视图中的按钮，使用身份查看器将按钮所属的类改为 RedButton。

构建并运行项目。当然，按钮现在是红色的！

还可以使用 nib 对象身份查看器中的 User Defined Runtime Attributes。可以通过它配置 nib 编辑器中没有内建界面的 nib 对象的方方面面。这里实际做的是在 nib 加载时发送 nib 对象，一个 `setValue:forKeyPath:` 消息，键路径会在第 10 章介绍。自然而然地，对象需要做一些准备来响应给定的键路径，否则 nib 加载时应用会崩溃。

比如，nib 编辑器的一个缺点是它无法配置层属性。假设我们要通过 nib 编辑器将红色按钮改为圆角的。在代码中，要通过设置按钮的 `layer.cornerRadius` 属性实现上述目标。nib 编辑器无法访问该属性。相反，可以在 nib 编辑器中选中按钮，使用身份查看器中的 User Defined Runtime Attributes。将 Key Path 设为 `layer.cornerRadius`，将 Type 为 Number，将 Value 值设置成什么都可以，如 10（如图 7-17 所示）。现在构建并运行；当然，按钮现在变成圆角的了。

还可以通过将属性设为可检查的来配置 nib 对象的自定义属性。为了做到这一点，请将 `@IBInspectable` 特性添加到属性声明中。这样，属性就会列在 nib 对象的属性查看器中。比如，现在准备在 nib 编辑器中配置按钮的边框。在 RedButton 类声明体的开头添加如下代码：

```
@IBInspectable var borderWidth : CGFloat {
    get {
        return self.layer.borderWidth
    }
    set {
```

```
            self.layer.borderWidth = newValue
        }
    }
```

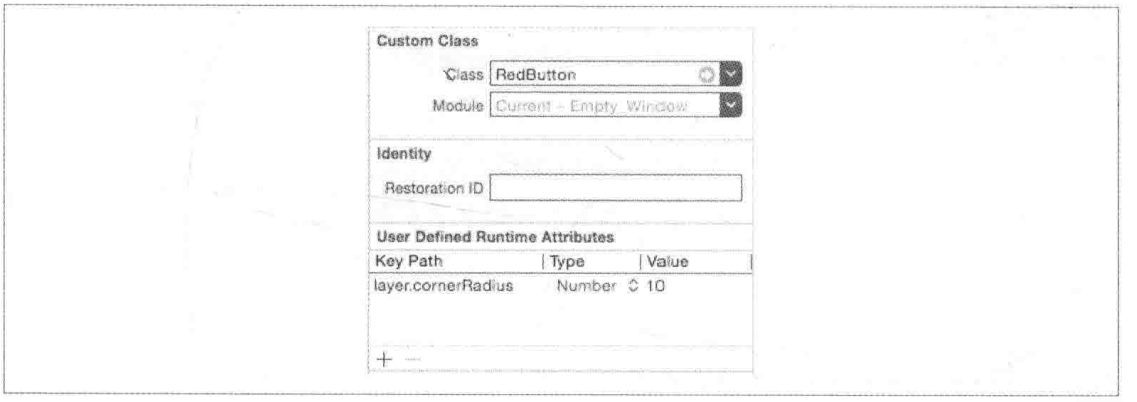


图 7-17：通过运行时特性将按钮修改为圆角

上述代码声明了一个 RedButton 属性 `borderWidth`，并使其成为层的 `borderWidth` 属性前的一个门面。这样，如果一个按钮是 RedButton 类的实例，那么 nib 编辑器还会在属性查看器中显示出该属性（如图 7-18 所示）。结果就是，当在 nib 编辑器中为该属性赋值时，这个值会在 nib 加载时发送给该属性的 setter，按钮边框就会变成值所指定的宽度。

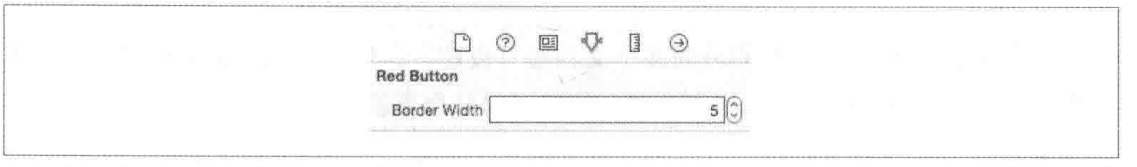


图 7-18：nib 编辑器中可检查的属性

要想更早地介入对象的初始化过程中，如果对象是个 UIView（或是 UIView 的子类），那么可以实现 `init(coder):`。值得注意的是，对于 UIView，nib 加载实例化时并不会调用 `init(frame:)`，而会调用 `init(coder:)`。（实现 `init(frame:)`，然后想知道当视图从 nib 实例化时为何代码不会被调用是初学者常犯的一个错误）。其最简单的实现如以下代码所示：

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder:aDecoder)
    // your code here
}
```

# 第 8 章

# 文档

知识分为两类。一类是我们要掌握的学科知识，
另一类是要知道在哪里可以找到有关知识的信息。
——Samuel Johnson，《Boswell's Life of Johnson》

iOS 编程再重要也不如流畅、清晰的文档重要。Cocoa 中有大量的内建类，其中有很多方法、属性和其他详细信息。虽然也有一些瑕疵，但 Apple 的文档却是权威的官方指南，因为你无法直接了解框架的内部机理，这样文档就成为你在使用这个庞大的框架时了解其行为的帮手。

安装在机器上的 Xcode 文档划分为多个文档集（或库）。你不能只安装文档集；你需要订阅它，这样当 Apple 发布文档更新时，你就可以获得更新后的版本了。

初次安装 Xcode 时，文档集并不会安装到计算机上；在文档窗口（8.1 节将会介绍）中查看文档需要联网，这样才能查看 Apple 网站的在线文档。这种方式很不方便；你需要在自己的计算机上保存一份文档副本。

因此，安装后应该立即启动 Xcode，让其下载并安装初始文档集。可以在某种程度上控制并监控这个过程，就在首选项窗口的下载窗格中（位于文档下）；还可以指定是否要自动安装更新，抑或时不时地手工单击 Check and Install Now。还可以在这里指定安装哪些文档集；我认为 iOS 9 文档集与 Xcode 7 文档集是你进行 iOS 开发时所需的全部文档，不过安装 OS X 10.11 文档集也没问题。初次安装文档集时，你需要提供计算机的管理员密码。文档集安装在主目录下的 *Library/Developer/Shared/Documentation/DocSets* 目录中。

## 8.1 文档窗口

在 Xcode 中，访问文档的主要途径是通过文档窗口（Window → Documentation and API Reference 或 Help → Documentation and API Reference，Command-Shift-0）。在文档窗

口中，查看文档的主要方式是通过搜索进行的；比如，按下 Command-Shift-0（如果已经在文档窗口中，那就请按下 Command-L），输入 `NSString` 并回车，选择第一个结果，即 NSString 类参考。如果需要，可以单击放大镜图标将结果限定在 iOS 相关的文档集中。

在文档窗口中有两种方式可以查看搜索结果：

*弹出结果窗口*

如果不断在搜索框中输入，那么会有很多结果列在弹出窗口中。使用鼠标单击，或通过箭头键导航，然后按下回车键，指定想要查看的结果。如果搜索框获得了焦点，那么还可以通过按下 Esc 键弹出或隐藏这个弹出窗口。

*完整的结果页面*

如果搜索框获得了焦点，但弹出结果窗口没有出现，那么可以按下回车键查看列出了所有搜索结果的页面；这些结果会根据类别列在 4 个单独的页面中，类别分别是 API 参考、SDK 指南、工具指南与示例代码。

还可以从代码中进行文档窗口搜索。你会经常这么做：你想在代码中直接查看用到的某个符号（类名、方法名或属性名等），并且想了解关于它的更多信息。按住 Option 键并将鼠标悬浮在代码中的某个符号上，直到出现一个蓝色的点状下划线；接下来（依然要按下 Option 键），双击该符号。这会打开文档窗口，你会直接进入类文档页面中对该术语的解释部分，或完整的搜索结果页面中。

（与之类似，第 9 章将会介绍的代码完成过程中，可以单击 More 链接完成相同的事情，直接跳转到当前符号对应的文档中。）

此外，可以在代码中（或其他地方）选定一段文本，然后选择 Help → Search Documentation for Selected Text（Command-Option-Control-/）。这相当于在文档窗口的搜索框中输入该文本，然后查看完整的结果页面。

文档窗口像是一个漂亮的 Web 浏览器，因为文档本质上包含的是网页。多个页面可以同时出现在文档窗口的页签中。要想导航到新的页签，请在导航时按住 Command 键（比如，按住 Command 键并单击某个链接，或按住 Command 键并单击弹出结果窗口中所选的某项），或从上下文菜单中选择 Open Link in New Tab。可以在页签之间导航（Window → Show Next Tab），每个页签都会记住其导航历史（Navigate → Go Back，或是使用窗口工具栏中的后退按钮，它也是个弹出菜单）。

可以在 Web 浏览器中打开当前在文档窗口中所查看的页面，方式是选择 Editor → Share → Open in Browser。

文档页面可能还会带有一个相关条目列表。列表开头会显示在页面上方的窗格中；当单击“More related items”链接时，完整列表会弹出来（如图 8-1 所示）。比如，NSString 类参考页面的相关条目窗格包含了 NSString 类继承与使用的协议的链接，同时在弹出层中还会显示出更多的信息与链接。本章后面将会介绍关于类相关条目的更多信息。

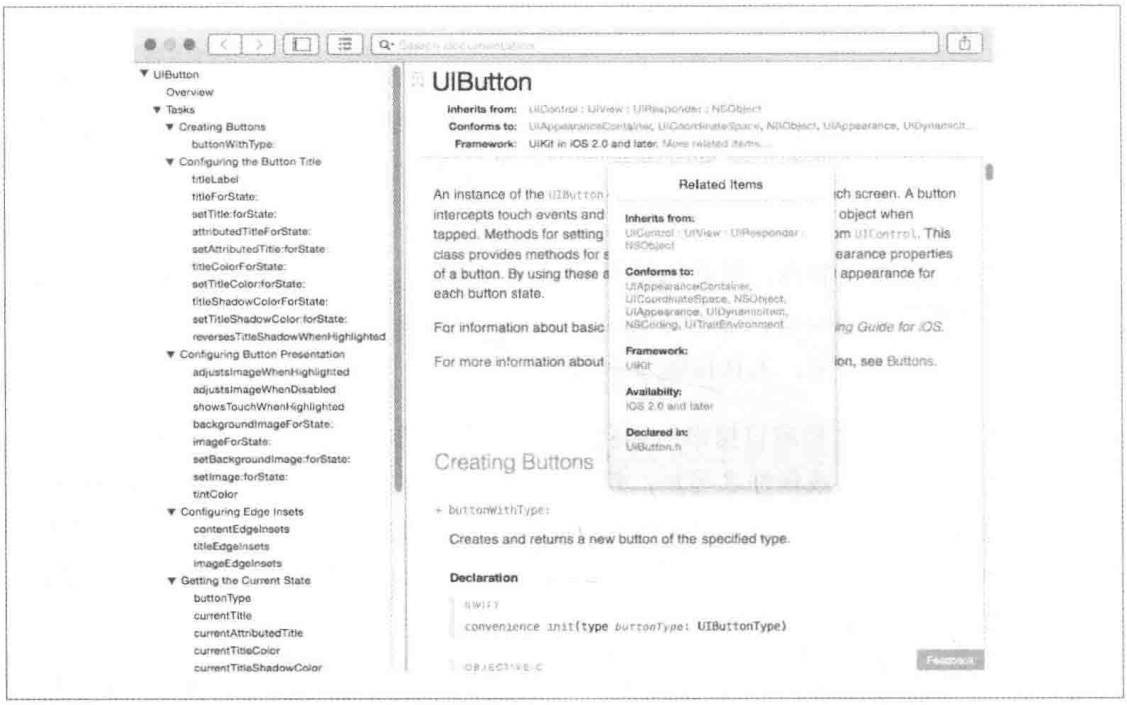


图 8-1：UIButton 类文档页的开头

文档页面可能会带有一个目录，显示在文档页面左侧的窗格中（如图 8-1 所示）；如果没有显示，那么请选择 Editor → Show Table of Contents，或单击窗口工具栏中的 Table of Contents 图标。比如，NSString 类参考页面就有一个目录窗格，它链接到该页面中的所有主题与方法。一些文档页面会通过目录来展示页面在更大规模的页面组中的位置；比如，String 编程指南就包含了多个页面，在查看一个页面时，目录窗格会列出所有的 String 编程指南页面以及每个页面的主题内容。

针对所有文档集（库）的完整的层次目录位于文档窗口的最左侧；如果没有显示，请选择 Editor → Show Library，或单击窗口工具栏中的导航按钮。该层次目录展示了所有的参考文档，同时还有指南与示例代码，并根据主题进行分类。在查看文档页面时，要想在完整的层次目录中看到它，请选择 Editor → Reveal in Library（或从上下文菜单中选择 Reveal in Library）。

如果希望后面再访问某个文档页，那么可以将其标记为书签，方式是选择 Editor → Share → Add Bookmark，单击工具栏中的 Share 按钮并选择 Add Bookmark，或单击文

档页左侧的书签图标（这也是最简单的方式）。书签会显示在文档窗口的左侧，在导航器中与完整的层次目录在一起；如果书签窗格没有显示出来，那么请选择 Editor → Show Bookmarks。可以通过导航器顶部的图标在库窗格与书签窗格之间切换。单击书签窗格中的书签会跳转到文档窗口。书签的管理是非常简单的，同时又很有用：可以重新排列或删除书签。

要在当前的文档页中搜索文本，请使用 Find 菜单命令。Find → Find（Command-F）会弹出一个搜索框，就像在 Safari 中一样。

第三方文档查看器应用，如 Dash（*http://kapeli.com/dash*），提供了比文档窗口更优秀的本地文档集搜索与查看功能。此外，大多数文档都可以通过 Web 浏览器查看，地址是 *http://developer.apple.com*，即 Apple 公司的开发者站点；通过 Web 浏览器可以显示或隐藏页面的各个部分，它包含了对方法与属性的按照字母搜索的索引，甚至还会显示出文档窗口遗漏的信息。

## 8.2 类文档页面

在大多数情况下，你所查找的文档页都是关于某个类的文档。熟知类文档页面所提供的典型特性与信息是非常重要的，下面就来看看吧（如图 8-1 所示）。

在学习某个类时，你可能还想关注相关的条目信息（单击“More related items”链接查看）：

*Inherits from*

父类链的列表，可以链接到相应的类。初学者常犯的一个严重错误就是不阅读父类链的文档。类是从其父类继承下来的，这样你所寻找的某些功能或信息可能位于父类中。你不可能在 UIButton 的类页面中找到 `addTarget:action:forControlEvents:`，因为该信息位于 UIControl 类页面中。同样不可能在 UIButton 的类页面中找到 frame 属性，因为该信息位于 UIView 类页面中。

*Conforms to*

该类所使用的协议列表，可以链接到相应的协议。不查看所使用的协议信息是初学者常犯的一个严重错误。比如，你不会在 UIViewController 类文档页面中看到 UIViewController 有一个 `viewWillTransitionToSize:withTransitionCoordinator:` 事件；要查看 UIContentContainer 协议的文档，它是 UIViewController 所使用的协议。

*Framework*

声明该类属于哪个框架。要想使用这个类，代码需要链接到该框架并导入框架的头文件；在 Swift 中，通过模块名导入框架就足够了（参见第 6 章）。

*Availability*

表明实现该类的最早的操作系统版本。比如，UIView `layoutGuides` 属性是个 UILayoutGuide 对象数组。不过 UILayoutGuide 是 iOS 9 才引入进来的。如果想要在应用中使用该特性，你需要确保应用针对的目标是 iOS 9 或更新的版本，或当应用运行在老版本的系统上时，代码不会用到这个类。

*Declared in*

声明该类的头文件。遗憾的是，它并非链接；我还没有找到从文档中查看头文件的便捷方式。这确实很遗憾，因为我们经常需要查看头文件，它可能包含了一些有价值的注释或其他细节信息。可以从项目窗口中打开头文件，本章后面将会介绍。

*Related documents*

如果类文档页面列出了相关指南，那么可以单击链接并阅读指南。比如，UIView 类文档页面列出了（也会链接到）*View Programming Guide for iOS*。指南会涵盖广泛的主题；它们提供了重要的信息（常常包含一些有价值的代码示例），可用于指导你的思考方向。

类文档页面划分为多个部分，它们都列在了目录窗格中：

*Overview*

一些类文档页面在 Overview 部分提供了非常重要的介绍性信息，包括对相关指南的链接以及进一步信息（比如，UIView 的类文档页面）。

*Tasks*

这部分会按照类别列出该类的属性与方法。

*Constants*

很多类都针对特定的方法定义了一些常量。比如，在 UIButton 类文档页面中，你会发现要想通过代码创建 UIButton 实例，可以调用 `init(type:)` 初始化器；参数值列在了 Constants 部分的 `UIButtonType` 下面。

最后谈谈类文档页面是如何介绍其属性与方法的。最近几年，这部分文档变得越来越好了，提供了很多超链接。如下部分位于属性或方法名后面：

*Description*

简要介绍属性或方法的作用。

*Declaration*

介绍方法参数与返回类型等信息。

*Parameters and Return Value*

详细介绍参数与返回值的含义与目的。

*Discussion*

通常包含关于方法行为的重要的细节信息。请重视这部分内容！

*Availability*

随着操作系统的不断发展，过去的类可能会添加新的方法；如果某个新方法对于应用很重要，那就需要确保应用不会在没有实现该方法的老操作系统上运行。

*See Also*

指向相关方法与属性的链接。有助于你从宏观上了解该方法对于类的总体行为的意义。

通过类别（参见第10章）注入类中的方法通常不会显示在类的文档页中，也很难找到。比如，`awakeFromNib`（参见第7章）并未在UIButton以及其父类和协议的文档中提及。这是Apple在文档组织上的一个主要缺点。

## 8.3 示例代码

Apple提供了很多示例代码项目，列在了文档窗口的完整目录中（Editor → Show Library）。可以在文档窗口中直接查看代码；有时这么做就够了，但这样做你只能一次查看一个文件，因此很难做到全盘掌控。另一种方式就是在Xcode中打开示例代码项目；单击文档窗口示例代码页顶部的Open Project链接。如果是在浏览器中通过访问 *http://developer.apple.com* 来查看示例代码，那么页面上会有一个Download Sample Code按钮。在项目窗口中打开示例代码项目后，你可以阅读代码，在代码间导航、编辑，当然还可以运行项目。

作为文档的一种形式，示例代码可谓是毁誉参半。它可以作为绝佳的工作代码来源，可以将其复制并粘贴到自己的项目中，只需做很少的改动即可。通常其注释会很多，因为Apple的工程师认为当他们在编写代码时，他们所写的代码主要起到了指导目的。示例代码还阐述了用户很难从文档中挖掘出来的概念（比如，没有掌握UITouch处理的用户经常发现在探索MoveMe示例时会出现灯泡）。但项目逻辑却经常散落在多个文件中，没有什么是比读懂别人写的代码更难的事情了（或许除了你自己编写的代码）。除此之外，学习者最需要的并不是编写完毕的项目，而是构建项目的合理化过程，而这些内容并非是注释所能提供的。

我认为Apple的示例代码并不是那么完美无瑕。有些代码中有疏漏，甚至还有错误；有一些则非常棒。不过一般来说，这些示例代码还是经过深思熟虑且颇具指导意义的，占

据了文档中的相当一部分比重；我们要充分利用好这些示例代码。但我觉得只有在你具备了一定的能力后这些代码才能发挥出最大的功效。

## 8.4 快速帮助

快速帮助是关于某个主题的缩略文档，主题通常是个符号名。它与当前所选或插入点有关，如果快速帮助查看器打开了，那么它就会自动出现在快速帮助查看器中（Command-Option-2）。比如，如果你正在编辑代码，而插入点或所选内容位于 `CGPointMake` 中，那么 `CGPointMake` 的文档就会出现在快速帮助查看器中（如果查看器可见）。

如果在 nib 编辑器中选择了界面对象（编辑项目或目标的同时又在构建设置）并打开了快速帮助查看器，那么也可以使用快速帮助。

快速帮助文档还可以显示为一个弹出窗口，这样就无须使用快速帮助查看器了。选中某个词并选择 Help → Quick Help for Selected Item（Command-Control-Shift-?）。此外，还可以按下 Option 键并将鼠标指针悬浮在某个词上，直到鼠标指针变成一个问号（这个词会变成蓝色，同时会有一个点下划线）；然后按住 Option 键并单击这个词。

在编写 Swift 代码时，快速帮助是非常重要的。如果单击其类型已经推断出来的某个 Swift 变量的名字，那么快速帮助就会显示出推断出的类型（如图 3-1 所示）。这有助于理解编译错误和其他问题。

快速帮助文档还包含了链接。比如，单击 Reference 链接会在文档窗口中打开整个文档。

可以将自己编写的代码的文档加到快速帮助中。要做到这一点，请在声明前加上注释 `/**...*/`（此外，还可以使用以 `///` 开头的单行注释）。可以在注释中使用 Markdown 格式（参见 *http://daringfireball.net/projects/markdown/syntax*）；使用 Markdown 是 Xcode 7 新增的功能。注释会变成快速帮助的描述部分；某些列表项（以 * 或是 - 开头，后跟空格的段落）会被特殊对待：

- 以“Parameter [paramname]:”开头的段落会成为 Parameters 域的一部分。
- 以“Throws:”开头的段落会成为 Throws 域的一部分。
- 以“Returns:”开头的段落会成为 Returns 域的一部分。

比如，下面是带有前置注释的函数声明：

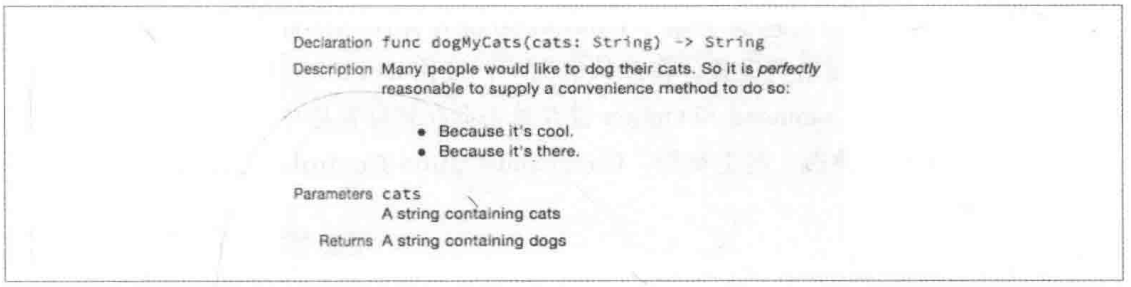


图 8-2：将自定义文档添加到快速帮助中

```
/**
Many people would like to dog their cats. So it is *perfectly*
reasonable to supply a convenience method to do so:

* Because it's cool.
* Because it's there.

* Parameter cats: A string containing cats

* Returns: A string containing dogs
*/

func dogMyCats(cats:String) -> String {
    return "Dogs"
}
```

注释起始处的两个星号表示这是个文档，注释的位置会自动将其关联到 dogMyCats 方法上。星号所包围的单词会被格式化为斜体；星号段落会变成无序列表；最后两个段落会成为特殊域。结果就是在代码中选中了 dogMyCats 后，其实现会显示在快速帮助中（如图 8-2 所示）。说明的第一部分也会显示在代码完成部分中（参见第 9 章）。

# 8.5 符号

符号指的是某个声明的词，如函数名、变量或对象类型等。如果在 Xcode 的代码中能够看到符号名，那就可以快速跳转到该符号声明处。选中文本，然后选择 Navigate → Jump to Definition（Command-Control-J）。此外，还可以按下 Command 键并将鼠标指针悬浮在某个词上，直到鼠标指针变成一个手指形状（这个词会变成蓝色，同时会有一个点状下划线）；按住 Command 键并单击这个词即可跳转到符号声明处，这时：

- 如果符号定义在自己编写的代码中，那么你就会跳转到其声明处；这不但对于理解代码很有帮助，而且对于代码的导航颇具价值。
- 如果符号声明在框架中，那就会跳转到头文件的声明处。如果从 .swift 文件开始，那么你所跳转到的头文件就会转换为 Swift（8.6 节将会介绍头文件）。

概念“跳转”的精确含义取决于除了 Command 键外你所用的修饰键，也取决于你在 Xcode 首选项中导航窗格的设置。在默认情况下，按住 Command 键并单击会在同一个编辑器中跳转，按住 Command 与 Option 键并单击会在辅助窗格中跳转，按住 Command 并双击会在新窗口中跳转。与之类似，Command-Option-Control-J 会在所选词声明的辅助窗格中跳转。

查看项目符号列表并导航到符号声明处的另一种方式是使用符号导航器（参见第 6 章）。如果过滤栏中的第 2 个图标是高亮的，那就说明项目中存在声明的符号；如果没有，那么来自于导入框架中的符号也会列出来。

要想跳转到名字已知的符号声明处，即便之前没有在代码中看到这个名字，你也可以选择 File → Open Quickly（Command-Shift-O）。在搜索框中，输入名字的主要字母，Xcode 会智能地对其进行解析；比如，要搜索 `application:didFinishLaunchingWithOptions:`，可以输入 `appdidf`。可能的匹配项会列在搜索框下面的滚动列表中；你可以通过鼠标或键盘导航该列表。除了来自于框架头文件的声明，自己代码中的声明也会列出来，因此这是个快速导航代码的方式。

## 8.6 头文件

通常，头文件可以作为文档的一种形式，而且可能是最有价值的一种文档形式。头文件一定是精确、最新且完备的；而类文档却不一定。首先，头文件包含了声明，但还有可能包含一些颇具价值的信息；这也会提供类文档可能不会提供的信息。此外，单个头文件可以包含多个类接口和协议的声明。因此它是非常棒的快速参考。

进入头文件的最简单方式就是跳转到那儿的符号声明处。比如，要想进入 *NSString.h*（`Foundation.NSString` 头文件），请按住 Command 键并单击代码中出现的 NSString。请参考 8.5 节了解跳转到符号声明的各种方式；大多数符号都声明在头文件中，因此这些也是跳转到头文件的方式。

在从代码跳转到头文件时，如果代码是个 Swift 文件，那么头文件（如果使用 Objective-C 编写）会自动转换为 Swift。这很棒，因为通过它可以了解到在 Swift 中可以做什么。不过如果希望看看实际的 Objective-C 头文件，情况就不那么妙了！在 Xcode 7 中，可以从 Swift 转换（生成）的头文件切换至原始的 Objective-C，方式是选择 Navigate → Jump to Original Source（或在跳转栏左侧的 Related Items 菜单中选择 Original Source）。

可以通过查看 Swift 头文件来了解关于 Swift 语言与内建库函数的更多信息。此外，还有针对 Core Graphics 与 Foundation 的特殊的 Swift 头文件。

一个有用的技巧是编写一个 `import` 语句，这样就可以按住 Command 键进入头文件了。比如，如果在 *.swift* 文件顶部导入了 `Swift`，那么单词 `Swift` 本身就是个符号，可以按住 Command 键并单击它跳转到 `Swift` 头文件。

## 8.7 互联网资源

自从互联网出现以及 Google 开始对其索引后，编程变得简单多了。你几乎可以通过 Google 搜索找到所需的任何内容。你所遇到的问题别人可能已经遇到过了，并且将解决方案发布到了网上。通常，你可以寻找一些示例代码并粘贴到项目中，然后做些修改即可。

Apple 的文档资源位于 *http://developer.apple.com/library/*。在 Apple 修改文档集并提供下载前会更新这些资源。上面还有一些资源并不属于计算机中 Xcode 文档的一部分。比如，可以下载 WWDC 2015 会议的视频（以及更早之前的视频）。

Apple 还提供了一些开发者论坛，地址是 *https://forums.developer.apple.com*。这里会有一些有趣的讨论，Apple 员工也会参与进去，不过，论坛界面做得非常差劲。

其他的在线资源随着 iOS 编程的流行如雨后春笋般涌现了出来，众多的 iOS 与 Cocoa 程序员都将经验分享到了博客上。我特别中意的是 Stack Overflow（*http://www.stackoverflow.com*）。当然，它并不是专门为 iOS 编程而设的，但众多的 iOS 程序员都在那儿，众多问题的回答也都是简洁而正确的，同时，其界面能让你快速而轻松地将精力放在正确的答案上。

# 第 9 章

# 项目的生命周期

本章将会介绍 Xcode 项目生命周期的几个主要阶段，从一开始到提交到 App Store。此外，还会介绍 Xcode 开发环境的一些附加特性：配置构建设置与 *Info.plist*；编辑、调试与测试代码；在设备上运行应用；为提交到 App Store 做分析、本地化与最终的准备工作。

## 9.1 设备架构与条件代码

在创建项目时（File → New → Project），当选择好项目模板后，在项目命名界面上会弹出 Devices 菜单，提供了 iPad、iPhone 与 Universal 选项。可以稍后修改这个设置，在编辑应用目标时使用 General 页签中的 Devices 弹出菜单；不过如果这里就能做出正确的决定，那么情况会变得更加简单，因为你的决定会影响新项目所使用的模板细节信息。在 Devices 弹出菜单中所做的选择还会影响项目的 Targeted Device Family 构建设置：

1（*iPhone*）
: 应用可以运行在 iPhone 或 iPod touch 上；还可以运行在 iPad 上，但并不是作为原生 iPad 应用运行（它会运行在一个简化的、可放大的窗口中，称为 iPhone 模拟器；Apple 有时称为“兼容模式”）。

2（*iPad*）
: 应用只会运行在 iPad 上。

1，2（*通用*）
: 应用可以运行在这两种设备上。

有两个项目级的构建设置可以决定设备运行在什么系统上：

*Base SDK*

应用可运行的最新系统。本书编写之际，在 Xcode 7.0 中，你有两个选择：iOS 9.0 与 Latest iOS (iOS 9.0)。它们看起来是一样的，但后者会更好一些（也是新项目的默认值）。如果更新 Xcode 来开发后续系统，那么已经设为 Latest iOS 的现有项目都会自动将新系统的 SDK 作为 Base SDK，你不必手工更新 Base SDK 设置。

*iOS Deployment Target*

应用可运行的最老的系统：在 Xcode 7 中，这可以一直追溯到 iOS 6.0。要想修改项目的 iOS Deployment Target 设置，请编辑项目并切换至 Info 页签，然后从 iOS Deployment Target 弹出菜单中进行选择。

## 9.1.1 向后兼容

如果应用的 Deployment Target 不同于 Base SDK（也就是说，应用要兼容于老版本的系统），那就是一件很有挑战的事情。主要会有两个问题：

*改变的行为*

对于每个新系统来说，Apple 都可能会改变一些特性的运作方式。结果就是不同系统上的一些特性可能会根据系统的不同而表现出不同的行为。一整块的功能可能会在不同的系统上得到不同的处理，这要求你实现或调用不同的方法，或使用完全不同的类来处理。甚至有可能相同的方法会表现出完全不同的行为，这取决于应用运行在什么系统上。

*不支持的特性*

对于每个新系统，Apple 都会增加一些新特性。如果在执行时遇到了系统不支持的特性，那么应用就会崩溃。

改变的行为是非常麻烦的事情，这里也没什么更好的建议。通常，问题要么是完全破坏的行为，要么是先破坏，后来又修复了。比如，使用了 UIProgressView 的 `progressImage` 属性的代码在 iOS 7 上可以正常运行，但却无法运行在 iOS 7.1 到 iOS 8.4 上，不过在 iOS 9 上又可以正常使用了。除了尝试然后根据错误来修正外，别无他法，这总是非常棘手的一个问题。

在 iOS 7 及之前的版本中，警告视图是通过 UIAlertView 呈现的，不过在 iOS 8 及之后的版本中变成了 UIAlertController。最简单的解决方案就是即便在 iOS 8 及之后的版本中也还是继续使用 UIAlertView，不过你无法保证这么做总是可行的，因为 UIAlertView 在 iOS 9 中已经标记为不建议使用，最终可能会被丢弃掉，你还失去了使用 UIAlertController 的机会，它是个更棒的 API。弹出框也是类似的（UIPopoverController

与 UIPopoverPresentationController）。这样，系统的更迭与改进就会给开发者造成这样一种困境：这些改进是开发者所需要的，但它们会导致向后兼容变得更加困难。

不过在 Xcode 7 中，编译器至少提供了之前所没有的一个功能，它使得代码很难在不支持某个特性的目标系统上使用该特性。在 Xcode 7 之前，如果将项目的 Deployment Target 设为一个老系统，那么代码是可以编译通过的，应用也可以运行在这个老系统之上的，即便代码中包含了老系统上并不存在的特性亦如此；如果遇到了这些特性，那么应用就会崩溃。在 Xcode 7 中，编译器会在一开始就防止这种情况的出现。

比如：

```
let arr = self.view.layoutGuides
```

UIView layoutGuides 属性只存在于 iOS 9.0 及之后的版本中。之前，即便部署目标设为了 iOS 8.0，编译器还是会允许该代码编译通过；你要确保代码绝对不能运行在 iOS 8 上。不过现在，编译器会阻止你这么做，并报错：“layoutGuides is only available on iOS 9.0 or newer”。除非你告诉编译器代码只会运行在 iOS 9 及之后的版本中，否则是无法继续下去的。Xcode 的 Fix-It 特性会告诉你该如何做：

```
if #available(iOS 9.0, *) {
    let arr = self.view.layoutGuides
} else {
    // Fallback on earlier versions
}
```

`#available` 条件（一个可用性检查）会比较当前系统与声明中所指定的特性要求。`layoutGuides` 属性声明前有如下注解（在 Swift 中）：

```
@available(iOS 9.0, *)
```

请查看文档来了解该注解的具体信息。不过，你无须理解它！`#available` 条件会匹配该注解，Xcode 的 Fix-It 会确保如此。可以在 `if` 条件或 `guard` 条件中使用 `#available`。

可以使用 `@available` 特性来注解自己的类型和成员声明，代码接下来还需要进行可用性检查。比如，如果方法声明使用了 `@available(iOS 9.0, *)`，那么当部署目标早于 iOS 9 时，就无法调用该方法了，这里也无须进行可用性检查。在该方法中，无须再进行 `#available(iOS 9.0, *)` 可用性检查，因为你已经确保该方法不会运行在 iOS 9 之前的系统上。

要想在老系统上测试应用，你需要一个运行着老系统的设备（物理设备或模拟器）。可以通过 Xcode 的 Downloads 首选项窗格下载 iOS 8 SDK（参见第 6 章），不过要想测试更老版本的系统，你需要更老版本的 Xcode，最好还要有一个更老的设备。请不要向 App Store 提交尚未在某个运行系统上测试过的应用。

## 9.1.2 设备类型

对于通用应用，能够知道代码运行在 iPad 还是 iPhone 或 iPod 上是很有用的。通过当前的 UIDevice（或层次体系中任何 UIViewController、UIView 的 `traitCollection`）可以获得当前设备的类型并作为 `userInterfaceIdiom` 返回给你，在 iPhone 上是 `UIUserInterfaceIdiom.Phone`，在 iPad 上则是 `UIUserInterfaceIdiom.Pad`。

可以根据设备类型或屏幕分辨率有条件地加载资源。对于从应用包顶层加载的图片，可以使用名字后缀，如 `@2x` 和 `@3x` 来表示屏幕分辨率，或 `~iphone` 和 `~ipad` 来表示设备类型；不过，更简单的做法则是使用资源类别，在 Xcode 7 与 iOS 9 中，可以针对任何类型的数据资源采取这种做法。

与之类似，某些 *Info.plist* 设置带有名字后缀，这样就可以在一种设备类型上使用一种设置，在另外的设备类型上使用另一种设置。比如，对于通用应用，常见的做法是在 iPhone 上使用一套方向，在 iPad 上使用另一套：一般来说，iPhone 版本只允许有限的方向，iPad 版本则允许所有方向。可以在编辑目标时通过 General 窗格来配置：

1. 将 Devices 弹出菜单切换至 iPhone，并勾选 iPhone 上所需要的 Device Orientation 复选框。
2. 将 Devices 弹出菜单切换至 iPad，并勾选 iPad 上所需要的 Device Orientation 复选框。
3. 将 Devices 弹出菜单切换至 Universal。

即便现在只看到一套方向，但实际上这两套方向都会保存起来。实际上，你所做的是在 *Info.plist* 中配置了两组“Supported interface orientations”设置，一组通用设置（`ISupportedInterfaceOrientations`），一组针对 iPad 的设置，当应用在 iPad 上运行时，它会覆盖通用设置（`UISupportedInterfaceOrientations~ipad`）。可以查看 *Info.plist* 文件了解详情。

按照相同的方式，应用可以加载不同的 nib 文件，因此也会显示不同的界面，这取决于设备的类型。比如，你可以拥有两个主故事板，如果在 iPhone 上运行，那么应用启动时就加载其中一个；如果在 iPad 上运行，那就加载另一个。编辑目标时依然可以通过 General 窗格进行配置。实际上，你所做的是让 *Info.plist* 设置“Main storyboard file base

name”出现两次，一次针对通用情况（UIMainStoryboardFile），另一次针对iPad（UIMainStoryboardFile~ipad）。如果应用根据名字加载nib，那么该nib文件的命名就会像图片文件一样：如果运行在iPad上，并且拥有相同的名字且后缀为~ipad的nib文件，那么它就会被自动加载。

不过，现在很少需要区分设备类型了。在iOS 7及之前的版本中，整个界面对象类（如弹出框）都只能在iPad上使用；iOS 8及后续版本中已经不存在只针对iPad的类了，如果代码运行在iPhone上，界面类本身会自适应。与之类似，在iOS 7及之前的版本中，通用应用可能需要完全不同的界面，因此根据设备类型的不同需要不同的nib文件；在iOS 8及后续版本中，可以通过尺寸等级根据设备类型的不同有条件地配置nib文件。一般来说，应用在iPad与iPhone上的物理差别并不像过去那么明显：这要归功于尺寸居于二者之间的iPhone 6，特别是iPhone 6 Plus，使得尺寸看起来更加连贯。

## 9.2 版本控制

对于实际的应用，应该尽早将其纳入版本控制下。版本控制是获取项目周期性快照的一种方式。其目的是：

*安全*

版本控制可以帮助你将提交存储到仓库中，这样代码就不会因为计算机故障或其他原因丢失了。

*协作*

版本控制支持多人开发，让大家合理地访问相同的代码。

*放心*

项目是个复杂的事物；有时需要做一些试验性质的修改，这可能涉及很多文件，可能经过很多天，然后才能测试新的特性。借助版本控制，如果出问题了，我可以轻松追踪每一步操作（之前的提交）；这样我就有信心做一些经过一段时间才能看到结果的试验。此外，如果被一些试验搞乱了，我可以通过版本控制系统列出最近做出的所有变更。如果出现了Bug，我可以通过版本控制系统查明何时出现的Bug并探查原因。

Xcode提供了各种版本控制设施，主要面向Git（*http://git-scm.com*）与Subversion（*http://subversion.apache.org*，也叫作svn）。但这并不表示你无法在项目中使用其他版本控制系统；这只意味着你无法在Xcode中以集成的方式使用其他版本控制系统。没关系；还有很多其他方式可以使用版本控制，甚至是Git与Subversion。我们可以不使用Xcode的集成版本控制，转而使用Terminal命令行，或使用专门的第三方GUI前端，比如，

面向 Subversion 的 svnX（*http://www.lachoseinteractive.net/en/products*），或面向 Git 的 SourceTree（*http://www.sourcetreeapp.com*）。

如果不想使用 Xcode 的集成版本控制，那么你可以将其关闭。如果未勾选 Source Control 首选项窗格中的 Enable Source Control，那么你只能从 Source Control 菜单中选择 Check Out，从远程服务器获取代码。如果勾选了 Enable Source Control，那么还有 3 个复选框可以使用，它们决定了你想要哪一种自动行为。从个人角度来说，我喜欢勾选 Enable Source Control 与“Refresh local status automatically”，这样 Xcode 会在项目导航器中显示出文件的状态；剩下的两个复选框我没有勾选，这是因为我喜欢自己控制。

在新建项目时，Save 对话框中有一个复选框，可以在一开始就在项目目录中创建一个 Git 仓库。这个仓库可以在自己的计算机上，也可以选择远程服务器。如果没有特别的原因，建议勾选这个复选框！

当打开已有的项目时，如果项目已经由 Subversion 或 Git 管理，那么 Xcode 会检测到这一点，并且会立刻在界面上显示出版本控制信息。如果使用的是远程仓库，那么 Xcode 会自动在 Accounts 首选项窗格中输入信息，这个窗格是仓库管理的统一界面。要想使用远程服务器，但又没有工作副本，那么请在 Accounts 首选项窗格中手工输入信息。

可以在两个地方使用源控制动作：Source Control 菜单与项目导航器中的上下文菜单。要想检出并打开存储在远程服务器上的项目，请选择 Source Control → Check Out。Source Control 中的其他项都是显而易见的，如 Commit、Push、Pull（或 Update）、Refresh Status 及 Discard Changes。值得注意的是 Source Control 菜单中的第一项，它会根据名字与分支列出所有打开的工作副本；可以通过其层次化菜单项进行基本的分支管理。

项目导航器中的文件会根据状态进行标记。比如，如果使用 Git，那么可以区分出修改的文件（M）、新的未追踪文件（?）以及添加到索引中的新文件（A）（如果没有勾选“Refresh local status automatically”，那么这些标记就都不会出现，除非选择了 Source Control → Refresh Status）。

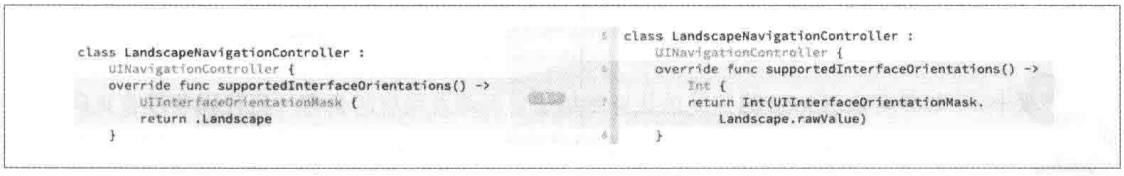


图 9-1：版本比较

如果选择了 Source Control → Commit，Xcode 会弹出一个比较视图，列出所有文件中出现的所有变更。每个变更都可以从此次提交中排除（或完全恢复），这样就可以将相关的文件分组到有意义的提交中。选择 Source Control → History 也会弹出一个类似的比较

视图（不过 Xcode 并未提供类似于 Git 自己的 gitk 工具这样的可视化分支展示工具）。合并冲突也可以通过一个图形化的比较界面来完成。

还可以通过版本编辑器在任何时刻查看当前正在编辑的文件的比较视图；方式是选择 View → Version Editor → Show Version Editor 或单击项目窗口工具栏中第 3 个 Editor 按钮。版本编辑器实际上有 3 种模式：比较视图、Blame 视图与日志视图（从 View → Version Editor 选择，或使用工具栏第 3 个 Editor 按钮的弹出菜单）。

比如，在图 9-1 中，我可以看到在该文件的最新版本（位于左侧）中对 `supportedInterfaceOrientations` 实现所做的修改（因为 Swift 语言发生了变化）。如果选择了 Editor → Copy Source Changes，那么相应的 diff 文本（一个补丁文件）就会被放到剪贴板中。如果切换到 Blame 视图，我就可以在编辑器中看到当前文件的所有提交版本。

还有一种方式可以查看某一行代码是如何修改的，方式是选中该行（在正常的编辑器中），然后选择 Editor → Show Blame For Line。这时会弹出一个窗口，描述了将这行文本修改为当前内容时的提交信息；可以通过弹出窗口中的按钮切换至 Blame 视图或比较视图。

## 9.3 编辑与代码导航

Xcode 编辑环境的很多地方都可以修改以满足你的需要。首先应该在 Xcode 的 Fonts & Colors 首选项窗格中选择喜欢的源码编辑器字体和大小。没什么是比舒服地阅读和编写代码更重要的事情了！我喜欢稍大点（13、14，甚至是 16）和等宽字体，如 Menlo、Consolas 或免费的 Inconsolata（*http://levien.com/type/myfonts/*）与 Source Code Pro（*https://github.com/adobe-fonts/source-code-pro*）。

Xcode 提供了一些自动格式化、自动输入与文本选择特性。其行为取决于你在 Xcode 的 Text Editing 首选项窗格的 Editing and Indentation 页签中的设置。这里并不打算详细介绍这些设置，但建议你充分利用它们。在 Editing 下，我习惯勾上所有选项，包括行号；可见的行号在调试时是非常有用的。在 Indentation 下，我也习惯勾上所有选项；我发现在这些设置下，Xcode 能以最佳的方式显示代码。

如果喜欢 Xcode 的智能语法感知缩进，但却发现有时会有一行代码并没有正确缩进，那么请选择 Editor → Structure → Reindent（Control-I 组合键），这会自动缩进当前行或所选文本。

勾选 “Enable type-over completions” 后，Xcode 会自动加上分隔符。比如，假设我通过调用初始化器 `init(frame:)` 来创建一个 UIView。我会这么写：

```
let v = UIView(fr
```

Xcode 会自动追加上右圆括号，同时插入点还在右圆括号之前：

```
let v = UIView(fr)
// I have typed ^
```

不过，这个右圆括号是试探性的；其颜色是灰色。现在输入参数；输入完后右圆括号依然是灰色的：

```
let v = UIView(frame:r)
// I have typed ^
```

现在可以通过几种方式来确认右圆括号：可以输入一个右圆括号，也可以按下 Tab 键或向右的方向键。这时，试探性的右圆括号会被实际的替换掉，插入点位于其之后了，准备接受后续输入。双引号、右花括号以及右方括号的行为与之类似。

## 9.3.1 自动补令

在编写代码时，你会用到 Xcode 的自动补令特性。Cocoa 类型名与方法名非常冗长，无论什么，只要能节省你输入代码的时间都是好事。然而，我并没有选中 Editing 下面的“Suggest completions while typing”；相反，我勾选了“Use Escape key to show completion suggestions”，当我希望自动补令时，我会手工实现，通过按下 Esc 键。

比如，我要通过代码创建一个警告框。我需要输入 `UIAlertController`（按下 Esc 会弹出一个菜单，列出适合 UIAlertController 的 4 个初始化器，如图 9-2 所示）。可以在菜单中导航、关闭它或接受所选，只需使用键盘即可。如果默认情况下没有选中，那么我会通过向下的方向键导航到 `title:...`，然后按下回车键将其选中。

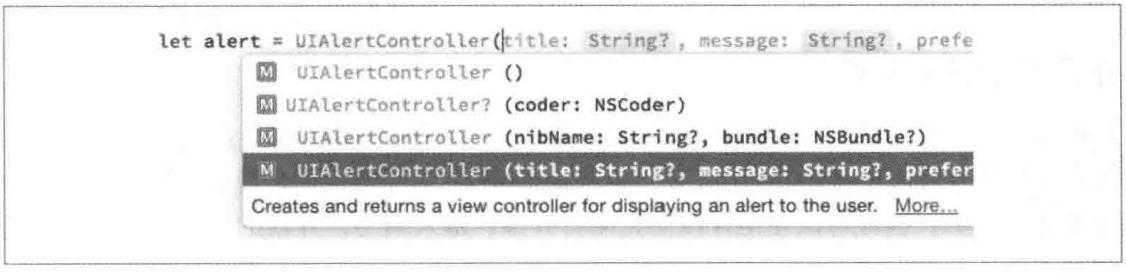


图 9-2：自动补令菜单

从自动补令菜单中选择后，所选方法调用的模板就会输入代码中（这里将其分解为多行显示）：

```
let alert = UIAlertController(
    title: <#String?#>,
```

```
    message: <#String?#>,
    preferredStyle: <#UIAlertControllerStyle#>)
```

<#...#> 中的表达式是占位符，展示了每个参数的类型；它们在 Xcode 中就像是“文本标记”一样（如图 9-2 所示），防止你不小心修改。可以通过 Tab 键或 Navigate → Jump to Next Placeholder（Control-/ 组合键）选择下一个占位符。这样就可以选择一个占位符，然后在上面输入想要传递的实参；接下来选择下一个占位符并输入其实参，以此类推。要想将占位符转换为一般的字符串且没有分隔符，可以将其选中并按下回车键，或双击它。

自动补令与上下文智能感知可用于对象类型名、方法调用与属性名。在输入函数声明时，如果这个函数是继承下来的，或定义在所使用的协议中，那也可以使用自动补令。你甚至都不需要输入起始的 func；只需要输入方法名的前几个字母即可。比如，在我的应用委托类中，我会输入：

```
applic
```

如果按下了 Esc 键，那么我会看到一个方法列表，比如，application:didFinishLaunchWithOptions:，这些是可以发送给应用委托的方法（第 11 章将会介绍）。如果选择了一个，那么其整个声明都会输入进来，包括花括号：

```
func application(application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [NSObject : AnyObject]?)
    -> Bool {
        <#code#>
}
```

代码占位符位于花括号之间，它会被选中，等待着我开始输入函数体。如果函数需要一个 override 标识，那么 Xcode 的代码补令特性会提供的。

## 9.3.2 代码片段

代码片段是代码自动补令的有益补充。代码片段就是个带有缩写的一段文本。代码片段保存在代码片段库中（Command-Option-Control-2），不过代码片段的缩写对于代码补令却是全局的，因此可以使用片段而无须打开库：输入缩写，片段的名字就会出现在代码补令中。

比如，要想在文件顶部输入类的声明，我会输入 class 并按下 Esc 键来打开自动补令，然后选择“Swift Class”或“Swift Subclass”。按下回车键后，模板就会出现在代码中：类名与父类名是占位符，同时还有花括号，声明体（位于花括号中）则是另一个占位符。

要了解代码片段的缩写，需要打开其编辑窗口（在代码片段库中双击代码片段）并单击Edit。如果觉得记住代码片段的缩写太麻烦，可以将其从代码片段库中拖曳到文本中。可以通过过滤栏（Edit → Filter → Filter in Library，Command-Option-L 组合键）根据名字快速找到所需的代码片段。

可以添加自己的代码片段，它会划分到 User 代码片段类别中；最简单的方式就是将文本拖曳到代码片段库中。然后编辑它以适应自己的需要，给它起个名字，提供一段说明和一个缩写；可以通过 Completion Scopes 弹出菜单缩小代码补令所显示的片段上下文。在代码片段文本中，使用 <#...#> 结构来构造任何所需的占位符。

比如，我创建了一个插座变量代码片段，如下所示：

```
@IBOutlet var <#name#> : <#type#>!
```

我又创建了一个动作代码片段，如下所示：

```
@IBAction func <#name#> (sender:AnyObject!) {
    <#code#>
}
```

我编写的其他代码片段构成了一个个人的辅助函数库。比如，delay 代码片段会插入 dispatch_after 包装器函数（参见 11.10 节）。

## 9.3.3 Fix-it 与实时语法检查

Xcode 的 Fix-it 特性会针对如何避免问题给出一些积极的建议。要弹出它，请单击左边栏的问题标记。编译后如果有问题会显示出这种问题标记。

比如，图 9-3 顶部显示我不小心丢掉了方法调用后的圆括号。这会导致编译错误，因为我设置的 backgroundColor 属性是个 UIColor，它不是函数。不过，错误旁边的停止图标告诉我 Fix-it 有个建议。单击这个停止图标，图 9-3 底部显示了发生的事情：弹出一个 Fix-it 对话框，告诉我该如何修复这个问题，即插入圆括号。此外，Xcode 也告诉我如果 Fix-it 按照这种方式修复了问题，那么代码会变成什么样子。如果按下回车键，或双击对话框中的“Fix-it”建议，Xcode 就会插入圆括号，错误会消失不见，因为问题已经得到了解决。

如果相信 Xcode 的做法是正确的，那么请选择 Editor → Fix All in Scope（Command-Option-Control-F 组合键），Xcode 会实现周围所有的 Fix-it 建议，并且不会再弹出对话框。

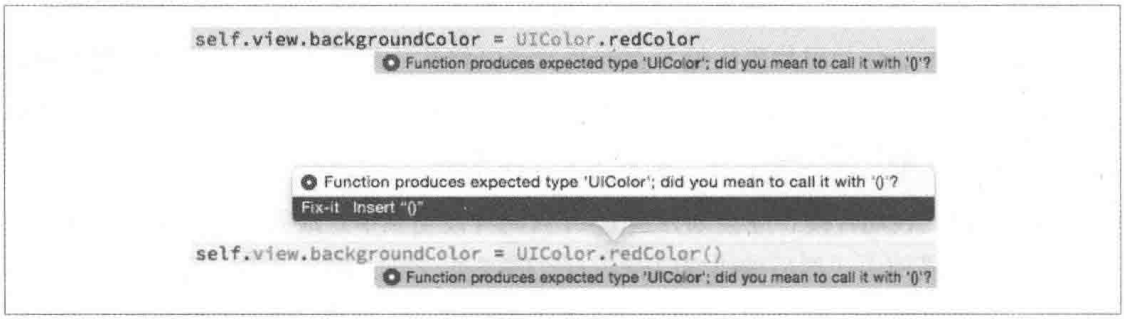


图 9-3：带有 Fix-it 建议的编译错误

实时语法检查像是一种持续不断的编译。即便没有编译或保存，它也可以检测到存在的问题，并且通过 Fix-it 给出建议的解决方案。可以通过 General 首选项窗格中的“Show live issues”复选框打开或关闭该特性。

我觉得实时语法检查会影响代码编写过程。在编写过程中，代码几乎不可能是合法的，因为单词与圆括号总是半成品；我准备输入这些内容！比如，仅仅输入 let 的首字母就会导致语法检查器报告无法解析的标识符错误；我非常讨厌这一点。因此，我并没有勾选“Show live issues”复选框。

## 9.3.4 导航

开发 Xcode 项目需要在多个文件中同时编辑代码。幸好，Xcode 提供了多种方式来导航代码。前几章已经介绍了一些。下面是 Xcode 提供的一些主要的导航方式：

*项目导航器*

如果你记得文件名的一部分，那么就可以在项目导航器（Command-1 组合键）中快速定位到该文件，通过在导航器底部过滤栏中的搜索框内搜索即可（Edit → Filter → Filter in Navigator, Command-Option-J 组合键）。比如，输入 story 就会列出 .storyboard 文件。此外，在使用完过滤栏后，你可以按下 Tab 键，然后通过向上、向下的方向箭在项目导航器中导航。这样，只通过键盘就能找到所需的文件了。

*符号导航器*

如果高亮显示了过滤栏中的前两个图标（前两个是蓝色的，第 3 个是暗色的），那么符号导航器就会列出项目的对象类型及其成员。单击符号可以在编辑器中跳转到其声明。与项目导航器一样，过滤栏中的搜索框可以帮助你跳转到想要去的地方。

*跳转栏*

代码编辑器的跳转栏的每个路径组件都是个菜单：

*底部*

跳转栏底部（最右边）是文件中对象与成员声明的列表，按照它们的显示顺序排序（按住 Command 键的同时选择菜单可以按照字母表顺序查看）；可以选择其一进行导航。

可以通过起始单词为 `MARK:` 的注释将加粗的章节标题添加到底部菜单中。比如，修改 Empty Window 项目中的 *ViewController.swift*：

