

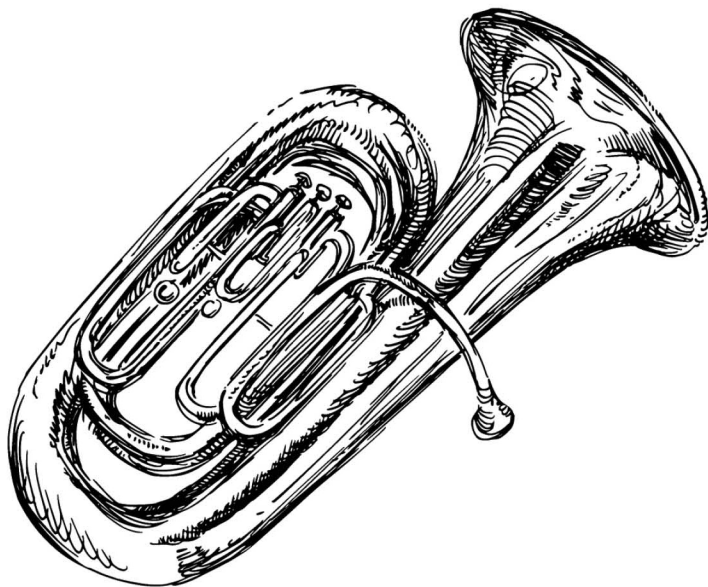
从基本原则、惯用法、语法、内存管理、设计、实现、设计模式、兼容性和性能优化等方面深入探讨编写高质量Objective-C代码的技巧、禁忌和最佳实践

Writing Solid Objective-C Code  
61 Suggestions to Improve Your Objective-C Program

# 编写高质量代码

## 改善Objective-C程序的 61个建议

刘一道 著



机械工业出版社  
China Machine Press

Effective 系列丛书

# 编写高质量代码：改善 Objective-C 程序的 61 个建议

刘一道 著



机械工业出版社  
China Machine Press

## 图书在版编目 (CIP) 数据

编写高质量代码：改善 Objective-C 程序的 61 个建议 / 刘一道著. —北京：机械工业出版社，2015.8

(Effective 系列丛书)

ISBN 978-7-111-51463-3

I. 编… II. 刘… III. C 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2015) 第 213529 号



## 编写高质量代码：改善 Objective-C 程序的 61 个建议

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：孙海亮

责任校对：董纪丽

印 刷：

版 次：2015 年 9 月第 1 版第 1 次印刷

开 本：186mm×240mm 1/16

印 张：14

书 号：ISBN 978-7-111-51463-3

定 价：59.00 元

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

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzit@hzbook.com

版权所有·侵权必究

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

## 如何写出高质量的代码？

我一直在思考，如何才能编写出高质量、优秀的代码，我也在不停地探寻，希望找出类似于武侠小说中所说的武功秘籍，在编写代码一途可以帮助大家走“捷径”从而达到事半功倍的效果。

《道德经》第四十八章中说“为学者日益，为道者日损。损之又损，以至于无为。无为而不为”。这句话是说，治学上，不要过于追求外在的经验知识，否则经验知识越积累增多，就会越僵化臃肿。要学会透过直观体悟把握事物未分化时的状态或者内索自身虚静，洞悉其内在的道化真谛，从而简之再简。这些也就是我们现在说的“大道至简”。

治学如此，写代码更是如此。在程序员写代码的职业生涯中，前 5 年，他看到的只是一行一行的代码，他会为自己洋洋洒洒写成的代码而陶醉；5 年之后，就不是单纯地写代码了，而是在做一件艺术品，此时的程序员就像雕刻家一样，在刻下每一刀之前，都需纵观全局，细细揣摩，落刀如有神，一气呵成。故此，写出优秀的高质量代码，需要像唐僧西天取经一样，踏踏实实，用平常心闯过一关又一关，如此，写出高质量的代码自然就是水到渠成的事了。写代码时切忌心态浮躁，急功近利。

## 本书适合哪些读者

本书不是一本介绍“Objective-C”代码如何编写的入门级的书籍。故此，如果你只想初步了解一下“Objective-C”开发，而不想做深入研究的话，那么本书就不适合你了。

本书主要面向专业从事 Objective-C 开发或者想转向“Objective-C”开发的研究人员，帮助其编写便于维护、执行迅速且不易出错的代码。如果你是“Objective-C”开发技术大

咖，翻阅本书，对你来说可能会有些浪费时间，故此也请你一瞥而过！

本书主要适合如下读者：

- ❑ 对软件开发，特别是对 Objective-C 开发有兴趣的人。
- ❑ 想成为一名专职的软件开发人员的人。
- ❑ 想进一步提高自己“Objective-C”技术水平的在校学生。
- ❑ 开设相关专业课程的大专院校的师生。

## 你该如何阅读本书

本书共 9 章，从内容上可以分 3 部分<sup>⊖</sup>。大家可以根据自身状况，选择性跳读，翻阅自己最感兴趣或与当前工作相关的章节来读。下面把这几章简单归类评述，为你进行选择性跳读时提供参考：

**第一部分（第 1 ~ 5 章）：**主要围绕如何写出高质量代码提出一些建议。这些建议一部分来源于苹果每年举行的一些开发大会，主要是针对 Objective-C 语法、性能与功能改进进行的阐述和解惑；另一部分来自于一线开发者平时工作中的点点滴滴的感受。作为笔者，我不过是把这点点滴滴像串珍珠一样把它们串起来，形成一个实用的、具有整体性的建议。

**第 1 章** 作为本书的首章，我感觉唠唠“Objective-C 的那些事儿”很有必要。希望通过这一章的唠叨能让你有所收获。

**第 2 章** 主讲数据类型、集合和控制语句。《道德经》第五十二章中有言“天下有始，以为天下母。既得其母，以知其子；既知其子，复守其母。”掌握了基本的开发语言，就可以“知其子”——程序；反过来，通过开发程序，又可以“复守其母”——开发语言，即通过开发程序可以进一步巩固、加深对开发语言的理解。在 Objective-C 中，构建语言的基本单元——值和集合，更是重中之重。本章将围绕这些构建开发语言的“基石”从不同角度加以阐述。

**第 3 章** 主讲内存管理。内存管理曾经是程序员的噩梦，特别是在面向过程开发的过程中。虽然，在纯面向对象的开发语言中，例如 C# 和 Java 等开发语言，有了自动内存垃圾回收的机制，但依赖这种自动内存垃圾回收机制的代价往往很高，容易造成程序性能的耗损，进而容易产生难以突破的“性能劲瓶”。因此，在编写高性能的应用程序，特别是服务器程序时，不得不依赖 C、Pascal 和 C++ 等非纯面向对象的语言来完成。现在，众多的 Android 手机或者平板上遇到的 App 莫名其妙崩溃的现象，在很大程度上与 Android 底层的内存回收处理不当有很大的关系。

---

⊖ 这只是逻辑上的划分，为了不影响读者选择性跳读，正文中未明确体现。

在本书定稿之时，虽然，Objective-C 已经具有了“垃圾自动回收”的机制，但洞悉其内存的管理机制，写出高质量的代码，还是至关重要的。

**第 4 章 主讲设计与声明。**《庄子·齐物论》中曰：“昔者庄周梦为蝴蝶，……不知周之梦为蝴蝶与？蝴蝶之梦为周与？”意思是说，庄子梦见自己变成蝴蝶，醒来后发现自己还是庄子。庄子分不清自己是梦中变成蝴蝶，还是蝴蝶梦中变成庄子了。在 Objective-C 中，类与对象的关系，就有些类似庄子和蝴蝶的关系，从一定的角度来看，类就是对象，而从另外一个角度来看，对象又是类。本章就以庄周梦蝶做引子，来谈谈设计和声明。

**第 5 章 众所周知，在面向对象的设计中，对于一个类，完成其定义，要包含类名、实例变量及方法的定义和声明，但这最多也只能算是完成 20% 的工作，其余 80% 的工作主要在于其实现。如何把类的实现写好，很考验一个人的功底。尽可能利用开发语言本身一些特性，是实现类的一个比较有效的途径。本章就结合 Objective-C 的特性，来介绍在类的实现中一些可利用的做法。**

**第二部分（第 6 ~ 8 章），**相对于第一部分，本部分更多关注一些“习以为常”的理念，结合 Objective-C 语言，进行一番新的解读。这几章里面，很多东西你也许早已经知道，但一些理念如何融入自己的代码和设计中，却是更高的要求。正如巴菲特的理财理念，很多人都能知道，且能背诵得滚瓜烂熟，而在实际操作上，结果却大相径庭，其主要原因是，很多“理念”自己不过是记住罢了，并未已融入自己的血液和骨髓里，在实际操作上，仍旧沉溺于“旧习”。

**第 6 章 主讲继承与面向对象的设计。**随着这几年面向对象编程（OOP）越来越盛行，在编程领域，现在几乎是面向对象编程语言的天下。继承，作为面向对象编程的三大特征（继承、多态和封装）之一，起着核心的作用。但不同面向对象的编程语言实现的方式，有着比较大的差异性。本章从纵向和横向两个维度，来阐述“继承”在 Objective-C 开发语言中的多面性，使你娴熟掌握“继承”在 Objective-C 开发语言中的“有所为而有所不为”。

**第 7 章 主讲设计模式与 Cocoa 编程。**设计模式是一种设计模板，用于解决在特定环境中反复出现的一般性问题。它是一种抽象工具，在架构、工程和软件开发领域相当有用。本章将介绍什么是设计模式，解释为什么设计模式在面向对象的设计中非常重要，并讨论一个设计模式的实例。

**第 8 章 主讲定制 init... 和 dealloc。**在 Objective-C 中，没有 new 和 delete 这两个关键字，但存在着 alloc 和 init...，它们承担着与 new 类似的功能，前者用于处理内存的分配，后者用于在分配的内存中进行数据的初始化；与 delete 类似的是 dealloc。掌握 init...，对于写出高性能的应用是至关重要的。

**第三部分（第 9 章），**在本书将近定稿时，恰遇到 Swift 的新版本推出，为了解决开发者在混用 Objective-C 和 Swift 过程中遇到的一些疑难点，故此添加了本部分。如果对

Swift “不感冒”，那么本部分你就没有阅读的必要性了。

**第9章 主讲 Objective-C 和 Swift 的兼容性。**Swift 是苹果公司在 WWDC 2014 新发布的一门编程语言，用来撰写 OS X 和 iOS 应用程序，但是 Swift 的推出，使很多原先很熟练使用 Objective-C 的码农产生了一定程度的“惊惶”：担忧苹果公司放弃对 Objective-C 的支持，不得不花费一定的时间和精力来学习一门新的开发语言——Swift。Xcode 6 或者更高的版本支持二者的相互兼容。也就是说，在 Objective-C 文件中可以使用 Swift 编写的代码，也可以在 Swift 文件中使用 Objective-C 编写的代码，但其相互兼容性是有条件的。本章就来讲解二者的兼容性问题。

本书没有采取循序渐进的方式。故此，读者可以根据“兴趣”选择自己喜欢的章节来阅读，这是最有效的读书方式，也是笔者推崇的读书方式。

## 勘误和支持

除封面署名外，参加本书编写工作的还有孙振江、陈连增、边伟、郭合苍、郑军、吴景峰、杨珍民、王文朝、崔少闯、韦闪雷、刘红娇、王洁、于雪龙、孔琴。由于笔者的水平有限，编写时间仓促，书中难免会出现一些错误或者不准确的地方，恳请读者批评指正。

书中的不足之处，还望各位读者多提意见，欢迎发送邮件至邮箱 [yifeilang@aliyun.com](mailto:yifeilang@aliyun.com)，期待能够得到你们的真挚反馈。

## 感恩致谢

本书之所以能出版，多亏华章公司的编辑孙海亮先生多次催促，不断地给予我鼓励。期间华章群一些网友的鼓励，也使我受益匪浅，在此向他们表示感谢。最后，要感谢的就是我亲爱的读者，感谢你拿起这本书，你的认可，就是我的最大的快乐。

刘一道  
于北京



## 前 言

<b>第 1 章 让自己习惯 Objective-C</b>	1
建议 1: 视 Objective-C 为一门动态语言	1
建议 2: 在头文件中尽量减少其他头文件的引用	6
建议 3: 尽量使用 const、enum 来替换预处理 #define	10
建议 4: 优先使用对象字面量语法而非等效方法	13
建议 5: 处理隐藏的返回类型, 优先选择实例类型而非 id	17
建议 6: 尽量使用模块方式与多类建立复合关系	19
建议 7: 明解 Objective-C++ 中的有所为而有所不为	23
<b>第 2 章 数据类型、集合和控制语句</b>	28
建议 8: C 语言与 Objective-C 语言的关系是充分而非必要条件	28
建议 9: 高度警惕空指针和野指针的袭击	31
建议 10: 在 64 位环境下尽可能利用标记指针	35
建议 11: 谨记兼容 32 位和 64 位环境下代码编写事项	38
建议 12: 清楚常量字符串和一般字符串的区别	43
建议 13: 在访问集合时要优先考虑使用快速枚举	44
建议 14: 有序对象适宜存于数组, 而无序对象适宜存于集	48
建议 15: 存在公共键时, 字典是在对象之间传递信息的绝佳方式	53
建议 16: 明智而审慎地使用 BOOL 类型	55



<b>第 3 章 内存管理</b>	57
建议 17: 理解内存和 Objective-C 内存管理规则	57
建议 18: 内存管理讲究“好借好还, 再借不难”	61
建议 19: 区别开 alloc、init、retain、release 和 dealloc 之间的差异	63
建议 20: 优先选用存取方法来简化内存管理	66
建议 21: 对象销毁或者被移除一定考虑所有权的释放	70
建议 22: 明智而审慎地使用 dealloc	73
<b>第 4 章 设计与声明</b>	75
建议 23: 编写代码要遵守 Cocoa API 约定	75
建议 24: 洞悉实例变量	77
建议 25: 透彻了解属性的里里外外	81
建议 26: 存取方法是良好的类接口必要组成部分	85
建议 27: 明晓类公共领域的方法都是虚方法	87
建议 28: 初始化还是解码取决于是否支持归档和解档	92
建议 29: 利用键-值机制访问类的私有成员变量和方法	93
建议 30: 浅复制适宜指针而深复制适宜数据	101
建议 31: 明智而审慎地使用 NSCopying	103
建议 32: 使用协议来实现匿名对象的提供	106
<b>第 5 章 实现</b>	108
建议 33: 使用类别把类的实现拆分成不同的文件	108
建议 34: 明智地使用内省可使程序更加高效和健壮	109
建议 35: 尽量使用不可变性对象而非可变性对象	113
建议 36: 利用复合能巧妙地把两个类或两个对象融合	115
建议 37: 使用类扩展来隐藏实现的细节	120
建议 38: 使用内联块应注意避免循环引用	122
建议 39: 利用类别把方法添加到现有的类	124
建议 40: 通过强弱引用来管理对象的所有权	127
<b>第 6 章 继承与面向对象设计</b>	133
建议 41: 明确 isa 在继承上的作用	133

建议 42: 利用类别和协议实现类似多重继承的机制 .....	136
建议 43: 类别和类扩展是类继承的延续性拓展 .....	139
建议 44: 继承基类的实现行为勿忘调用 <code>super</code> .....	141
<b>第 7 章 设计模式与 Cocoa 编程 .....</b>	<b>145</b>
建议 45: 设计模式是特定环境下的特定问题的解决方案 .....	145
建议 46: MVC 模式是一种复合或聚合模式 .....	147
建议 47: 对象建模在数据库中也广泛使用 .....	155
建议 48: 类簇可简化框架的公开架构而又不减少功能的丰富性 .....	160
建议 49: 委托用于界面控制, 而数据源用于数据控制 .....	165
<b>第 8 章 定制 <code>init...</code> 和 <code>dealloc</code> .....</b>	<b>171</b>
建议 50: 了解对象的 <code>alloc</code> 和 <code>init...</code> .....	171
建议 51: 直接访问实例变量的 <code>init...</code> 方法 .....	174
建议 52: 初始化方法必须以 “init” 字母开头 .....	176
建议 53: 从 <code>init...</code> 方法得到的对象可能是不想要的 .....	177
建议 54: 实现 <code>init...</code> 方法的唯一性或者指定性并非 “不可能” .....	179
建议 55: <code>init...</code> 方法有 “轻重级别” 之分 .....	181
<b>第 9 章 Objective-C 与 Swift 的兼容性 .....</b>	<b>184</b>
建议 56: Objective-C 和 Swift 的互用性基于映射机制 .....	184
建议 57: 利用 Swift 的特性可增强已有的 Objective-C 代码 .....	191
建议 58: 洞悉 Objective-C 和 Swift 类型转换的处理机制 .....	194
建议 59: C 语言的数据类型在 Swift 中 “有所变有所不变” .....	199
建议 60: Swift 和 Objective-C 兼容性是基于混搭机制 .....	204
建议 61: 利用迁移机制实现 Objective-C 代码的重生 .....	209

# 让自己习惯 Objective-C

听说有本书叫《明朝那些事儿》，很多人喜欢，但我没有看过。我这人有些“老帽儿”，对一些时髦的东西不是很感兴趣。我看的都是那些经过千百年“浪沙冲洗”沉淀下来的书籍（技术书籍除外）。作为本书的首章，我感觉聊一些“Objective-C 的那些事儿”很有必要，希望读者能有所收获。

## 建议 1：视 Objective-C 为一门动态语言

Objective-C，是美国人布莱德·确斯（Brad Cox，见图 1-1）于 1980 年年初发明的一种程序设计语言，其与同时代的 C++ 一样，都是在 C 的基础上加入面向对象特性扩充而成的。C++ 如日中天，红火已有 30 年之久，而 Objective-C 到 2010 年才被人注意，并逐渐红火起来。造成这种结果的原因跟其版权为苹果独自拥有和苹果自我封闭性、不作为性有很大的关系。这也是 Objective-C 在语法上跟其他语言（如 C++、Pascal、Java 和 C#）等相比有些不足的原因。可喜的是，苹果公司于 2010 年起，在每年的苹果年度发布大会上都会针对 Objective-C 语言的语法发布新的改良版本。

虽然 Objective-C 和 C++ 都是在 C 的基础上加入面向对象特性扩充而成的程序设计语言，但二者实现的机制差异很大。Objective-C 基于动态运行时类型，而 C++ 基于静态类型。也就是说，用 Objective-C 编写的程序不能直接编译成可令机器读懂



图 1-1 布莱德·确斯

的机器语言（二进制编码），而是在程序运行时，通过运行时（Runtime）把程序转译成可令机器读懂的机器语言；用 C++ 编写的程序，编译时就直接编译成了可令机器读懂的机器语言。这也就是为什么把 Objective-C 视为一门动态开发语言的原因。

从原理上来讲，目前常用的 Java 和 C# 等程序开发语言都是动态开发语言，只不过 Java 用的是虚拟机 JVM（Java Virtual Machine），如图 1-2 所示，C# 用的是公共语言运行时 CLR（Common Language Runtime）。与此相对，C++ 可视为静态开发语言。

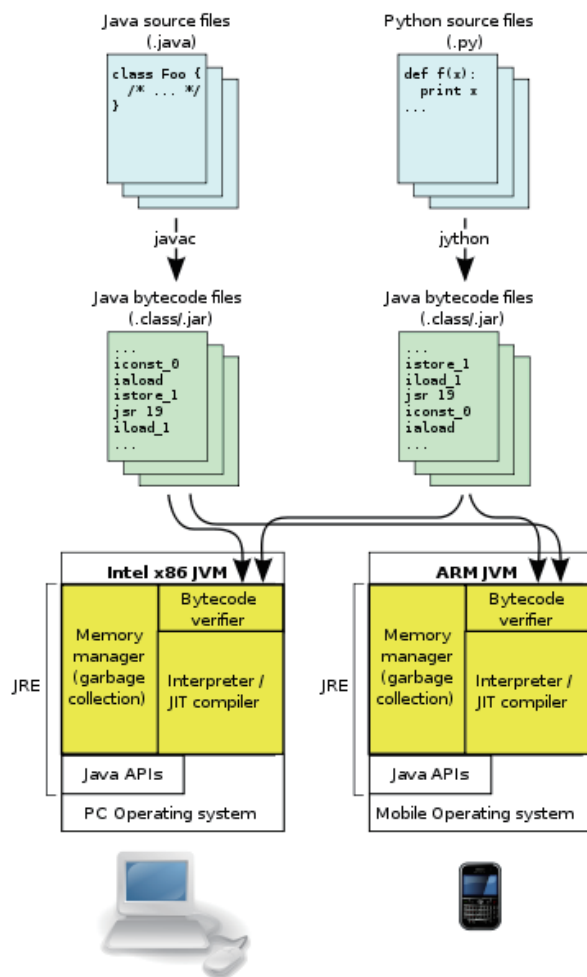


图 1-2 Java 虚拟机实现机制

Objective-C 作为一门动态开发语言，会尽可能地将编译和链接时要做的事情推迟到运行时。只要有可能，Objective-C 总是使用动态的方式来解决问题。这意味着 Objective-C 语言不仅需要一个编译环境，同时也需要一个运行时系统来执行编译好的代码。

而运行时 (Runtime) 正是扮演着这样的角色。在一定程度上, 运行时系统类似于 Objective-C 语言的操作系统, Objective-C 就基于该系统来工作。因此, 运行时系统好比 Objective-C 的灵魂, 很多东西都是在这个基础上出现的。很多开发人员常常对运行时系统充满疑惑, 而恰恰这是一个非常重要的概念。可以这么问: “如果让你实现 (设计) 一个计算机语言, 你会如何下手?” 很少程序员这么思考过。但是这么一问, 就会强迫你从更高层次来思考以前的问题了。

要想理解 “Objective-C 是一门动态开发语言”, 就不得不理解开发语言的三个不同层次。

(1) 传统的面向过程的语言开发。例如 C 语言, 实现 C 语言编译器很简单, 只要按照语法规则实现一个 LALR 语法分析器就可以了。编译器优化是非常难的, 不在本节讨论的范围内。这里实现了编译器中最基础和原始的目标之一, 就是把一份代码里的函数名称转化成一个相对内存地址, 把调用这个函数的语句转换成一个跳转指令。在程序开始运行时, 调用语句可以正确跳转到对应的函数地址。这样很好, 也很直白, 但是太死板了。

(2) 改进的开发面向对象的语言。相对面向过程的语言来说, 面向对象的语言更加灵活了。例如 C++, C++ 在 C 的基础上增加了类的部分。但这到底意味着什么呢? 再写它的编译器要如何考虑呢? 其实, 就是让编译器多绕个弯, 在严格的 C 编译器上增加一层类处理的机制, 把一个函数限制在它处在的类环境里, 每次请求一个函数调用, 先找到它的对象, 其类型、返回值、参数等确定后再跳转到需要的函数。这样很多程序增加了灵活性, 同样一个函数调用会根据请求参数和类的环境返回完全不同的结果。增加类机制后, 就模拟了现实世界的抽象模式, 不同的对象有不同的属性和方法。同样的方法, 不同的类有不同的行为! 这里大家就可以看到作为一个编译器开发者都做了哪些进一步的思考。虽然面相对象的语言有所改进, 但还是死板, 故此仍称 C++ 是静态语言。

(3) 动态开发语言。希望更加灵活, 于是完全把上面那个类的实现部分抽象出来, 做成一套完整运行阶段的检测环境, 形成动态开发语言。这次再写编译器甚至保留部分代码里的 syntax 名称, 名称错误检测, 运行时系统环境注册所有全局的类、函数、变量等信息, 可以无限地为这个层增加必要的功能。调用函数时, 会先从这个运行时系统环境里检测所有可能的参数再做 jmp 跳转。这就是运行时系统。编译器开发起来比上面更加绕弯, 但是这个层极大地增加了程序的灵活性。例如, 当调用一个函数时, 前两种语言很有可能一个跳到了一个非法地址导致程序崩溃, 但是在这个层次里面, 运行时系统过滤掉了这些可能性。这就是动态开发语言更加强壮的原因, 因为编译器和运行时系统环境开发人员已经帮你处理了这些问题。

好了, 上面说了这么多, 再返回来看 Objective-C 的这些语句:

```
id obj=self;
if ([obj respondsToSelector:@selector(function1:)] {
```

#### 4 ❖ 编写高质量代码：改善Objective-C程序的61个建议

```
}
if ([obj isKindOfClass:[NSArray class]] ) {
}
if ([obj conformsToProtocol:@protocol(myProtocol)]) {
}
if ([[obj class] isKindOfClass:[NSArray class]]) {
}
[obj someNonExistFunction];
```

看似很简单的语句，但是为了让语言实现这个能力，语言开发者要付出很多努力来实现运行时系统环境。这里运行时系统环境处理了弱类型及函数存在检查工作。运行时系统会检测注册列表里是否存在对应的函数，类型是否正确，最后确定正确的函数地址，再进行保存寄存器状态、压栈、函数调用等实际的操作。

```
id knife=[Knife grateKnife];
NSArray *monsterList=[NSArray array];
[monsterList makeObjectsPerformSelector:@selector(killMonster:)
withObject:knife];
```

用 C 和 C++ 完成这个功能还是比较麻烦的，但是动态语言处理却非常简单，并且这些语句让 Objective-C 语言更加直观。

在 Objective-C 中，针对对象的函数调用将不再是普通的函数调用：

```
[obj function1With:var1];
```

这样的函数调用将被运行时系统环境转换成：

```
objc_msgSend(target,@selector(function1With:),var1);
```

Objective-C 的运行时系统环境是开源的，这里可以进行简单介绍，可以看到 objc\_msgSend 由汇编语言实现，甚至不必阅读代码，只需查看注释就可以了解。运行时系统环境在函数调用前做了比较全面的安全检查，已确保动态语言函数调用不会导致程序崩溃。对于希望深入学习的朋友可以自行下载 Objective-C 的运行时系统源代码来阅读，进行更深入的了解。

```
/* *****
 * id      objc_msgSend(id self,
 *      SEL op,
 *      ...)
 *
 * a1 是消息接收者 ,a2 是选择器
 * ***** */
ENTRY objc_msgSend
# 检查接收者是否空
    teq    a1, #0
    moveq  a2, #0
    bxeq   lr

# 保存寄存器 (对象), 通过在缓存里查找, 来加载接收者 (对象) 类
```

```

    stmfd    sp!, {a4,v1-v3}
    ldr      v1, [a1, #ISA]

# 寄存器(对象)非空, 缓存里可查找到
    CacheLookup a2, LMsgSendCacheMiss

# 缓存碰撞 (imp in ip) ——准备转发, 恢复寄存器和调用
    teq v1, v1      /* 设置 nonstret (eq) */
    ldmfd    sp!, {a4,v1-v3}
    bx      ip

# 缓存漏掉: 查找方法列表
LMsgSendCacheMiss:
    ldmfd    sp!, {a4,v1-v3}
    b       _objc_msgSend_uncached

LMsgSendExit:
    END_ENTRY objc_msgSend

    .text
    .align 2
_objc_msgSend_uncached:

# 压入堆栈帧
    stmfd    sp!, {a1-a4,r7,lr}
    add      r7, sp, #16
    SAVE_VFP

# 加载类和选择器
    ldr a1, [a1, #ISA]      /* class = receiver->isa */
    # MOVE    a2, a2        /* 选择器已经在 a2 */

# 做查找
    MI_CALL_EXTERNAL(__class_lookupMethodAndLoadCache)
    MOVE     ip, a1

# 准备转发, 挤出堆栈帧并且调用 imp
    teq v1, v1      /* 设置 nonstret (eq) */
    RESTORE_VFP
    ldmfd    sp!, {a1-a4,r7,lr}
    bx      ip

```

现在说一下动态开发语言的负面影响, 其负面影响可以归纳为两方面。

## 1. 执行效率问题

“静态开发语言执行效率要比动态开发语言高”, 这句没错。因为一部分 CPU 计算损耗在了运行时系统过程中, 而从上面的汇编代码也可以看出大概损耗在哪些地方。而静态语言生成的机器指令更简洁。正因为知道这个原因, 所以开发语言的人付出很大一部分努力



来保持运行时系统的小巧。所以，Objective-C 是 C 的超集加上一个小巧的运行时系统环境。但是，换句话说，从算法角度考虑，这点复杂度是可忽略的。

## 2. 安全性问题

动态语言由于运行时系统环境的需求，会保留一些源码级别的程序结构。这样就给破解带来了方便之门。一个现成的说明就是 Java, 大家都知道 Java 运行在 jre 上面，这就是典型的运行时例子。它的执行文件 .class 全部可以反编译回近似源代码。所以，这里的额外提示就是，如果需要写和安全有关的代码，离 Objective-C 远点，直接用 C。

简单理解：“Runtime is everything between your each function call.”

但是大家要明白，提到运行时系统并不只是因为它带来了这些简便的语言特性，而是这些简单的语言特性在实际运用中，需要从完全不同的角度考虑。



### 要点

- (1) Objective-C 是动态语言，C++ 是静态语言。
- (2) 静态语言执行效率和安全性要比动态语言高，但其简便性没有动态语言高。
- (3) 运行时 (Runtime) 环境可处理弱类型、函数存在检查工作，会检测注册列表里是否存在对应的函数，类型是否正确，最后确定正确的函数地址，再进行保存寄存器状态、压栈、函数调用等实际操作，确保了 Objective-C 的灵活性。

## 建议 2：在头文件中尽量减少其他头文件的引用

在面向对象开发语言中，如 C++、C#、Java 等语言中，对于类的描述，通常划分为头文件和源文件。头文件用于描述类的声明和可公开部分，而源文件用于描述类的方法或函数的具体实现，这也体现了面向对象语言的“封闭性”和“高内聚低耦合”的特性。而对于基于面向对象而设计的 Objective-C 也不例外，类分为头文件 (.h) 和源文件 (.m)。

在 OOP 编程中有两个技术用于描述类与类或对象与对象之间的关系：一个是继承；另一个是复合。在 Objective-C 中，当一个类需要引用另一个类，即建立复合关系时，需要在类的头文件 (.h) 中，通过“#import”修饰符来建立被引用类的指针。例如 Car.h：

```
// Car.h

#import <Foundation/Foundation.h>
@interface Car:NSObject
{
    Tire *tires[4];
```

```

    Engine *engine;
}

- (void) setEngine: (Engine *) newEngine;
- (Engine *) engine;
- (void) setTire: (Tire *) tire atIndex: (int) index;
- (Tire *) tireAtIndex: (int) index;
- (void) print;
@end // Car

```

在这里先省略类 Car 的源文件 (.m)。对于上面的代码，如果直接这么编译，编译器会报错，提示它不知道 Tire 和 Engine 是什么。为了使上面的代码能编译通过，在上面代码中就不得不添加对类 Tire 和 Engine 的头文件 (.h) 的引用，即通过关键字 “#import” 来建立起它们之间的复合关系。

要建立正确的复合关系，正确的代码写法如下：

```

// Car.h

#import <Foundation/Foundation.h>
#import "Tire.h"
#import "Engine.h"

@interface Car: NSObject
{
    Tire *tires[4];
    Engine *engine;
}

- (void) setEngine: (Engine *) newEngine;
- (Engine *) engine;
- (void) setTire: (Tire *) tire atIndex: (int) index;
- (Tire *) tireAtIndex: (int) index;
- (void) print;
@end // Car

```

现在，上面的代码虽然能正确编译了，但从“代码的高品质高安全”角度来看，在使用 “#import” 建立类之间的复合关系时，也暴露了所引用类 Tire 和 Engine 的实体变量和方法，与只需知道有一个类名叫 Tire 和 Engine 的初衷有些违背。在解决问题的同时，也带来了代码的安全性问题。

那么如何解决上面的问题呢？可以使用关键字 @class 来告诉编译器：这是一个类，所以只需要通过指针来引用它。它并不需要知道关于这个类的更多信息，只要了解它是通过指针引用即可，减少由依赖关系引起的重新编译所产生的影响。

对于上面的代码，通过 @Class 即可来建立对于类 Tire 和 Engine 的引用，具体写法如下：

```
// Car.h

#import <Foundation/Foundation.h>
@class Tire
@class Engine

@interface Car:NSObject
{
    Tire *tires[4];
    Engine *engine;
}

- (void) setEngine: (Engine *) newEngine;
- (Engine *) engine;
- (void) setTire: (Tire *) tire atIndex: (int) index;
- (Tire *) tireAtIndex: (int) index;
- (void) print;
@end // Car
```

上面介绍了使用“#import”和“@class”在“依赖关系”方面所产生的影响。同时二者在编译效率方面也存在巨大的差异。假如，有 100 个头文件，都用“#import”引用了同一个头文件，或者这些文件是依次引用的，如 A→B、B→C、C→D 这样的引用关系。当最开始的那个头文件有变化时，后面所有引用它的类都需要重新编译，如果自己的类有很多的话，这将耗费大量的时间，而使用 @class 则不会。