```
// MARK: - view lifecycle
override func viewDidLoad() {
    super.viewDidLoad()
}
```

结果就是底部菜单中的 `viewDidLoad` 会位于 `view lifecycle` 之前。要在菜单中创建分隔符，请输入一条 `MARK:` 注释，其值是连字符；在上述示例中，连字符（创建一个分隔符行）与标题（创建一个加粗标题）都用到了。与之类似，以 `TODO:` 和 `FIXME:` 开头的注释都会出现在底部菜单中。

*上部*

上部路径组件是个层次化菜单；这样你就可以通过它们遍历文件层次了。

*历史*

每个编辑器窗格都记得你曾经编辑的文件名。向后与向前这两个三角形既是按钮也是弹出菜单（或选择 Navigate → Go Back 和 Navigate → Go Forward，分别对应 Command-Control-Left 与 Command-Control-Right 组合键）。

*相关条目*

跳转栏中最左侧的按钮会弹出相关条目菜单，这是与当前文件相关的一个层次化的文件菜单，比如，父类与所使用的协议等。该列表甚至还包含了当前所选函数调用或被它调用的函数。

跳转栏中的路径组件菜单是可以过滤的！打开跳转栏菜单，输入文本来过滤菜单所显示的信息。此处的过滤使用了“智能”搜索而不是严格的文本包含搜索；比如，输入“adf”会找到 `application:didFinishLaunchingWithOptions:`（如果位于菜单中）。

*辅助窗格*

可以通过辅助窗格同时身处两处（参见第 6 章）。按住 Option 键并导航会在辅助窗格而非主编辑器窗格中打开文件。辅助窗格跳转栏中的 Tracking 菜单会设定其与主窗格的自动化关系。

*页签与窗口*

还可以通过打开页签或单独的窗口而同时身处两处（参见第 6 章）。

*跳转到定义*

可以通过 Navigate → Jump to Definition（Command-Control-J 组合键）跳转到代码中所选符号的声明位置处。

*快速打开*

可以通过 File → Open Quickly（Command-Shift-O 组合键）打开一个对话框，并在这里搜索代码和框架头文件中的符号。

*断点*

断点导航器会列出代码中的所有断点。Xcode 缺少代码书签，不过可以将禁用的断点当作书签。本章后面将会介绍断点。

## 9.3.5 查找

查找是导航的一种形式。Xcode 提供了全局查找（Find → Find in Project，Command-Shift-F 组合键），这与使用查找导航器的效果是一样的；还提供了编辑器级别的查找（Find → Find，Command-F 组合键）；不要搞混了。

查找选项是非常重要的。对于编辑器级别的查找，请单击搜索域中的放大镜图标来打开 Edit Find Options 条目。可以搜索单词中间部分或单词开头，指定是否区分大小写等，甚至可以使用正则表达式进行查找。这里有大量的功能！全局查找选项位于搜索框上下，它包含了一个范围，用于指定搜索哪些文件：单击当前范围可以看到 Search Scopes 面板，可以选择不同的范围或创建自定义范围。

搜索框上方的全局查找选项包含了文本、正则表达式、定义（定义符号的地方）与引用（使用符号的地方）。Xcode 7 新增了调用层次查找选项，可以向后追踪代码中的调用关系。单击搜索栏中的第 2 个条目会显示一个弹出菜单，选择 Call Hierarchy 即可；此外，还可以在代码中选中一个词，然后选择 Find → Find Call Hierarchy（Shift-Control-Command-H 组合键）。调用层次会显示在查找导航器中（如图 9-4 所示）。

要替换文本，请单击搜索栏最左侧的 Find 来弹出菜单，然后选择 Replace。可以将单词出现的所有位置都替换掉（Replace All），或在查找导航器中选择特定的搜索结果，然后只替换这些（Replace）；还可以从查找导航器中删除搜索结果，从而使其不会被 Replace All 所影响。查找导航器的 Preview 按钮会弹出一个对话框，展示了每个可能的替换的效果，可以在执行替换前接受或拒绝特定的替换。对于编辑器级别的查找，在单击 Replace All 前按下 Option 键，这样查找与替换就只会对所选文本起作用。

比较复杂的一种编辑器级别的查找形式是 Editor → Edit All In Scope，它会在相同范围内同时查找所选文本所有出现的地方；可以通过它在范围内修改变量或函数的名字，或查看名字是怎么使用的。

图 9-4：查找导航器中的调用层次

# 9.4 在模拟器中运行

在将模拟器作为构建与运行的目标时，你会在 iOS 模拟器中运行应用。模拟器窗口代表一个设备。根据应用目标的 Base SDK、部署目标、目标设备家族的构建设置，以及安装了哪些 SDK，可以在运行前选择模拟器所代表的设备与系统（参见第 6 章）。

模拟器窗口可以显示为各种尺寸：从 Window → Scale 进行选择。这仅仅是个显示问题，类似于缩放窗口。比如，你能以实际大小在模拟器中运行双倍分辨率的设备，从而看清楚每个像素；也能以一半大小运行，从而节省空间。

可以像与设备交互那样与模拟器进行一些基本的交互。借助鼠标，可以轻拍设备的屏幕；按住 Option 键可以让鼠标表示两根手指，沿着中心对称移动；按住 Option 与 Shift 键可以表示两根同时移动的手指。要单击 Home 键，请选择 Hardware → Home（Command-Shift-H 组合键）。还可以通过 Hardware 菜单中的条目执行一些硬件手势，如旋转设备、摇晃设备、锁屏等；也可以通过模拟某些不常出现的事件（如内存不足等）来测试应用。

单击 Home 键从运行在 Xcode 中的应用切换至主屏幕并不会导致应用停止，无论在 Xcode 还是模拟器中均如此。要让模拟器中的应用停止运行，请终止模拟器的运行，或切换到 Xcode 并选择 Product → Stop。

模拟器中的 Debug 菜单有助于检测到动画与绘制方面的问题。打开 Slow Animations，使得动画以很慢的速度出现，这样就能看到动画的细节信息。下面 4 个菜单项（名字以 Color 开头）类似于使用 Instruments 时所用的特性，在 Instruments 中，这些特性位于 Core Animation instrument 下，用于显示出在屏幕绘制时可能的低效之源。

还可以通过 Debug 菜单在 Console 应用中打开日志，并设置模拟设备的位置（在测试 Core Location 应用时很有帮助）。

# 9.5 调试

调试就是在应用运行时寻找其问题的技术。我将这种技术分为两个大的类别：原始调试与暂停运行中的应用。

## 9.5.1 原始调试

原始调试需要修改代码，这通常是临时的，一般是添加一些代码向控制台输出一些信息。可以通过调试窗格查看控制台；第 6 章介绍了如何在自己的页签中显示控制台的技术。

用于向控制台发送消息的标准 Swift 命令是 print 函数。借助 Swift 的字符串插值与 CustomStringConvertible 协议（需要一个 `description` 属性；参见第 4 章），可以向 `print` 调用提供大量有价值的信息。Cocoa 对象通常都有内建的 `description` 属性实现。比如：

```
print(self.view)
```

控制台的输出如下所示（我已经对输出格式化了，便于查看）：

```
<UIView: 0x79121d40;
  frame = (0 0; 320 480);
  autoresize = RM+BM;
  layer = <CALayer: 0x79121eb0>>
```

从中可以看到对象所属的类，其内存地址（用于判断两个实例是否是相同的实例），以及其他一些属性的值。

如果导入了 Foundation（在实际的 iOS 编程中都会导入的），那就可以使用 NSLog C 函数了。它接收一个 NSString 作为格式化字符串，后跟格式化参数。格式化字符串是个包含符号的字符串，这里的符号叫作格式化说明符，其值（格式化参数）会在运行期被替换。所有的格式化说明符都以一个百分号（%）开头，因此在格式化字符串中输入百分号字面值的唯一方法就是使用两个百分号（%%）。百分号后面的字符指定了运行期需要提供的值类型。最常见的格式化说明符是 %@（对象引用）、%d（int）、%ld（long），以及 %f（double）。比如：

```
NSLog("the view: %@", self.view)
```

在该示例中，`self.view` 是第一个，也是唯一一个格式化参数，因此在将格式化字符串输出到控制台时，其值会被第一个，也是唯一一个格式化说明符 %@ 所替换：

```
2015-01-26 10:43:35.314 Empty Window[23702:809945]
  the view: <UIView: 0x7c233b90;
    frame = (0 0; 320 480);
    autoresize = RM+BM;
    layer = <CALayer: 0x7c233d00>>
```

我喜欢NSLog的输出，因为它提供了当前的时间与日期，还有进程名、进程ID以及线程ID（有助于确定两条日志语句是否由相同的线程所调用）。此外，NSLog是线程安全的，而print则不是。

要想查询格式化字符串中可用的全部格式化说明符，请阅读Apple的文档*String Format Specifiers*（在*String Programming Guide*中）。格式化说明符在很大程度上是基于C printf标准库函数的。

使用NSLog（或其他格式化字符串）时常犯的错误就是提供的格式化参数数量与字符串中格式化说明符的数量不一致，或提供的参数值与相应的格式化说明符所声明的类型不一致。我常发现初学者说日志输出的值没有意义，而实际上却是其NSLog调用是没有意义的；比如，格式化说明符是%d，而相应的参数值却是个浮点型。另一个常犯的错误是将NSNumber看作它所包含的数字类型；NSNumber并不是任何一种数字类型，它是个对象（%@）。诸如有符号与无符号整数、32位与64位数字之类的问题都很棘手。

C结构体并非对象，因此它们无法提供description。不过，Swift扩展了最常见的一些C结构体，并形成了Swift结构体，这样就可以使用print输出了。比如，下面这样做是可以的：

```
print(self.view.frame) // (0.0,0.0,320.0,480.0)
```

不过，你不能对NSLog这么做。出于这个原因，常见的Cocoa结构体通常都带有一些便捷函数，用于将其转换为字符串。比如：

```
NSLog("%@", NSStringFromCGRect(self.view.frame)) // {{0, 0}, {320, 480}}
```

Swift定义了4个特殊的字面值，这在记录日志时非常有用，因为它们描述了自己在外部文件中的位置：__FILE__、__LINE__、__COLUMN__与__FUNCTION__。

在发布应用时需要删除日志调用，因为不能让最终的应用向控制台输出不必要的信息。一个技巧就是将自定义的全局函数放到Swift的print函数前：

```
func print(object: Any) {
    Swift.print(object)
}
```

如果不需要记录日志，那么只需注释掉第 2 行即可：

```
func print(object: Any) {
    // Swift.print(object)
}
```

如果希望这一切是自动进行的，那么可以使用条件编译。Swift 的条件编译还不够强大，不过对于这件事已经足够了。比如，我们可以让函数体依赖于 DEBUG 标记：

```
func print(object: Any) {
    #if DEBUG
        Swift.print(object)
    #endif
}
```

上述代码依赖于并不存在的 DEBUG 标记。请在目标构建设置中创建它，位于 Other Swift Flags 下。定义 DEBUG 标记的值是 -D DEBUG。如果为 Debug 配置定义它，但不为 Release 配置定义（如图 9-5 所示），那么调试构建（在 Xcode 中构建并运行）就会通过 print 输出日志，但发布构建（归档并提交到 App Store）则不会。

图 9-5：定义 Swift 标记

原始调试另一种很有用的形式是有意中止应用，因为某些地方出现了严重的问题。请参见第 5 章关于 assert、precondition 与 fatalError 的介绍。precondition 与 fatalError 甚至可以用于发布构建。在默认情况下，assert 在发布构建中永远不会失败，因此在发布应用时将其留在代码中是不会产生什么问题的；当然，那时你应该胸有成竹地说，断言所检测的各种问题都已经在调试阶段解决掉了，不会再发生了。

纯粹主义者可能会嘲笑原始调试，不过我会经常用到它：它很简单，给出的信息量足够大，并且轻量级。有时它也是唯一的办法。与调试器不同，控制台日志可用于任何构建配置（调试或发布），无论应用运行在哪里都可以（模拟器或物理设备）。即便没法暂停，它也可以正常使用（比如，由于线程问题）。它甚至可用在物理设备上，比如，测试人员的设备上。对于测试者，查看控制台并将信息发给你可能有点麻烦，不过也是可以做到的：比如，测试者可以将设备连接到计算机上并在 Xcode 的设备窗口中查看日志。

## 9.5.2 Xcode 调试器

当在 Xcode 中构建和运行时，你可以在调试器中暂停并使用 Xcode 的调试功能。重点在于，

如果想要使用调试器，那么你应该使用调试构建配置来构建应用（这也是方案中 Run 动作的默认配置）。如果使用了发布构建配置来构建应用，那么调试器就没什么用了，因为编译器优化会破坏编译后的代码与代码行之间的对应关系。

### 1. 断点

在 Xcode 中调试和运行之间并没有什么大的差别；主要的不同在于断点是有效的，还是会被忽略。断点的效果可以在两个层级间切换：

*全局（激活与未激活）*

总的来说，断点分为激活与未激活两种状态。如果断点处于未激活状态，那就无法暂停任何断点。

*个体（启用与禁用）*

任何给定的断点要么是启用的，要么是未启用的。即便断点是激活的，但如果其被禁用了，那我们也无法暂停下来。可以通过禁用断点在未来需要的地方放置好断点，从而无须每次需要时再来暂停。

要创建断点（如图 9-6 所示），请在编辑器中选择你想要暂停的行，然后选择 Debug → Breakpoints → Add Breakpoint at Current Line（Command-\ 组合键）。该键盘快捷键会在为当前行添加断点与删除断点间切换。断点通过边列上的箭头表示。此外，单击边列也会添加断点；要想除断点，请将其拖曳出边列即可。

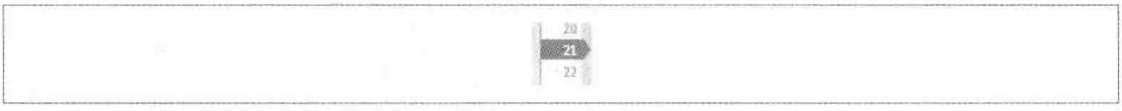


图 9-6：断点

要禁用当前行的断点，请单击边列上的断点以修改其启用状态。此外，还可以按住 Control 键并单击断点，然后从上下文菜单中选择 Disable Breakpoint。深色断点处于启用状态，而浅色断点则处于禁用状态（如图 9-7 所示）。

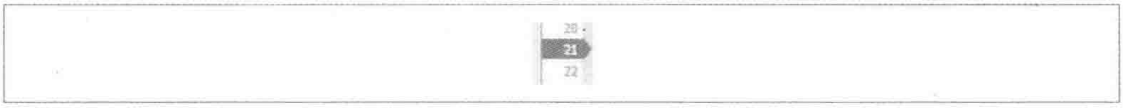


图 9-7：禁用的断点

要整体性地切换断点的激活状态，请单击调试窗格顶部的断点按钮，或选择 Debug → Activate/Deactivate Breakpoints（Command-Y 组合键）。整体的断点激活状态并不会影响每个断点的启用或禁用状态；如果断点是未激活的，那么它们会被忽略，这样断点处

就不会暂停了。如果断点处于激活状态，那么断点箭头就是蓝色的；如果处于未激活状态，那么箭头就是灰色的。

一旦在代码中设定了断点，就可以管理这些断点了。这正是断点导航器的目的所在。可以导航到断点处，通过单击导航器中的箭头来启用或禁用断点，或删除断点。

还可以编辑断点的行为。在边列或断点导航器的断点上按住 Control 键并单击，然后选择 Edit Breakpoint，或按住 Command 与 Option 键并单击断点。这是个非常强大的功能：可以在某种情况下或执行了某些次数后才在断点处暂停；可以在遇到断点时执行一个或多个动作，比如，发出调试命令、记录日志、播放声音、朗读文本，或运行一段脚本。

可以配置断点，在遇到断点并执行完其动作后再自动继续执行。这是比原始调试更为强大的功能：相比于插入 print 或 NSLog 调用（需要插入到代码中，并在发布应用时再将其删除），可以设定用于记录日志和继续执行的断点。根据定义，这种断点只在调试项目时才会起作用；当应用运行在用户设备上时，它不会向控制台输出任何信息，因为用户设备上是没有断点的。

可以在断点导航器中创建某些特殊类型的断点（单击导航器底部的“+”按钮，然后从弹出菜单中选择）或从 Debug → Breakpoints 层次菜单中选择：

*异常断点*

异常断点会让应用在异常抛出或捕获时暂停，而不考虑该异常是否会在后面导致应用崩溃。建议你创建异常断点以便在异常抛出时能够暂停，因为这样就可以在异常发生的时刻查看到调用堆栈和变量值了（而不必等到后面出现崩溃时再查看）；可以查看到在代码中的位置，并且可以检查变量值，这有助于你理解问题的原因所在。如果创建了这种异常断点，那么还建议你使用上下文菜单 Move Breakpoint To → User，这会持久化该断点并且让所有项目都可以使用它。

有时，Apple 的代码会有意抛出异常并将其捕获。这并不会导致应用崩溃，也不会出现什么问题；不过，如果创建了异常断点，那么应用就会暂停，这可能会对你造成困扰。

*符号断点*

符号断点会在调用某个方法或函数时让应用暂停，不管是什么对象调用的方法或消息发给哪个对象都是如此。方法可以通过两种方式来指定：

*使用 Objective-C 符号*

实例方法或类方法符号（- 或 +），后跟方括号，里面是类名与方法名。比如：

```
-[UIApplication beginReceivingRemoteControlEvents]
```

*根据方法名*

只有方法名。调试器会针对所有可能的类－方法对进行解析，就好像使用上面提到的 Objective-C 符号输入的一样。比如：

```
beginReceivingRemoteControlEvents
```

如果进入了不正确的方法名或类名，那么符号断点就不会做任何事情。一般来说，如果对了，自己应该是知道的，因为你会看到解析后的断点以层次化的结构列在了你的断点的下面。

## 2. 在断点处暂停

激活断点并运行应用，如果应用遇到了启用的断点（假设满足了断点的条件），那么应用就会暂停。在活动项目窗口中，编辑器会显示出包含了执行点的文件，这通常就是包含了断点的文件。执行点会显示为绿色的箭头；这是将要执行的代码行（如图 9-8 所示）。根据 Behaviors 首选项窗格中对 Running → Pauses 的设置，调试导航器与调试窗格会出现。

**图 9-8：在断点处暂停**

下面是应用在断点处暂停下来后你可能想要执行的动作：

*查看所在何处*

设置断点的一个常见原因就是确保执行路径通过了某一行。调试导航器的调用堆栈中所列出的函数如果带有 User 图标，其文本又是空的，那就表明这是自定义的方法；可以单击函数查看在方法中的哪一行暂停了（灰色文本的函数与方法是没有源代码的，因此单击这些方法是没什么意义的，除非你了解汇编语言）。还可以通过调试窗格顶部的跳转栏查看并导航调用堆栈。

*查看变量值*

在调试窗格中，当前作用域中的变量值（对应于调用堆栈中所选的变量）会显示在变量列表中。可以通过展开三角箭头查看到额外的对象特性，比如，集合元素、属性，甚至是某些私有信息。（局部变量值甚至会在暂停处显示出来，这些变量尚未初始化；这种值是没有意义的，请忽略。）

可以通过搜索框根据名字或值来过滤变量。如果格式化的摘要信息还不够，那么可以向对象变量发送 `description`（如果对象使用了 CustomDebugStringConvertible，

那就发送 debugDescription），并在控制台查看输出：从上下文菜单选择 Print Description of [Variable]，或选中变量并单击变量列表下方的 Info 按钮。

还可以以图形化方式查看变量值：选中某个变量，单击变量列表下的 Quick Look 按钮（一只眼睛的图标），或按下空格键。比如，对于 CGRect 来说，其图形化表示是个成比例的矩形。可以按照相同方式创建自定义类的实例；声明如下方法，并返回所允许的一个类型的实例（参见 Apple 的 *Quick Look for Custom Types in the Xcode Debugger*）：

```
@objc func debugQuickLookObject() -> AnyObject {
    // ... create and return your graphical object here ...
}
```

还可以直接在代码中查看变量值，只需查看其数据提示即可。要想查看数据提示，请将鼠标指针悬浮在代码中的变量名之上。数据提示非常类似于变量列表中所显示的值：有一个小三角，可以打开它查看更多信息，此外还有一个 Info 按钮，显示了这里与控制台上的值的描述，Quick Look 按钮则以图形化形式显示了一个值（如图 9-9 所示）。

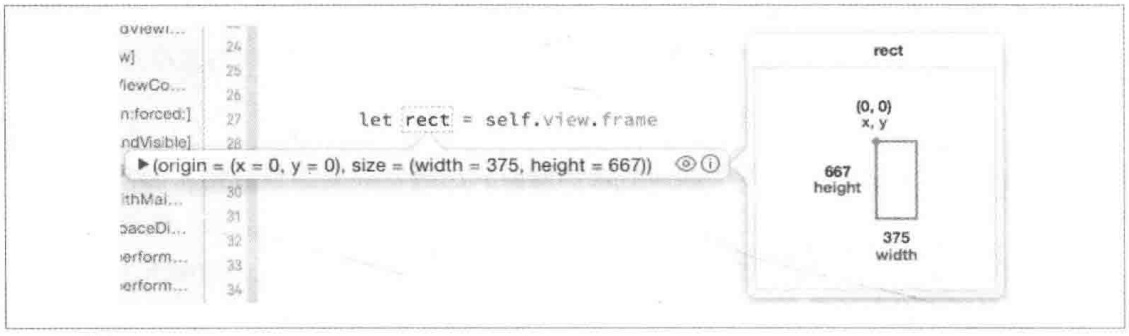


图 9-9：数据提示

*查看视图层次*

可以在调试器暂停时查看视图层次。单击调试窗格顶部栏中的 Debug View Hierarchy 按钮，或选择 Debug → View Debugging → Capture View Hierarchy。视图会以大纲形式列在调试导航器中。编辑器会显示出你的视图；这是个可旋转的三维投影。对象查看器与尺寸查看器会显示出关于当前所选视图的信息。

*管理表达式*

表达式是添加到变量列表中的代码，每次暂停时都会进行计算求值。可以从变量列表上下文菜单中选择 Add Expression 来添加表达式。表达式会在代码的当前上下文中进行求值，因此请小心其副作用。

*与调试器通信*

可以通过控制台直接与调试器通信。Xcode 的调试器界面是真正的调试器 LLDB

（*http://lldb.llvm.org*）的一个前端；通过直接与 LLDB 通信，可以完成 Xcode 调试器界面所能做的任何事情，甚至还可以做到更多。常见的命令有：

`fr v`（`frame variable` *的简称*）

输出作用域中的所有局部变量，类似于在变量列表中显示。此外，其后还可以跟着你想要查看的变量名。

`po`（*表示“print object”*）

后跟作用域中对象变量的名字，类似于 Print Description: 根据 `description` 或 `debugDescription` 显示对象变量值一样。

`p`（*或* `expression`、`expr`、`e`）

计算当前上下文中当前语言的任何表达式。

*操控断点*

可以在应用运行时自由创建、删除、编辑、启用、禁用和管理断点，这是非常有用的，因为下一次暂停的位置可能取决于现在暂停的位置。事实上，这是断点相比于原始调试的一个主要优势。要修改原始调试，需要停止应用、编辑、重新构建，然后再次运行应用。不过，通过操纵断点，无须停止应用；甚至都不需要暂停！如果某个操作出错了（但不会导致应用崩溃），那么它可以实时地重复多次；这样，只须添加一个断点并重试即可。比如，如果轻拍按钮会生成错误的结果，那么你就可以向动作处理器添加一个断点，并再次轻拍按钮；执行的代码都是一样的，但这次可以看到问题出在了哪里。

*步进或继续*

要让暂停的应用继续执行，可以继续运行，直到遇到了下一个断点（Debug → Continue），或步进并再次暂停。此外，可以选中一行，然后选择 Debug → Continue to Current Line（或从上下文菜单中选择 Continue to Here），这会在所选行上设置一个断点，继续执行并删除该断点。步进命令如下所示（位于 Debug 菜单中）：

*Step Over*

在下一行暂停。

*Step Into*

如果当前行调用了函数，那么就会在该函数中暂停；否则，在下一行暂停。

*Step Out*

从当前函数返回处暂停。

可以通过调试窗格顶部的快捷按钮使用这些命令。即便调试窗格被收起，在应用运行时，包含按钮的工具栏也会呈现出来。

*重新开始，或终止*

要终止运行着的应用，请单击工具栏中的 Stop（Product → Stop，Command-Period 组合键）。单击模拟器或设备中的 Home 按钮（Hardware → Home）并不会停止运行着的应用，这是因为在 iOS 4 及之后的系统中都是多任务运行了。要终止运行着的应用，但在不重新构建的情况下还要重新启动它，请按住 Control 键并单击工具栏中的 Run（Product → Perform Action → Run Without Building，Command-Control-R 组合键）。

可以在应用运行或暂停时修改代码，不过这些修改并不会对运行着的应用起作用；有一些编程环境会让这一美梦成真，但 Xcode 不行。你需要终止应用，并按照正常方式运行它（包括构建）才能看到修改效果。

## 9.6 测试

测试代码并不属于应用目标的一部分，其目的在于确保应用运行与期望保持一致。测试可以分为如下两类：

*单元测试*

单元测试会在内部执行应用目标，这是从代码的角度来看待的。比如，单元测试可能会调用应用目标代码的某个方法，传给其若干参数，然后看看是否每次都能返回期望的结果，不仅仅在正常情况下，还要看不正确或极端输入情况下是否也能如此。

*界面（UI）测试*

界面测试（Xcode 7 新增的功能）会在外部执行应用，这是从用户的角度来看待的。这种测试会让应用通过一系列用例场景，用手指轻拍界面上的按钮，观察结果并确保界面行为与期望保持一致。

在理想情况下，测试应该伴随着应用开发的过程来编写和运行。在编写实际代码前编写单元测试会更好一些，可作为实现算法的一种方式。在确定代码通过了测试后，可以继续运行这些测试来检测在开发过程中是否引入了 Bug。

测试会被打包到项目的一个单独的目标当中（参见第 6 章）。借助应用模板，可以在创建项目时添加测试目标：在第 2 个对话框（“Choose options”，即命名项目时）中，可以勾选 Include Unit Tests、Include UI Tests，或二者都勾选上。此外，可以随时轻松创建新的测试目标：创建一个新的目标，并指定 iOS → Test → iOS Unit TestingBundle 或 iOS UI Testing Bundle 即可。需要显式运行测试才行。可以在测试导航器（Command-5 组合键）与测试类文件中轻松管理并运行测试。

测试类是 XCTestCase（它本身又是 XCTest 的子类）的子类。测试方法是测试类的实例

方法，它没有返回值并且不接收参数，并且名字以 test 开头。测试目标依赖于应用目标，这意味着在编译和构建测试类之前，我们需要先编译和构建应用目标。运行测试也会运行应用，测试目标的产物是个包，它会在应用启动时加载到应用中。

一个测试方法需要调用一个或多个测试断言；在 Swift 中，这些断言都是全局函数，名字以 XCTAssert 开头。请参阅 Apple 文档 *Testing With Xcode* 的“Writing Test Classes and Methods”章节了解完整的函数列表，具体位于“Assertions Listed by Category”一节下。与相应的 Objective-C 宏不同，Swift 测试断言函数并不接收格式化字符串（NSLog 所采取的方式）；每个函数都接收一个简单的消息字符串。在 Swift 中，标记为“针对标量”的测试断言函数并非真的如此，因为 Swift 中是不存在标量的（相对于对象来说）：它们会应用于使用了 Equatable 或 Comparable 的类型。

测试类还可以包含一些辅助方法，这些方法会被测试方法所调用。此外，可以重写从 XCTestCase 继承下来的 4 个特殊方法：

setUp *类方法*
: 只会调用一次，并且在类中所有测试方法执行之前调用。

setUp *实例方法*
: 在每个测试方法调用前调用。

tearDown *实例方法*
: 在每个测试方法调用后调用。

tearDown *类方法*
: 只会调用一次，并且在类中所有测试方法执行之后调用。

测试目标也是个目标，其产出是个包，其构建阶段类似于应用目标。这意味着，如测试数据等资源可以放到包中。可以通过 setUp 加载这些资源；通过测试类获得对包的引用：作为 NSBundle(forClass: self.dynamicType)。

测试目标也是个模块，就像应用目标一样。为了能够使用应用目标，测试目标需要将应用目标以模块的形式导入进来。为了克服私有性限制，import 语句前面应该加上 @testable 特性；该特性是 Xcode 7 中新引入的，它会将应用目标中的 internal（显式或隐式）临时改为 public。

下面编写并运行一个单元测试方法，这是使用的是 Empty Window 项目。为 ViewController 类添加一个没有实际意义的实例方法 dogMyCats：

```
func dogMyCats(s:String) -> String {
    return ""
}
```

方法 dogMyCats 接收一个字符串并返回字符串 "dogs"。但此时却并非如此；它返回一个空字符串。这是个 Bug。现在编写一个测试方法以找出这个 Bug。

在 Empty Window 项目中，选择 File → New → Target 并指定 iOS → Test → iOS Unit Testing Bundle。将产品命名为 *EmptyWindowTests*；待测试的目标就是应用目标。单击 Finish。在项目导航器中会新建一个分组 EmptyWindowTests，它包含一个测试文件 *EmptyWindowTests.swift*。该文件中有一个测试类 EmptyWindowTests，其中包含两个测试方法的桩：testExample 与 testPerformanceExample；将这两个方法注释掉。我们打算使用一个会调用 dogMyCats 的测试方法将其替换，该方法会对结果作出断言：

1. 在 *EmptyWindowTests.swift* 顶部导入 XCTest 的地方，也需要导入应用目标：

```
@testable import Empty_Window
```

2. 请在 EmptyWindowTests 类的声明中添加一个实例属性，用于存储 ViewController 实例：

```
var viewController = ViewController()
```

3. 编写测试方法。测试方法的名字要以 test 开头！我们将其命名为 testDogMyCats。它可以通过 self.viewController 访问 ViewController 实例：

```
func testDogMyCats() {
    let input = "cats"
    let output = "dogs"
    XCTAssertEqual(output,
        self.viewController.dogMyCats(input),
        "Failed to produce \(output) from \(input)")
}
```

如果向老项目中添加单元测试，你可能需要做一些额外的配置。为了确保可以通过 @testable 特性导入应用目标，请在构建设置中找到 Enable Testability 并确保在调试配置中将其设为 Yes。此外，编辑方案，确保构建动作在测试动作发生时只会构建测试目标。

现在可以运行测试了！有多种方式可以做到这一点。切换到测试导航器，你会看到里面列出了测试目标、测试类与测试方法。将鼠标指针悬停在任意名字上，这时会在右侧弹出一个按钮。通过单击恰当的按钮，可以运行每个测试类中的所有测试、EmptyWindowTests 类中的所有测试，还可以单独运行 testDogMyCats 测试。不过请等一下，还有呢！回到 *EmptyWindowTests.swift*，在类声明与测试方法名左侧的边列中有一个菱形指示器；还可以单击这个指示器运行测试，既可以运行这个类中的所有测试，也可以运行单个测试。要运行所有测试，还可以选择 Product → Test。

下面运行 testDogMyCats。应用目标已经编译并构建完毕；测试目标亦如此（如果其中有任意一步失败了，测试就将无法进行，我们需要回过头来解决编译错误或构建错误）。应用会在模拟器中启动，测试也将运行。

测试失败了（我们其实知道会失败）！错误说明会出现在代码中失败断言的旁边，以及问题导航器与日志导航器中。此外，测试导航器中 testDogMyCats 旁边、问题导航器、日志导航器与 *EmptyWindowTests.swift* 类声明旁边与 testDogMyCats 的第一行还会出现红色的 X 标记。

现在来修复代码。在 *ViewController.swift* 中，将 dogMyCats 的返回值由空字符串修改为 "dogs"。再次运行测试。通过！

最近运行的测试会列在报告导航器中。如果选择了其中一个，编辑器就会显示出两个窗格。测试窗格会以简单的大纲形式列出成功与失败的测试，包括断言失败消息的文本。日志窗格则会列出更为详尽的信息；展开后会看到运行过的测试的完整控制台输出，包括测试代码中所输出（print）的原始调试消息。

当测试失败时，你可能希望在断言失败之处暂停。要做到这一点，请在断点导航器中单击底部的“+”按钮并选择 Add Test Failure Breakpoint。这类似于异常断点，它会在报告失败前，在测试方法中断言失败那一行处暂停。接下来可以切换到被测试的方法，对其进行调试，查看其变量，从而找出失败的原因所在。

有一个很有用的特性可以帮助你在方法与调用该方法的测试间切换：当选中方法中的某些代码时，跳转栏中的 Related 菜单就会包含进测试调用者。对于辅助窗格中的 Tracking 菜单来说亦如此。

在该示例中，创建了一个新的 ViewController 实例来初始化 EmptyWindowTests 的 self.viewController。不过，如果测试需要引用现有的 ViewController 实例该怎么办呢？这与 iOS 编程中频繁出现的实例引用是一个问题。测试代码运行在一个包中，它会被注入运行着的应用中。这意味着它能看到应用的全局信息，如 UIApplication.sharedApplication()。可以通过它得到所需的引用：

```
if let viewController =
    (UIApplication.sharedApplication().delegate as? AppDelegate)?
        .window?.rootViewController as? ViewController {
            // ...
}
```

将测试方法组织到测试目标（套件）与测试类中在很大程度上是为了方便；这会对测试导航器的布局以及哪些测试会一起运行产生影响，同时每个测试类都有自己的属性，自己的 setUp 方法等。要创建新的测试目标或测试类，请单击测试导航器底部的“+”按钮。

除了刚才介绍的简单的单元测试类型，还有另外两种形式的单元测试：

*异步测试*

可以在一个耗时操作执行完毕后回调测试方法。在测试方法中，可以通过调用 `expectationWithDescription` 来创建一个 `XCTestExpectation` 对象；然后初始化一个接收完成处理器的操作，调用 `waitForExpectationsWithTimeout:handler:`。这样会出现下面两种情况之一：

*操作完成*

完成处理器会被调用。在完成处理器中，可以执行与操作结果相关的任何断言，然后调用 XCTestExpectation 对象的 `fulfill`。这会导致超时处理器得到调用。

*操作超时*

超时处理器会被调用。这样，超时处理器都会得到调用，可以执行必要的清理工作。

*性能测试*

可以测试成功操作的速度。在测试方法中调用 `measureBlock`，在块中做一些事情（可以执行很多次，从而得到合理的时间度量样本）。如果块中涉及度量不想包含的创建与清理工作，那就可以调用 `measureMetrics:automaticallyStartMeasuring:forBlock:`，然后将块的核心包装到 `startMeasuring` 与 `stopMeasuring` 中。

性能测试会执行块多次，记录每次运行时间。首次执行性能测试时会失败，不过却建立了基准度量。在随后的运行中，如果标准偏差距离基准太远，或平均时间变得过长，测试都会失败。

现在来看看界面测试。假设 Empty Window 界面上依旧有一个按钮（第 7 章），它有一个动作方法挂接到了 ViewController 方法上，会弹出一个警告框。我们会编写一个测试，轻拍按钮并确保会弹出警告框。向项目添加一个 iOS UI Testing Bundle，命名为 *EmptyWindowUITests*。

界面测试代码是基于可访问性的，该特性可以描述屏幕的界面，然后以编程的方式操纵它。它涉及 3 个类：XCUIElement、XCUIApplication（XCUIElement 的子类）以及 XCUIElementQuery。在很大程度上，你不必了解这些类，因为可访问性动作是可记录的。这意味着可以通过执行构成测试的实际动作来生成代码。下面就来试一下：

1. 在 `testExample` 桩方法中，创建一个新的空行，并将插入点置于其中。
2. 选择 Editor → Start Recording UI Test（此外，还可以使用项目窗口底部调试栏上的 Record 按钮）。应用会在模拟器中启动。
3. 轻拍界面上的按钮。当警告框出现时，轻拍 OK 将其关闭。

4. 回到 Xcode，选择 Editor → Stop Recording UI Test。选择 Product → Stop 会停止在模拟器中运行。

这会生成如下代码（假设界面按钮上的文字是“Hello”）：