对于初学者，最容易犯“类循环依赖”错误。所谓的“类循环依赖”，也就是说，两个类相互引用对方。在本条款最初的 Car.h 头文件中，通过“#import”引用了 Tire.h 头文件，假如在 Tire.h 头文件里引用 Car.h 头文件，即如下：

```
// Tire.h

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

上面的代码进行编译时会出现编译错误，如果使用 @class 在两个类的头文件中相互声明，则不会有编译错误出现。虽然使用 @class 不会出现编译错误，但还是尽量避免这种“类循环依赖”的出现，因为这样容易造成类之间“高耦合”现象的产生，给以后代码的维护和管理带来很大的麻烦。

“#import”并非一无是处。既然“#import”与“@class”相比有很多不足，那么是否可以用“@class”来完全代替“#import”？不可以，在一个头文件（.h）中包含多个类的声明定义时，要与该头文件声明的多个类建立复合关系，比较好的方式是，采用关键字“#import”来建立复合关系。

例如，下面是头文件 PersonType.h 的定义：

```
//PersonType.h

#import <Foundation/Foundation.h>

// Person 类的声明定义
@interface Person:NSObject
{
    NSInteger    *sexType;
}
@property (nonatomic copy) NSString    *firstname;
@property (nonatomic copy) NSString    *lastname;

- (void) setSexType: : (int) index;
@end
// man 类的声明定义
@interface man:Person
end

//woman 类的声明定义
@interface woman:Person
end;
```

要与上面头文件 `PersonType.h` 中所声明的类建立复合关系，这个时候就不得不用关键字 “`#import`”。使用 “`#import`” 建立复合关系，会把所引用的头文件（.h）的所有类进行预编译，这样就会消耗很长时间。是否是有一种更好的方式来处理这个问题，请参阅“条款 9 尽量使用模块方式与多类建立复合关系”。

一般来说，关键字 “`@class`” 放在头文件中只是为了在头文件中引用这个类，把这个类作为一个类型来用。这就要求引用的头文件（.h）名与类的名称一致，且在类头文件（.h）只包含该类的声明定义的情况下，才可以使用关键字 “`@class`” 来建立复合关系。同时，在实现这个类的接口的类源文件（.m）中，如果需要引用这个类的实体变量或方法等，还需要通过 “`#import`” 把在 “`@class`” 中声明的类引用进来。例如下面的类 `a` 引用类 `Rectangle` 的示例：

```
a.h
@class Rectangle;
@interface A : NSObject {
    ...
}
a.m
#import Rectangle
@implementation A
...
```

上面的种种介绍，其核心的目的就是为“降低类与类之间的耦合度”。也就是说，降低类与类之间的复合关系黏性度。

在自己设计类的时候，除了 “`#import`” 和 “`@class`” 之外，有没有一种更好的方式？

有的，一种是通过使用模块方式与多类建立复合关系，详细情况请参阅建议 6；另一种是通过使用“协议”的方式来实现。

在 Objective-C 中，实际上，协议就是一个穿了“马甲”的接口。通过使用“协议”来降低类与类之间的耦合度。例如，把协议单独写在一个文件中，注意，千万不要把协议写入到一个大的头文件中，这样做，凡是只要引入此协议，就必定会引入头文件中的全部内容，如此一来，类与类之间的耦合度就会大大增加，不利于代码的管理及程序的稳定性和安全性，为以后的工作带来很大的麻烦。

故此，在自己设计类的时候，首先要明白，通过使用“#import”和“@class”，每次引入其他的头文件是否有必要。如果要引用的类和该类所在的文件同名，最好采用“@class”方式来引入。如果引用的类所处的文件有多个类或者多个其他的定义，最好采用“模块方式”来针对性引入自己所需要的类，详细情况请参与建议 6。不管采取哪种方式，降低类与类的耦合度，降低不同文件代码之间过度的黏合性是首要的目的。代码的依赖关系过于复杂则会失去代码的重用性，给维护代码带来很大的麻烦，同时，使编译的应用的稳定性和高效性也大打折扣。



### 要点

- (1) 在头文件 (.h) 中，关键字“@class”，只是为了在头文件中引用这个类，把这个类作为一个类型来用，这就要求引用的头文件 (.h) 名与类的名称一致。
- (2) 在头文件 (.h) 中使用“@class”，在源文件 (.m) 中使用“#import”，不但可以减少不必要的编译时间，降低类之间的耦合度，而且还可以避免循环引用。
- (3) 在设计类时，尽量多采用协议，避免 #import 过多，引进不必要的部分。
- (4) 如果头文件 (.h) 中有多个类的定义，尽量采用模块方式，只针对性引进所需要的类。

## 建议 3：尽量使用 const、enum 来替换预处理 #define

#define 定义了一个宏，在编译开始之前就会被替换。const 只是对变量进行修饰，当试图去修改该变量时，编译器会报错。在一些场合里你只能用 #define，而不能用 const。理论上来说，const 不仅在运行时需要占用空间，而且还需要一个内存的引用；但从时间上来说，这是无关紧要的，编译器可能会对其进行优化。

const 在编译和调试的时候比 #define 更友好。在大多数情况下，当你决定用哪一个时，这是你应该考虑的一个非常重要的点。

想象一下，下面这样一个应该使用 `#define` 而不是 `const` 的场景：如果想在大量的 `.c` 文件中使用一个常量，只需要使用 `#define` 放在头文件中；而使用 `const`，则需要在 `.c` 文件和头文件中都进行定义，如下所示。

```
// in a C file
const int MY_INT_CONST = 12345;
// in a header
extern const int MY_INT_CONST;
```

`MY_INT_CONST`，在任何 C 文件中都不能当作一个静态变量或全局作用域使用，除非它已被定义。然而，对于一个整型常量，你可以使用枚举（`enum`）。事实上，这就是 Apple 一直在做的事情。它（`enum`）兼有 `#define` 和 `const` 的所有优点，但是只能用在整型常量上。

```
// In a header
enum
{
    MY_INT_CONST = 12345,
};
```

哪一个更高效或更安全呢？`#define` 在理论上来说更高效，但就像之前说的那样，在现代的编译器上，它们可能没什么不同。`#define` 会更安全，因为当试图赋值给它时，总会出现一个编译器错误。

因此，相对字符串字面量或数字，更推荐适用常量。应使用 `static` 方式声明常量，而非使用 `#define` 的方式来定义宏。

恰当用法如下所示：

```
static NSString * const NYTAboutViewControllerCompanyName = @"The New York Times Company";

static const CGFloat NYTImageThumbnailHeight = 50.0;
```

不当用法如下所示：

```
#define CompanyName @"The New York Times Company"

#define thumbnailHeight 2
```

对于整型类型，代替 `#define` 比较好的方法是使用 `enum`，在使用 `enum` 时，推荐使用最新的 fixed underlying type 规范的 `NS_ENUM` 和 `NS_OPTIONS` 宏，因为它们是基于 C 语言的枚举，保留了 C 语言的简洁和简单的特色。这些宏能明确指定枚举类型、大小和选项，改善在 Xcode 中的代码质量。此外，在旧的编译器中，这种语法声明能正确编译通过，在新的编译器中也可明解基础类型的类型信息。

下面，使用 `NS_ENUM` 宏定义枚举。该组值是互斥的，代码如下：

```
typedef NS_ENUM(NSInteger, UITableViewCellStyle) {
    UITableViewCellStyleDefault,
```

```

UITableViewCellStyleValue1,
UITableViewCellStyleValue2,
UITableViewCellStyleSubtitle
};

```

在本例中，在命名 `UITableViewCellStyle` 的 `NSInteger` 类型时，通过使用 `NS_ENUM` 宏使界定双方的名称和枚举类型更容易了。

在下面的代码中，通过使用 `NS_OPTIONS` 宏定义了一组可以组合在一起的位掩码值，实现方式如下：

```

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
};

```

像枚举一样，`NS_OPTIONS` 宏定义了一个名称和一个类型。然而，对于选项的类型通常应该是 `NSUInteger`。

在实际编码中，如何使用枚举宏来更换 `enum`，比如这样一个 `enum` 定义：

```

enum {
    UITableViewCellStyleDefault,
    UITableViewCellStyleValue1,
    UITableViewCellStyleValue2,
    UITableViewCellStyleSubtitle
};
typedef NSInteger UITableViewCellStyle;

```

用 `NS_ENUM` 宏来实现上面的定义，其语法如下：

```

typedef NS_ENUM(NSInteger, UITableViewCellStyle) {
    UITableViewCellStyleDefault,
    UITableViewCellStyleValue1,
    UITableViewCellStyleValue2,
    UITableViewCellStyleSubtitle
};

```

在实际开发中，经常会使用 `enum` 来定义一个位掩码，如下面一个用 `enum` 来定义位掩码的示例：

```

enum {
    UIViewAutoresizingNone           = 0,
    UIViewAutoresizingFlexibleLeftMargin = 1 << 0,
    UIViewAutoresizingFlexibleWidth   = 1 << 1,
    UIViewAutoresizingFlexibleRightMargin = 1 << 2,
    UIViewAutoresizingFlexibleTopMargin  = 1 << 3,
    UIViewAutoresizingFlexibleHeight   = 1 << 4,
};

```



```

        UIViewAutoresizingFlexibleBottomMargin = 1 << 5
    };
    typedef NSUInteger UIViewAutoresizing;

```

对于上面的通过用 `enum` 定义位掩码的示例，可使用 `NS_OPTIONS` 宏实现如下：

```

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
};

```

或者，也可以使用现代化的 Objective-C 的转换器在 Xcode 自动进行此更改。



### 要点

(1) 尽量避免使用 `#define` 预处理命令。`#define` 预处理命令不包含任何的类型信息，仅仅是在编译前做替换操作。它们在重复定义时不会发出警告，容易在整个程序中产生不一致的值。

(2) 在源文件 (.m) 中定义的 `static const` 类型常量因为无须全局引用，所以它们的名字不需要包含命名空间。

(3) 在头文件 (.h) 中定义的全局引用的常量，需要关联定义在源文件 (.m) 中的部分。因为需要被全局引用，所以它们的名字需要包含命名空间，通常是用它们的类名作为命名前缀。

(4) 尽量用 `NS_ENUM` 和 `NS_OPTIONS` 宏来实现枚举。

## 建议 4：优先使用对象字面量语法而非等效方法

很多刚从其他编程语言转到 Objective-C 的程序员，往往一看到长长的函数名就会感到崩溃，这种语法让消息的传递像一个英语句子，虽有不足但确实大大增强了可读性。比如想初始化一个浮点数，需要这么写：

```
NSNumber value = [NSNumber numberWithFloat:123.45f];
```

从这句中能够明确地知道代码的含义，但是，是否连简单的赋值语句也要这么处理呢？在 2012 年的苹果年度大会上，苹果介绍了大量 Objective-C 的新特性之一——对象字面量 (Object Literals)，能够帮助 iOS 程序员更加高效地编写代码。在 XCode 4.4 版本中，这个新特性已经可以使用了。

对象字面量（Object Literals）允许方便地定义数字、数组和字典对象。这个功能类似于 Java 5 提供的 auto boxing 功能。这虽然是一个语法改进，但是对提高写代码的效率帮助很大。苹果在本次新特性中采用了折中的处理方式，针对很多基础类型采用了简写的方式，实现语法简化。简化以后，会发现在语法层面这些简化的 Objective-C 更像 Python 和 Ruby 等动态语言的语法了。

下面先来看看以前定义数字、数组和字典对象的方法：

```
NSNumber * number = [NSNumber numberWithInt:1];
NSArray * array = [NSArray arrayWithObjects:@"one", @"two", nil];
NSDictionary * dict = [NSDictionary dictionaryWithObjectsAndKeys:@"value1",
@"key1", @"value2", @"key2", nil];
```

是不是很烦琐？现在以上代码可以简化成以下形式，不用再在参数的最后加 nil 了，字典的 key 和 value 也不再是倒着先写 value，再写 key 了：

```
NSNumber * number = @1;
NSArray * array = @[@"one", @"two"];
NSDictionary * dict = @{@"key1":@"value1", @"key2":@"value2"};
```

下面逐一介绍。

## 1. 数字 (NSNumber)

简化前的写法：

```
NSNumber *value;
value = [NSNumber numberWithInt:12345];
value = [NSNumber numberWithFloat:123.45f];
value = [NSNumber numberWithDouble:123.45];
value = [NSNumber numberWithBool:YES];
```

简化后的写法：

```
NSNumber *value;
value = @12345;
value = @123.45f;
value = @123.45;
value = @YES;
```

装箱表达式也可以采用类似的写法：

```
NSNumber *piOverSixteen = [NSNumber numberWithDouble: ( M_PI / 16 )];
NSString *path = [NSString stringWithUTF8String: getenv("PATH")];
```

可以分别简写为：

```
NSNumber *piOverSixteen = @( M_PI / 16 );
NSString *path = @( getenv("PATH") );
```

对于字符串表达式来说，需要注意的是，表达式的值一定不能是 NULL，否则会抛出异常。

## 2. 数组 (NSArray)

对于 NSArray 的初始化来说，有非常多的写法，这里就不再一一罗列，直接看新的写法：

```
NSArray *array;
array = @[];           // 空数组
array = @[ a ];        // 一个对象的数组
array = @[ a, b, c ];   // 多个对象的数组
```

非常简单，再也不用记住初始化多个对象的数组时，后面还要跟一个 nil。现在看一下当声明多个对象的数组时，编译器是如何处理的。

```
array = @[ a, b, c ];
```

编译器生成的代码：

```
id objects[] = { a, b, c };
NSUInteger count = sizeof(objects)/ sizeof(id);
array = [NSArray arrayWithObjects:objects count:count];
```

好吧，编译器已经把这些简单重复的工作都做了，现在可以安心解决真正的问题了。不过有一点要注意，如果 a、b、c 对象有 nil 的话，运行时系统会抛出异常，这点和原来的处理方式不同，编码时要多加小心。



**注意** 数字 (NSArray) 和字典 (NSDictionary) 等类，由于能像“容器”一样容纳东西，所以，通常把这些具有容器特性的类称为容器类。

## 3. 字典 (NSDictionary)

同样，对于字典这个数据结构来说，有很多种初始化的方式，来看新的写法：

```
NSDictionary *dict;
dict = @{};           // 空字典
dict = @{ k1 : o1 };   // 包含一个键值对的字典
dict = @{ k1 : o1, k2 : o2, k3 : o3 }; // 包含多个键值对的字典
```

## 4. 下标法与容器类

容器的语法简化让人不难想到，可以通过下标的方式存取数组和字典的数据。比如对于数组：

```
NSArray *array = @[ a, b, c ];
```

可以这样写：

```
// 通过下标方式获取数组对象，替换原有写法：array objectAtIndex:i];
id obj = array[i];
// 也可以直接为数组对象赋值。替换原有写法
//[array replaceObjectAtIndex:i withObject:newObj];
array[i] = newObj;
```

对于字典：

```
NSMutableDictionary *dict = @{ k1 : o1, k2 : o2, k3 : o3 };
```

可以这样写：

```
// 获取 o2 对象，替换原有写法：[dic objectForKey:k2];
id obj = dict[k2];
// 重新为键为 k2 的对象赋值，替换原有写法：[dic setObject:newObj forKey:k2]
dic[k2] = newObj;
```

同时，自己定义的容器类只要实现了规定的下标方法，就可以采用下标的方式访问数据。要实现的方法如下。

数组类型的下标方法：

```
- (elementType)objectAtIndexedSubscript:(indexType)idx;
- (void)setObject:(elementType)object atIndex:(indexType)idx;
```

字典类型的下标方法：

```
- (elementType)objectForKeyedSubscript:(keyType)key;
- (void)setObject:(elementType)object forKey:(keyType)key;
```

其中需要注意的是，`indexType` 必须是整数，`elementType` 和 `keyType` 必须是对象指针。

## 5. 容器类数据结构简化的限制

采用上述写法构建的容器都是不可变的，如果需要生成可变容器，可以传递 `-mutableCopy` 消息。例如：

```
NSMutableArray *mutablePlanets = [@[
    @"Mercury", @"Venus", @"Earth",
    @"Mars", @"Jupiter", @"Saturn",
    @"Uranus", @"Neptune"
] mutableCopy];
```

不能对常量数组直接赋值，解决办法是在类方法 `(void)initialize` 中进行赋值处理，如下：

```
@implementation MyClass
static NSArray *thePlanets;
+ (void)initialize {
    if (self == [MyClass class]) {
        thePlanets = @[
            @"Mercury", @"Venus", @"Earth",
            @"Mars", @"Jupiter", @"Saturn",
            @"Uranus", @"Neptune"
        ];
    }
}
```



### 要点

- (1) 尽量使用对象字面量语法来创建字符串、数字、数组和字典等，使用它比使用以前的常规对象创建方法语法更为精简，同时可以避免一些常见的陷阱。
- (2) 对象字面量语法特性是完全向下兼容，使用新特性编写出来的代码，经过编译后形成的二进制程序可以运行在之前发布的任何 OS 中。
- (3) 在数字和字典中，要使用关键字和索引做下标来获取数据。
- (4) 使用对象字面量语法时，容器类的不可是 nil，否则运行时将会抛出异常。

## 建议 5：处理隐藏的返回类型，优先选择实例类型而非 id

实例类型（instancetype）是 Objective-C 语言中新添加的一个返回类型，实例类型作为方法返回的实例的类型，是苹果在 2013 年的年度大会上宣布的。这个新添加的实例类型不仅可用来作为 Objective-C 方法的返回类型，且能用这个实例类型来作为向编译器的提示，提示方法返回的类型将是方法所属的类的实例。

类的实例，作为方法返回类型，宜采用关键字 `instancetype` 作为方法的返回类型，如 `alloc`、`init` 和类工厂方法等。使用 `instancetype` 作为类（或者类的子类）的实例返回类型，可以大大改善 Objective-C 代码的类型安全。例如，考虑下面的代码：

```
@interface MyObject : NSObject
+ (instancetype)factoryMethodA;
+ (id)factoryMethodB;
@end

@implementation MyObject
+ (instancetype)factoryMethodA {
    return [[[self class] alloc] init];
}
+ (id)factoryMethodB {
    return [[[self class] alloc] init];
}
@end

void doSomething() {
    NSUInteger x, y;
    // Return type of +factoryMethodA is taken to be "MyObject *"
    x = [[MyObject factoryMethodA] count];
    // Return type of +factoryMethodB is "id"
    y = [[MyObject factoryMethodB] count]; }
```

通过上面的代码可以看到，`instancetype` 作为 `+ factoryMethodA` 的返回类型，也就是说，

该消息的类型表达式是 `MyObject *`。但是 `MyObject` 由于先天缺乏一个 `-count` 方法，编译器将会对此给出一个关于 `x` 行的警告：

```
main.m: 'MyObject' may not respond to 'count'
```

对于这样的情况，如果把 `instancetype` 换成 `id` 作为实例的方法返回类型，也就是如上面的代码中的实例，`id` 作为类方法 + `factoryMethodB` 的返回类型。在编译器编译的过程中不会发出关于 `y` 行的警告。

为什么编译器没有给出警告？因为 `id` 类型的对象可以作为任何类，并且调用的方法 `-count` 在一些类中存在，故此向编译器发出方法 + `factoryMethodB` 返回值实现了 `-count` 的信息，从而编译器没有给出警告。对于该种编写代码的方法，在无形中埋下了隐患。

为了确保 `instancetype` 工厂方法有正确的子类的行为，一定要使用 `[self class]` 分配类，而不是直接引用类名。遵循这个惯例，记住，务必要使编译器能正确地推断出子类类型。例如，依据前面的示例，考虑尝试做一个前面 `MyObject` 子类示例：

```
@interface MyObjectSubclass : MyObject
@end

void doSomethingElse() {
    NSString *aString = [MyObjectSubclass factoryMethodA];
}
```

对于上述代码，编译器将会给出警告，如下面的警告：

```
main.m: Incompatible pointer types initializing 'NSString **' with an expression of
type 'MyObjectSubclass *'
```

在该示例中，+ `factoryMethodA` 消息发送之后，将返回一个类型 `MyObjectSubclass` 的对象实例。编译器就能恰当地确定 + `factoryMethodA` 的返回类型应该是子类 `MyObjectSubclass`，而不是工厂方法中所声明的超类。

在编写代码中，通常在处理 `init` 方法和类工厂方法时，宜用 `instancetype` 类替换 `id` 作为返回值。在新版本 Xcode 5 中，虽然，编译器会自动地把 `alloc`、`init`、`new` 方法之中的 `id` 转化为 `instancetype` 类型，但对于这几种方法之外的其他方法，编译器则不会进行转化。在 Objective-C 的公约之中，明确地建议对于所有方法尽可能用 `instancetype` 而非 `id`。也就是说，作为返回值，`id` 由于其自身的缺陷，在 Objective-C 中会逐渐退出，由 `instancetype` 来替代。



**注意** 仅有在作为返回值时，宜用 `instancetype` 来替换 `id`，而不是代替代码中所有 `id`。与 `id` 不同，`instancetype` 关键字仅能作为方法声明的返回类型。也就是说，在某一个特定区域，`instancetype` 可以替代 `id`，并非所有区域都可以替代 `id`。

---

例如：

```
@interface MyObject
- (id)myFactoryMethod;
@end
```

应该成为：

```
@interface MyObject
- (instancetype)myFactoryMethod;
@end
```



### 要点

- (1) `instancetype` 仅仅用来作为 Objective-C 方法的返回类型。
- (2) 使用 `instancetype` 可避免隐式转换 `id` 而造成的欺骗性编译无误通过的现象，防止程序正式运行时出现崩溃现象，可以大大改善 Objective-C 代码的类型安全。
- (3) 在某一个特定区域，`instancetype` 可以替代 `id`，并非所有区域都可以替代 `id`。

## 建议 6：尽量使用模块方式与多类建立复合关系

在 2013 年的苹果年度大会上，苹果在 Objective-C 的性能改进上大的变化之一就是加入了模块 (Modules)。

### 1. 文件编译问题的存在性——编译时间过长

在了解模块 (Modules) 之前，需要先了解一下 Objective-C 的 `#import` 机制。通过使用 `#import`，用来引用其他的头文件。

熟悉 C 或者 C++ 的人可能会知道，在 C 和 C++ 里是没有 `#import` 的，只有 `#include` (虽然 GCC 现在为 C 和 C++ 做了特殊处理，使 `#import` 可以被编译)，用来包含头文件。`#include` 做的事情其实就是简单的复制、粘贴，将目标 .h 文件中的内容一字不落地复制到当前文件中，并替换掉这句 `include`；而 `#import` 实质上做的事情和 `#include` 是一样的，只不过 Objective-C 为了避免重复引用可能带来的编译错误 (这种情况在引用关系复杂的时候很可能发生，比如 B 和 C 都引用了 A，D 又同时引用了 B 和 C，这样 A 中定义的东西就在 D 中被定义了两次，造成重复) 而加入了 `#import`，从而保证每个头文件只会被引用一次。仔细探究一下，`#import` 的实现是通过对 `#ifndef` 一个标志进行判断，然后再引入 `#define` 这个标志，来避免重复引用的。

实质上，`#import` 也是复制、粘贴，这样就带来一个问题：当引用关系很复杂或一个头文件被非常多的实现文件引用时，编译时引用所占的代码量就会大幅上升 (因为被引用的头



文件在各个地方都被复制了一遍)。

在编写 Objective-C 代码中，估计很多人已经写了一千遍或更多 `#import` 语句：

```
//
//AppDelegate
//
#import <UIKit/UIKit.h>
```

按照上面所说，这意味着，对于 UIKit 框架，通过计算所有行的全部 UIKit 中的头，就会发现它相当于超过 11 000 行代码！在一个标准的 iOS 应用中，就会在大部分文件中导入 UIKit，这意味着每一个文件最终变成 11000 行。这是不够理想的，更多的代码意味着更长的编译时间。

## 2. 预编译头文件 (Pre-compiled Headers) 处理方式——不实用

理论上讲，解决这个问题可采取 C 语言的方式，引入预编译头文件 (Pre-compiled Headers, PCH)，即把公用的头文件放入预编译头文件中预先进行编译。通过在编译的预处理阶段，预先计算和缓存需要的代码；然后在真正编译工程时再将预先编译好的产物加入到所有待编译的源文件中去，来加快编译速度。比如 iOS 开发中 Supporting Files 组内的 .pch 文件，就是一个预编译头文件。默认情况下，它引用了 UIKit 和 Foundation 两个头文件，这是在 iOS 开发中基本上每个实现文件都会用到的东西。

下面是 Xcode 生成的 stock PCH 文件，像这样：

```
#import <Availability.h>
#ifdef __IPHONE_5_0
#warning "This project uses features only available in iOS SDK 5.0 and later."
#endif

#ifdef __OBJC__
    #import <UIKit/UIKit.h>
    #import <Foundation/Foundation.h>
#endif
```

如果开发的应用是使用 iOS 5 之前的 SDK，那么 `#warning` 将通知它们。UIKit 和 Foundation 头文件是 stockPCH 的一部分。因为在应用程序里的每一个文件将使用 Foundation，并且大部分会使用 UIKit。因此，这些都被很好地添加进了预编译头文件，以便于在自己的应用程序中预先计算和缓存这些文件的编译文件。

但维护项目的预编译头文件是很棘手的。利用预编译头文件虽然可以加快编译的时间，但是这样面临的问题是，在工程中随处可用本来应该不能访问的东西，而编译器也无法准确给出错误或者警告，无形中增加了出错的可能性。

## 3. 利用模块 (Modules) 来解决历史问题——事半功倍

模块 (Modules)，第一次在 Objective-C 中公共露面是在 2012 LLVM 开发者大会上

Apple's Doug Gregor 的一次谈话中。

模块 (Modules), 封装框架比以往任何时候都更加清洁。不再需要预处理逐行地用文件的所有内容替换 `#import` 指令。相反, 一个模块包含了一个框架到自包含的块中, 就像预编译文件预编译的方式一样提升了编译速度。并且不需要在预编译头文件中声明自己要用到哪些框架, 使用模块简单地获得了速度上的提升。图 1-3 所示为预编译文件和模块功能在编译上的比较。

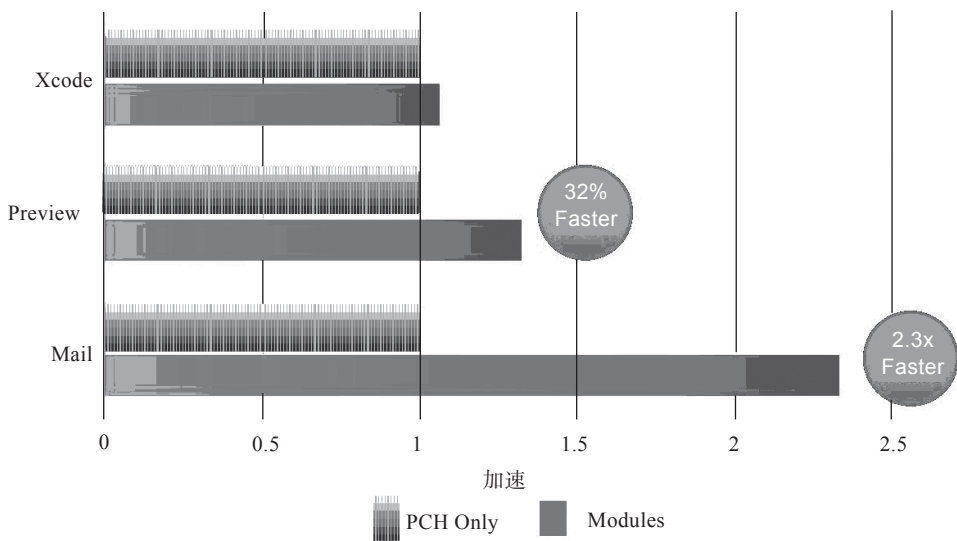


图 1-3 预编译文件和模块功能编译改进的比较

模块还有其他方面的优点, 在下列的操作编写代码的步骤过程中, 都可以感受到模块的特性。

- (1) 在使用框架的文件中添加 `#import`。
- (2) 用框架写代码。
- (3) 编译。
- (4) 查看链接错误。
- (5) 忘记链接的框架。
- (6) 添加忘记的框架到项目中。
- (7) 重新编译。

忘记链接框架式是编写程序代码中经常会犯的一个错误, 利用模块就能解决这个问题。一个模块不仅告诉编译器哪些头文件组成了模块, 而且还告诉编译器什么需要链接。这样就不用去手动链接框架了。虽然这是一件小事, 但是能让开发变得更加简单, 这就是

一件好事。

#### 4. 开启使用模块

模块的使用相当简单。对于存在的工程，第一件事情就是使这个功能生效。可以在项目的 Build Settings 中通过搜索 Modules 找到这个选项，将 Enable Modules 选项设为 Yes，如图 1-4 所示。

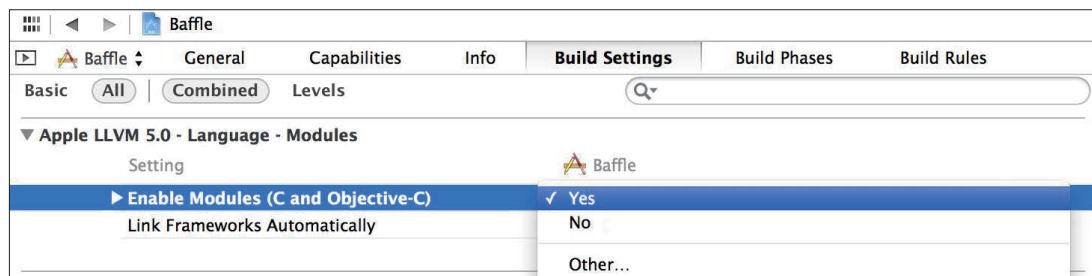


图 1-4 开启模块功能

在默认情况下，模块功能所有的新工程都是开启的，但是应该在自己所有存在的工程中都开启这个功能。在图 1-4 中，Link Frameworks Automatically 选项，可以用来开启或者关闭自动链接框架的功能。

一旦模块（Modules）功能开启，就可以在自己代码中使用它了。要这样做，对以前用到的语法有一点小小的改动，那用 @import 代替 #import：

```
@import UIKit;  
@import MapKit;  
@import iAd;
```

另外，只导入一个框架中自己需要的部分也是可以的。例如，只想要导入 UIView，就可以这样写：

```
@import UIKit.UIView;
```

使用模块功能就是这么简单。技术上，自己不需要把所有的 #import 都换成 @import，因为编译器会隐式地转换它们，但是还是建议尽可能地用新的语法来编写代码。



**注意** Xcode 5.0 的模块功能还不支持自己的或第三方的框架。目前 Xcode 6 的测试版都出来了，在很多 Xcode 的新版本中，都支持模块功能。



### 要点

- (1) `#include` 和 `#import`，其根本就是简单的复制、粘贴，将目标 `.h` 文件中的内容一字不落地复制到当前文件中，后者可以避免多次的重复引用。
- (2) 以预编译头文件的方式，虽可缩短编译时间，但其维护棘手，不利于广泛应用。
- (3) 模块功能，其应用不仅仅表现于编译的速度加快，同时在链接框架等方面也非常好用。
- (4) 启动模块功能后，编译器会隐式地把所有的 `#import` 都转换成 `@import`。

## 建议 7：明解 Objective-C++ 中的有所为而有所不为

苹果的 Objective-C 编译器允许用户在同一个源文件（`.m`）里自由地混合使用 C++ 和 Objective-C，混编后的语言叫作 Objective-C++。有了它，你就可以在 Objective-C 应用程序中使用已有的 C++ 类库。

在 Objective-C++ 中，可以用 C++ 代码调用方法，也可以从 Objective-C 调用方法。在这两种语言中对象都是指针，可以在任何地方使用。例如，C++ 类可以使用 Objective-C 对象的指针作为数据成员，Objective-C 类也可以有 C++ 对象指针做实例变量。



**注意** Xcode 需要源文件以“`.mm`”为扩展名，这样才能启动编译器的 Objective-C++ 扩展。

下面，将一一介绍二者的联系及区别。

### 1. 二者的定义结构一样，但是 Objective-C 的继承是封闭的

(1) 定义一个 C++ 类 Hello，代码如下：

```
#import <Foundation/Foundation.h>
class Hello {
private:
    id greeting_text; // holds an NSString
public:
    // 方法 Hello
    Hello() {
        greeting_text = @"Hello, world!";
    }
    // 方法 Hello
    Hello(const char* initial_greeting_text) {
        greeting_text = [NSString alloc]
            initWithUTF8String:initial_greeting_text];
    }
}
```

```

    }
    // 方法 say_hello
    void say_hello() {
        printf("%s\n", [greeting_text UTF8String]);
    }
};

```

(2) 定义一个 Objective-C 类 Greeting，代码如下：

```

@interface Greeting : NSObject {
@private
    Hello *hello;
}
- (id)init;
- (void)dealloc;
- (void)sayGreeting;
- (void)sayGreeting:(Hello*)greeting;
@end

@implementation Greeting
- (id)init {
    if (self = [super init]) {
        hello = new Hello();
    }
    return self;
}
- (void)dealloc {
    delete hello;
    [super dealloc];
}
- (void)sayGreeting {
    hello->say_hello();
}
- (void)sayGreeting:(Hello*)greeting {
    greeting->say_hello();
}
@end

```

(3) C++ 类 Hello 和 Objective-C 类 Greeting 混合使用，代码如下：

```

int main() {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    Greeting *greeting = [[Greeting alloc] init];
    [greeting sayGreeting]; // 将输出 Hello, world!