```
let app = XCUIApplication()
app.buttons["Hello"].tap()
app.alerts["Howdy!"].collectionViews.buttons["OK"].tap()
```

显然，app 对象是个 XCUIApplication 实例。buttons 与 alerts 等属性返回 XCUIElementQuery 对象。对该对象进行下标计算会返回一个 XCUIElement，接下来可以向其发送 tap 等动作方法。

现在运行测试，单击 testExample 声明边栏上的菱形图标。应用会在模拟器中启动，手指会执行我们之前所执行的相同动作，轻拍界面上的第 1 个按钮，当警告框出现后，轻拍 OK 按钮，警告框就会消失。测试结束，应用在模拟器中停止运行。测试通过！

不过，重要的事情在于如果界面停止响应，那么测试就不会通过。比如，在 *Main.storyboard* 中，选择按钮，在属性查看器下方的 Control 中取消勾选 Enabled。按钮依旧在那儿，不过却无法轻拍它；我们破坏了界面。运行测试。如期望一般，测试失败了，报告导航器会给出原因：当进入轻拍“OK”按钮这一步时，我们首先要进行查找“OK”按钮的操作，尝试两次后失败了，因为根本就没有警告框。报告还会截屏，这样我们就可以检测到测试过程中界面的状态了。将鼠标悬浮在“OK”按钮上，一只眼睛的图标会出现。单击它，截图会展示出该时刻的界面，清晰地显示出禁用的界面按钮（没有警告框）。

再次启用按钮来修复这个 Bug。如果现在选择 Product → Test，那么测试套件中的所有测试都会运行，包括单元测试与界面测试，它们都会通过。这个应用很简单，不过却是可用的！

如前所述，界面测试依赖于可访问性。标准的界面对象是可访问的，不过你所创建的其他界面可能未必如此。在 nib 编辑器中选择一个界面，在身份查看器中查看其可访问性。在模拟器中运行应用，选择 Xcode → Open Developer Tool → Accessibility Inspector 来实时查看鼠标指针下方内容的可访问性。要了解如何向界面对象添加有用的可访问性特性，请参考 Apple 的 Accessibility Programming Guide for iOS。

## 9.7 清理

有时，在重复的测试与调试期间，最好在进行不同类型的构建前（从 Debug 切换到 Release，或从模拟器切换到设备中运行）清理目标。这意味着将会删除现有的构建并清除缓存，这样所有代码才会被编译，下一次构建才会从头开始构建应用。

与字面上的意思一样，清理会清除不需要的东西。比如，假设应用中包含了某个资源，但未来不再需要。可以在 Copy Bundle Resources 构建阶段将其删除（或从项目中删除），不过这并不会从构建好的应用中删除。这种残留资源会导致一些莫名其妙的问题。错误的 nib 版本可能会出现在界面中；编辑过的代码行为可能与编辑前一样。清理则会删除构建好的应用，很快就能解决问题。

我将清理划分为几个层次：

*浅层清理*

选择 Product → Clean，它会删除构建好的应用以及构建目录中的一些中间信息。

*深层清理*

按住 Option 键并选择 Product → Clean Build Folder，它会删除整个构建目录。

*完全清理*

关闭项目。打开项目窗口（Window → Projects）。找到左侧列出的项目并单击。在右侧选择 Delete。这会删除用户目录下 *Library/Developer/Xcode/DerivedData* 目录中的全部目录。

*彻底清理*

关闭 Xcode。打开用户目录下的 */Developer/Xcode/DerivedData*，将其内容全部移至废纸篓。这是对最近打开的所有项目的完全清理，再加上模块缓存。删除模块缓存会重置 Swift 本身，这可能会导致一些编辑、代码完成或语法着色等出现问题。

除了清理项目，你还应该将模拟器中的应用删除。原因与清理项目一样：当应用构建完毕并被复制到模拟器中时，构建好的应用中的已有资源可能不会被删除（为了节省时间），这可能会导致应用表现出不正确的行为。要在运行模拟器时进行清理，请选择 iOS Simulator → Reset Content and Settings。

## 9.8 在设备中运行

你迟早会将应用的运行、测试与调试从模拟器转换到实际设备上。模拟器很好，但它只是模拟而已；模拟器与实际设备间还是有很多差别的。模拟器实际上就是你的计算机，它速度很快并且拥有很多内存，这样内存管理与速度上的问题直到在设备上运行才会发现。与模拟器之间的用户交互只能限定在鼠标上：可以单击、拖曳，可以按住 Option 键来模拟用户的两指，但更多的手势只能在实际设备上使用。很多 iOS 功能（如加速计和访问音乐库等）是无法在模拟器上使用的，如果应用使用了这些功能，那么只能在设备上进行测试。

开发应用而不在设备上测试是绝对行不通的。应用只有在设备上运行，你才能知道应用的样子和行为。向 App Store 提交并未在设备上运行过的应用也是自讨苦吃。

在设备上运行应用是一件很复杂的事情。你需要在构建时对应用签名。没有针对设备进行恰当签名的应用是无法在该设备上运行的（假设没有越狱）。对应用签名需要两个东西：

*一个身份*

身份代表 Apple 允许团队在特定的计算机上开发的应用运行在设备上。它包含两部分：

*私钥*

私钥存储在计算机的钥匙链中。因此，它能识别出特定的计算机，团队可以在该计算机上开发应用，然后在设备上运行。

*证书*

证书是 Apple 颁发的一个虚拟许可。它包含了与私钥匹配的公钥（因为在申请证书时你向 Apple 提供了公钥）。借助证书的副本，拥有私钥的任何计算机实际上都可以在相应的团队名称下开发应用并在设备上运行。

*配置文件*

配置文件是 Apple 提供的虚拟许可，它包含了如下 4 项：

- 一个身份。
- 一个应用，通过其包 id 进行识别。
- 符合条件的设备列表，通过其 UDID（唯一识别标识符）进行识别。
- 一个权利列表。权利指的是并非每个应用都需要的一个特权，比如，与 iCloud 通信的能力。只有编写需要权利的应用时才需要考虑这一点。

因此，在构建时，配置文件足以完成对应用的签名。它表示在这个 Mac 上所构建的应用是可以运行在这些设备上的。

有两种类型的身份，因此也有两种类型的证书，两种类型的配置文件：开发与发布（发布证书也叫作产品证书）。这里只关注开发身份、证书与配置；本章后面将会介绍发布方面的内容。

Apple 公司是所有信息的最终保留者：证书、配置文件，以及注册的应用与设备等。当需要验证或获得这个信息的副本时，你与 Apple 公司之间的通信是通过如下两种方式达成的：

*会员中心*

一些网页，地址是 *https://developer.apple.com/membercenter*。如果你是开发者计划成员，那么可以通过单击 Certificates, Identifiers, & Profiles 来访问当前会员类型与角色所能访问的所有特性与信息（这部分内容的正式名称叫作 Portal）。

*Xcode*

除了获取发布配置文件，可以通过 Xcode 完成会员中心所能完成的一切事项。如果一切顺利，那么使用 Xcode 会更加简单！如果有问题，那么你可以访问会员中心寻求解答。

## 9.8.1 在没有开发者计划成员资格的情况下运行

过去，拥有 iOS 开发者计划成员资格是必要的前提，这意味着你需要支付年费才能在自己的设备上测试应用。不过在 Xcode 7 中，你可以在没有开发者计划成员资格的情况下配置应用在你的设备上运行。你所需要的只不过是一个 Apple ID 而已，而你肯定会有的。

因此，在介绍设备上运行应用的详情前，我先来谈谈在没有做任何准备的情况下如何在设备上运行你的应用：没有开发者计划成员资格、没有在 Xcode 中输入任何账户信息，之前也从未在设备上运行过应用：

1. 编辑应用目标，切换至 General 窗格，查看 Team 弹出菜单。假设现在还没有团队，你首先要做的事情就是创建一个。从 Team 弹出菜单中选择 Add an Account；这会打开 Xcode 账户首选项窗格，类似于按下“+”按钮并选择 Add Apple ID。输入你的 Apple ID 与密码，这会创建一个免费账户。关闭账户首选项窗格。回到 Team 弹出菜单，选择刚才创建的团队。
2. 在 Team 弹出菜单下，你会看到一个警告，告诉你还没有代码签名身份。单击 Fix Issue。Xcode 会与会员中心进行通信。这个问题会得到部分解决，不过你会看到一个对话框：“Unable to create a provisioning profile because your team has no devices registered in the Member Center...”。单击 Done。
3. 将设备关联到计算机上。等待符号文件进行处理（可以追踪这一过程，方式是选择 Window → Devices 并选中设备，这时可以喝杯咖啡，过会儿再过来了）。
4. 现在，设备可用于开发了。在方案弹出菜单中将设备作为目标，运行项目！你会看到一个对话框：“Failed to code sign...”。单击 Fix Issue。Xcode 会再次与会员中心进行通信，接下来应用就会构建并在设备上运行了。

在后台，Xcode 执行了如下几个必要的步骤：

- 它在钥匙链中创建了一个开发者身份（可以通过钥匙链访问应用看到）。

- 将你的设备注册到了会员中心（由于没有开发者计划成员资格，你无法直接登录到会员中心看到这一点）。
- 创建并下载了一个团队配置文件，对上述内容进行了整合，也就是说，你的应用可以从这台计算机在该设备上运行了。

## 9.8.2 获取开发者计划成员资格

你早晚会需要一个开发者计划成员资格。进入 iOS 开发者计划页面（*http://developer.apple.com/programs/ios*）完成注册流程。一开始，个人计划就足够了。组织计划不会增加成本，不过可以添加其他开发者，并赋予不同角色。如果只是向其他用户分发应用来进行测试，那就不需要组织计划了。

iOS 开发者计划成员涉及如下两点：

*一个 Apple ID*

这是个用于在 Apple 网站标识你自己的用户 ID（还有相应的密码）。你可以通过自己的开发者计划 Apple ID 做所有事情。除了准备应用以在设备上运行，你还可以通过该 Apple ID 在 Apple 开发论坛上发帖、下载 Xcode Beta 版等。

*一个团队名称*

同一个 Apple ID 可以隶属于多个团队。在每个团队中，你都会有一个角色，指明了你的权利是什么。如果你是团队的领导（或是团队中的唯一成员），那么你的角色就是 Agent，这意味着你可以做一切事情：可以开发应用、在设备上运行、向 App Store 提交应用，并获取付费应用的收益所得。

创建了开发者计划 Apple ID 后，你应该在 Xcode 的账户首选项窗格中输入它。单击左下角的“+”按钮并选择 Add Apple ID，输入 Apple ID 与密码。从现在开始，Xcode 可以通过与这个 Apple ID 关联的团队名称识别出你；无须再向 Xcode 提供密码了。

## 9.8.3 获取证书

创建身份并获取证书（见图 9-10）只须做一次即可（也许一年最多一次；如果每年的开发者计划成员过期并进行更新，你可能还要做一次）。还记得吧，证书依赖于私钥公钥对。私钥位于钥匙链中；公钥则会发送给 Apple，它会被构建到证书中。你发给 Apple 公钥是通过对证书的请求进行的。理想情况下，你可以通过 Xcode 轻松做到这一点：

1. 打开 Xcode 的账户首选项窗格。
2. 如果没有输入过开发者 Apple ID 与密码，请现在输入。

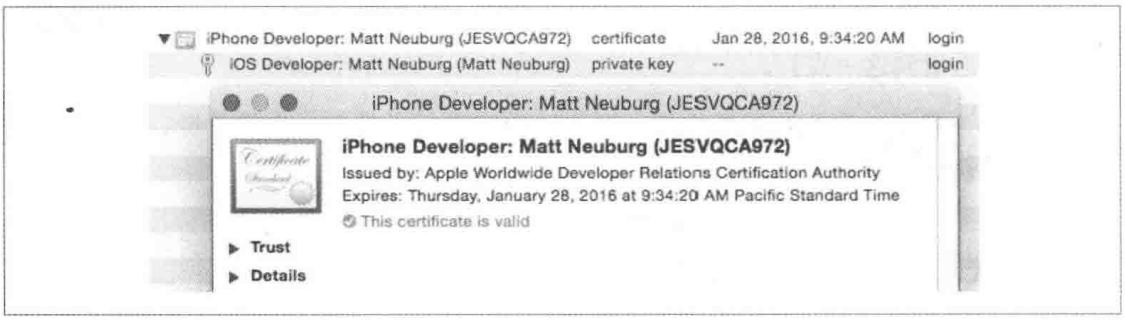


图 9-10：Keychain Access 中显示的有效的开发证书

3. 在左侧选择 Apple ID，在右侧选择团队，单击 View Details。
4. 如果有证书，但被会员中心取消，但证书依旧有效，那就会看到一个请求并下载证书的对话框。单击 Request。否则，单击 iOS Development 右侧的 Create 按钮。

接下来一切都会自动发生：私钥公钥对会在钥匙链中生成，证书请求会发送给会员中心、生成并下载，然后存储到钥匙链中，并列在 Xcode 账户首选项窗格 View Details 对话框的 Signing Identities 下面。此外，通用的团队开发配置文件也可能会生成，如图 9-11 所示。这样就具备了在设备上运行应用的全部。

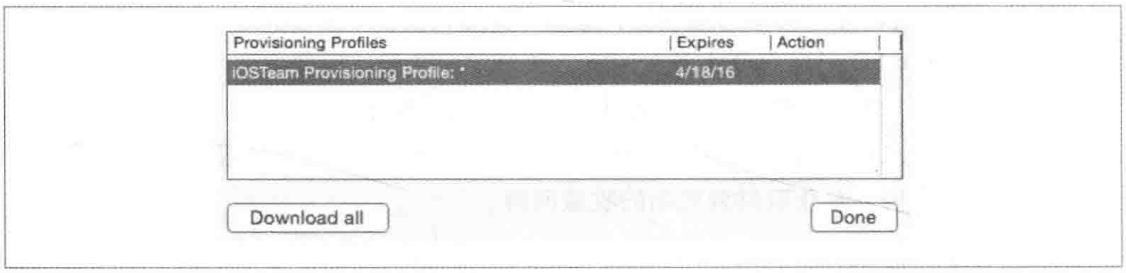


图 9-11：通用开发配置

了解生成私钥公钥对与证书请求的手工处理过程也是很有益的。过程发起后，处理命令也可以在会员中心找到（进入 Certificates 页面，单击右上角的“+”按钮）：

1. 启动 Keychain Access 并选择 Keychain Access → Certificate Assistant → Request a Certificate from a Certificate Authority。将你的名字与 Email 地址作为标识符，生成一个 2048 位的 RSA 证书请求文件，并保存到磁盘中。私钥存储在钥匙链中；包含公钥的证书请求会被临时保存到计算机中（比如，可以保存到桌面上）。
2. 在会员中心，你会看到一个上传所保存的证书请求文件的界面。上传，接下来会生成证书；单击会员中心的列表显示出 Download 按钮，然后单击 Download。
3. 找到并双击刚才下载的文件；Keychain Access 会自动导入证书并将其存储到钥匙链中，现在 Xcode 也会看到它。

你不需要保存证书请求文件与下载的证书文件；钥匙链现在包含了所有需要的凭证。如果一切正常，你可以在钥匙链中看到证书，查阅其详细信息，你会发现它是有效的，并且链接到了私钥（如图 9-10 所示）。此外，你可以确认 Xcode 现在已经知道了该证书：在账户首选项窗格中，单击左侧的 Apple ID 与右侧的团队名称，然后单击 View Details；这时会弹出一个对话框，你会看到顶部列出了一个 iOS 开发签名身份，其状态是有效的。

如果这是你第一次从会员中心获取证书，那么你还需要另一个证书：WWDR Intermediate Certificate。该证书用于证明由 WWDR（Apple Worldwide Developer Relations Certification Authority）所颁发的证书是受信的（你不能自己创建）。Xcode 应该会自动在钥匙链中安装该证书；如果没有，那么可以在添加证书的过程中，手工单击会员中心页面底部的链接获取其一份副本。

## 9.8.4 获取开发配置文件

如前所述，配置文件统一了身份、设备与应用包。如果一切顺利，那么你可以通过 Xcode 一步获取到开发配置文件，这是最简单的情形。如果应用不需要特殊的功能，那么与团队关联的单个开发配置就可以满足所有应用的需求，因此这一步只需执行一次即可。

通过 9.8.3 节的操作，你已经拥有了一个开发身份，可能还获取到了一个统一的团队开发配置文件！如果没有，那么最简单的办法就是打开 Xcode 并将设备连接到计算机上，经过一小段时间后（比如，告诉设备信任计算机等），将设备作为目标，并在其上运行项目。Xcode 会帮你在会员中心注册设备，并且为该设备创建和下载统一的团队配置文件。

要确认设备已经添加到了会员中心，请打开浏览器访问会员中心，并单击 Devices。要确认已经拥有了统一的团队开发配置文件，请单击账户首选项窗格的 View Details（选择恰当的团队）。证书与文件都列在那儿。除了标题“iOS Team Provisioning Profile”，统一的团队开发文件有一个与之关联的普通的应用包 id，通过一个星号标识（如图 9-11 所示）。

可以通过统一开发文件针对测试目的在目标设备上运行任何应用，只要应用不需要特殊功能即可（比如，使用 iCloud）。

还可以手工在会员中心注册设备。在 Devices 下，单击“+”按钮并输入设备名与 UDID。可以从 Xcode 的设备窗口中复制设备的 UDID。此外，可以提交 Tab 分隔的 UDID 文本文件与名字。

根据需要，可以在会员中心为特定应用创建配置文件：

1. 确保应用已经通过 Identifiers → App ID 在会员中心注册了。如果尚未注册，那就添加一个，如下所示：单击“+”按钮并输入该应用的名字。会员中心会为包标识符添加一个无意义的字母与数字前缀，不用管它们；使用 Team ID。在 Explicit App ID 下输入 Xcode 中所显示的包标识符，在编辑应用目标时，这个包标识符位于 Xcode General 窗格的 Bundle Identifier 域中。
2. 在 Provisioning Profiles 下单击“+”按钮。申请一个 iOS 应用开发配置文件。在下一个界面中选择 App ID。接下来检查开发证书。然后选择想要运行的设备。接下来为该配置文件起个名字，单击 Generate。最后单击 Download 按钮。
3. 找到下载的配置文件，双击它以在 Xcode 中将其打开。然后就可以将下载的配置文件删除了，因为 Xcode 已经拥有了一个副本。

## 9.8.5 运行应用

拥有了适用于应用与设备的开发配置文件后（对于统一团队配置文件来说，就是所有应用与所有注册的设备），请连接设备，在方案弹出菜单中将其作为目标，然后构建并运行应用。如果要求提供对钥匙链的访问，请授权。如果必要，Xcode 会将相关的配置文件安装到设备上。

应用构建，然后加载到设备上，最后在设备上运行。只要是在 Xcode 中启动应用，那么一切就像是在模拟器中运行一样；你可以运行、调试，运行着的应用可以与 Xcode 通信，这样就可以停在断点处，查看控制台中的信息等。区别在于你是通过设备（连接到了电脑上）而非模拟器与应用进行交互。

通过 Xcode 在设备上运行应用还可以用于将当前版本的应用复制到设备上。接下来可以停止应用的运行（在 Xcode 中操作），断开设备与电脑的连接，在设备上启动应用，然后使用。这是一种非常棒的测试方式。现在不是在调试，因此无法从 Xcode 获得反馈，不过稍后可以获得写到内部控制台的信息。

## 9.8.6 配置文件与设备管理

可以通过 Xcode 的账户首选项窗格查看身份与配置文件。

账户首选项窗格的一个重要特性是它可以导出账户信息。如果想在不同的计算机上开发，那么你就需要这个特性。选中一个 Apple ID，然后选择窗格底部齿轮菜单中的 Export Developer Accounts。你需要提供一个文件名以及保存的位置，还需要一个密码；这个密码只与该文件有关系，并且只在另外一台计算机上打开该文件时才需要。将之前导出

的文件复制到另外一台计算机上，然后运行 Xcode 并双击导出的文件；Xcode 会要求你提供密码。输入完密码后，整个团队、身份、证书与配置文件就会神奇地出现在这个 Xcode 中，甚至包括钥匙链中的那些内容。

此外，你可能只想导出身份，而不导出配置文件。这可以通过账户首选项窗格 View Details 对话框中的上下文菜单实现。

如果账户首选项窗格 View Details 对话框中列出的配置文件与会员中心的不同步，那么请单击左下角的 Download All 按钮。如果这么做不起作用，那么请关闭 Xcode，然后在 Finder 中打开用户目录下的 *Library/MobileDevice/Provisioning Profiles* 目录，删除里面的全部内容，重新启动 Xcode。在账户窗格下，配置文件会消失不见，现在再单击 Download All 按钮。Xcode 会下载配置文件的新副本，这样配置文件就会与会员中心同步了。

当设备连接到了计算机上时，它会出现在 Xcode 的设备窗口中。单击其名字可以查看设备的信息。你会看到（也可以复制）设备的 UDID，以及（也可以删除）使用 Xcode 进行开发时在设备上安装的应用。可以实时查看设备的控制台日志（这个界面有点隐蔽：请单击设备窗口主窗格左下角的向上小箭头）。借助齿轮菜单，你可以查看到安装到设备上的配置文件。可以查看到设备上出现的崩溃日志报告；还可以对设备界面进行截屏；在将应用提交到 App Store 时，你需要这么做。

# 9.9 分析

Xcode 提供了用于以图形化和数字化形式探索应用内部行为的工具，你应该了解这些工具的使用方式。可以通过调试导航器中的仪表盘在应用运行时监控其关键指标，如 CPU 与内存使用。Instruments 则是个复杂且强大的辅助应用，它会收集有助于追踪问题的分析数据，并提供你所需要的数字化信息来改进应用的性能与响应性。在应用开发趋于结束之际，你应该花些时间了解一下 Instruments 的用法（众所周知，过早的优化是万恶之源）。

## 9.9.1 仪表盘

调试导航器中的仪表盘会在应用构建和运行时起作用。单击某一项可以在编辑器中查看到进一步的细节信息。仪表盘并未提供非常详尽的信息，不过它却非常轻量级且总是处于激活状态，因此可以通过它在任何时候了解到应用的总体运行情况。特别地，如果出现了问题，比如，长时间的高 CPU 使用率或内存使用不断飙升，那就可以在仪表盘中定位到，然后通过 Instruments 找出问题所在。

有 4 种基本的仪表盘：CPU、内存、磁盘与网络。根据环境的不同，你可能还会看到其他仪表盘。比如，在 Xcode 7 中，当在设备上运行时，电量仪表盘会出现；对于某些设备来说还可能会出现 GPU 仪表盘。如果应用使用了 iCloud，那还会看到 iCloud 仪表盘。

在图 9-12 中，我频繁使用了应用一段时间，不断重复地执行用户可能会操作的最消耗内存的动作。这些动作会导致内存使用量攀升，不过应用的内存使用量最后会趋于平稳，因此我认为应用在内存使用量上是没问题的。

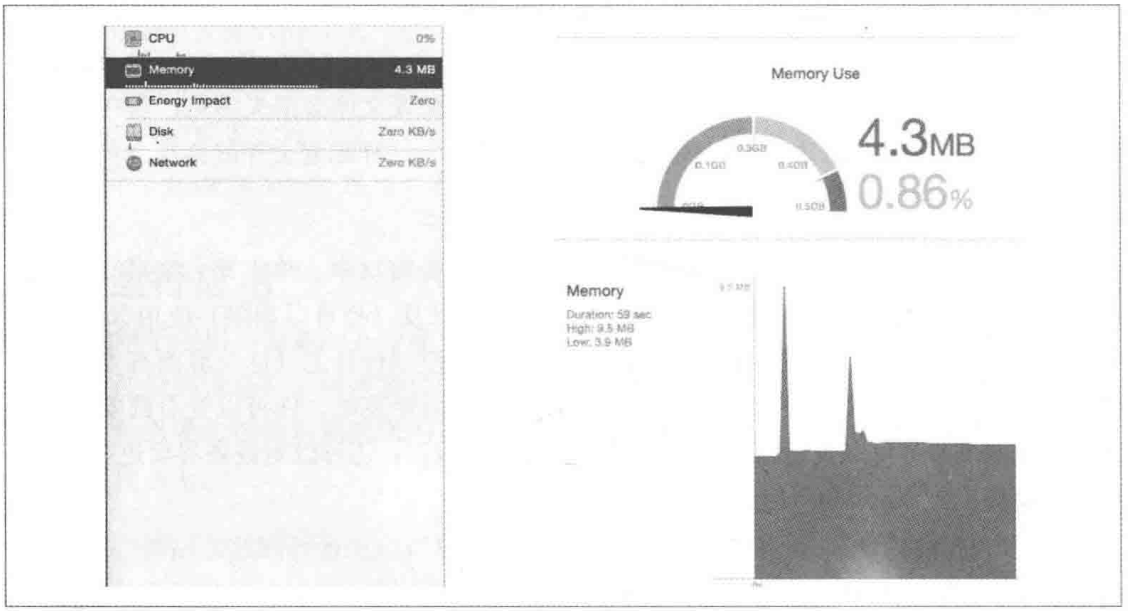


图 9-12：调试导航仪表盘

值得注意的是，图 9-12 是在设备上运行的结果。在模拟器上运行则会得到完全不同的结果，这个结果是不正确的。

## 9.9.2 Instruments

可以在模拟器或设备上使用 Instruments。在设备上进行的是最终的测试，目的是得到尽可能准确的结果。

要使用 Instruments，请在项目窗口工具栏的方案弹出菜单中设置所需的目标，然后选择 Product → Profile。这样应用就会使用方案下的 Profile 动作进行构建；在默认情况下，这会使用发布构建配置，这可能就是你所期望的。如果在设备上运行，那么你可能会看到一些验证警告，不过可以忽略它们。Instruments 会启动；如果方案的 Instrument 弹出

菜单将 Profile 动作设为了 Ask on Launch（默认值），那么 Instruments 就会弹出一个对话框，你可以从中选择追踪模板。

此外，还可以单击调试导航器仪表盘编辑器中的 Profile In Instruments；如果仪表盘发现了问题，你又想通过更为详尽的 Instruments 监控来重现问题，这么做就是很方便的。Instruments 启动后，选择合适的追踪模板。对话框会给出两个选择：Restart 会先停止应用，然后使用 Instruments 再次启动；Transfer 则会保持应用的运行，并将 Instruments 挂接到应用中。

当 Instruments 的主窗口出现时，可以进一步对其定制来分析感兴趣的数据，可以将 Instruments 窗口的结构保存为自定义模板。需要单击 Record 按钮，或选择 File → Record Trace 来运行应用。现在，可以像用户那样与应用交互了，而 Instruments 则会记录下统计数据。

如果之前将发布配置的代码签名身份构建设置为 iOS Distribution 来归档或分发应用（本章后面将会介绍），那就无法在设备上使用 Instruments 进行分析了。必须要将该构建设置为 iOS Developer。

Instruments 的使用是个高级主题，这超出了本书的讨论范围。事实上，仅是介绍 Instruments 本身就可以写一本书。要想了解进一步的信息，请参考 Apple 的文档，特别是 *Instruments User Reference* 与 *Instruments User Guide*。此外，往年的很多 WWDC 都有关于 Instruments 的视频介绍；请查找名字中包含“Instruments”或“Performance”的资料。这里仅仅简单介绍一下 Instruments 到底能做什么。

图 9-13 展示了在 Instruments 中能够完成与图 9-12 调试导航器仪表盘所能完成的相同事情。我将目标设定为我的设备。选择 Product → Profile；当 Instruments 启动后，我选择 Allocations 追踪模板。当应用在 Instruments 下运行时，我使用了一会儿，然后暂停了 Instruments，与此同时它会绘制应用的内存使用情况表。查看这张表，我发现内存使用量最高达到了 10MB，不过大部分时间，内存使用量都处在一个较低的水平上（不到 4MB）。我对这个结果感到很满意。

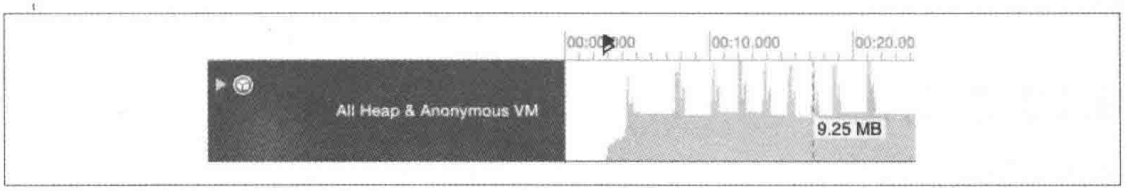


图 9-13：Instruments 以图形化形式展示了一段时间内的内存使用

Instruments 的另一个强大之处就是检测内存泄漏的能力。在图 9-14 中，我运行了第 5 章的保持循环代码：有一个 Dog 类实例和一个 Cat 类实例，它们彼此间都引用了对方。没有其他引用再指向这两个实例了，因此它们都存在泄漏问题。我通过 Leaks 追踪模板来分析应用。Instruments 检测到了泄漏，甚至还绘制了图表展示了错误的结构！

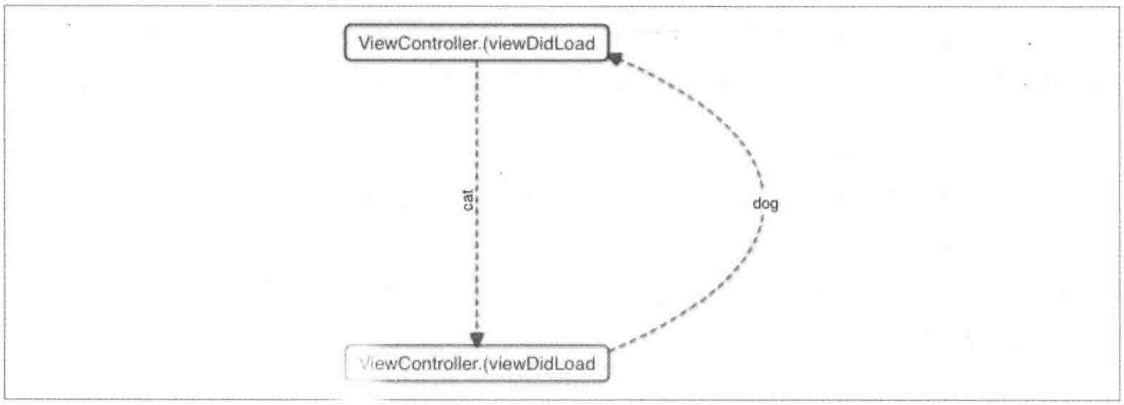


图 9-14：Instruments 展示了保持循环

在最后这个示例中，我想知道是否可以缩短 Diabelli's Theme 应用加载图片的时间。我将目标设为了设备，因为只有真正的设备才能体会到速度的重要性并且需要进行度量。选择 Product → Profile。Instruments 启动，我选择了 Time Profiler 追踪模板。当应用在设备上随 Instruments 启动后，我不断加载新图片来执行这部分代码。

在图 9-15 中，我已经暂停了 Instruments，看看图上都有什么。打开窗口下方的小三角，我可以钻取到自己的代码，这是由模块名 MomApp2 所标识的（之所以叫这个名字是因为一开始是将这个应用作为我母亲的生日礼物的）。

双击这一行可以看到自己代码的执行时间（如图 9-16 所示）。分析器所指出的对 `CGImageSourceCreateThumbnailAtIndex` 的调用引起了我的注意；此处消耗了大部分的 CPU 时间。该调用位于 ImageIO 框架中；它并不是我写的代码，因此我对其速度的提升无能为力。不过，我可以通过另外一种方式加载图片；比如，以一些临时的内存作为代码，我可以将图片全部加载进来并缩放。如果担心速度问题，我可以花点时间做试验。关键在于我现在知道了该如何做试验。这只不过是 Instruments 所擅长的基于事实的数值分析的一个方面而已。

# 9.10 本地化

用户可以在设备上将某种语言作为其主语言。你可能希望应用界面的文本能够对用户的选择做出响应，从而以用户所选的语言来显示。这是通过应用语言的本地化来实现的。你可能会在应用生命周期相对较晚的时间（应用已经开发完毕，准备发布）实现本地化。

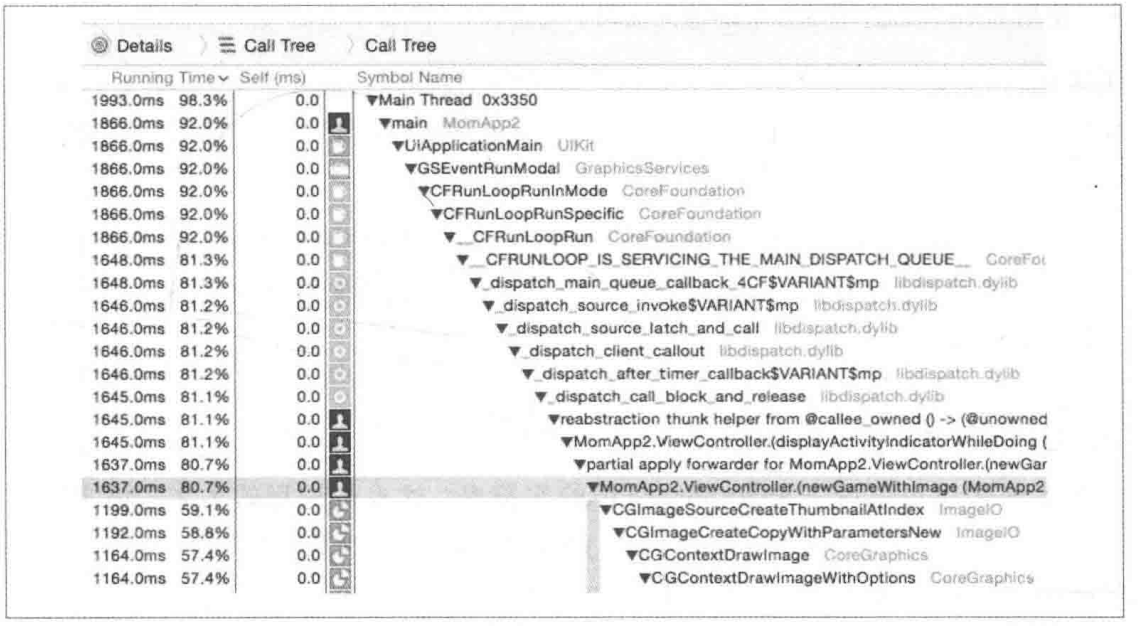


图 9-15：钻取到时间表

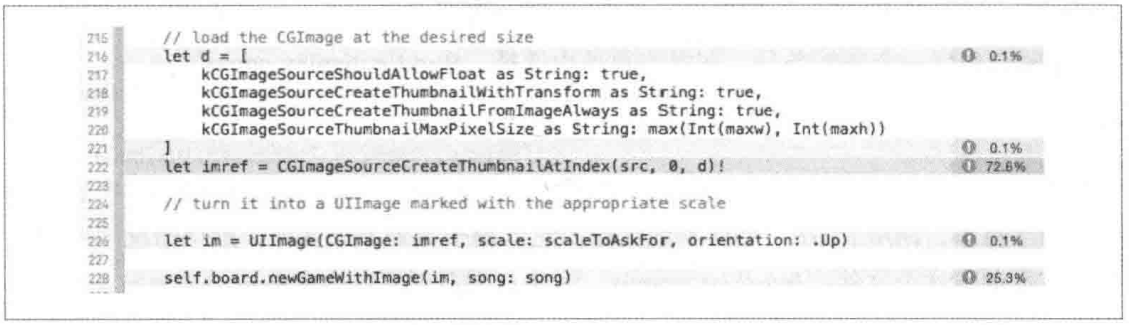


图 9-16：对我的代码进行时间分析

本地化是通过项目目录与构建好的应用包中的本地化目录来实现的。假设一个本地化目录中的资源在另外一个本地化目录中也有对应的一份。在应用加载该资源时，它会自动加载适合用户首选语言的那个。

任何类型的资源都可以放在这些本地化目录中；比如，一种语言会加载图片的一个版本，而另一种语言会加载这张图片的另一个版本。不过，你最应该关心的还是显示在界面上的文本。这些文本需要以特殊的格式化 *.strings* 文件的形式进行维护，并带有特殊的名字。比如：

- 使用 *InfoPlist.strings* 来本地化 *Info.plist* 文件。
- 使用 *Main.strings* 来本地化 *Main.storyboard*。

- 使用 *Localizable.strings* 来本地化代码字符串。

无须再手工创建或维护这些文件了。相反，可以通过标准的 *.xliff* 格式使用导出的 XML 文件。Xcode 会根据项目的结构与内容自动生成这些文件；还会读取它们，并将其自动转换为各种本地化的 *.strings* 文件。

为了帮助你理解 *.xliff* 导出与导入过程的工作方式，首先介绍如何手工创建和维护 *.strings* 文件；接下来介绍如何通过 *.xliff* 文件完成同样的事情。我将使用之前的 Empty Window 项目作为该示例的基础。

## 9.10.1 本地化 Info.plist

首先本地化应用图标下 Springboard 中的字符串，这也是应用的名字。该字符串是 *Info.plist* 文件中 `CFBundleDisplayName` 键的值。如果 *Info.plist* 文件中没有 `CFBundleDisplayName` 键，那就先要创建一个：

1. 编辑 *Info.plist* 文件。
2. 选中“Bundle name”，然后单击右侧的“+”按钮。
3. 这时会出现一个新的条目。在弹出菜单中选择“Bundle display name”。
4. 输入“Empty Window”作为新值并保存。

现在本地化该字符串：如果设备的语言是法语，那么我们需要在 Springboard 中显示出不同的字符串。*Info.plist* 该如何本地化呢？它依赖于另外一个文件，默认情况下应用模板并不会创建这个文件：*InfoPlist.strings*。因此，需要创建这个文件：

1. 选择 File → New → File。
2. 选择 iOS → Resource → Strings File，单击 Next 按钮。
3. 确保该文件属于应用目标，将其命名为 InfoPlist，注意名字与大小写，单击 Create 按钮。
4. 这时，一个名为 *InfoPlist.strings* 的文件会出现在项目导航器中。将其选中，在文件查看器中单击 Localize 按钮。
5. 这时会弹出一个对话框，让我们选择初始语言。默认是 Base，这就可以，单击 Localize 按钮。

现在准备添加语言！下面是具体步骤：

1. 编辑项目。在 Info 下，Localizations 表格列出了应用的本地化信息。我们一开始只对开发语言做了本地化（我选择的是英语）。

2. 单击 Localizations 表格下方的“+”按钮。从弹出的菜单中选择 French。
3. 这时会弹出一个对话框，列出了当前已经针对英语进行了本地化的文件（因为它们是应用模板的一部分）。我们这里只操作 *InfoPlist.strings*，因此勾选上它，同时不要勾选其他的文件，单击 Finish 按钮。

我们现在已经创建了 *InfoPlist.strings* 供英语与法语本地化使用。在项目导航器中，*InfoPlist.strings* 的清单已经有了一个三角箭头。展开这个三角箭头，我们会看到项目现在包含了 *InfoPlist.strings* 的两个副本，一个用于 Base（即英语），一个用于法语。现在就可以分别编辑这两个文件了。

下面编辑 *InfoPlist.strings* 文件。*.strings* 文件是个键值对的集合，其格式如下所示：

```
/* Optional comments are C-style comments */
"key" = "value";
```

对于 *InfoPlist.strings*，键是 *Info.plist* 中的键名，即原始的键名而不是类似于英语的那个名字。这样，英语的 *InfoPlist.strings* 文件应该如下所示：

```
"CFBundleDisplayName" = "Empty Window";
```

法语的 *InfoPlist.strings* 应该如下所示：

```
"CFBundleDisplayName" = "Fenêtre Vide";
```

就是这些！下面来试一下：

1. 在模拟器中构建并运行 Empty Window。
2. 在 Xcode 中，停止运行着的项目。在模拟器中会显示出主界面。
3. 查看应用的名字，它显示在模拟器主界面中（Springboard），名字是 Empty Window（也许会有截断）。
4. 在模拟器中，打开设置应用，将语言修改为 French（General → Language & Region → iPhone Language → Français），单击 Done 按钮。系统会提示我们是否要修改为 French。确定。
5. 短暂的停顿之后，语言就会改变。关掉设置应用，再次在 Springboard 中查看应用。其名字现在已经显示为了 Fenêtre Vide！

有意思吧？操作之后，请将模拟器的语言改回到 English。

## 9.10.2 本地化 nib 文件

现在介绍一下如何本地化 nib 文件。曾经，我们需要本地化整个 nib 副本。比如，如果需要法语版本的 nib 文件，你就需要维护两个单独的 nib 文件。如果在一个 nib 文件中创建了一个按钮，那就需要在另一个 nib 文件中创建一个相同的按钮——只不过一个按钮上的文字是英语，另一个是法语。诸如此类，每个界面对象与每个本地化语言都如此。看起来太枯燥了吧？

时至今日，我们已经有了更好的方式。如果项目使用了基础国际化，那么 *Base.lproj* 目录中创建的 nib 文件与本地化目录中创建的 *.strings* 文件之间就可以形成一种对应关系。这样，开发者只需维护一个 nib 文件副本即可。如果应用所运行的设备的本地化语言有对应的 *.strings* 文件，那么 *.strings* 文件中的字符串就会替换掉 nib 文件中的字符串。

在默认情况下，Empty Window 项目会使用基础国际化，其 *Main.storyboard* 文件位于 *Base.lproj* 目录中。我们准备将故事板文件本地化为法语。你还需要对故事板文件做些操作才能进行本地化：

1. 编辑 *Main.storyboard*，确保初始主视图包含一个按钮，按钮上的文字为 "Hello"。如果没有就添加一个。将按钮宽度设为 100 像素，保存（这很重要）。
2. 继续编辑 *Main.storyboard*，打开文件查看器。在 Localization 下，Base 应该已经勾选了。此外，勾选 French。
3. 在项目导航器中，查看 *Main.storyboard* 列表。它现在应该有一个小三角，展开这个小三角。当然，现在应该会有一个基于 Base 的本地化 *Main.storyboard* 与一个基于 French 的本地化 *Main.strings*。
4. 编辑 French *Main.strings*。它会自动创建出来，其键对应于 *Main.storyboard* 中每个有文本的界面元素。你需要从注释与键名中推断出这种对应关系。对于这个示例来说，*Main.storyboard* 中只有一个界面元素，因此很容易就能猜出来键代表的是哪个界面元素。它应该如下所示：

```
/* Class = "UIButton"; normalTitle = "Hello"; ObjectID = "PYn-zN-WlH"; */
"PYn-zN-WlH.normalTitle" = "Hello";
```

5. 将第 2 行（包含键值对这一行）的值修改为“Bonjour”。不要修改键！它是自动生成的，也是正确无误的，用于指定值与按钮文本之间的对应关系。

运行项目并查看界面。由于现在是在查看自己的应用，有一个更快的方式可以在各种本地化语言中查看：相对于切换设备语言，可以切换应用语言。要做到这一点，请编辑方案，在运行动作的 Options 页签中修改应用语言弹出菜单。当然，当应用使用法语时，按钮上的文本显示为 "Bonjour"！

如果修改 nib 会出现什么结果呢？假设在 *Main.storyboard* 中向视图再添加一个按钮。这时与 nib 对应的 *.strings* 文件不会发生任何变化；我们需要手工重新生成这些文件（这也是在实际情况下，为何要在界面开发工作基本完成时才开始本地化 nib 文件的原因所在）。不过内容并未丢失：

1. 选中 *Main.storyboard*，然后选择 File → Show in Finder。
2. 运行 Terminal。输入命令 `xcrun ibtool --export-strings-file output.strings`，后跟一个空格，然后将 *Main.storyboard* 从 Finder 拖曳到 Terminal 窗口，按回车键。

结果就是基于 *Main.storyboard* 的，名为 *output.strings* 的新文件会在主目录（也就是当前目录）下生成。可以根据 *Main.storyboard* 将这部分信息与现有的本地化 *.strings* 文件合并到一起。

在该示例中，我让你提前增加 `"Hello"` 按钮的宽度，从而为更长的本地化文本 `"Bonjour"` 留出足够的空间。在实际情况下，你可能会使用自动布局；这样按钮与标签就会自动伸缩了，同时界面的其他部分会相应地进行补偿。

要在不同本地化的情况下测试界面，还可以在 Xcode 中预览本地化 nib 文件，而无须运行应用。编辑 *.storyboard* 或 *.xib* 文件，打开辅助窗格，将追踪菜单切换至 Preview。右下角的菜单会列出本地化信息；可以在菜单中进行切换。“两倍长度的伪语言”会通过非常长的替换文本来测试界面在这种情况下的反应。

在 iOS 9 中，当应用运行在自右向左的语言中时，运行时会自动颠倒（镜像）整个界面及其行为。比如，推动变换会沿着老视图向右滑动，然后从左侧加载新视图。如果使用了自动布局和两端约束，那么界面就会颠倒过来，但如果代码依赖于从左向右的方向性，那就需要使用一些新的 UIView API。

## 9.10.3 本地化代码字符串

如何本地化其值是通过代码生成的字符串呢？在 Empty Window 应用中，轻拍按钮所弹出的警告就是个很好的示例。它会显示文本—警告的标题与消息，以及用于关闭警告的按钮文本：

```
@IBAction func buttonPressed(sender:AnyObject) {
    let alert = UIAlertController(
        title: "Howdy!", message: "You tapped me!", preferredStyle: .Alert)
    alert.addAction(
        UIAlertAction(title: "OK", style: .Cancel, handler: nil))
    self.presentViewController(alert, animated: true, completion: nil)
}
```

该文本该如何本地化呢？方式是一样的（需要一个 *.strings* 文件），不过需要修改代码才能显式使用它。代码会调用全局的 `NSLocalizedString` 函数；函数的第 1 个参数是 *.strings* 文件中的键，注释参数给出了非常好的说明，比如，待翻译的原始文本。`NSLocalizedString` 还接收几个可选参数；如果省略，那么默认会使用一个名为 *Localizable.strings* 的文件。

比如，我们将 `buttonPressed:` 方法修改为下面这个样子：

```
@IBAction func buttonPressed(sender:AnyObject) {
    let alert = UIAlertController(
        title: NSLocalizedString("ATitle", comment:"Howdy!"),
        message: NSLocalizedString("AMessage", comment:"You tapped me!"),
        preferredStyle: .Alert)
        alert.addAction(
            UIAlertAction(title: NSLocalizedString("Accept", comment:"OK"),
                style: .Cancel, handler: nil))
        self.presentViewController(alert, animated: true, completion: nil)
}
```

当然，上述代码是有问题的，因为没有 *Localizable.strings* 文件。下面创建一个，过程与之前一样：

1. 选择 File → New → File。
2. 选择 iOS → Resource → Strings File，单击 Next 按钮。
3. 确保该文件属于应用目标，将其命名为 *Localizable*，注意名字与大小写，单击 Create 按钮。
4. 这时，一个名为 *Localizable.strings* 的文件会出现在项目导航器中。将其选中，在文件查看器中单击 Localize 按钮。
5. 这时会弹出一个对话框，让我们选择初始语言。默认是 Base，这就可以，单击 Localize 按钮。
6. 在文件导航器中勾选 French。

现在，*Localizable.strings* 文件对应于两个本地化，Base（即 English）与 French。我们需要在文件中添加内容。就像之前使用 `ibtool` 那样，可以通过 `genstrings` 工具自动生成初始内容。比如，我会在计算机上打开 Terminal，然后输入 `xcrun genstrings`，后跟空格。接下来将 *ViewController.swift* 从 Finder 拖曳到 Terminal 窗口，按回车键。这会在当前目录下生成一个 *Localizable.strings* 文件，其内容如下所示：

```
/* OK */
"Accept" = "Accept";
```

```
/* You tapped me! */
"AMessage" = "AMessage";

/* Howdy! */
"ATitle" = "ATitle";
```

将上述内容复制并粘贴到项目 *Localizable.strings* 文件的 English 与 French 版本中，然后检查键值对，修改每一个键值对的值，使得值是我们所需要的。比如，在 English 版的 *Localizable.strings* 文件中：

```
/* Howdy! */
"ATitle" = "Howdy!";
```

在 French 版的 *Localizable.strings* 文件中：

```
/* Howdy! */
"ATitle" = "Bonjour!";
```

以此类推。

## 9.10.4 使用 XML 文件进行本地化

从 Xcode 6 开始，我们可以通过另外一种方式完成之前的工作。从表面来看，文本本地化可以看作对 *.xliff* 文件导入与导出的解析。这意味着你实际上无须按下任何 Localize 按钮或编辑任何 *.strings* 文件！相反，你可以编辑目标并选择 Editor → Export For Localization；在保存时，Xcode 会创建一个目录，里面包含了用于各种本地化的 *.xliff* 文件。接下来编辑这些文件（或让专门负责编辑的人帮你）并将编辑好的文件导入；编辑目标并选择 Editor → Import Localizations。Xcode 会读取编辑好的 *.xliff* 文件并完成其他操作，根据需要自动创建好本地化，生成或修改 *.strings* 文件。

为了演示，我们再向本地化添加一种语言——Spanish。

1. 编辑目标并选择 Editor → Export For Localization。
2. 我们可以在现有的本地化与基础语言中导入字符串。如果要编辑 French 本地化，那就需要将其导出，不过我不打算在该示例中这么做。相反，只需要将 Include 弹出菜单切换至 Development Language Only 即可。
3. 指定好保存的位置（如桌面）。这时要创建一个目录，因此请不要让目录名与保存位置处的现有目录重名。比如，如果保存到包含了项目目录的相同目录下，那么可以将其命名为 Empty Window Localizations，单击 Save 按钮。
4. 在 Finder 中，打开刚才创建的目录。它包含了项目基础语言的 *.xliff* 文件。比如，我的文件叫作 *en.xliff*，因为开发语言是 English。

查看这个 *.xliff* 文件，你会看到 Xcode 已经帮我们做好了之前需要手工完成的一切。不再需要 *.strings* 文件了！ Xcode 完成了所有工作：

- 对于项目中的每个 *Info.plist* 文件，Xcode 都会创建一个相应的 `<file>` 元素。在导入时，这些文件会转换为本地化的 *InfoPlist.strings* 文件。
- 对于每个 *.storyboard* 与 *.xib* 文件，Xcode 都会运行 `ibtool` 来提取出文本，并且创建相应的 `<file>` 元素。在导入时，这些元素会转换为齐名的本地化 *.strings* 文件。
- 对于包含了对 `NSLocalizedString` 调用的每个代码文件，Xcode 都会调用 `genstrings`，并且创建相应的 `<file>` 元素。在导入时，这些元素会转换为本地化 *Localizable.strings* 文件。

我们现在继续将该文件中的字符串翻译为其他语言，保存编辑好的 *.xliff* 文件，将其导入：

1. 在合适的文本编辑器（或 XML 编辑器）中打开 *.xliff* 文件。
2. 对于该示例来说，我只对故事板中的 `"Hello"` 按钮进行本地化。因此，删除（请小心，不要搞乱了 XML）除 `original` 属性为 `"Empty Window/Base.lproj/Main.storyboard"` 的其他所有 `<file>...</file>` 元素组。删除除 `<source>` 为 `"Hello"` 的其他所有 `<trans-unit>...</trans-unit>` 元素。
3. 将 Spanish 作为目标语言，向 `<file>` 元素添加一个属性：`target-language="es"`。
4. 提供一个翻译，在 `<source>` 元素后添加一个 `<target>` 元素，加上一些文本，如 `"Hola"`。文件的内容现在应该如下所示：

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xliff xmlns="urn:oasis:names:tc:xliff:document:1.2"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.2"
  xsi:schemaLocation="urn:oasis:names:tc:xliff:document:1.2
  http://docs.oasis-open.org/xliff/v1.2/os/xliff-core-1.2-strict.xsd">
  <file original="Empty Window/Base.lproj/Main.storyboard"
  source-language="en" target-language="es" datatype="plaintext">
    <header>
      <tool tool-id="com.apple.dt.xcode" tool-name="Xcode"
      tool-version="6.2" build-num="6C107a"/>
    </header>
    <body>
      <trans-unit id="PYn-zN-WlH.normalTitle">
        <source>Hello</source>
        <target>Hola</target>
        <note>Class = "UIButton"; normalTitle = "Hello";
          ObjectID = "PYn-zN-WlH";</note>
      </trans-unit>
    </body>
  </file>
</xliff>
```

5. 回到 Xcode，编辑目标并选择 Editor → Import Localizations。在 Open 对话框中，选择编辑好的 *en.xliff* 并单击 Open。

6. Xcode 会提示我们没有翻译全部内容。忽略该提示并单击 Import。

下面就是见证奇迹的时刻！在没有任何提示的情况下，Xcode 为本地化添加了 Spanish，并且又创建了一个 *InfoPlist.strings* 文件、一个 *Main.strings* 文件和一个 *Localizable.strings* 文件，所有这些文件都本地化为 Spanish。查看 *Main.strings*，你会发现其内容与我们手工编辑的一模一样：

```
/* Class = "UIButton"; normalTitle = "Hello"; ObjectID = "PYn-zN-WlH"; */
"PYn-zN-WlH.normalTitle" = "Hola";
```

显然，*.xliff* 文件是创建并维护本地化的一种非常便捷的手段。项目中本地化的结构与本节之前介绍的完全一样，不过 *.xliff* 文件将相同的信息具化为可通过单个文件编辑的格式。*.xliff* 导出过程会使用 `ibtool` 与 `genstrings`，这样在添加界面和代码时就可以轻松维护本地化内容了。

## 9.11 归档与发布

发布指的是将构建好的应用提供给不是你的团队中的开发者的其他人，并在他们的设备上运行。有两种发布方式：

*Ad Hoc 发布*

将应用副本提供给有限的一些已知用户以便他们能够在自己特定的设备上使用并报告 Bug、提出建议等。

*App Store 发布*

将应用提供给 App Store，这样任何人都可以下载（可能是收费的）并运行应用。

要想创建应用副本来发布，首先需要构建应用归档，随后将这个归档导出供 Ad Hoc 或 App Store 发布使用。归档本质上是个保存好的构建。它有 3 个主要的目的：

*发布*

归档作为 Ad Hoc 发布或 App Store 发布的基础。

*重现*

每次构建时，条件都可能会发生变化，这样生成的应用的行为就可能出现些许不同。不过归档会保留特定的二进制构建；通过特定归档的每次发布都可以确保包含相同的二进制文件，这样其行为就是完全一致的。这对于测试非常重要：如果 Bug 报告

是根据从特定归档发布的应用生成的，那么你可以通过 Ad Hoc 发布该归档，并在自己的设备上运行，这时测试的就是完全一样的应用。

*符号化*

归档包含了一个 .dSYM 文件，Xcode 可以通过它接收到崩溃日志并报告代码中的崩溃位置。这样就可以处理来自于用户的崩溃报告了。

下面介绍如何构建应用归档：

1. 将项目窗口工具栏方案弹出菜单中的目标设为 iOS Device。设定好之后，Product → Archive 菜单项将会被禁用。无须再连接设备；你所构建的输出并不是要在特定的设备上运行，而是要构建在某些设备上运行的归档。
2. 如果愿意，还可以编辑方案，确认发布构建配置用于 Archive 动作。这是默认值，不过复查一下也没什么坏处。
3. 选择 Product → Archive。应用会编译并构建。归档本身会存储到用户目录 *Library/Developer/Xcode/Archives* 下的一个日期目录中。此外，它还会列在 Xcode 组织器窗口（Window → Organizer）Archives 下；该窗口可能会自动打开，显示刚才创建的归档。你可以在这里添加注释；还可以修改归档的名字（这并不会对应用的名字造成影响）。

要想基于归档进行发布，你还需要一个发布身份（电脑钥匙链中存储的一个私钥和一个发布证书）和针对该应用的发布配置。如果进行 Ad Hoc 发布与 App Store 发布，那就需要针对每个发布的单独的发布配置。只有开发者计划成员才能获得发布身份与配置文件。

如果在组织器窗口中看到如下信息："Distribution requires enrollment in the Apple Developer Program"，那就说明你之前并没有注册开发者计划成员。现在正是时候！如果没有开发者计划成员，组织器窗口就像个蟑螂汽车旅馆：只能登记，无法结账。

可以像之前介绍的获取开发身份那样在 Xcode 中获取发布身份：在账户首选项窗格团队 View Details 对话框中，单击 iOS Distribution 右侧的 Create 按钮。如果不起作用，那就请手工获取发布证书，就像之前介绍的手工获取开发证书一样。

从理论上来说，在导出归档时，Xcode 还会创建出恰当的发布配置。不过，这个功能常常不好用；我总是通过浏览器在会员中心手工创建发布配置。下面是具体做法：

1. 如果是 Ad Hoc 发布配置，那么请收集应用所要运行的所有设备的 UDID，然后在会员中心 Devices 下将其添加进去（对于 App Store 发布配置，请忽略这步）。
2. 确保应用在会员中心注册过了，就像本章之前所介绍的那样。

3. 在会员中心 Provisioning Profiles 下，单击“+”按钮添加一个新的配置。在 Add iOS Provisioning Profile 表单中，指定一个 Ad Hoc 配置或 App Store 配置。在下一个页面中，从弹出菜单中选择应用。接下来，选择发布证书。然后（只针对 Ad Hoc 发布），指定希望这个应用所运行的设备。在下一个页面中，为配置起个名字。

   请注意配置的名字，因为接下来需要在 Xcode 中能够识别出这个名字！我的做法是通过单词“Ad Hoc”或“AppStore”再加上应用名作为配置的名字。

4. 单击 Generate 生成配置。要想获得该配置，请单击 Download，然后找到下载的配置并双击它在 Xcode 中查看，或是打开 Xcode 账户首选项窗格 View Details 对话框，单击左下角的 Download All 按钮让 Xcode 下载它。

## 9.12 Ad Hoc 发布

Apple 文档认为 Ad Hoc 发布构建应该包含一个图标，显示在 iTunes 中，不过根据我的经验，这一步虽然起作用，但没必要。如果想要加入这个图标，那么它应该是个 PNG 或 JPEG 文件，512×512 像素大小，名字应该是 *iTunesArtwork*，并且没有文件扩展名。请确保将图标加到构建中，在 Copy Bundle Resources 构建阶段完成。

下面是创建 Ad Hoc 发布文件的步骤（假设你已经有了发布身份，如 9.11 节所介绍的那样）：

1. 如果必要，创建、下载并安装该应用的 Ad Hoc 发布配置，就像 9.11 节介绍的那样。
2. 如果必要，创建应用归档，就像 9.11 节介绍的那样。在创建归档前，双击代码签名构建设置：发布构建的代码签名身份（或方案对于归档动作所用的任何构建配置）应该是 iOS Distribution，配置文件应该是 Automatic（可以更加精细地指定这些设置，不过现在这些通用设置就足够了）。
3. 在组织器窗口 Archives 下，选中归档并单击窗口右上角的 Export 按钮。这会弹出一个对话框。你可以指定一个方法；选择 Save for Ad Hoc Deployment，单击 Next。
4. 现在需要选择一个开发团队。选择正确的团队并单击 Choose。
5. 在 Xcode 7 中，你会看到一个对话框，询问是否要导出精简的应用，这表示应用只会包含适用于一种设备类型的资源，这会在用户将应用下载到设备上时模拟 App Store 的做法。你可能不需要这么做，不过知道精简后的应用大小总归是有用的。
6. 归档会准备好，并且会显示出一个摘要窗口。配置文件的名字会显示出来，你可以确认一下。单击 Next。
7. 文件会被保存到桌面上的一个目录中，其后缀名为 *.ipa*（“表示 iPhone app”）。

8. 在 Finder 中找到刚才保存的文件，将该文件发送给用户。

用户应该将 *.ipa* 文件复制到安全的地方，如桌面，然后启动 iTunes，并将 *.ipa* 文件从 Finder 拖曳到 Dock 的 iTunes 图标上（或双击 *.ipa* 文件）。然后需要将设备连接到电脑上，确保该应用位于此设备可用的应用列表中，它会在下次同步时安装到设备上，最后，同步设备会将应用复制到设备上（如果这并非发布给 Ad Hoc 测试者的第一个版本的应用，那么用户可能需要先删除设备上的当前版本；否则在同步时，新版本可能无法复制到设备上）。

如果将自己的设备作为该 Ad Hoc 发布配置将会启用的设备之一，那么你就可以遵循这些指令以确保 Ad Hoc 发布能像预期一样使用。首先，请将设备中该应用之前的版本全部删除（比如，开发副本等），同时还要删除与该应用相关的配置（可以通过 Xcode 的设备窗口完成）。接下来像之前介绍的那样，通过与 iTunes 同步将应用复制到设备中。现在应用应该可以运行在设备上了，你会在设备上看到 Ad Hoc 发布配置。因为你自己的权限与其他 Ad Hoc 测试者一样，所以你这里的使用情况应该和其他测试者一样。

每年每个开发者（不是每个应用）有 100 个设备的注册限制，这限制了 Ad Hoc 测试者的数量。这个数量不利于用于开发的设备。你可以突破这个限制，向用户更便捷地提供 Beta 版的应用，方式就是使用 TestFlight Beta 测试。

TestFlight Beta 测试将 100 个设备的限制提升到了 1000 个测试者，并且要比 Ad Hoc 发布更加方便，这是因为用户可以通过 TestFlight 应用（Apple 在 2014 年通过收购 Burstly 而获得）直接从 App Store 就可以将预发布版本的应用安装到设备上。其配置是在 iTunes Connect 网站上进行的；上传到 iTunes Connect 的预发布版本必须要像 App Store 发布那样归档（参见本章后面介绍的 App Store 提交）。具体请参见 Apple *iTunes Connect Developer Guide* 的“TestFlight Beta Testing”一章。

应用的预发布版本旨在发布给 Beta 测试者（与可以直接访问你的 iTunes Connect 账户的内部测试者不同），这需要 Apple 的审核才行。

## 9.13 最后的准备

随着将应用提交到 App Store 日期的日益临近，请不要让应用的美好前景或巨大的利润搞乱了你的节奏，导致越过了应用最后准备阶段的各个重要步骤。Apple 对应用有很多要求，如果不满足这些要求会导致应用提交被拒。请花点时间做好准备，列个清单，然后仔细检查。参见 Apple 的 *App Distribution Guide* 与 *Human Interface Guidelines* 的“Icon and Image Design”一章了解详情。

## 9.13.1 应用图标

为应用提供图标的最简单的方式是使用资源目录。如果之前没有对图标使用资源目录，而现在又想使用，那么请编辑目标，在通用窗格 App Icons and Launch Images 下，App Icons Source 旁边单击 Use Asset Catalog 按钮。之后，Use Asset Catalog 按钮会变成一个弹出菜单，列出资源目录名与目录中用作图标的图片集的名字。

所需的图片大小会列在资源目录中。选中一个图片，然后在属性查看器中查看。令人困惑的是，“2x”与“3x”表示图片大小应该是列出的图标大小的 2 倍与 3 倍；比如，iPhone 应用图标显示为“60pt”或“60×60”，不过“3x”表示你应该提供一个 180×180 大小的图片。要想确定该显示哪一个，在选中图标集或加载图片集时请勾选上属性查看器中的复选框（如图 9-17 所示）。要想添加图片，请将其从 Finder 拖曳到恰当的位置处。

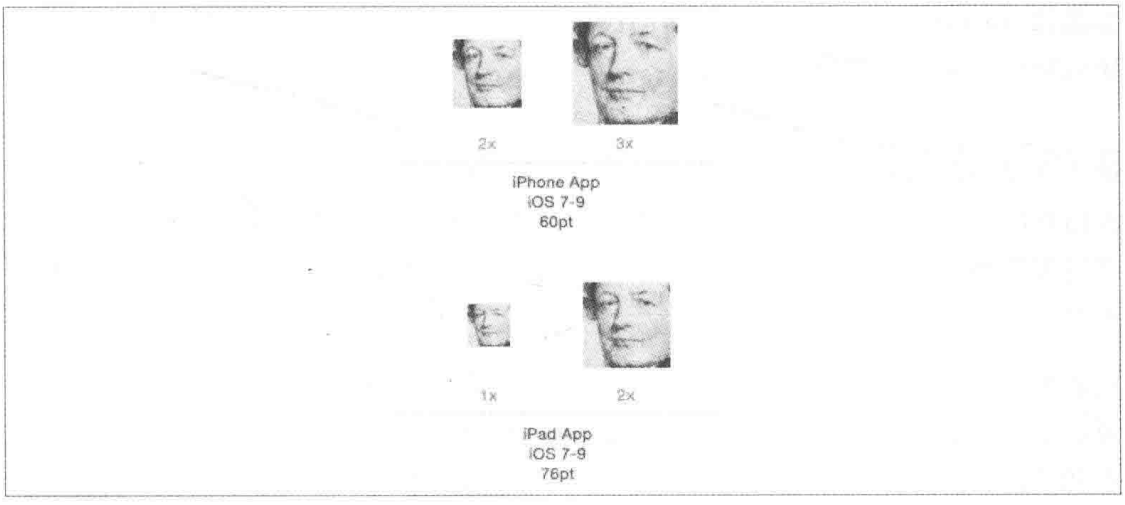


图 9-17：资源目录中的图标位置

图标文件必须是个 PNG 文件，不能有 alpha 透明度。它应该是个正方形，系统会为其添加圆角。目前，Apple 似乎更喜欢简单、卡通的图片，拥有明亮的颜色以及渐变的背景色。

在构建应用并处理资源目录时，图标会被写到应用包的顶层并被赋予恰当的名字（如图 6-15 所示）；同时，一个恰当的条目会被写到应用的 *Info.plist* 中，这样系统就可以找到图标并在设备上显示了。具体细节很复杂，不过你不必关心这些，这也正是使用资源目录的原因所在！

应用图标大小随着时间的变化也在发生着变化。如果应用要向后兼容于早期系统，那么你还需要拥有不同尺寸的额外的图标，以满足这些老系统的需要。这正是资源目录的价值所在。

此外，还可以加入更小的图标，用于在用户进行搜索时显示，如果使用了设置包，那么还会显示在 Settings 应用中。不过，我从来都没有使用过这些图标。

## 9.13.2 其他图标

在向 App Store 提交应用时，你需要提供一个 1024×1024 大小的 PNG，或高质量的 JPEG 图标以显示在 App Store 中。Apple 指南说它不应该只是应用图标的放大版，同时也不能与应用图标差别太大，否则应用将会被拒绝（这一点是从我的经验得来的）。

App Store 图标不需要构建到应用中；事实上，它也不应该构建到应用中，因为这么做只会毫无必要地增加应用的大小。另外，可能想在项目中保留该图标（在项目目录中），这样就能轻松找到并维护它了。我建议将其导入项目中，并复制到项目目录中，但不要将其添加到任何目标中。

如果为 Ad Hoc 发布创建了 iTunesArtwork 图标，那么你现在可能需要将其从 Copy Bundle Resources 构建阶段中删除。

## 9.13.3 启动图片

在用户轻拍应用图标来启动应用与应用开始运行并显示初始窗口之间会有一个延迟。为了掩盖这种延迟，使用户觉得应用正在运行，你应该在这个时间间隔内显示一张启动图片。

启动图片无须追求细节，它可以是应用完成启动后界面主要元素或内容的一个简单描绘。通过这种方式，当应用启动完毕后，从启动图片到实际应用的过渡就是填充这些元素与内容的事情了。

在 iOS 7 与之前版本中，启动图片就是个图片（一个 PNG 文件）。它需要添加到应用包中，也需要遵循某些命名约定。随着 iOS 设备的屏幕尺寸与分辨率不断变化，启动图片的数量也随之发生了变化。iOS 7 引入的资源目录就派上了用场。不过随着 iPhone 6 与 iPhone 6 Plus 的出现，整个情况变得难以管理了。

出于这个原因，iOS 8 引入了更好的解决方案。相对于使用一组启动图片，你需要提供一个启动 nib 文件，即一个 *.xib* 或 *.storyboard* 文件，其中包含了作为启动图片显示的视图。可以通过子视图和自动布局来构建这个视图。这样，视图就会自动进行重新配置，匹配应用所运行的设备的屏幕尺寸与方向。

在默认情况下，新的应用项目都会带有一个 *LaunchScreen.storyboard* 文件，这是用于设计启动图片的文件。*Info.plist* 通过键“Launch screen interface file base name”

（UILaunchStoryboardName）来指向该文件。如果必要，可以通过编辑目标并设置 *Launch Screen File* 域（位于 App Icons and Launch Images 下）来配置 Info.plist。

你应该充分利用该特性，而不仅仅是因为这么做很方便。*Info.plist* 中的“Launch screen interface file base name”键告诉系统应用运行在更新的设备类型上，比如，iPhone 6 与 iPhone 6 Plus。如果没有这个键，那么应用就会缩放显示，就好像 iPhone 6 只是个巨大的 iPhone 5S 一样。实际上，你无法利用本可以使用的像素（显示会有些模糊）。

使用启动 nib 文件的另一个原因在于它是可以本地化的！与任何 *.xib* 和 *.storyboard* 文件一样，显示在基础本地化启动界面 *.xib* 或 *.storyboard* 文件中的字符串可以通过 *.strings* 文件进行本地化。

据我所知，应用包中的自定义字体是无法显示在启动 nib 文件中的。这是因为在启动界面显示时，它们尚未加载进来。

坏消息是如果应用要向后兼容于早期系统，那除了启动 nib 文件，你还需要提供老式的启动图片。iOS 7 及之前的系统对于启动图片的要求是非常复杂的，而且随着时间的流逝规则还发生了一些变化，这又加剧了复杂性，结果就是要兼容的系统越多，需要满足的条件就越多。我已经在本书的前一版中介绍过这些条件，这里就不再赘述了。

Apple 提供了一个名为 Application Icons and Launch Images for iOS 的非常有价值的示例代码项目。该项目提供了各种尺寸的图标与启动图片，同时还介绍了恰当的命名约定。

## 9.13.4 屏幕截图与视频预览

在向 App Store 提交应用时，你需要提供应用的一个或多个截图以显示在 App Store 上。你应该事先就准备好屏幕截图并在应用提交过程中提供它们。你至少需要根据应用所运行的设备的屏幕尺寸提供一张屏幕截图，并且使用相应的分辨率。

可以通过模拟器或与电脑连接的设备来创建屏幕截图：

*模拟器*

在模拟器中运行应用，首先设置目标以获得所需的设备类型。选择 File → Save Screen Shot。

设备

在 Xcode 的设备窗口中，在 Devices 下找到连接的设备，然后单击 Take Screenshot。此外，还可以选择 Debug → View Debugging → Take Screenshot of [Device]。

在这两种情况下，屏幕截图文件都会保存到电脑上通常用来保存屏幕截图的位置处（一般在桌面上）。

还可以同时按下锁屏按钮与 Home 按钮在设备上进行屏幕截图。这样，屏幕截图就会保存到照片应用的相机胶卷中，你可以通过任何方便的方式将其发送到电脑上（比如，给自己发邮件）。

你还可以向 App Store 提交用于介绍应用的视频预览。视频最多可以是 30 秒的时长，格式为 H.264 或 Apple ProRes。如果电脑使用的是 OS X 10.10（“Yosemite”）或更新的版本，那么它可以捕获到设备的视频。设备要新一些，拥有雷电连接器才行：

1. 将设备连接到电脑上并打开 QuickTime Player。选择 Choose File → New Movie Recording。
2. 如果必要，当鼠标悬浮在 QuickTime Player 窗口上时，使用 Record 按钮旁边向下的 v 形按钮打开弹出菜单，将相机与麦克风设为设备。
3. 开始录制，在设备上使用应用。录制完毕后，停止然后保存。

可以通过 iMovie 或 Final Cut Pro 编辑生成的影片文件，然后提交到 App Store。比如，在 iMovie 中：

1. 在导入影片文件后，选择 File → New App Preview。
2. 编辑！完成后，选择 File → Share → App Preview，确保得到的是正确的分辨率与格式。

要想了解更多信息，请参阅 Apple *iTunes Connect Developer Guide* “First Steps”一章中的“App Preview”一节。

## 9.13.5 属性列表设置

*Info.plist* 中的很多设置对于应用的行为都是至关重要的。你应该仔细阅读 Apple 的 *Information Property List Key Reference* 以了解全面的信息。大多数所需的键都是作为模板的一部分而创建的，并且赋予了合理的默认值，但你还是应该检查一下。下面这些键尤其值得你注意：

*Bundle display name*（CFBundleDisplayName）

位于设备屏幕上应用图标下方的名字；这个名字要短一些，以免被截断。本章之前曾介绍过如何本地化显示名。

*Supported interface orientations*（UISupportedInterfaceOrientations）

这个键指定了应用可以显示的方向。你可以通过目标编辑器 General 页签的复选框进行设置。不过可能还需要手工编辑 *Info.plist* 以重新排列可能的方向顺序，因为在 iPhone 上，列出的第一个方向是应用实际启动的方向。

*Required device capabilities*（UIRequiredDeviceCapabilities）

如果应用所需的能力并不是所有设备都具备，那么你就应该设置该键。对于应用来说，如果运行在缺乏特定能力的设备上是无意义的，那就不要使用该键。

*Bundle version*（CFBundleVersion）

应用需要一个版本号。最好在目标编辑器 General 页签中设置它。这里可能会让你有些迷惑，因为它有两个域：

*Version*

对应于 *Info.plist* 中的“Bundle versions string, short”（CFBundleShortVersionString）。

*Build*

对应于 *Info.plist* 中的“Bundle version”（CFBundleVersion）。

据我所知，如果设置了前者，那么 Apple 就会使用它，否则会使用后者。一般来说，在提交到 App Store 时，安全起见，请将这两个域设为相同的值。这个值是个版本字符串，如 "1.0"。版本字符串会显示在 App Store 中，用于区分各个版本的发布。提交更新时如果没有增加版本字符串会导致更新被拒。不过，增加 Build 号但没有增加 Version 号是可以的，如果提交了相同发布的几个连续构建，那就需要这么做了（在 TestFlight 测试过程中，或发现了 Bug，导致不得不在 App Store 上架前撤回提交的二进制文件）。

## 9.14 向 App Store 提交应用

如果觉得应用没问题，并且已经安装或收集好了所有必要的资源，那么你就可以向 App Store 提交应用进行发布了。要想做到这一点，你需要在 iTunes Connect 网站上做些准备工作。登录 Apple 网站后，你会在 iOS 开发者页面上发现一个指向它的链接。你可以直接访问 *http://itunesconnect.apple.com*，但还是需要使用 iOS 开发者用户名与密码登录。

访问 iTunes Connect 的第一件事就是进入 Contracts 部分，完成合同的提交。只有提交完合同后才能开始销售应用，即便免费应用也需要填写好合同表单。

我这里不想列出将应用提交给 iTunes Connect 的所有步骤，因为这些内容已经在 Apple 的 *iTunes Connect Developer Guide* 上有非常详尽的介绍，这都是非常权威的指南。下面介绍一些你需要提供的主要信息：

*应用的名字*

该名字将会出现在 App Store 上；它与设备上应用图标下的简短名字无需一致，后者是由 *Info.plist* 文件中的“Bundle display name”设置决定的。Apple 建议这个名字最多 25 个字符，不过也可以长一些。在向 iTunes Connect 提交应用信息后，你可能很不爽地发现你想起的名字已经被占用了；但你没法提前预知这一点，这样就得多花一些时间了。

*说明*

你需要提供一份小于 4000 字符的说明。Apple 建议说明长度要小于 580 个字符，第一段是最为重要的，因为这可能是用户访问 App Store 时一眼所能看到的全部内容。说明必须是纯文本，没有 HTML 和字体样式。

*关键词*

这是个逗号分隔的小于 100 个字符的列表。除了应用名，这些关键词用于帮助用户在 App Store 中找到你的应用。

*支持*

这是个网站的 URL，用户可以通过它找到关于应用的更多信息；最好提前就建好这个网站。

*版权*

不要在该字符串中加入版权符号，App Store 会帮你添加。

*SKU 号*

这个无关紧要，不用过多地考虑它。它只是个唯一标识符而已，在你自己的应用世界中是唯一的。如果它与应用名有关就很方便了。它不一定是个数字；可以是任意字符串。

*价格*

现在还没到定价的时候，你需要从价格“层次”列表中选择。

*上架日期*

其中有一个选项可以在应用审核通过后就立刻上架，不过你可以自己选择。

在提交信息时，请时不时地单击 Save！如果连接断了，同时又没有保存，所有工作都会丢失。（你能猜出我怎么知道这一点的吗？）

在 iTunes Connect 提交了关于应用的信息后，如果想要上传应用，那么可以使用 Xcode。你应该有一个 iOS 开发身份，应用也已经归档完毕（将发布配置的代码签名身份构建设置设为 iOS Distribution，这应该是使用 Ad Hoc 或 TestFlight 分发所创建的归档）。在组织器中选择归档构建并单击 Upload to App Store。这会上传应用，同时应用也会在服务端进行验证。

此外，还可以使用 Application Loader。将归档导出为 *.ipa* 文件用作 Ad Hoc 发布，不过在选择导出方式时，请选择 Save for iOS App Store Deployment。选择 Xcode → Open Developer Tool → Application Loader 来启动 Application Loader，并将 *.ipa* 文件交给它处理。

归档上传完毕后，还有最后一步。等待 5 ~ 10 分钟，让二进制文件在 Apple 服务端处理完。然后回到 iTunes Connect，也就是提交应用信息的地方。你现在可以选中二进制文件、保存，并提交应用进行审核了。

随后你会收到来自 Apple 的邮件，在应用状态经历了各个阶段时会通知到你：“Waiting For Review”“In Review”，如果一切顺利，那么最后则是“Ready For Sale”（即便免费应用也是如此）。接下来，应用就会出现在 App Store 上了。

# 第三部分

# Cocoa

Cocoa Touch 框架提供了 iOS 应用所需的一般功能。按钮可以按下、文本可以读取、界面可以一个接着一个出现，这些都是 Cocoa 的功劳。要想使用该框架，你需要先去学习。你得将代码放到正确的位置，这样才能在正确的时刻得到调用。你需要实现 Cocoa 期望你去做的事情。通过理解 Cocoa 来掌握它。本部分将会介绍这些内容。

- 第 10 章将会介绍 Cocoa 是如何通过子类化、类别与协议等 Objective-C 语言特性来组织和结构化的。接下来将会介绍一些重要的内建 Cocoa 对象类型。本章最后将会介绍 Cocoa 的键值编码，同时还会谈及根 NSObject 类的组织方式。
- 第 11 章将会介绍 Cocoa 的事件驱动的活动模型，以及其主要的设计模式和事件相关的特性：通知、委托、数据源、目标－动作、响应器链及键值观测等。本章最后将会给出关于如何管理 Cocoa 诸多事件的一些建议，以及如何通过延迟执行来规避事件泥潭。
- 第 12 章将会介绍 Cocoa 内存管理，这里将会谈及引用类型内存管理的工作方式。接下来将会介绍特殊的内存管理情况：自动释放池、保持循环、通知与定时器、nib 加载与 CFTypeRefs。本章最后将会介绍 Cocoa 属性的内存管理，并给出关于如何调试内存管理问题的一些建议。
- 第 13 章将会介绍在 Cocoa 世界中对象之间的可见性与通信问题。本章最后将会给出使用模型－视图－控制器架构的一些建议。

最后，不要忘记阅读附录 A 来深入了解 Objective-C 与 Swift 之间的交互方式。

# 第 10 章

# Cocoa 类

在进行 iOS 编程时，你实际上是在进行 Cocoa 编程，因此需要了解 Cocoa；你应该知道，在使用 Cocoa 时到底使用的是什么，以及 Cocoa 希望你应该怎样使用它们。Cocoa 是个庞大的框架，又细分为了多个小框架，熟悉 Cocoa 需要花费不少时间和精力。不过，Cocoa 有一些重要的约定与组件，一开始可以作为指引你的路标。

Cocoa API 大部分都是由 Objective-C 编写的，Cocoa 本身所包含的大多数也是 Objective-C 类，这些类都继承自根类 NSObject。在进行 iOS 编程时，你主要会使用内建的 Cocoa 类。Objective-C 类相当于 Swift 类，并且也兼容于 Swift 类，不过 Swift 的另外两种对象类型（结构体与枚举）在 Objective-C 中却没有对应之物。Swift 中声明的结构体与枚举是无法从 Swift 桥接到 Objective-C 的。幸好，一些最为重要的原生 Swift 对象类型可以桥接到 Cocoa 类（参见附录 A 了解关于 Objective-C 语言以及如何实现 Swift 与 Objective-C 通信的更多信息）。

本章将会介绍 Cocoa 的类结构，探讨 Cocoa 在概念上是如何根据底层的 Objective-C 特性进行组织的，然后再来介绍最为常见的一些 Cocoa 辅助类，最后介绍 Cocoa 根类及其特性，这些特性会被所有的 Cocoa 类所继承。

## 10.1 子类化

Cocoa 提供了大量的对象，这些对象知道如何以你所期望的方式运作。比如，UIButton 知道如何绘制自身，当用户轻拍时该如何响应；UITextField 知道如何显示可编辑的文本，如何弹出键盘，以及如何接收键盘输入。

通常，Cocoa 所提供的对象的默认行为与外观可能并不符合你的要求，你需要对其进行

定制。不过，这并不表示你需要子类化！ Cocoa 类提供了很多方法供你调用，提供了很多属性供你设置，这都是用于自定义实例的，你首先应该使用它们。你应该查阅 Cocoa 类的文档来了解实例是否已经满足了你的要求。比如，UIButton 的类文档表明你可以设置按钮的文本、文本颜色、内部图片、背景图片，以及其他很多特性与行为，无须子类化！

然而，有时设置属性和调用方法并不足以按照你所期望的方式来定制实例。在这种情况下，Cocoa 提供了一些内部方法，这些方法在实例完成某些事情时会得到调用，你可以通过子类化和重写（参见第 4 章）来定制其行为。你没法获得任何 Cocoa 内建类的源代码，但依然可以对其进行子类化，创建新的类，除了你所进行的修改，其行为非常类似于内建类。

某些 Cocoa Touch 类总是需要子类化。比如，没有子类化的单纯的 UIViewController 是非常少见的，没有 UIViewController 子类的 iOS 应用基本上是不可用的。

另一个例子就是 UIView。Cocoa Touch 有很多内建的 UIView 子类，它们会按照需要运作并绘制自身（如 UIButton、UITextField 等），你很少需要对其进行子类化。另一方面，你可能会创建自己的 UIView 子类，其作用是以全新方式绘制自身。实际上绘制的并非 UIView；在 UIView 需要绘制时，其 `drawRect:` 方法会得到调用，这样视图就可以绘制自身了。因此，以完全自定义的方式绘制 UIView 就需要子类化 UIView 并在子类中实现 `drawRect:`。正如其文档中所述“绘制视图内容的子类应该重写该方法并实现自己的绘制代码”。文档说你需要子类化 UIView，这样才能完全以自己的方式绘制内容。

比如，假设我们想让窗口包含一个水平线。Cocoa 中并没有提供水平线接口部件，这样我们就需要创建自己的了，即将自身绘制为一条水平线的 UIView。现在就来试一下：

1. 在 Empty Window 示例项目中，选择 File → New → File 并指定 iOS → Source → Cocoa Touch Class，让其成为 UIView 的子类，将其命名为 MyHorizLine。Xcode 会创建 *MyHorizLine.swift*。请确保它是应用目标的一部分。
2. 在 *MyHorizLine.swift* 中，将类声明的内容替换为如以下代码（不再解释了）：

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder:aDecoder)
    self.backgroundColor = UIColor.clearColor()
}
override func drawRect(rect: CGRect) {
    let c = UIGraphicsGetCurrentContext()!
    CGContextMoveToPoint(c, 0, 0)
    CGContextAddLineToPoint(c, self.bounds.size.width, 0)
    CGContextStrokePath(c)
}
```

3. 编辑故事板。在对象库中找到 UIView（叫作“View”），然后将其拖曳到画布的 View 对象上。你可以将其高度压缩一些。
4. 选中刚才拖曳到画布上的 UIView，使用身份查看器将其类修改为 MyHorizLine。

构建并在模拟器中运行应用。你会看到一条水平线出现在 nib 中 MyHorizLine 实例顶部的位置处。该视图将自身绘制成了一条水平线，因为我们对其子类化想要这么做。

在该示例中，我们从一个没有绘制功能的 UIView 开始。这正是我们无须调用 super 的原因所在；UIView 中 `drawRect:` 的默认实现什么都不做。但你还可以子类化内建的 UIView 子类以修改其绘制自身的方式。比如，UILabel 的文档说该类中有两个方法用于实现该目的。`drawTextInRect:` 与 `textRectForBounds:limitedToNumberOfLines:` 都显式表明：“该方法只应该由想要修改标签绘制方式的子类所重写。”这表明这些方法会在标签绘制自身时被 Cocoa 自动调用；这样，我们就可以子类化 UILabel 并在我们的子类中实现这些方法来修改特定类型的标签绘制自身的方式。

下面这个示例来自于我自己的应用，我子类化了 UILabel，创建了一个标签，它通过重写 `drawTextInRect:` 来绘制自己的矩形边框并使其内容从边框中嵌入。文档中说道：“在重写的方法中，你可以进一步配置当前上下文，然后调用 `super` 完成实际的文本绘制工作。”下面来试一下：

1. 在 Empty Window 项目中，创建一个新的类文件，它是 UILabel 的子类；将该类命名为 MyBoundedLabel。
2. 在 *MyBoundedLabel.swift* 中，将如下代码插入类声明体中：

```
override func drawTextInRect(rect: CGRect) {
    let context = UIGraphicsGetCurrentContext()!
    CGContextStrokeRect(context, CGRectInset(self.bounds, 1.0, 1.0))
    super.drawTextInRect(CGRectInset(rect, 5.0, 5.0))
}
```

3. 编辑故事板，向界面添加一个 UILabel，在身份查看器中将其类修改为 MyBoundedLabel。

构建并运行应用，你会看到矩形是如何绘制的以及标签的文本是如何插入其中的。

说来也奇怪（如果使用过其他面向对象应用框架，那么你就会感到奇怪），在你的代码与 Cocoa 的交互过程中，子类化却是使用较少的一种方式。知道或是确定何时该子类化有点棘手，但通常的原则是如果没有要求，那么你就不应该使用子类化。大多数内建的 Cocoa Touch 类都不需要子类化（一些类的文档明确表示不允许子类化）。

子类化在 Cocoa 中很少见的一个原因是很多内建类都将委托（参见第 11 章）作为定制

实例行为的首选方式。比如，你不会子类化 UIApplication（单例共享应用实例的类）仅仅为了在应用加载完毕后做出响应，因为委托机制提供了解决方案（`application:didFinishLaunchingWithOptions:`）。这正是模板会创建 AppDelegate 类的原因所在，它不是 UIApplication 的子类，而是使用了 UIApplicationDelegate 协议。

另外，如果需要对应用基础的事件处理行为进行某些比较复杂的定制化工作，那么你可以子类化 UIApplication 来重写 `sendEvent:`。文档专门有一节“Subclassing Notes”用来告诉你，这么做“非常少见”。（参见第 6 章关于如何确保 UIApplication 子类在应用启动时进行实例化以了解详情。）

# 10.2 类别与扩展

类别是 Objective-C 的一个语言特性，你可以通过它探究现有的类并注入额外的方法。类别对应于 Swift 的扩展（参见第 4 章）。借助 Swift 扩展，你可以将类或实例方法添加到 Cocoa 类中；Swift 头文件大量使用了扩展，既用于组织 Swift 自己的对象类型，也用于修改 Cocoa 类。与之相同，Cocoa 使用类别来组织自己的类。

Objective-C 的类别有名字，你会在头文件、文档中看到对这些名字的引用。不过，名字本身是没什么意义的，因此不用考虑太多。

## 10.2.1 Swift 如何使用扩展

查看主 Swift 头文件，你会看到很多原生对象类型声明都包含了一个初始声明，后跟一系列扩展。比如，在声明了泛型结构体 `Array<Element>` 后，Array 结构体头会继续声明不少于 7 个扩展。其中有些扩展增加了协议的使用；不过大多数并没有。它们都会向 Array 添加属性或方法声明；这正是扩展的意义所在。

扩展的功能性不是最重要的；头文件本可以在 Array 结构体的声明中加入所有这些属性与方法。相反，它将这些内容分散到了多个扩展中。扩展用于将相关的功能聚合到一起，组织对象类型的成员，从而使得开发者能够更容易地理解。

在 Swift Core Graphics 头文件中，一切都是扩展。Swift 在这里适配了其他地方定义的类型，将 Swift 数字类型用于 Core Graphics 和 CGFloat 数字类型，将 Cocoa 结构体如 CGPoint 与 CGRect 用作 Swift 对象类型。特别地，它向 CGRect 提供了多个附加属性、初始化器与方法，这样就可以直接按照 Swift 结构体与之交互，而不必调用 Cocoa Core Graphics C 辅助函数来操纵 CGRect 了。

## 10.2.2 你应该如何使用扩展

Swift 允许编写全局函数，这么做也没什么错的。不过，为了面向对象的封装，你常常需要编写作为已有对象类型一部分的函数。最简单，也是最有效的方式就是通过扩展将这种函数作为方法注入已有的对象类型中。如果仅仅添加一两个方法就使用子类化显得太过于笨重了；此外，这么做也没什么太大的好处。（另外，扩展可用于 Swift 全部 3 种对象类型，但我们却无法子类化 Swift 枚举和结构体。）

比如，假设想要向 Cocoa 的 UIView 类中添加一个方法。你可以子类化 UIView 并声明自己的方法，不过这样会使方法只位于你的 UIView 子类以及该子类的子类中：它不会出现在 UIButton、UILabel 及其他内建的 UIView 子类中，这是因为它们都是 UIView 的子类，而不是你定义的子类的子类，你也无法改变这一点！另外，扩展可以漂亮地解决这个问题：将方法注入到 UIView 中，那么它会被所有内建的 UIView 子类所继承。

在 Swift 2.0 中，你可以通过协议扩展以一种有选择但却统一的方式将功能注入类中。假设我需要一个 UIButton 和一个 UIBarButtonItem（它们不是 UIView，但却拥有类似于按钮的行为）来共享某个方法。我可以声明一个协议，它拥有一个方法，同时在协议扩展中实现该方法，然后通过扩展让 UIButton 和 UIBarButtonItem 使用该协议，因此就会拥有该方法：

```
protocol ButtonLike {
    func behaveInButtonLikeWay()
}
extension ButtonLike {
    func behaveInButtonLikeWay() {
        // ...
    }
}
extension UIButton : ButtonLike {}
extension UIBarButtonItem : ButtonLike {}
```

第 4 章介绍了几个扩展示例，这些示例都来自于我所编写的 iOS 程序（参见 4.10 节）。此外，我常常会与 Swift 头文件相同的方式来使用扩展，将单个对象类型的代码组织到多个扩展中，目的在于表述清晰。

## 10.2.3 Cocoa 如何使用类别

Cocoa 将类别作为一种组织工具，这一点与 Swift 扩展很相似。类的声明常常会按照功能拆分为多个类别，这些类别位于单独的头文件中。

NSString 就是个很好的示例。它定义在 Foundation 框架中，基本的方法声明在 *NSString.h* 中。除了初始化器，我们发现 NSString 本身只有两个方法，分别是 `length` 与 `characterAtIndex:`，因为这两个方法是字符串所需的最基础的功能。

额外的 NSString 方法（创建字符串、处理字符串编码、分割字符串、字符串搜索等）都分布在各个类别当中。它们都位于 Swift 转换后的扩展当中。比如，在 String 类本身的声明之后，我们发现 Swift 的转换中有如下代码：

```
extension NSString {
    func substringFromIndex(from: Int) -> String
    func substringToIndex(to: Int) -> String
    // ...
}
```

这实际上是 Swift 对如下 Objective-C 代码的转换结果：

```
@interface NSString (NSStringExtensionMethods)
- (NSString *)substringFromIndex:(NSUInteger)from;
- (NSString *)substringToIndex:(NSUInteger)to;
// ...
@end
```

符号（关键字 `@interface`，后跟类名，再后跟一对圆括号，里面是另一个名字）是 Objective-C 类别。

此外，虽然有一些 Cocoa NSString 类别出现在相同的文件 *NSString.h* 中，不过还有很多位于其他文件中。比如：

- 字符串可能作为文件路径名，因此我们在 *NSPathUtilities.h* 中发现了一个 NSString 类别，其方法和属性如 `pathComponents` 等用于将路径名字符串划分为各个组成部分。
- 在 *NSURL.h*（主要用于声明 NSURL 类及其类别）中还有另一个 NSString 类别，它声明了用于处理 URL 字符串中百分号转义的方法，如 `stringByAddingPercentEscapesUsingEncoding`。
- 在另一个完全不同的框架（UIKit）中，*NSStringDrawing.h* 还添加了两个 NSString 类别，其方法 `drawAtPoint:` 等用于在图形上下文中绘制字符串。

这种组织方式对于程序员并没有那么重要，因为 NSString 就是个 NSString，无论它如何获得其方法都是如此。不过在查阅文档时就很重要了！声明在 *NSString.h*、*NSPathUtilities.h* 与 *NSURL.h* 中的 NSString 方法文档都集中在 NSString 类文档页面中。不过，声明在 *NSStringDrawing.h* 中的 NSString 方法却不是这样的；相反，它们位于单独的文档 NSString UIKit Additions Reference 中（推测一下，这是因为它们来自于不同的框架）。这样，字符串绘制方法就会难以找到，特别是 NSString 类文档页面没有指向其他文档的链接。我认为这是 Cocoa 文档结构的一个败笔。

# 10.3 协议

Objective-C 拥有协议，这相当于 Swift 的协议（参见第 4 章）。由于类是 Objective-C 唯一的对象类型，因此所有 Objective-C 协议都会被 Swift 看作类协议。与之相反，标记为 `@objc` 的 Swift 协议（隐式表示类协议）可以被 Objective-C 所看到。Cocoa 大量使用了协议。

比如，下面来看看 Cocoa 对象是如何复制的。有些对象可以复制，有些则不行。这与对象的类继承没有关系。我们需要一个统一的方法，可复制的任何对象都会响应这个方法。因此，Cocoa 定义了一个名为 NSCopying 的协议，它只声明了一个必要的方法 `copyWithZone:`。下面是 Objective-C 中 NSCopying 协议的声明（在 *NSObject.h* 中）：

```
@protocol NSCopying
- (id)copyWithZone:(nullable NSZone *)zone;
@end
```

转换为 Swift 如以下代码所示：

```
protocol NSCopying {
    func copyWithZone(zone: NSZone) -> AnyObject
}
```

不过，*NSObject.h* 中的 NSCopying 协议声明并没有表示 NSObject 遵循着 NSCopying。实际上，NSObject 并不遵循 NSCopying！如下代码无法编译通过：

```
let obj = NSObject().copyWithZone(nil) // compile error
```

不过下面的代码可以编译通过，因为 NSString 遵循了 NSCopying（字面值“howdy”会被隐式桥接为 NSString）：

```
let s = "hello".copyWithZone(nil)
```

典型的Cocoa模式是“只要实现了如下方法，那么任何类的实例都可以”。这显然是个协议。比如，考虑一下协议是如何与表视图（UITableView）产生联系的。表视图从数据源获取数据。出于这个目的，UITableView 声明了一个 `dataSource` 属性，如下所示：

```
@property (nonatomic, weak, nullable) id <UITableViewDataSource> dataSource;
```

转换为 Swift 如以下代码所示：

```
weak var dataSource: UITableViewDataSource?
```

UITableViewDataSource 是个协议。表视图说的是“我不管数据源属于哪个类，但不管哪个，它都应该遵循 UITableViewDataSource 协议”。这形成了一种承诺，数据源至少要

实现必需的实例方法 tableView:numberOfRowsInSection: 与 tableView:cellForRowAtIndexPath:，在需要知道显示什么数据时，表视图将会调用它们。在使用 UITableView 并且想要提供给它一个数据源对象时，该对象的类将会使用 UITableViewDataSource，并实现所需的方法；否则，代码将无法编译通过：

```
let obj = NSObject()
let tv = UITableView()
tv.dataSource = obj // compile error
```

毫无疑问，协议在 Cocoa 中最常使用的地方就是与委托模式有关了。第 11 章将会对此进行详尽的介绍，不过我们先来看看 Empty Window 项目中的一个示例：项目模板所提供的 AppDelegate 类的声明如下所示：

```
class AppDelegate: UIResponder, UIApplicationDelegate { // ...
```

AppDelegate 的主要目的是作为共享的应用委托。共享的应用对象是一个 UIApplication，而 UIApplication 的 delegate 属性的声明如下所示：

```
unowned(unsafe) var delegate: UIApplicationDelegate?
```

（第 12 章将会介绍 unsafe 修饰符。）UIApplicationDelegate 类型是个协议。共享的 UIApplication 对象正是通过它知道其委托可以接收如 application:didFinishLaunchingWithOptions: 这样的消息。因此，AppDelegate 类通过显式使用 UIApplicationDelegate 协议来表明其角色。

Cocoa 协议拥有自己的文档页面。当 UIApplication 类文档告诉你 delegate 属性的类型为 UIApplicationDelegate 时，它实际上是隐式告诉你如果想要了解 UIApplication 的委托可以接收什么消息，那就需要查看 UIApplicationDelegate 协议文档。你在 UIApplication 类文档页面上找不到刚才提到的 application:didFinishLaunchingWithOptions:！它的介绍位于 UIApplicationDelegate 协议的文档页面中。

当类使用了协议时，这种信息分离会让你感到困惑。当类文档上说类遵循了某个协议时，请不要忘记查看协议的文档！后者可能包含了关于类行为的重要信息。要想了解可以向某个对象发送什么消息，你需要沿着父类继承链向上查找；还需要查看该对象的类（或父类）所遵循的协议。比如，正如第 8 章所介绍的，只查看 UIViewController 类文档页面是不可能发现 UIViewController 有一个 viewWillTransitionToSize:withTransitionCoordinator: 事件的：你需要查看 UIViewController 所使用的协议 UIContentContainer 的文档。

### 10.3.1 非正式协议

你可能会在网上或文档中遇到非正式协议的说法。非正式协议实际上并不是协议；它只不过是向编译器提供了一个方法签名，这样编译器就允许发送消息而不会发出警告了。

有两种互补的方式可以实现非正式协议。一是在 NSObject 上定义一个类别；这样任何对象都可以接收类别中的消息了。二是定义一个协议，但却不必遵循它；相反，协议中的消息只会发送给类型为 id（AnyObject）的对象，这样编译器就不会发出警告了。

这些技术在协议可以声明可选方法前得到了广泛的应用；但现在这么做就完全没必要了，而且这些技术还存在一定的风险。在 iOS 9 中，只有极少的非正式协议还得以留存。比如说，NSKeyValueCoding（本章后面将会介绍）是个非正式协议；你还会在文档和其他地方看到术语 NSKeyValueCoding，不过实际上并没有该类型；它是 NSObject 上的一个类别。

### 10.3.2 可选方法

Objective-C 协议以及标记为 `@objc` 的 Swift 协议可以拥有可选成员（参见 4.8.4 节）。问题在于：在实际开发中，可选方法是如何使用的？我们知道，如果向对象发送消息，但对象无法处理该消息，那就会抛出异常，应用有可能崩溃。不过方法声明是个契约，表示对象可以处理该消息。如果声明一个可能会，也可能不会实现的方法，那就破坏了契约，这么做是否会造成应用崩溃呢？

答案就是 Objective-C 是一门既动态又内省的语言。它可以询问对象是否能够处理消息，而无须实际发送消息。这里的关键方法是 NSObject 的 `respondsToSelector:`，它接收一个选择器参数并返回 Bool（选择器本质上是个方法名，不过其表示方式独立于任何方法调用；参见附录 A）。因此，我们可以只在安全的情况下才向对象发送消息。

在 Swift 中演示 `respondsToSelector:` 有点棘手，因为让 Swift 抛弃严格的类型检查而允许我们向对象发送可能无法响应的消息是很困难的事情。在这个杜撰的示例中，我首先在顶层定义两个类：一个继承自 NSObject，否则无法向其发送 `respondsToSelector:`；另一个声明会根据条件发送的消息：

```
class MyClass : NSObject {
}
class MyOtherClass {
    @objc func woohoo() {}
}
```

现在可以这么做：

```
let mc = MyClass()
```

```
if mc.respondsToSelector("woohoo") {
    (mc as AnyObject).woohoo()
}
```

注意到从 mc 到 AnyObject 的类型转换。这会导致 Swift 放弃其严格的类型检查；现在可以向该对象发送 Swift 知道的任何消息了，就好像 Objective-C 的内省机制一样，这正是将 woohoo 声明标记为 @objc 的原因所在。如你所知，Swift 提供了一种简写来根据条件发送消息，即将一个问号放到消息名的后面：

```
let mc = MyClass()
(mc as AnyObject).woohoo?()
```

在背后，这两种方式是完全一样的；后者是前者的语法糖。对于问号来说，Swift 会调用 respondsToSelector:，如果无法响应该选择器，那就不会向该对象发送 woohoo 消息。

这也说明了可选协议成员的工作方式。Swift 对待可选协议成员的方式与 AnyObject 成员一样，这并非巧合。下面是第 4 章曾经介绍过的一个示例：

```
@objc protocol Flier {
    optional var song : String {get}
    optional func sing()
}
```

在类型为 Flier 的对象上调用 sing?() 时，背后会调用 respondsToSelector:，用于确定这个调用是否是安全的。

你不应该随意发送消息，也不要在发送任何旧的消息前显式调用 respondsToSelector:，因为除了可选方法，这么做是毫无必要的，还会增加处理时间。不过 Cocoa 实际上会调用对象的 respondsToSelector:。为了证实这一点，在 Empty Window 项目的 AppDelegate 中实现 respondsToSelector:，并将日志打印出来：

```
override func respondsToSelector(aSelector: Selector) -> Bool {
    print(aSelector)
    return super.respondsToSelector(aSelector)
}
```

当 Empty Window 应用启动后，我的电脑上的输出如下所示（省略了私有方法与对同一个方法多次调用的输出）：

```
application:handleOpenURL:
application:openURL:sourceApplication:annotation:
application:openURL:options:
applicationDidReceiveMemoryWarning:
applicationWillTerminate:
applicationSignificantTimeChange:
application:willChangeStatusBarOrientation:duration:
application:didChangeStatusBarOrientation:
```

```
application:willChangeStatusBarFrame:
application:didChangeStatusBarFrame:
application:deviceAccelerated:
application:deviceChangedOrientation:
applicationDidBecomeActive:
applicationWillResignActive:
applicationDidEnterBackground:
applicationWillEnterForeground:
application:didResumeWithOptions:
application:handleWatchKitExtensionRequest:reply:
application:shouldSaveApplicationState:
application:supportedInterfaceOrientationsForWindow:
application:performFetchWithCompletionHandler:
application:didReceiveRemoteNotification:fetchCompletionHandler:
application:willFinishLaunchingWithOptions:
application:didFinishLaunchingWithOptions:
```

Cocoa 会检查哪个可选的 UIApplicationDelegate 协议方法（包括一些文档中没有提及的方法）被 AppDelegate 实例实现了，因为它是 UIApplication 对象的委托，遵循 UIApplicationDelegate 协议，它显式表明可以响应所有这些消息。整个委托模式（第 11 章将会介绍）都依赖于该项技术。注意到 Cocoa 所遵循的策略：当首次遇到目标对象时，它会检查所有的可选协议方法一次，并可能会将结果存储起来；这样，应用的速度会受到这个一次性的初始 `respondsToSelector:` 调用的影响，不过现在 Cocoa 已经知道了答案，所以后面就不会再对相同的对象进行同样的检查了。

# 10.4 Foundation 类精讲

Cocoa 的 Foundation 类提供了一些基本数据类型与辅助方法，它们构成了使用 Cocoa 的基础。显然，我无法将其一一列举，更不必说完整介绍它们了，但我可以介绍一些你在编写最简单的 iOS 程序前所需要了解的内容。要想了解更多信息，请从 *Foundation Framework Reference* 的 Foundation 类列表开始。

## 10.4.1 常用的结构体与常量

NSRange 是个 C 结构体（参见附录 A），它对于处理我将要介绍的一些类是非常重要的。其组成都是整数，`location` 与 `length`。比如，`location` 为 `1` 的 NSRange 表示从第 2 个元素开始（因为元素计数总是基于 0 的），如果 `length` 为 `2`，那就表示这个元素与下一个。

Cocoa 提供了各种便捷函数来处理 NSRange；比如，你可以调用 `NSMakeRange` 通过两个整数创建 NSRange（注意到名字 `NSMakeRange` 向后类比于 `CGPointMake` 与 `CGRectMake` 等名字）。Swift 通过将 NSRange 桥接为 Swift 结构体来解决遇到的问题。你可以对 NSRange 与 Swift Range（端点为 Int）进行转换：Swift 为 NSRange 增加了一个初始化器，它接收一个 Swift Range，另外还有一个 `toRange` 方法。

NSNotFound 是个整型常量，表示找不到所请求的元素。NSNotFound 真正的数值是什么并不重要；只要与 NSNotFound 本身进行比较即可，从而判断结果是否有意义。比如，如果获取某个对象在 NSArray 中的索引，但该对象不存在，那么结果就是个 NSNotFound：

```
let arr = ["hey"] as NSArray
let ix = arr.indexOfObject("ho")
if ix == NSNotFound {
    print("it wasn't found")
}
```

Cocoa 为何要以这种方式依赖于拥有特殊含义的整型值呢？这是因为它只能这么做。表示对象不存在的结果不能为 0，因为 0 表示数组的第一个元素。结果也不能是 -1，因为 NSArray 索引值总是正数。它也不能为 nil，因为当需要返回一个整数时，Objective-C 不能返回 nil（即便可以，那它也会被看作 0 的另一种表示方式）。相反，Swift 的 indexOf 方法会返回一个包装了 Int 的 Optional，这样就可以返回 nil 来表示目标对象没有找到了。

如果搜索返回一个范围，但并没有找到任何结果，那么生成的 NSRange 的 location 就为 NSNotFound。第 3 章曾经介绍过，Swift 有时会自动帮你做一些聪明且自动化的桥接工作，从而无需再与 NSNotFound 进行比较了。典型示例就是 NSString 的 rangeOfString: 方法。在 Cocoa 的定义中，它会返回一个 NSRange；Swift 则将其改造为返回一个包装了 Swift Range（String.Index）的 Optional，如果 NSRange 的 location 为 NSNotFound，那就会返回 nil：

```
let s = "hello"
let r = s.rangeOfString("ha") // nil; an Optional wrapping a Swift Range
```

如果你需要的是个 Swift Range，那就正合适，适合于进一步切割 Swift String；但如果需要的是个 NSRange，想要返回给 Cocoa，那就需要将原来的 Swift String 转换为 NSString，这样结果依然是 Cocoa 类：

```
let s = " hello" as NSString
let r = s.rangeOfString("ha") // an NSRange
if r.location == NSNotFound {
    print("it wasn't found")
}
```

## 10.4.2 NSString 及相关类

NSString 是字符串的 Cocoa 对象版本。NSString 与 Swift String 会彼此桥接，你常常会不自觉地在这两者间切换，需要 NSString 时就将 Swift String 传递给 Cocoa，在 Swift String 上调用 Cocoa NSString 方法，诸如此类。比如：

```
let s = "hello"
let s2 = s.capitalizedString
```

在上述代码中，s 是个 Swift String，s2 也是个 Swift String，但 capitalizedString 属性实际上是 Cocoa 的。在执行上述代码时，Swift String 会被桥接到 NSString 并传递给 Cocoa，Cocoa 则会处理它并得到大写的字符串；这个大写的字符串是个 NSString，不过它可以桥接到 Swift String。你基本意识不到桥接过程；capitalizedString 就像是原生 String 的属性一样，不过它并不是，可以在没有导入 Foundation 的环境下使用它来证明这一点（其实是不行的）。

在某些情况下，你需要进行显式类型转换来桥接。Swift 可能会在桥接时失败；比如，如果 s 是个 Swift 字符串，那你就不能直接对其调用 stringByAppendingPathExtension:：

```
let s = "MyFile"
let s2 = s.stringByAppendingPathExtension("txt") // compile error
```

你需要显式将其转换为 NSString：

```
let s2 = (s as NSString).stringByAppendingPathExtension("txt")
```

此外，对字符串使用索引时会出现问题。比如：

```
let s = "hello"
let s2 = s.substringToIndex(4) // compile error
```

问题在于桥接是你自己做的。Swift 并不会阻止你在 Swift String 上调用 substring-ToIndex: 方法，不过索引值必须是个 String.Index，这很难构建（参见第 3 章）：

```
let s2 = s.substringToIndex(s.startIndex.advancedBy(4))
```

如果不想这么做，那就需要提前将 String 强制类型转换为 NSString；现在处理的都是 Cocoa 了，字符串索引是整型值：

```
let s2 = (s as NSString).substringToIndex(4)
```

不过，正如第 3 章所介绍的那样，这两个调用实际上并不是等价的：其结果是不同的！原因在于从根本上来说，String 与 NSString 在字符串的元素构成上拥有完全不同的表示方式。String 会将元素解析为字符，这意味着它会遍历字符串，将任何可合并的代码点聚合起来；NSString 的行为就好像它是个 UTF16 代码点的数组。从 Swift 的角度来看，String.Index 的增加都对应于真正的字符，不过通过索引或范围访问却需要遍历字符串；从 Cocoa 的角度来看，通过索引或范围访问是非常快的，不过可能无法对应上字符边界（参见 Apple *String Programming Guide* 的“Characters and Grapheme Clusters”一章）。

Swift String 与 Cocoa NSString 之间的另一个主要差别在于 NSString 是不可变的。这意味着对于 NSString，你可以做到根据一个字符串来获得另一个新的字符串（就像 `capitalizedString` 与 `substringToIndex:` 所做的那样），不过不能就地修改字符串。要想做到这一点，你需要另一个类 NSMutableString，它是 NSString 的子类。NSMutableString 有很多有用的方法，你可以充分利用这些方法；不过 Swift String 并没有桥接到 NSMutableString，因此无法仅通过类型转换将 String 转换为 NSMutableString。要想得到 NSMutableString，你需要创建一个。最简单的方式是使用 NSMutableString 的初始化器 `init(string:)`，它接收一个 NSString，这意味着你可以传递一个 Swift String 进去。这样，只需一步就可以将 NSMutableString 转换为 Swift String。

```
let s = "hello"
let ms = NSMutableString(string:s)
ms.deleteCharactersInRange(NSMakeRange(ms.length-1,1))
let s2 = (ms as String) + "ion" // now s2 is a Swift String
```

正如第 3 章所介绍的，原生 Swift String 方法数量并不多。所有的字符串处理能力都依赖于桥接的另一方 Cocoa。因此，你会经常通过桥接完成一些功能！这并不是只针对 NSString 与 NSMutableString 类的。很多其他的常见类都与之相关。

比如，假设要查找某个字符串中的子字符串。最佳做法都来自于 Cocoa：

- 可以通过各种 `rangeOfString:...` 方法搜索 NSString，同时还可以使用大量的选项，比如，忽略临界值、忽略大小写、从尾部开始，以及待搜索的子字符串一定要位于被搜索字符串的起始或结束位置处。
- 也许不太确定要搜索的是什么：你需要描述出其结构。可以通过 NSScanner 遍历字符串，查找满足某些条件的子字符串；比如，借助 NSScanner（以及 NSCharacterSet），你可以跳过以数字开头的子字符串并提取出数字。
- 通过指定选项 `.RegularExpressionSearch`，你可以使用正则表达式搜索。正则表达式也是通过单独一个类 NSRegularExpression 得到支持的，它会使用 NSTextCheckingResult 描述匹配结果。
- 更加复杂的自动化文本分析是通过其他一些类得到支持的，比如，NSDataDetector，它是 NSRegularExpression 的子类，可以迅速找到某些类型的字符串表达式，如 URL 或电话号码；还有 NSLinguisticTagger，它会根据文法词性规则分析文本。

在该示例中，我们要将所有“hello”替换为“heaven”。我们并不希望见到子字符串“hell”就替换，比如，“hello”就不应该替换。搜索需要智能一些，知道单词的边界是什么。

这看起来是正则表达式的事情。Swift 并没有提供正则表达式支持，因此一切都要通过 Cocoa 来完成：

```
let s = NSMutableString(string:"hello world, go to hell")
let r = try! NSRegularExpression(
    pattern: "\\bhell\\b",
    options: .CaseInsensitive)
r.replaceMatchesInString(
    s, options: [], range: NSMakeRange(0,s.length),
    withTemplate: "heaven")
// s is "hello world, go to heaven"
```

NSString 还提供了一些便捷的功能用以处理文件路径字符串，常用于 NSURL，这是另一个值得探究的 Foundation 类。此外，NSString（就像本节介绍的其他类一样）提供了写到文件以及从文件读取的方法；可以通过 NSString 文件路径或 NSURL 来指定文件。

NSString 并没有字体与大小等信息。显示字符串的界面对象（如 UILabel）有一个类型为 UIFont 的 font 属性；不过，它只用于确定该组件上所显示的字符串的字体与大小。如果需要带样式的文本（不同的文本有不同的样式属性，如大小、字体及颜色等），那么可以使用 NSAttributedString 及其支持类：NSMutableAttributedString、NSParagraphStyle 与 NSMutableParagraphStyle。你可以通过它们从各个方面为文本和段落增加样式。显示文本的内建界面对象可以显示 NSAttributedString。

可以通过 NSString（参见 *String UIKit Additions Reference*）及 NSAttributedString（参见 *NSAttributedString UIKit Additions Reference*）上的 NSStringDrawing 类别所提供的方法在图形上下文中绘制字符串。

## 10.4.3 NSDate 及相关类

NSDate 是个日期与时间，内部表示为从某个参考日期开始所经过的秒数（NSTimeInterval）。调用 NSDate 的初始化器 `init()`（即 `NSDate()`）会生成一个代表当前日期与时间的日期对象。很多日期操作还会用到 NSDateComponents，NSDate 与 NSDateComponents 之间的转换需要传递一个 NSCalendar。下述示例展示了如何根据日历值构建一个日期：

```
let greg = NSCalendar(calendarIdentifier:NSCalendarIdentifierGregorian)!
let comp = NSDateComponents()
comp.year = 2016
comp.month = 8
comp.day = 10
comp.hour = 15
let d = greg.dateFromComponents(comp) // Optional wrapping NSDate
```

与之类似，NSDateComponents 提供了进行日期计算的正确方式。如下示例展示了如何为给定的日期增加一个月：

```
let d = NSDate() // or whatever
let comp = NSDateComponents()
comp.month = 1
let greg = NSCalendar(calendarIdentifier:NSCalendarIdentifierGregorian)!
let d2 = greg.dateByAddingComponents(comp, toDate:d, options:[])
```

你可能还会考虑以字符串表示的日期。如果不对日期的字符串表示进行显式的处理，那么其字符串表示格式会让你吃惊的。比如，如果 `print` 一个 NSDate，那么它会以 GMT 时区的形式表示日期，如果不在这儿住，那么这个结果会让你感到困惑。一个简单的解决办法就是调用 `descriptionWithLocale:`；它会考虑到用户当前的时区、语言、区域格式以及日历设置等：

```
print(d)
// 2016-08-10 22:00:00 +0000
print(d.descriptionWithLocale(NSLocale.currentLocale()))
// Wednesday, August 10, 2016 at 3:00:00 PM Pacific Daylight Time
```

请使用 NSDateFormatter 来创建并解析日期字符串，它使用了类似于 NSLog（以及 NSString 的 `stringWithFormat:`）的格式化字符串。在该示例中，我们完全使用了用户的区域设置，通过 `dateFormatFromTemplate:options:locale:` 与当前区域设置生成一个 NSDateFormatter。“模板”是个字符串，列出了将要使用的日期组件，不过其顺序、标点符号和语言则留给区域设置来处理：

```
let df = NSDateFormatter()
let format = NSDateFormatter.dateFormatFromTemplate(
    "dMMMMyyyyhmmaz", options:0, locale:NSLocale.currentLocale())
df.dateFormat = format
let s = df.stringFromDate(NSDate())
```

生成的日期会使用用户的时区和语言，并使用正确的语言规范。这涉及区域设置格式与语言的组合，它们是两个单独的设置。这样：

- 在我的设备上，结果是“July 16, 2015, 7:44 AM PDT.”。
- 如果将设备的区域设置修改为 France，那么结果就变成了“16 July 2015 7:44 AM GMT-7.”。
- 如果再将设备的语言修改为 French，那么结果又会变成“16 juillet 2015 7:44 AM UTC−7.”。

## 10.4.4 NSNumber

NSNumber 是个包装了数值的对象。被包装的值可以是任何标准的 Objective-C 数值类型（包括 BOOL，这是 Objective-C 中与 Swift Bool 的对应类型）。让 Swift 用户感到惊讶的是竟然还需要 NSNumber。不过，Objective-C 中的普通数字并不是对象（它是标量，参见附录 A），因此无法在需要对象的地方使用。这样，NSNumber 就解决了一个重要问题，可以将数字转换为对象，反之亦然。

Swift 会尽一切努力不让你直接使用 NSNumber。它通过两种不同方式桥接了 Swift 数值类型与 Objective-C：

- 如果需要普通的数字，那么 Swift 数字就会桥接到普通的数字（标量）。
- 如果需要对象，那么基本的数字类型的 Swift 数字就会桥接到 NSNumber。基本的数字类型有 Int、UIInt、Float、Double 以及 Bool，因为 NSNumber 能够包装 Objective-C BOOL。

看看下面这个示例：

```
let ud = NSUserDefaults.standardUserDefaults()
let i = 0
ud.setInteger(i, forKey: "Score") ❶
ud.setObject(i, forKey: "Score") ❷
```

后两行看起来很像，不过 Swift 对待 Int 值 i 的方式却是不同的：

❶ `setInteger:forKey:` 的第 1 个参数需要一个整型（标量），因此 Swift 会将 Int 结构体值 i 转换为普通的 Objective-C 数字。

❷ `setObject:forKey:` 的第 1 个参数需要一个对象，因此 Swift 会将 Int 结构体值 i 转换为 NSNumber。

自然，如果想要显式跨过这种桥接，那也是可行的。可以将 Swift 数字（基本的数字类型）强制转换为 NSNumber：

```
let n = 0 as NSNumber
```

要想更好地控制 NSNumber 所包装的数值类型，你可以调用 NSNumber 的初始化器：

```
let n = NSNumber(float:0)
```

从 Objective-C 回到 Swift，值一般会作为 AnyObject，你需要进行向下类型转换。NSNumber 拥有一些属性可根据数字类型访问被包装的值。回忆一下第 5 章的示例，它会从 NSNotification 的 `userInfo` 字典中将值提取出来并作为 NSNumber 返回：

```
if let prog = (n.userInfo?["progress"] as? NSNumber)?.doubleValue {
    self.progress = prog
}
```

NSNumber 还可以向下类型转换为 Swift 数值类型。因此，包装 NSNumber 的 AnyObject 也可以。这样，该示例可以改写成下面这样，不会显式用到 NSNumber：

```
if let prog = n.userInfo?["progress"] as? Double {
    self.progress = prog
}
```

在第 2 个版本中，Swift 实际上在背后做的是与第 1 个示例相同的事情，将 AnyObject 当作 NSNumber，并通过 doubleValue 属性提取出被包装的数字。

NSNumber 对象只是一个包装器而已。无法将其直接用在数字计算上；它并非数字，而是包装了一个数字。不管怎样，如果需要数字，那就需要从 NSNumber 中提取。

另外，NSNumber 的子类 NSDecimalNumber 可用于计算，这多亏了大量的数学计算方法：

```
let dec1 = NSDecimalNumber(float: 4.0)
let dec2 = NSDecimalNumber(float: 5.0)
let sum = dec1.decimalNumberByAdding(dec2) // 9.0
```

NSDecimalNumber 在取整方面非常有用，因为它提供了一种便捷的方式来指定所需的取整方式。

NSDecimalNumber 底层是 NSDecimal 结构体（它是 NSDecimalNumber 的 decimalValue）。NSDecimal 拥有一些 C 函数，速度上要比 NSDecimalNumber 方法快很多。

## 10.4.5 NSValue

NSValue 是 NSNumber 的父类。它用于在需要对象的时候包装非数字的 C 值，比如，C 结构体。它所解决的问题类似于 NSNumber：Swift 结构体是个对象，不过 C 结构体不是，因此在 Objective-C 中，如果需要对象，那么使用结构体是行不通的。

可以通过 NSValue 上的 NSValueUIGeometryExtensions 类别所提供的便捷方法（参见 *NSValue UIKit Additions Reference*）轻松包装和展开 CGPoint、CGSize、CGRect、CGAffineTransform、UIEdgeInsets 与 UIOffset；还有其他一些类别可以轻松包装和展开 NSRange、CATransform3D、CMTime、CMTimeMapping、CMTimeRange、MKCoordinate 与 MKCoordinateSpan。一般不需要在 NSValue 中存储其他类型的 C 值，不过如果需要也是可以的。

Swift 并不会神奇地桥接这些 C 结构体类型与 NSValue。你需要显式对其进行管理，正

如使用 Objective-C 代码所做的那样。如下示例使用 Core Animation 实现界面上的按钮从一个位置到另一个位置的移动；按钮的起止位置都表示为 CGPoint，不过动画的 fromValue 与 toValue 必须是对象。CGPoint 并非 Objective-C 对象，因此需要将 CGPoint 值包装到 NSValue 对象中：

```
let ba = CABasicAnimation(keyPath:"position")
ba.duration = 10
ba.fromValue = NSValue(CGPoint:self.oldButtonCenter)
ba.toValue = NSValue(CGPoint:goal)
self.button.layer.addAnimation(ba, forKey:nil)
```

与之类似，可以在 Swift 中创建 CGPoint 的数组，这是因为 CGPoint 变成了一个 Swift 对象类型（Swift 结构体），而 Swift Array 可以持有任意类型的元素；不过不能将该数组传递给 Objective-C，因为 Objective-C NSArray 中的元素必须是对象，而 Objective-C 中的 CGPoint 并不是对象。这样，首先就需要将 CGPoints 包装到 NSValue 对象中。下面是另一个动画示例，我通过将 CGPoints 数组转换为 NSValues 数组来设置关键帧动画的 values 数组（NSArray）。

```
anim.values = [oldP,p1,p2,newP].map{NSValue(CGPoint:$0)}
```

## 10.4.6 NSData

NSData 是个字节序列；基本上，它是个缓存，占据了一块内存。它是不可变的；其可变版本是其子类 NSMutableData。

在实际开发中，NSData 主要用在如下两种情况当中：

- 从 Internet 上下载数据。比如，NSURLConnection 与 NSURLSession 会将从 Internet 上接收到的东西当作 NSData。你可以根据需要将其转换为字符串，并指定正确的编码。
- 将对象存储为文件或用户首选项（NSUserDefaults）。比如，你无法直接将 UIColor 值存储为用户首选项。如果用户选择了某个颜色，你需要将其保存起来，那么你可以将 UIColor 转换为 NSData（使用 NSKeyedArchiver）并保存：

```
let ud = NSUserDefaults.standardUserDefaults()
let c = UIColor.blueColor()
let cdata = NSKeyedArchiver.archivedDataWithRootObject(c)
ud.setObject(cdata, forKey: "myColor")
```

## 10.4.7 相等与比较

在 Swift 中，如果对象类型使用了 Equatable 与 Comparable 协议，那么我们就可以针对

该对象类型重写相等与比较运算符。不过 Objective-C 运算符则不行，在 Objective-C 中，相等与比较运算符只能用于标量。

要想对两个对象进行“相等”判断（无论对于该对象类型来说相等意味着什么），Objective-C 类必须要实现 `isEqual:`，它继承自 NSObject。Swift 则会将 NSObject 看作 Equatable，并且允许使用 == 运算符，从而解决了各种问题，它会隐式将 == 运算符转换为 `isEqual:` 调用。这样，如果一个类实现了 `isEqual:`，那么我们就可以使用普通的 Swift 比较。比如：

```
let n1 = NSNumber(integer:1)
let n2 = NSNumber(integer:2)
let n3 = NSNumber(integer:3)
let ok = n2 == 2 // true ❶
let ok2 = n2 == NSNumber(integer:2) // true ❷
let ix = [n1,n2,n3].indexOf(2) // Optional wrapping 1 ❸
```

上述代码似乎做了 3 件不可能的事情：

❶ 我们直接比较了 Int 与 NSNumber，并且得到了正确的结果，就好像比较的是 Int 与 NSNumber 所包装的那个整数一样。

❷ 我们直接比较了两个 NSNumber 对象，并且得到了正确的结果，就好像比较的是这两个 NSNumber 对象所包装的整数一样。

❸ 我们将 NSNumber 数组看作 Equatables 数组，并调用了 `indexOf` 方法，最后成功找出“等于”那个实际值的 NSNumber 对象。

这种魔法分为两块：

- 数字被包装到了 NSNumber 对象中。
- == 运算符（背后也被 `indexOf` 方法所用）会被转换为 `isEqual:` 调用。

NSNumber 实现了 `isEqual:` 来比较两个 NSNumber 对象，这是通过比较所包装的数值来实现的；因此，相等比较可以正常使用。

如果 NSObject 子类没有实现 `isEqual:`，那么它会继承 NSObject 的实现，比较两个对象的相等性（就像 Swift 的 === 运算符一样）。比如，两个 Dog 对象可以通过 == 运算符进行比较，虽然 Dog 并未使用 Equatable 也是可以的，因为它们都继承自 NSObject，但 Dog 没有实现 `isEqual:`，因此 == 默认将会使用 NSObject 的相等比较：

```
class Dog : NSObject {
    var name : String
    init(_ name:String) {self.name = name}
}
```

```
let d1 = Dog("Fido")
let d2 = Dog("Fido")
let ok = d1 == d2 // false
```

很多实现了isEqual:的类还实现了更为具体和高效的测试。对于Objective-C，判断两个NSNumber对象是否相等（即包装了相同的数字）的常见做法是调用isEqualToNumber:。与之类似，NSString有isEqualToString:、NSDate有isEqualToDate:，诸如此类。不过，这些类还实现了isEqual:，因此我觉得最好的方式还是使用Swift == 运算符。

与之类似，在Objective-C中，提供排序比较方法是每个类的职责。标准方法是compare:，它会返回3个NSComparisonResult case之一：

.OrderedAscending
: 接收者小于参数。

.OrderedSame
: 接收者等于参数。

.OrderedDescending
: 接收者大于参数。

Swift比较运算符（<之类的）并不会神奇地调用compare:。你不能直接比较两个NSNumber值：

```
let n1 = NSNumber(integer:1)
let n2 = NSNumber(integer:2)
let ok = n1 < n2 // compile error
```

你常常需要自己调用compare:，就像在Objective-C中所做的那样：

```
let n1 = NSNumber(integer:1)
let n2 = NSNumber(integer:2)
let ok = n1.compare(n2) == .OrderedAscending // true
```

## 10.4.8 NSIndexSet

NSIndexSet表示不重复的数字集合；其目的在于表示出有序的集合元素数字，如NSArray。这样，比如，我们要从数组中同时获取多个对象，那么你就需要将所需要的索引指定为NSIndexSet。它还可以用在类似于数组的结构中；比如，可以向UITableView传递一个NSIndexSet来指定要插入或删除哪个部分。

来看个具体的示例。假设一个NSArray中包含了元素1、2、3、4、8、9与10。NSIndexSet以一种更加简洁的实现来表达这个概念，并且很容易查询。真正的实现我们

是看不到的，不过你可以认为这个 NSIndexSet 包含了两个 NSRange 结构体：{1,4} 与 {8,3}，NSIndexSet 的方法实际上会让你觉得一个 NSIndexSet 是由多个范围构成的。

NSIndexSet 是不可变的；其可变的子类是 NSMutableIndexSet。你可以通过向 indexSetWithIndexesInRange: 传递一个 NSRange 来直接构造只有一个连续范围的 NSIndexSet；但要想构造更加复杂的索引集合，你就需要使用 NSMutableIndexSet，这样就可以附加更多的范围了。

```
let arr = ["zero", "one", "two", "three", "four", "five",
    "six", "seven", "eight", "nine", "ten"]
let ixs = NSMutableIndexSet()
ixs.addIndexesInRange(NSRange(1...4))
ixs.addIndexesInRange(NSRange(8...10))
let arr2 = (arr as NSArray).objectsAtIndexes(ixs)
```

可以通过 for...in 来遍历（枚举）NSIndexSet 所指定的索引值；此外，还可以通过调用 enumerateIndexesUsingBlock:、enumerateRangesUsingBlock: 等方法来遍历 NSIndexSet 的索引或范围。

## 10.4.9 NSArray 与 NSMutableArray

NSArray 是 Objective-C 的数组对象类型。基本上，它相当于 Swift Array，并且可以彼此桥接。不过，NSArray 的元素必须是对象（类与类的实例），这些对象的类型可以不同。要想完整理解 Swift Array 与 Objective-C NSArray 之间的隐式桥接与类型转换，请参见 4.12.1 的“Swift Array 与 Objective-C NSArray ”部分。

在 iOS 9 中，如果 NSArray 对象只有一种元素类型，那么 Objective-C 就可以在其声明中标记出其类型。Swift 2.0 可以读取该标记。这意味着你不会再像过去那样接收到 [AnyObject] 了（不得不向下类型转换为真正的类型）。这一点对于 NSSet 及 NSDictionary 来说也是一样的。

NSArray 的长度是其 count，可以通过 objectAtIndex: 根据索引号获得特定的对象。与 Swift Array 一样，第一个对象的索引是 0，因此最后一个对象的索引是其 count 减 1。

相对于调用 objectAtIndex:，你可以对 NSArray 使用下标。这并非因为 NSArray 会桥接到 Swift Array，而是 NSArray 实现了 objectAtIndexedSubscript:。该方法是 Swift subscript getter 在 Objective-C 中的对应之物，Swift 知道这一点。实际上，当 NSArray 头文件转换为 Swift 时，该方法会表示为一个 subscript 声明！这样，该头文件的 Objective-C 版本声明如下所示：

```
- (ObjectType)objectAtIndexedSubscript:(NSUInteger)idx;
```

不过，相同头文件的 Swift 版本则如下所示：