    Hello *hello = new Hello("Bonjour, monde!");
    [greeting sayGreeting:hello]; // 将输出 Bonjour, monde!

    delete hello;
    [greeting release];
    [pool release];
    return 0
}

```

从上面的代码可以看出，正如可以在 Objective-C 接口中声明 C 结构一样，也可以在 Objective-C 接口中声明 C++ 类。跟 C 结构一样，Objective-C 接口中定义的 C++ 类是全局范围的，不是 Objective-C 类的内嵌类（这与标准 C 提升嵌套结构定义为文件范围是一致的）。

为了允许基于语言变种条件化编写代码，Objective++ 编译器定义了 `__cplusplus` 和 `__OBJC__` 预处理器常量，分别指定 C++ 和 Objective-C。如前所述，Objective++ 不允许 C++ 类继承自 Objective-C 对象，也不允许 Objective-C 类继承自 C++ 对象。

```
class Base { /* ... */ };
@interface ObjCClass: Base ... @end // 错误！
class Derived: public ObjCClass ... // 错误！
```

## 2. 两者的对象模型不能直接兼容

与 Objective-C 不同的是，C++ 对象是静态类型的，有运行时系统多态是特殊情况。两种语言的对象模型因此不能直接兼容。更根本的原因是，Objective-C 和 C++ 对象在内存中的布局是互不相容的，也就是说，一般不可能创建一个对象实例从两种语言的角度来看都是有效的。因此，两种类型层次结构不能被混合。

可以在 Objective-C 类内部声明 C++ 类，编译器把这些类当作已声明在全局名称空间来对待，例如：

```
@interface Foo {
    class Bar { ... } // OK
}
@end
```

```
Bar *barPtr; // OK
```

Objective-C 允许 C 结构作为实例变量，不管它是否声明在 Objective-C 声明内部。

```
@interface Foo {
    struct CStruct { ... };
    struct CStruct bigIvar; // OK
} ... @end
```

Mac OS X 10.4 以后，如果设置 `fobjc-call-cxx-cttors` 编译器标志，就可以使用包含虚函数和有意用户自定义零参数构造函数、析构函数的 C++ 类实例来作为实例变量（gcc-4.2 默认设置编译器标志 `fobjc-call-cpp-cttors`）。Objective-C 成员变量 `alloc` 完成以后，`alloc` 函数会按声明顺序调用构造器。构造器使用公共无参数恰当的构造函数。Objective-C 成员变量 `dealloc` 之前，`dealloc` 方法按声明顺序反序调用析构函数。Objective-C 没有名称空间的概念，不能在 C++ 名称空间内部声明 Objective-C 类，也不能在 Objective-C 类里声明名称空间。

Objective-C 类、协议、分类不能声明在 C++ template 里，C++ template 也不能声明在 Objective-C 接口、协议、分类的范围内。

但是，Objective-C 类可以做 C++ template 的参数，C++ template 参数也可以做 Objective-C 消息表达式的接收者或参数（不能通过 selector）。

### 3. 两者有词汇歧义和冲突

Objective-C 头文件中定义了一些标识符，所有的 Objective-C 程序必须包含这些标识符：id、Class、SEL、IMP 和 BOOL。

Objective-C 方法内，编译器预声明了标识符 self 和 super，就像 C++ 中的关键字 this。跟 C++ 的 this 不同的是，self 和 super 是上下文相关的，除 Objective-C 方法外，它们还可以用于普通标识符。

协议内方法的参数列表，有 5 个上下文相关的关键字（oneway、in、out、inout、bycopy）。这些在其他内容中不是关键字。

从 Objective-C 程序员的角度来看，C++ 增加了不少新的关键字。你仍然可以使用 C++ 的关键字做 OC selector 的一部分，所以影响并不严重，但不能使用它们命名 Objective-C 类和实例变量。例如，尽管 class 是 C++ 的关键字，但是你仍然能够使用 NSObject 的 class 方法。

然而，因为它是一个关键字，所以不能用 class 做变量名称：

```
NSObject *class; // Error
```

Objective-C 里类名和分类名有单独的命名空间。@interface foo 和 @interface(foo) 能够同时存在在一个源代码中。Objective ++ 中也可以用 C++ 中的类名或结构名来命名你的分类。

协议和 template 标识符使用语法相同但目的不同：

```
id<someProtocolName> foo;
TemplateType<SomeTypeName> bar;
```

为了避免这种含糊之处，编译器不允许把 id 做 template 名称。最后，C++ 有一个语法歧义，当一个 label 后面跟了一个表达式表示一个全局名称时，就像下面：

```
label: ::global_name = 3;
```

第一个冒号后面需要空格。Objective ++ 有类似情况，也需要一个空格：

```
receiver selector: ::global_c++_name;
```

### 4. 两者功能上有限制

Objective C++ 没有为 Objective-C 类增加 C++ 的功能，也没有为 C++ 类增加 Objective-C 的功能。例如，你不能用 Objective-C 语法调用 C++ 对象，也不能为 Objective-C 对象增加构造函数和析构函数，也不能将 this 和 self 互相替换使用。类的体系结构是独立的。C++ 类不能继承 Objective-C 类，Objective-C 类也不能继承 C++ 类。另外，多语言异常处理是不支持的。也就是说，一个 Objective-C 抛出的异常不能被 C++ 代码捕获；反过来，C++ 代



码抛出的异常也不能被 Objective-C 代码捕获。



### 要点

- (1) C++ 和 Objective-C 在定义结构上一样，但是后者的继承是封闭的。
- (2) Objective-C 接口中定义的 C++ 类是全局范围的，而不是 Objective-C 类的内嵌类。
- (3) C++ 和 Objective-C 的对象模型不能直接兼容。与 Objective-C 不同的是，C++ 对象是静态类型的，有运行时系统多态是特殊情况。
- (4) C++ 和 Objective-C 有词汇歧义和冲突。
- (5) C++ 和 Objective-C 两者功能上有限制。Objective C++ 没有为 Objective-C 类增加 C++ 的功能，也没有为 C++ 类增加 Objective-C 的功能。

## 数据类型、集合和控制语句

《道德经》第五十二章中言曰“天下有始，以为天下母。既得其母，以知其子；既知其子，复守其母。”掌握了基本的开发语言，就可以“知其子”——程序；反过来，通过开发程序，又可以“复守其母”——开发语言，即可以通过开发程序进一步巩固加深对开发语言的理解。

在 Objective-C 中，构建语言的基本单元——值和集合，更是重中之重。本章将围绕这些构建开发语言的“基石”并站在不同角度加以阐述。

### 建议 8：C 语言与 Objective-C 语言的关系是充分而非必要条件

在众多武侠小说描述的武林界，武术泰斗张三丰，虽然师从少林觉远和尚，但是真人张三丰人集百家之长，融道家养身，刚柔相济，动静结合，以柔克刚，以静制动，而独创武当派。Objective-C 和 C 语言的关系正如“武当和少林”的关系一样，两者虽有关系，但并非是继承关系，故称 Objective-C 是 C 语言的超集。

Objective-C 作为 C 程序设计语言的超集，支持与 C 相同的基本语法。会看到所有熟悉的元素，如基本类型（int、float 等）、结构、函数、指针，以及流程控制结构，如 if...else 语句和 for 语句。还可以访问标准 C 库例程，如在 stdlib.h 和 stdio.h 中声明的那些例程。

(1) C 语言中每个标准变量类型在 Objective-C 中都可用，例如：

```
int someInteger = 42;
float someFloatingPointNumber = 3.1415;
double someDoublePrecisionFloatingPointNumber = 6.02214199e23;
```

(2) C 语言中的标准运算符在 Objective-C 中可用, 例如:

```
int someInteger = 42;
someInteger++;           // someInteger == 43

int anotherInteger = 64;
anotherInteger--;        // anotherInteger == 63

anotherInteger *= 2;      // anotherInteger == 126
```

(3) 可用标量来表示 Objective-C 的属性, 例如:

```
@interface XYZCalculator : NSObject
@property double currentValue;
@end
```

(4) 通过点语法访问值时, 可以在属性中使用 C 操作符, 例如:

```
@implementation XYZCalculator
- (void)increment {
    self.currentValue++;
}
- (void)decrement {
    self.currentValue--;
}
- (void)multiplyBy:(double)factor {
    self.currentValue *= factor;
}
@end
```

点语法纯粹是对存取方法调用的一个语法包装, 所以在这个例子中的每个操作相当于先使用 `get` 访问器方法来获取值, 然后执行操作, 最后使用 `set` 访问器方法来把该值设置为结果。

(5) 在 Objective-C 中, 定义了新的基本数据类型。BOOL 标量类型在 Objective-C 中仍然被定义布尔值, 其值表示为 YES 或 NO。正如所料, YES 逻辑上等同于 true 和 1, 而 NO 等同于 false 和 0。在 Cocoa 和 Cocoa Touch 对象中方法中的许多参数, 也可以使用特殊的标量数值类型, 如 NSInteger 或 CGFloat。例如, 如前一章所述, NSTableViewDataSource UITableViewDataSource 协议都有方法要求要显示的行数:

```
@protocol NSTableViewDataSource <NSObject>
- (NSInteger)numberOfRowsInTableView:(NSTableView *)tableView;
...
@end
```

这些类型, 如 NSInteger 和 NSUInteger, 类型不同的定义, 这取决于目标架构。在 32 位环境 (如 iOS) 生成时, 它们是 32 位有符号和无符号整数; 在 64 位环境 (如现代 OS X 运行时) 时, 它们是 64 位有符号和无符号整数。

如果要作可跨 API 边界 (包括内部和导出的 API) 的应用, 它的最佳实践是使用这些平

台特定的类型，如应用代码和框架之间的方法或者函数调用的参数或返回值。

对于局部变量，如在一个循环计数器中，如果知道该值是标准限值内，使用基本的 C 类型是很好的。

(6) C 语言中的数据结构在 Objective-C 中可保持其基本值。在一些 Cocoa 和 Cocoa Touch API 中，仍然使用 C 语言结构来保存它们的值。作为一个例子，它可能要向一个字符串对象来查询子串的范围，例如：

```
NSString *mainString = @"This is a long string";
NSRange substringRange = [mainString rangeOfString:@"long"];
```

同样，如果需要编写自定义的绘图代码，需要根据 CGFloat 相关的数据类型结构，与 Quartz 进行交互，如 OS X 上的 NSPoint 和 CGPoint，iOS 上的 NSSize 和 CGSize。再次，CGFloat 会在不同的目标结构以不同方式来定义。

(7) Objective-C 值对象比 C 类型变量具有封装常用操作的优势。在 Objective-C 代码中，复制值对象，表示属性的对象，是一种很普遍的做法。C- 类型的变量通常可以取代值对象，但值对象具有封装常用操作的优势。例如，NSString 对象被用来代替字符指针，因为它们封装了编码和存储。

当值对象作为方法的参数被传递或者从一个方法被返回时，通常会使用对象的副本，而不是对象本身。例如，下面的方法将一个字符串赋值给对象的 name 实例变量。

```
- (void)setName:(NSString *)aName {
    [name autorelease];
    name = [aName copy];
}
```

存储 aName 的一个副本，其效果是产生一个独立于原始对象，但与原始对象具有相同内容的对象。副本的后续变化不会影响原始对象，并且原始对象的变化也不会影响副本。类似的，一种常见的做法是返回实例变量的副本，而不是实例变量本身。例如，下面的方法返回 name 实例变量的一个副本：

```
- (NSString *)name {
    return [[name copy] autorelease];
}
```



## 要点

- (1) C 语言的基本语法在 Objective-C 语言中是可用的。
- (2) 与 C 语言相比，Objective-C 语言又定义了新的基本数据类型，如 BOOL 等。
- (3) Objective-C 值对象比 C 类型变量具有封装常用操作的优势，但在数值计算中，使用 C 类型标量更为简洁。

## 建议 9：高度警惕空指针和野指针的袭击

在 Objective-C 中，利用指针写代码，特别对于指针掌握不熟练的人，经常会遭遇到空指针和野指针的困扰，造成应用出现一些莫名其妙的崩溃。因此，有必要在写 Objective-C 代码时，高度警惕空指针和野指针的袭击。

兵法上讲究“知己知彼，百战不殆”，那么就从什么是空指针和野指针来入手，认识这两个经常搞袭击的常客。

### 1. 认识空指针和野指针

没有存储任何内存地址的指针就称为空指针（NULL 指针）。空指针就是被赋值为 0 的指针，在没有被具体初始化之前，其值为 0。也就是说，一个指针变量分配一个 NULL 值的情况下，没有确切的地址被分配。

下面两个都是空指针：

```
Student *s1 = NULL;
Student *s2 = nil;
```

NULL 指针是一个常数与几个标准库中定义的一个零值。考虑下面的程序：

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

int main ()
{
    int *ptr = NULL;

    NSLog(@"The value of ptr is : %x\n", ptr );

    return 0;
}
```

上面的代码编译和执行时，它会产生以下结果：

```
2013-09-13 03:21:19.447 demo[28027] The value of ptr is : 0
```

大部分的作业系统程序不允许被保留，因为该内存由操作系统来访问内存地址 0 处。然而，存储器地址 0 具有特殊的意义，它的信号指针不指向一个可访问的存储器位置。但是，按照惯例，如果一个指针包含空值（零），它被假定为指向什么。

要检查空指针，可以使用一个 if 语句，具体如下：

```
if(ptr)      /* succeeds if p is not null */
if(!ptr)     /* succeeds if p is null */
```

野指针不是 NULL 指针，而是指向“垃圾”内存（不可用内存）的指针。野指针是非常危险的。

## 2. 空指针和野指针的区别及防御策略

接下来用一个简单的例子对比一下野指针和空指针的区别，同时，介绍如何防止空指针和野指针的产生。

(1) 首先，打开 Xcode 的内存管理调试开关，它能帮助用户检测垃圾内存，如图 2-1 和图 2-2 所示。

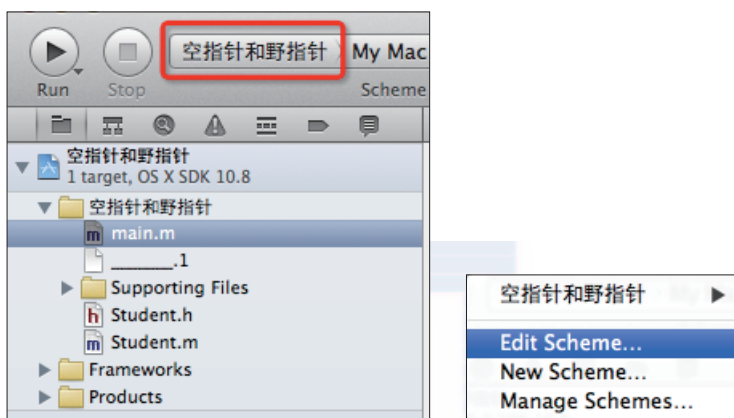


图 2-1 设置内存管理调试开关

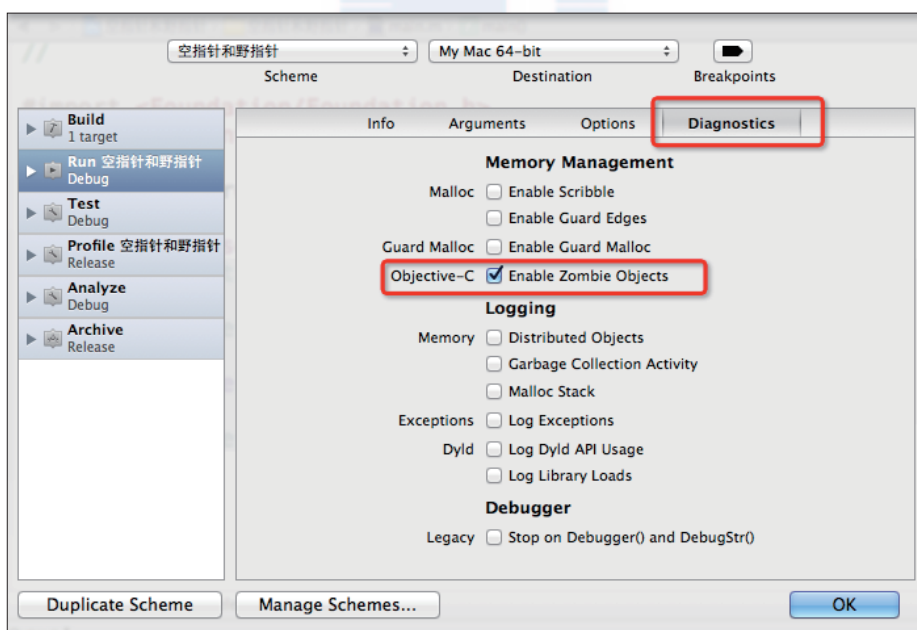
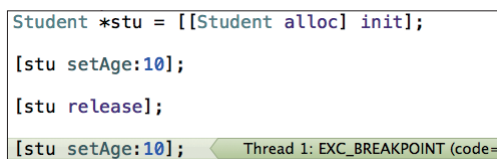