```
subscript (idx: Int) -> AnyObject { get }
```

（要想理解 Objective-C 声明中 `ObjectType` 的含义，请参见附录 A。）

可以通过 `indexOfObject:` 或 `indexOfObjectIdenticalTo:` 查找数组中的某个对象；前者会调用 `isEqual:`，后者则会使用对象同一性（类似于 Swift 中的 `===`）。如前所述，如果在数组中找不到对象，那么结果就是 `NSNotFound`。

与 Swift Array 不同，但类似于 Objective-C NSString，NSArray 是不可变的。这并不意味着你无法修改其所包含的任何对象；相反，这表示一旦构建好了 NSArray，你就不能再从中删除对象、向其插入对象，或替换掉指定索引处的对象。要想在 Objective-C 中完成这些事情，你可以创建一个新数组，里面包含着原来的数组元素再加上或减去一些对象，或使用 NSArray 的子类 NSMutableArray。Swift Array 并未桥接到 NSMutableArray；如果需要 NSMutableArray，那就得创建它。最简单的方式是使用 NSMutableArray 的初始化器 `init()` 或是 `init(array:)`。

有了 NSMutableArray 后，你就可以调用 NSMutableArray 的 `addObject:` 及 `replaceObjectAtIndex:withObject:` 之类的方法了；还可以通过下标给 NSMutableArray 赋值。这是因为 NSMutableArray 实现了一个特殊的方法 `setObject:atIndexedSubscript:`，Swift 将其看作 `subscript` setter。

此外，除了 `[AnyObject]`，你无法直接将任何类型的 NSMutableArray 转换为 Swift Array；通常的做法是从 NSMutableArray 向上类型转换为 NSArray，然后再向下类型转换为特定类型的 Swift Array：

```
let marr = NSMutableArray()
marr.addObject(1) // an NSNumber
marr.addObject(2) // an NSNumber
let arr = marr as NSArray as! [Int]
```

Cocoa 提供了通过块来搜索或过滤数组的方式。还可以使用排序数组，并通过各种方式提供排序规则；如果是可变数组，那么还可以直接对其排序。你可能更希望在 Swift Array 中执行这些操作，不过了解如何通过 Cocoa 的方式做到这一点也是很有意义的。比如：

```
let pep = ["Manny", "Moe", "Jack"] as NSArray
let ems = pep.objectsAtIndexes(
    pep.indexesOfObjectsPassingTest {
        obj, idx, stop in
```

```
        return (obj as! NSString).rangeOfString(
                "m", options:.CaseInsensitiveSearch
            ).location == 0
    }
) // ["Manny", "Moe"]
```

## 10.4.10 NSDictionary 与 NSMutableDictionary

NSDictionary 是 Objective-C 的字典对象类型。它基本上类似于 Swift Dictionary，并且二者之间会彼此桥接。不过，NSDictionary 的键值必须是对象（类与类的实例），这些对象的类型可以不同；键必须要遵循 NSCopying，并且是可以散列的。请参见 4.12.2 节的“Swift Dictionary 与 Objective-C NSDictionary”部分了解关于如何桥接 Swift Dictionary 与 Objective-C NSDictionary 以及类型转换的详细信息。

NSDictionary 是不可变的；其可变子类是 NSMutableDictionary。Swift Dictionary 并没有桥接到 NSMutableDictionary；你可以通过初始化器 `init()` 或 `init(dictionary:)` 方便地创建一个 NSMutableDictionary。

NSDictionary 的键是不同的（使用 `isEqual:` 进行比较）。如果向 NSMutableDictionary 添加一个键值对，键要是不在其中，那么这个键值对就会被添加进去；但如果键已经存在了，那么相应的值就会被替换掉。这与 Swift Dictionary 的行为类似。

NSDictionary 的基本用法是通过键来获取一个条目的值（使用 `objectForKey:`）；如果键不存在，那么结果就是 `nil`。在 Objective-C 中，`nil` 并非对象，因此它不可能是 NSDictionary 中的值；这个结果的含义是非常明确的。Swift 通过将 `objectForKey:` 的结果看作 `AnyObject?` 来解决这一问题，即包装了 AnyObject 的 Optional。

我们也可以对 NSDictionary 和 NSMutableDictionary 使用下标，原因与可以对 NSArray 和 NSMutableArray 使用下标一样。NSDictionary 实现了 `objectForKeyedSubscript:`，Swift 将其看作 subscript getter。此外，NSMutableDictionary 实现了 `setObject:forKeyedSubscript:`，Swift 将其看作 `subscript` setter。

可以从 NSDictionary 获取键的列表（`allKeys`）、值的列表（`allValues`），以及根据值排序的键的列表。还可以通过块来遍历键值对，甚至可以通过比较值来过滤 NSDictionary。

## 10.4.11 NSSet 及相关类

NSSet 是个由不同对象构成的无序集合。“不同”意味着在使用 `isEqual:` 比较集合中的两个对象时不会返回 `true`。判断集合中是否存在某个对象要比在数组中搜索高效得

多，你可以判断某个集合是否是另一个集合的子集或两个集合是否相交。你可以使用 `for...in` 结构遍历（枚举）集合，当然，顺序是不确定的。你可以过滤集合，就像过滤 NSArray 一样。实际上，你对集合所能进行的操作类似于数组，当然，你不能对集合执行任何涉及排序含义的操作。

要想摆脱这个限制，可以使用有序集合。有序集合（NSOrderedSet）非常类似于数组，并且操作有序集合的方法也非常类似于数组的，你甚至可以通过下标获取元素（因为实现了 `objectAtIndexedSubscript:`）。不过，有序集合的元素必须是不同的。有序集合提供了很多优势：比如，与 NSSet 一样，判断一个对象是否位于有序集合中要比判断数组高效得多，你可以对集合进行并集、交集与差集等运算。既然要求元素不同，这个约束基本上算不上什么约束（因为元素无论如何也得是不同的），因此请尽量使用 NSOrderedSet 而非 NSArray。

将数组传递给有序集合会去重，这意味着顺序不会发生变化，但只有相同对象的第 1 个才会被添加到集合中。

NSSet 是不可变的。你可以通过添加或删除元素从另一个 NSSet 生成一个新的，还可以使用其子类 NSMutableSet。与之类似，NSOrderedSet 也有其可变版本 NSMutableOrderedSet（可以通过下标插入，因为它实现了 `setObject:atIndexed-Subscript:`）。向集合中添加一个对象时，如果这个对象已经在集合中了，那么是不会有什么副作用的；结果就是什么也不会添加进去（唯一性规则会起作用），但也不会报错。

NSCountedSet 是 NSMutableSet 的子类，它是个可变无序的对象集合，并且集合中的对象可以是相同的（这个概念通常也叫作 *Bag*）。它被实现为一个集合，同时还会记录下每个元素被添加的次数。

Swift Set 会被桥接到 NSSet。不过，NSSet 的元素必须是对象（类与类的实例），这些对象的类型可以不同。请参见 4.12.3 节“Swift Set 与 Objective-C NSSet”部分了解详情。Swift 中并没有与 NSMutableSet、NSCountedSet、NSOrderedSet 及 NSMutableOrderedSet 对应的桥接之物，不过可以通过初始化器从集合或数组轻松构建出来。此外，你可以将 NSMutableSet 或 NSCountedSet 向上类型转换为 NSSet，然后再向下类型转换为 Swift Set（类似于 NSMutableArray）。NSOrderedSet 带有一个“门面”属性，可以将其表示为数组或集合。不过由于其特殊的行为，你更倾向于以 Objective-C 的形式来使用 NSCountedSet 与 NSOrderedSet。

## 10.4.12 NSNull

NSNull 类什么都不做，但却提供了一个指向单例对象的指针 `NSNull()`。有时，我们需要一个实际的 Objective-C 对象，但不能为 `nil`，这个单例对象就表示 `nil`。比如，不能将 `nil` 作为 Objective-C 集合元素值（如 NSArray、NSSet 及 NSDictionary），因此需要使用 `NSNull()`。

可以通过普通的相等运算符判断一个对象是否等于 `NSNull()`，因为它会使用 NSObject 的 `isEqual:`，它进行的是同一性比较。这是个单例实例，因此可以使用同一性比较。

## 10.4.13 不变与可变

初学者有时难以理解 Cocoa Foundation 中成对出现的不变与可变类，其中父类都是不变的，子类都是可变的。这不禁令人想起 Swift 对于常量（`let`）与真正的变量（`var`）的区分，它们也有类似的结果。比如，NSArray 是“不变的”，这与我们使用 `let` 来表示 Swift Array 是一个意思：你不能向该数组追加或插入元素，也不能替换或删除该数组中的元素，不过如果数组中的元素是引用类型（当然，对于 NSArray 来说，其元素肯定是引用类型），那么你可以就地修改元素。

Cocoa 需要这些不变 / 可变类的原因在于防止非法修改。这些都是普通的类，NSArray 对象是个普通的类实例——引用类型。如果类有一个 NSArray 属性，并且该数组是可变的，那么该数组就有可能在该类不知情的情况下被其他对象修改。为了防止这种情况发生，类在内部会临时使用一个可变实例，然后将其存储起来并提供给其他类一个不可变的实例，这样可以保护值不被意外修改（Swift 中就没有这个问题，因为其基本的内建对象类型如 String、Array 与 Dictionary 等都是结构体，因此它们是值类型，无法就地修改；它们只能被替换，这可以通过 setter 观察者进行防护或检测）。

文档中可能没有明确表示出可变类已经重写了其不可变父类的方法。比如，NSMutableArray 的很多方法就没有列在 NSMutableArray 类的文档页面中，因为它们都继承自 NSArray。当这种方法被可变子类继承下来时，它们会被重写以符合可变子类的需要。比如，NSArray 的 `init(array:)` 会生成一个不可变数组，不过 NSMutableArray 的 `init(array:)`（它甚至都没有列在 NSMutableArray 的文档页面中，因为它继承自 NSArray）会生成一个可变数组。

这也回答了如何让不可变数组成为可变以及如何让可变数组成为不可变的问题。如果发送给 NSArray 类的 `init(array:)` 生成了一个新的不可变数组，新数组中包含了与原始数组相同的对象且顺序相同，那么发送给 NSMutableArray 类的相同的初始化器 `init(array:)` 就会生成一个可变数组，其中包含了与原始数组相同的对象且顺序相同。

因此，这个方法可以在不变与可变这两个方向传递数组。还可以使用 copy（生成一个不可变副本）与 mutableCopy（生成一个可变副本），它们都继承自 NSObject；不过这两个方法都不是很方便，因为它们生成的都是 AnyObject，你还需要进行类型转换。

这些不变 / 可变类都实现为了类簇，这意味着 Cocoa 会使用一个秘密类，这个类与文档中所记录的那个类是不同的。可以通过查看底层代码了解到这一点；就拿 NSStringFromClass(s.dynamicType) 来说，其中 s 是个 NSString，它会生成一个神秘值 "__NSCFString"。你不应该在这个秘密类上花太多时间。随着时间的流逝，这个类可能会发生变化，但却不会通知你，而且与你也没有任何关系；你永远都不需要知道它。

### 10.4.14 属性列表

属性列表是数据的字符串（XML）表示。只有 Foundation 类 NSString、NSData、NSArray 与 NSDictionary 才能被转换为属性列表。此外，只有当 NSArray 与 NSDictionary 中的类是这些类以及 NSDate 与 NSNumber 时，它们才能被转换为属性列表。（如前所述，这正是你需要将 UIColor 转换为 NSData 才能在 user defaults 中存储的原因所在；user defaults 就是个属性列表。）

属性列表的主要作用是将数据存储为文件。它是值序列化的一种方式，即将值以一种形式存储到磁盘中，然后还可以将这种形式的值重建。NSArray 与 NSDictionary 提供了便捷方法 writeToFile:atomically: 与 writeToURL:atomically: 来分别根据给定的路径名与 URL 生成属性列表文件；相反，它们还提供了初始化器，可以根据给定文件的属性列表内容来创建 NSArray 对象与 NSDictionary 对象。出于这个原因，在创建属性列表时，你可以从这些类开始。（(NSString 与 NSData 的方法 writeToFile:... 与 writeToURL:... 只是将数据直接写到文件中，而非属性列表。）

当通过这种方式从属性列表文件重建 NSArray 或 NSDictionary 对象时，集合、字符串对象与集合中的数据对象都是不可变的。如果希望它们是可变的，或是想将一个属性列表类的实例转换为另一个属性列表，你需要使用 NSPropertyListSerialization 类。（参见 *Property List Programming Guide*。）

## 10.5 访问器、属性与键值编码

从结构上来说，Objective-C 实例变量类似于 Swift 实例属性：它是一个伴随着类的每个实例的变量，其生命周期与值都关联到这个特定的实例。不过，Objective-C 实例变量通常是私有的，这意味着其他类的实例是看不到它的（Swift 看不到）。如果实例变量是公

共的，那么 Objective-C 类通常都会实现访问器方法：getter 方法与 setter 方法（如果外界可以改写这个实例变量）。这种情况很常见，因此有如下命名约定：

*getter 方法*

getter 应该与实例变量有相同的名字（如果实例变量前有下划线，那么 getter 是没有这个下划线的）。这样，如果实例变量是 `myVar`（或 `_myVar`），那么 getter 方法应该命名为 `myVar`。

*setter 方法*

setter 方法名应该以 `set` 开头，后跟大写的实例变量名（如果实例变量前有下划线，那么 setter 是没有这个下划线的）。setter 应该接收一个参数，即准备赋给实例变量的新值。这样，如果实例变量是 `myVar`（或 `_myVar`），那么 setter 应该命名为 `setMyVar:`。

这种模式（一个 getter 方法，可能还有一个命名适当的 setter 方法）非常常见，它还有一种简写形式：Objective-C 类可以通过关键字 `@property` 与一个名字来声明属性。比如，下面这行代码来自于 UIView 类的声明：

```
@property(nonatomic) CGRect frame;
```

（请忽略掉圆括号中的内容。）在 Objective-C 中，这种声明构成了一种承诺，它会提供一个 getter 访问器方法 `frame` 并返回一个 CGRect，同时还会提供一个 setter 访问器方法 `setFrame:` 并接收一个 CGRect 参数。

如果 Objective-C 以这种形式声明 `@property`，那么 Swift 就会将其看作 Swift 属性。这样，UIView 的 `frame` 属性声明就会被直接转换为 Swift 中类型为 CGRect 的实例属性 `frame`：

```
var frame: CGRect
```

Objective-C 的属性名只不过是个语法糖而已。在设置 UIView 的 `frame` 属性时，实际会调用其 `setFrame:` setter 方法；在获取 UIView 的 `frame` 属性时，实际会调用其 `frame` getter 方法。在 Objective-C 中，属性的使用是可选的；Objective-C 可以并且经常会直接调用 `setFrame:` 与 `frame` 方法。不过在 Swift 中却不行。如果 Objective-C 类有正式的 `@property` 声明，那么其访问器方法会对 Swift 隐藏。

Objective-C 属性声明可以在圆括号中包含单词 `readonly`。这表示只有 getter 但却没有 setter。比如（请忽略掉圆括号中的其他内容）：

```
@property(nonatomic,readonly,strong) CALayer *layer;
```

Swift 会在声明后通过 {get} 来反映出这种限制，就好像这是个计算只读属性一样；编译器不允许为这样的属性赋值：

```
var layer: CALayer { get }
```

Objective-C 属性及相应的访问器方法都有自己的生命周期，独立于任何底层的实例变量。虽然访问器方法可用于访问不可见的实例变量，但不一定总是这样的。在设置 UIView 的 frame 属性并且 setFrame: 访问器方法得到调用时，你是不知道该方法到底做了哪些事情：它可能会设置一个名为 frame 或 _frame 的实例变量，但谁知道呢？从这个意义上来说，访问器与属性就是门面，隐藏了底层的实现。这与 Swift 中设置变量但又不知道或不关心它是个存储变量还是个计算变量类似；对于设置变量的代码来说，变量真正所设置的东西是不重要的（可能也是不知道的）。

## 10.5.1 Swift 访问器

就像 Objective-C 属性实际上只是访问器方法的简写一样，Objective-C 将 Swift 属性也看作访问器方法的简写形式，即便没有这样的方法亦如此。如果在 Swift 中声明的类有一个名为 prop 的属性，Objective-C 就会调用 prop 方法获取其值，调用 setProp: 方法设置其值，即便没有实现这样的方法亦如此。这些调用会通过隐式的访问器方法路由到属性。

在 Swift 中，你不应该为属性编写显式的访问器方法！编译器不允许你这么做。如果需要显式实现访问器方法，那么请使用计算属性。比如，我向 UIViewController 子类添加了一个名为 color 的计算属性并提供 getter 与 setter：

```
class ViewController: UIViewController {
    var color : UIColor {
        get {
            print("someone called the getter")
            return UIColor.redColor()
        }
        set {
            print("someone called the setter")
        }
    }
}
```

Objective-C 代码现在可以显式调用隐式的 setColor: 与 color 访问器方法，当调用时，你会看到计算属性的 setter 与 getter 方法实际上会被调用：

```
ViewController* vc = [ViewController new];
[vc setColor:[UIColor redColor]]; // "someone called the setter"
UIColor* c = [vc color]; // "someone called the getter"
```

这证明了在 Objective-C 中，你已经提供了 setColor: 与 color 访问器方法。你甚至可以修改这些访问器方法的 Objective-C 名字！要想做到这一点，请添加一个 @objc(...) 特性，并在圆括号中放入其 Objective-C 名字。可以将其添加到计算属性的 setter 与 getter 方法中，或添加到属性自身当中：

```
@objc(hue) var color : UIColor?
```

Objective-C 代码现在可以直接调用 hue 与 setHue: 访问器方法了。

如果只是想向 setter 添加功能，那么可以使用 setter 观察者。比如，要想向 UIView 子类中的 Objective-C setFrame: 方法添加功能，那么可以覆写 frame 属性并添加一个 didSet 观察者：

```
class MyView: UIView {
    override var frame : CGRect {
        didSet {
            print("the frame setter was called: \(super.frame)")
        }
    }
}
```

## 10.5.2 键值编码

Cocoa 可以通过运行期指定的字符串名动态调用访问器（这样就可以访问 Swift 属性了），这种机制叫作键值编码（KVC），类似于通过 respondsToSelector: 使用选择器名进行内省的能力。字符串名是键；传递给访问器或从访问器返回的是值。键值编码的基础是 NSKeyValueCoding 协议，这是个非正式协议；它实际上是个注入 NSObject 中的类别。因此，要想使用键值编码，Swfit 类必须要继承自 NSObject。

基本的键值编码方法是 valueForKey: 与 setValue:forKey:。当在对象上调用其中一个方法时，该对象就会被内省。简而言之，首先会寻找恰当的选择器；如果不存在，那就会直接访问实例变量。另外一对有用的方法是 dictionaryWithValuesForKeys: 与 setValuesForKeysWithDictionary:，你可以通过它们仅使用一个命令就能以 NSDictionary 的方式获取与设置多个键值对。

键值编码中的值必须是个 Objective-C 对象，其 Swift 类型是 AnyObject。在调用 valueForKey: 时，你会接收到一个包装了 AnyObject 的 Optional，需要将其向下类型转换为真实的类型。

对于给定的键来说，如果一个类提供了访问器方法或拥有实例变量来对其进行访问，那么我们会说这个类针对于该键是键值编码兼容的（或 KVC 兼容的）。对于给定的键来说，

如果一个类不是对其键值编码兼容的，那么访问该键会导致运行期异常。当出现这种崩溃时，如果熟悉抛出的消息将是大有裨益的，下面就来故意制造崩溃的结果：

```
let obj = NSObject()
obj.setValue("hello", forKey:"keyName") // crash
```

控制台会打印出消息“This class is not key value coding-compliant for the key keyName.”。虽然缺少引号，但这条错误消息的最后一个单词是导致崩溃的键字符串。

如何才能让上述方法调用不崩溃呢？接收方法调用的对象所属的类需要有一个 setKeyName: setter 方法（或 keyName 及 _keyName 实例变量）。如 10.5.1 节所述，在 Swift 中，实例属性表示存在访问器方法。这样，我们可以在拥有所声明的属性的任何 NSObject 子类实例上使用键值编码，前提是键字符串是该属性的字符串名。下面就来试一下！类声明如以下代码所示：

```
class Dog : NSObject {
    var name : String = ""
}
```

下面是测试代码：

```
let d = Dog()
d.setValue("Fido", forKey:"name") // no crash!
print(d.name) // "Fido" - it worked!
```

## 10.5.3 键值编码的使用

实际上，你可以通过键值编码在运行期根据字符串来决定调用哪个访问器。最简单的一种情况就是，你可以通过字符串访问动态指定的属性。这在 Objective-C 代码中是非常有价值的；不过，这种自由的内省能力与 Swift 的精神恰恰相反，在将我自己编写的 Objective-C 代码转换为 Swift 时，我发现可以通过其他方式达到相同的目的。

下面是个示例。在 flashcard 应用中有个名为 Term 的类，代表一个拉丁语单词。它声明了很多属性。每个卡片会显示一个单词，其各种属性会显示在不同的文本域中。如果用户轻拍了 3 个文本域之一，那么我希望界面上显示的单词换成下一个，并且这个单词要不同于上一个。这样，代码对于三个文本域来说都是一样的；唯一的差别是在寻找下一个显示的单词时，我们应该考虑哪个属性。在 Objective-C 中，到目前为止最简单的方式就是使用键值编码：

```
NSInteger tag = g.view.tag; // the tag tells us which text field was tapped
NSString* key = nil;
switch (tag) {
    case 1: key = @"lesson"; break;
```

```
        case 2: key = @"lessonSection"; break;
        case 3: key = @"lessonSectionPartFirstWord"; break;
    }
    // get current value of corresponding instance variable
    NSString* curValue = [[self currentCardController].term valueForKey: key];
```

不过在 Swift 中，可以通过匿名函数数组来完成相同的功能：

```
let tag = g.view!.tag - 1
let arr : [(Term) -> String] = [
    {$0.lesson}, {$0.lessonSection}, {$0.lessonSectionPartFirstWord}
]
let f = arr[tag]
let curValue = f(self.currentCardController().term)
```

不过，键值编码在 iOS 编程中也是颇具价值的，特别是因为很多内建的 Cocoa 类都允许以特殊的方式使用键值编码。比如：

- 如果向 NSArray 发送 `valueForKey:`，那么它会向数组中的每个元素发送 `valueForKey:`，并返回一个包含了结果的新数组，这是一种优雅的简写方式，NSSet 亦如此。
- NSDictionary 实现了 `valueForKey:` 以作为 `objectForKey:` 的替代方案（这对于字典构成的 NSArray 来说特别有用）。与之类似，NSMutableDictionary 会将 `setValue:forKey:` 作为 `setObject:forKey:` 的同义；只不过在调用 `removeObject:forKey:` 时 `value:` 可以为 `nil`。
- NSSortDescriptor 通过向 NSArray 中的每个元素发送 `valueForKey:` 来对 NSArray 排序。这样，根据特定的字典键值对字典数组排序，以及根据特定的属性值对对象数组排序就变得很容易了。
- NSManagedObject（与 Core Data 配合使用）用于确保对在实体模型中所配置的特性做到键值编码兼容。这样，我们就可以通过 `valueForKey:` 与 `setValue:forKey:` 访问这些特性了。
- 可以通过 CALayer 与 CAAnimation 使用键值编码来定义并获取任意键的值，就好像它们是字典一样；它们实际上对于每个键都是键值编码兼容的。这对于向这些类的实例附加识别与配置信息是非常有用的。实际上，这是我在 Swift 中使用键值编码最常用的方式。

## 10.5.4 KVC 与插座变量

键值编码是插座变量连接能够正常运作的关键所在（参见第 7 章）。Nib 中的插座变量名是个字符串，键值编码会在 nib 加载时将该字符串转换为所要寻找的属性。

假设有一个 Dog 类，它有一个 @IBOutlet 属性 master，你绘制了一个从 nib 中的这个类到 Person nib 对象上的 "master" 插座变量。当 nib 加载时，插座变量名 "master" 会通过键值编码转换为访问器方法名 setMaster:，Dog 实例的 setMaster: 隐式访问器方法在调用时会将 Person 实例作为其参数，这样会将 Dog 实例的 master 属性值设为 Person 实例（如图 7-9 所示）。

如果 nib 中的插座变量名与类中的属性名之间不匹配，那么在运行期，当 nib 加载时，Cocoa 会使用键值编码根据插座变量名来设置对象中的值，但却会失败，并抛出异常，错误消息表示该类针对于这个键（插座变量名）并不是键值编码兼容的；也就是说，应用会在 nib 加载时崩溃。另一种类似的情况是插座变量是正确的，但后面却修改或删除了类中的属性名（参见 7.3.4 节）。

## 10.5.5 键路径

可以通过键路径在一个表达式中将键串联起来。如果一个对象针对某个键是键值编码兼容的，并且该键所对应的值又针对另一个键是键值编码兼容的，那就可以通过调用 valueForKeyPath: 与 setValue:forKeyPath: 将这两个键串联起来。键路径字符串看起来像是使用点符号链接起来的一连串键名。比如，valueForKeyPath("key1.key2") 会调用消息接收者的 valueForKey:，并且将 "key1" 作为键，然后获得调用所返回的对象，并对该对象调用 valueForKey:，并且将 "key2" 作为键。

为了演示这种简写方式，假设对象 myObject 有一个实例属性 theData，它是个字典数组，这样每个字典都有一个 name 键和一个 description 键：

```
var theData = [
    [
        "description" : "The one with glasses.",
        "name" : "Manny"
    ],
    [
        "description" : "Looks a little like Governor Dewey.",
        "name" : "Moe"
    ],
    [
        "description" : "The one without a mustache.",
        "name" : "Jack"
    ]
]
```

我们可以通过键值编码与键路径钻取到该字典数组中：

```
let arr = myObject.valueForKeyPath("theData.name") as! [String]
```

结果是个包含了字符串 "Manny""Moe" 与 "Jack" 的数组。如果不清楚原因，那么请回顾一下之前介绍的 NSArray 与 NSDictionary 是如何实现 valueForKey: 的吧。

再回顾一下第 7 章介绍的自定义运行时特性。该特性就使用了键值编码！当在对象的身份查看器中定义了运行时特性时，你在第 1 列中所输入的字符串就是个键路径。

### 10.5.6 数组访问器

键值编码是一项强大的技术，同时拥有很多衍生品（参见 Apple 的 *Key-Value Coding Programming Guide* 了解详情）。这里只介绍其中一种。如果键的值看起来像是个数组或是集合（即便实际情况并非如此），那么借助于键值编码，对象可以将键合成起来。你需要实现特别命名的访问器方法；在使用相应的键时，键值编码会看到它们。

为了说明这一点，向对象 myObject 所属的类添加如下方法：

```
func countOfPepBoys() -> Int {
    return self.theData.count
}
func objectInPepBoysAtIndex(ix:Int) -> AnyObject {
    return self.theData[ix]
}
```

通过实现 countOf... 与 objectIn...AtIndex:，我告诉键值编码系统认为给定的键 "pepBoys" 存在并且是个数组。通过键值编码方式获取键 "pepBoys" 的值的操作可以成功，并且返回的对象可以看作 NSArray，但实际上它是个代理对象（NSKeyValueArray）。现在可以这样做：

```
let arr : AnyObject = myObject.valueForKey("pepBoys")!
let arr2 : AnyObject = myObject.valueForKeyPath("pepBoys.name")!
```

在上述代码中，arr 是个数组代理；arr2 是由 3 个男孩的名字构成的相同的数组。这个示例看起来毫无意义：底层实现已经是个数组了，使用 "pepBoys" 与之前所用的 "theData" 有何区别呢？表面上看没什么不同，但实际情况却并非如此。假设并没有数组存在，countOfPepBoys 与 objectInPepBoysAtIndex: 的结果是通过完全不同的操作得到的。实际上，我们创建了一个类似于 NSArray 的键；并且将一些实现细节隐藏在其后。

## 10.6 NSObject 揭秘

由于每个 Objective-C 类都继承自 NSObject，因此我们有必要花些时间了解一下 NSObject。NSObject 的构造方式很复杂：

- 它定义了一些本地类方法与实例方法，主要与实例化基本功能及方法发送和解析相关。（参见 *NSObject Class Reference*。）

- 它使用了 NSObject 协议。该协议声明了与内存管理相关的实例方法、实例与类之间的关系以及内省机制。由于所有的 NSObject 协议方法都是必需的， NSObject 类会将其全部实现出来。（参见 *NSObject Protocol Reference*。）在 Swift 中，NSObject 协议叫作 NSObjectProtocol，目的在于避免命名冲突。
- 它实现了与 NSCopying、NSMutableCopying 与 NSCoding 协议相关的便捷方法，但却没有正式使用这些协议。NSObject 有意不使用这些协议的原因在于这会导致所有其他的类都要使用该协议，但这么做是错误的。多亏了该架构，如果某个类使用了其中一个协议，那么你就可以调用相应的便捷方法。比如，NSObject 实现了 `copy` 实例方法，这样你就可以在任何实例上调用 `copy` 了，但如果实例所属的类没有使用 NSCopying 协议并实现 `copyWithZone:`，那就会导致应用崩溃。
- 有大量方法通过 NSObject 上的 20 多个类别注入了 NSObject 中，它们散落在各种头文件中。比如，`awakeFromNib`（参见第 7 章）来自于 NSObject 上的 UINibLoadingAdditions 类别，它声明在 *UINibLoading.h* 中。
- class 对象也是个对象。因此，所有 Objective-C 类（Class 类型的对象）都继承自 NSObject。这样，NSObject 所定义的任何实例方法都能以类方法的形式在 class 对象上调用！比如，`respondsToSelector:` 是 NSObject 定义的一个实例方法，但也可以将其看作类方法并发送给 class 对象。

程序员所面临的一个问题就是 Apple 的文档在分类上过于死板。如果想了解某个对象，那么就不要管该对象的方法来自于何处；只要知道方法是什么就行。但 Apple 通过来源对方法进行了区分。虽然 NSObject 是根类，也是最重要的类，所有其他的类都继承自它，但没有一页文档概述了所有这些方法。相反，你需要同时查询 *NSObject Class Reference* 与 *NSObject Protocol Reference*，还有 NSCopying、NSMutableCopying 与 NSCoding 协议的文档页（为了理解它们如何与 NSObject 定义的方法进行交互），你还要提供每个 NSObject 实例方法的类方法版本！

还有一些方法通过类别注入了 NSObject 中。一些通用方法会在 NSObject 类文档页面中进行说明；比如，`awakeAfterUsingCoder:` 来自于声明在单独头文件中的类别，不过它却在 NSObject 文档中进行了说明，因为它是个类方法，也是个全局方法，你可以随时使用它。其他一些则是使用受限的委托方法（其实是非正式协议），不需要集中说明；比如，`animationDidStart:` 记录在 CAAnimation 类中，因为只有在使用 CAAnimation 时你才需要了解它。不过，每个对象都可以响应 `awakeFromNib`，它对于你所编写的每个应用都是至关重要的，因此需要单独学习；对其的介绍在 *NSObject UIKit Additions Reference* 中，你可能找不到！同样，所有的键值编码方法（参见本章前面的介绍）与键值观测方法（参见第 11 章）都如此。

使出浑身解数找出了所有的 NSObject 方法后，你会发现它们可以很自然地划分为几类：

*创建、销毁与内存管理*

用于创建实例的方法，如 `alloc` 与 `copy`，以及用于了解对象生命周期中发生了哪些事情的方法，如 `initialize` 与 `dealloc`，还有用于管理内存的方法。

*类关系*

用于获取对象所属类与继承关系的方法，如 `superclass`、`isKindOfClass:` 与 `isMemberOfClass:`。

*对象内省与比较*

用于确定假如向某个对象发送某条消息时将会发生什么事情的方法，如 `respondsToSelector:`；用于将对象表示为字符串（`description`）与对象比较（`isEqual:`）的方法。

*消息响应*

用于确定当向某个对象发送某条消息时将会发生什么的方法，如 `doesNotRecognizeSelector:`。如果感兴趣，请参见 *Objective-C Runtime Programming Guide*。

*消息发送*

用于动态发送消息的方法。比如，`performSelector:` 接收一个选择器作为参数，并会将其发送给一个对象，告诉该对象执行这个选择器。这看起来与将该消息发送给这个对象是一样的，不过如果直到运行期才知道所发送的消息该怎么做呢？此外，`performSelector:` 的各变种可以在指定的线程上发送消息，或是经过一段时间后再发送消息（`performSelector:withObject:afterDelay:` 等）。

`performSelector...` 方法是 Swift 2.0 新引入的。之前，在 Swift 中是无法调用它的；我经常在 Objective-C 中使用它们，不过将代码转换为 Swift 迫使我寻找解决问题的其他方法，我发现在没有它们的情况下我也能很好地进行管理。

# 第 11 章

# Cocoa 事件

应用的所有可执行代码都位于函数中，并且函数会被其他地方所调用。其中会有一个函数调用另一个函数，不过第一个函数是被谁调用的呢？从根本上来说，你的代码是如何运行的呢？正如第 6 章所述，当应用启动后，“`UIApplicationMain` 就在那儿，等待用户的操作，维护事件循环，并且响应用户的动作”。

事件循环是关键。运行时会监控并等待某些事情的发生，比如，用户在屏幕上的手势操作，或是应用生命周期某个特定的阶段出现。当这样的事情发生时，运行时会调用你的代码。不过，只有准备好了代码，运行时才能调用。你的代码就像是一个按钮面板一样，等待着 Cocoa 按下。如果发生了 Cocoa 认为你的代码需要知道并响应的事情，那么它就会按下正确的按钮，前提是按钮得在那儿。

Cocoa 编程的艺术在于要知道 Cocoa 想要做什么。在一开始组织代码时就要知道 Cocoa 的行为。Cocoa 对于如何以及何时向你的代码分发消息做出了一些承诺。这是 Cocoa 的事件。你知道这些事件是什么，当 Cocoa 分发这些事件时，你的代码需要对其做出响应。

文档中列出了你所能接收到的具体事件。如何以及何时分发事件，你的代码以何种方式接收这些事件的整体架构是本章将要介绍的主题。

## 11.1 为何使用事件

总的来说，接收到一个事件的原因可以分为 4 类。这种分类并不是正式的，而是我划分的。通常，一个事件属于哪个类别并不是特别明确的；一个事件可能属于两个类别。但这些事件类别对于搞清楚 Cocoa 与你的代码交互的方式与原因还是颇具价值的。

*用户事件*

用户做了某个交互式的动作，事件就会直接被触发。显而易见的就是当用户轻拍、滑动屏幕或是在键盘上输入时所获得的事件。

*生命周期事件*

这些事件用于通知你应用生命周期的某个阶段到来了，比如，应用启动或即将进入到后台；还可以通知你应用组件生命周期中的某个阶段到来了，比如，UIViewController 的视图刚刚加载完毕或即将从界面中被移除。

*功能性事件*

Cocoa 将要做某事，如果你想要提供额外的功能，那么 Cocoa 就会将控制权交给你。我可以将诸如 UIView 的 `drawRect:`（让视图绘制自身）、UILabel 的 `drawTextInRect:`（修改标签的外观）归于此类，第 10 章曾对其做过介绍。

*查询事件*

Cocoa 会向你提问；其行为取决于你的答案。比如，数据出现在表格（UITableView）中的方式就是当 Cocoa 需要为表格的行添加单元格时，它会向你索要该单元格。

## 11.2 子类化

内建的 Cocoa 类所定义的方法可能会被 Cocoa 本身所调用，也可能需要在子类中重写，这样在调用方法时才会执行自定义行为而不仅仅是默认行为。

第 10 章曾介绍过的一个示例是 UIView 的 `drawRect:`，它就是我所说的功能性事件。通过在 UIView 子类中重写 `drawRect:`，你可以描绘出视图绘制自身的完整过程。你并不知道该方法何时会被调用，你也不在意这个；确定的是在绘制时，它能够确保视图总是按照你所期望的样子呈现出来（你永远不会自己调用 `drawRect:`；如果底层情况发生了变化，你希望视图重绘，那么就需要调用 `setNeedsDisplay` 并让 Cocoa 调用 `drawRect:` 进行响应）。

内建的 UIView 子类还有其他一些功能性事件方法需要你通过子类化进行定制。一般来说，定制的目的在于改变视图的绘制方式，但又不需要自己控制完整的绘制过程。第 10 章提及的一个示例涉及了 UILabel 及其 `drawTextInRect:`。另一个类似的示例是 UISlider，你可以通过重写 `thumbRectForBounds:trackRect:value:` 来自定义滚动条“拇指”的位置与尺寸。

UIViewController 这个类的设计目的就是让你子类化。在 UIViewController 类文档所列出的方法中，几乎全部方法都有重写的必要。如果在 Xcode 中创建了一个 `UIViewController` 子类，那么你会看到模板中已经包含了一些重写的方法供你起步。比

如，viewDidLoad会被调用以便让你知晓视图控制器已经获得了其主视图（即它的视图），这样就可以进行初始化了；显然，这是个生命周期事件。UIViewController还有其他很多生命周期事件，你可以重写它们对发生的事情进行控制。比如，viewWillAppear:表示视图控制器的视图将会被放置到界面上；viewDidAppear:表示视图控制器的视图已经被放置到了界面上；viewDidLayoutSubviews表示视图已经在其父视图中定位了，诸如此类。

我称supportedInterfaceOrientations之类的UIViewController方法为查询事件。你的任务就是返回一个位掩码来告诉Cocoa，视图在某个时刻可以处于哪个方向。你相信Cocoa会在恰当的时刻调用这个方法，这样如果用户旋转了设备，应用的界面就会根据返回值决定旋转还是不旋转。

在寻找通过子类化可以接收到的事件时，请记得沿着继承体系向上查找。比如，如果想知道在将自定义UILabel子类嵌入另一个视图时如何收到通知，你不应该在UILabel类文档中寻求答案；UILabel是个UIView，因此它会接收到恰当的事件。在UIView类文档中，你会发现可以通过重写didMoveToSuperview以便在这种情况下接收到通知。同样，还要记得沿着所使用的协议向上查找。如果想知道当视图控制器的视图将要旋转时该如何收到通知，你不应该在UIViewController类文档中寻求答案；UIViewController使用了UIContentContainer协议，因此它会接收到恰当的事件。在UIContentContainer协议文档中，你会发现可以通过重写viewWillTransitionToSize:withTransitionCoordinator:来做到这一点。

不过，正如第10章所述，子类化与重写并非接收事件最重要与最常见的方式。除了UIViewController，我们很少会为了这个目的而子类化其他内建的Cocoa类。那么，你的代码还可以通过哪些方式接收事件呢？这正是本章所要介绍的主题。

## 11.3 通知

Cocoa为应用提供了一个单例的NSNotificationCenter，它的非正式名称叫作通知中心。该实例可以通过调用NSNotificationCenter.defaultCenter()来获取，它是消息发送（叫作通知）机制的基础。一个通知就是一个NSNotification实例。其想法是任何对象都可以注册到通知中心以接收某些通知。另一个对象可以向通知中心发送通知对象（叫作发布通知）。接下来，通知中心就会向所有注册以准备接收通知的对象发送该通知。

人们经常将通知机制称为分发或广播机制，这样描述的理由也很充分。凭借该机制，对象可以发送消息而不必了解或关心什么对象、多少对象会接收到该通知。这样做简化了应用的架构，使得系统不必再将实例连接起来才能实现彼此间的消息传递（这有时是很

难做到的，第 13 章将会介绍）。当对象从概念上做到了彼此间的“隔离”，通知就是一个相当轻量级的方式，可以让一个对象向另一个对象发送消息。

一个 NSNotification 对象包含了 3 部分信息，这些信息可以通过其实例方法获取到：

`name`

一个 NSString，表示通知的含义。

`object`

与通知关联的一个实例；一般来说是发送通知的实例。

`userInfo`

并非每个通知都有 `userInfo`；它是个 NSDictionary，可以包含与通知相关的一些附加信息，该 NSDictionary 到底包含什么信息，信息位于哪些键中取决于具体的通知；你需要查询文档才能获悉。比如，文档表明 UIApplication 的 `UIApplicationDidChangeStatusBarOrientationNotification` 包含了一个 `userInfo` 字典，字典有一个 `UIApplicationStatusBarOrientationUserInfoKey` 键，其值是状态栏之前的方向。在自己发布通知时，你可以将任何感兴趣的信息放到 `userInfo` 中供通知接收者获取。

Cocoa 本身会通过通知中心来发送通知，你的代码可以注册到通知中心来接收通知。对于提供了通知的类的文档，你会看到有一个单独的 Notifications 部分来介绍它们。

## 11.3.1 接收通知

要想注册以接收通知，你需要向通知中心发送如下两条消息之一，一个是 `addObserver:selector:name:object:`，其参数如下所示。

`observer:`

通知发向的实例。它通常是 `self`；一般不会出现一个实例将另一个不同的实例注册为通知接收者的情况。

`selector:`

当通知出现时，发送给观察者实例的消息。指定的方法应该不返回结果（Void）并且带有一个参数，参数是 NSNotification 实例（因此，参数的类型应该是 `NSNotification` 或 `AnyObject`）。在 Swift 中，可以通过将方法名作为字符串来指定选择器。

不要将方法的字符串名搞错了，也不要忘记实现方法！如果通知中心通过调用作为选择器的方法来发送通知，并且该方法不存在，那么应用就会崩溃。参见附录 A 了解关于如何将方法转换为字符串名的规则。

只有当选择器所命名的方法公开给了 Objective-C 时才能调用它。如果通知中心通

过调用作为选择器的方法来发送通知，但 Objective-C 不知道这个方法，那么应用就会崩溃。如果类是 NSObject 的子类，或方法标记为 @objc（或 dynamic），那么 Objective-C 才能知道这个方法。

name:

你想要接收的通知的字符串名。如果该参数为 nil，那么你就会接收到与 object: 参数中所指定的对象相关的所有通知。内建的 Cocoa 通知名通常是个常量。这是很有用的，因为如果搞乱了常量名，编译器就会报错，如果直接以字符串字面值的形式输入通知名，但却输错了，那么编译器就不会报错，但却无法收到任何通知了（因为没有与你输入的名字所对应的通知），这种错误是很难追踪的。

object:

你所感兴趣的通知对象，通常是发布通知的那个对象。如果为 nil，那么你就会接收到 name: 参数中所指定名字的所有通知（如果 name: 与 object: 参数都为 nil，那么你就会接收到所有通知）。

比如，在我的一个应用中，我希望在设备的音乐播放器开始播放下一首歌曲时界面能够变化。设备内建的音乐播放器 API 属于 MPMusicPlayerController 类；该类提供了一个通知，告诉我内建的音乐播放器何时改变了正在播放的歌曲，该通知的说明位于 MPMusicPlayerController 类文档中的 Notifications 下，名字是 MPMusicPlayerController-NowPlayingItemDidChangeNotification。

查看文档会发现，只有先调用 MPMusicPlayerController 的 beginGeneratingPlaybackNotifications 实例方法后才会发送该通知。这种架构很常见；对于某些通知来说，只有开启后 Cocoa 才会发送，从而提升了效率。这样，我首先需要获取到 MPMusicPlayerController 的一个实例，然后调用该方法：

```
let mp = MPMusicPlayerController.systemMusicPlayer()
mp.beginGeneratingPlaybackNotifications()
```

现在，注册自身以接收所需的播放通知：

```
NSNotificationCenter.defaultCenter().addObserver(self,
    selector: "nowPlayingItemChanged:",
    name: MPMusicPlayerControllerNowPlayingItemDidChangeNotification,
    object: nil)
```

这样，当发布 MPMusicPlayerControllerNowPlayingItemDidChangeNotification 通知后，nowPlayingItemChanged: 方法就会被调用：

```
func nowPlayingItemChanged (n:NSNotification) {
    self.updateNowPlayingItem()
```

```
        // ... and so on ...
    }
```

要想让 addObserver:selector:name:object: 能够正常运作，你需要获取到正确的选择器，并确保实现了相应的方法。大量使用 addObserver:selector:name:object: 意味着代码中会充斥着大量仅供通知中心调用的方法。并没有关于这些方法的说明（你需要添加一些注释来提醒自己），同时这些方法又独立于注册调用，所有这一切使代码变得非常混乱。

可以通过另一种通知注册方式来解决这个问题，即调用 addObserverForName:object:queue:usingBlock:。它会返回一个值，这个值的目的将会在后面介绍。queue: 通常为 nil；非 nil 的 queue: 用于后台线程。name: 与 object: 参数就像是 addObserver:selector:name:object: 中相应的参数一样。相对于使用观察者与选择器，你需要提供一个 Swift 函数，它包含了通知到达时所要执行的实际代码。该函数接收一个参数，即 NSNotification 本身。如果使用了匿名函数，那么对于所注册的通知的响应就会成为注册的一部分：

```
let ob = NSNotificationCenter.defaultCenter()
    .addObserverForName(
        MPMusicPlayerControllerNowPlayingItemDidChangeNotification,
        object: nil, queue: nil) {
            _ in
            self.updateNowPlayingItem()
            // ... and so on ...
        }
```

使用 addObserverForName:... 会导致一些额外的内存管理问题，第 12 章将会对此进行介绍。

## 11.3.2 取消注册

对于注册为通知接收者的每个对象，你可以在其销毁之前取消注册。如果没有取消注册，对象又不存在了，同时该对象注册以接收的消息又发送出来了，那么通知中心就会尝试向该对象发送恰当的消息，现在就接收不到了。这样，最好的结果就是应用崩溃，最糟糕的结果就是出现了混乱。

要想取消注册通知接收者对象，请向通知中心发送 removeObserver: 消息（此外，还可以使用 removeObserver:name:object: 让对象取消注册特定的通知集）。作为 observer: 参数所传递的对象就是不再接收通知的那个对象。这个对象是什么取决于你一开始是如何注册的：

*调用了 addObserver:...*

一开始就提供了观察者；它就是现在要取消注册的那个观察者。

*调用了 addObserverForName:...*

对 addObserverForName:... 的调用会返回一个类型为 NSObjectProtocol 的观察者标记对象（无须关心其真实的类型与特性）；它就是现在要取消注册的那个观察者。

棘手之处在于要找到恰当的时机来取消注册。靠谱的解决方案是注册实例的 deinit 方法，它是实例销毁前所接收到的最后一个生命周期事件。

如果调用了同一个类的 addObserverForName:... 多次，那就会从通知中心接收到多个观察者标记，你需要将其保存起来以便后续可以取消他们的注册。如果想一次取消注册全部对象，一种方案就是使用类型为可变集合的实例属性。我喜欢使用 Set 属性：

```
var observers = Set<NSObject>()
```

每次使用 addObserverForName... 注册通知时，我都会捕获到结果并将其添加到集合中：

```
let ob = NSNotificationCenter.defaultCenter().addObserverForName(/*...*/)
self.observers.insert(ob as! NSObject)
```

在取消注册时，我会枚举集合并将其清空：

```
for ob in self.observers {
    NSNotificationCenter.defaultCenter().removeObserver(ob)
}
self.observers.removeAll()
```

NSNotificationCenter 是无法内省的：你不能通过 NSNotificationCenter 获取到注册为通知接收者的对象。这是 Cocoa 功能的一个欠缺，如果犯了诸如过早取消某个观察者的注册这类错误，那么 Bug 是很难追踪的（与往常一样，这也来自于我痛苦的经历）。

## 11.3.3 发布通知

虽然很多时候都是从 Cocoa 接收通知的，不过也可以利用通知机制实现自定义对象间的通信。这么做的一个原因在于两个对象从概念上会彼此独立。不过，值得注意的是不要过多地使用通知，也不要将其作为对象间通信的链路；但在某些情况下它们还是非常适合的（第 13 章将会介绍）。

要想按照这种方式使用通知，对象会在通信链路中扮演两个角色。一个或多个对象会注册以接收通知，如前所述，这是通过名字、对象或二者的结合体来标识的。另一个对象

会发布通知，标识方式也是一样的。接下来，通知中心会将消息从发送者传递给注册的接收者。

要想发布通知，请向通知中心发送消息 postNotificationName:object:userInfo:。

比如，我曾开发过一款简单的纸牌游戏。游戏需要知道用户什么时候轻拍了纸牌，不过纸牌却对游戏一无所知；当用户轻拍纸牌时，它只是通过发布通知发出声音而已：

```
NSNotificationCenter.defaultCenter().postNotificationName(
    "cardTapped", object: self)
```

游戏对象注册过了 "cardTapped" 通知，因此它会知道这一点并接收到通知的 object；现在，它知道用户轻拍了哪个纸牌，并且可以正确地进行处理。

## 11.3.4 NSTimer

严格来说，定时器（NSTimer）并非通知；但其行为非常类似于通知。它是一个对象，在某段时间间隔后会发出一个信号。这个信号就是发给一个实例的消息。这样，当某段时间过后，你就可以收到通知了。时间并不是非常精确的，但用起来没什么问题。

定时器管理并不难，但有点与众不同。定时器会不断检查时钟，我们称这种行为为调度。定时器可能会被触发一次，也可能是个重复定时器。要想销毁定时器，首先要将其置为无效状态。设定为只触发一次的定时器会在触发后自动变为无效状态；重复定时器会不断重复执行，直到你通过向其发送 invalidate 消息将其置为无效状态。你不应该再使用处于无效状态的定时器；也不能将其复活或使用它做别的事情，你也不应该向其发送任何消息。

创建定时器的直接方式是使用 NSTimer 类的 scheduledTimerWithTimeInterval:target:selector:userInfo:repeats: 方法。这会创建定时器并对其进行调度，这样定时器就会自动开始检查时钟了。目标与选择符决定了当定时器触发时可以向什么对象发送什么消息；处理的方法应该接收一个参数，该参数是指向定时器的引用。userInfo 就像是通知的 userInfo 一样。

对于 Timer 的 target: 与关于 NSNotifications 的 selector: 也要小心。当定时器触发时，目标一定要存在，它要有一个与动作选择器相对应的方法，Objective-C 必须要能调用该方法。否则就会出问题。

NSTimer 有个 tolerance 属性，它是个时间间隔，表示定时器可以在指定的触发时间与

这个时间加上 tolerance 之间的某一时刻触发。文档表明可以通过为其提供一个至少为 timeInterval 10% 的值来改进设备电池寿命与应用响应性。

比如，我开发过一个应用，它是个游戏并且带有分数；如果用户在 10 秒内没有移动，那么我就要减分来惩罚用户。这样，每次用户移动时，我都会创建一个重复定时器，其时间间隔是 10 秒（我还会将任何现有的定时器都置为无效）；在定时器调用的方法中，我会减分。

定时器存在一些内存管理问题，第 12 章将会介绍，此外定时器还有基于块的替代方案。

# 11.4 委托

委托是一种面向对象的设计模式，指的是两个对象间的关系，其中主对象的行为是通过另一个对象定制或协助处理的。第 2 个对象就是主对象的委托。这里面不涉及子类化，实际上，第 1 个对象对委托类一无所知。

由于 Cocoa 实现了委托，下面来看看委托的运作方式。内建的 Cocoa 类有一个通常叫作 delegate 的实例变量（名字中当然会有 delegate 了）。对于 Cocoa 类的某些实例来说，你会将该实例变量的值设为你自己的类的实例。在运行中的某些时刻，Cocoa 类会通过向其委托发送消息来决定接下来该做什么：如果 Cocoa 类实例发现其委托不为 nil，同时该委托可以接收这个消息，那么 Cocoa 实例就会向其委托发送消息。

回忆一下第 10 章关于协议的介绍，委托大量使用了协议。过去，委托方法列在了 Cocoa 类的文档中，其方法签名通过非正式协议（NSObject 的一个类别）来告知编译器。但现在，类的委托方法通常会列在协议自己的文档中。目前有 70 多个 Cocoa 委托协议，这表明 Cocoa 在大量使用委托。大多数委托方法都是可选的，但有时你会发现有些则是必需的。

## 11.4.1 Cocoa 委托

要想通过委托定制 Cocoa 实例的行为，你需要从一个类开始，这个类需要实现相关的委托协议。当应用运行时，你会将 Cocoa 实例的 delegate 属性（或其他名字）设为你的类一个实例。你可以通过代码完成，也可以在 nib 中完成，方式是将实例的 delegate 插座变量（或其他名字）连接到作为委托的恰当的实例上。除了作为该实例的委托，委托类还可能会做其他一些事情。事实上，委托的一个好处就在于你可以随意在类架构中插入委托代码；委托类型是个协议，因此实际的委托可以是任何类的实例。

在这个简单的示例中，我要确保应用的根视图控制器（一个 UINavigationController）不允许应用旋转，当该视图控制器起作用时，应用界面只能位于纵向。不过 UINavigationController 并不是我定义的类；它是 Cocoa 定义的。我自己的类是个不同的视图控制器，即 UIViewController 子类，它作为 UINavigationController 的孩子。那么这个孩子如何告诉父亲该如何旋转呢？ UINavigationController 有个类型为 UINavigationControllerDelegate（这是个协议）的 delegate 属性。在需要知道该如何旋转时，它会向这个委托发送 navigationControllerSupportedInterfaceOrientations 消息。因此，为了能够对非常早期的生命周期事件作出响应，我的视图控制器会将其自身设为 UINavigationController 的委托。它还实现了 navigationControllerSupportedInterfaceOrientations 方法。问题很快就迎刃而解了：

```
class ViewController : UIViewController, UINavigationControllerDelegate {
    override func viewDidLoad() {
        super.viewDidLoad()
        self.navigationController?.delegate = self
    }
    func navigationControllerSupportedInterfaceOrientations(
        nav: UINavigationController) -> UIInterfaceOrientationMask {
            return .Portrait
    }
}
```

Apple 的共享应用实例 UIApplication.sharedApplication() 有一个委托，它在应用的生命周期中扮演着重要的角色，甚至连 Xcode 应用模版都会自动提供一个，即名为 AppDelegate 的类。第 6 章介绍过如何通过调用 UIApplicationMain 来启动应用，它会实例化 AppDelegate 类，并让该实例成为共享应用实例（已经创建好了）的委托。正如第 10 章所指出的那样，AppDelegate 正式使用了 UIApplicationDelegate 协议，这表示它已经为该角色做好了准备；接下来会向应用委托发送 respondsToSelector:，看看实现了哪些 UIApplicationDelegate 协议方法。然后会向应用委托实例发送消息，让其知晓应用生命周期中的主要事件。这正是 UIApplicationDelegate 协议方法 UIApplicationDelegateOptions: 如此重要的原因所在；它是你的代码可以运行的最早期阶段之一。

UIApplication 委托方法也用作通知。这样，除了应用委托，其他实例也能很便捷地接收到应用生命周期事件（通过注册）。还有其他一些类提供了类似的重复事件；比如，UITableView 的 tableView:didSelectRowAtIndexPath: 委托方法就是通过通知 UITableViewSelectionDidChangeNotification 进行匹配的。

根据约定，很多 Cocoa 委托方法名都会包含情态动词 should、will 或 did。will 消息会在某件事发生前发送给委托；did 消息会在某件事刚刚发生后发送给委托。should 消

息比较特殊：它返回一个 Bool，如果为 true 就做出响应；如果为 false 就不会。文档会列出其默认响应是什么；如果默认响应可以接受，那就无须再实现 should 方法了。

在很多情况下，属性会控制某种总体性行为，而我们可以通过委托方法在运行期根据情况来修改该行为。比如，用户是否可以轻拍状态栏让滚动视图快速滚动到顶部是由滚动视图的 scrollsToTop 属性决定的；不过，即便该属性值为 true，你也可以通过让滚动视图委托的 scrollViewShouldScrollToTop: 返回 false 来针对某种特定的轻拍动作而禁止该行为。

在搜索文档以查找如何收到某种事件的通知时，请确保查看相对应的委托协议（如果有）。你可能想要知道用户什么时候轻拍了 UITextField 开始进行编辑了？这是无法在 UITextField 类文档中找到的；你需要查看的是 UITextFieldDelegate 协议的 textFieldDidBeginEditing:，诸如此类。

## 11.4.2 实现委托

对于 Cocoa 中的委托来说，其职责是由协议来描述的，这种模式值得你在编写代码时效仿。你需要通过实践来掌握这种模式，并且要多花时间，不过它通常都是正确的解决方案，因为它会恰当地将知识与职责分配给相关的各种对象。

我们来考虑一个实际的情况。在我开发的一个应用中有一个视图控制器，其视图包含了 3 个滑块，用户可以移动滑块来选择颜色。此外，该视图控制器是 UIViewController 的子类，名字是 ColorPickerController。当用户轻拍 Done 或 Cancel 时，视图会隐藏起来；不过首先，用于展现该视图的代码需要知道用户选择了哪个颜色。因此，我需要从 ColorPickerController 实例向展现该视图的实例发送一条消息。

下面是个消息声明，ColorPickerController 在销毁前会发送这条消息：