图 2-2 检测内存垃圾设置

(2) 自定义 Student 类，在 main 函数中添加下列代码：

```
Student *stu = [[Student alloc] init];
[stu setAge:10];
[stu release];
[stu setAge:10];
```

运行程序，你会发现第 7 行报错了，是一个野指针错误，如图 2-3 所示。



```
Student *stu = [[Student alloc] init];
[stu setAge:10];
[stu release];
[stu setAge:10];
```

图 2-3 编译错误提示

(3) 接下来分析一下报错原因。

① 执行完第 1 行代码后，内存中有个指针变量 stu，指向了 Student 对象。

```
Student *stu = [[Student alloc] init];
```

假设 Student 对象的地址为 0xff43，指针变量 stu 的地址为 0xee45，stu 中存储的是 Student 对象的地址 0xff43，即指针变量 stu 指向了这个 Student 对象，如图 2-4 所示。

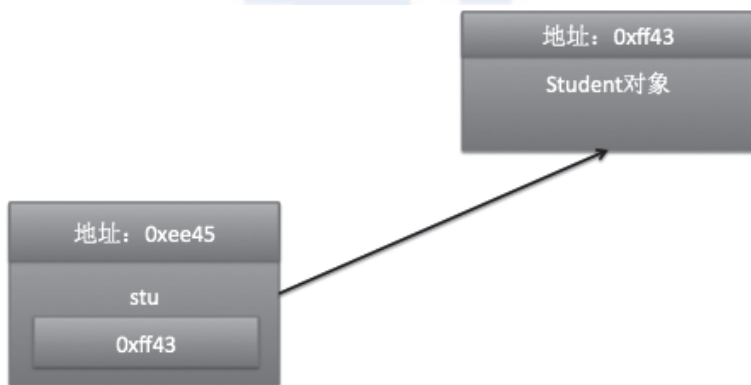


图 2-4 指针变量 stu 与 Student 的关联关系

② 接下来是第 3 行代码：

```
[stu setAge:10];
```

这行代码的意思是：给 stu 所指向的 Student 对象发送一条 setAge: 消息，即调用这个 Student 对象的 setAge: 方法。目前来说，这个 Student 对象仍存在于内存中，所以这句代码没有任何问题。

③ 接下来是第 5 行代码：

```
[stu release];
```



这行代码的意思是：给 stu 指向的 Student 对象发送一条 release 消息。在这里，Student 对象接收到 release 消息后，会马上被销毁，所占用的内存会被回收。

Student 对象被销毁了，地址为 0xff43 的内存就变成了“垃圾内存”，然而，指针变量 stu 仍然指向这一块内存，如图 2-5 所示，这时 stu 就成为野指针。

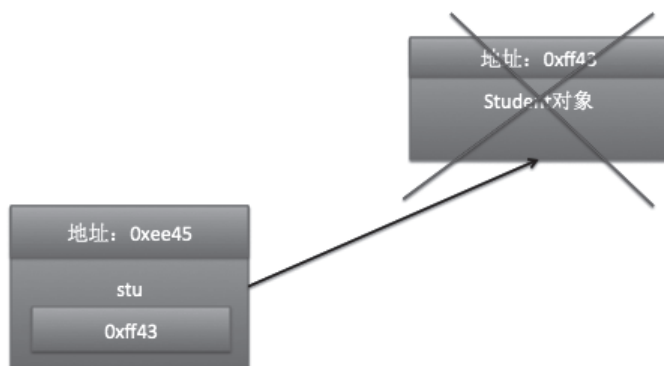


图 2-5 野指针产生的原因

④最后执行了第 7 行代码：

```
[stu setAge:10];
```

这句代码的意思仍然是：给 stu 所指向的 Student 对象发送一条 setAge: 消息。但是在执行完第 5 行代码后，Student 对象已经被销毁了，它所占用的内存已经是垃圾内存，如果你还去访问这一块内存，那就会报野指针错误。这块内存已经不可用了，也不属于你了，你还去访问它，肯定是不合法的。所以，这行代码报错了！

（4）如果改动一下代码，就不会报错了，如下所示：

```
Student *stu = [[Student alloc] init];
[stu setAge:10];
[stu release];

stu = nil;
[stu setAge:10];
```

注意第 7 行代码，stu 变成了空指针，stu 就不再指向任何内存了，如图 2-6 所示。因为 stu 是个空指针，没有指向任何对象，因此第 9 行的 setAge: 消息是发不出去的，不会造成任何影响。当然，肯定也不会报错。

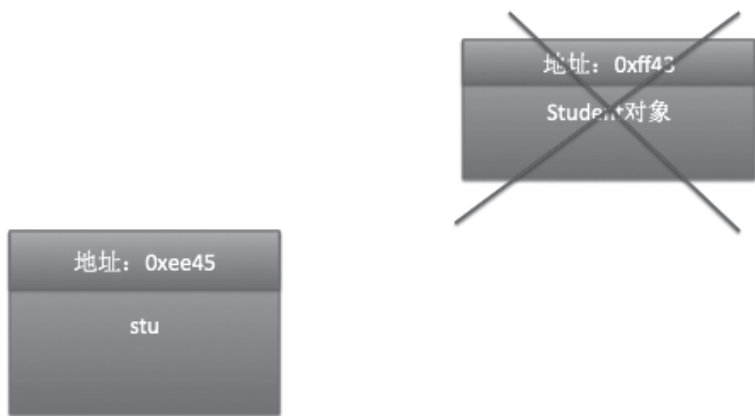


图 2-6 空指针产生的原因



### 要点

- (1) 空指针 (NULL 指针), 是指没有存储任何内存地址的指针。野指针, 是指向“垃圾内存”(不可用内存) 的指针。
- (2) 利用野指针发消息是很危险的, 会报错。也就是说, 如果一个对象已经被回收了, 就不要再去做操作它, 不要再尝试给它发消息。
- (3) 利用空指针发消息是没有任何问题的, 也就是说代码是没有错误的。

## 建议 10: 在 64 位环境下尽可能利用标记指针

在 Mac OS X 10.6 (雪豹) 中开始支持 64 位, 如今最新版本 iPhone 5s 也开始采用 Arm64 架构。在 64 位化的过程中, 其中一个比较关键的改进就是, Mac OS 10.7 (美洲虎) 和 iOS 7 的 64 位环境先后引入了标记 (Tagged) 指针。

下面就简单地来介绍一下标记 (Tagged) 指针。在介绍标记 (Tagged) 指针之前有必要介绍一下指针地址对齐概念和 64 位环境的一些变化。

### 1. 指针地址对齐

在 32 位环境下, 要读取一个 32 位整数, 如果这个 32 位整数在内存地址为 0x00000002-0x00000006 (仅作举例, 这个地址一般是被系统保留的) 的内存上, 读取这个整数会消耗 2 个 CPU 周期, 而如果这个数在 0x00000004 ~ 0x00000008 的内存上只需要一个 CPU 周期。为了加快内存的 CPU 访问, 程序都使用了指针地址对齐概念。指针地址对齐就是指在分配堆中的内存时往往采用偶数倍或以 2 为指数倍的内存地址作为地址边界。

几乎所有系统架构，包括 Mac OS 和 iOS，都使用了地址对齐概念。

```
void *a = malloc(1);  
void *b = malloc(3);  
NSLog(@"a: %p", a);  
NSLog(@"b: %p", b);
```

运行这段代码后，得到了如下结果：

```
a: 0x8c11e20  
b: 0x8c11e30
```

可以看到，a 和 b 指针的最后 4 位都是 0，虽然 a 只占用 31 个字节，但是 a 和 b 的地址却相差 16 个字节。因为 iOS 中是以 16 个字节为内存分配边界的，或者说 iOS 的指针地址对齐是以 16 个字节为对齐边界的。进一步说，iOS 中分配的内存地址最后 4 位永远都是 0。

## 2. 64 位地址

iPhone 5s 中采用了 Arm64 的 CPU，同时也支持了 64 位的 App。64 位 App 中指针大小也扩大到 64 位，就是理论上可以支持最大  $2^{64}$  字节（达千万 T 字节）的内存地址空间。而对于大多数应用来说，这么大的地址空间完全是浪费的。也就是说，64 位环境下，内存地址的前面很多位一般都是 0。

## 3. 标记 (Tagged) 指针

由于指针地址对齐概念和 64 位超大地址的出现，指针地址仅仅作为内存的地址是比较浪费的，故此，可以在指针地址中保存或附加更多的信息。这就引入了标记 (Tagged) 指针概念。标记 (Tagged) 指针是指那些指针中包含特殊属性或信息的指针。其中指针对齐概念可以来标识一个指针是否是标记 (Tagged) 指针及相关类型，64 位的地址指针又可以提供保存额外信息的足够空间。如今，iOS 7 的 64 位环境和 Mac OS 10.7 (Lion) 中开始引入了标记 (Tagged) 指针。

## 4. 标记 (Tagged) 指针对 NSNumber 的优化

标记 (Tagged) 指针一个比较典型的应用就是 NSNumber。在 64 位环境下，对于一般的数字，NSNumber 不用再分配内存了。现在，看看 NSNumber 是如何运用标记 (Tagged) 指针的：

```
NSNumber *number3 = @3;  
NSNumber *number4 = @4;  
NSNumber *number9 = @9;  
NSLog(@"number3 pointer is %p", number3);  
NSLog(@"number4 pointer is %p", number4);  
NSLog(@"number9 pointer is %p", number9);
```

在 64 位模拟器中运行后，得到了如下结果：

```
number3 pointer is 0xb000000000000032
number4 pointer is 0xb000000000000042
number9 pointer is 0xb000000000000092
```

可以看出 number3、number4 和 number9 的值前 4 位都是 0xb，后 4 位都是 0x2（指针的 Tag），中间就是实际的取值。因此，这些 NSNumber 已经不需要再分配内存（指堆中内存）了，直接可以把实际的值保存到指针中，而无须再去访问堆中的数据。这无疑提高了内存访问速度和整体运算速度。

也就是说，标记（Tagged）指针本身就可以表示一个 NSNumber 了，在 64 位环境下运行这段代码：

```
NSLog(@"0xb000000000000052's class is %@", [(NSNumber*)0xb000000000000052 class]);
```

会输出如下结果：

```
0xb000000000000052's class is __NSCFNumber
```

那么，如果一个数超过了标记（Tagged）指针所能表示的范围，系统会怎么处理？看看这段代码：

```
NSNumber *numberBig = @(0x1234567890ABCDEF);
NSLog(@"numberBig pointer is %p", numberBig);
```

在 64 位模拟器中运行后，得到了如下结果：

```
numberBig pointer is 0x1094026a0
```

可以看出 numberBig 指针最后 4 位都是 0，应该是分配在堆中的对象。因此，如果 NSNumber 超出了标记（Tagged）指针所能表示的范围，系统会自动采用分配成对象，可以根据指针的最后 4 位是否为 0 来区分。

## 5. 标记（Tagged）指针对 isa 指针优化

查看 NSObject 类的头文件，你会发现这段定义：

```
@interface NSObject <NSObject> {
    Class isa;
}
```

所有类都继承自 NSObject，因此每个对象都有一个 isa 指针指向它所属的类。在 32 位和 64 位的环境下，isa 指针会产生不同的变化。

在 32 位环境下，对象的引用计数都保存在一个外部的表中，而对引用计数的增减操作都要先锁定这个表，操作完成后才解锁。这个效率是非常慢的。

而在 64 位环境下，isa 也是 64 位，实际作为指针部分只用到其中的 33 位，剩余的部分会运用到标记（Tagged）指针的概念。其中 19 位将保存对象的引用计数，这样对引用计数的操作只需要原子的修改这个指针即可。如果引用计数超出 19 位，才会将引用计数保存到外部表，而这种情况往往是很少的，因此效率将会大大提高。



### 要点

- (1) 利用标记 (Tagged) 指针，可以在指针地址中保存或附加更多的信息。
- (2) 利用标记 (Tagged) 指针处理 NSNumber，直接可以把实际的值保存到指针中，而无须再去访问堆中的数据，可提高内存访问速度和整体运算速度。
- (3) 在 32 位和 64 位的环境下，isa 指针会产生不同的变化。在 64 位环境下，标记 (Tagged) 指针可加快 isa 指针的处理效率。

## 建议 11：谨记兼容 32 位和 64 位环境下代码编写事项

在 iOS 7 版本出现之前，应用程序主要都是基于 32 位的 iOS 运行环境设计的，很少会考虑到要兼容 64 位的 iOS 运行环境。现在 64 位的 iOS 运行环境已经出现了。这个时候，在编写应用程序的时候，就不得不考虑了如何确保自己写的应用程序，既能在 iOS 的 32 位环境下运行又能在 64 位的环境下运行。

下面就编写兼容 iOS 32 位和 64 位运行环境的应用程序容易犯的错误，进行逐一介绍，希望能对各位有所帮助。

### 1. 不要将长整型数据赋予整型

在许多导致编程错误产生因素之中，最为典型的因素莫过于在应用程序的整个代码中，不能使用一贯的数据类型，导致编译应用程序代码时候，产生大量的警告提醒信息。

故此，当调用函数时，要确保接收到的结果与该函数返回的变量的类型相匹配。与接收变量相比，如果返回类型是一个较大的整数，那么该值将会被截断。

下面的代码示例就表现出了此种错误，分配给一个变量时却截断一个返回值。在此代码示例中 PerformCalculation 函数将返回一个长整型。在 32 位运行时中，int 和 long 都是 32 位，即使代码不正确，但也能确保分配为 int 类型能有效工作。在 64 位运行时中，结果的高 32 位被分配时，将被丢掉。要保证不出现数据丢失，就应将结果赋给一个长整数，确保代码在 32 位和 64 位的运行时环境中都能有效运行。

```
long PerformCalculation(void);  
// 不正确  
int x = PerformCalculation();  
// 正确  
long y = PerformCalculation();
```

当作为一个参数传递值时，也会出现与上边一样的问题。例如，在下面的示例代码中 64 位运行时执行的输入参数时被截断。

```
int PerformAnotherCalculation(int input);
    long i = LONG_MAX;
    int x = PerformCalculation(i);
```

在下面的代码示例中，在 64 位运行时中返回值也被截断，因为该函数的返回类型超出返回值的范围。

```
int ReturnMax()
{
    return LONG_MAX;
}
```

在上面的这些示例中，都是假定 `int` 和 `long` 是完全相同的代码，即相同的数据类型，但是 ANSI C 标准不能确保假设的这些情况都成立。当应用程序运行在 64 位环境中时，上面的代码就会出现明显错误。在默认编译环境下，编译器会自动启用 32 位和 64 位校验机制，一旦一个值被截断，在大部分情况下，编译器将会自动抛出警告。如果编译器没有启用 32 位和 64 位校验机制，这时候应该在编译器选项中明确启用它，或者同时选择转化选项，这样就能利用编译器的校验机制，发现更多更详细的潜在错误。

在 Cocoa Touch 的应用程序中，查找以下的整数类型并确保正确地使用它们：

```
long
NSInteger
CFIndex
size_t (调用的 sizeof 内在操作的结果)
```

在这两个运行时环境中的 `fpos_t` 和 `off_t` 的类型都是 64 位的，所以从来没有将它们分配给一个 `int` 类型。

## 2. 善用 NSInteger 来处理 32 位和 64 位之间的转换

在编写应用程序时，可以在代码中多用 `NSInteger` 来处理数字类型的变量，因为 `NSInteger` 可以与 iOS 不同的运行环境兼容。

无论是在运行 32 位运行环境中，还是在运行在 64 位环境中，`NSInteger` 的类型的的应用贯穿于整个 Cocoa Touch。其在 32 位运行时中是 32 位整数，其在 64 位运行时中是 64 位整数。所以，当从一个框架的方法接收信息时，它采用一个 `NSInteger` 的类型，因此务必使用 `NSInteger` 的类型来保存结果。

永远不要假设 `NSInteger` 的类型是一个大小相同的 `int` 类型，这里有几个关键例子来看看：

- ❑ 转换成 `NSNumber` 对象或转换 `NSNumber` 对象为其他。
- ❑ 编码和解码数据里，使用的是 `NSCoder` 类。尤其是，编码 `NSInteger` 在 64 位的设备上，但将它解码在 32 位的设备上，如果值超过一个 32 位整数的范围，解码方法将会引发异常。

❑ 使用 NSInteger 作为框架中定义的常量。特别值得注意的是 NSNotFound 常数。它的价值是大于一个 int 类型的最大范围，所以在的应用程序中截断其价值往往会导致错误的出现。

在 64 位的代码中，CGFloat 的大小改变了，GFloat 类型变为一个 64 位单精度数。作为 NSInteger 的类型，不能想当然地把 CGFloat 认为是一个单精度或双精度数类型。因此，要使用一致的 CGFloat。下面的示例就是使用 Core Foundation 来创建 CFNumber 的。

```
// 不正确
CGFloat value = 200.0;
CFNumberCreate(kCFAllocatorDefault, kCFNumberFloatType, &value);

// 正确
CGFloat value = 200.0;
CFNumberCreate(kCFAllocatorDefault, kCFNumberCGFloatType, &value);
```

3. 创建数据结构要注意固定大小和对齐

当数据在 32 位和 64 位版本的应用程序之间共享时，就有必要确保创建兼容 32 位和 64 位的数据结构是完全相同的。当数据存储在一个文件或在一个传输网络的设备中时，其运行环境可能是相对立的。例如，用户把数据备份存储在 32 位的设备中，却在 64 位的设备中进行数据恢复，环境的差异性，在一定的程度上，会影响数据的正常使用，故此，对于这样的数据互操作存在的问题，必须要找到解决的方法。

(1) 使用明确的整数数据类型

不管底层的硬件结构，C99 标准提供了内置的数据类型都是特定大小的。当数据必须是一个固定大小时，或者当知道一个特定的变量有一个有限的可能值范围时，这个时候就应该考虑使用这些数据类型。通过选择适当的数据类型，会得到一个固定宽度的类型，可以存储在内存中，也避免分配一个变量时浪费内存，其范围远大于需要。

表 2-1 列出每个 C99 类型和范围的允许值。

表 2-1 C99 明确的整数类型

类型	范围
int8_t	-128 到 127
int16_t	-32 768 到 32 767
int32_t	-2 147 483 648 到 2 147 483 647
int64_t	-9 223 372 036 854 775 808 ~ 9 223 372 036 854 775 807
uint8_t	0 到 255 的
uint16_t	0 到 65 535
uint32_t	0 到 4 294 967 295
uint64_t	0 至 18 446 744 073 709 551 615



## (2) 对齐 64 位整数类型时要小心

在 64 位运行时，所有的 64 位整数类型的变化从 4 字节到 8 字节对齐。即使明确指定每个整数类型，这两种结构仍然可能是不相同的两个运行时。在代码清单 2-1 中，对齐改变，即使该字段声明明确的整数类型。

代码清单 2-1 对齐的 64 位整数结构

---

```
struct bar {
    int32_t foo0;
    int32_t foo1;
    int32_t foo2;
    int64_t bar;
};
```

---

当使用 32 位的编译器编译此代码时，字段栏（field bar）是从结构起始开始的 12 字节；当使用 64 位编译器编译该代码时，字段栏（field bar）是从结构起始的 16 字节开始的。在 foo2 中填充 4 字节，就可以确保 foo2 的栏（bar）具有 8 字节，从而实现边界对齐。

如果定义新的数据结构，首先组织的元素具有最大对齐值，最后的是最小的元素。这个结构组织消除正是大多数的填充字节需要的。如果正在使用现有的结构，其中包括未对齐的 64 位整数，可以使用 pragma 来强制其正确对齐。代码清单 2-2 给出了相同的数据结构，但这里的结构是被迫使用 32 位对齐规则的。

代码清单 2-2 使用的 pragma 控制对齐

---

```
#pragma pack(4)
struct bar {
    int32_t foo0;
    int32_t foo1;
    int32_t foo2;
    int64_t bar;
};
#pragma options align=reset
```

---



注意

只有在必要时才使用此选项，主要因为访问未对齐的数据，易造成性能上损失。故此要想确保自己的 32 位版本的应用程序中数据结构，能具有向后兼容性，就很有必要使用此选项。

## 4. 选择一种紧凑的数据表示形式

在编写应用程序代码时，选择一种恰当的数据结构形式，就可以比较好地表示数据。例如，使用下面的数据结构存储日历日期：

```
struct date
{
    NSInteger second;
    NSInteger minute;
    NSInteger hour;
    NSInteger day;
    NSInteger month;
    NSInteger year;
};
```

这种结构的长度为 24 字节；在 64 位运行时，它需要 48 字节，只为一个日期！一个更紧凑的表示方式是以秒数来存储某一特定时间。必要时，将此紧凑的表示形式转换为日历的日期和时间。

```
struct date
{
    uint32_t seconds;
};
```

对于对齐的数据结构，编译器有时会向其中添加填充物，例如：

```
struct bad
{
    char        a;           // offset 0
    int32_t     b;           // offset 4
    char        c;           // offset 8
    int64_t     d;           // offset 16
};
```

这种结构包括 14 字节的数据，但由于填充，它占用的 24 字节的空间。更好的设计方式是对字段进行从大到小的排序来对齐。

```
struct good
{
    int64_t     d;           // offset 0
    int32_t     b;           // offset 8
    char        a;           // offset 12;
    char        c;           // offset 13;
};
```



### 要点

- (1) 不要将长整型数据赋予整型。
- (2) 利用用 NSInteger 来处理 32 位和 64 位之间的转换。
- (3) 创建数据结构要注意固定大小和对齐。

## 建议 12：清楚常量字符串和一般字符串的区别

在 Objective-C 中，经常会用到常量字符串，常量字符串和一般的字符串还是有一定区别的。本节将介绍一些常量字符串的特性，用来加强对常量字符串的理解。请看下面一段代码。

```
NSString *string1 = @"Hello";
NSString *string2 = @"Hello";
if (string1==string2) {
    NSLog(@"They are same address");
}
```

对字符串常量 `string1` 和 `string2` 的地址值进行比较，就会发现二者竟然是相等的，产生这样的结果要归咎于编译器优化的结果。

由于常量会占用一块特殊的代码段，加载到内存时会映射到一块常量存储区，以加快访问速度。编译器在编译时发现 `string1` 和 `string2` 的内容是相同的常量字符串，会把它们都指向一个相同的区域，而不是再开辟出一块额外的空间。因此，它们是相同的地址值。

再看看这段代码：

```
NSString *string1 = @"Hello";
NSString *string2 = [NSString alloc];
NSString *string3 = [string2 initWithString:string1];
if (string2!=string3) {
    NSLog(@"string2 are not same to string3!");
}
if (string1==string3) {
    NSLog(@"string1 are same to string3!");
}
```

首先，申明上面这一段代码不是一段合法的代码，因为在第 2 行 `alloc` 之后没有立即 `init`。虽然这种做法是非常不推荐的，但这次为了更加清晰地说明问题，不得已而为之。

通过程序比较分析，就会发现 `string2` 和 `string3` 的地址值竟然不相等，而 `string1` 和 `string3` 竟然相等。通过这些可看出，如果使用一个常量字符串来初始化另一个字符串，另一个字符串会直接通过地址赋值为常量字符串，`alloc` 的内存也会立即释放。再看看下面这段代码：

```
NSString *string1 = [[NSString alloc] initWithString:@"Hello"];
[string1 release];
[string1 release];
[string1 release];
NSLog(@"%@", string1);
```

`string1` 经过多次 `release` 竟然还能继续访问，由此说明常量字符串不会 `release`。



### 要点

- (1) 由于编译器的优化，相同内容的常量字符串的地址值是完全相同的。
- (2) 如果使用常量字符串来初始化一个字符串，那么这个字符串也将是相同的常量。
- (3) 对常量字符串永远不要 `release`。

## 建议 13：在访问集合时要优先考虑使用快速枚举

在 Objective-C 语言中，集合是最常用的数据类型。而对于集合的访问，要优先考虑使用快速枚举。使用快速枚举，要尽可能使用枚举新的写法。

### 1. 尽可能使用枚举新的写法

使用 Objective-C 新的枚举写法，编写更简洁的代码，同时避免一些常见的陷阱。更重要的是，这些语法特性是完全向下兼容的，使用新特性编写出来的代码经过编译后形成的二进制程序可以运行在之前发布的任何 OS 中。

在使用枚举新的写法之前，先梳理一下最近几年来枚举类型几次功能的改进。

枚举在 OS X 10.5 之前的版本中，如何在 Objective-C 中定义一个枚举类型呢？定义方法如下：

```
typedef enum {  
    ObjectiveC,  
    Java,  
    Ruby,  
    Python,  
    Erlang }  
Language;
```

这种写法简单明了，用起来也不复杂，但是有一个问题，就是其枚举值的数据范围是模糊的，这个数值可能非常大，可能是负数，无法界定。

在 OS X 10.5 之后的版本中，就可以这样写：

```
enum {  
    ObjectiveC,  
    Java,  
    Ruby,  
    Python,  
    Erlang  
};  
typedef NSUInteger Language;
```

这种写法的好处是，首先这个枚举的数据类型是确定的，无符号整数；其次，由于采

用了 `NSUInteger`，就可以不用考虑 32 位和 64 位的问题。但这样写所带来的问题是，数据类型和枚举常量没有显式的关联。

在 XCode 4.4 中，就可以这样写枚举了：

```
typedef enum Language : NSUInteger{
    ObjectiveC,
    Java,
    Ruby,
    Python,
    Erlang
}Language;
```

在列出枚举内容的同时绑定了枚举数据类型 `NSUInteger`，这样带来的好处是增强了类型检查和更好的代码可读性。

当然，对于普通开发者来说，枚举类型可能不会涉及复杂的数据，使用之前的两种写法也不会有什么大问题，但还是建议使用枚举新的写法，以增强现在写的代码在未来中有更强的适用性。

上面介绍了为何尽可能使用枚举新的写法，下面将介绍枚举在集合中的应用。

Objective-C 和 Cocoa 或 Cocoa Touch 提供了多种方式来枚举集合的内容。虽然它可以使用传统的 C 循环来遍历内容，像这样：

```
int count = [array count];
for (int index = 0; index < count; index++) {
    id eachObject = [array objectAtIndex:index];
    ...
}
```

这时最好的做法是使用本节中描述的其他技术之一。

## 2. 理解快速枚举

快速枚举是一种语言中用于快速安全的枚举一个集合的表达式，下面将从快速枚举的定义及其应用来分别介绍。

`for...in` 表达式，这种快速枚举表达式是如下定义的：

```
for ( Type newVariable in expression ) { statements }
```

或者：

```
Type existingItem;
for ( existingItem in expression ) { statements }
```

这两种表达式中，`expression` 是一个遵守 `NSFastEnumeration` 协议的对象。每次循环被迭代的对象都会返回一个对象并赋给一个循环变量，同时 `statements` 中定义的代码被执行一次。当被迭代的对象中已经没有数据可以取出时，循环变量被设为 `nil`，如果循环在这之前被停止，那么循环变量会指向最后一次返回的值。

### 3. 使用快速枚举可以很容易地枚举集合

许多集合类遵照了 NSFastEnumeration 协议，如 Foundation 中的集合类 NSArray、NSDictionary 和 NSSet。显而易见，枚举操作可以遍历 NSArray 和 NSSet 的内容。对于其他类，相关文档会说明哪些内容会被枚举。例如，NSDictionary 和 NSManagedObjectModel 也支持快速枚举，但是 NSDictionary 枚举的是它的键值，NSManagedObjectModel 枚举的是它的实体。

作为一个例子，可以使用快速枚举，像这样在一个数组记录每个对象的描述：

```
for (id eachObject in array) {
    NSLog(@"Object: %@", eachObject);
}
```

变量 eachObject 被自动设置为每个通过循环到当前对象，所以，一个日志报表显示每个对象。如果使用快速枚举字典，可以快速遍历字典键，像这样：

```
for (NSString *eachKey in dictionary) {
    id object = dictionary[eachKey];
    NSLog(@"Object: %@ for key: %@", object, eachKey);
}
```

快速枚举的行为很像一个标准的 C 循环，这样就可以使用打破关键字来中断迭代，或继续前进到下一个元素。如果枚举有序的集合，枚举按顺序来进行。对于一个 NSArray，这意味着第一次将索引为 0，第二个对象在索引 1 处，等等。如果需要跟踪当前索引，只需迭代计数，因为它们发生：

```
int index = 0;
for (id eachObject in array) {
    NSLog(@"Object at index %i is: %@", index, eachObject);
    index++;
}
```

在快速枚举集合期间，不能变异集合，即使集合是可变的。如果从循环内，尝试添加或删除集合的对象，则会生成运行时异常。

### 4. 大多数集合也支持枚举器对象

也可以通过使用 NSEnumerator 对象枚举 Cocoa 和 Cocoa Touch 中的许多集合。例如，对于 objectEnumerator 或 reverseObjectEnumerator，可以查询 NSArray。快速枚举可以使用这些对象，像这样：

```
for (id eachObject in [array reverseObjectEnumerator]) {
    ...
}
```

在此示例中，循环将按照相反的顺序来让变数可以递回取得集合的对象，所以最后一个对象将是第一个，等等。通过反复调用枚举的 nextObject 的方法，它也可以遍历这些内

容，像这样：

```
id eachObject;
while ( (eachObject = [enumerator nextObject]) ) {
    NSLog(@"Current object is: %@", eachObject);
}
```

在这个例子中，一个 `while` 循环用于将 `eachObject` 变量设置为每次循环的下一个对象。当没有更多的对象保留时，`nextObject` 方法将返回 `nil`，评估为 `false` 的逻辑值，以使循环停止。



注意

当想表达相等运算符 (`==`) 时，程序员常见的错误是使用 `C` 赋值运算符 (`=`)，如果在一个条件分支或循环中设置一个变量，这样将会造成编译器发出警告，像这样：

```
if (someVariable = YES) {
    ...
}
```

如果真的这样做意味着重新分配一个变量（整体转让的逻辑值是左侧的最终值），可以将括号中的分配，像这样：

```
if ( (someVariable = YES) ) {
    ...
}
```

与快速枚举一样，在枚举进行时不可改变集合。而且，可根据名字来进行收集，使用快速枚举，相比使用手动枚举对象更快。



### 要点

- (1) 使用快速枚举，要尽可能使用枚举新的写法。
- (2) 和直接使用 `NSEnumerator` 相比，使用快速枚举可以更有效率，表达式更简洁。
- (3) 使用快速枚举，枚举更安全，因为枚举会监控枚举对象的变化，如果在枚举的过程中枚举对象发生变化会抛出一个异常。
- (4) 多个枚举可以同时进行，因为在循环中被循环对象是禁止修改的。另外，同其他的循环一样，可以使用 `break` 来停止循环或者使用 `continue` 来略过当次循环而进行到下一元素。



## 建议 14：有序对象适宜存于数组，而无序对象适宜存于集

虽然可以使用 C 语言数组来保存标值的集合，甚至是对象指针，但是在 Objective-C 代码中，大多数集合是 Cocoa 和 Cocoa Touch 集合类中的某一个类的实例，如 NSArray、NSSet 和 NSDictionary。

使用这些类可用来管理对象组，这意味着任何添加到集合的项必须是 Objective-C 类的一个实例。如果需要添加一个标量值，就必须首先创建一个合适的 NSNumber 或 NSValue 的实例来表示它。

某种程度上维护着每个集合对象的单独的副本，集合类使用强引用来跟踪它们的内容。这意味着添加到集合中的任何对象，其生命周期至少要跟集合的生命周期一样长，正如“通过所有权和责任来管理的对象图”。除了跟踪它们的内容之外，Cocoa 和 Cocoa Touch 的每个集合类还可以很容易地执行某些任务，例如枚举，访问特定项或找出特定的对象是否是集合的一部分。

虽然 NSArray、NSSet 和 NSDictionary 类都是不可改变的，这意味着他们的内容在创建时将要进行设置；但它们每一个都有可变的子类，利用他们的子类，可以随意添加或删除对象。

下面对于集合中的数组和集，适宜存储对象的差异性分别进行介绍。

### 1. 有序对象适合存于数组

NSArray 用来表示对象有序的集合。唯一的要求是，每一项必须是 Objective-C 对象，也就是说，没有要求每个对象必须是同一类的一个实例。因此，一组对象的顺序很重要时，就该使用数组。例如，许多应用程序使用数组向表格视图中的行或菜单中的项目提供内容；索引为 0 的对象对应于第一行，索引为 1 的对象对应于第二行，依此类推。访问数组中对象的时间，比访问集合中对象的时间长，如图 2-7 所示。