```
func colorPicker (picker:ColorPickerController,
    didSetColorNamed theName:String?,
    toColor theColor:UIColor?)
```

问题在于：应该在哪里以及如何声明这个方法呢？

现在，我知道应用中实际用于呈现 ColorPickerController: 的实例所对应的类，那就是 SettingsController。因此，我可以在 SettingsController 中声明这个方法。不过，如果这么做，那就意味着为了向 SettingsController 发送这条消息，ColorPickerController 必须得知道用于呈现它的视图是 SettingsController。但让 SettingsController 接收消息仅仅是个特例而已；它应该对用于呈现与隐藏 ColorPickerController 的所有类开放，这样才能接收到这条消息。

因此，我们希望 ColorPickerController 本身来声明自己会调用的方法；它可以向某个接收者随意发送消息，无论该接收者所对应的类是什么都如此。这正是协议的用武之地！解决方案就是让 ColorPickerController 定义一个协议，并且将该方法作为协议的一部分；让呈现 ColorPickerController 的类遵循该协议。ColorPickerController 还有一个类型适当的 delegate 属性；这提供了通信的通道，并且告诉编译器发送这条消息是合法的：

```
protocol ColorPickerDelegate : class {
    // color == nil on cancel
    func colorPicker (picker:ColorPickerController,
        didSetColorNamed theName:String?,
        toColor theColor:UIColor?)
    }
    class ColorPickerController : UIViewController {
        weak var delegate: ColorPickerDelegate?
    // ...
}
```

（请参见第 5 章了解这里所用的 weak 特性的含义与原因。）当 SettingsController 实例创建并配置好了 ColorPickerController 实例后，它还会将自身设为 ColorPickerController 的 delegate——它是可以这么做的，因为它使用了协议：

```
extension SettingsController : ColorPickerDelegate {
    func showColorPicker() {
        let colorName = // ...
        let c = // ...
        let cpc = ColorPickerController(colorName:colorName, andColor:c)
        cpc.delegate = self
        self.presentViewController(cpc, animated: true, completion: nil)
    }
    func colorPicker (picker:ColorPickerController,
        didSetColorNamed theName:String?,
        toColor theColor:UIColor?) {
            // ...
    }
}
```

现在，当用户选择颜色时，ColorPickerController 就知道该向谁发送 colorPicker:didSetColorNamed:toColor: 了，就是其委托！编译器也允许这么做，因为委托使用了 ColorPickerDelegate 协议：

```
@IBAction func dismissColorPicker(sender : AnyObject?) { // user tapped Done
    let c : UIColor? = self.color
    self.delegate?.colorPicker(
        self, didSetColorNamed: self.colorName, toColor: c)
}
```

## 11.5 数据源

数据源类似于委托，只不过它的方法提供了供其他对象显示的数据。Cocoa 中带有数据源的类主要有 UITableView、UICollectionView、UIPickerView 与 UIPageViewController。对于每个类来说，数据源必须要正式使用数据源协议并实现必需的方法。

有些初学者对于数据源的必要性感到惊奇。为何表数据不是表的一部分？为何要有一些包含着数据的固定的数据结构？原因在于这种架构违背了一般性。使用数据源可以将显示数据的对象与管理数据的对象分离开来，后者可以自由存储和获取所需的数据（参见第 13 章的模型—视图—控制器）。唯一的要求就是数据源必须能快速提供信息，因为当需要显示数据时会实时地向数据源请求数据。

另一个惊奇之处在于数据源不同于委托。但这又回到一般性问题了；这是一个选项而不是必需的。并没有什么理由限制数据源与委托不能成为同一个对象，大多数时候它们可能都是一样的。实际上，在大多数情况下，数据源方法与委托方法可以密切配合；你可能都意识不到这种差别。

下面这个示例来自于我编写的应用，它实现了 UIPickerView，让用户可以根据自己输入的阶段数（“1 阶段” “2 阶段”等）来配置游戏。前两个是 UIPickerView 数据源方法；第 3 个是 UIPickerView 委托方法。它通过这 3 个方法向选择器视图提供内容。

```
extension NewGameController: UIPickerViewDelegate, UIPickerViewDataSource {
    func numberOfComponentsInPickerView(pickerView: UIPickerView) -> Int {
        return 1
    }
    func pickerView(pickerView: UIPickerView,
        numberOfRowsInComponent component: Int) -> Int {
            return 9
    }
    func pickerView(pickerView: UIPickerView,
        titleForRow row: Int, forComponent component: Int) -> String? {
            return "\(row+1) Stage" + ( row > 0 ? "s" : "")
    }
}
```

## 11.6 动作

所谓动作就是由 UIControl 子类（一个控件）实例发出的一条消息，用于通知你该控件上发生了一个重要的用户事件。UIControl 子类都是非常简单的界面对象，用户可以直接与其交互，比如，按钮（UIButton）和分割控件（UISegmentedControl）等。

重要的用户事件（控件事件）列在了 UIControl 类文档 Constants 中的 UIControlEvents 下。不同控件实现了不同的控件事件；比如，分割控件的 Value Changed 事件表示用户轻拍并选择了不同的分段，按钮的 Touch Up Inside 事件则表示用户轻拍了按钮。控件事件本身并没有外在效果；控件会形象地做出响应（比如，被轻拍的按钮看起来就像按下去了一样），不过它并不会自动共享事件发生的信息。如果想知道一个控件事件是何时发生的以便能够在代码中对其做出响应，你就需要让该控件事件触发一条动作消息。

下面是其运作方式。控件会维护一个内部分发表：对于每个控件事件来说都可以有任意数量的目标－动作对，其中动作是个消息选择器（即方法名），目标则是消息将会发送到的对象。当控件事件发生时，控件会查询其分发表，寻找与该控件事件相关的所有目标－动作对，并将每一条动作消息发送给相应的目标（如图 11-1 所示）。

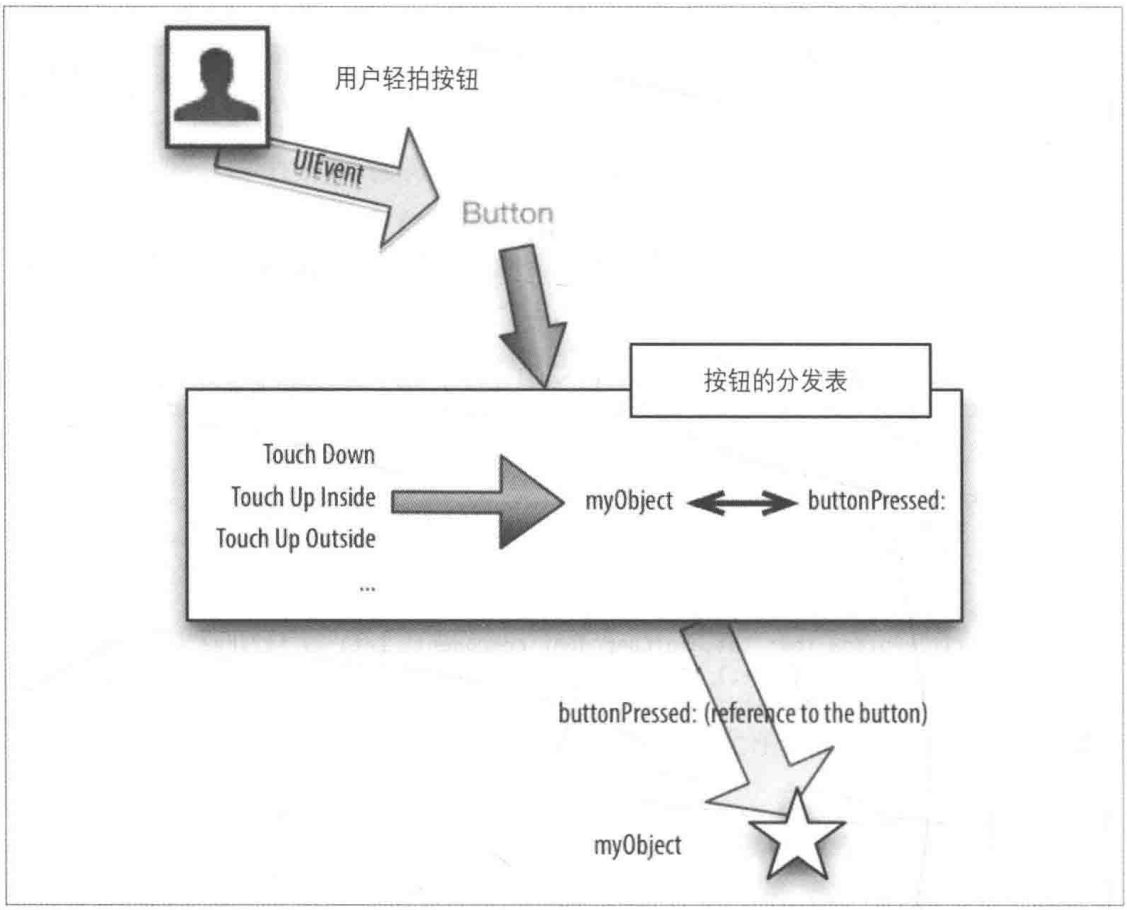


图 11-1：目标－动作架构

有两种方式可以操纵控件的动作分发表：

*动作连接*

可以在 nib 中配置动作连接。第 7 章曾介绍过如何做到这一点，不过并未介绍底层

机制。现在一切都很明显了：nib 编辑器中形成的动作连接是配置控件动作分发表的一种可视化方式。

*代码*

可以通过代码直接操纵控件的动作分发表。这里所用的关键方法是 UIControl 的实例方法 `addTarget:action:forControlEvents:`，其中 `target:` 是个对象，`action:` 是个选择器（在 Swift 中是个字符串），`controlEvents:` 是通过位掩码指定的。与通知中心不同，控件还提供了用于内省分发表的方法。

对于 UIControl 的 `target:` 与关于 NSNotifications 的 `selector:` 也要小心。当控件事件触发时，目标一定要存在才行，它要有一个与动作选择器相对应的方法，Objective-C 必须要能调用该方法。否则就会出问题。

回忆一下第 7 章介绍的关于控件与动作的示例。我们有一个 `buttonPressed:` 方法：

```
@IBAction func buttonPressed(sender:AnyObject) {
    let alert = UIAlertController(
        title: "Howdy!", message: "You tapped me!", preferredStyle: .Alert)
    alert.addAction(
        UIAlertAction(title: "OK", style: .Cancel, handler: nil))
    self.presentViewController(alert, animated: true, completion: nil)
}
```

该方法的目的在于当用户轻拍了界面上的某个按钮时它会被调用。在第 7 章中，我们通过在 nib 中创建了一个动作连接来做到这一点：将按钮的 Touch Up Inside 事件连接到了 ViewController 的 `buttonPressed:` 方法。实际上，我们构造了一个目标－动作对，并将该目标－动作对添加到了按钮的 Touch Up Inside 控件事件分发表中。

相对于在 nib 中进行操作，我们可以通过代码达成所愿。假设我们从来没有绘制过这个动作连接；相反，我们有一个从视图控制器到按钮的名为 `button` 的插座变量连接。当 nib 加载后，视图控制器就可以像如下代码一样配置按钮的分发表：

```
self.button.addTarget(self,
    action: "buttonPressed:",
    forControlEvents: .TouchUpInside)
```

一个控件事件可以有多个目标－动作对。你可能有意这么配置，但也有可能无意而为之。不小心为一个控件事件指定一个目标－动作对但又没有移除现有的目标－动作对是非常容易犯的一个错误，这会导致一些非常奇怪的行为。比如，如果在 nib 中构造了一个动作连接并且又通过代码配置了分发表，那么轻拍按钮就会导致 `buttonPressed:` 被调用两次。

动作选择器的签名可以是如下 3 种形式之一：

- 完整形式接收两个参数：
  - 控件，通常是 AnyObject 类型。
  - 生成控件事件的 UIEvent。
- 一种简写形式，也是最常使用的一种形式，省略了第 2 个参数。buttonPressed: 就是个例子；它只接收一个参数 sender。当 buttonPressed: 通过来自于按钮的动作消息被调用时，sender 就是对该按钮的引用。
- 还有一种简写形式，它会将这两个参数全部省略。

UIEvent 是什么，作用又是什么呢？当用户使用手指进行操作时就会生成触摸事件（轻拍屏幕、移动手指、将手指从屏幕移开）。UIEvent 是最底层的对象，用于实现触摸事件与应用之间的通信。UIEvent 基本上就是个时间戳（一个 Double）外加上一个触摸事件（UITouch）的集合（Set）。动作机制对你屏蔽了触摸事件的复杂性，不过通过接收 UIEvent，你依然可以处理这些复杂性。

奇怪的是，没有一个动作选择器参数提供了一种方式来获悉哪个控件事件触发了当前动作选择器的调用！比如，要想区分 Touch Up Inside 控件事件与 Touch Up Outside 控件事件，其相应的目标－动作对就必须指定两个不同的动作处理器；如果将其分发给相同的动作处理器，那么该处理器就无法判断发生的是哪个控件事件了。

## 11.7 响应器链

响应器是个知道如何直接接收 UIEvent 的对象（参见 11.6 节内容）。它之所以知道是因为它是 UIResponder 或 UIResponder 子类的实例。查看 Cocoa 类的继承体系，你会发现与屏幕显示相关的任何类都是个响应器。UIView 是个响应器、UIWindow 是个响应器、UIViewController 是个响应器，甚至连 UIApplication 与应用委托也是个响应器。

UIResponder 有 4 个底层方法来接收与触摸相关的 UIEvent：

- touchesBegan:withEvent:
- touchesMoved:withEvent:
- touchesEnded:withEvent:
- touchesCancelled:withEvent:

这些方法（touch 方法）会被调用以通知某个触摸事件的响应器。无论代码最终以何种

方式获悉某个用户相关的触摸事件，实际上，即使代码并不知晓某个触摸事件（因为Cocoa以某种自动化的方式对触摸进行响应而无需你的代码介入），该触摸最初也会通过这4个方法中的其中一个来告知响应器。

该通信机制首先会确定用户触摸了哪个响应器。当找到了正确的视图后（单击测试视图），UIView的方法`hitTest:withEvent:`与`pointInside:withEvent:`就会得到调用。然后会调用UIApplication的`sendEvent:`方法，它又会调用UIWindow的`sendEvent:`，而它又会调用点击测试视图（响应器）正确的触摸方法。

应用中的响应器会加入响应器链中，响应器链本质上会沿着视图层次体系将响应器链接起来。一个UIView可能位于另一个UIView中（其父视图）等，直到到达了应用的UIWindow（它没有父视图）。响应器链从下向上的样子如下所示：

1. 起始的UIView（这里指的就是单击测试视图）。
2. 如果该UIView是UIViewController的视图，那么就是该UIViewController。
3. UIView的父视图。
4. 重复第2步！不断重复，直到到达.....
5. UIWindow。
6. UIApplication。
7. UIApplication的委托。

## 11.7.1 推迟职责

我们可以使用响应器链推迟一个响应器对某个触摸事件的处理。如果响应器接收到某个触摸事件但却不能对其进行处理，那么该事件就会沿着响应器链向上查找能够处理的响应器。这主要发生在如下两种情况中：

- 响应器没有实现相关的触摸方法。
- 响应器实现了相关的触摸方法，调用的是`super`。

比如，基本的UIView本身并没有实现触摸方法。这样，在默认情况下，虽然UIView是个单击测试视图，但触摸事件却无法进入UIView中，它会沿着响应器链向上查找能够对其响应的响应器。在某些情况下，将对这种触摸的处理推迟到主背景视图，甚至是控制它的UIViewController中是合情合理的。

下面要介绍的这个应用是我开发的。它是个简单的拼图游戏：一个矩形图片被划分为多个小块，并且被打乱了。用户需要轻拍连续的两个小块来交换它们的位置。其背景视图

是个名为 Board 的 UIView 子类；每个小块都是普通的 UIView 对象，并且是 Board 的子类。当用户轻拍 Board 时，我们需要知道哪个块会做出响应以及拼图的总体布局，不需要每一小块包含任何轻拍检测逻辑。因此，我利用响应器链来推迟职责：每一小块都没有实现任何触摸方法，对每一小块的轻拍会直接落到 Board 上，它会进行触摸检测并处理轻拍事件，并且告诉被轻拍的小块应该做什么。当然，用户对此一无所知；从表面上看，你触摸的是每一个小块，并且由它作出了响应。

## 11.7.2 Nil-Targeted 动作

所谓 *nil-targeted* 动作就是个目标－动作对，其中的目标为 `nil`。由于并没有指定目标动作，因此使用下面的规则：从单击测试视图（用户与之交互的视图）开始，Cocoa 会沿着响应器链向上查找（一次查找一个响应器）能够响应该动作消息的对象：

- 如果找到了能够处理该消息的响应器，那就会调用该响应器的方法，流程结束。
- 如果一直到响应器链的顶部也找不到能够处理该消息的响应器，那么消息就不会得到处理（也没有任何副作用），换言之，什么都不会发生。

假设我们要通过代码来配置一个按钮，如下所示：

```
self.button.addTarget(nil,
    action: "buttonPressed:",
    forControlEvents: .TouchUpInside)
```

这是个 nil-targeted 动作。当用户轻拍按钮时会发生什么事情呢？首先，Cocoa 会查看 UIButton 本身，看看它能否响应 `buttonPressed:`。如果不能，那么它会查找其父视图 UIView，诸如此类，一直沿着响应器链向上查找。对于包含了按钮的视图来说，如果 `self` 是包含了这个视图的视图控制器，并且该视图控制器所对应的类实现了 `buttonPressed:`，那么轻拍按钮就会导致视图控制器的 `buttonPressed:` 被调用！

通过代码来构建 nil-targeted 动作是显而易见的事情：创建一个目标－动作对，其中目标为 nil，就像上一个示例那样。不过，如何在 nib 中构建 nil-targeted 动作呢？答案就是：构造一个对 First Responder 代理对象的连接。这正是 First Responder 代理对象的作用！First Responder 并不是某个已知类的真实对象，因此在将动作连接到它之前，你需要在 First Responder 代理对象中定义动作消息，如下所示：

1. 选中 nib 中的 First Responder 代理，并切换至属性查看器。
2. 你会看到一个用户定义的 nil-targeted First Responder 动作表格（可能为空）。单击 + 按钮，为新动作指定一个签名；它必须接收一个参数（这样其名字会以一个冒号结尾）。

3. 现在可以按住 Control 键并将控件（如 UIButton）拖曳到 First Responder 代理上使用指定的签名来定义一个 nil-targeted 动作。

## 11.8 键值观测

键值观测（KVO）是一种不使用 NSNotificationCenter 的通知机制。一个对象可以通过 KVO 直接注册到第 2 个对象上，当第 2 个对象中的值发生变化时，第 1 个对象就会收到通知。此外，第 2 个对象（被观察的对象）不必做任何额外的事情，它甚至都意识不到注册已经发生了。当被观测对象中的值发生了变化，注册对象（观察者）就会自动收到通知（也许更好的一个架构上的类比就是目标－动作机制；这是一种介于任意两个对象之间的目标－动作机制）。

在使用 KVO 时，观察者就是你自己的对象；当观察者接收到被观察者改变的通知时，你需要编写代码进行响应。不过，被观察的对象（注册到其上以监听变化的对象）无需是你自己的对象；实际上，通常情况下它也不是你自己的对象。很多 Cocoa 对象的行为都是 KVO 形式的，你可以对其使用 KVO。一般来说，KVO 主要用于替代委托与通知。

KVO 的使用可以划分为如下 3 个阶段：

*注册*

要想监听被观察对象中某个值的变化，你需要注册到被观察对象上。这通常需要调用被观察对象的 `addObserver:forKeyPath:options:context:` 方法（所有继承自 NSObject 的对象都有这个方法，因为它是通过非正式协议 NSKeyValueObserving 注入 NSObject 中的，而 NSKeyValueObserving 则是 NSObject 与其他类上的一组类别）。

*变化*

变化发生在被观察对象中的值上，而且方式比较特别，即必须以 KVO 兼容的形式。通常，这意味着要使用键值编码兼容的访问器来作出改变。通过键值编码兼容的访问器来设置属性。

*通知*

当被观察对象中的值发生变化时，观察者会自动收到通知：其 `observeValueForKeyPath:ofObject:change:context:` 方法（我们已经针对这个目的实现了该方法）会在运行期得到调用。

如果不想再接收通知了，那就需要取消对被观察对象的注册，这是通过向其发送 `removeObserver:forKeyPath:`（或 `removeObserver:forKeyPath:context:`）来实现的。这是非常重要的，原因与取消对 NSNotification 的注册相同：如果不取消注册，那么当通知发送给了已经不存在的观察者时，应用就会崩溃。你需要显式取消观察者所注册的

每个键路径；不能将 nil 作为第 2 个参数来表示“所有键路径”。取消注册的最后一个机会是观察者的 deinit；显然，这要求观察者拥有对被观察对象的引用。

事情还没有结束。在被观察对象销毁前，所有的观察者都必须要显式取消对其的注册！如果对象销毁了，但观察者并没有取消注册，那么应用就会崩溃，同时控制台会打印出一条消息：“An instance ... was deallocated while key value observers were still registered with it.”

如下示例来自于我自己的代码。AVPlayerViewController 是个视图控制器，其视图用于显示视频内容。当该视图首次出现时会闪一下，因为视图是黑色的，到视频内容出现前中间会有一点延时。解决办法就是一开始让视图不可见，直到视频内容出现后才让其可见。这样，我们希望当视频内容出现后能够收到通知。AVPlayerViewController 有个 readyForDisplay 属性，我们希望该属性变为 true 时能够收到通知。不过，AVPlayerViewController 并没有委托，也没有提供通知。那么，解决之道就是使用 KVO：将自身注册到 AVPlayerViewController 上，监听其 readyForDisplay 属性的变化。如下代码展示了如何配置并呈现 AVPlayerViewController 的视图：

```
func setUpChild() {
    // ...
    let av = AVPlayerViewController()
    av.player = player
    av.view.frame = CGRectMake(10,10,300,200)
    av.view.hidden = true // looks nicer if we don't show until ready
    av.addObserver(self,
        forKeyPath: "readyForDisplay", options: [], context: nil) ❶
    // ...
}
override func observeValueForKeyPath(keyPath: String?,
    ofObject object: AnyObject?, change: [String : AnyObject]?,
    context: UnsafeMutablePointer<()>) { ❷
        if keyPath == "readyForDisplay" {
            if let obj = object as? AVPlayerViewController {
                dispatch_async(dispatch_get_main_queue(), {
                    self.finishConstructingInterface(obj)
                })
            }
        }
}
func finishConstructingInterface (vc:AVPlayerViewController) {
    if !vc.readyForDisplay {
        return
    }
    vc.removeObserver(self, forKeyPath:"readyForDisplay") ❸
    vc.view.hidden = false
}
```

❶ AVPlayerViewController 的视图一开始是不可见的（hidden 为 true）。我们注册并监听其 readyForDisplay 属性的变化。

❷ AVPlayerViewController 的 readyForDisplay 属性发生了变化，我们收到了通知，因为 observeValueForKeyPath:... 得到了调用。我们要确保这是个正确的通知；如果是，那么就继续完成界面的构建。注意到被观察对象（AVPlayerViewController）会作为 object 参数传递进来；这不仅有助于识别通知，还可以让我们与该对象通信。对于 observeValueForKeyPath:... 是在哪个线程上调用的是没有任何保证的，因此在做任何会影响界面的事情前我们需要移到主线程外。

❸ 最后检查一次，确保 readyForDisplay 已经从 false 变为了 true，取消注册（我们只需要监听其改变一次）并让视图可见（hidden 为 false）。

options: 参数是个位掩码（NSKeyValueObservingOptions）。该参数可以将改变的属性的新值以 change: 字典的形式发给我们。这样，我们就可以改写代码，将检查 readyForDisplay 是否为 true 的代码移动到 observeValueForKeyPath.... 实现中。现在的注册代码如下所示：

```
av.addObserver(
    self, forKeyPath: "readyForDisplay", options: .New, context: nil)
```

下面是剩余部分的代码；如第 5 章所述，这是一系列 guad 语句：

```
override func observeValueForKeyPath(keyPath: String?,
    ofObject object: AnyObject?, change: [String : AnyObject]?,
    context: UnsafeMutablePointer<()>) {
        guard keyPath == "readyForDisplay" else {return}
        guard let obj = object as? AVPlayerViewController else {return}
        guard let ok = change?[NSKeyValueChangeNewKey] as? Bool else {return}
        guard ok else {return}
        dispatch_async(dispatch_get_main_queue(), {
            self.finishConstructingInterface(obj)
        })
}
func finishConstructingInterface (vc:AVPlayerViewController) {
    vc.removeObserver(self, forKeyPath:"readyForDisplay")
    vc.view.hidden = false
}
```

你可能想知道 addObserver:... 与 observeValueForKeyPath:... 中 context: 参数的含义。基本上，我不建议你使用这个参数，不过无论怎样还是要介绍一下。context: 参数表示传递给 addObserver:... 以及从 observeValueForKeyPath:... 获取的“任何数据”。不过，你需要注意其值，因为其类型是 UnsafeMutablePointer<Void>。这意味着即便运行时持有它，其内存也不是由运行时管理的；你需要通过持有它的一个持久化

引用来管理其内存。通常的做法是使用全局变量（声明在文件顶部的变量）；为了防止任何地方都能访问这个变量，你可以将其声明为 private 的，如以下代码所示：

```
private var con = "ObserveValue"
```

在调用 addObserver:... 时，你会将该变量的地址 &con 作为 context: 参数传递进去。当 observeValueForKeyPath:... 接收到通知时，你可以将 context: 参数作为标识符，将其与 &con 进行比较：

```
override func observeValueForKeyPath(keyPath: String?,
    ofObject object: AnyObject?, change: [String : AnyObject]?,
    context: UnsafeMutablePointer<Void>) {
        if context != &con {
        return // wrong notification
    }
    // ...
}
```

在上述代码中，存储在全局变量中的值是没什么意义的；我们只是将其地址作为标识符而已。如果想要使用存储在全局变量中的值，请将 UnsafeMutablePointer 强制类型转换为另一个 UnsafeMutablePointer 底层类型。接下来就可以将底层值作为 UnsafeMutablePointer 的 memory 属性了。在该示例中，con 是个 String：

```
override func observeValueForKeyPath(keyPath: String?,
    ofObject object: AnyObject?, change: [String : AnyObject]?,
    context: UnsafeMutablePointer<Void>) {
        let c = UnsafeMutablePointer<String>(context)
        let s = c.memory // "ObserveValue"
        // ...
    }
```

键值观测是个很复杂的机制，请查阅 Apple 的 *Key-Value Observing Guide* 了解详细信息（比如，可以观测可变的 NSArray，不过其机制要比之前介绍的更加复杂）。KVO 也有一些令人遗憾的缺点。首先，所有通知都是通过调用同一个方法出现的，而这个方法则会成为瓶颈，这非常遗憾。追踪谁观察了谁，确保观察者与被观察者都有恰当的生命周期并且能够及时取消注册是一件很棘手的事情。不过一般来说，KVO 有助于确保不同对象中的值协调一致；如前所述，Cocoa 中的一些地方希望你使用 KVO。

KVO 中被观察者与观察者都要继承自 NSObject。此外，如果被观察的属性声明在 Swift 中，那就必须将其标记为 dynamic，否则 KVO 将无法使用（原因在于 KVO 通过改写访问器方法来工作；Cocoa 要能进入方法中并修改对象代码才可以，如果属性没有声明为 dynamic，那么这一切是无法实现的）。

## 11.9 事件泥潭

你的代码之所以能运行是因为 Cocoa 发送了事件，而你已经创建好了方法来接收这个事件。Cocoa 会发送大量事件，告诉你用户做了什么事情，通知你应用进入到了生命周期中的哪个阶段及其目标是什么、等待你的输入以便继续。要想接收到监听的事件，你需要通过叫作入口点的方法来达成所愿，所谓入口点指的是这样一些方法：它们拥有正确的名字，位于正确的类中，这样就可以通过事件被 Cocoa 所调用。事实上，很容易就会想到，在很多情况下，一个类中的代码几乎都是入口点。

作为一名 iOS 程序员来说，合理安排这些入口点是面临的主要挑战之一。你知道要做什么，但却不能“想做就做”。你需要划分好应用的功能，使之与 Cocoa 调用你的代码的时间与方式保持一致。在编写自己的代码前，类的框架结构其实已经大致勾画出来了，这是根据要接收的 Cocoa 事件而实现的。

假设一个 iPhone 应用要显示出一个包含了表视图的界面（这种情况其实很常见）。你可能要有一个相应的 UITableViewController 子类；UITableViewController 是个内建的 UIViewController 子类，你所定义的 UITableViewController 子类的实例将会拥有并控制表视图，同时还可能会将这个类作为表视图的数据源与委托。在这个类中，你至少需要实现如下方法：

`initWithCoder:` 或 `initWithNibName:bundle:`

UIViewController 生命周期方法，在这里进行实例初始化。

`viewDidLoad`

UIViewController 生命周期方法，在这里进行视图相关的初始化。

`viewDidAppear:`

UIViewController 生命周期方法，在这里设置一些界面显示后需要使用的状态。比如，如果要注册通知或创建定时器，那么这就是一个很适合的地方。

`viewDidDisappear:`

UIViewController 生命周期方法，这里所做的事情与 `viewDidAppear:` 正好相反。比如，可以在这里取消通知注册，或禁用在 `viewDidAppear:` 中所创建的定时器。

`supportedInterfaceOrientations`

UIViewController 查询方法，在这里指定该视图控制器的主视图可以使用哪些设备方向。

`numberOfSectionsInTableView:`
`tableView:numberOfRowsInSection:`
`tableView:cellForRowAtIndexPath:`

UITableView 数据源查询方法，在这里指定表的内容。

`tableView:didSelectRowAtIndexPath:`

UITableView 委托用户动作方法，在这里对用户轻拍表的一行这一动作进行响应。

`deinit`

Swift 类实例生命周期方法，在这里执行生命结束的清理工作。

假设你使用 `viewDidAppear:` 注册通知并创建了一个定时器。该通知有一个选择器（如果没有使用块），定时器也有一个选择器；因此，你还需要实现这两个选择器所指定的方法。

我们已经有很多方法了，其存在的目的只是作为样板代码而已。它们并不是你定义的方法；你也永远不会调用它们。它们是 Cocoa 的方法，放在这里就是为了能在应用生命周期的某个恰当时刻对其进行调用。

按照这种方式，程序的逻辑将变得很难理解！我这里并不是要批评 Cocoa，事实上，我们很难想象其他的应用框架是如何工作的；不过，客观上来讲，Cocoa 程序，甚至是你自己编写的程序，在开发时都是难以阅读的，因为包含了大量分离的入口点，每个入口点都有自己存在的意义，并且会在某个时刻被调用，然后这一切从程序的角度来看是非常晦涩的。要想理解我们假设的这个类到底在做什么，你需要知道 `viewDidAppear:` 何时会被调用，它是如何使用的，诸如此类；否则，你就完全无法理解程序的逻辑与行为，更不必说程序的代码含义了。在阅读其他人的代码时，这种痛苦还会加剧（这也是我在第 8 章曾说过示例代码对于初学者来说帮助并不大的原因所在）。

查看一个 iOS 程序的代码（甚至是你自己的代码），当看到那么多在各种情况下会被 Cocoa 自动调用的方法时，我相信你一定会惊呆。然而，经验会告诉你诸如重写的 UIViewController 方法、表视图委托以及数据源方法等。另外，即便经验再多，你也不可能知道某个方法会作为按钮的动作或通过通知被调用。注释是很有用的，我强烈建议你在开发任何 iOS 应用时都要对每个方法进行注释，如果需要，注释还要很详尽，写清楚方法要做的事情，以及在什么情况下会被调用：特别是如果方法是一个入口点，那么谁会调用它。

也许在编写 Cocoa 应用时，最常犯的错误并不是代码本身有 Bug，而是将代码放到了错误的地方。代码没有运行、在错误的时间运行，或运行的顺序不对。我发现在各种在线用户论坛中，这类问题一直都有人在问（下面就是用户常问的一些问题）：

- *在视图出现与按钮呈现其文本之间存在延迟。*

  这是因为你将设置按钮文本的代码放到了 `viewDidAppear:` 中，这太迟了；代码应该更早一些运行，放在 `viewWillAppear:` 中比较合理。

- *我的子视图是通过代码定位的，不过其位置全都错乱了。*

  这是因为你将定位子视图的代码放到了 `viewDidLoad` 中。这太早了；代码应该晚一些在视图的大小确定后再运行。

- *虽然视图控制器的 `supportedInterfaceOrientations` 不允许，但视图还是可以旋转。*

  这是因为你在错误的类中实现了 `supportedInterfaceOrientations`；应该在包含了视图控制器的 UINavigationController 中实现（或如本章前面所述，使用委托的 `navigationControllerSupportedInterfaceOrientations`）。

- *我为文本框中的 Value Changed 创建了动作连接，但当用户编辑时，代码并未得到调用。*

  这是因为你连接了错误的控件事件；文本框会发出 Editing Changed 而非 Value Changed 事件。

另外的挑战在于你不可能精确知道入口点何时会被调用。文档会给出概要性的介绍，不过在大多数情况下，对于事件何时会发生，以什么顺序发生并没有保证。你可能觉得某个事件会发生，而且文档也使你相信这个事件会发生，但可能并不会发生。你自己的代码可能会触发一些意料之外的事件。文档可能并没有清楚说明何时会发出通知。Cocoa 中可能还会存在 Bug，导致事件的调用方式与文档不符。你没法看到 Cocoa 源代码，因此也搞不清底层实现细节。因此，我建议在开发应用时，使用原始调试（println 与 NSLog，如第 9 章所示）来分析代码。在测试代码时，请密切关注控制台输出，寻找有意义的消息。你可能会对自己的发现感到惊讶。

## 11.10 延迟执行

你的代码会以响应某个事件执行；不过反过来，代码可能又会触发新的事件或事件链。有时，这会导致一些恶果：应用可能会崩溃，或是 Cocoa 可能不会按照你的要求去做。为了解决这个问题，有时需要暂时跳出 Cocoa 本身的事件链，在继续之前等待一切就绪。

这项技术叫作延迟执行。你告诉 Cocoa 去做某件事情，但不是现在，而是不久的将来，当一切就绪后再做。也许只需要非常短暂的延迟，甚至接近 0 秒钟，只是为了让 Cocoa 完成某些事情，如布局界面。从技术上来说，这是在继续执行代码前让当前的运行循环完成，并展开当前方法的整个调用栈。

在开发 iOS 应用时，使用延迟执行的频率可能会比你想象得要多。随着经验的不断累积，你会有一种感觉，知道何时应该使用延迟执行来解决问题。

在 iOS 编程中，使用延迟执行的主要方式是通过调用 dispatch_after 实现的。它接收一个块（一个函数），表示指定的时间过后会发生什么事情。不过调用 dispatch_after 有点复杂，特别是在 Swift 中，因为要进行大量的类型转换；因此，我编写了一个辅助函数，用于简化以及调用 dispatch_after：

```
func delay(delay:Double, closure:()->()) {
    dispatch_after(
        dispatch_time(
            DISPATCH_TIME_NOW,
            Int64(delay * Double(NSEC_PER_SEC))
        ),
        dispatch_get_main_queue(), closure)
}
```

该辅助函数非常重要，因此我将其粘贴到编写的每个应用的 AppDelegate 类文件顶部。这样使用起来就会方便很多！为了使用它，我需要调用 delay 并传递一个延迟时间（通常是个很短的时间，如 0.1 秒）和一个匿名函数，表示延迟过后要做什么事情。注意，该匿名函数中所要做的事情在不久的将来才会做；你会将自己的代码划分为一行一行的执行序列。这样，延迟执行就是其所在函数中的最后一个调用，并且不返回任何值。

如下示例来自于我所编写的一个应用，用户轻拍表中的一行，我的代码会通过创建并展示一个新的视图控制器进行响应：

```
override func tableView(tableView: UITableView,
    didSelectRowAtIndexPath indexPath: NSIndexPath) {
        let t = TracksViewController(
            mediaItemCollection: self.albums[indexPath.row])
        self.navigationController!.pushViewController(
            t, animated: true)
}
```

但遗憾的是，对 TracksViewController 初始化器 init(mediaItemCollection:) 的调用需要一些时间，这样应用就会停下来，同时表中的这一行会高亮显示；虽然时间很短，但会让用户感到奇怪。为了通过某种动作来掩饰这种延迟，我在用户轻拍表中某一行时，让 UITableViewCell 子类显示一个旋转的活动指示器：

```
override func setSelected(selected: Bool, animated: Bool) {
    if selected {
        self.activityIndicator.startAnimating()
    } else {
        self.activityIndicator.stopAnimating()
    }
    super.setSelected(selected, animated: animated)
}
```

不过还有问题：这个旋转的活动指示器不会出现，也不会旋转。原因在于事件叠加到了一起。直到 UITableView 的委托方法 tableView:didSelectRowAtIndexPath: 完成时才会调用 UITableViewCell 的 setSelected:animated:。不过，我们想要掩盖的延迟是在 tableView:didSelectRowAtIndexPath: 过程中的，整个问题就在于它完成的没那么快。

这时，延迟执行就派上用场了！我重写了 tableView:didSelectRowAtIndexPath:，使之能够立刻完成，这样触发 setSelected:animated: 就会导致活动指示器立刻出现并开始旋转，稍后，我会使用延迟执行来调用 init(mediaItemCollection:)，就在界面恢复原状之时：

```
override func tableView(tableView: UITableView,
    didSelectRowAtIndexPath indexPath: NSIndexPath) {
        delay(0.1) { // let spinner start spinning
            let t = TracksViewController(
                mediaItemCollection: self.albums[indexPath.row])
            self.navigationController!.pushViewController(
                t, animated: true)
        }
}
```

# 第 12 章

# 内存管理

Swift 与 Objective-C 中的类实例都是引用类型（参见 4.4.1 节）。在底层，Swift 与 Objective-C 对于引用类型的内存管理方式本质上是一样的。正如第 5 章所指出的那样，这种内存管理是比较困难的事情。

幸好，Swift 使用了 ARC（自动引用计数），这样就无须显式和分别管理每个引用类型对象的内存了，而曾经在 Objective-C 中是必须要这么做的。归功于 ARC，我们遇到内存管理错误的概率大大降低了，这样就可以将更多的时间放在应用本身上，而非处理内存管理问题。

不过，即便使用 ARC，我们还是有可能会遇到内存管理问题，或是不知不觉中陷入了 Cocoa 的内存管理行为当中。内存管理问题会导致过多的内存占用、应用崩溃以及各种奇怪的行为，甚至在 Swift 中也有可能出现此类问题。Cocoa 内存管理可能会让你感到惊讶万分，因此需要理解并清楚 Cocoa 要做什么。

## 12.1 Cocoa 内存管理的原理

之所以要管理引用类型内存是因为对于引用类型对象的引用只不过是指针而已。被指向的真实对象占据着内存，当该对象产生时，我们需要为其留出这块内存；当该对象销毁时，我们需要显式清理这块内存。当对象实例化时，内存被预留出来，不过该如何清理内存，应该什么时候清理呢？

至少，当一个对象不再有其他对象指向它时，那么这个对象就应该被销毁。没有指针指向的对象是无用的对象；它会占据着内存，不过没有其他对象可以引用它。这叫作内存泄漏。很多计算机语言都是通过一种叫作垃圾收集的策略来解决这个问题的。通过周

期性地沿着对象链进行清理，并销毁那些没有指针存在的对象来防止内存泄漏。不过在iOS设备上，垃圾收集是一种代价高昂的策略，其内存非常有限，处理器相对来说也比较慢（可能只有一个核心）。这样，我们就需要手工管理iOS中的内存，当不再需要某个对象时要能精确地销毁它。

上面所说的困难之处在于“精确”。对象销毁不能过早，也不能太迟。多个对象可能都会持有相同对象的指针（引用）。如果对象Manny与Moe都拥有指向对象Jack的指针，并且Manny通过某种方式告诉Jack现在就要销毁，那么可怜的Moe就会持有一个什么都不指向的指针（更糟糕的是指向了垃圾）。如果指针所指向的对象在指针不知情的情况下被销毁了，那么这个指针就叫作野指针。如果Moe随后通过该野指针向它认为还存在的对象发送了消息，那么应用就会崩溃。

为了防止野指针与内存泄漏的出现，有一种基于数字的手工内存管理策略，这个数字由每个引用类型的对象所维护，叫作其保持计数。原则就是其他对象可以增加或减少一个对象的保持计数，并且其他对象只能做这两件事情。只要对象的保持计数为正数，那么对象就会存在。其他对象都无法销毁另一个对象；相反，当对象的保持计数降为0时，它就会被自动销毁。

根据该策略，需要让Jack一直存在的每个对象都应该增加Jack的保持计数，当不需要Jack存在时则需要将其保持计数减1。只要所有对象都能很好地遵循这个策略，那么手工内存管理的问题就会迎刃而解：

- 不会再有任何野指针，因为指向Jack的任何对象都会增加Jack的保持计数，这确保Jack会一直存在。
- 不会再有内存泄漏，因为不需要Jack的任何对象都会减少Jack的保持计数，这确保Jack最终会被销毁（保持计数为0表示不再有对象需要Jack了）。

## 12.2 Cocoa内存管理的原则

如果一个对象能够遵循一些简单且明确的原则，符合内存管理的基本概念，那么它在内存管理上就不会出现什么问题。其本质是如果某个对象拥有对另外一个引用类型对象的引用，那么它只会负责自己那一部分的内存管理工作，这符合上述原则。如果拥有该引用类型对象引用的所有对象都能按照这些原则行事，那么该对象的内存就会被正确地管理，并且在不需要时被精确地销毁。

考虑这3个对象：Manny、Moe与Jack。Jack是我们的目标对象：我们来管理他的内存，如果Jack的内存被正确管理，那么它就会被正确地销毁。Manny与Moe会参与到Jack

内存的管理工作中。他们是如何做到这一点的呢？只要 Manny 与 Moe 遵循如下原则，那就会万事大吉：

- 如果 Manny 或 Moe 显式实例化了 Jack（通过直接调用初始化器），那么该初始化器就会增加 Jack 的保持计数。
- 如果 Manny 或 Moe 创建了 Jack 的一个副本（通过调用 `copy`、`copyWithZone:`、`mutableCopy` 或其他名字中带有 `copy` 的方法），那么复制方法就会增加这个新创建的 Jack 副本的保持计数值。
- 如果 Manny 或 Moe 获得了对 Jack 的引用（不是通过显式的实例化或复制），并且要求 Jack 一直存在（比如，可以通过代码使用 Jack，或让 Jack 作为一个实例属性值），那么他本身就会增加 Jack 的保持计数（这叫作保持 Jack）。
- 如果只有 Manny 或 Moe 自身做了上面这些事情（即 Manny 或 Moe 直接或间接地导致 Jack 的保持计数增加了），那么当他自身不再需要引用 Jack 时，在释放对其的引用前，他会减少 Jack 的保持计数，从而平衡之前对保持计数的增加值（这叫作释放 Jack）。释放掉 Jack 后，Manny 与 Moe 就会认为 Jack 已经不复存在了，因为如果这导致 Jack 的保持计数归 0，那么 Jack 就不复存在了。这是内存管理的黄金法则，这个原则会让内存管理一致且正确地工作。

理解内存管理黄金法则的一般做法是从所有权角度进行思考。如果 Manny 创建、复制或保持了 Jack（也就是说，Manny 增加了 Jack 的保持计数），那么 Manny 就宣称了对 Jack 的所有权。Manny 与 Moe 可以同时拥有对 Jack 的所有权，不过每个人都只负责正确管理自己对 Jack 的所有权。最终减少 Jack 的保持计数是 Jack 的每个所有者的职责，释放 Jack，从而释放了对 Jack 的所有权。拥有者会说："在这儿之后，Jack 可能存在，也可能不复存在，不过对于我来说，我已经使用完了 Jack，就我而言，Jack 已经销毁了。"与此同时，非 Jack 的所有者永远也不会释放 Jack。只要所有对象都是这样处理 Jack 的，那么 Jack 就永远不会出现内存泄漏，指向 Jack 的指针也不会变成野指针。

## 12.3 ARC 及其作用

曾几何时，保持与释放对象是你自己的事情，程序员需要向对象发送 `retain` 与 `release` 消息。NSObject 还实现了 `retain` 与 `release`，不过在 ARC 下（以及在 Swift 中），你不能再调用它们了。这是因为 ARC 会替你调用！这是 ARC 的职责：帮你完成本应该由程序员自己完成的内存管理工作。

ARC 是编译器的一部分。编译器会在背后插入 `retain` 与 `release` 调用来修改你的代码。比如，当通过调用某个方法接收到了一个引用类型的对象时，ARC 会立刻保持它，这样

在代码运行时对象就会一直存在；当代码执行完毕时，ARC 就会释放对象。与之类似，在创建或复制一个引用类型的对象时，ARC 会增加其保持计数，当代码执行完毕时会释放它。

ARC 很保守，但却非常精确。实际上，ARC 会在每个结合处保持计数（可能很多人并没有注意到这里也需要进行内存管理）：当接收到对象作为参数时它会保持计数、在将对象赋给变量时它会保持计数，诸如此类。它甚至还会在背后插入临时变量，使其能够尽早指向对象，从而可以保持它。当然，最终它还会释放以与保持相匹配。

## 12.4 Cocoa 对象管理内存的方式

如果需要，那么内建的Cocoa对象会通过保持来获得你传递给它们的对象的所有权，当然，接下来会通过释放来平衡之前的保持。实际上，这是非常普遍的情况；如果 Cocoa 对象没有保持你传递给它的对象，那么文档中会有相应的说明。

集合（如 NSArray 或 NSDictionary）就是个显而易见的示例（参见第 10 章关于常见集合类的介绍）。如果一个对象可以在任意时刻销毁，那么它几乎无法成为集合的元素；因此，在向集合中添加元素时，集合会通过保持来声明对该对象的所有权。接下来，集合就成为一个功能良好的所有者。如果是可变集合，并且其中的元素被删除了，那么集合就会释放该元素。如果集合对象销毁了，那么它会释放其中的所有元素。

在 ARC 之前，从可变集合中删除对象存在一个潜在的陷阱。考虑如下 Objective-C 代码：

```
id obj = myMutableArray[0];
[myMutableArray removeObjectAtIndex: 0]; // bad idea in non-ARC code!
// ... could crash here by referring to obj ...,
```

如前所述，在从可变集合中删除对象时，集合会释放它。因此，上述示例中被注释的一行涉及对 myMutableArray 中元素 0 对象的隐式释放。如果将对象的保持计数减为 0，那么它就会被销毁。指针 obj 就会变成一个野指针，在将其当作实际对象使用时会导致应用崩溃。

不过在 ARC 中，这种危险情况已经不复存在。将一个引用类型的对象赋给一个变量时会保持它！这样，代码就变得安全了，下面是与之等价的 Swift 代码：

```
let obj = myMutableArray[0]
myMutableArray.removeObjectAtIndex(0)
// ... safe to refer to obj ...
```

第 1 行会保持对象，第 2 行会释放对象，不过这个释放会平衡掉之前将对象放到集合中时对该对象的保持。这样，对象的保持计数依旧大于 0，它会在代码执行期间继续存活。

# 12.5 自动释放池

当一个方法创建了一个实例并将其返回时，一些内存管理技巧就要派上用场了。比如，考虑如下简单代码：

```
func makeImage() -> UIImage? {
    if let im = UIImage(named:"myImage") {
        return im
    }
    return nil
}
```

思考一下返回的UIImage类型的im的保持计数。调用UIImage的初始化器UIImage(named:)会增加其保持计数。根据内存管理的黄金法则，通过函数返回让im脱离我们自己的控制时，我们应该减少它的保持计数，从而平衡之前的增加并交出所有权。不过应该什么时候做呢？如果在return im这一行之前做，那么im的保持计数就会为0，它将被销毁；函数将会返回一个野指针。不过也不能在return im这一行之后做，因为当这行代码执行时，函数代码就宣布执行完毕了。

显然，我们需要通过一种方式来返回这个对象，现在不会减少其保持计数（这样在调用者接收并处理它时，它就会一直存在），同时又要确保在未来的某一时刻我们可以减少其保持计数，从而平衡对其的init(named:)调用，并实现对该对象内存的管理。解决之道就是介于释放对象与不释放对象之间的一种策略，即ARC会自动释放它。

下面来介绍一下自动释放的工作原理。你的代码运行时会有一个自动释放池存在。当ARC自动释放对象时，该对象会被放到自动释放池当中，并且一个数字会增加，这个数字表示该对象被放到这个自动释放池当中的次数。时不时地，当没有其他事情发生时，自动释放池会被自动清空。这意味着自动释放池会释放其中的每一个对象、清除对象被添加到自动释放池中的次数，并清空所有对象。如果这导致对象的保持计数变为0，那么对象就会像通常那样被销毁。因此，自动释放一个对象就好比是释放它，但带有一个附加条款，即“稍后再释放，而不是此时此刻”。

一般来说，自动释放与自动释放池只不过是一种实现细节而已。你看不到其实现；它们只是ARC工作过程的一部分而已。那我为何还要介绍它们呢？这是因为，有时（非常少见）你想要自己来清空自动释放池。考虑如下代码（代码是我编造出来的，因为演示清空自动释放池并不是那么容易）：

```
func test() {
    let path = NSBundle.mainBundle().pathForResource("001", ofType: "png")!
    for j in 0 ..< 50 {
        for i in 0 ..< 100 {
            let im = UIImage(contentsOfFile: path)
```

```
            }
        }
    }
```

该方法所做的事情并没有什么实际意义；它会加载一张图片，不过是在一个循环中重复加载。循环运行时，内存占用量在持续攀升（如图 12-1 所示）；当方法执行完毕时，应用的内存使用量已经达到了约 34MB。这并不是因为每次循环遍历时没有释放图片，而是因为存在着大量的中间对象（一些你从来没听说过的对象，如 NSPathStore2 对象），它们都是在调用 `init(contentsOfFile:)` 时生成的，它们会被自动释放，因此都在那儿等着，导致自动释放池中的对象越来越多，它们在等待自动释放池被清空。当代码执行完毕时，自动释放池会被清空，内存使用量会迅速向下跌落，直到基本没有多少内存被使用。

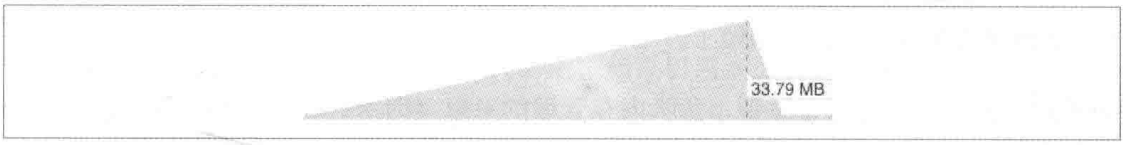


图 12-1：循环中的内存使用增长

当然，34MB 的内存并不算大。不过，可以想象的是更加复杂的内部循环会产生更多、更大的自动释放对象，内存使用也会持续增长。这样，要是能手工清空自动释放池，然后在循环过程中不断清空就好了。Swift 提供了这种方式：全局的 `autoreleasepool` 函数，它接收一个参数，这个参数是个匿名函数。在调用匿名函数前会创建一个特殊的临时自动释放池，它用于随后自动释放的对象。当该匿名函数执行完毕时，这个临时自动释放池会被清空并销毁。下面这个方法与之前一样，不过使用了 `autoreleasepool` 调用包装了内部循环：

```
func test() {
    let path = NSBundle.mainBundle().pathForResource("001", ofType: "png")!
    for j in 0 ..< 50 {
        autoreleasepool {
            for i in 0 ..< 100 {
                let im = UIImage(contentsOfFile: path)
            }
        }
    }
}
```

内存使用上的差异是非常明显的：内存占用量稳定在 2MB 以下（如图 12-2 所示）。创建与清空临时的自动释放池可能会有一些成本，因此如果可能，你需要将循环划分为一个外部循环与一个内部循环，如该示例所示，这样每次迭代时就不会再创建和销毁自动释放池了。

图 12-2：使用自动释放池时，内存使用保持在稳定的状态

# 12.6 实例属性的内存管理

在 ARC 之前，管理实例属性（参见第 10 章关于 Objective-C 实例变量的介绍）的内存是 Cocoa 编程中最棘手的困难之一。正确的行为应该是在给属性赋值时保持一个引用类型的对象，在如下两种情况中将其释放：

- 为相同的属性赋予了不同的值。
- 实例属性所在的实例被销毁了。

为了遵循内存管理的黄金法则，负责内存管理的对象（即所有者）显然需要是该实例属性所在的对象。要想确保能够正确地对属性的内存进行管理，唯一的做法就是在该属性的 setter 方法中进行实现。setter 需要释放该属性当前值所对应的那个对象，然后保持赋给该属性的对象。具体细节是很烦琐的（它们要是同一个对象该怎么办），在 ARC 出现之前，程序员很容易出错。当然，内存管理不只这些；为了防止所有者销毁所导致的内存泄漏问题，我们需要实现所有者的 `dealloc` 方法（对应于 Objective-C 的 `deinit`）来释放掉作为属性值而保持的每个对象。

幸好，ARC 对此完全理解，它会帮你正确地管理好实例属性的内存，就像所有变量的内存一样。

这一事实也让我们知道该如何根据需要释放对象，这么做非常有价值，因为一个对象可能会使用大量内存。你不希望对设备的内存造成太大的压力，因此在使用完对象后就需要将其释放。此外，当应用进入后台并挂起时，如果发现它使用了过多的内存，那么 Watchdog 进程就会在后台终止它；因此，当知道应用将要进入后台时，你可能想要释放该对象（第 3 章对此作过介绍）。

你不能（也不可以）显式调用 `release`，因此需要另辟蹊径，所采用的方式应该与 ARC 的设计和行为保持一致。解决办法就是将另外的值（占用较少内存）赋给该属性。这会导致该属性之前的值被释放。常见的做法是将该属性的类型声明为 Optional，即从而简化隐式展开 Optional 等。这意味着可以将 `nil` 赋给它，这么做纯粹是为了释放当前值。

# 12.7 保持循环与弱引用

如第 5 章所述，当两个对象彼此引用时就会陷入保持循环当中，比如，每个对象都是另外一个对象的实例属性值。如果这种情况存在，并且没有其他对象指向这两个对象中的任何一个，那么这两个对象就都不会销毁，因为每个对象的保持计数都大于 0，谁都不会“先迈出一步”并释放另外一个。除了彼此，这两个对象不会再由其他对象所指向，我们也没有任何办法补救，最终这两个对象就会导致内存泄漏。

解决办法就是改变对引用的内存管理方式。在默认情况下，引用都是个持久引用（ARC 称为 *strong* 或 *retain* 引用）；为其赋值会保持被赋的值。在 Swift 中，你可以将引用类型的变量声明为 weak 或 unowned，从而改变内存管理的方式：

weak

: weak 引用会利用到 ARC 特性的强大功能。如果引用是弱引用，那么 ARC 就不会保持赋给它的对象。这看起来很危险，因为对象可能会在我们不知情的情况下销毁，留下一个野指针，后面可能会导致潜在的崩溃风险。不过 ARC 是非常聪明的。它会记录下所有的弱引用以及赋给它们的所有对象。当这样一个对象的保持计数减为 0 时，那就会销毁该对象，ARC 会自动将 nil 赋给该引用，这也是 Swift 中 weak 引用必须是声明为 var 的 Optional 的原因所在，这样 ARC 就可以将 ni 赋给它了。如果能够前后一致地处理好 Optional，那就不会出现任何问题。

unowned

: unowned 引用则完全不同。在将引用标记为 unowned 时，你实际上会告诉 ARC 不要理睬它：为该引用赋值时，ARC 不会做任何内存管理工作。这实际上有些危险，如果被引用的对象销毁了，那就会留下一个野指针，应用就可能会崩溃。除非知道被引用的对象不会销毁，否则就不应该使用 unowned。如果被引用对象的存活时间比引用它的对象长，那么 unowned 就是安全的。因此，unowned 对象应该是单个对象，只被赋值一次，否则引用者将不复存在。

在实际开发中，弱引用常常用于将一个对象连接到其委托上（参见第 11 章）。委托是个独立的实体；通常来说对象都不会将自己声明为其委托的所有者，实际上对象常常属于其委托，而不是委托的所有者。所有权常常是颠倒过来的；对象 A 创建并保持了对象 B，并让自己成为对象 B 的委托。这可能会导致保持循环。因此，大多数委托都应该声明为弱引用：

```
class ColorPickerController : UIViewController {
    weak var delegate: ColorPickerDelegate?
    // ...
}
```

但遗憾的是，持有弱引用的内建 Cocoa 类的属性有时不是 ARC 弱引用（因为这些类太老了，还要保持向后兼容，而 ARC 是比较新的概念）。这种属性会通过关键字 `assign` 声明。比如，AVSpeechSynthesizer 的 `delegate` 属性的声明如下所示：

```
@property(nonatomic, assign, nullable)
    id<AVSpeechSynthesizerDelegate> delegate;
```

在 Swift 中，该声明如下所示：

```
unowned(unsafe) var delegate: AVSpeechSynthesizerDelegate?
```

```
libobjc.A.dylib`objc_retain:
    0x110706d00 <+0>:  xorl   %eax, %eax
    0x110706d02 <+2>:  testq  %rdi, %rdi
    0x110706d05 <+5>:  je     0x110706d0c               ; <+12>
    0x110706d07 <+7>:  jns    0x110706d0d               ; <+13>
    0x110706d09 <+9>:  movq   %rdi, %rax
    0x110706d0c <+12>: retq
    0x110706d0d <+13>: movq   (%rdi), %rax
->  0x110706d10 <+16>: testb  $0x2, 0x20(%rax)   Thread 1: EXC_BAD_ACCESS (code=1, address=0x24)
    0x110706d14 <+20>: jne    0x110706d25               ; <+37>
```

图 12-3：向野指针发送消息造成了崩溃

Swift 中的 `unowned` 与 Objective-C 中的 `assign` 是一个意思；它们都是告诉你这里不会使用 ARC 内存管理。Swift 还会发出 `unsafe` 警告；对于你自己的代码来说，除非安全，否则你是不会使用 `unowned` 的，Cocoa 的 `unowned` 则是存在潜在风险的，你需要格外小心。

即便你的代码使用了 ARC，但如果 Cocoa 代码没有使用，那就表示还可能会出现内存管理问题。诸如 AVSpeechSynthesizer 的 `delegate` 这样的引用最终可能会变成一个野指针（如果该引用所指向的对象销毁了），指向了垃圾。如果你或 Cocoa 通过该引用发送了消息，那么应用就会崩溃，而这常常出现在真正的错误发生很久之后，所以寻找崩溃根源就会变得相当困难。这种崩溃的典型症状是在与内存管理活动交互时出现 `EXC_BAD_ACCESS`（如图 12-3 所示）。（这种情况需要打开 Zombies 进行调试，本章后面将会对此进行介绍。）

防止这种情况出现的责任在于你自己。如果将某个对象赋给了非 ARC 的不安全引用，如 AVSpeechSynthesizer 的 `delegate`，并且该对象会在引用尚存的情况下销毁，那么你就需要将 nil（或其他对象）赋给该引用，从而防止其变成野指针。

# 12.8 值得注意的内存管理情况

如果使用 NSNotificationCenter 注册通知（参见第 11 章），并且使用 `addObserver:selector:name:object:` 注册了通知中心，那就会将某个对象的引用（通常是 `self`）作为第 1 个参数传递给通知中心；通知中心对该对象的引用是个非 ARC 的不安全引用，当该对

象销毁后就会存在风险，因为通知中心可能还会向它所引用的对象发送通知，而它所引用的却是垃圾。这正是要先取消注册的原因所在。这与之前介绍的委托情况是类似的。

如果使用 addObserverForName:object:queue:usingBlock: 注册了通知中心，那么内存管理就会变得更加棘手，因为：

- 从 addObserverForName:object:queue:usingBlock: 调用返回的观察者标识会被通知中心保持，直到你取消其注册。
- 如果观察者标识引用了 self，那么它也有可能通过块（一个函数，可能是匿名函数）保持你（self）。如果这样，那么在将观察者标识从通知中心取消注册前，通知中心都会保持你。这意味着在取消注册前，内存会泄漏。不过，你不能通过 deinit 从通知中心取消注册，因为只要还未取消注册，deinit 就不会被调用。
- 此外，如果还保持了观察者标识，并且观察者标识保持了你，那就会出现保持循环。

这样，使用 addObserverForName:object:queue:usingBlock: 也会导致之前介绍的“匿名函数中弱引用与无主引用”相同的状况。解决办法是一样的：在作为 block: 参数传递的匿名函数中将 self 标记为 weak 或 unowned。

比如，考虑如下代码示例，其中视图控制器注册了通知，并将观察者标识赋给了一个实例属性：

```
var observer : AnyObject!
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    self.observer = NSNotificationCenter.defaultCenter().addObserverForName(
        "woohoo", object:nil, queue:nil) {
            _ in
            self.description;
    }
}
```

我们的最终意图是取消注册观察者；这也是要保持对其的引用的原因所在。自然，我们会在 viewDidDisappear: 中这么做：

```
override func viewDidDisappear(animated: Bool) {
    super.viewDidDisappear(animated)
    NSNotificationCenter.defaultCenter().removeObserver(self.observer)
}
```

上述代码中，观察者取消了注册，但视图控制器本身却泄露了。可以通过 deinit 查看到日志：

```
deinit {
    print("deinit")
}
```

当需要销毁这个视图控制器时（比如，它是个展示用的视图控制器，现在需要隐藏起来），那就不会调用 deinit。这样就有了一个保持循环！最简单的解决办法就是在进入到匿名函数时将 self 标记为 unowned；这么做是安全的，因为 self 的存活时间不会超过匿名函数：

```
self.observer = NSNotificationCenter.defaultCenter().addObserverForName(
    "woohoo", object:nil, queue:nil) {
        [unowned self] _ in // fix the leak
        self.description;
}
```

另一个值得注意的情况就是 NSTimer（参见第 10 章）。NSTimer 类文档说“运行循环会维护着对其定时器的强引用”；接下来又提到了 scheduledTimerWithTimeInterval:target:...，说“定时器会维护着对目标的强引用，直到它变为无效”。这应该引起你的警觉，一定要小心行事！文档实际上在警告你，只要重复定时器没有变成无效状态，那么目标就会被运行循环所保持；要想停止，唯一的方式就是向定时器发送 invalidate 消息（这个问题在非重复定时器身上不会出现，因为对于非重复定时器来说，定时器会在触发后立刻让自身变为无效）。

在调用 scheduledTimerWithTimeInterval:target:... 时，你可能会将 self 作为 target: 参数。这意味着你（self）会被保持，直到将定时器置为无效时它才能被销毁。不能在 deinit 实现中这么做，因为只要定时器还在重复执行，并且没有接收到 invalidate 消息，deinit 就不会被调用。因此，你需要寻找另外一个恰当的时刻来向定时器发送 invalidate 消息。并没有什么万全的办法，你只需找到这样一个恰当的时刻，就是这些。比如，可以在 viewDidAppear: 与 viewWillDisappear: 中做这些事情来平衡定时器的创建与失效：

```
var timer : NSTimer!
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)
    self.timer = NSTimer.scheduledTimerWithTimeInterval(
        1, target: self, selector: "dummy:", userInfo: nil, repeats: true)
    self.timer.tolerance = 0.1
}
func dummy(t:NSTimer) {
    print("timer fired")
}
override func viewDidDisappear(animated: Bool) {
    super.viewDidDisappear(animated)
    self.timer?.invalidate()
}
```

更加灵活的解决办法是使用块来代替重复定时器，这是通过 GCD 做到的。你还是需要未雨绸缪，防止定时器的块保持自身并导致保持循环，就像通知观察者一样；不过，这

是很容易做到的，结果并不会出现保持循环，因此可以在需要时在 deinit 中让定时器变为无效。定时器“对象”是个 dispatch_source_t，通常作为实例属性保持（ARC 会帮你管理，虽然它是个“假”对象）。在“继续”后，定时器会不断地被重复触发，当被释放后它就会停止，这通常是通过将实例属性设为 nil 来实现的。

为了总结这种方式，我创建了一个 CancelableTimer 类，它可作为 NSTimer 的替代者。基本上，它是一个 Swift 闭包与一个 GCD 定时器分发源的组合。其初始化器是 init(once:handler:)。当定时器触发时会调用 handler:。如果 once: 为 false，那么它就是个重复定时器。它有两个方法，分别是 startWithInterval: 与 cancel:

```
class CancelableTimer: NSObject {
    private var q = dispatch_queue_create("timer",nil)
    private var timer : dispatch_source_t!
    private var firsttime = true
    private var once : Bool
    private var handler : () -> ()
    init(once:Bool, handler:()->()) {
        self.once = once
        self.handler = handler
        super.init()
    }
    func startWithInterval(interval:Double) {
        self.firsttime = true
        self.cancel()
        self.timer = dispatch_source_create(
            DISPATCH_SOURCE_TYPE_TIMER,
            0, 0, self.q)
        dispatch_source_set_timer(self.timer,
            dispatch_walltime(nil, 0),
            UInt64(interval * Double(NSEC_PER_SEC)),
            UInt64(0.05 * Double(NSEC_PER_SEC)))
        dispatch_source_set_event_handler(self.timer, {
            if self.firsttime {
                self.firsttime = false
                return
            }
            self.handler()
            if self.once {
                self.cancel()
            }
        })
        dispatch_resume(self.timer)
    }
    func cancel() {
       if self.timer != nil {
           dispatch_source_cancel(timer)
       }
   }
}
```

如下代码展示了如何在视图控制器中使用它；注意到我们可以在 deinit 中取消定时器，前提是 handler: 匿名函数没有保持循环：

```
var timer : CancelableTimer!
override func viewDidLoad() {
    super.viewDidLoad()
    self.timer = CancelableTimer(once: false) {
        [unowned self] in // avoid retain cycle
        self.dummy()
    }
    self.timer.startWithInterval(1)
}
func dummy() {
    print("timer fired")
}
deinit {
    print("deinit")
    self.timer?.cancel()
}
```

内存管理行为值得关注的其他 Cocoa 对象通常都会在文档中进行清晰的说明。比如，UIWebView 文档警告说：“在释放拥有委托的 UIWebView 实例前，你必须先将其 delegate 属性设为 nil”。CAAnimation 对象保持了其委托，这是个例外情况，如果不小心就会陷入麻烦之中。

但遗憾的是，还有一些情况，文档并没有给出关于特殊的内存管理考量的任何警告信息，你有可能陷入保持循环的陷阱当中。这种问题是很难发现的。Cocoa 中的 UIKit Dynamics（UIDynamicBehavior 的动作处理器）与 WebKit（WKWebKit 的 WKScriptMessageHandler）曾给我制造了很多麻烦。

3 个 Foundation 集合类 NSPointerArray、NSHashTable 与 NSMapTable 分别对应于 NSMutableArray、NSMutableSet 与 NSMutableDictionary，只不过其内存管理策略取决于你自己。比如，通过类方法 weakObjectsHashTable 创建的 NSHashTable 会对其元素维护着 ARC 弱引用，这意味着如果其所指向的对象的保持计数减为 0，那么它们就会被 nil 所替代。你需要自己探索这些类的用法，从而防止保持循环。

## 12.9 nib 加载与内存管理

当 nib 加载时，它会实例化其 nib 对象（参见第 7 章）。这些实例化后的对象会发生什么呢？视图会保持其子视图，但顶层对象呢，它们不是任何视图的子视图。答案就是它们不会增加保持计数；如果没有其他对象保持它们，它们就会销毁。

如果不希望这种情况发生（否则为何一开始要加载这个 nib 呢），那就需要捕获到从 nib 中实例化的顶层对象的引用。可以通过两种方式做到这一点。在通过调用 NSBundle 的

loadNibNamed:owner:options: 或 UINib 的 instantiateWithOwner:options: 加载 nib 时会返回一个 NSArray，其中包含了 nib 加载机制所实例化的顶层对象。因此，保持这个 NSArray 或其中的对象即可。

在某些情况下，你可能根本就意识不到发生了这种情况。比如，当一个视图控制器自动从故事板中实例化时，它实际上会从 nib 中加载，并且只有一个顶层对象，即视图控制器本身。因此，该视图控制器就是 instantiateWithOwner:options: 所返回的数组中的唯一一个元素。接下来，视图控制器会被从该数组中提取出来，并由运行时保持，这是通过将其添加到视图控制器层次体系中做到的。

另一种可能是通过插座变量来配置 nib 所有者，该插座变量会在 nib 顶层对象实例化时保持它们。第 7 章曾这么做过，那时创建了一个如下所示的插座变量：

```
class ViewController: UIViewController {
    @IBOutlet var coolview : UIView!
```

接下来手工加载 nib，并将该视图控制器作为所有者：

```
NSBundle.mainBundle().loadNibNamed("View", owner: self, options: nil)
self.view.addSubview(self.coolview)
```

第一行从 nib 中实例化了顶层视图，nib 加载机制会将其赋给 self.coolview。由于 self.coolview 是个强引用，它会保持视图。这样，第 2 行将视图插入界面中时，它会还存在。

当视图控制器加载包含了主视图的 nib 时也会出现相同的情况。视图控制器有一个 view 插座变量，并且是 nib 的所有者。这样，视图会由 nib 加载机制实例化并赋给视图控制器的 view 属性，该属性会保持它。

不过，对于声明的 @IBOutlet 属性，你常常会将其标记为 weak。这并不是必需的，省略 weak 标记也不会出现什么问题。将这种插座变量标记为 weak 也可以正常使用，原因在于你知道该插座变量所指向的对象还会由其他对象保持，比如，它是视图控制器主视图的一个子视图。视图会由其父视图保持，这样除非后面将其从父视图中移除，否则你知道它会一直存在，@IBOutlet 属性没必要再保持它了。

## 12.10 CFTypeRefs 的内存管理

CFTypeRef 纯粹是 C 中与 Objective-C 对象的等价之物。它是指向某个实现“细节未知”的 C 结构体的指针（参见附录 A），“细节未知”指的是该结构体没有可直接访问的组件。这个结构体作为一个假对象；CFTypeRef 等价于对象类型。不过，它并不是对象类型，

处理 CFTypeRef 的代码也不是面向对象的；CFTypeRef 没有属性和方法，你也不能向其发送任何消息。可以完全通过全局函数来使用 CFTypeRefs，它们实际上是 C 函数。

在 Objective-C 中，CFTypeRef 类型的特性是名字以 Ref 后缀结尾。不过在 Swift 中已经去掉了这个 Ref 后缀。

下面是用于绘制渐变色的 Swift 代码：

```
let con = UIGraphicsGetCurrentContext()!
let locs : [CGFloat] = [ 0.0, 0.5, 1.0 ]
let colors : [CGFloat] = [
    0.8, 0.4, // starting color, transparent light gray
    0.1, 0.5, // intermediate color, darker less transparent gray
    0.8, 0.4, // ending color, transparent light gray
]
let sp = CGColorSpaceCreateDeviceGray()
let grad =
    CGGradientCreateWithColorComponents (sp, colors, locs, 3)
CGContextDrawLinearGradient (
    con, grad, CGPointMake(89,0), CGPointMake(111,0), [])
```

在上述代码中，con 是个 CGContextRef，在 Swift 中则是 CGContext；sp 是个 CGColorSpaceRef，在 Swift 中则是 CGColorSpace；grad 是个 CGGradientRef，在 Swift 中则是 CGGradient。它们都是 CFTypeRefs。其代码并不是面向对象的；而是对全局 C 函数的一系列调用。

不过，CFTypeRef 假对象也是个对象。这意味着被指针指向的 C 结构体实际上与引用所指向的类实例是一样的。这也意味着我们需要管理 CFTypeRefs 的内存。特别地，CFTypeRef 假对象也有一个保持计数！其使用方式与真实对象别无二致，也遵循着内存管理的黄金法则。如果其所有者希望它一直存在，那么 CFTypeRef 就必须要保持；当所有者不再需要它时，我们得将其释放。

在 Objective-C 中，对于 CFTypeRefs 使用的黄金法则就是，如果通过一个函数（其名字包含了单词 Create 或 Copy）获得了一个 CFTypeRef 对象，那么其保持计数就会增加。此外，如果担心对象的存活时间，那就需要显式调用 CFRetain 函数增加其保持计数来保持它。要想平衡 Create、Copy 与 CFRetain 调用，最终需要释放对象。在默认情况下，这是通过调用 CFRelease 函数实现的；不过，一些 CFTypeRefs 拥有自己专门的对象释放函数。比如，CGPath 就有一个专门的 CGPathRelease 函数。

不过在 Swift 中，你永远不需要调用 CFRetain 或任何形式的 CFRelease；事实上，你也无法调用。Swift 会在背后自动调用。

可以将 CFTypeRefs 看作存在于两个世界中：纯 C 的 CFTypeRef 世界，以及面向对象内存管理的 Swift 世界。在获取到一个 CFTypeRef 假对象时，它会从 CFTypeRef 世界来到

Swift 世界。从那时起直到使用完，你都需要管理其内存。Swift 知道这一点，大多数情况下，Swift 本身都会使用黄金法则并进行正确的内存管理。这样，上面展示的绘制渐变色的代码实际上就会正确地管理好内存。在 Objective-C 中，我们需要释放 sp 与 grad，因为它们是通过 Create 调用创建的；不这么做就会导致内存泄漏。不过在 Swift 中就没这个必要了，因为它会帮我们做这些事情。

在 Swift 中使用 CFTypeRefs 要比在 Objective-C 中简单。在 Swift 中，你可以将 CFTypeRef 假对象看作真实对象！比如，你可以在 Swift 中将 CFTypeRef 赋给属性，或将其作为参数传递给 Swift 函数，其内存会得到正确的管理；在 Objective-C 中，这些事情都很棘手。

不过，你可能会通过一些缺乏内存管理信息的 API 来接收 CFTypeRef。这样的值应该引起你的注意，因为它会以包装了实际 CFTypeRef 的非托管值的形式进入 Swift 中。这会产生一个警告，因为 Swift 并不知道如何对这个假对象进行内存管理。实际上，只有通过调用 Unmanaged 对象的 takeRetainedValue 或 takeUnretainedValue 方法展开 CFTypeRef 后才能继续。你需要通过这两个方法来告诉 Swift 该如何正确管理这个对象的内存。对于通过名字中带有 Create 或 Copy 的内建函数所返回的 CFTypeRef 来说，请调用 takeRetainedValue；否则请调用 takeUnretainedValue。

## 12.11 属性的内存管理策略

在 Objective-C 中，@property 声明（参见第 10 章）包含了一个内存管理策略语句，后面跟着相应的 setter 访问器方法。意识到这一点并知道如何将这种策略语句转换为 Swift 是很有必要的。

比如，之前曾提到过 UIViewController 会保持其 view（其主视图）。我是怎么知道的呢？这是 @property 声明告诉我的：

```
@property(null_resettable, nonatomic, strong) UIView *view;
```

关键字 strong 表示 setter 会保持接收到的 UIView 对象。这个声明转换为 Swift 后并不会向变量添加任何特性：

```
var view: UIView!
```

在 Swift 中，默认做法是指向引用对象类型的变量是个强引用，即持久化引用。这意味着它会保持对象。这样就可以从这个声明中推断出，UIViewController 会保持其 view。

对于 Cocoa 属性来说，其内存管理策略有：

`strong`, `retain`（*Swift 中没有等价之物*）

默认值。这两个关键字彼此等价；`retain` 源自 ARC 出现之前。为该属性赋值会保持接收到的值并释放现有值。

`copy`（*Swift 中没有等价之物，或* `@NSCopying`）

与 `strong` 和 `retain` 相同，只不过 setter 会通过向其发送 `copy` 复制接收到的值；进来的值必须是使用了 NSCopying 的对象类型，从而确保这么做是可行的。副本（已经增加了保持计数值）会成为新值。

`weak`（*Swift* `weak`）

一个 ARC 弱引用。接收到的值不会被保持，不过如果在我们不知情的情况下销毁了，那么 ARC 会将 `nil` 替换为该属性的值，其类型必须是声明为 `var` 的 Optional。

`assign`（*Swift* `unowned(unsafe)`）

无内存管理。该策略源自 ARC 出现之前，本身就是不安全的（因此，转换为 Swift 后会有额外的 `unsafe` 警告）：如果被引用的对象销毁了，那么该引用就会变成一个野指针，后面如果使用它就会导致应用崩溃。

你可能很想了解关于 `copy` 策略的一些内容，因为之前并没有介绍过。当不变类有可变子类时（比如，NSString 与 NSMutableString，以及 NSArray 与 NSMutableArray；参见第 10 章），Cocoa 就会使用该策略。它解决了 setter 调用者传递可变子类对象的风险。稍微想一下就知道这是可能的，因为根据多态的替换法则（参见第 4 章），在需要一个类的实例时，我们可以传递其子类的实例。不过，这么做可能不太好，因为现在调用者会保持着对传递进来的值的引用，因为它是可变的，因此后面可能会在我们不知情的情况下发生变化。为了防止这种情况的发生，setter 会调用传递进来对象的 `copy`；这会创建新的实例，它与所提供的对象不同，并且属于不可变类。

在 Swift 中，这个问题基本不会出现在字符串与数组上，因为在 Swift 这一边，它们是值类型（结构体），赋值时会被复制，然后才作为参数传递，或作为返回值被接收。这样，Cocoa 的 NSString 与 NSArray 属性声明在转换为 Swift 的 String 与 Array 属性声明时，它们并不需要与 Objective-C copy 对应的任何特殊标记。不过，不会自动从 Swift 结构体桥接的 Cocoa 类型是会显示一个标记的，即 `@NSCopying`。比如，在 Swift 中，UILabel 的 `attributedText` 属性声明如下所示：

```
@NSCopying var attributedText: NSAttributedString?
```

NSAttributedString 有一个可变子类 NSMutableAttributedString。你可能已经将这个特性字符串配置为了 NSMutableAttributedString，现在要将 UILabel 的 `attributedText` 赋给它。UILabel 并不希望你保持一个对该可变字符串的引用并修改它，因为这会在不使用

setter 的情况下改变属性值。这样，它会复制传递进来的值，确保它是一个不同的可变 NSAttributedString。

你也可以在自己的代码中这么做，并且也愿意这么做。如果类中有一个 NSAttributed-String 实例属性，那么可以将其标识为 `@NSCopying`，对于其他的可变 / 不变成员对也是类似的，比如，NSIndexSet、NSParagraphStyle 及 NSURLRequest 等。只提供 `@NSCopying` 标识即可；Swift 会强制应用 copy 策略，并且在为该属性赋值时进行实际的复制动作。

有时，你希望自己的类拥有改变属性值的能力，同时又想防止外部传递进来可变的值，那么就需要在其前面加上一个私有的计算属性门面，它会将其转换为相应的可变类型：

```
class StringDrawer {
    @NSCopying var attributedString : NSAttributedString!
    private var mutableAttributedString : NSMutableAttributedString! {
        get {
            if self.attributedString == nil {return nil}
            return NSMutableAttributedString(
                attributedString:self.attributedString)
        }
        set {
            self.attributedString = newValue
        }
    }
}
```

`@NSCopying` 只能用于类的实例属性，结构体与枚举是不行的，并且只能用在使用了 Foundation 的场合中，这是因为 NSCopying 协议定义在 Foundation 中，标记为 `@NSCopying` 的变量类型都需要使用该协议。

# 12.12 调试内存管理的错误

虽然在 ARC（与 Swift）中出现的概率很低，但内存管理错误还是有可能出现的，程序员会错误地认为这种问题不会发生。经验表明，你应该尽可能使用每一个工具来找出可能的内存管理错误。下面就列出一些工具（参见第 9 章）：

- 在应用运行时，你可以通过调试导航器图表中的内存使用仪表盘来观测可能出现的内存泄漏或其他不太合理的大量内存使用情况。值得注意的是，模拟器中的内存使用不一定会反映出实际情况！当在设备上运行应用时，请密切关注内存使用仪表盘，然后再决定下一步动作。
- Instruments（Product → Profile）提供了很多优秀的工具来检测内存泄漏并追踪每个对象的内存使用情况。

- 原始调试有助于确保对象的行为与预期一致。请通过一个`print`调用来实现`deinit`。如果它没有被调用，那就说明对象并没有销毁。这可能会发现连Instruments都无法直接探测到的问题。
- 野指针是难以追踪的，不过可以通过“打开zombies”定位它们。可以通过Instruments中的Zombies模板轻松做到这一点。此外，还可以编辑方案中的Run动作，切换至Diagnostics页签，然后勾选上Enable Zombie Objects。结果就是不会再有对象销毁；相反，它会被“zombie”所代替，如果向其发送了消息，那么它就会向控制台打印出消息（“message sent to deallocated instance”）。不过，在追踪完野指针后，请记得关闭zombies。不要同时使用zombies与Leaks instrument：zombies会导致内存泄漏。

虽然介绍了这么多工具，但它们也无法解决每一个潜在的内存管理问题。比如，一些对象本身（如包含了一张很大图片的UIView）很小（可能导致内存管理仪表盘或Instruments并不会认为它们使用了大量内存），但却需要很大的一块存储空间；维护太多对这种对象的引用会导致应用很快就被系统杀死。这种问题是很难追踪的。

# 第 13 章

# 对象间通信

随着应用中的对象变得越来越多，如何在对象间进行消息的发送或数据的通信就成为摆在我们面前的一个问题，这本质上还是一个架构问题。需要对代码的构建做些规划，使得代码能够各司其职，并且能在恰当的时刻根据需要共享信息。本章将会介绍一些系统性的思考方法，帮助你实现对象间的通信。

通信的问题常常可以归结为一个对象要能看到另一个对象：对象 Manny 要能找到对象 Jack，这样才能够向 Jack 发送消息。

一个显而易见的解决办法就是将 Jack 作为 Manny 的一个实例属性值。这在 Manny 与 Jack 共享某些职责或彼此互为补充的情况下尤为有用。应用对象及其委托、表视图及其数据源、视图控制器及其所控制的视图，在这些情况中，前者都有一个指向了后者的实例属性。

这并不是说由于内存管理策略的问题（参见第 12 章），Manny 需要声明为 Jack 的所有者，不过它是可以这么做的。对象不一定要保持其委托或数据源；与之类似，实现了目标－动作模式的对象（如 UIControl）并没有保持其目标。通过使用弱引用以及将属性类型声明为 Optional，然后以前后一致且安全的方式来对待这个 Optional，Manny 可以在不拥有 Jack 的情况下，让其假定对 Jack 的引用最终变为 nil。另外，如果没有可控制的视图，那么视图控制器就是毫无意义的；当它有了视图后，它会保持这个视图，只有当视图控制器自身销毁时，它才会释放这个视图。

对象可以在没有彼此引用的情况下进行双向通信。其中一个对象拥有对另一个对象的引用就足够了，因为前者（作为向后者发送的消息的一部分）可以包含对自身的引用。

比如，Manny 可能会向 Jack 发送消息，其中一个参数就是对 Manny 的引用；这仅仅构成了一种身份证明形式，或一种邀请形式，表示 Jack 自己的方法完成处理后，如果还需要进一步的信息，那么他可以向 Manny 发回消息。这样，Manny 可以令自身对 Jack 处于暂时可见的状态；Jack 不应该保持 Manny（因为这显然会导致保持循环的风险）。另外，这还是一种常见的模式。委托消息 `textFieldShouldBeginEditing:` 的参数是个对发送消息的 UITextField 的引用。目标－动作消息的第 1 个参数就是个对发送消息的控件的引用。

不过，Manny 一开始是如何获得对 Jack 的引用的呢？这是个重要的问题。iOS 编程与面向对象编程的很重要的一个话题就是一个对象如何获得其他对象的引用。情况纷繁复杂，需要具体问题具体分析，不过还是存在着一些通用模式，本章就将介绍这些模式。

Manny 可以通过一些方式向 Jack 发送消息，同时又不必直接发送给 Jack，可能他都不知道或不关心谁是 Jack。通知与 KVO 就是这样的，本章也将对其进行介绍。

最后，13.4 节还将介绍这样一个问题：在典型的 iOS 程序中，什么样的对象需要彼此能够看到对方。

# 13.1 实例化可见性

每一个实例都有自己的来源，并且根据需要创建出来：某个对象会发送一条消息，命令创建出一个实例。因此，命令对象在这个时刻就会拥有一个指向该实例的引用。当 Manny 创建 Jack 时，Manny 就会拥有对 Jack 的引用。

这个简单的事实可作为建立未来通信机制的出发点。如果 Manny 创建了 Jack 并且知道未来他需要一个对 Jack 的引用，那么 Manny 就可以将创建 Jack 所返回的引用保存起来。如果 Manny 知道 Jack 在未来需要一个对 Manny 的引用，那么 Manny 就可以在创建 Jack 后立刻向其提供引用，Jack 会将该引用保存起来。

委托就是这样一种情况。Manny 可能会在创建完 Jack 后立刻将自身作为 Jack 的委托，如下面这个来自于第 11 章的示例代码所示：

```
let cpc = ColorPickerController(colorName:colorName, andColor:c)
cpc.delegate = self
```

事实上，如果这很重要，那么你应该为 Jack 声明一个初始化器，这样 Manny 就可以在创建 Jack 后向其传递一个对自身的引用，从而防止任何失误的发生。看看 UIBarButtonItem 所采取的方式，它有 3 个不同的初始化器，比如，`init(title:style:target:action:)`，它们都需要一个 `target` 参数，表示 UIBarButtonItem 发送消息的目标。

当 Manny 创建 Jack 时，Jack 需要的可能不是对 Manny 自身的引用，而是需要 Manny 知道或拥有的某个引用。你可以向 Jack 提供一个方法，这样 Manny 就可以将该信息提供给 Jack 了；另外，如果没有这个信息 Jack 将不复存在，那么将该方法作为 Jack 的初始化器也是合情合理的。

回忆一下第 11 章的这个示例。它来自于一个表视图控制器。用户轻拍了表中的一行。我们又创建了一个表视图控制器 TracksViewController 实例，并将其所需要的数据传递给它，然后展现出这个表视图。我故意让 TracksViewController 拥有一个指定初始化器 `init(mediaItemCollection:)`，这样 TracksViewController 在创建出来到获取所需数据之间就必须使用它：

```
override func tableView(tableView: UITableView,
    didSelectRowAtIndexPath indexPath: NSIndexPath) {
        delay(0.1) { // let spinner start spinning
            let t = TracksViewController(
                mediaItemCollection: self.albums[indexPath.row])
            self.navigationController!.pushViewController(
                t, animated: true)
        }
}
```

在该示例中，`self` 并没有保持对新创建的 TracksViewController 实例的引用，TracksViewController 也不需要对 `self` 的引用。不过，`self` 创建了 TracksViewController 实例，并且在短暂的时刻内拥有对其的引用。因此，`self` 利用这个时刻向 TracksViewController 实例传递了它所需的数据，这是个绝佳的时刻。知道这个时刻，并且充分利用它，这是数据通信的关键所在。

Nib 加载也是一样的。Nib 加载是一种从 nib 中实例化对象的方式。我们需要精心准备以确保存在着对这些对象的引用，这样它们就不会消失不见了（见 12.9 节）。Nib 加载时刻也是 nib 所有者或加载 nib 的代码与这些对象产生联系的时刻；它会充分利用这个时刻来确保引用的安全性。

初学者常感到困惑的是，如果两个对象从不同的 nib 中加载（从不同的 *.xib* 文件中或故事板中的不同场景中），那么它们是如何获得彼此的引用的。令人沮丧的是，你不能在 nib A 中的对象与 nib B 中的对象间绘制连接；更有甚者，你可能看到同一个故事板中的两个对象就在那儿，但却无法连接它们。不过，如前所述（见 7.3.11 节），这种连接是毫无意义的，这也是不能将其连接起来的原因所在。它们是不同的 nib，会在不同的时间加载。不过，当 nib A 加载时，某个对象（Manny）会成为所有者；当 nib B 加载时，会有另外的对象（Jack）成为所有者。也许它们（Manny 与 Jack）会看到彼此，在这种情况下，指定好所有必要的插座变量后，问题就会得以解决。也许还有第 3 个对象（Moe）能够看到 Manny 与 Jack，并为其提供通信路径。

比如，当故事板中的 Segue 被触发时，该 Segue 的目标视图控制器就会被实例化，并且这个 Segue 会持有一个对其的引用。同时，这个 Segue 的源视图控制器已经存在了，它也会持有一个对其的引用。这样，该 Segue 就可以向源视图控制器发送 prepareForSegue:sender: 消息，其中包含了对自身（Segue）的引用。这个 Segue 就是 Moe；它会将 Manny（源视图控制器）与 Jack（目标视图控制器）连接起来。这样，源视图控制器（Manny）就可以获得对新实例化的目标视图控制器（对 Jack 的引用）的引用，这是通过让 Segue 获取来做到的；现在，源视图控制器就可以让自身成为目标视图控制器的委托，并向其传递任何必要的信息，如此种种。

# 13.2 关系可见性

对象可以通过其在所包含的结构中的位置获取自动看到彼此的能力。在考虑如何向一个对象提供另一个对象的引用前，请先看看它们之间是否存在从一个到另一个的引用链。

比如，子视图可以通过其 superview 属性看到其父视图。父视图可以通过其 subviews 属性看到其所有子视图，并且可以通过该子视图的 tag 属性（调用 viewWithTag:）获取到特定的子视图。窗口中的子视图可以通过其 window 属性看到其窗口。这样，通过这些属性并沿着视图层次体系向上或向下查找，一个对象可以获得所需的引用。

与之类似，响应器（参见第 11 章）可以通过 nextResponder 方法看到响应器链中的下一个对象，这意味着，根据响应器链的结构，视图控制器的主视图可以看到视图控制器。如下代码来自于我所编写的一个应用，我从一个视图开始沿着视图层次体系获得了负责整个场景的视图控制器引用（第 5 章也介绍了类似的示例）：

```
var r = sender as! UIResponder
repeat { r = r.nextResponder()! } while !(r is UIViewController)
```

与之类似，视图控制器本身也是层次体系的一部分，因此也可以看到彼此。如果某个视图控制器当前正通过另一个视图控制器展现了一个视图，那么后者就是前者的 presentedViewController，前者是后者的 presentingViewController。如果某个视图控制器是 UINavigationController 的孩子，那么后者就是其 navigationController。UINavigationController 的可见视图是由其 visibleViewController 所控制的。你可以从这些视图中的任何一个通过其 view 属性获得视图控制器的 view，诸如此类。

所有这些关系都是公开的。如果能够获得这些结构或类似结构中的任何一个对象的引用，那么你就可以通过引用链在整个结构中导航，并且可以操纵结构中的任何其他对象。

# 13.3 全局可见性

有些对象是全局可见的，也就是说，它们对所有对象都是可见的。对象类型本身就是个很好的例子。正如我在第 4 章所指出的那样，我们可以在使用 Swift 结构体的同时通过静态成员来提供全局可用的命名空间约束（见 4.3.2 节）。

有时，类会通过类方法来提供单例。这些单例反过来又提供了指向其他对象的属性，这使得其他对象也变成全局可见的了。比如，任何对象都可以通过调用 UIApplication.sharedApplication() 看到单例的 UIApplication 实例。这样，任何对象也都可以看到应用的主窗口，因为它是单例 UIApplication 实例的 keyWindow 属性；任何对象也都可以看到应用委托，因为它是其 delegate 属性。这个链条还会继续：任何对象都可以看到应用的根视图控制器，因为它是主窗口的 rootViewController；正如 13.2 节所述，我们可以从这里导航视图控制器与视图层次体系。

你也可以通过将自己的对象附加到全局可见对象上使其全局可见。比如，你可以自由创建的应用委托的公共属性就是全局可见的，这是因为应用委托是全局可见的（因为共享应用是全局可见的）。

另一个全局可见对象是调用 NSUserDefaults.standardUserDefaults() 所返回的共享默认对象。该对象是个网关，用于存储和获取用户默认值，它像是一个字典（一个值的集合，根据键来获取）。当应用终止时，用户默认值会自动保存；当应用再次启动时，它们又会自动恢复。因此，这是应用在两次启动之间维护信息的一种方式。不过，由于是全局可见的，因此它们还是应用中通信的一种媒介。

比如，在我开发的一个应用中有一个名为 HazyStripy 的设置。它决定了某个可见的界面对象（游戏中的一张纸牌）是模糊的还是条纹的。用户可以修改这个设置，因此会有一个首选项界面让用户修改。当用户打开这个首选项界面时，我会在用户默认值中检查 HazyStripy 设置，配置这个界面以在分割控件中反映出来（叫作 self.hazyStripy）。

```
func setHazyStripy () {
    let hs = NSUserDefaults.standardUserDefaults()
        .objectForKey(Default.HazyStripy) as! Int
    self.hazyStripy.selectedSegmentIndex = hs
}
```

相反，如果用户操作了首选项界面，轻拍 hazyStripy 分割控件来修改其设置，那么我会通过修改用户默认值中实际的 HazyStripy 设置来作出响应：

```
@IBAction func hazyStripyChange(sender:AnyObject) {
    let hs = self.hazyStripy.selectedSegmentIndex
```

```
        NSUserDefaults.standardUserDefaults().setObject(
            hs, forKey: Default.HazyStripy)
    }
```

这里还有一个地方很有意思。首选项界面并非唯一一个使用用户默认值中 HazyStripy 设置的地方；实际绘制模糊或条纹卡片的绘制代码也会用到它，这样才能知道卡片该如何绘制自身！当用户关闭首选项界面，纸牌游戏重新出现时，纸牌会被重新绘制，它会查询 NSUserDefaults 中的 HazyStripy 设置。

```
override func drawRect(rect: CGRect) {
    let hazy : Bool = NSUserDefaults.standardUserDefaults()
        .integerForKey(Default.HazyStripy) == HazyStripy.Hazy.rawValue
    CardPainter.sharedPainter().drawCard(self.card, hazy:hazy)
}
```

这样，纸牌对象与管理首选项界面的视图控制器对象就没必要看到彼此了，因为它们都能看到这个共同的对象，即 HazyStripy 用户默认值。NSUserDefaults 本质上会成为一个全局的媒介，用于实现应用中不同部分之间信息的通信。

# 13.4 通知与 KVO

可以使用通知（参见第 11 章）实现概念上远离的两个对象间的通信，同时两个对象间又不必看到彼此。这两个对象的共同点是它们都要知道通知的名字。每个对象都能看到通知中心（它是个全局的可见对象），因此每个对象都可以发送或接收通知。

以这种方式使用通知看起来有些逃避责任，没有以显而易见的方式来架构你的对象。不过有时，一个对象不需要知道，也不应该知道发送消息的对象到底是什么。

回忆一下第 11 章的示例。在这个简单的纸牌游戏应用中，游戏需要知道用户什么时候轻拍了纸牌。在用户轻拍时，纸牌对游戏一无所知，只是通过发送通知发出了声音而已；游戏对象已经注册了该通知，并开始进行处理：

```
NSNotificationCenter.defaultCenter().postNotificationName(
    "cardTapped", object: self)
```

再来看一个示例，这个示例利用了通知就是一种广播机制的事实。在我开发的一个应用中，应用委托需要销毁界面，然后从头开始再构建出来。要想不造成内存泄漏（以及其他影响），当前运行着重复 NSTimer 的每个视图控制器都需要将其定时器置为无效状态（参见第 12 章）。相对于找出这些视图控制器，并为每个视图控制器添加一个方法进行调用，我只需发送一个通知，让应用委托发出“Everybody stop timers!”。运行定时器的所有视图控制器都注册了该通知，它们知道在接收到这个通知后应该做什么。

与之类似，KVO（参见第 11 章）可用于实现概念上远离的两个对象之间的同步：当一个对象的一个属性发生变化时，另外一个对象会知晓这个变化。

## 13.5 模型－视图－控制器

Apple 的文档和其他地方都会提及术语：模型－视图－控制器（简称为 MVC）。这指的是一种架构目标，对于带有图形用户界面的程序（用户可以查看和编辑信息）来说，这个目标旨在实现程序在3个功能性方面的分离。MVC的概念可以追溯到Smalltalk的年代，从那以后关于它的介绍已经不胜枚举，下面是对这 3 个术语的通俗解释：

*模型*

数据及对数据的管理，通常称为程序的“业务逻辑”，程序真正关注的核心部分。

*视图*

用户看到并与之交互的部分。

*控制器*

介于模型与视图中间的协调部分。

考虑一个游戏，当前的分数显示在用户眼前：

- 向用户展示当前游戏分数的 UILabel 是个视图；它只不过是个显示而已，其业务知道如何绘制自身。它应该绘制什么（即分数）取决于其他地方。

  新手程序员会将 UILabel 所显示的分数作为实际的分数：为了增加分数，他会读取 UILabel 上的字符串、将其转换为数字、增加该数字、将数字转换回字符串，然后用该字符串替换掉之前的字符串。这完全违背了 MVC 哲学！呈现给用户的视图应该反映出分数，但不应该存储分数。

- 分数是个内部维护的数据；它是个模型。它可能会像拥有公开的 `increment` 方法的实例属性那么简单，也可能像拥有大量方法的 Score 对象那么复杂。

  分数是个数字，而 UILabel 显示的是个字符串；这足以表明视图与控制器本质上是不同的。

- 告诉分数何时应该变化，并在用户界面上显示出更新后的分数，这是控制器的职责。模型的数字分数会通过某种方式进行转换以显示给用户，如果这么想就很清晰了。

  比如，假设 UILabel 显示的分数内容是“The score is 20.”。模型会存储并提供数字 20，那么短语“The score is...”来自于何处呢？控制器会将这个短语放到数字前并呈现给用户。

这个简单的示例（如图 13-1 所示）能够很好地说明 MVC 的优势。通过这种方式进行分离，程序的不同部分可以在很大程度上独自演化。需要使用不同的字体与大小展现分数吗？修改视图即可；模型与控制器不需要知道这些，只是按照之前那样工作即可。想要修改分数前的短语吗？修改控制器即可；模型与视图不会发生任何变化。

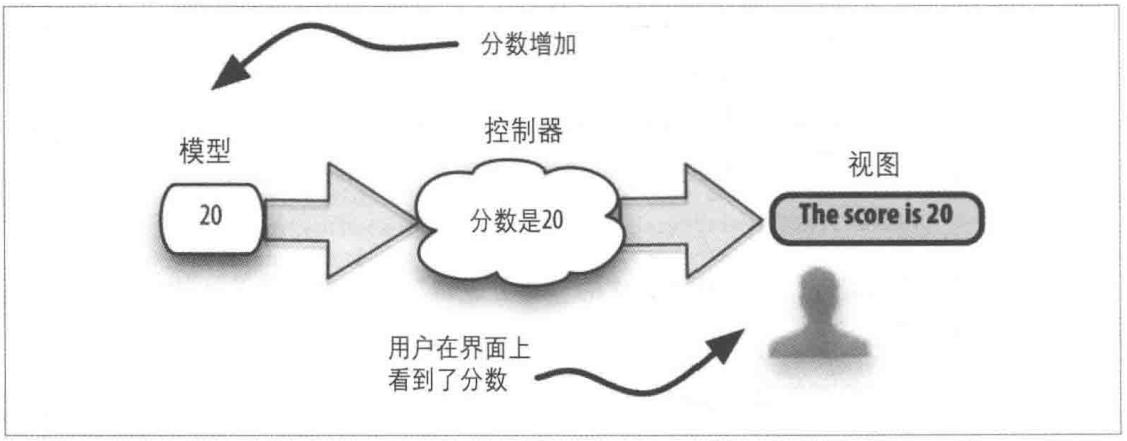


**图 13-1：模型－视图－控制器**

在 Cocoa 应用中遵循 MVC 尤为重要，因为 Cocoa 本身就遵循了 MVC。Cocoa 类的名字揭示出了底层的 MVC 哲学。UIView 是个视图；UIViewController 是个控制器；其目的体现出了视图应该显示什么的逻辑。第 11 章我们看到一个 UIPickerView 并没有持有所要显示的数据；它是从数据源获得数据的。因此，UIPickerView 是个视图；由数据源所维护的数据则是个模型。

Apple 的文档是这样表述的：真正的模型与真正的视图应该是可重用的，它们可以迁移到其他应用中；控制器一般来说是不可重用的，因为它关注的是如何协调模型与视图。

在我开发的一个应用中，我会下载一个 XML（RSS）新闻种子并以表格形式将文章标题展现给用户。XML 的存储与解析完全是模型的事情，因此是可重用的，我甚至都没有编写这部分代码（使用了 Kevin Ballard 编写的名为 FeedParser 的代码）。表格是个 UITableView，它显然是可重用的，因为它来自于 Cocoa。不过，当 UITableView 转向我的代码并询问应该在单元格中显示什么时，我的代码会转向 XML 并请求与表格的这一行相对应的文章标题，这是控制器代码，它只适用于这个应用。

MVC 为应用中对象之间的可见性提供了答案。控制器对象通常要能看到模型对象与视图对象。模型对象或模型对象组通常不需要看到外面。视图对象通常不需要看到外面，不过诸如单例、数据源与目标－动作等结构化设置可以让视图对象与控制器通信。

# C、Objective-C 与 Swift

你是一名 iOS 程序员，并且已经选择使用了 Apple 的全新语言 Swift。这意味着你再也不会关心 Apple 过去的语言 Objective-C 了吗？当然不是这样。

Objective-C 不死。你可以使用 Swift，但 Cocoa 不行。编写 iOS 程序涉及与 Cocoa 及其补充框架的通信。这些框架的 API 是用 Objective-C 或其底层语言 C 编写的。使用 Swift 向 Cocoa 发送的消息会被转换为 Objective-C。跨越 Swift/Objective-C 桥所发送或接收的对象都是 Objective-C 对象。从 Swift 向 Objective-C 所发送的一些对象甚至会被转换为其他对象类型或非对象类型。

在跨越语言之间的桥接发送消息时，你需要知道 Objective-C 期望的到底是什么、Objective-C 会如何处理这些消息、Objective-C 会返回什么结果，这些结果在 Swift 中会是什么样子的。应用可能需要包含一些 Objective-C 代码和 Swift 代码，因此你需要知道应用内部之间的通信方式。

本附录总结了 C 与 Objective-C 的一些语言特性，并介绍了 Swift 会如何使用这些特性。这里并不会讲述如何编写 Objective-C 代码！比如，我会谈及 Objective-C 方法与方法声明，因为你需要知道如何从 Swift 中调用 Objective-C 方法；不过，我并不会介绍如何在 Objective-C 中调用 Objective-C 方法。本书的上一版系统且详尽地介绍了 C 与 Objective-C，因此我建议你参考它以了解关于这些语言的信息。

## A.1 C 语言

Objective-C 是 C 的超集；换句话说，C 构成了 Objective-C 的语言基础。C 中的一切均可用在 Objective-C 中。我们可以（通常也是必要的）编写本质上就是纯 C 的长长的

Objective-C 代码。一些 Cocoa API 是用 C 编写的。因此，为了掌握 Objective-C，我们有必要先了解 C。

C 语句（包括声明）必须以分号结尾。变量在使用前需要先声明。变量声明的语法是：数据类型名后跟变量名，然后跟着初始值的赋值（此为可选）：

```
int i;
double d = 3.14159;
```

C typedef 语句以现有的类型名开始，并为其定义了一个新的同义词：

```
typedef double NSTimeInterval;
```

## A.1.1 C 数据类型

C 并不是面向对象的语言；其数据类型不是对象（它们是标量）。C 中基本的内建数据类型都是数字：char（1 个字节）、int（4 个字节）、float 与 double（浮点数）及各种变种，如 short（短整型）、long（长整型）、unsigned short 等。Objective-C 增加了 NSInteger、NSUInteger（无符号）与 CGFloat。C 中的布尔类型实际上是个数字，0 表示 false；Objective-C 增加了 BOOL，它也是个数字。C 中的原生文本类型（字符串）实际上是个以 null 结尾的字符数组。

Swift 显式提供了可以与 C 数字类型直接交互的数字类型，不过 Swift 的类型是对象，而 C 的类型则不是。Swift 类型别名提供了与 C 类型名字相对应的名字：Swift CBool 是个 C bool、Swift CChar 是个 C char（一个 Swift Int8）、Swift CInt 是个 C int（一个 Swift Int32）、Swift CFloat 是个 C float（一个 Swift Float），诸如此类。Swift Int 可以与 NSInteger 交换使用、Swift UInt 可以与 NSUInteger 交换使用、Swift Bool 可以与 Swift ObjCBool 交换使用，后者表示 Objective-C BOOL。CGFloat 被 Swift 作为一个类型名。

C 与 Swift 之间的一个主要差别在于，当对不同的数字类型进行赋值、传递或比较时，C（以及 Objective-C）会隐式进行转换；但 Swift 不会，因此你需要进行显式转换来匹配类型。

原生的 C 字符串类型（以 null 结尾的字符数组）在 Swift 中的类型为 UnsafePointer<Int8>（回忆一下，Int8 就是个 CChar），稍后将会介绍这么做的原因。我们无法在 Swift 中构造 C 字符串字面值，不过在需要 C 字符串时，你可以传递一个 Swift String：

```
let q = dispatch_queue_create("MyQueue", nil)
```

如果需要创建 C 字符串变量，那么可以使用 NSString 的 UTF8String 属性与 cString-UsingEncoding: 方法来构造 C 字符串。此外，还可以使用 Swift String 的 withCString 实例方法，不过其语法有点麻烦。在该示例中，我遍历了 C 字符串的“字符”，直到遇到 null 终止符（稍后将会介绍 memory 属性）：

```
let _ : Void = "hello".withCString {
    var cs = $0
    while cs.memory != 0 {
        print(cs.memory)
        cs = cs.successor()
    }
}
```

此外，可以通过 Swift String 的静态方法 fromCString 将 C 字符串转换为 Swift String（包装在 Optional 中）。

## A.1.2 C 枚举

C 枚举是个数字；其值是某种形式的整型，可以隐式（从 0 开始）或显式指定其值。C 枚举可以通过各种形式转换为 Swift，这取决于其声明方式。下面就从最简单的形式开始：

```
enum State {
     kDead,
     kAlive
};
typedef enum State State;
```

（最后一行的 typedef 可以让 C 程序使用 State 代替冗长的 enum State 作为类型名）。C 枚举名 kDead 与 kAlive 并不是任何东西的“case”；它们并没有命名空间。它们是常量，由于并未对其进行显式初始化，因此它们分别代表 0 和 1。枚举声明可以进一步指定整型类型；但该示例没有这么做，因此其值在 Swift 中是 UInt32 类型。

在 Swift 2.0 中，老式的 C 枚举会成为使用了 RawRepresentable 协议的 Swift 结构体。当从 C 传递过来或向 C 传递了 State 枚举时，该结构体会自动作为交换的媒介。这样，如果 C 函数 setState 接收一个 State 枚举参数时，你可以通过一个 State 名字调用它：

```
setState(kDead)
```

通过这种方式，Swift 会尽可能以更加方便的方式导入这些名字，将 State 表示为一种类型，不过在 C 中它并不是类型。如果想知道名字 kDead 到底表示什么整数，那只能使用其 rawValue。还可以通过调用 init(rawValue:) 初始化器创建任意的 State 值，并没有编译器或运行期检查来确保该值是一个预定义的常量。不过也不建议你这么做。

Xcode 4.4 引入了一个全新的 C 枚举符号，它基于 NS_ENUM 宏：

```
typedef NS_ENUM(NSInteger, UIStatusBarAnimation) {
    UIStatusBarAnimationNone,
    UIStatusBarAnimationFade,
    UIStatusBarAnimationSlide,
};
```

该符号显式指定了整型类型并将一个类型名关联到了这个枚举。Swift 会将以这种方式声明的枚举用 Swift 枚举的形式导入，保持其名字与类型不变。此外，*Swift* 会自动将导入的 case 名前面的共有前缀去掉：

```
enum UIStatusBarAnimation : Int {
    case None
    case Fade
    case Slide
}
```

此外，带有 Int 原生值类型的 Swift 枚举可以通过 `@objc` 特性公开给 Objective-C。比如：

```
@objc enum Star : Int {
    case Blue
    case White
    case Yellow
    case Red
}
```

Objective-C 会将其看作一个 NSInteger 类型的枚举，枚举名分别是 `StarBlue`、`StarWhite` 等。

还有另外一个 C 枚举符号的变种，它基于 `NS_OPTIONS` 宏，适合于位掩码：

```
typedef NS_OPTIONS(NSUInteger, UIViewAutoresizing) {
    UIViewAutoresizingNone = 0,
    UIViewAutoresizingFlexibleLeftMargin = 1 << 0,
    UIViewAutoresizingFlexibleWidth = 1 << 1,
    UIViewAutoresizingFlexibleRightMargin = 1 << 2,
    UIViewAutoresizingFlexibleTopMargin = 1 << 3,
    UIViewAutoresizingFlexibleHeight = 1 << 4,
    UIViewAutoresizingFlexibleBottomMargin = 1 << 5
};
```

这种方式声明的枚举会以使用了 OptionSetType 协议的结构体的形式进入 Swift 中。OptionSetType 协议使用了 RawRepresentable 协议，因此该结构体有一个 `rawValue` 实例属性，它持有底层的整型值。C 枚举的 case 名是通过静态属性表示的，其每个值都是该结构体的实例；在导入这些静态属性名时会去掉其共有前缀：