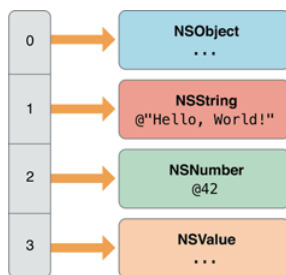


图 2-7 Objective-C 对象数组

数组以有序序列储存对象。

### 1) 创建数组

NSArray 类提供许多初始化程序和类工厂方法，用于创建数组和对数组进行初始化，但有几个方法尤其常见和实用。可以使用 arrayWithObjects:count: 和 arrayWithObjects: 方法（及其对应的初始化程序 initWithObjects:count: 和 initWithObjects:），从一系列对象创建数组。使用前一种方法时，第二个参数指定第一个参数（静态 C 数组）中的对象数；使用后一种方法时，其参数为以逗号分隔的对象序列（以 nil 终止）。

```
// Compose a static array of string objects
NSString *objs[3] = @{@"One", @"Two", @"Three"};
// Create an array object with the static array
NSArray *arrayOne = [NSArray arrayWithObjects:objs count:3];
// Create an array with a nil-terminated list of objects
NSArray *arrayTwo = [[NSArray alloc] initWithObjects:@"One", @"Two", @"Three",
nil];
```

创建可变数组时，可以使用 arrayWithCapacity:（或 initWithCapacity:）方法创建数组。容量参数将有关数组预期大小的提示提供给类，从而使数组在运行时更高效。数组甚至可以超过所指定的容量。

还可以使用容器字面常量 @[ ...] 创建数组，其中方括号之间的项目是以逗号分隔的对象。例如，要创建包含一个字符串、一个数字和一个日期的数组，可以编写如下代码：

```
NSArray *myArray = @[ @"Hello World", @67, [NSDate date] ];
```

### 2) 访问数组中的对象

通常，调用 objectAtIndex: 方法访问数组中的对象，方法是指定该对象在该数组中的索引位置（从 0 开始）。

```
NSString *theString = [arrayTwo objectAtIndex:1]; // returns second object in
array
```

NSArray 提供其他方式来访问数组中的对象或其索引。例如，有 lastObject、firstObjectCommonWithArray: 和 indexOfObjectPassingTest:。

可以使用下标记号（而非使用 NSArray 的方法）访问数组中的对象。例如，要访问 myArray（上面已创建）中的第二个对象，可以编写如下代码：

```
id theObject = myArray[1];
```

与数组有关的另一个常见任务是，对数组中的每个对象执行某种操作——这时称为枚举的过程。通常通过枚举数组来决定一个对象或多个对象是否与某个值或条件匹配；如果有一个对象匹配，则使用该对象完成一项操作。可以采用以下三种方式之一枚举数组：快速枚举、使用块枚举或使用 NSEnumerator 对象。顾名思义，快速枚举通常比使用其他技巧访问数组中的对象要快。快速枚举是一项需要特定语法的语言功能：

```
for (type variable in array){ /* inspect variable, do something with it */ }
```

例如：

```
NSArray *myArray = // get array
for (NSString *cityName in myArray) {
    if ([cityName isEqualToString:@"Cupertino"]) {
        NSLog(@"We're near the mothership!");
        break;
    }
}
```

数个 NSArray 方法使用块来枚举数组，其中最简单的是 `enumerateObjectsUsingBlock:`。此块具有三个参数：当前对象、其索引和引用的 Boolean 值（设置为 YES 时终止枚举）。此块中的代码执行的工作，与快速枚举语句中大括号内的代码完全相同。

```
NSArray *myArray = // get array
[myArray enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop) {
    if ([obj isEqual:@"Cupertino"]) {
        NSLog(@"We're near the mothership!");
        *stop = YES;
    }
}];
```

### 3) 管理可变数组

NSArray 具有其他方法用于给数组排序、搜索数组和在数组中的每个对象上调用方法。

通过调用 `addObject:` 方法，可将对象添加到可变数组；对象放在数组末尾。也可以使用 `insertObject:atIndex:`，将对象放在可变数组中的特定位置。通过调用 `removeObject:` 方法或 `removeObjectAtIndex:` 方法，可以从可变数组中移除对象。

还可以使用下标记号，将对象插入可变数组中的特定位置。

```
NSMutableArray *myMutableArray = [NSMutableArray arrayWithCapacity:1];
NSDate *today = [NSDate date];
myMutableArray[0] = today;
```

### 4) 同一数组可以保存不同的对象

比如一个数组 NSArray，这种数组里面可以保存各种不同的对象，比如这个数组里：

```
myArray <
0: (float) 234.33f
1: @"我是个好人"
2: (NSImage *) (俺的美图)
3: @"我真的是好人"
```

这是一个由 4 个东西组成的数组，这个数组包括一个浮点数，两个字符串和一个图片。



**注意** NSArray 中不能存储基本类型，如 float, int, double 等，否则都会被设置为 0。另外，上面这个调用必须用 nil 来结尾，这也意味着 NSArray 中不能存储 nil。

## 2. 无序对象适合储存于集

集（Sets）是类似于数组的一组对象，但按照无序组的方式来维持着不同的对象，如图 2-8 所示。只是其中包含的项目是无序的（而数组是有序的）。通过枚举集合中的对象，或者将过滤器或测试应用到集合，来随机访问集合中的对象（使用 `anyObject` 方法），而不是按索引位置或通过键访问它们。

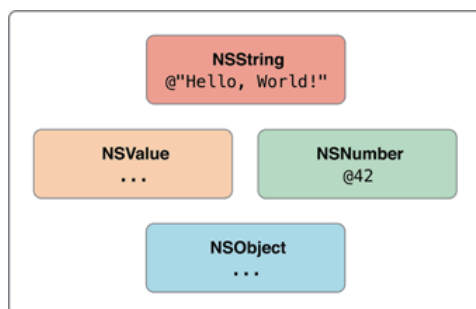


图 2-8 一组对象

由于集（Sets）不维持秩序，当它涉及成员资格的测试时，它们对于数组就提供了一个性能改进处理。又加上基本的 `NSSet` 类是不可改变的，所以在用分配和初始化类工厂方法来创建其内容时，必须是规定好的，像这样：

```

NSSet *simpleSet =
    [NSSet setWithObjects:@"Hello, World!", @42, aValue, anObject, nil];

```

用 `NSArray`，通过 `initWithObjects:` 和 `setWithObjects:` 这两种方法可能会使其中一个零终止和参数的数目可变。可变的 `NSSet` 子类是 `NSMutableSet`。即使不止一次在集（Sets）中尝试添加一个对象，集也只能存储一个单独对象的一个引用，如下所示：

```

NSNumber *number = @42;
NSSet *numberSet =
    [NSSet setWithObjects:number, number, number, number, nil];
// numberSet only contains one object

```

尽管集合对象在 Objective-C 编程中不如字典和数组那么常用，但它们在某些技术中是重要的集类型。在 Core Data（一种数据管理技术）中，当声明对多关系的属性时，属性类型应该是 `NSSet` 或 `NSOrderedSet`。集合对于 UIKit 框架中的原生触摸事件处理也很重要，例如：

```

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *theTouch = [touches anyObject];
    // handle the touch...
}

```

有序集合是集合基本定义的一个例外。在有序集合中，集合中的项目顺序很重要。有

序集合中测试成员资格比数组中要快。

### 3. 使用集合可保持对象图的持久化

使用 NSArray 和 NSDictionary 类可以容易地直接将其内容写入磁盘，像这样：

```
NSURL *fileURL = ...
NSArray *array = @[@"first", @"second", @"third"];

BOOL success = [array writeToURL:fileURL atomically:YES];
if (!success) {
    // an error occurred...
}
```

如果每个容器性对象是属性列表类型 NSArray、NSDictionary、NSString、NSData、NSDate 和 NSNumber 之一，则可能要从磁盘重新创建整个层次结构，像这样：

```
NSURL *fileURL = ...
NSArray *array = [NSArray arrayWithContentsOfURL:fileURL];
if (!array) {
    // an error occurred...
}
```

如果需要在容器中继续存在其他类型的对象，且其是上面所示的标准属性列表类，那么就可以使用归档器对象来创建容器，如 NSKeyedArchiver，来创建档案收集的对象。

创建一个归档文件，唯一的要求是每个对象必须支持 NSCoder 的协议。这意味着每个对象必须知道如何对自己进行编码到存档（通过实施 encodeWithCoder: 方法）和解码本身从一个现有的存档中读取（initWithCoder: 方法）。

NSArray、NSSet、NSDictionary 类和其可变的子类，都支持 NSCoder，这意味着可以坚持使用归档器对象的复杂层次结构。如果使用界面生成器窗口和视图的布局，例如，生成的 nib 文件只是直观地创建了对象层次结构的存档。在运行时，nib 文件是未归档到使用相关的类的对象的层次结构。



#### 要点

(1) 数组 (NSArray) 可维持持续性，故适宜存储有序的对象，但每一项必须是 Objective-C 对象。集 (Sets) 不维持秩序，故适宜存储无序对象。

(2) 同一数组 (NSArray) 可以保存不同的对象，但不能存储 float、int、double 等基本类型和 nil，否则存储基本类型都会被设置为 0，不能存储 nil 是因为数组必须用 nil 来结尾。

(3) 快速枚举是访问数组 (NSArray) 中的对象的一种比较快的方法。

(4) 使用 NSArray 和 NSDictionary 类可以直接将其内容写入磁盘进行持久化。

## 建议 15：存在公共键时，字典是在对象之间传递信息的绝佳方式

NSDictionary 不是仅仅维持对象有序或无序的集合，它根据给定键来存储对象，然后可以用于检索。使用字符串对象作为字典的键，这是最佳的做法，如图 2-9 所示。

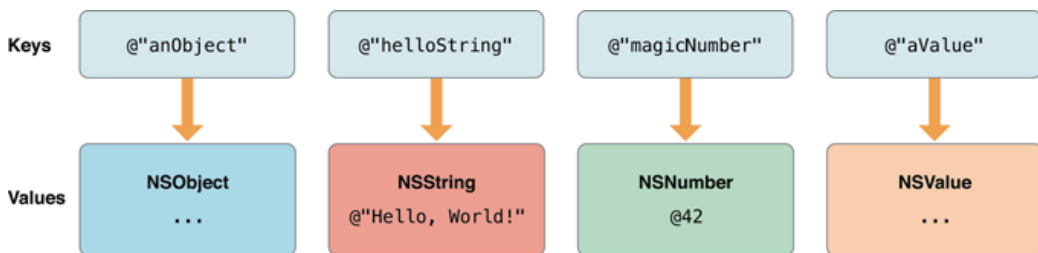


图 2-9 对象字典



注意

也可以使用其他对象作为键，但要注意，字典使用的每个键都是可以复制的，因此其必须支持 `NSCopying`。如果希望能够使用键-值编码，正如《键-值编码编程指南》中所介绍的，必须使用字符串键的字典对象。

使用字典将对象储存为键-值对，即标识符（键）和对象（值）对。字典是无序集，因为键-值对可采用任何顺序。尽管键几乎可以是任何内容，但通常是描述值的字符串，如 `NSFileModificationDate` 或 `UIApplicationStatusBarFrameUserInfoKey`（为字符串常量）。存在公共键时，字典是在对象之间传递各种信息的绝佳方式。

### 1. 创建字典

NSDictionary 类通过初始化程序和类工厂方法，提供了多种创建字典的方法，但只有两个类方法特别常用：`dictionaryWithObjects:forKeys:` 和 `dictionaryWithObjectsAndKeys:`（或它们对应的初始化程序）。使用前一种方法时，传入对象数组和键数组；键在位置上与其值匹配。使用第二种方法时，指定第一个对象值及其键，第二个对象值及其键，依此类推；使用 `nil` 标记此对象序列的结尾。

```
// First create an array of keys and a complementary array of values
NSArray *keyArray = [NSArray arrayWithObjects:@"IssueDate", @"IssueName",
    @"IssueIcon", nil];
NSArray *valueArray = [NSArray arrayWithObjects:[NSDate date], @"Numerology
    Today",self.currentIssueIcon, nil];
// Create a dictionary, passing in the key array and value array
NSDictionary *dictionaryOne = [NSDictionary dictionaryWithObjects:valueArray
    forKeys:keyArray];
```

```
// Create a dictionary by alternating value and key and terminating with nil
NSMutableDictionary *dictionaryTwo = [[NSMutableDictionary alloc]
initWithObjectsAndKeys:[NSDate date],
    @"IssueDate", @"Numerology Today", @"IssueName", self.currentIssueIcon,
    @"IssueIcon", nil];
```

如同数组，创建 NSDictionary 对象时，可使用容器字面常量 @{key:value, ...}，其中“...”表示任意数量的键—值对。例如，以下代码创建含 3 个键—值对的不可变字典对象：

```
NSMutableDictionary *myDictionary = @{
    @"name" :NSUserName(),
    @"date" :[NSDate date],
    @"processInfo" :[NSProcessInfo processInfo]
};
```

## 2. 访问字典中的对象

通过调用 objectForKey: 方法并将键指定为参数，访问字典中的对象值。

```
NSDate *date = [dictionaryTwo objectForKey:@"IssueDate"];
```

还可以使用下标访问字典中的对象，键出现在方括号内（方括号紧接在字典变量后面）。

```
NSString *theName = myDictionary[@"name"];
```

## 3. 管理可变字典

通过调用 setObject:forKey: 和 removeObjectForKey: 方法，在可变字典中插入和删除项目。setObject:forKey: 方法替换给定键的任何现有值。这些方法都很快捷。

还可以使用下标，将键—值对添加到可变字典中。键下标位于赋值左侧，而值位于右侧。

```
NSMutableDictionary *mutableDict = [[NSMutableDictionary alloc] init];
mutableDict[@"name"] = @"John Doe";
```



### 要点

- (1) 字典不仅可以作为无序对象的集合，还可以作为有序对象的集合。
- (2) 字典可作为有序对象的集合，主要依赖于键值可采用有序。
- (3) 存在公共键时，字典是在对象之间传递各种信息的绝佳方式。



## 建议 16：明智而审慎地使用 BOOL 类型

整型转换为 BOOL 型时，要小心，不要直接和 YES 作比较。

BOOL 在 Objective-C 里被定义为 unsigned char，这意味着它不仅仅只有 YES (1) 和 NO (0) 两个值。不要直接把整型强制转换为 BOOL 型。常见的错误发生在把数组大小、指针的值或者逻辑位运算的结果赋值到 BOOL 型中，而这样就导致 BOOL 值仅取决于之前整型值的最后一个字节，有可能出现整型值不为 0 但被转为 NO 的情况。因此把整型转为 BOOL 型的时候请使用三元 (Ternery) 操作符，保证返回 YES 或 NO 值。

在 BOOL、\_BOOL 及 bool (见 C++ Std 4.7.4、4.12 及 C99 Std 6.3.1.2) 之间可以安全地交换值或转型。但 BOOL 和 Boolean 之间不可以，所以对待 Boolean 就像上面讲的整型一样就可以了。在 Objective-C 函数签名里仅使用 BOOL。

对 BOOL 值使用逻辑运算 (&&, ||, !) 都是有效的，返回值也可以安全地转为 BOOL 型而不需要三元 (Ternery) 操作符。

```
// AVOID
- (BOOL)isBold {
    return [self fontTraits] & NSFontBoldTrait;
}

- (BOOL)isValid {
    return [self stringValue];
}

// GOOD
- (BOOL)isBold {
    return ([self fontTraits] & NSFontBoldTrait) ? YES : NO;
}

- (BOOL)isValid {
    return [self stringValue] != nil;
}

- (BOOL)isEnabled {
    return [self isValid] && [self isBold];
}
```

还有，不要把 BOOL 型变量直接与 YES 比较。这样不仅对于精通 C 的人很有难度，而且此条款的第一点也说明了这样做未必能得到你想要的结果。

```
// AVOID
BOOL great = [foo isGreat];
if (great == YES)
    // ...be great!

// GOOD
BOOL great = [foo isGreat];
```

```
if (great)
    // ...be great!
```



### 要点

- (1) 整型转为 **BOOL** 型，使用三元 (Ternery) 操作符，以保证返回 YES 或 NO 值。
- (2) 整型转换为 **BOOL** 型的时候要避免直接和 YES 做比较。
- (3) **BOOL** 值进行逻辑运算 (**&&**, **||**, **!**) 不但有效，而且还可以确保返回值安全地转为 **BOOL** 型，无须三元 (Ternery) 操作符。