```
struct UIViewAutoresizing : OptionSetType {
    init(rawValue: UInt)
    static var None: UIViewAutoresizing { get }
    static var FlexibleLeftMargin: UIViewAutoresizing { get }
    static var FlexibleWidth: UIViewAutoresizing { get }
    static var FlexibleRightMargin: UIViewAutoresizing { get }
    static var FlexibleTopMargin: UIViewAutoresizing { get }
    static var FlexibleHeight: UIViewAutoresizing { get }
    static var FlexibleBottomMargin: UIViewAutoresizing { get }
}
```

这样，调用 UIViewAutoresizing.FlexibleLeftMargin 时就好像在初始化 Swift 枚举的一个 case，不过实际上，它是 UIViewAutoresizing 结构体的一个实例，其 rawValue 属性会被设为原来的 C 枚举所声明的值，对于 .FlexibleLeftMargin 来说就是 1<<0。由于该结构体的静态属性是相同结构体的一个实例；因此像枚举一样，在需要结构体时，可以提供一个静态属性名并省略结构体名：

```
self.view.autoresizingMask = .FlexibleWidth
```

此外，由于这是个 OptionSetType 结构体，因此可以使用集合相关操作。这样就可以通过实例来操纵位掩码了，就好像它是个 Set 一样：

```
self.view.autoresizingMask = [.FlexibleWidth, .FlexibleHeight]
```

如果 Objective-C 中需要一个 NS_OPTIONS 枚举，那么可以通过传递 0 来表示没有提供任何选项。在 Swift 2.0 中，如果需要相应的结构体，那么可以传递 [ ]（一个空集合）。

但遗憾的是，很多常见的替代方案一开始并没有以枚举的形式实现出来。这不是什么问题，但却很不方便。比如，AVFoundation 音频会话类别名只不过是 NSString 常量而已：

```
NSString *const AVAudioSessionCategoryAmbient;
NSString *const AVAudioSessionCategorySoloAmbient;
NSString *const AVAudioSessionCategoryPlayback;
// ... and so on ...
```

虽然这个列表还有着显而易见的共有前缀，但 Swift 却不能通过缩略名将其转换为 AVAudioSessionCategory 枚举或结构体。如果想要指定 Playback 类别，你需要使用全名 AVAudioSessionCategoryPlayback。

## A.1.3 C 结构体

C 结构体是个复合类型，其元素可以通过名字来访问，方式是在对结构体的引用后使用点符号。比如：

```
struct CGPoint {
   CGFloat x;
   CGFloat y;
};
typedef struct CGPoint CGPoint;
```

声明之后，在 C 中就可以这样使用了：

```
CGPoint p;
p.x = 100;
p.y = 200;
```

C 结构体进入 Swift 中后就会变成 Swift 结构体，然后就拥有了 Swift 结构体的特性。比如，Swift 中的 CGPoint 会拥有 x 与 y CGPoint 实例属性；不过，它还会神奇地拥有隐式的成员初始化器！此外，还会注入一个不带参数的初始化器；因此，`CGPoint()` 会创建一个 x 与 y 值都为 0 的 CGPoint。扩展可以提供额外的特性，Swift CoreGraphics 头会向 CGPoint 添加一些：

```
extension CGPoint {
    static var zeroPoint: CGPoint { get }
    init(x: Int, y: Int)
    init(x: Double, y: Double)
}
```

如你所见，Swift CGPoint 拥有一些额外的初始化器，接收 Int 或 Double 参数，还有另外一种创建 0 值 CGPoint 的方式 `CGPoint.zeroPoint`。CGSize 与之类似。特别地，Swift 中的 CGRect 还会拥有额外的方法与属性；如果无法通过 Core Graphics 框架提供的内建 C 函数来操纵 CGRect，那么你也不能通过这些额外的方法实现；不过，你可以以 Swift 的方式来做到这一点。

Swift 结构体是对象，C 结构体却不是，不过这一点并不会对通信造成任何影响。比如，你可以在需要 C CGPoint 时赋值或传递一个 Swift CGPoint，因为 CGPoint 首先来自于 C。Swift 为 CGPoint 添加了对象方法与属性，不过这也没关系；C 看不到它们。C 所关心的只是该 CGPoint 的 x 与 y 元素，而它们可以轻松从 Swift 传递给 C。

## A.1.4 C 指针

C指针是个整型，它指向了内存中真实数据的位置（地址）。分配与回收内存是分开进行的。指向某个数据类型的指针声明是通过在数据类型名后加上一个星号实现的；星号左侧、右侧或两侧可以添加空格。如下代码声明了一个指向 int 的指针，它们是等价的：

```
int *intPtr1;
int* intPtr2;
int * intPtr3;
```

类型名本身是 `int*`（或加上一个空格，即 `int *`）。如前所述，Objective-C 重度使用了 C 指针，查看 Objective-C 代码就会发现很多地方都会出现星号。

C 指针转换为 Swift 后会变成 UnsafePointer，如果可写则会变成 UnsafeMutablePointer；这是个泛型，并且特定于所指向的实际数据类型（指针是“不安全的”，因为 Swift 并不会管理其内存，甚至都不会保证所指数据的完整性）。

比如，下面是个 C 函数声明；之前并没有介绍过 C 函数语法，不过请重点关注每个参数名前的类型即可：

```
void CGRectDivide(CGRect rect,
    CGRect *slice,
    CGRect *remainder,
    CGFloat amount,
    CGRectEdge edge)
```

关键字 void 表示该函数不返回值。CGRect 与 CGRectEdge 都是 C 结构体；CGFloat 则是个基本的数字类型。`CGRect *slice` 与 `CGRect *remainder`（空格的位置不同，不过没关系）表示 `slice` 与 `remainder` 都是 `CGRect*`，即指向 CGRect 的指针。上述声明转换为 Swift 后将如下所示：

```
func CGRectDivide(rect: CGRect,
    _ slice: UnsafeMutablePointer<CGRect>,
    _ remainder: UnsafeMutablePointer<CGRect>,
    _ amount: CGFloat,
    _ edge: CGRectEdge)
```

该上下文中的 UnsafeMutablePointer 类似于 Swift `inout` 参数：你提前声明并初始化了一个恰当类型的 `var`，然后通过 `&` 前缀运算符将其地址作为参数进行传递。当以这种方式传递引用地址时，实际上会创建并传递一个指针：

```
var arrow = CGRectZero
var body = CGRectZero
CGRectDivide(rect, &arrow, &body, Arrow.ARHEIGHT, .MinYEdge)
```

在 C 中，要想访问指针所指向的内存，可以在指针名前使用一个星号：`*intPtr` 表示“指针 `intPtr` 所指向的东西”。在 Swift 中，你可以使用指针的 `memory` 属性。

在该示例中，我们会接收到一个 `stop` 参数，其原始类型为 `BOOL*`，即一个指向 BOOL 的指针；在 Swift 中，它是个 `UnsafeMutablePointer<ObjCBool>`。要想设置指针所指向的这个 BOOL，我们需要设置指针的 `memory`（mas 是个 NSMutableAttributedString）：

```
mas.enumerateAttribute("HERE", inRange: r, options: []) {
    value, r, stop in
    if let value = value as? Int where value == 1 {
        // ...
        stop.memory = true
    }
}
```

最通用的 C 指针类型是指向 void 的指针（`void*`），也叫作通用指针。这里的 `void` 表示没有指定类型；在 C 中，如果需要具体类型的指针，那么我们是可以使用通用指针的，反之亦然。实际上，指向 void 的指针不会再对指针所指向的东西进行类型检查。在

Swift 中，这是个特定于 Void 的指针，一般来说就是 UnsafeMutablePointer<Void> 或相应的 UnsafeMutablePointer<()>。一般来说，当遇到这种类型的指针时，如果需要访问底层数据，那么首先要将 UnsafeMutablePointer 泛型转换为底层数据的类型。

## A.1.5 C 数组

C 数组包含了某种数据类型固定数目的元素。在底层，其所占据的内存大小等于该数据类型的这些固定数目的元素所占据的内存大小总和。由于这一点，C 中的数组名就是指针名，它指向了数组中的首个元素。比如，如果将 arr 声明为一个 int 数组，那么 arr 就可以用在需要 int*（指向 int 的指针）类型值的地方。C 语言通过对引用使用方括号或使用指针来表示数组类型。

（这也说明了为何 Swift 中涉及 C 字符串的字符串方法会将这些字符串当作指向 Int8 的不安全的指针：C 字符串是个字符数组，而 Int8 是个字符。）

比如，C 函数 CGContextStrokeLineSegments 的声明如下所示：

```
void CGContextStrokeLineSegments(CGContextRef c,
   const CGPoint points[],
   size_t count
);
```

第 2 个参数是个 CGPoint 类型的 C 数组；这是根据方括号得出的结论。C 数组并不会表示出其中所包含的元素个数，因此要想将该 C 数组传递给这个函数，你还需要告诉函数数组中所包含的元素个数；这正是第 3 个参数的意义。CGPoint 类型的 C 数组是个指向 CGPoint 的指针，因此该函数声明转换为 Swift 后如下所示：

```
func CGContextStrokeLineSegments(c: CGContext?,
    _ points: UnsafePointer<CGPoint>,
    _ count: Int)
```

要想调用该函数并将其传递给 CGPoint 类型的 C 数组，你需要创建一个 CGPoint 类型的 C 数组。C 数组并不是 Swift 数组；那么该如何做到这一点呢？其实你什么都不用做。虽然 Swift 数组不是 C 数组，但你可以传递一个指向 Swift 数组的指针。实际上，你甚至都不需要传递指针，你可以传递一个对 Swift 数组本身的引用。由于这并不是一个可变指针，因此可以通过 let 声明数组；事实上，你甚至可以传递一个 Swift 数组字面值！无论选择哪种方式，Swift 都会帮你转换为 C 数组，并将其作为参数跨越从 Swift 到 C 的桥梁：

```
let c = UIGraphicsGetCurrentContext()!
let arr = [CGPoint(x:0,y:0),
    CGPoint(x:50,y:50),
```

```
    CGPoint(x:50,y:50),
    CGPoint(x:0,y:100),
]
CGContextStrokeLineSegments(c, arr, 4)
```

不过，如果需要，你还是可以构造出一个 C 数组。要想做到这一点，首先要留出内存块：声明一个所需类型的 UnsafeMutablePointer，调用静态方法 alloc 并传递所需的元素数量。接下来通过下标将元素直接写进去来初始化内存。最后，由于 UnsafeMutablePointer 是个指针，你将其（而不是指向它的指针）作为参数进行传递：

```
let c = UIGraphicsGetCurrentContext()!
let arr = UnsafeMutablePointer<CGPoint>.alloc(4)
arr[0] = CGPoint(x:0,y:0)
arr[1] = CGPoint(x:50,y:50)
arr[2] = CGPoint(x:50,y:50)
arr[3] = CGPoint(x:0,y:100)
CGContextStrokeLineSegments(c, arr, 4)
```

接收到 C 数组时也可以使用同样便捷的下标。比如：

```
let col = UIColor(red: 0.5, green: 0.6, blue: 0.7, alpha: 1.0)
let comp = CGColorGetComponents(col.CGColor)
```

上述代码执行完毕后，comp 的类型就是个指向 CGFloat 的 UnsafePointer。这实际上意味着它是个 CGFloat 类型的 C 数组，你可以通过下标访问其元素：

```
if let sp = CGColorGetColorSpace(col.CGColor) {
   if CGColorSpaceGetModel(sp) == .RGB {
       let red = comp[0]
       let green = comp[1]
       let blue = comp[2]
       let alpha = comp[3]
       // ...
    }
}
```

## A.1.6 C 函数

C 函数声明以返回类型开始（返回类型可能为 void，表示没有返回值），后跟函数名，然后是一对圆括号，括号里面是逗号分隔的参数列表，列表中的每一项都是由类型与参数名构成的。参数名都是内部使用的，调用 C 函数时不会用到它们。

下面是 CGPointMake 的 C 声明，它返回一个初始化过的 CGPoint：

```
CGPoint CGPointMake (
    CGFloat x,
    CGFloat y
);
```

下面展示了如何在 Swift 中调用它：

```
let p = CGPointMake(50,50)
```

在 Objective-C 中，CGPoint 并不是对象，CGPointMake 是创建 CGPoint 的主要方式。如前所述，Swift 提供了初始化器，不过我个人仍然倾向于使用 CGPointMake。

在 C 中，函数有一个类型，这是根据其签名得来的，函数名就是对函数的引用，因此在需要某个类型的函数时，我们可以通过函数名来传递这个函数（这有时也叫作指向函数的指针）。在声明中，指向函数的指针可以通过在圆括号中使用星号来表示。

比如，下面是 Audio Toolbox 框架的一个 C 函数的声明：

```
extern OSStatus
AudioServicesAddSystemSoundCompletion(SystemSoundID inSystemSoundID,
    CFRunLoopRef __nullable inRunLoop,
    CFStringRef __nullable inRunLoopMode,
    AudioServicesSystemSoundCompletionProc inCompletionRoutine,
    void * __nullable inClientData)
```

（现在请忽略 __nullable，稍后将会对其进行介绍；extern 也不用管，后面也不会介绍它）。SystemSoundID 仅仅是个 UInt32 而已。不过，AudioServicesSystemSoundCompletionProc 是什么呢？它是：

```
typedef void (*AudioServicesSystemSoundCompletionProc)(SystemSoundID ssID,
    void* __nullable clientData);
```

SystemSoundID 是个 UInt32，它告诉你在 C 用于表示这个含义的令人费解的语法中，AudioServicesSystemSoundCompletionProc 是个指向函数的指针，该函数接收两个参数（类型为 UInt32 以及指向 void 的指针），不返回结果。

在 Swift 1.2 及之前版本中，调用 AudioServicesAddSystemSoundCompletion 的唯一方式是在 Objective-C 中构造 AudioServicesSystemSoundCompletionProc。这个 C 函数的参数类型为 CFunctionPointer，这是个细节未知的结构体，你无法在 Swift 中创建。

不过在 Swift 2.0 中，你可以在需要 C 中指向函数的指针的情况下传递一个 Swift 函数！与往常一样，在传递函数时，你可以单独定义函数并传递其名字，或是以内联的方式将函数构造为匿名函数。如果准备单独定义函数，那么它必须要是个函数，这意味着它不能是方法。定义在文件顶部的函数是可以的；定义在函数内部的函数也是没问题的。

如下是我编写的 AudioServicesSystemSoundCompletionProc，它声明在文件顶部：

```
func soundFinished(snd:UInt32, _ c:UnsafeMutablePointer<Void>) -> Void {
    AudioServicesRemoveSystemSoundCompletion(snd)
```

```
    AudioServicesDisposeSystemSoundID(snd)
}
```

这是用于播放音频文件（作为系统声音）的代码，包含了对 AudioServicesAddSystemSoundCompletion 的调用：

```
let sndurl =
    NSBundle.mainBundle().URLForResource("test", withExtension: "aif")!
var snd : SystemSoundID = 0
AudioServicesCreateSystemSoundID(sndurl, &snd)
AudioServicesAddSystemSoundCompletion(snd, nil, nil, soundFinished, nil)
AudioServicesPlaySystemSound(snd)
```

# A.2 Objective-C

Objective-C 构建在 C 之上。它添加了一些语法与特性，不过继续使用着 C 语法与数据类型，其底层依然是 C。

与 Swift 不同，Objective-C 没有命名空间。出于这个原因，不同框架通过不同的前缀作为名字的开始以进行区分。“CGFloat”中的“CG”表示 Core Graphics，因为它声明在 Core Graphics 框架中。“NSString”中的“NS”表示 NeXTStep，这是 Cocoa 框架过去的名字，诸如此类。

## A.2.1 Objective-C 对象与 C 指针

所有的 C 数据类型与语法都是 Objective-C 的一部分。不过，Objective-C 还是面向对象的，因此它需要通过某种方式向 C 中添加对象。它是通过 C 指针做到这一点的。C 指针可以指向任何东西；对所指向的目标的管理是另外一件事，这正是 Objective-C 所关注的。这样，Objective-C 对象类型都是通过 C 指针语法来表达的。

比如，下面是 addSubview: 方法的 Objective-C 声明：

```
- (void)addSubview:(UIView *)view;
```

目前还没有介绍过 Objective-C 方法声明语法，不过请将注意力放在圆括号中 view 参数的类型声明上：它是个 UIView*。看起来表示的好像是“指向 UIView 的指针”。说是也是，说不是也不是。所有的 Objective-C 对象引用都是指针。这样，说它是指针只是表示它是个对象而已。指针的另一边则是个 UIView 实例。

不过，将该方法转换为 Swift 后就看不到任何指针的影子了：

```
func addSubview(view: UIView)
```

一般来说，当 Objective-C 需要一个类实例时，在 Swift 中只需传递一个指向类实例的引用即可；Objective-C 声明中会通过星号来表示对象，不过你不用管这些。从 Swift 中调用 `addSubview:` 方法时作为参数传递的是个 UIView 实例。你会有这样一种感觉，当传递类实例时，实际传递的是一个指针，因为类实例是引用类型。这样，Swift 与 Objective-C 看待类实例的方式其实是一样的。区别在于 Swift 不会使用指针符号。

Objective-C 的 `id` 类型是个指向对象的通用指针，相当于指向 void 的指针对象。任何对象类型都可以赋值给 `id`，也可以转换为 `id`，还可以通过 `id` 来构造（Swift 的 AnyObject 与之类似）。由于 `id` 本身就是个指针，因此声明为 `id` 的引用并不会使用星号；你可能永远都看不到 `id*` 这种写法。

## A.2.2 Objective-C 对象与 Swift 对象

Objective-C 对象是类与类的实例，进入 Swift 中后基本上都是原封不动的。你在子类化 Objective-C 类或使用 Objective-C 类实例时不会遇到任何问题。

反之亦然。如果 Objective-C 需要一个对象，那么它实际上需要的是一个类，Swift 则可以提供。对于最一般的情况来说，即 Objective-C 需要一个 `id`，你可以传递任何一个类型使用了 AnyObject 的实例，也就是说，其类型是个类。此外，Swift 还会将某些非类的类型转换为 Objective-C 类的等价物。如下结构体可以转换为 AnyObject，并且在 Objective-C 需要对象时能够自动桥接为 Objective-C 类类型：

- String 到 NSString
- Int、UInt、Double、Float 与 Bool 到 NSNumber
- Array 到 NSArray
- Dictionary 到 NSDictionary
- Set 到 NSSet

Swift 的自动桥接使得数字类型的处理要比在 Objective-C 中容易得多。Swift Int 可用在需要 Objective-C 对象的地方，因为 Swift 会将其包装为一个 NSNumber；在 Objective-C 中，你就得记得将一个整型包装到 NSNumber 中。

只要元素类型是类类型或是可以桥接为类类型（可以转换为 AnyObject），并且不是 Optional（因为 Objective-C 集合中不能包含 `nil`），那么 Swift 集合（Array、Dictionary 与 Set）就可以桥接为 Objective-C 集合。

Swift 可以看到 Objective-C 类类型的方方面面（请参考第 10 章了解 Swift 是如何看到 Objective-C 属性的）。不过，很多 Swift 类型是 Objective-C 所看不到的（也没什么问题）。Objective-C 无法看到如下类型：

- Swift 枚举，除了拥有 Int 原生值的 `@objc` 枚举。
- Swift 结构体，除了可以桥接的或是最终来自于 C 的那些结构体。
- 没有从 NSObject 继承的 Swift 类。
- 嵌套类型、泛型与元组。

虽然 Objective-C 可以看到 Swift 类型，但却无法看到类型里面的某些属性（属性的类型是 Swift 所无法看到的），如果方法的参数或返回值类型是 Swift 所看不到的，那么这些方法也是看不到的。你可以自由使用这些属性与方法，甚至在 Objective-C 类类型的子类或扩展中；Objective-C 对此没有什么问题，因为对于 Objective-C 来说，它们根本就不存在。

如果 Objective-C 能够看到某个类型，那么它就能看到包装该类型的 Optional，除了数字类型。比如，Objective-C 无法看到类型为 `Int?` 的属性。推测起来这是因为 Int 无法直接桥接到 Objective-C；需要将其包装到 NSNumber 中，但仅仅通过类型声明是做不到这一点的。

`@obj` 特性会向 Objective-C 公开一些通常情况下 Objective-C 无法看到的东西，前提是满足合法性要求。它还有另外一个目的：在将某个东西标记为 `@obj` 时，你可以添加一对圆括号，里面包含着你希望 Objective-C 看到的该成员的名字。你甚至可以自由对 Objective-C 所能看到的类或类成员这么做，比如：

```
@objc(ViewController) class ViewController : UIViewController { // ...
```

上述代码演示了在实际开发中颇具价值的一件事。在默认情况下，Objective-C 会认为你的类名是根据模块名（一般来说就是项目名）划分了命名空间（前缀）的。这样，该 ViewController 类就会被 Objective-C 当作 `MyCoolApp.ViewController`。这会破坏类名与其他东西之间的关联关系。比如，在将现有的 Objective-C 项目转换为 Swift 时，你可能会使用 `@objc(...)` 语法防止 nib 对象或 NSCoding 归档失去与其关联类的关联关系。

## A.2.3 Objective-C 方法

在 Objective-C 中，方法参数可以有自己的名字，整体来看，方法名与参数名是一样的。参数名是方法名的一部分，每个参数名后都有一个冒号。比如，UIViewController 类有

一个名为 presentViewController:animated:completion: 的实例方法。方法名中包含了 3 个冒号，因此这个方法会接收 3 个参数。如下代码展示了如何在 Objective-C 中调用它：

```
SecondViewController* svc = [SecondViewController new];
[self presentViewController:svc animated:YES completion:nil];
```

一个 Objective-C 方法的声明包含如下 3 部分：

- 一个 + 或 −，分别表示方法是类方法还是实例方法。
- 一对圆括号，里面是返回值的数据类型。它可能是 void，表示没有返回值。
- 方法名，通过冒号分隔以便为参数留出位置。每个冒号后是个圆括号，里面是参数的数据类型，后跟参数的占位符名。

比如，UIViewController 实例方法 presentViewController:animated:completion: 的 Objective-C 声明如以下代码所示：

```
- (void)presentViewController: (UIViewController *)viewControllerToPresent
    animated: (BOOL)flag
    completion: (void (^ __nullable)(void))completion;
```

（看起来比较奇怪的第 3 个参数类型是个块，稍后将会介绍。）

回忆一下，在默认情况下，Swift 方法会外化除第一个参数外的所有其他参数名。因此，Objective-C 方法声明会按照如下规则转换为 Swift：

- 第一个冒号前面的内容会成为函数名。
- 除了第一个冒号，其他每个冒号前面的内容会成为一个外部参数名。第一个参数没有外部名。
- 参数类型后面的名字会成为内部（局部）参数名。如果外部参数名与内部（局部）参数名同名，那就没必要再重复声明一次了。

这样，上述 Objective-C 方法声明转换为 Swift 后如以下代码所示：

```
func presentViewController(viewControllerToPresent: UIViewController,
    animated flag: Bool,
    completion: (() -> Void)?)
```

当在 Swift 中调用方法时，内部参数名是不起作用的：

```
let svc = SecondViewController()
self.presentViewController(svc, animated: true, completion: nil)
```

在实现 Objective-C 中声明的方法时需要遵循所使用的协议或重写继承下来的方法。Xcode 的代码完成特性会帮你提供好内部参数名，不过你可以修改它们。不过，外部参数名是不能修改的；它们是方法名的一部分！

这样，如果要重写 presentViewController:animated:completion:（你可能不会这么做），那么可以像下面这样做：

```
override func presentViewController(vc: UIViewController,
    animated anim: Bool,
    completion handler: (() -> Void)?) {
    // ...
}
```

与 Swift 不同，Objective-C 并不允许方法重载。在 Objective-C 中，如果两个 ViewController 实例方法都叫作 myMethod: 并且不返回结果，其中一个方法接收 CGFloat 参数，另一个方法接收 NSString 参数，那么这是不合法的。因此，对于这样两个 Swift 方法来说，虽然在 Swift 中是合法的，但如果它们都对 Objective-C 可见，结果就是不合法的了。在 Swift 2.0 中，你可以通过 @nonobjc 特性将某些正常情况下对 Objective-C 可见的东西隐藏。这样，将其中一个方法标记为 @nonobjc 就可以解决问题。

Objective-C 有自己的可变参数形式。比如，NSArray 实例方法 arrayWithObjects: 的声明如下所示：

```
+ (id)arrayWithObjects:(id)firstObj, ... ;
```

与 Swift 不同，我们必须得显式告诉这样的方法所提供的参数个数。很多这样的方法（包括 arrayWithObjects:）都使用了 nil 终止符；也就是说，调用者在最后一个参数后会提供一个 nil，被调用者知道最后一个参数是什么时候传递的，因为它遇到了 nil。在 Objective-C 中是这样调用 arrayWithObjects: 的：

```
NSArray* pep = [NSArray arrayWithObjects: manny, moe, jack, nil];
```

Objective-C 无法调用（也看不到）接收可变参数的 Swift 方法。不过，Swift 却可以调用接收可变参数的 Objective-C 方法，只要方法被标识为 NS_REQUIRES_NIL_TERMINATION 即可。arrayWithObjects: 就是通过这种方式标记的，因此可以这样使用 NSArray(objects:1, 2, 3)，Swift 会提供缺失的 nil 终止符。

## A.2.4 Objective-C 初始化器与工厂

Objective-C 初始化器方法是实例方法；实际的实例化是由 NSObject 的类方法 alloc 执行的，Swift 并未提供对应之物（其实也不需要），初始化器消息会发送给生成的实例。

比如，如下代码展示了如何在 Objective-C 中通过提供 red、green、blue 与 alpha 值来创建 UIColor 实例：

```
UIColor* col = [[UIColor alloc] initWithRed:0.5 green:0.6 blue:0.7 alpha:1];
```

在 Objective-C 中，这个初始化器的名字是 `initWithRed:green:blue:alpha:`。其声明如下所示：

```
- (UIColor *)initWithRed:(CGFloat)red green:(CGFloat)green
    blue:(CGFloat)blue alpha:(CGFloat)alpha;
```

简而言之，初始化器方法从外表来看就是个实例方法，与 Objective-C 中的其他实例方法一样。

不过，Swift 可以检测到 Objective-C 中的初始化器，因为其名字很特殊，以 `init` 开头。因此，Swift 可以将 Objective-C 初始化器转换为 Swift 初始化器。

这种转换是以一种特殊的方式进行的。与普通方法不同，Objective-C 初始化器在转换为 Swift 时会将所有参数名作为圆括号中的外部名。同时，第一个参数的外部名会自动缩短：单词 `init` 会从第一个参数名的开头去掉，如果存在单词 `With`，它也会被去掉。这样，Swift 中该初始化器的第一个参数的外部名就是 `red:`。如果外部名与内部名相同，那就没必要重复使用了。这样，Swift 会将 Objective-C 的 `initWithRed:green:blue:alpha:` 转换为 Swift 初始化器 `init(red:green:blue:alpha:)`，其声明如下所示：

```
init(red: CGFloat, green: CGFloat, blue: CGFloat, alpha: CGFloat)
```

下面是调用：

```
let col = UIColor(red: 0.5, green: 0.6, blue: 0.7, alpha: 1.0)
```

还有一种方式可以在 Objective-C 中创建实例。很多时候，类会提供一个类方法，这个类方法是实例工厂。比如，UIColor 类有一个类方法 `colorWithRed:green:blue:alpha:`，其声明如下所示：

```
+ (UIColor*) colorWithRed: (CGFloat) red green: (CGFloat) green
                     blue: (CGFloat) blue alpha: (CGFloat) alpha;
```

Swift 会通过一些模式匹配规则检测到这种工厂方法，即返回类实例的类方法，其名字以类名开头并去掉了前缀，同时会将其转换为初始化器，并去掉第一个参数名开头的类名（以及 `With`）。

如果得到的初始化器已经存在，就像这个示例中那样，那么 Swift 就会认为这个工厂方法是多余的，并且不再使用它！这样，Objective-C 的类方法 `colorWithRed:green:-blue:alpha:` 就无法从 Swift 调用，因为它与已经存在的 `init(red:green:blue:alpha:)` 相同。

这种同名规则反过来也是适用的：比如，Swift 初始化器 `init(value:)` 会被 Objective-C 看作 `initWithValue:` 并调用。

## A.2.5 选择器

有时，Objective-C 方法希望接收一个稍后会被调用的方法名作为参数。这样的名字叫作选择器。比如，`addTarget:action:forControlEvents:` 方法调用时会告诉界面上的按钮："从现在起，当用户轻拍你时，请将这条消息发送给这个对象。"消息 `action:` 参数就是个选择器。

可以这么想，如果这是个 Swift 方法，那么你会传递一个函数。不过，选择器与函数不同。它仅仅是个名字而已。与 Swift 不同，Objective-C 是动态的，它可以在运行期只根据名字即可构建并向任意对象发送任意消息。

不过，虽然仅仅是个名字，选择器并非字符串。实际上，它是个独立的对象类型，在 Objective-C 声明中被指定为 `SEL`，在 Swift 声明中被指定为 Selector。不过在大多数情况下，如果需要一个选择器，那么 Swift 还是允许你传递一个字符串的，这是一种简便方式！比如：

```
b.addTarget(self, action: "doNewGame:", forControlEvents: .TouchUpInside)
```

有时，你需要构造实际的 Selector 对象，这可以通过将字符串转换为 Selector 来实现。在该示例中，Selector 是一个参数，我们需要通过比较来确定它。不能将 Selector 与字符串进行比较，因此需要将字符串转换为 Selector，从而比较两个 Selector：

```
override func canPerformAction(action: Selector,
    withSender sender: AnyObject!) -> Bool {
        if action == Selector("undo:") { // ...
```

在提供选择器时，如果要获得它的名字该怎么办呢？如果调用了 `addTarget:action:forControlEvents:` 这样的方法，并且在提供 `action:` 参数时搞错了方法名，那么编译期是不会有错误和警告的，不过 Objective-C 会尝试将这个错误的消息发送给目标，这时应用就会崩溃，控制台会打印出"unrecognized selector"消息。这是为数不多的 Swift 给 Objective-C 程序员带来麻烦的地方，而这本可以避免（我认为这是 Swift 语言的一个严重的问题）。

要想得到正确的名字，你需要将 Swift 方法声明转换为对应的 Objective-C 名字。这种转换很简单，并且遵循着一些确定的原则，不过你会将这个名字作为字面值输入，这太容易敲错了，因此请小心行事：

1. 名字以方法名中左圆括号前面的字符开头。

2. 如果方法不接收参数，那就结束了。
3. 如果方法接收参数，那么请添加一个冒号。
4. 如果方法接收多个参数，那么请添加除第一个参数外的其他所有参数的外部名，并且在每个外部参数名后加上一个冒号。

这意味着如果方法接收参数，那么其 Objective-C 名字就会以一个冒号结尾。这里是区分大小写的，除了冒号，名字不应该包含任何空格或其他符号。

下面就来说明一下，这里有 3 个 Swift 方法声明，注释中给出的是其对应的 Objective-C 名字：

```
func sayHello() -> String // "sayHello"

func say(s:String) // "say:"

func say(s:String, times n:Int) // "say:times:"
```

如果不喜欢外化 Swift 方法的第 1 个参数名，那么 Objective-C 对方法名的第一部分添加了 "With" 和大写的外部参数名。比如：

```
func say(string s:String) // "sayWithString:"
```

即便选择器名能够正确对应上所声明的方法，应用还是可能会崩溃。比如，下面是个简单的测试类，它创建了一个 NSTimer，并让其每隔一秒钟调用某个方法一次：

```
class MyClass {
    var timer : NSTimer?
    func startTimer() {
        self.timer = NSTimer.scheduledTimerWithTimeInterval(1,
            target: self, selector: "timerFired:",
            userInfo: nil, repeats: true)
    }
    func timerFired(t:NSTimer) {
        print("timer fired")
    }
}
```

从结构上来看，这个类没有任何问题；它可以编译通过，并且当应用运行时会实例化。不过在调用 `startTimer` 时，应用会崩溃。问题并不是因为 `timerFired` 不存在，或 "`timerFired:`"" 不是其名字；问题在于 Cocoa 找不到 `timerFired`。这是因为 MyClass 类是个纯 Swift 类；因此，它缺少 Objective-C 的内省能力与消息发送机制，而 Cocoa 正是通过它们发现并调用 `timerFired` 的。这个问题有如下几种解决方案：

- 将 MyClass 声明为 NSObject 子类。
- 声明 `timerFired` 时加上 `@objc` 特性。

- 声明 `timerFired` 时加上 `dynamic` 关键字（不过这么做有些过犹不及；当 Objective-C 需要修改类成员实现时需要用到 `dynamic`，不过不应该过度使用这个关键字）。

## A.2.6 CFTypeRefs

CFTypeRef 是个全局 C 函数，调用起来也很简单。其代码看起来给人的感觉就好像 Swift 跟 C 一样。

要想了解 CFTypeRef 假对象及其内存管理，请参见第 12 章。CFTypeRef 是个指针，因此它可以与 C 中指向 void 的指针互换。由于它是个指向假对象的指针，因此它可以与 Objective-C `id` 和 Swift AnyObject 互换。

很多 CFTypeRefs 可以自动桥接到相应的 Objective-C 对象类型。比如，CFString 与 NSString、CFNumber 与 NSNumber、CFArray 与 NSArray、CFDictionary 与 NSDictionary 都是自动桥接的（除此之外还有很多）。每一对都可以通过类型转换进行互换，有时也需要这么做。此外，在 Swift 中要比 Objective-C 中更容易一些。在 Objective-C 中，你需要执行桥接转换，告诉 Objective-C 当这个对象跨越了 Objective-C 的内存管理方式与 C 和 CFTypeRefs 的非托管内存管理方式时该如何管理其内存。不过在 Swift 中，CFTypeRefs 的内存是托管的，因此没必要进行桥接转换；你只需进行普通的转换即可。实际上在很多情况下，Swift 都知道自动桥接，并且会自动进行类型转换。

比如，如下代码来自于我开发的一个应用，这里使用了 ImageIO 框架。该框架有一个 C API 并使用了 CFTypeRefs。`CGImageSourceCopyPropertiesAtIndex` 会返回一个 CFDictionary，其键是 CFStrings。从字典中获取值最简单的方式是通过下标，不过无法对 CFDictionary 这么做，因为它并不是对象；因此，我将其转换为 NSDictionary。键 `kCGImagePropertyPixelWidth` 是个 CFString，它并非 Hashable（它并不是一个真正的对象，不能使用协议），因此无法作为 Swift 字典的键；不过，当我尝试直接通过下标来使用它时，Swift 是允许的，因为它会帮我将其转换为 NSString：

```
let result =
    CGImageSourceCopyPropertiesAtIndex(src, 0, nil)! as [NSObject:AnyObject]
let width = result[kCGImagePropertyPixelWidth] as! CGFloat
```

与之类似，在如下代码中，我通过 CFString 键构造了一个字典 `d` 并将其传递给 `CGImageSourceCreateThumbnailAtIndex` 函数（该函数接收一个 CFDictionary）。我不需要显式做任何强制类型转换！不过，我需要指定字典类型，从而让 Swift 能帮我将所有键和值转换为 Objective-C 对象：

```
let d : [NSObject:AnyObject] = [
    kCGImageSourceShouldAllowFloat : true,
    kCGImageSourceCreateThumbnailWithTransform : true,
    kCGImageSourceCreateThumbnailFromImageAlways : true,
    kCGImageSourceThumbnailMaxPixelSize : w
]
let imref = CGImageSourceCreateThumbnailAtIndex(src, 0, d)!
```

## A.2.7 块

块是 Apple 在 iOS 4 中引入的一个 C 语言特性。它非常类似于 C 函数，但并不是 C 函数；其行为类似于闭包，可以作为引用类型进行传递。实际上，块相当于 Swift 函数并与之兼容，它们之间可以互换：当需要块时，你可以传递一个 Swift 函数；当 Cocoa 将块传递给你时，它看起来就像是函数一样。

在 C 与 Objective-C 中，块的声明是通过插入符号（^）表示的，它可以用在 C 函数声明中函数名出现的地方（或是圆括号中的星号）。比如，NSArray 的实例方法 `sortedArrayUsingComparator:` 接收一个 NSComparator 参数，它是通过 `typedef` 定义的，如以下代码所示：

```
typedef NSComparisonResult (^NSComparator)(id obj1, id obj2);
```

要想读懂上述声明，请从中间开始，然后向两边看；它表示的是“NSComparator 是块的类型，它接收两个 `id` 参数并返回一个 NSComparisonResult”。因此在 Swift 中，该 `typedef` 会被转换为：

```
typealias NSComparator = (AnyObject, AnyObject) -> NSComparisonResult
```

很多时候都没有 `typedef`，块的类型会直接出现在方法声明中。下面是 Objective-C 中 UIView 类方法的声明，它接收两个块参数：

```
+ (void)animateWithDuration:(NSTimeInterval)duration
    animations:(void (^)(void))animations
    completion:(void (^ __nullable)(BOOL finished))completion;
```

在上述声明中，`animations:` 是个块，它不接收参数（`void`），也没有返回值；`completion:` 也是个块，它接收一个类型为 BOOL 的参数，没有返回值。下面是转换后的 Swift 代码：

```
class func animateWithDuration(duration: NSTimeInterval,
    animations: () -> Void,
    completion: ((Bool) -> Void)?)
```

对于这些方法来说，在调用时如果需要一个块参数，那么可以将函数作为参数传递进去。下面这个方法示例中，一个函数会传递给你。这是其 Objective-C 声明：

```
- (void)webView:(WKWebView *)webView
    decidePolicyForNavigationAction:(WKNavigationAction *)navigationAction
    decisionHandler:(void (^)(WKNavigationActionPolicy))decisionHandler;
```

实现这个方法后，当用户轻拍了 Web View 上的链接时，它会被调用，从而可以决定该如何响应。第 3 个参数是个块，它接收一个枚举类型的参数 WKNavigationActionPolicy，并且没有返回值。块会作为一个 Swift 函数传递给你，你通过调用该函数作出响应：

```
func webView(webView: WKWebView,
    decidePolicyForNavigationAction navigationAction: WKNavigationAction,
    decisionHandler: ((WKNavigationActionPolicy) -> Void)) {
      // ...
      decisionHandler(.Allow)
}
```

在 Objective-C 中，块可以转换为 id。不过，Swift 函数却无法转换为 AnyObject。然而，有时在 Objective-C 中，当需要 id 时你可以提供块；你希望在 Swift 中也可以这样，当需要 AnyObject 时可以提供 Swift 函数。比如，一些对象类型（如 CALayer 与 CAAnimation）允许使用键值编码来追加任意键值对并在后面获取到它；将函数作为值追加上去也是合情合理的。

一个简单的解决办法就是声明一个 NSObject 子类，其中包含一个函数类型的属性：

```
typealias MyStringExpecter = (String) -> ()
class StringExpecterHolder : NSObject {
     var f : MyStringExpecter!
}
```

现在可以将函数包装到类实例中：

```
func f (s:String) {print(s)}
let holder = StringExpecterHolder()
holder.f = f
```

接下来在需要 AnyObject 时将该实例传递过去：

```
\let lay = CALayer()
lay.setValue(holder, forKey:"myFunction")
```

后面就可以抽取该实例，将其从 AnyObject 进行向下类型转换，并调用它所包装的函数，这一切都是非常简单的：

```
let holder2 = lay.valueForKey("myFunction") as! StringExpecterHolder
holder2.f("testing")
```

C 函数并不是块，不过在 Swift 2.0 中，你还可以在需要 C 函数的地方使用 Swift 函数，这一点在之前已经介绍过了。另外，为了将某个类型声明为 C 中指向函数的指针，请将类型标记为 @convention(c)。比如，如下是两个 Swift 方法声明：

```
func blockTaker(f:()->()) {}
func functionTaker(f:@convention(c)() -> ()) {}
```

Objective-C 将第 1 个看作接收一个 Objective-C 块，将第 2 个看作接收一个 C 中的指向函数的指针。

## A.2.8 API 标记

当 Swift 于 2014 年 6 月首次进入公众视线时，人们认为其严格、具体的类型相对于 Objective-C 动态、松散的类型来说很不相配。主要的问题有：

- 在 Objective-C 中，任何对象实例引用都可以为 `nil`。不过在 Swift 中，只有 Optional 才能为 `nil`。默认的解决方案是将隐式展开的 Optional 作为 Objective-C 与 Swift 之间对象交换的媒介。不过这么做有些一刀切，因为来自于 Objective-C 的大多数对象实际上都不会为 `nil`。
- 在 Objective-C 中，诸如 NSArray 这样的集合类型可以包含多种对象类型的元素，而集合本身并不知道其所包含的元素类型。不过，Swift 集合类型只能包含一种类型的元素，其本身的类型也是由所包含的元素类型所决定的。默认的解决方案是对来自于 Objective-C 的通用类型的集合来说，我们需要在 Swift 端显式对其进行向下类型转换。当我们要获取一个视图的子视图时，常常会得到一个 `[AnyObject]`，然后需要将其向下类型转换为 `[UIView]`；但实际上，视图的子视图一定是 UIView 对象，这么做有些麻烦。

这些问题随后通过修改Objective-C语言得到了解决，解决方案是允许在声明时使用标记，从而让 Objective-C 能与 Swift 对所需要的对象类型进行更为具体的沟通。

Objective-C 对象类型可以标记为 `nonnull` 或 `nullable`，分别表示对象不会为 `nil` 或可能为 `nil`。与之类似，C 指针类型可以标记为 `__nonnull` 或 `__nullable`。借助于这些标记，我们就不必再将隐式展开的 Optional 作为交换的中间媒介了；每种类型都可以是常规类型或是常规的 Optional。这样，现如今的 Cocoa API 中就很少会出现隐式展开的 Optional 了。

如果编写 Objective-C 头文件，但没有将任何类型标记为可空类型，那就会回到以往的阴暗日子了：Swift 会将你定义的类型看作隐式展开的 Optional。比如，下面是一个 Objective-C 方法声明：

```
- (NSString*) badMethod: (NSString*) s;
```

由于缺少标记，Swift 看到的是下面这个样子：

```
func badMethod(s: String!) -> String!
```

如果头文件包含了标记，但标记不完全，Objective-C 编译器就会发出警告。为了解决这个问题，你可以使用默认的 nonnull 设置将整个头文件都标记起来；接下来则需要只标记那些 nullable 类型：

```
NS_ASSUME_NONNULL_BEGIN
- (NSString*) badMethod: (NSString*) s;
- (nullable NSString*) goodMethod: (NSString*) s;
NS_ASSUME_NONNULL_END
```

Swift 不会再将其看作隐式展开的 Optional 了：

```
func badMethod(s: String) -> String
func goodMethod(s: String) -> String?
```

这种标记还可以让 Swift 编译器对继承下来或基于协议的 Objective-C 方法声明的正确性执行更为严格的检查。过去，你可以修改一个类型的可选择性（optionality）；现在，如果做得不对编译器会告诉你。比如，如果 Objective-C 将一个类型声明为 nullable-NSString*，那么你就不能将其声明为 String；你必须得将其声明为 String?。

要想标记包含某种元素类型的集合类型，请将元素类型放到尖括号（<>）中，置于集合类型名之后，星号之前。下面是一个返回字符串数组的 Objective-C 方法：

```
- (NSArray<NSString*>*) pepBoys;
```

Swift 会将该方法的返回类型看作 [String]，并且不需要再对其进行向下类型转换了。

在实际的 Objective-C 集合类型的声明中，占位符名表示尖括号中的类型。比如，NSArray 的声明以如下内容开始：

```
@interface NSArray<ObjectType>
- (NSArray<ObjectType> *)arrayByAddingObject:(ObjectType)anObject;
// ...
```

第 1 行表示我们将要使用 ObjectType 作为元素类型的占位符名。第 2 行表示 arrayByAddingObject: 方法接收一个 ObjectType 元素类型的对象，并返回该元素类型的一个数组。如果某个数组声明为 NSArray<NSString*>*，那么 ObjectType 占位符就会被解析为 NSString*（从这里面可以看到为何 Apple 将其叫作“轻量级泛型”）。

## A.3 双语言目标

一个目标可以是双语言目标：既包含 Swift 文件又包含 Objective-C 文件的目标。出于几个原因，双语言目标是很有用的。你想要充分利用 Objective-C 的语言特性；想要利用上由 Objective-C 编写的第三方代码；想要利用自己使用 Objective-C 编写的既有代码。

应用本身原来可能是由 Objective-C 编写的，现在想要将其中一部分（或是逐步将全部代码）迁移到 Swift。

关键的问题是在单个目标中，Swift 与 Objective-C 如何能在第一时间就理解对方的代码。回想一下，与 Swift 不同，Objective-C 已经有可见性问题：Objective-C 文件不能自动看到彼此。相反，能看到其他 Objective-C 文件的每个 Objective-C 文件都需要显式声明才行，通常是在第 1 行顶部通过一个 `#import` 指令来实现的。为了避免私有信息的意外暴露，Objective-C 类声明按照惯例会位于两个以上的文件中：一个头文件（*.h*），它包含了 `@interface` 部分；一个代码文件（*.m*），它包含了 `@implementation` 部分。此外，只需要导入 *.h* 文件即可。这样，如果类成员、常量等的声明为公开的，那么它们就会被放到 .h 文件中。

Swift 与 Objective-C 之间的可见性取决于这种约定：这是通过 *.h* 文件实现的。有两个方向的可见性，需要分别对待：

*Swift 如何看到 Objective-C*

在将 Swift 文件添加到 Objective-C 目标中，或将 Objective-C 文件添加到 Swift 目标中时，Xcode 会创建一个桥接头文件。它在项目中是个 *.h* 文件。其默认名源自目标名（如，*MyCoolApp-Bridging-Header.h*），不过该名字是任意的，也可以修改，只要修改目标的 Objective-C Bridging Header 构建设置与之匹配即可。（与之类似，如果没有生成桥接头文件，后面又想拥有一个，那么可以手工创建一个 *.h* 文件，并在应用目标的 Objective-C Bridging Header 构建设置中指向它即可。）如果在该桥接头文件中 `#import` 它，那么 Objective-C *.h* 文件就会对 Swift 可见了。

*Objective-C 如何看到 Swift*

如果有了桥接头文件，那么在构建目标时，所有 Swift 文件的顶层声明都会自动转换为 Objective-C，并用于构建隐藏的桥接头文件，它位于该目标的 *Intermediates* 构建目录中，在 *DerivedData* 目录内。查看它的最简单的方式是使用如下终端命令：

```
$ find ~/Library/Developer/Xcode/DerivedData -name "*Swift.h"
```

上述命令会显示出隐藏的桥接头文件名。比如，对于名为 MyCoolApp 的目标来说，隐藏的桥接头文件叫作 *MyCoolApp-Swift.h*；名字可能会涉及一些转换；比如，目标名中的空格已经被转换为了下划线。此外，查看（或修改）目标的 Product Module Name 构建设置；隐藏的桥接头文件名源自于它。要想让 Objective-C 文件可以看到 Swift 声明，需要将这个隐藏的桥接头文件 `#import` 到需要看到它的每个 Objective-C 文件中。

出于简洁性的考虑，我分别称这两个桥接头文件为可见与不可见桥接头文件。

比如，假设向名为MyCoolApp的Swift目标添加了一个使用Objective-C编写的Thing类。它由两个文件构成，分别是*Thing.h*与*Thing.m*，那么：

- 要想让Swift代码能够看到Thing类，我需要在可见桥接头文件（*MyCoolApp-Bridging-Header.h*）中`#import "Thing.h"`。
- 要想让Thing类代码看到Swift声明，我需要在*Thing.m*顶部导入不可见桥接头文件（`#import "MyCoolApp-Swift.h"`）。

在此基础上，下面是将我自己的Objective-C应用转换为Swift应用的步骤：

1. 选取一个待转换为Swift的*.m*文件。Objective-C不能继承Swift类，因此如果使用Objective-C定义了一个类及其子类，那就从子类开始。将应用委托类留在最后。
2. 从目标中删除该*.m*文件。要想做到这一点，请选择该*.m*文件，然后使用文件查看器。
3. 在`#import`了相应*.h*文件的每个Objective-C文件中，删除该`#import`语句，并导入不可见桥接头文件（如果之前没有导入过）。
4. 如果在可见桥接头文件中导入了相应的*.h*文件，请删除`#import`语句。
5. 为该类创建*.swift*文件，请确保将其添加到目标中。
6. 在*.swift*文件中，声明类并为*.h*文件中声明为公开的所有成员提供桩声明。如果该类需要使用Cocoa协议，那就使用它们；还需要提供所需协议方法的桩声明。如果该文件引用了目标在Objective-C中声明的其他类，那么在可见桥接头文件中导入其*.h*文件。
7. 项目现在应该可以编译通过了！当然，没法使用，因为还没有在*.swift*文件中编写任何实际代码。不过，这都是小事！
8. 现在在*.swift*文件中编写代码。我的做法是逐行转换原始的Objective-C代码，这个因人而异。
9. 当*.m*文件中的代码全部被转换为了Swift后，构建、运行并测试。如果运行时说（很可能还会出现崩溃）找不到类，那就请在nib编辑器中寻找对其的所有引用，并在身份查看器中重新加入类名（按下Tab来设定修改）。保存并重试。
10. 进入下一个*.m*文件！重复上述所有步骤。
11. 转换完所有文件后，请转换应用委托类。这时，如果目标中已经没有Objective-C文件，那就请删除*main.m*文件（在应用委托类声明中将其替换为`@UIApplicationMain`特性）与*.pch*（预编译头文件）文件。

应用现在应该可以运行了，并且现在是由纯 Swift 编写的（至少是按照你的期望来的）。回过头来思考一下代码，使其更加符合 Swift 习惯。你会发现，Objective-C 中笨拙且难以解决的事情在 Swift 会变得更加简洁和清晰。

此外，还可以通过在 Swift 中对 Objective-C 进行扩展来部分转换 Objective-C 类。这对于整体的转换过程是很有帮助的，也可以只在 Swift 中编写一两个 Objective-C 类的方法，因为 Swift 能够很好地理解它们。不过，如果不是公开的， 那么 Swift 就无法看到 Objective-C 类的成员，因此之前在 Objective-C 类的 *.m* 文件中设定为私有的方法和属性需要在 *.h* 文件中进行声明。

## 作者介绍

**Matt Neuburg** 从 1968 年就开始学习计算机编程了，那时他才 14 岁，是一家地下高中俱乐部的成员，成员们每周见一次面，在银行的 PDP-10s 上进行分时操作，方式则是使用原始的电传打字机。他还偶尔使用过普林斯顿大学的 IBM-360/67，不过有一天弄丢了穿孔卡片，这令他感到非常沮丧，最后放弃了。他在斯沃斯莫尔学院主修希腊语，并于 1981 年获得康奈尔大学的博士学位，他在一台大型机上完成了博士论文（关于埃斯库罗斯）的编写工作。接下来，他在多家知名的高等院校教授古典语言、文学与文化课程，不过现在有很多高校都否认他所讲授的知识。他发表过大量学术文章，但这些文章对人毫无吸引力。与此同时，他收到了一台 Apple IIc，在绝望中又开始从事计算机工作，后来在 1990 年迁移到了一台 Macintosh 上。他编写过一些教育与实用的自由软件，并且成为在线杂志 TidBITS 早期的一位定期撰稿人，他在 1995 年离开了学术界并开始编辑 MacTech 杂志。在 1996 年 8 月，他成为了一名自由职业者，这意味着从那时起他一直在寻找工作。他是《Frontier: The Definitive Guide》《REALbasic: The Definitive Guide》及《AppleScript: The Definitive Guide》的作者，同时也是《Programming iOS 7》（由 O'Reilly Media 出版）及《Take Control of Using Mountain Lion》（由 TidBITS Publishing 出版）的作者。

## 封面介绍

本书封面的动物是一只格陵兰海豹（Pagophilus groenlandicus），这是个拉丁语名字，表示“来自格陵兰的冰块爱好者”。这些动物生长在北大西洋与北冰洋，大部分时间都待在水中，只有在生产和脱毛的时候才浮出冰面。由于耳朵是密闭的，其流线型的身体与节省体力的游泳方式使得他们非常适合于水栖生活。虽然海狮之类的有耳物种是游泳好手，但他们却是半水栖的，因为它们在陆地上交配和休息。

格陵兰海豹拥有银灰色的皮毛，后背上有一个巨大的黑色标记，像是一个竖琴或是叉骨。成年格陵兰海豹会长至 5 到 6 英尺长，300 到 400 磅重。由于栖息地寒冷，它们有着一层厚厚的鲸脂来隔绝外界。格陵兰海豹的饮食变化多样，包括几种鱼类与甲壳类动物。他们可以在水下潜伏 16 分钟左右来寻觅食物，并且可以潜入几百英尺的水下。

刚出生的格陵兰海豹并没有任何防护装备，它们通过白色的皮毛来保暖的，皮毛会吸收阳光的热量。12 天后，小格陵兰海豹就会被父母丢弃，并且因为吃母乳，其体重会长到刚出生时的 3 倍。随后的几周一直到小格陵兰海豹可以在冰上游走为止，它们都非常容易遭受来自食肉动物的攻击，并且体重会减轻一半。幸存下来的格陵兰海豹会在 4 到 8 年后成熟（取决于性别），其平均寿命大约为 35 岁。

格陵兰海豹会在加拿大、挪威、俄罗斯与格陵兰的海岸遭到捕杀，其肉、油与皮毛会被拿来交易。虽然一些政府出台了条例并强制捕杀配额，但每年被捕杀的格陵兰海豹数量都会被少报。不过，自然资源保护论者的疾呼与努力使得市场上对于海豹皮毛与其他商品的需求量降低了。

本书封面图片来自于 Wood 的 Animate Creation。

O'REILLY®

# iOS编程基础：Swift、Xcode和Cocoa入门指南

通过掌握重要的基础知识来进入iOS开发的世界，本书主要介绍Xcode IDE、Cocoa Touch框架与Swift 2.0。通过阅读本书，你将学到Swift面向对象的概念、理解如何使用Apple的开发工具，以及探索Cocoa是如何提供iOS应用所需的底层功能的。

"Neuburg是我最喜欢的编程图书作者。"

——John Gruber
Daring Fireball

- 探索Swift面向对象的概念：变量与函数、作用域与命名空间，对象类型与实例
- 熟悉内建的Swift类型，如数字、字符串、范围、元组、Optional、数组、字典与集合
- 学习如何声明、实例化与定制Swift对象类型
- 探索强大的Swift特性，如协议与泛型
- 学习Swift 2.0的新特性：选项集合、协议扩展、错误处理、guard语句、可用性检测等
- 了解Xcode项目的生命周期
- 使用nib、nib编辑器Interface Builder来创建应用界面
- 理解Cocoa的事件驱动模型与主要的设计模式和特性
- 掌握Swift与Cocoa的C和Objective-C API通信的方式

**Matt Neuburg**，拥有古典文学博士学位，曾在多所大学授课。他是《MacTech》杂志的编辑以及TidBITS的特约编辑。Neuburg编写过很多OS X与iOS应用。此前曾出版过《Programming iOS 8》《REALbasic: The Definitive Guide》以及《AppleScript: The Definitive Guide》等著作。

MOBILE DEVELOPMENT / IOS

O'Reilly Media, Inc.授权机械工业出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/移动开发
ISBN 978-7-111-55635-0
9 787111 556350 >
定价：129.00元